

AI: Intelligent systems with LangChain and LangGraph

Volume 1 Release 1



Alessandro Ciambrone

Intelligent systems with LangChain and LangGraph

Volume 1

Release 1

2026 - Alessandro Ciambrone

CONTENTS

1	Generative AI and large language models	18
1.1	AI, ML, deep learning, generative models, and language models	19
1.1.1	Artificial intelligence: the broad goal	19
1.1.2	Machine learning: learning patterns from data	19
1.1.3	Deep learning: neural networks at scale.....	19
1.1.4	Generative models and “generative AI”	20
1.1.5	Language models and large language models	20
1.1.6	How the layers fit together	21
1.2	Transformer architecture and attention mechanisms	23
1.2.1	From sequential models to transformers.....	23
1.2.2	Tokens, embeddings, and position	23
1.2.3	Self-attention in simple terms	24
1.2.4	Multi-head attention and layers.....	24
1.2.5	Decoder-only transformers and generation	25
1.2.6	Cost, context windows, and long sequences	25
1.2.7	Why prompt structure matters	26
1.2.8	Connecting back to agents	26
1.2.9	What makes a system an agent	27
1.3	Modalities: text, image, audio, video, multimodal	28
1.3.1	Text as the core reasoning channel.....	28
1.3.2	Images as inputs and outputs	28
1.3.3	Audio as a natural interface layer	29
1.3.4	Video as a composite modality	29
1.3.5	Multimodal models and evolving modality strategy	29
1.4	Major LLM families and providers; open vs closed vs hybrid models	30
1.4.1	Why model choice matters for your application	30
1.4.2	Closed, hosted model families	30
1.4.3	Open-weight and local model families	31
1.4.4	Open vs closed vs hybrid: practical trade-offs	31
1.5	Capabilities, benchmarks, and typical failure modes (hallucinations, reasoning gaps, bias, context limits)	33
1.5.1	What LLMs are actually good at.....	33

1.5.2	How people measure capabilities: a quick tour of benchmarks	34
1.5.3	Common ways models fail	34
1.5.4	Why this matters for everything that follows.....	36
1.6	Summary.....	37
2	From models to applications and agents.....	38
2.1	What an LLM application is (beyond a single API call)	38
2.1.1	From raw model call to application	39
2.1.2	The main building blocks of an LLM application.....	39
2.1.3	From application to “agentic system”	41
2.1.4	Why this distinction matters before you write any code	41
2.2	Limitations of raw LLM APIs: tools, memory, reliability, evaluation.....	42
2.2.1	No tools, no actions	42
2.2.2	No memory beyond the current call	42
2.2.3	Narrow context windows and static knowledge.....	43
2.2.4	Reliability problems: hallucinations, brittle reasoning, unpredictable formats ..	43
2.2.5	Safety, bias, and misuse	44
2.2.6	No built-in evaluation or observability	44
2.2.7	Why this matters before you write any framework code.....	45
2.3	Agents vs traditional applications: environment, tools, memory, goals	46
2.3.1	Traditional applications: the familiar mental model	46
2.3.2	Agentic applications: an LLM in the control loop	46
2.3.3	Environment: what the agent can actually see	46
2.3.4	Tools: how the agent acts on the world.....	47
2.3.5	Memory: keeping state as input to reasoning.....	47
2.3.6	Goals: making intent explicit	48
2.3.7	Putting it together: wrapping, not replacing, your existing systems.....	49
2.4	Levels of agent sophistication (Level 0–3) and use-case mapping	50
2.4.1	The agent ladder in overview	50
2.4.2	Level 0 – Core reasoning engine	50
2.4.3	Level 1 – Connected problem-solver.....	51
2.4.4	Level 2 – Strategic problem-solver	51
2.4.5	Level 3 – Collaborative multi-agent systems.....	52
2.4.6	Choosing the right level.....	53
2.4.7	When not to use an agent	53

2.5	Socio-economic and governance context (costs, risk, regulation)	54
2.5.1	From model calls to system-level costs	54
2.5.2	Risk landscape for LLM-based systems	55
2.5.3	Guardrails and governance-by-design.....	56
2.5.4	Regulation and evolving AI governance.....	57
2.5.5	Costs, risk, and governance as joint design forces.....	58
2.6	Summary.....	59
3	The LangChain / LangGraph / LangSmith ecosystem.....	60
3.1	LangChain: goals, architecture, and abstraction layers	60
3.1.1	Why LangChain exists.....	61
3.1.2	Where LangChain fits in the stack	62
3.1.3	Architectural principles	62
3.1.4	The abstraction layers: from components to chains	63
3.1.5	How this book will use LangChain.....	64
3.1.6	Lang* ecosystem boundary.....	65
3.2	LangGraph: graphs, stateful agents, and long-running workflows	66
3.2.1	From chains to graphs	66
3.2.2	StateGraph and explicit state.....	67
3.2.3	Checkpoints, threads, and long-running workflows	68
3.2.4	Interrupts and human-in-the-loop patterns	69
3.2.5	Advanced patterns: multi-agent systems and iterative loops	70
3.2.6	Development, observability, and deployment	70
3.2.7	When to reach for LangGraph	71
3.3	LangSmith: tracing, datasets, evaluation, monitoring	72
3.3.1	Core concepts: projects, runs, and traces.....	72
3.3.2	Tracing and debugging.....	73
3.3.3	Datasets: turning interactions into tests	73
3.3.4	Evaluation: from examples to metrics	75
3.3.5	Monitoring and dashboards	76
3.3.6	Human feedback, guardrails, and governance.....	77
3.3.7	LangSmith in the development lifecycle	78
3.4	The LangChain / LangGraph / LangSmith feedback loop	79
3.5	Conceptual comparison with other canvases (CrewAI, Google ADK)	80
3.5.1	CrewAI: teams and roles as the primary abstraction.....	80

3.5.2	Google ADK: agentic applications as managed cloud services.....	80
3.5.3	Other ecosystems and how they fit	81
3.5.4	Choosing and combining canvases	82
3.6	Summary.....	83
4	Development environment and model integration	84
4.1	Python project setup: virtual envs, dependency management, and secrets management	85
4.1.1	Install Python and pip on Windows	85
4.1.2	Create a project folder and virtual environment	86
4.1.3	Install dependencies	87
4.1.4	Get your API keys and LangSmith credentials.....	88
4.1.5	Store configuration in a .env file	88
4.1.6	Create the basic project structure	89
4.1.7	Implement config.py – loading environment configuration.....	89
4.1.8	Implement main.py – testing OpenAI and Google Gemini	90
4.1.9	Run the smoke test	91
4.1.10	Other environment options (Poetry, Conda, Docker) – short overview	91
4.2	Running local models (Transformers, llama.cpp, GPT4All, Ollama-style setups) .	93
4.2.1	Shared integration pattern in LangChain	93
4.2.2	Local models with Hugging Face Transformers.....	94
4.2.3	llama.cpp models via LlamaCpp	94
4.2.4	GPT4All as a local model runtime	95
4.2.5	Ollama-style local model servers	95
4.2.6	Choosing and abstracting your local backend.....	96
4.3	Summary.....	96
5	Prompt design and dynamic prompting	99
5.1	Chat models and generation parameters (OpenAI and Google AI).....	100
5.1.1	LLM vs chat wrappers in LangChain.....	100
5.1.2	OpenAI.....	100
5.1.3	Google AI	104
5.1.4	Core chat generation parameters	105
5.2	Prompt templates and chat prompt patterns (system, user, assistant).....	114
5.2.1	PromptTemplate: parameterised single-string prompts.....	114
5.2.2	ChatPromptTemplate: structured multi-message prompts	117

5.2.3	Composing templates inside a chat template	118
5.2.4	Chat history and <code>MessagesPlaceholder</code>	119
5.2.5	<code>RunnableWithMessageHistory</code> : making chat history a runtime concern ..	121
5.2.6	<code>RunnableWithMessageHistory</code> : persistence options	122
5.2.7	Structuring prompts: persona, task, context, format	123
5.2.8	Templates as first-class LCEL components	125
5.2.9	Prompt lifecycle: versioning, testing, and output validation	127
5.2.10	Prompt Hub: shared prompts as a managed dependency	127
5.3	Token Accounting, callbacks, special cases and policies enforcement	129
5.3.1	Callback-based token accounting.....	129
5.3.2	<code>LLMMathChain</code>	130
5.3.3	<code>SQLDatabaseChain</code> : answering questions with SQL databases.....	131
5.3.4	Policy enforcement chains: moderation, constitutional revision, and self-checking	135
5.3.5	Google GenAI safety settings.....	140
5.4	Zero-shot and few-shot prompting; dynamic few-shot selection	141
5.4.1	Zero-shot prompting: starting with instructions only	141
5.4.2	One-shot and few-shot prompting: teaching by example.....	142
5.4.3	Dynamic few-shot selection: choosing examples per input	144
5.4.4	Overview of example selectors in LangChain	147
5.4.5	Few-shot prompting with chat prompts	148
5.4.6	When to use zero-shot, few-shot, and dynamic few-shot	150
5.5	Chain-of-thought and self-consistency prompting.....	151
5.5.1	Chain-of-thought prompting: explicit intermediate reasoning	151
5.5.2	Self-consistency prompting: sampling multiple reasoning paths	152
5.5.3	How CoT and self-consistency fit into LangChain workflows	153
5.5.4	CoT and self-consistency in a trading assistant	154
5.6	Tree-of-thought prompting and structured reasoning paths.....	158
5.6.1	From linear chains to branching trees	158
5.6.2	Anatomy of a thought tree.....	158
5.6.3	When tree-of-thoughts helps	158
5.6.4	Costs, limits, and search control	159
5.6.5	Role in agentic systems and relation to other techniques	159
5.7	Summary.....	166

6	Context engineering	167
6.1	What context engineering is and why it matters	168
6.2	Context selection and packaging for a single call.....	171
6.2.1	Normalising context inputs	171
6.2.2	The per-call context pipeline: candidates → selection → prompt	172
6.2.3	Retrieval strategies for documents.....	173
6.2.4	Post-retrieval refinement: keep less, keep better	175
6.2.5	Ordering and packaging evidence	178
6.2.6	Using metadata and structured state	183
6.2.7	Worked pipelines	188
6.2.8	Governance and evolution of context pipelines	193
6.3	Summarisation	194
6.3.1	Single-pass summarisation and prompt templates	194
6.3.2	Structured summaries: few-shot and schema-based	195
6.3.3	Reasoning-first summarisation: extract then write (extract-first).....	195
6.3.4	Hierarchical summarisation and map–reduce pipelines	198
6.3.5	Reasoning-first summarisation	202
6.3.6	Conversation summarisation and mixed memory	202
6.3.7	Classic conversation memory classes (buffer, window, summary, entities, graph)	
	206	
6.4	Salience Scoring and Windowing Strategies	209
6.4.1	Salience-focused summarisation: chain-of-density and similar patterns	209
6.4.2	Chain-of-density.....	209
6.4.3	Windowing mechanisms in practice	213
6.5	Context formats for different tasks (Q&A, planning, analysis).....	217
6.5.1	Question answering and retrieval-style assistants	217
6.5.2	Planning and multi-step workflows.....	218
6.5.3	Analysis and decision support.....	220
6.6	Measuring and improving context effectiveness.....	222
6.6.1	What “effective context” means.....	222
6.6.2	Model-level benchmarks	224
6.6.3	Common metrics (what to measure).....	225
6.6.4	Application-level datasets for context.....	226
6.6.5	Evaluators for context-sensitive behaviour	227

6.6.6	Statistical comparison and A/B testing.....	239
6.6.7	From offline evaluation to production monitoring.....	240
6.6.8	Diagnosing context failures	241
6.6.9	Techniques to improve context	242
6.7	Summary.....	244
7	LCEL and composable workflows in LangChain	245
7.1	Runnables and the LangChain Expression Language	246
7.1.1	Runnables.....	246
7.1.2	LCEL: the pipe-based composition layer.....	247
7.1.3	A first LCEL example: summarisation pipeline	247
7.1.4	Runnables in a support assistant scenario.....	248
7.1.5	Execution modes: single calls, batches, and streams	250
7.1.6	Configuration, tags, and observability	251
7.1.7	Custom runnables	255
7.2	Sequential, branching, routing and parallel pipelines with LCEL	256
7.2.1	Sequential pipelines.....	256
7.2.2	Routing pipelines	256
7.2.3	Parallel pipelines with <code>RunnableParallel</code>	261
7.2.4	Map-style pipelines	262
7.3	Input/output handling: mappings, passthroughs, assignments	263
7.3.1	Dict-based inputs and mappings	263
7.3.2	Passthroughs and field preservation	265
7.3.3	<code>ItemGetter</code> – manual construction	266
7.3.4	Assigning and picking fields	267
7.4	Using structured outputs in LCEL.....	269
7.4.1	Parsing structured data with output parsers.....	269
7.4.2	Output parser catalog.....	271
7.4.3	Basic and list parsers	271
7.4.4	Primitive type parsers.....	271
7.4.5	Utility and correction parsers	271
7.4.6	Specialised and advanced parsers.....	271
7.4.7	Parsers Examples.....	272
7.4.8	Controlled generation with <code>with_structured_output</code>	273
7.4.9	Structured outputs as the glue in multi-step LCEL pipelines	275

7.4.10	Schema validation outside the model call	276
7.4.11	Practical selection guidance	276
7.5	Fallbacks, retries, and error handling	278
7.5.1	Failure detection: make errors explicit at boundaries.....	278
7.5.2	Retries with <code>with_retry</code> (transient problems).....	278
7.5.3	Fallbacks with <code>with_fallbacks</code>	279
7.5.4	Correcting parsers and output-fixing.....	279
7.6	Callbacks	284
7.6.1	The run tree model	284
7.6.2	Where callbacks live in the modern API	285
7.6.3	Callback handlers: the hook surface you implement.....	286
7.6.4	Example: “Tell me what this chain actually did”	287
7.7	Summary.....	292
8	LangGraph fundamentals and state management	293
8.1	Nodes, edges, state objects, reducers, and configuration	294
8.1.1	Nodes.....	294
8.1.2	Edges.....	295
8.1.3	State objects and schemas	297
8.1.4	Input/output schemas and internal channels	298
8.1.5	Message-centric state and <code>MessagesState</code>	301
8.1.6	Reducers.....	302
8.1.7	Custom Reducers	304
8.1.8	Configuration.....	306
8.1.9	Runtime context and <code>context_schema</code>	307
8.1.10	Putting it together.....	308
8.2	Checkpoints and persistence for long-running workflows.....	312
8.2.1	The core model: threads + checkpointers	312
8.2.2	Designing nodes for persistence: idempotency and side effects	314
8.2.3	Short-term memory vs long-term memory	316
8.2.4	Backend choices for persistence.....	319
8.3	Control-flow patterns in graphs: loops, branches, subgraphs	322
8.3.1	Dynamic fan-out with Send	322
8.3.2	Durable execution: why “in-memory persistence” is not enough.....	324

8.3.3	Resumable human approval using <code>interrupt()</code> and <code>Command(resume=...)</code>	326
8.3.4	Subgraphs for maintainable domain decomposition.....	327
8.4	Error-handling paths and recovery inside graphs.....	330
8.4.1	Transient errors: local retries, then a controlled fallback	330
8.4.2	LLM-recoverable errors: validate, repair, and loop with a stop condition.....	332
8.4.3	User-fixable errors: interrupts and human-in-the-loop recovery	334
8.4.4	Unexpected errors: escalate cleanly, then recover from checkpoints when appropriate.....	336
8.4.5	Keeping recovery logic visible with shared subgraphs	338
8.4.6	Preventing recovery loops with explicit stop conditions.....	339
8.4.7	Testing failure paths deliberately	339
8.5	Reasoning using Graphs.....	345
8.5.1	A single reasoning pass is one node.....	345
8.5.2	Self-consistency	346
8.5.3	Tree-of-thought	348
8.5.4	Least-to-most prompting	353
8.5.5	Verification.....	356
8.5.6	Reflection	359
8.6	Short-context strategies: Map–Reduce summarization, long-video and long-document workflows	361
8.6.1	Map–reduce summarization as a graph pattern.....	362
8.6.2	Long-document workflows in practice	365
8.6.3	Long-video workflows	366
8.6.4	State, memory, and trade-offs.....	368
8.7	Summary.....	370

About the author

Alessandro Ciambrone is a Chief Architect and Cloud Lead with over a decade of experience delivering large-scale digital transformations across banking, pensions, insurance, automotive, and the public sector. He has designed and led enterprise and solution architectures for complex organizations, focusing on platforms that must be scalable, resilient, secure, and cost-effective.

Across cloud, hybrid, and on-premises environments, Alessandro has built and modernized systems using serverless architectures, container-based APIs, microservices, and event-driven

architectures. His work consistently balances delivery speed with the realities of production engineering: availability, observability, governance, compliance, and operational simplicity.

Alessandro is recognized for deep hands-on expertise across the major cloud ecosystems—Amazon Web Services (AWS), Google Cloud Platform (GCP), and Microsoft Azure—and for translating architectural strategy into pragmatic implementation plans that teams can execute.

Over the past 18 months, he has focused heavily on applied AI engineering, experimenting with the Lang* ecosystem (LangChain, LangGraph, and LangSmith) to build agent-based and LLM-powered applications that can operate reliably in real workflows. That practical exploration—what works, what breaks, and how to design for traceability and control—directly informs the patterns and guidance in this book.

When he isn't designing systems, Alessandro teaches and writes, with a clear goal: help engineers turn complex technology into dependable, maintainable solutions they can ship with confidence. (alexciambrone@gmail.com)

Preface

Large language models are now good enough to be useful in real products, and still unreliable enough to be dangerous when you treat them like a normal software component. They can write fluent answers, follow instructions (mostly), and adapt to many tasks. They can also hallucinate, misunderstand constraints, break output formats, and confidently do the wrong thing at the worst possible moment. If you have built anything beyond a toy demo, you have already felt the tension: the model is powerful, but the system around it is where success or failure is decided.

This book is about that surrounding system. It treats LLMs as one component inside an application: a decision-and-generation engine that needs scaffolding. That scaffolding includes retrieval, tools, state, control flow, error handling, and evaluation. The goal is not to produce clever prompts. The goal is to build intelligent systems that are debuggable, auditable, and maintainable, systems you can run in production without relying on hope as an engineering strategy.

The practical focus of the book is the Lang ecosystem:

- LangChain for composing model calls, retrieval, prompts, tools, and parsing into repeatable pipelines.
- LangGraph for turning those pipelines into explicit, stateful workflows with branching, loops, concurrency, checkpoints, and human-in-the-loop steps.
- LangSmith for tracing, datasets, evaluation, and monitoring—so behaviour becomes something you can measure and improve, not something you argue about after an incident.

You can build LLM applications without these libraries. You can also build a distributed system without a framework. People do. The point of Langis to make the “boring but critical” parts easier: standard interfaces, composable components, visible execution, and the feedback loop that turns experiments into engineering.

To keep the material grounded, the examples in this book are not random one-off demos. They are reframed into a small set of realistic scenarios that show up repeatedly:

- A customer support assistant that has to answer from policy and knowledge bases, follow escalation rules, and stay within safe boundaries.
- An e-commerce assistant*that handles product search, recommendations, and order-related workflows.
- A trading and market analysis assistant that turns noisy market and news inputs into structured analysis and explanations.

These scenarios are deliberately chosen because they force the issues that matter: context limits, retrieval quality, tool reliability, state management, and evaluation. When you see the same domains used across chapters, it is not for storytelling flair. It is to show how the same primitives—retrievers, raphs, checkpoints, evaluators—compose into larger systems over time.

The book starts with foundations: what models are, where they fail, and why failures are predictable enough to engineer around. It then moves from “single-call prompts” into real application design: context construction, retrieval patterns, output parsing, and defensive techniques that keep systems stable under messy inputs and partial failures. From there, it introduces the shift from chains to graphs: once you need branching, loops, retries, parallel work, or human approval, straight-line workflows become fragile and hard to reason about. LangGraph makes that control flow explicit, and it makes state and persistence first-class, so long-running workflows remain manageable.

A major thread throughout the book is that “agentic” does not mean “uncontrolled.” An agent is best understood as a goal-directed control loop with an LLM in the decision seat. That loop is only safe when you define what the system can observe, what tools it can use, what it should remember, and what stops it. You are not replacing software engineering with vibes; you are choosing where to put the model and how tightly to constrain it.

Another thread is the reality of context limits. Long documents, long conversations, and long-running jobs do not fit neatly into one prompt. The book treats “short-context strategies” as engineering patterns, not tricks: trimming, summarization, map-reduce workflows, and graph-shaped pipelines for long documents and video-like inputs. These strategies are expensive in different ways—tokens, latency, complexity—so the book focuses on trade-offs and on keeping intermediate artefacts visible so failures can be traced.

Finally, the book treats evaluation and governance as part of the build, not an afterthought. Once you can trace runs, promote edge cases into datasets, and run evaluations when you change prompts, retrieval, tools, or models, quality becomes something you can iterate on systematically. That discipline matters because LLM systems drift: providers change models, your data changes, and user behaviour changes. If you do not measure, you will eventually ship a regression with a smile.

This is a practical book for working engineers. You do not need to be a machine learning researcher. You do need the willingness to treat LLM behaviour as something to constrain, observe, and test—like any other component that can fail. If you approach the material with that mindset, you will come away with a set of reusable patterns for building agent-based systems that can survive contact with reality.

What this book covers

This book is organised into two parts, each with four chapters, designed to take you from core concepts to practical system-building with the Lang* ecosystem (LangChain, LangGraph, and LangSmith).

Part 1 — Foundations: generative AI, LLMs, and the LangChain stack

Chapter 1: Generative AI and large language models

You build a practical mental model of what LLMs are, how transformer-based generation works at a high level, and why context windows, token costs, and failure modes matter. The chapter covers modalities beyond text and explains why hallucinations, reasoning gaps, bias, and context limits are normal engineering constraints, not rare anomalies.

Chapter 2: From models to applications and agents

This chapter reframes an LLM from “an API you call” into “a component inside a system.” It explains what an LLM application needs around the model—context construction, tool use, memory, reliability controls, and observability—and then introduces agentic systems as goal-directed control loops with explicit constraints and stop conditions.

Chapter 3: The LangChain / LangGraph / LangSmith ecosystem

You get a map of the stack used throughout the book. LangChain is presented as the composition layer for prompts, retrieval, tools, and post-processing; LangGraph is introduced as the point where chains stop being enough and explicit stateful control flow becomes necessary; LangSmith is covered as the tracing and evaluation layer that turns behaviour into something you can inspect and improve.

Chapter 4: Development environment and model integration

You set up a repeatable Python environment and learn how to integrate hosted models (e.g., OpenAI and Gemini) as well as local backends. The emphasis is on isolation, reproducibility, secrets management, and provider-agnostic wiring so the rest of your code stays stable while model backends change.

Part 2 — Prompts, LCEL workflows, and LangGraph fundamentals

Chapter 5: Prompt design and dynamic prompting

This chapter treats prompts as engineered inputs: clear roles, tasks, constraints, and output formats. It shows how dynamic prompting adapts instructions and examples to the current request, so prompts remain consistent while still being context-aware.

Chapter 6: Context engineering

You learn how to build context that is useful rather than merely longer: selecting, packaging, compressing, summarising, and windowing information so the model sees what it needs without drowning in noise. The chapter connects these techniques to governance, observability, and the real costs of tokens and retries.

Chapter 7: LCEL and composable workflows in LangChain

This chapter shows how to build maintainable LangChain pipelines using LCEL and Runnables, keeping dataflow explicit and components reusable. It focuses on practical

workflow composition, structured outputs, and reliability patterns that make pipelines easier to test and evolve.

Chapter 8: LangGraph fundamentals and state management

You move from straight-line workflows to graphs: explicit state schemas, reducers for deterministic merges, concurrency patterns, checkpoints for durability, and the separation between thread-scoped continuity and longer-term memory stores. The chapter also covers reasoning patterns expressed as graph shapes and short-context strategies for oversized inputs, including map-reduce and long-document/long-video workflows with clear trade-offs.

Code examples and repository

All code examples and supporting resources for this book are available in the companion GitHub repository:

[https://github.com/alexciambrone/Intelligent Systems with LangChain and LangGraph](https://github.com/alexciambrone/Intelligent%20Systems%20with%20LangChain%20and%20LangGraph)

The repository is structured to mirror the book's progression, so you can follow along chapter by chapter, run the examples locally, and adapt them into your own projects.

Part 1 - Foundations: generative AI, LLMs, and the LangChain stack

Chapter 1: Generative AI and large language models

Chapter 2: From models to applications and agents

**Chapter 3: The LangChain / LangGraph / LangSmith
ecosystem**

Chapter 4: Development environment and model integration

1 GENERATIVE AI AND LARGE LANGUAGE MODELS

Large language models sit at the centre of everything you will build in this book. They write the replies your support assistant sends to customers, they interpret queries in your e-commerce search flow, and they help turn market data and news into readable explanations for a trading assistant. From the outside, they look like a simple API that turns text into more text. Underneath, they are the result of several decades of work in artificial intelligence, machine learning, and deep learning, combined with newer architectures, training methods, and deployment options. Before you can design reliable systems around them, you need a clear picture of what they are, how they work at a high level, and where their limits sit. Those limits are why the rest of the book focuses on the scaffolding around the model: retrieval, tools, state, and evaluation.

This chapter builds that foundation. It starts by unpacking the overloaded word “AI” into more precise layers: artificial intelligence as the broad goal, machine learning as a way of learning from data, deep learning as large neural networks, generative models as systems that create new data, and finally language models and large language models as the tools we use for text. It then introduces the transformer architecture and attention mechanisms, not as a mathematical treatment, but as a practical mental model: why inputs are split into tokens, how context windows work, why prompt structure matters, and why long sequences are both powerful and expensive. By the end of the chapter, you should be able to make informed choices about which models to use and how much you can trust them when you design real systems.

Generative AI today is not limited to text. The chapter briefly surveys other modalities—images, audio, video, and multimodal models that accept several of these at once—and explains how they show up in realistic applications. A support assistant that reads screenshots or call transcripts, an e-commerce assistant that works with product photos as well as descriptions, and a trading assistant that reasons over charts and news are all examples of the same underlying pattern: different types of data feeding into the same family of models.

Finally, the chapter maps out the model landscape you will be choosing from. It describes major model families and the trade-offs between closed, hosted APIs and open-weight models you can run yourself, including what a hybrid strategy looks like in practice. It closes with a realistic view of capabilities and failure modes: what current models are genuinely good at, how benchmarks are used to assess them, and the ways they predictably go wrong, from hallucinations and reasoning gaps to bias and context limits. The goal is not to turn you into a researcher or to teach you advanced mathematics, but to give you a solid, “good enough” mental model so you can make sound engineering decisions when you start wiring LangChain, LangGraph, and LangSmith around these models in the rest of the book.

1.1 AI, ML, DEEP LEARNING, GENERATIVE MODELS, AND LANGUAGE MODELS

AI is a broad label for systems that seem “smart” when they solve tasks for us. Large language models are one specific kind of AI that work with text. To understand what they can and cannot do, it helps to see how they sit on top of several simpler ideas.

People often use “AI” as if it were one thing. In reality, what you will use in this book sits on top of a stack of ideas and technologies: artificial intelligence, machine learning, deep learning, generative models, and finally language models. Getting these layers straight helps you understand what large language models can and cannot do, and why frameworks like LangChain and LangGraph are needed around them.

1.1.1 Artificial intelligence: the broad goal

Artificial intelligence (AI) is the broadest term. It covers any technique that lets machines perform tasks we associate with human intelligence: understanding language, recognising patterns, making decisions, planning actions, or learning from experience. Classic AI includes rule-based expert systems, path-finding algorithms for games, optimisation and scheduling software, and today’s data-driven systems. In this book, when we say “AI system” in a loose sense, we usually mean a system that uses modern machine learning and, more specifically, large language models as one of its main components.

1.1.2 Machine learning: learning patterns from data

Machine learning (ML) is a subset of AI. Instead of hand-coding all the rules, you let an algorithm learn patterns from data. You supply examples—inputs and desired outputs, or just a large collection of raw data—and the learning algorithm adjusts internal parameters so that its predictions or decisions improve over time. In practical terms, parameters are the internal numeric “knobs” of the model that training tunes so that it behaves differently on new inputs.

Traditional ML powers things like:

- Classifying emails as spam or not spam.
- Predicting customer churn.
- Ranking search results.
- Forecasting demand or prices.

In a customer-support context, a classic ML model might predict whether a ticket will be escalated. In e-commerce, it might score how likely a user is to click on a product. In trading, it might forecast volatility or classify news as positive, neutral, or negative. These models are usually focused, supervised, and task specific.

1.1.3 Deep learning: neural networks at scale

Deep learning is a particular family of machine learning methods built on deep neural networks: many layers of simple computation units stacked together. These networks can learn complex, hierarchical patterns directly from raw data such as images, audio, and text. Deep learning became practical because of three trends arriving together:

- More powerful hardware (GPUs, TPUs).
- Larger datasets (web-scale text, image, audio, code).

- Better training techniques and architectures.

For text and language tasks, deep learning made it possible to move away from hand-crafted features and instead train models directly on large corpora. This is the foundation on which modern language models and generative models are built.

1.1.4 Generative models and “generative AI”

Many traditional ML models only analyse or score existing data: classify an email, predict a number, choose between options. Generative models do something different: they create new data that looks like the data they were trained on. Generative models come in several flavours, across different modalities:

- Text-to-text: generate new text from input text. This includes chatbots, summarisation systems, and code generators.
- Text-to-image: generate images from a text description.
- Text-to-audio or text-to-music: generate speech or music from text prompts.
- Text-to-video: generate short video clips from textual descriptions.

These models are useful both directly (for example, a support assistant that writes answers, a product-description generator for e-commerce, or a trading assistant that drafts market commentary) and indirectly, by producing synthetic data to train or fine-tune other systems when real data is scarce or sensitive.

1.1.5 Language models and large language models

Language models (LMs) focus specifically on natural language. At their core, they are probability models over sequences of tokens (words or subword pieces). Given the tokens seen so far, a language model estimates the probability of each possible next token. Early language models were relatively small and often built with n-gram statistics or modest neural architectures, where an n-gram model looks only at the last n tokens (for example, a 3-word sliding window) to predict the next one. Even then they powered useful features such as autocomplete and basic translation. Modern large language models (LLMs) scale this idea up dramatically:

- They use deep neural architectures, in practice almost always transformers.
- They are trained on very large text corpora, often containing billions or trillions of tokens.
- They have very large numbers of parameters, which lets them internalise a wide range of linguistic and factual patterns.

Because of this scale and architecture, LLMs can:

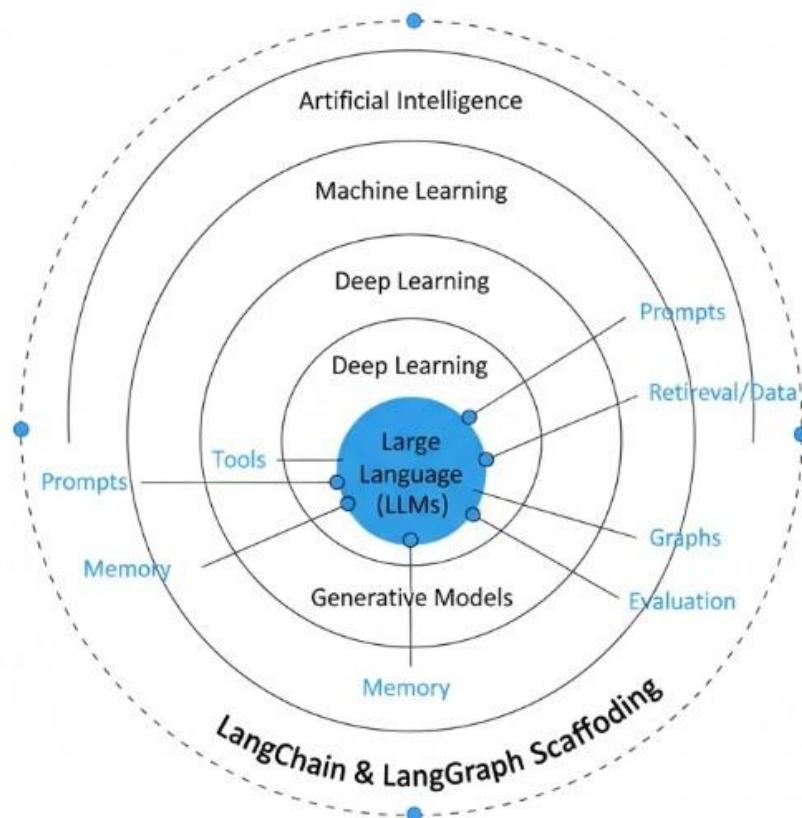
- Generate long, coherent passages of text.
- Answer questions across many domains.
- Summarise documents.
- Translate between languages.
- Write and explain code.

In a customer-support setting, an LLM can draft answers and explanations. In e-commerce, it can describe products, clarify requirements, and guide a user through choices. In trading and market analysis, it can summarise news, explain indicators, and produce structured reports based on textual and numerical context.

1.1.6 How the layers fit together

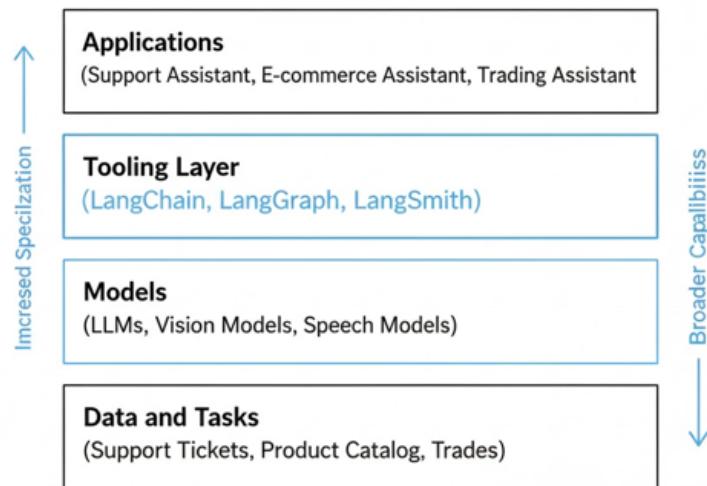
You can picture the concepts as nested and overlapping layers:

- AI is the overall goal: systems that show intelligent behaviour.
- ML is the part of AI that learns from data.
- Deep learning is a subset of ML that uses deep neural networks.
- Generative models are ML models (often deep) that create new data rather than just scoring existing data.
- Language models are models specialised for text, and large language models are deep, generative language models trained at very large scale.



The systems we build with LangChain and LangGraph sit on top of this last layer. They do not train new LLMs from scratch. Instead, they take existing large language models—usually provided by a cloud API or a local runtime—and provide the scaffolding around them: prompts, tools, retrieval over your own data, memory, graphs, and evaluation.

AI → LLMs Application Stack



In other words, LLMs are the “cognitive engine” inside your applications, but they are only one part of the overall system. The rest of this book shows how to connect that engine to real-world data and workflows in customer support, e-commerce, and trading, and how to shape its behaviour so that it serves concrete business goals rather than just generating plausible text.

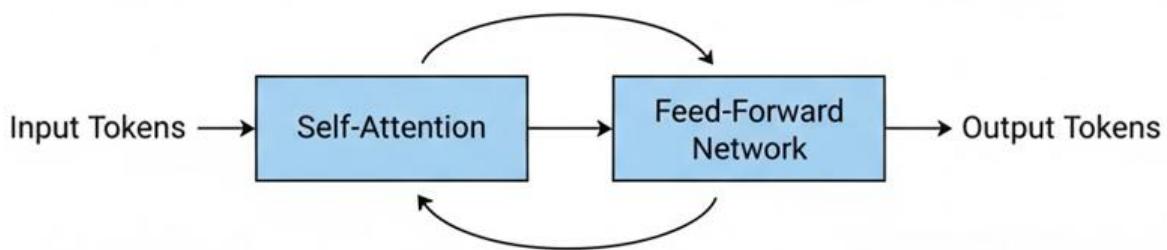
1.2 TRANSFORMER ARCHITECTURE AND ATTENTION MECHANISMS

Transformers are the standard neural network design behind modern language models. Their job is to read all the tokens in your prompt together and decide which ones should influence each other the most. You do not need to know the maths behind it, but you do need a simple picture of how they handle sequences and why that affects cost and limits.

Most of the models you will use with LangChain and LangGraph are based on the transformer architecture. In this chapter I try to explain why context windows exist, why prompt structure and ordering matter, why retrieval chunking is not optional, and why long conversations get slow and expensive.

1.2.1 From sequential models to transformers

Earlier neural models for text, such as recurrent networks and LSTMs (Long Short-Term Memory), processed sentences one token at a time from left to right. They carried a hidden state forward as they read, updating it at each step. That design made it hard to capture long-range relationships and difficult to train efficiently: the model had to “march” through the sequence step by step and struggled to keep information from far in the past.

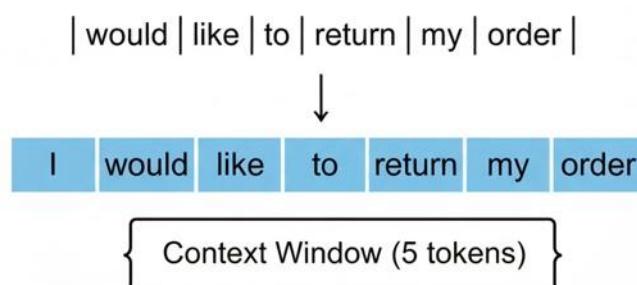


Transformers changed this by introducing a different idea. Instead of reading text strictly in order, they let every token in the sequence “look at” every other token. This mechanism is called self-attention. Because of self-attention, transformers can process all tokens in a prompt at once, and they can connect words that are far apart in the text more easily. This is one of the reasons modern LLMs feel coherent even on long, complex queries.

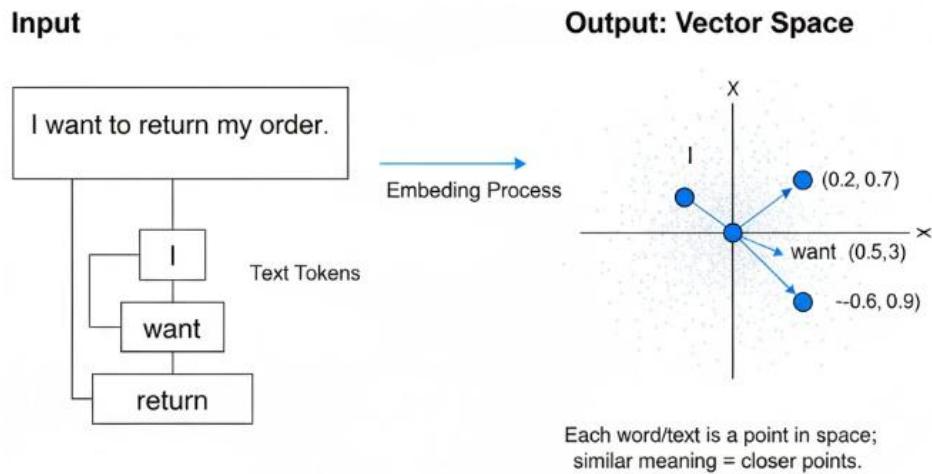
1.2.2 Tokens, embeddings, and position

The input to a transformer is a sequence of tokens. In practice, a token is usually a subword or small chunk of text; your prompt, previous messages, and retrieved documents are all broken into tokens before they reach the model.

Example Sentence:



Each token is mapped to a vector called an embedding. These vectors represent aspects of meaning in a numerical form that the network can work with. You can think of an embedding as a point in a high-dimensional space where tokens with similar meanings end up close to each other. On their own, embeddings do not say anything about order. If you only had embeddings, the model could not tell whether a token came first, last, or somewhere in the middle.



To handle order, transformers add positional information. For each position in the sequence, the model has a way to encode “this is the first token”, “this is the second”, and so on. Different model families implement positions in different technical ways, but the intention is always the same: let the network distinguish “dog bites man” from “man bites dog”, even though the words are the same.

1.2.3 Self-attention in simple terms

Self-attention is the operation that lets tokens look at one another. For each token, the model computes three internal vectors derived from its embedding:

- A query, which represents what this token is looking for.
- A key, which represents what this token offers to others.
- A value, which is the information that can be passed along.

For a given token, the model compares its query with the keys of all tokens in the sequence, including itself. These comparisons produce a set of scores that indicate which other tokens are more relevant. The model then normalises these scores and uses them as weights to combine the corresponding value vectors. The result is a new representation of the token that blends information from all the places it decided were important. You can think of this as: for each word, the model builds a summary of the other words that matter for interpreting it right now. Because this happens for all tokens in parallel, the transformer can model relationships across the whole prompt, not just between neighbours.

1.2.4 Multi-head attention and layers

In a transformer, self-attention is not run only once. It is run in several attention “heads” in parallel, and each head has its own learned projections for queries, keys, and values. Multiple heads increase expressiveness because different heads can attend to different aspects of the input representation.

Each head produces its own output (a context vector). The model combines the outputs from all heads (typically by concatenating them) to form the output of the attention sub-layer. A transformer layer then applies a separate feed-forward network as the next sub-layer. Each sub-layer is wrapped with residual connections and layer normalization. A full transformer stacks many of these layers.

Stacking many layers increases model capacity, but it also increases cost and can introduce optimization challenges.

1.2.5 Decoder-only transformers and generation

The original transformer architecture was an encoder–decoder. The encoder read an input sequence and built contextual representations; the decoder generated an output sequence, attending both to the encoded input and to what it had already generated. This design is still used for some tasks such as translation.

Most large language models you will use, however, are “decoder-only” transformers. They have only the decoder side of the architecture, specialised for predicting the next token given all previous ones. You can think of a decoder-only transformer as a text generator that reads everything you have written so far and then continues the text, one token at a time. During training, the model sees long sequences of tokens and, at each position, learns to predict the next token. During inference, generation is autoregressive (Autoregressive here just means that each new token is generated based on all the tokens already produced and then fed back into the model as part of the input for the next step.):

1. Take the current sequence (your system message, user prompt, conversation history, and any retrieved context).
2. Run it through the transformer stack to compute a probability distribution over the next token.
3. Select or sample the next token according to decoding settings (such as temperature or top-p).
4. Append the new token and repeat until you decide to stop.

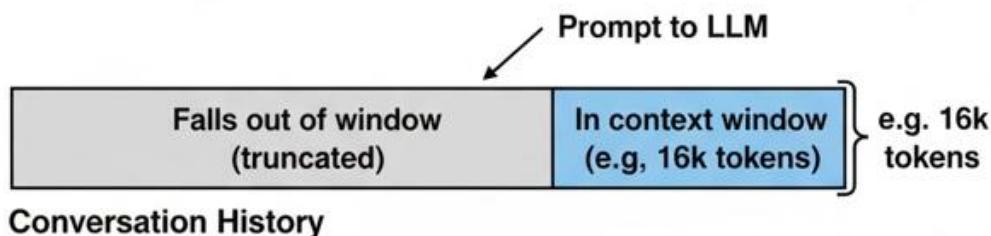
1.2.6 Cost, context windows, and long sequences

Every LLM call has a hard input limit: the context window. The model can only condition its output on the tokens you send in that request, which typically include system instructions, the user’s message, any retained conversation history, and any retrieved context you attach. If information is not inside that window, the model cannot use it directly.

This constraint is not just technical; it is economic. Most providers charge by token, so longer inputs and longer outputs increase cost. As conversations grow, the input token count can rise quickly unless you actively manage what you include each turn.

In production, you handle long inputs by being selective rather than exhaustive. For documents, that typically means chunking content and retrieving only the small set of chunks relevant to the current question (RAG). For conversations, it means trimming history, summarising older turns, or combining both so that the model sees what matters without paying to resend everything.

When the source material is far larger than any single context window, summarisation becomes a workflow rather than a single prompt. A common pattern is Map–Reduce summarisation: split the input into pieces, summarise each piece (map), then combine and summarise the summaries (reduce), repeating if needed. This same pattern is also used for long-form media (for example long videos) where “one-shot” prompting is not feasible.



1.2.7 Why prompt structure matters

In modern LLM applications, a “prompt” is rarely a single string. For chat models, it is typically represented as a sequence of messages with roles (system, user, assistant). Role separation matters because it lets you express stable behavioural constraints (system) separately from the user’s changing requests (user), even though providers may combine these into a single formatted input internally.

As applications grow, prompts become a maintained artefact rather than an ad-hoc paragraph. Prompt templates are the standard way to do this: they separate fixed instructions from runtime variables, standardise wording across an application, and make prompts easier to test and evolve without scattering string concatenation throughout the codebase.

Structure is also a portability issue. Different model providers publish different prompting and formatting guidance, and those differences can be significant enough that switching providers can change output quality unless you adapt your prompt structure. For that reason, production systems often keep provider-specific templates and select them dynamically.

As prompts get longer, you can keep them readable by composing them from parts. The LangChain reference describes partial substitution (pre-filling stable variables) and chaining prompt fragments together, which keeps complex prompts maintainable and reduces duplication.

1.2.8 Connecting back to agents

The constraints above explain why “agentic” systems exist. Raw LLM calls are limited by context windows, lack built-in orchestration for external tools, and do not provide a reliable control structure for multi-step tasks. Frameworks such as LangChain exist to supply those missing pieces: tooling integration, workflow control, and memory patterns that make multi-step behaviour repeatable rather than improvised.

From a design point of view, it helps to separate three layers: the model (which produces text), the prompt/context you provide (which shapes what it can do in that call), and the control loop around the model (which decides when to call tools, what to store, what to retrieve, and when to stop). That outer loop is the key difference between “a single completion” and an agentic system.

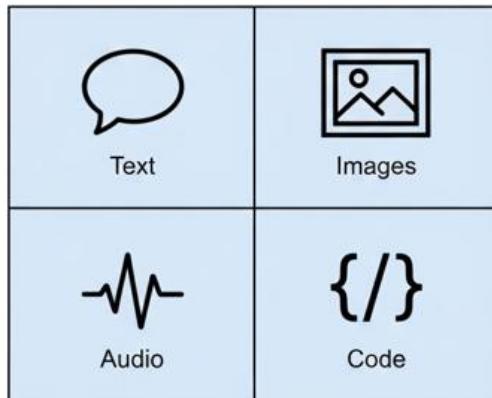
1.2.9 What makes a system an agent

An agent is best understood as a goal-directed control loop with an LLM in the decision seat. The system has an explicit goal to achieve (for example: “resolve the customer’s issue” or “produce a complete order summary”), and it keeps taking steps until it reaches that goal or hits a stopping condition.

A practical way to describe the mechanics is a repeated cycle: observe the current state, decide the next step, take an action (often by calling a tool), record what happened, and repeat. The key behavioural jump is that the system is no longer only generating text. It is choosing actions that change something outside the model—often in the “environment” of your own application: APIs, databases, and services—and then using the results to decide what to do next.

1.3 MODALITIES: TEXT, IMAGE, AUDIO, VIDEO, MULTIMODAL

Models do not only work with plain text. They can read and generate images, audio, video, or combinations of these. In real systems, you quickly move from toy text prompts to messy inputs like screenshots, PDFs, and voice notes.



Same core ideas, different input/output formats

In practice, “generative AI” is not one thing but a family of model types, each working on different kinds of data. Each kind of data is called a modality: text, images, audio, video, or any other structured signal the model can process. The most familiar to developers are text models, but the ecosystem now spans images, audio, video, and combinations of all of them. Understanding these modalities is important because the moment you design a real system you start bumping into documents, screenshots, PDFs, voice notes, and dashboards, not just plain text. The LangChain stack is designed to let you orchestrate different model types behind a single application surface.

1.3.1 Text as the core reasoning channel

Text is still the primary modality for most applications in this book. Large language models that generate and understand text power chatbots, summarizers, code assistants, and tools that reason over logs, emails, tickets, and product descriptions. In a customer-support assistant, the core tasks—understanding the user’s message, reading knowledge-base articles, and drafting a reply—are all text-centric. In an e-commerce assistant, the model works with product titles, descriptions, reviews, and order records. In a trading assistant, it reads news headlines, analyst reports, and chat-style questions about market moves. Architecturally, these are all “text in, text out”, even when the data originates from other formats that you first convert to text.

1.3.2 Images as inputs and outputs

Image models extend this by either generating images from text or interpreting existing images. Text-to-image models can synthesize marketing creatives, product visuals, or schematic diagrams given a description. Vision models that can “see” images let an assistant read screenshots of error messages, interpret charts saved as PNGs, or understand a product photo. For instance, a support workflow might accept a user-uploaded screenshot of an error dialog, route it through an image-understanding model, and pass the extracted description

into a text LLM for troubleshooting. In e-commerce, an assistant could help a user “find similar products” based on an uploaded photo. From a system-design point of view, images become another stream of information you can translate into text, embeddings, or structured data and then route through your LangChain or LangGraph flows.

1.3.3 Audio as a natural interface layer

Audio brings in both speech recognition and speech synthesis. On the input side, speech-to-text models let users talk naturally to your assistant: a customer describing a billing problem over the phone, or a trader dictating a query while on the move. On the output side, text-to-speech gives you voice responses that can be played back in call centres, mobile apps, or car dashboards. The underlying pattern for the rest of the system is the same: you convert audio to text, process it with your usual chains, tools, and graphs, then optionally turn the text answer back into audio. The intelligence of the system still sits mostly in the text-centric agent; audio models serve as adapters at the edges.

1.3.4 Video as a composite modality

Video is a richer and more complex modality that usually combines vision, audio, and sometimes text overlays. Today, many “video” applications in production are actually long-form processing of what is effectively a sequence of frames plus a transcript: summarizing recorded support calls, generating highlights for product demo videos, or extracting signals from earnings-call recordings in a trading context. Instead of reasoning at the raw pixel level frame by frame, you typically run specialized services to extract the transcript, key frames, or scene descriptions, then feed that distilled representation into your text models. Full text–image–audio video generation exists, but for most enterprise systems it is still more common to analyze and summarize video than to generate it from scratch.

1.3.5 Multimodal models and evolving modality strategy

Multimodal models sit at the intersection of all these modalities. A multimodal LLM can accept both text and other inputs (such as images, audio, or structured data) in a single prompt and reason over them jointly. This enables workflows like: “Here is a screenshot of my order status page and my last three support emails; explain why I was charged twice” or “Given this candlestick chart and the attached news article, explain the likely cause of the price spike.” In a multimodal e-commerce assistant, you might combine product photos, textual descriptions, and user queries to produce better recommendations than text alone would allow. Architecturally, multimodal models simplify some pipelines (fewer conversion steps) but raise new concerns about input formatting, context limits, and cost, which you must factor into your design.

Across the rest of the book, most examples will still look text-based: prompts, responses, and documents. Behind the scenes, however, you should treat modality as a design choice. The LangChain, LangGraph, and LangSmith stack gives you the orchestration tools to plug in these different modalities as your product matures, while keeping the core reasoning layer—usually a text or multimodal LLM—under coherent, testable control.

1.4 MAJOR LLM FAMILIES AND PROVIDERS; OPEN VS CLOSED VS HYBRID MODELS

When you call “the model”, you are always talking to a specific engine from a specific provider. Different models have different strengths, costs, and deployment options. Choosing between them is an engineering decision, not just a configuration detail.

When you build with LangChain and LangGraph, you never talk to “an LLM in general”. You always talk to a specific model, exposed by a specific provider, with its own strengths, limits, pricing, and constraints. LangChain hides many surface differences, so your chains and graphs look similar across providers, but model choice still has a big impact on quality, latency, cost, and how you handle data. This section gives you a practical map of the main families and the trade-offs between closed, open, and hybrid setups, so you can make sensible decisions for customer support, e-commerce, and trading systems.

1.4.1 Why model choice matters for your application

From the point of view of your code, every chat model does roughly the same thing: you send messages, you get messages back. Under the surface, though, different models vary in:

- General reasoning quality (how well they follow multi-step instructions).
- Domain behaviour (how they handle code, legal or financial text, product descriptions, multilingual input).
- Context window size (how much prompt and history they can “see” at once).
- Modality support (text only vs text plus images, audio, or video).
- Performance and price (latency, throughput, per-token cost).

Different model families emphasise different combinations of these qualities.

1.4.2 Closed, hosted model families

Closed models are provided as managed APIs. You do not run the model yourself; you send requests over HTTPS and pay per token. The provider controls training, updates, scaling, and security on the model side. In this category you will typically encounter:

- GPT-style models from general-purpose AI providers. These models are positioned as strong all-rounders for reasoning, coding, and multilingual text, often with support for images and other modalities. They are widely used for chat interfaces, copilots, and agent backends in many industries (OpenAI, Google, Meta).
- Assistant families from safety-focused providers. These models are tuned to be helpful and less likely to produce harmful content. They often offer very large context windows (useful for long documents, big codebases, or complex conversations) and are popular choices for enterprise-grade assistants and analysis tools (Anthropic, Cohere, Aleph Alpha).
- Cloud-platform models accessed through gateways such as managed AI services in major clouds. These services expose several model families under one authentication and billing layer, letting you combine, for example, a general chat model, a code model, and an embeddings model without juggling separate credentials (Amazon Bedrock, Microsoft Azure AI Foundry Model Catalog / Foundry Models, Google Cloud Vertex AI Model Garden).

1.4.3 Open-weight and local model families

Open-weight models publish their trained parameters so you can run them on your own hardware or in a cloud environment you control. You can still access many of these models via hosted APIs, but you have the option of self-hosting when you need more control over data or deployment. Important families here include:

- Llama models from Meta. Recent generations are released as open-weight transformers in several sizes, designed to be competitive with strong proprietary models at similar scales. They are widely used as base models for fine-tuning and as drop-in replacements for closed models in internal systems.
- Mistral models. These include small general-purpose language models, larger reasoning models, and specialised code models such as Codestral, many of which are released as open weights under permissive licences. They are designed to be efficient enough to run on commodity cloud hardware while remaining competitive on standard benchmarks.
- Other open models published through model hubs. Model platforms host a large catalogue of checkpoints from research labs and companies, including multilingual models, compact models optimised for edge devices, and domain-specialised variants.

Using open-weight models changes the balance of responsibilities:

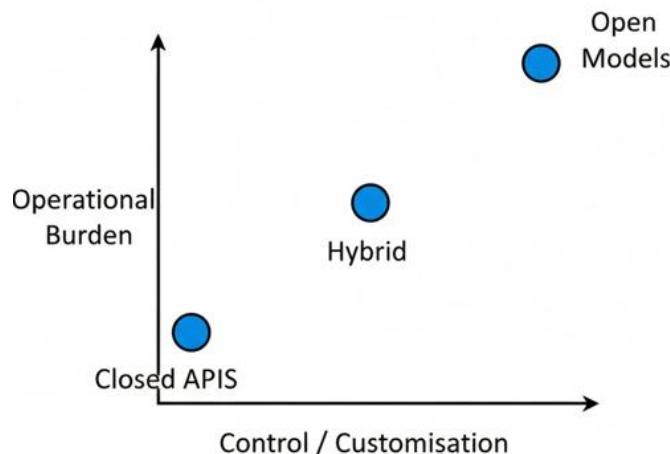
- You gain more control over where prompts, logs, and fine-tuning data live. For support tickets, order histories, or trading logs, this can be important for compliance and internal governance.
- You can shape the model more closely to your domain: fine-tune on your product catalogue, internal documentation, or domain-specific language.
- You take on the work of deployment, scaling, monitoring, and capacity planning for the model servers.

1.4.4 Open vs closed vs hybrid: practical trade-offs

Rather than a strict “open vs closed” choice, most production systems land in a hybrid position. Key trade-offs look like this:

- Capability vs control: Closed frontier models tend to lead in raw benchmark performance and multimodal features. Open-weight models give you more control over data, deployment, and customisation, and they are improving quickly. Hybrid designs let you reserve the most capable closed models for the hardest problems while using open models where control and cost matter more.
- Cost profile: Hosted models bill per token and hide infrastructure costs. Self-hosted open models require you to pay for compute and operations but can become cheaper at scale or when you reuse the same model heavily across internal workloads. A hybrid system might use a small open model for routing and simple tasks, calling an expensive closed model only for complex reasoning.
- Data and compliance: With closed APIs, prompts and outputs live on infrastructure controlled by the provider, subject to their retention and usage policies. Open-weight deployments can keep data within your own network and logging stack, which can simplify compliance with data-protection rules and sector regulations, especially for trading systems and regulated customer support flows.

- Flexibility and lock-in: If your application is tied to a single proprietary model, changes in pricing, terms, or performance are harder to absorb. LangChain's abstractions make it easier to build "model-agnostic" chains and graphs so you can test and gradually introduce alternative models—closed or open—without rewriting the rest of your application.



In practice, you might end up with:

- A support assistant that uses a closed model for free-form replies but relies on smaller open models for classification, routing, and summarisation of long ticket histories.
- An e-commerce assistant that calls an open model hosted in your own cloud for catalog RAG and rule-based price calculations, while using a hosted model for natural-language interaction with shoppers.
- A trading assistant that keeps all sensitive portfolio and order data behind your firewall, using an open-weight model for internal analysis while occasionally querying a closed model for general-market commentary without exposing client-specific details.

The rest of the book will treat "which model" as a pluggable choice. Chains, graphs, and evaluation flows will be designed so you can start with one provider, compare alternatives, and, when needed, run several model families side by side. That flexibility is one of the main reasons to use a framework stack in the first place: you can evolve your model strategy as the ecosystem and your requirements change, without redesigning your entire system every time a new model appears.

1.5 CAPABILITIES, BENCHMARKS, AND TYPICAL FAILURE MODES (HALLUCINATIONS, REASONING GAPS, BIAS, CONTEXT LIMITS)

Large language models are very good at sounding confident and fluent. That does not mean they are always correct or reliable. To use them safely, you need a clear view of what they are good at, how we test that, and where they tend to fail.

From the outside, an LLM looks simple: you send text in, you get text out, and the response often reads like something a careful human might write. Under the surface, it is a statistical machine that has learned patterns from huge amounts of data. It has real strengths, but also specific weaknesses that appear in predictable ways. If you treat it as “smart autocomplete” that occasionally says odd things, you will underestimate it and miss a lot of value. If you treat it as a reliable expert, you will eventually ship something that fails in an embarrassing or dangerous way. The goal of this section is to give you a realistic mental model of what LLMs can do, how we measure that in practice, and where they tend to break.

1.5.1 What LLMs are actually good at

The training objective is blunt: predict the next token. Nothing in that description mentions “answer questions”, “write code”, or “summarise documents”. Those abilities emerge at scale. When a model has seen enough text, several things happen:

- Models become very good at producing fluent text that looks like natural language. They internalise how sentences and paragraphs are usually formed and how different styles of writing tend to look. They have seen polite email closings, informal chat messages, legal clauses, technical documentation, and exam answers. They pick up patterns like “if someone asks a how-question, responses often start with an explanation” or “if a list starts with ‘first’, the next items often start with ‘second’, ‘third’, and so on”. This is why the output often feels “professional” even when you do not specify a style.
- The training data includes definitions, explanations, tutorials, Q&A threads, and reference documentation. The model is not a database, but it has internalised many regularities. It “knows” that Paris is a capital, that HTTP has methods like GET and POST, that a typical refund process involves conditions and time windows. When you ask about a topic, it can often assemble a plausible explanation by stitching together patterns it has seen before.
- Task flexibility appears once you start using instructions. With instruction-tuned models, you can say “Translate this text to English”, “Summarise this incident report”, or “Classify this message as bug, feature request, or general question” and the model will change its behaviour without you changing the code. You are still doing next-token prediction, but the space of likely continuations shifts depending on the instructions.
- If the training data includes enough source code, tests, and discussions about code, the model learns that “def add(a, b):” is often followed by “return a + b”, that you should close files, that HTTP clients raise exceptions on bad status codes, and so on. When you ask for a small function, it splices together what “looks right” from similar patterns.

All of this is powerful, but there is a common thread. The model does not “understand” in the human sense. It is very good at continuing patterns that look like the texts it has seen before.

1.5.2 How people measure capabilities: a quick tour of benchmarks

Benchmarks are standard tests used to compare models. Think of them as a rough map rather than a precise instrument. A benchmark is just a fixed collection of questions or tasks plus scoring rules that lets you compare models under the same conditions. Benchmarks tell you where a model family is strong or weak, and how new releases compare to older ones, but they do not guarantee anything about your specific application.

Some benchmarks look like exam papers. They present multiple-choice questions across many school and university subjects and measure how often the model picks the right answer. This gives a sense of general knowledge and reading comprehension. Over the last few years, newer models have moved from “does reasonably well” to “competitive with non-expert humans” on many of these tests. Other benchmarks focus on maths and reasoning. They give word problems that require several steps of logic or arithmetic. Here performance is still unstable. Models may solve some problems consistently, fail on others that look very similar, and be heavily influenced by how the question is phrased or by whether you ask them to “think step by step”.

There are also tests focused on code. They provide a function signature and a docstring, ask the model to fill in the body, and then run hidden tests. Results here show that models can generate many small correct solutions, but they also regularly produce code that passes simple tests while hiding edge-case bugs.

Finally, there are benchmarks concerned with behaviour and safety. They check whether models repeat common myths, whether they respond differently to similar questions about different groups, and how easily they can be pushed into producing harmful content. Scores on these tests move when providers adjust training and safety layers, which tells you that behaviour is not fixed; it is a design choice. The key point is that benchmarks paint a consistent picture:

- Language and general knowledge are strong and getting stronger.
- Formal reasoning remains fragile.
- Code generation is useful but not reliable enough to bypass testing.
- Safety and bias require explicit design and continuous monitoring.

When you pick models for your system, you will not choose based on a single number. You will read these patterns, run your own evaluations, and then use frameworks like LangSmith to track how things behave in your specific domain.

1.5.3 Common ways models fail

Once you understand the broad capabilities, you also need to internalise the typical failures. These are not rare edge cases; they appear as soon as you build anything non-trivial.

Hallucinations: sounding right while being wrong

Hallucinations are the most visible failure mode. The model produces output that is fluent, detailed, and wrong. It might:

- Invent a policy that resembles your real one but includes conditions you never defined.
- Describe a configuration flag or API parameter that does not exist.
- Quote a number or date that has no basis in any data you provided.

From the model's point of view, this is normal. Given your prompt and its internal state, the invented text is a perfectly reasonable continuation. It has no built-in mechanism to check whether that continuation matches reality. Without grounding, it will happily fill gaps with whatever fits the pattern.

Reasoning gaps: failing on simple structured tasks

Reasoning failures are subtler. The answer often looks reasonable at first glance but falls apart when you inspect the logic. You might see:

- Conditions that are ignored or applied inconsistently.
- Arithmetic that is off by small but important amounts.
- Chains of explanation where early statements contradict later ones.

The model can “act” as if it is reasoning, and sometimes it does get the steps right, especially if you prompt it to show its work. But it has no guarantee of completeness or consistency. It is still optimising for a likely continuation, not for a formal proof.

Bias and harmful content: inheriting patterns from data

Because the training data comes from human sources, models absorb both the good and the bad:

- They may repeat stereotypes when asked about certain professions, regions, or groups.
- They may generate aggressive or offensive responses under adversarial prompts.
- They may give systematically different advice depending on how a user describes themselves, even when that should not matter.

Mitigation happens through many levers: training choices, prompt design, filters, and human policies. None of these completely eliminate the underlying tendency. For any system that interacts with real users or sensitive topics, you have to treat bias and harmful content as risks to be managed, not as bugs that will eventually disappear.

Context limits and stale knowledge: hard boundaries you cannot ignore

Every model has two structural limits that your architecture must respect. The first is the context window. A model can only see a certain number of tokens at once. When you exceed that, you have to drop or summarise content. Long prompts also cost more and take longer to process. This is why you will later spend time on:

- Splitting documents into chunks and retrieving only what matters.
- Summarising old parts of a dialogue rather than replaying everything.
- Being intentional about which tool outputs and metadata you include.

The second is the training cut-off. The model does not automatically update itself when the world changes. If you ask about something that happened after its last training date and do not provide external information, it will fall back to guessing based on patterns from older data. That is why any system that cares about current facts, prices, or policies must connect to live data stores, not rely on the model's internal knowledge.

No actions, no memory: what is missing by design

Two final limitations are easy to overlook but fundamental. By default, a model cannot act. It cannot call an API, write to a database, or trigger a workflow. It can only suggest those actions in text. When you see an “agent” that books something, updates a record, or runs a query, that behaviour comes from the application code and tools you have wired around the model.

The model also does not remember previous interactions across calls. There is no built-in long-term memory. Everything it “knows” about the current situation is what you choose to include in the prompt: recent messages, retrieved context, and maybe some stored profile data. If you want continuity across turns or sessions, you must design and implement that yourself.

These gaps are exactly what frameworks like LangChain and LangGraph help with: they provide structures for tools, memory, retrieval, and control flow so you do not re-solve the same problems for every project.

1.5.4 Why this matters for everything that follows

It is tempting to think of an LLM as a smarter API and move on. The reality is more nuanced. You are working with a component that:

- Writes and rewrites text with high fluency.
- Contains a wide but shallow map of the world.
- Struggles with certain kinds of reasoning.
- Sometimes invents facts to keep the story going.
- Has hard limits on what it can see and when it was last updated.
- Cannot act or remember without help.

The rest of the book is about building systems around these facts instead of fighting them. Retrieval and tools exist to give the model access to real data. Memory and state exist to provide continuity without blowing up the context window. Graphs exist to break complex tasks into manageable steps. Evaluation and tracing exist so you can see what the system is actually doing and improve it safely.

If you keep this capability–failure-mode picture in mind, the design choices behind LangChain, LangGraph, and LangSmith will make much more sense. You are not trying to make the model perfect. You are trying to surround it with enough structure that its strengths are amplified and its weaknesses are contained.

1.6 SUMMARY

This chapter builds a usable mental model for designing systems around large language models, and it does it from the engineer’s point of view: what the model is, what it is good at, and where it will surprise you in production. It starts by tightening “AI” into a practical stack—AI, machine learning, deep learning, generative models, and language models—so you can place LLMs in context. The goal is not taxonomy. The goal is expectation management: an LLM is a component that generates and transforms text well, but it is not a complete application.

You then get a working picture of transformer-based models. Text becomes tokens, tokens become embeddings, and self-attention lets each token weigh the rest of the context through learned query/key/value projections. Multi-head attention runs several attention heads in parallel and combines their outputs, while transformer layers stack attention and feed-forward blocks with residual connections and normalization. You do not need the math to benefit from this model. You need the shape of the machine so you can reason about its limits and costs.

Those limits are concrete. Every call is bounded by a context window, and most pricing is token-based, so long prompts and long outputs increase cost. This is why long documents and long conversations force design choices: what to include, what to retrieve, what to summarise, and what to drop. This is also where agentic design fits: an agent is a goal-directed loop that uses the LLM as a decision component—observe state, decide, act via tools, record results, repeat until a stopping condition.

Finally, the chapter closes with a practical warning label: common ways models fail. They can produce fluent but incorrect content, miss constraints hidden in long context, follow the wrong instruction when prompts conflict, format outputs incorrectly, and behave inconsistently across small prompt changes. Treat these as normal failure modes, not surprises. The rest of the book is about building the surrounding structure—prompts, retrieval, tools, state, evaluation, and safeguards—so the system stays reliable when the model is not.

2 FROM MODELS TO APPLICATIONS AND AGENTS

Language models on their own are only one function in a much larger system. A playground demo or a short script that sends a prompt and prints a response is useful for exploration but does not resemble what you will run in production. Real applications need to talk to users and other services, access data, take actions, remember what has happened before, and behave in a way that is predictable enough to monitor and improve. This chapter is about that shift: from treating an LLM as a clever black box to treating it as one component inside an application or agentic system that you can design, constrain, and evolve.

The first part of the chapter clarifies what “LLM application” actually means. It walks through the path from a raw API call to a deployed system: how inputs are collected, how prompts and context are constructed, how outputs are parsed into actions or data structures, and how state, tools, and observability fit around the model. Along the way, it makes explicit the main limitations of raw LLM APIs that you inevitably hit when you move beyond experiments: no built-in tools or actions, no memory beyond the current call, narrow context windows and static knowledge, unreliable free-form outputs, safety concerns, and the absence of native evaluation or tracing.

With that foundation, the chapter introduces agentic applications by contrasting them with traditional software. Instead of hard-coded control flow, an agent operates in a loop: it observes a curated view of its environment, reasons using an LLM, chooses tools to call, and updates its state towards an explicit goal. Concepts such as environment, tools, memory, and goals become first-class design elements rather than implicit background details. The chapter shows how this changes your responsibilities as an engineer: you are no longer just wiring endpoints together, you are designing what the model can see, what it is allowed to do, what it should remember, and how it should decide that it is finished.

To keep these ideas concrete, the chapter proposes a simple ladder of agent sophistication, from Level 0 (a core reasoning engine with no tools or memory) through Level 1 (data-connected but code-driven flows) and Level 2 (single strategic agents) up to Level 3 (collaborating multi-agent systems). Each level is mapped to typical use cases and trade-offs in complexity, cost, and risk. The final section broadens the view to the socio-economic and governance context in which these systems live: how token pricing, infrastructure, and human effort add up to system-level cost; where risk comes from in practice; how guardrails and governance-by-design shape your architecture; and how emerging regulation affects design choices. By the end of the chapter you should have a clear mental model of what you are actually building when you say “LLM application” or “agent”, and a vocabulary for deciding how far up the agent ladder your own use cases need to go.

2.1 WHAT AN LLM APPLICATION IS (BEYOND A SINGLE API CALL)

If you have ever tried a language model through a playground or a simple script, you have already built the smallest possible “LLM app”: read a prompt, send it to the API, print the answer. It feels impressive the first time, but it is still just a glorified function call. There is no

persistent state, no connection to your data, no constraints, and no way to reason about safety or quality beyond reading the output. An LLM application starts where that demo stops. It treats the model as one component in a larger system that has users, data, tools, memory, and goals. The model is the “brain”, but the rest of the body—senses, muscles, nervous system, immune system—lives in your code, infrastructure, and processes.

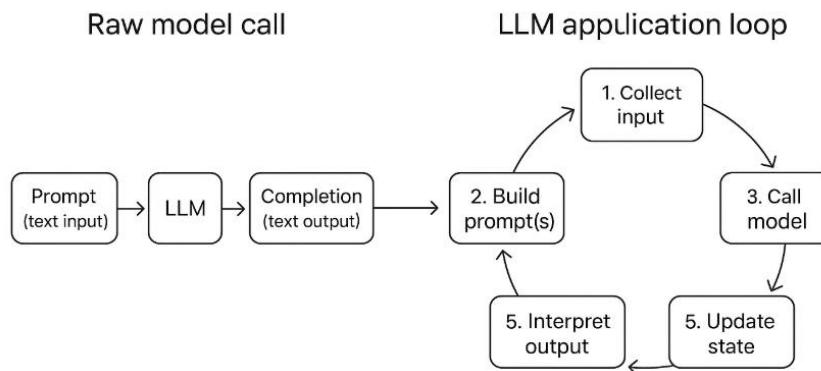
2.1.1 From raw model call to application

A raw model call is a simple mapping:

- Input: a string of text (the prompt).
- Output: a string of text (the completion).

There is no notion of what that text represents. It could be a question, a log line, a partial JSON object, or a story. The API does not care. An LLM application wraps this in a full loop:

1. Collect input from a user or another system.
2. Construct one or more prompts that mix this input with instructions and context.
3. Call the model (sometimes several times, in sequence or in parallel).
4. Interpret the output so it becomes actions, structured data, or UI changes.
5. Update state (databases, logs, long-term memory).
6. Repeat, often across multiple turns of interaction.



That loop is what you deploy, monitor, and maintain. The model is a powerful step inside it, but never the whole story.

2.1.2 The main building blocks of an LLM application

In practice, most LLM applications end up with a similar set of layers, even if the details differ.

- Interface layer: Something has to talk to the model on behalf of users or other services: a web page, a chatbot widget, an API endpoint, a CLI tool, a background worker. This layer handles authentication, rate limiting, and basic validation, just like any other service.
- Prompt and context construction: Raw user text is rarely enough. You need to decide:
 - What instructions the model should follow (“answer briefly”, “return JSON”, “follow this policy”).
 - What extra context to add (previous messages, system settings, retrieved documents, tool results).

- How to structure all of this so the model can tell instructions, user input, and context apart.

This is where prompt templates, message schemas, and retrieval pipelines come in. You are essentially building the input language that the model will see.

- Model backend(s): Behind the scenes, you may call:
 - One main model for the core task.
 - Smaller or cheaper models for classification, routing, or safety checks.
 - Different providers or local models for different parts of the workflow.

An application decides when to call which model, with which parameters, and under what constraints (time, cost, privacy).

- Output parsing and post-processing: The model returns text. The rest of your system wants something more specific:
 - A JSON object with fields you can store.
 - A decision ("approve" / "reject").
 - A sequence of tool calls with arguments.
 - A message to show to the user.

That means parsing, validating, and sometimes correcting the output, then mapping it onto your domain objects and workflows.

- Data and tools: Most useful behaviour comes from connecting the model to the rest of your stack:
 - Data sources: relational databases, search indexes, logs, document stores, vector stores.
 - Tools: HTTP APIs, internal services, scripts, calculators, rule engines.

The model does not magically know these exist; you expose them in a structured way and decide which ones are safe to use.

- State and memory: Applications need continuity. You might track:
 - Short-term conversation state.
 - Long-term user preferences or past interactions.
 - Intermediate results in a multi-step workflow.

Some of this state is for your own systems (databases, analytics). Some is fed back to the model as context so it can behave consistently.

- Observability and control: Finally, you need visibility:
 - What prompts are actually being sent?
 - Which tools are used, and how often do they fail?
 - How do latency, cost, and accuracy change over time?

This is where systematic tracing, logging, and evaluation come in. Without them, you are effectively debugging by hunch.

LangChain exists largely to give you abstractions for these pieces—prompt templates, model wrappers, parsers, tools, retrievers, memory components—so you can assemble them without reinventing everything. LangGraph adds a more explicit control layer for state and flow. LangSmith adds structured tracing and evaluation.

2.1.3 From application to “agentic system”

Once you have these building blocks, you can start thinking in terms of agents rather than static pipelines. In this book, when we say, “agentic system”, we mean a system that:

- Receives a mission or goal, not just a one-shot question.
- Perceives its environment by reading user input, state, and retrieved data.
- Thinks by calling models, possibly in several steps.
- Acts through tools and services, not just text responses.
- Learns by updating its state and adjusting future behaviour.

A single API call covers only a thin slice of that loop. An agentic application implements the whole thing. It might:

- Ask follow-up questions when information is missing.
- Choose between different tools depending on the situation.
- Maintain a working state over several hops in a graph.
- Escalate to a human or a simpler deterministic flow when a threshold is exceeded.

The model becomes the decision and reasoning engine inside a controlled environment that you define. That environment—data, tools, state, policies—is what ultimately determines whether the application is safe and useful.

2.1.4 Why this distinction matters before you write any code

If you keep thinking of LLM usage as “call the API and see what happens”, you will tend to:

- Hide too much logic in one giant prompt.
- Treat failures as random quirks instead of symptoms of missing structure.
- Struggle to debug, test, or evolve the system over time.

If you think in terms of applications and agentic systems, you start from different questions:

- What is the goal of this system, and how will we know it is doing a good job?
- What information does it need to see, and how will we provide it?
- Which actions must be done by deterministic code, and which can be delegated to the model?
- Where do we store state, and how do we feed only the relevant parts back into prompts?
- How do we observe behaviour and change it safely?

The rest of this part of the book takes that perspective for granted. We will treat the model as a powerful but limited component, and focus on the surrounding architecture—the prompts, graphs, tools, memory, and evaluation loops—that turns isolated API calls into something you can run, monitor, and improve as a real system.

2.2 LIMITATIONS OF RAW LLM APIs: TOOLS, MEMORY, RELIABILITY, EVALUATION

When you first meet an LLM as an API, it feels wonderfully simple: you send a prompt string, you get back model-generated text. For a quick script or a prototype chatbot, that can be enough. The trouble starts the moment you try to build a real application around it. You discover that the model cannot see your data, cannot take real actions, forgets everything between calls, occasionally makes things up, and gives you almost no help when you try to debug or evaluate its behaviour at scale. None of this is a bug in the API. It is simply not designed to be an application platform; it is a narrow interface to a text engine. This section walks through the main limitations you hit if you rely on raw LLM APIs alone, grouped around four themes: tools, memory, reliability, and evaluation. Later parts of the book will show how LangChain, LangGraph, and LangSmith exist almost entirely to plug these gaps.

2.2.1 No tools, no actions

A raw model call can only do one thing: generate text. That text can describe actions (“I have cancelled your order”, “I have updated your account”), but nothing in the API actually touches your systems. There is no built-in ability to:

- Run a database query.
- Call a REST API.
- Execute a calculation in your code.
- Update a record or trigger a workflow.

Even when providers expose “function calling” or “tool calling” features, what you get is still a primitive. The model can propose a structured call like:

```
{"tool": "get_order_status", "arguments": {"order_id": "123"}}
```

but the runtime does not magically know what `get_order_status` is, how to execute it, what to do if it fails, or how to combine several such calls into a coherent workflow. All of that orchestration logic is left to your application. If you try to manage this by hand around a bare API, you quickly end up with:

- Ad-hoc JSON protocols between your code and the model.
- Custom parsing and validation for every tool call.
- One-off error handling scattered across prompts and glue code.

You can build systems like this, but they become fragile and hard to evolve. Frameworks and agent patterns exist because almost every serious application needs a systematic way to expose tools, run them safely, and stitch their results back into the model’s context.

2.2.2 No memory beyond the current call

LLMs are stateless at the API boundary. Each request is independent: the model sees only the text you send this time. It does not remember what happened in previous calls, which user is speaking, or what the application did earlier. In simple demos, this limitation is papered over by sending the entire conversation history in every request. That starts to fall apart once you do anything non-trivial:

- Conversations get long, and you hit the model’s context window.

- Different kinds of state need different lifetimes (short-term chat, medium-term session state, long-term profile).
- You want to reuse information across sessions without replaying everything as text.

The raw API gives you no concepts for any of this. There is:

- No notion of “session” or “user” built into the model.
- No persistent memory store.
- No guidance on what to keep, what to summarise, and what to forget.

As the developer, you must design a memory architecture: where state lives (databases, key-value stores, vector indices), how it is retrieved, and how a small relevant slice is turned back into prompt text. Later, when we talk about memory components, graph state, and retrieval over history, we are really talking about reusable answers to this fundamental limitation.

2.2.3 Narrow context windows and static knowledge

Two more structural constraints show up as soon as you leave toy examples behind: context windows and training cut-offs.

The context window is the maximum number of tokens (roughly: words and punctuation) a model can process at once. Every piece of text you send counts against this: system instructions, user messages, conversation history, tool outputs, retrieved documents. When you hit the limit, you must drop or summarise something. Long prompts also cost more and are slower to process.

The training cut-off is the point in time when the model stopped seeing new data during training. Beyond that date, it has no built-in knowledge of events, changes, or new content. It can guess, but it cannot know. Taken together, this means:

- You cannot simply “dump the database” or “dump the entire chat log” into every prompt.
- You cannot rely on the model to know recent facts, updated policies, or current values.

Around the raw API, there is no help for document chunking, selective retrieval, or knowledge integration. Those are left for you to solve. Much of the practical machinery in LangChain—document loaders, splitters, retrievers, vector stores, exists precisely to give you reusable patterns for feeding the right slice of context to the model at the right time.

2.2.4 Reliability problems: hallucinations, brittle reasoning, unpredictable formats

From the outside, a model’s responses can look reassuringly polished. Inside, it is still just a probabilistic next-token predictor. That mismatch shows up in three reliability issues that are hard to ignore once you go beyond demos.

Hallucinations: When the model lacks the information it needs, it usually does not say “I don’t know”. It produces the continuation that best fits its training patterns, which often means:

- Invented facts or references.
- Made-up field names, parameters, or API shapes.
- Confident statements that have no backing in any data you provided.

Scaling models helps somewhat but does not eliminate hallucinations. Without external grounding, they are an intrinsic behaviour, not a corner case.

Reasoning gaps: The model can simulate reasoning, and sometimes it gets the steps right, especially when prompted to show its work. But it has no formal notion of “proof” or “invariant”. You will see:

- Conditions that are ignored halfway through an answer.
- Multi-step instructions that are followed partially but not completely.
- Calculations that look right locally but are wrong when you recompute them.

If you rely on the model alone for business rules, financial calculations, or strict constraints, you are asking for trouble (LLM are not deterministic). The raw API gives you no mechanism to attach external calculators, rule engines, or validators. That wiring must come from your application.

Unpredictable formats: For many tasks, you want structured output: JSON objects, lists, or well-formed code. Basic APIs expose a text field and little else. You can ask politely the model to respond in JSON, and often it will, but not always. You will encounter:

- Trailing comments or explanations mixed with the JSON.
- Slightly malformed structures (missing quotes, extra commas).
- Deviations from your schema when the model thinks a different structure “makes more sense”.

The more you integrate model output into downstream systems, the more painful this becomes. From the raw API’s point of view, all of this is the same: just more text. Structured-output helpers, schema enforcement, and repair loops all live outside.

2.2.5 Safety, bias, and misuse

Technical limitations are only half the story. A model trained on human data will also absorb human biases and harmful patterns. Out of the box, it may:

- Produce biased or unfair language when describing people or groups.
- Echo offensive or unsafe content if prompted in certain ways.
- Give misleading advice when a user implicitly asks for something risky or unethical.

Providers often add safety layers on their side of the API, but those are generic and conservative by design. They cannot see your specific policies, use cases, or risk appetite. If you simply forward model outputs to users, you inherit all of these behaviours without any extra control. Raw APIs also have no concept of:

- Role-based access to tools.
- Different safety expectations for different actions.
- Human approval points for high-impact decisions.

Those are system-level concerns. Guardrails, moderation filters, least-privilege tool design, and human-in-the-loop workflows all sit in the application layer, not in the model endpoint.

2.2.6 No built-in evaluation or observability

With a bare API, you see prompts and completions. That is enough when you are experimenting manually. Once your system sees real traffic, it is nowhere near sufficient. You will want to answer questions such as:

- How often does the system fail on a particular kind of query?
- Did yesterday’s prompt change improve or degrade quality on key tasks?

- Which tool calls are slow, flaky, or rarely helpful?
- What does a typical “bad interaction” look like end-to-end?

The raw API offers no:

- Standard tracing format for intermediate steps and tool calls.
- Concept of evaluation datasets, test runs, or regression checks.
- Built-in metrics linked to your business goals.

You can log everything yourself and build a bespoke analysis stack, but that quickly turns into a parallel project. Framework-level tracing and evaluation exist precisely because most teams do not want to reinvent their own ad-hoc observability platform for every LLM application.

2.2.7 Why this matters before you write any framework code

The limitations of raw LLM APIs are not academic curiosities. They are exactly the reasons higher-level frameworks and patterns exist.

- The absence of tools and actions is why you need a standard way to wrap functions, APIs, and services and expose them to the model.
- The absence of memory is why you need explicit state, retrieval, and summarisation strategies.
- The unreliability of free-form text is why you need structured output, external calculators, guardrails, and clear control flow.
- The lack of evaluation and observability is why you need tracing and systematic test runs, not just eyeballing a few responses.

Once you see the raw API for what it is: a powerful but blind, amnesic, probabilistic text box, the rest of the stack starts to make sense. LangChain and LangGraph give you the building blocks to wrap that box in tools, memory, and explicit workflows. LangSmith lets you see what that wrapped system is doing and how well it is behaving over time. The following chapters are not about making the model “smarter”. They are about building enough structure around it that its strengths become usable in production, while its weaknesses are contained and monitored rather than left to chance.

2.3 AGENTS VS TRADITIONAL APPLICATIONS: ENVIRONMENT, TOOLS, MEMORY, GOALS

If you have spent years building web services, you probably carry a very stable mental model of how backend systems work. A request arrives, some deterministic logic runs, data is fetched or updated, and a response is returned. The behaviour is meant to be predictable: the same input, under the same conditions, leads to the same output.

Agents change that picture, but not by throwing it away. Instead, they add a new layer on top: a reasoning loop powered by an LLM that can decide how to use the deterministic pieces you already have. To work with agents effectively, it helps to compare them with traditional applications along four axes: environment, tools, memory, and goals.

2.3.1 Traditional applications: the familiar mental model

In a conventional service, the environment is mostly implicit. You have:

- A request object (headers, body, authentication).
- Some in-memory data (configuration, caches).
- Databases and external APIs behind client libraries.

Your code “knows” how to navigate this environment. A handler receives an HTTP request, parses it, validates inputs, calls a series of functions and services, and returns a response. The control flow is hard-coded in the language you use. State is externalised in well-defined places: relational databases, key-value stores, message queues, session stores. You do not talk about “memory”; you talk about tables and IDs. Goals are encoded as rules and invariants. “An order is valid if this condition holds”, “a user is authorised if that check passes”, “this endpoint always returns JSON in this schema”. You rarely write down the goal as a separate object; it is scattered across routes, services, and tests. This model has major advantages: debuggability, reproducibility, and clear contracts. You can trace what happened by following logs and call graphs. You can prove certain properties by inspecting the code. Agents layer themselves on top of this world; they do not replace it.

2.3.2 Agentic applications: an LLM in the control loop

An agent, in the sense we use in this book, is a software component that:

- Perceives some representation of the current situation.
- Chooses actions using an LLM as a reasoning engine.
- Acts by calling tools (your existing services and functions).
- Repeats this observe-decide-act loop until a goal is reached or it decides to stop.

Where traditional code has fixed control flow, an agent has flexible control flow guided by the model’s decisions. The logic is not “on event X, always call A then B then C”; it is “given this goal and what you see right now, choose the next thing to do from this toolbox”.

For this to be safe and useful, you have to make four elements explicit: environment, tools, memory, and goals.

2.3.3 Environment: what the agent can actually see

The environment is everything that is potentially relevant to a decision: user input, recent interactions, database records, configuration, logs, external events. A traditional service can

access all of this directly: it can read any table, call any API, and inspect any part of the request. The code is in charge and can reach into whatever layer it needs. An agent is different. The LLM at its core has no direct access to your runtime. It cannot magically see your database or request objects. The only environment it knows is what you encode and feed into its context window:

- The current user message and recent exchanges.
- Selected records fetched from storage or search.
- Tool outputs from previous steps in the loop.
- Any additional state you decide to include (flags, configuration, metadata).

Designing the environment for an agent is therefore a conscious act. You decide:

- What evidence to retrieve for this step.
- How to format it (raw text, structured snippets, summaries).
- How much to include without blowing the context window or confusing the model.

If you expose too little, the agent will guess and hallucinate. If you expose too much, important details get drowned in noise and you waste tokens. Good agent design is largely about curating the slice of the environment the model sees at each decision point.

2.3.4 Tools: how the agent acts on the world

In traditional applications, tools are just functions and services. You write code that calls a database client, an HTTP API, a search engine, a payment gateway. The call graph is fixed: when a given route is hit, you know exactly which functions will run and in what order. In an agentic system, you wrap these capabilities as tools: named operations with a description and a callable implementation. The agent runtime then follows a pattern like:

1. Show the model the current context and the list of available tools.
2. Ask it to choose either a tool to call (with arguments) or to produce a final answer.
3. Execute the chosen tool in regular deterministic code.
4. Feed the result back into the model as new context.
5. Repeat until the agent decides it is done, or you hit a safety limit.

From the outside, the tools are just your existing services. From the inside, the agent sees them as options it can pick from to move towards its goal. This inversion raises questions you did not have to ask before:

- How do you describe tools, so the model picks the right one?
- How do you validate and sanitise tool arguments before executing them?
- What happens if a tool fails, times out, or returns unexpected data?
- Which tools should be available to which agents, with which permissions?

The answers live in your application code, not in the model. The LLM proposes actions; your runtime is responsible for guarding the boundaries, enforcing least privilege, and handling errors gracefully.

2.3.5 Memory: keeping state as input to reasoning

Agents live or die by what you show them at each step. “Context engineering” is the discipline of choosing what information goes into the model’s prompt, how it is packaged, and how it is

kept small enough to be useful. In that sense, memory design is not just about persistence. It is a context-engineering choice: it determines the quality of the next prompt, and therefore the quality of the next decision.

Traditional systems already have state, but they do not treat it as “memory for a thinker.” They typically separate concerns like:

- Long-term state in databases.
- Short-lived state in sessions or caches.
- Logs for auditing and debugging.

Agents add a new question: which parts of that state should be presented back to the model as context for reasoning, and when. “Memory” here is the subset of state that is deliberately re-injected into the model’s input. It usually comes in layers:

- Short-term history: what was just said or done in this conversation or workflow.
- Working memory: intermediate results, retrieved documents, partial plans, tool outputs.
- Long-term facts: user preferences, past decisions, durable constraints that should carry across sessions.

Designing memory means making explicit trade-offs:

- What gets stored across steps and across sessions.
- In what form it is stored (raw text, structured records, embeddings for similarity search).
- How it is retrieved and summarized so the model sees what matters without being buried in noise.

This is not only a storage problem. It is an input-shaping problem. Memory is doing its job when the model can behave consistently and efficiently over time, without you having to re-teach it the same context in every prompt.

2.3.6 Goals: making intent explicit

In a traditional application, the “goal” of a piece of code is usually implicit. You don’t pass around an object called Mission. Instead, success is baked into the implementation: route handlers, validation rules, and business logic collectively define what “done” means.

Agents behave more reliably when you make that intent explicit. A goal is a short statement of what the system is trying to achieve, for example:

- Answer this question faithfully, using the tools you are allowed to use.
- Produce a draft that satisfies these constraints.
- Drive this workflow to a valid end state; ask for help when you cannot proceed safely.

You can express goals in a few different places:

- System instructions that define the role and objective (“You are...”, “Your objective is...”).
- Task inputs that describe the job to be done (“Given this task, plan the steps and execute them.”).
- Explicit goal fields carried in your graph state, so every step can read the same objective.

Once a goal is explicit, the agent loop becomes straightforward: given the goal and the current state, what is the best next action? That action might be to call a tool, ask a clarifying question, consult memory, or stop and respond. Explicit goals also make evaluation much less hand-

wavy. You can measure outcomes against the stated intent: how often the agent reaches the target result, how many steps and tool calls it typically needs, and where it tends to stall or require human intervention. Those metrics connect directly to tracing and evaluation later in the stack.

2.3.7 Putting it together: wrapping, not replacing, your existing systems

Seen through these four lenses, the contrast becomes clear:

- Traditional applications operate in an implicit environment, call tools in fixed sequences, treat state as data rather than memory, and bake goals into code paths.
- Agentic applications expose a curated environment to an LLM, wrap existing capabilities as tools, design memory as input to reasoning, and express goals as first-class, evaluable concepts.

The important thing is that agents sit on top of the systems you already know how to build. Databases, APIs, queues, configuration management, and monitoring do not disappear. They become the canvas on which you place agents, with frameworks like LangChain and LangGraph providing the machinery to connect environment, tools, memory, and goals into a controlled loop. Once you start thinking in these terms, “using an LLM” stops meaning “make a single API call somewhere in the stack” and starts meaning “give a reasoning engine a carefully chosen view of the world, a safe set of actions, some structured memory, and a clear mission—and then wrap that in enough structure that you can observe, constrain, and improve what it does over time.”

2.4 LEVELS OF AGENT SOPHISTICATION (LEVEL 0–3) AND USE-CASE MAPPING

When people say “agent” they often mean wildly different things. Sometimes it is just “we wrapped a chat model in a web API”. Sometimes it is a long-running system that coordinates multiple models, tools, and humans over days. If you do not name these differences, design discussions get fuzzy very quickly: one person imagines a simple helper that calls a single tool; another imagines an autonomous workflow that can open tickets, send emails, and schedule jobs. To keep this book grounded, we will use a simple ladder of sophistication with four rungs: Level 0 to Level 3. Each level adds specific capabilities and also new ways to go wrong. Higher is not automatically better; it is just more flexible, more powerful, and more expensive (in all senses: engineering effort, operational risk, and compute cost). The ladder gives you a shared vocabulary: “this feature only needs Level 1”, “this workflow really demands Level 2”, or “this governance requirement pushes us into Level 3 territory”. We will reuse this ladder throughout the book when we introduce new patterns in LangChain, LangGraph, and LangSmith. You should start to get a feel for where your own ideas sit on it as you read.

2.4.1 The agent ladder in overview

At a high level, the four levels look like this:

- Level 0 – Core reasoning engine: A plain model (or simple chain) that takes a prompt and returns text. No tools, no real memory, no environment.
- Level 1 – Connected problem-solver: A model wrapped in deterministic logic that can call tools and use retrieval. Control flow is still mostly fixed in code.
- Level 2 – Strategic problem-solver: A single agent that uses an LLM to decide which tools to call, in what order, and with what context, maintaining richer state over time.
- Level 3 – Collaborative multi-agent system: A set of specialised agents and tools coordinated by an explicit workflow (often a graph) to pursue more complex, multi-step goals.

You can think of this as shifting the “intelligence” and flexibility from the surrounding code into the model-driven part of the system. At Level 0, the model is a glorified library function. At Level 3, the model(s) participate in high-level planning and coordination, with your code providing structure, guardrails, and integration points.

2.4.2 Level 0 – Core reasoning engine

Level 0 is where most people start: you send a prompt to a model and read the answer. The model might explain a concept, rewrite a paragraph, suggest a regular expression, or summarise a log snippet. In LangChain terms, this can be as simple as a `PromptTemplate` wired directly into a model via the LangChain Expression Language (LCEL). At this level:

- The only “environment” the model sees is the current prompt.
- There are no tools. If the model appears to “look something up”, it is just guessing from training data.
- There is no state across calls beyond whatever you manually replay in the prompt, such as previous chat turns pasted back in.

The upside is simplicity. You can prototype very quickly, experiment with prompts, and attach basic functionality to existing products: text rewriting widgets, “explain this page” features, small code helpers. The downside is that everything is in-model and out of your control. There is no reliable way to ensure the answer reflects current data, no way to act on external systems, and no principled way to keep long-running context beyond crude “just paste the whole conversation back in” approaches. In the rest of the book, Level 0 will mostly appear as the inner reasoning step inside more structured systems: the core “think” operation that we surround with tools, memory, and workflows.

2.4.3 Level 1 – Connected problem-solver

Level 1 is the first step where the system starts to know anything about the real world. Here you keep control flow simple and mostly deterministic, but you connect the model to your data and services:

- You use retrieval-augmented generation (RAG) over your documents, logs, or product data.
- You call databases, search engines, or HTTP APIs in fixed places in the pipeline.
- You might combine several of these operations into chains, but the sequence is decided by your code, not by the model.

A typical Level 1 flow looks like:

1. Take the user input.
2. Decide, in code, which data sources to query (vector store, SQL, API...).
3. Retrieve or compute the relevant information.
4. Build a structured prompt that includes the user input and retrieved context.
5. Call the model once to generate an answer.

LangChain’s loaders, text splitters, embeddings, vector stores, retrievers, and simple tools live comfortably at this level. LangGraph may or may not be involved; if it is, you are usually using it as an explicit state machine for a relatively straightforward pipeline rather than as a flexible agentic controller. This level gets you surprisingly far. You can build grounded question-answering, document chat, and data-aware assistants where retrieval and tool outputs substantially reduce the chance of unsupported claims. However, grounding does not remove the problem: the model can still misinterpret retrieved text, overgeneralize, or fabricate details around gaps. Treat Level 1 as ‘lower risk, not zero risk’, and use output checks, evaluation datasets, and monitoring to quantify how often the system stays faithful to its sources.

2.4.4 Level 2 – Strategic problem-solver

Level 2 is where we cross into what this book will consistently call “agents”. The key change is who decides what happens next. At Level 1, your code owns the plan: “first retrieve, then call the model, then parse, then maybe call another tool”. At Level 2, you let the model participate in the planning and adapt its behaviour based on intermediate results. You give it:

- Descriptions of available tools (what they do, what arguments they take).
- Access to some form of memory (recent messages, stored facts, or graph state).
- A way to see previous tool outputs and its own earlier messages.

Then you run a loop:

1. The agent sees the current goal, the recent context, and the tools it can use.
2. It decides either to call a tool (with specific arguments) or to produce a final answer.
3. If it calls a tool, your system executes that tool, records the result, and feeds it back.
4. Repeat until the agent chooses to stop or you hit some limit (steps, time, tokens).

In LangChain, this is the world of agent executors and tool-calling models. In LangGraph, it is the world of graph nodes that encapsulate LLM calls and tool invocations, with edges representing possible paths and a shared state object holding memory and intermediate results. Two concepts become central at this level:

- Planning: breaking a goal into smaller steps, choosing the next tool or subtask based on current progress, and sometimes revising the plan when something fails.
- Context engineering: deciding what subset of state, retrieved data, and history to show the model at each step so it can make good decisions without drowning in irrelevant detail.

Level 2 systems feel qualitatively different to work with. You are no longer just designing a pipeline; you are designing a “colleague” that can read instructions, look things up, take actions in a controlled environment, and explain what it did. You get more flexibility and expressiveness, but you now have to think about:

- How many steps you allow before you stop the loop.
- How you detect when the agent is stuck or oscillating.
- How you handle tool errors, malformed tool calls, or partial failures.
- How you evaluate behaviour across many runs instead of just spot-checking one answer.

This is where LangSmith-style tracing and evaluation starts to feel mandatory rather than optional.

2.4.5 Level 3 – Collaborative multi-agent systems

Level 3 systems stop pretending that one agent can or should do everything. Instead, they mirror how human organisations handle complexity: by dividing responsibilities, specialising roles, and coordinating through shared state and communication patterns. In a Level 3 design you have multiple agents, each with:

- A specific role or area of expertise.
- Its own prompts, tools, and sometimes its own model.
- Access to a slice of shared context and memory.

You then define how they collaborate:

- A “planner” agent might turn a high-level request into a set of sub-tasks and assign them to other agents.
- Specialist agents might perform retrieval, analysis, drafting, or policy checks.
- A “critic” or “reviewer” agent might examine outputs from others and request revisions.
- Human participants might be explicitly placed in the loop at key points, with the system pausing until they approve or comment.

LangGraph comes into its own here. A multi-agent system is naturally expressed as a graph where agents and tools are nodes, messages and state updates flow along edges, and

checkpoints allow you to persist and resume long-running processes. You can encode complex topologies: parallel branches, voting mechanisms, escalation paths, or structured negotiations between agents. Level 3 buys you:

- Modularity: you can evolve or replace one agent without rewriting the whole system.
- Interpretability: you can inspect which agent did what, in what order, using which tools.
- Specialisation: you can tune prompts, tools, and even models for each role.

It also introduces serious complexity. You now have to reason about:

- Coordination failures (two agents pulling in different directions).
- Resource usage (multiple agents all iterating and calling tools).
- Emergent behaviours that are difficult to predict from looking at any single agent in isolation.

For many applications, Level 3 is overkill. But if you are building workflows that span departments, require multiple forms of expertise, or must satisfy tight governance and audit requirements, this is the level where the design space opens up.

2.4.6 Choosing the right level

A natural temptation, especially after reading about sophisticated agentic patterns, is to go straight to Level 3 for everything. This almost always backfires. The ladder is most useful when you treat it as a progression, not a menu of equally easy options. A pragmatic approach looks like this:

- Start at Level 0 when you are exploring ideas, prototyping prompts, or adding small “assistive” features.
- Move to Level 1 as soon as correctness and freshness matter, and you need the model to speak about your actual data, not just its pretraining.
- Promote a use case to Level 2 when the interaction becomes multi-step, when you want the system to decide which tools to use, or when writing all control flow by hand becomes unwieldy.
- Reach for Level 3 only when you can clearly articulate the different roles, responsibilities, and oversight points that demand multiple collaborating agents.

As you move up, each level reuses and extends the previous one. Level 2 still uses RAG and tools; Level 3 still uses strategic agents; all of them still rely on a solid understanding of what the underlying model can and cannot do. The rest of the book will build on this ladder, showing how to implement each level using LangChain and LangGraph, and how to observe and evaluate them using LangSmith so that sophistication does not come at the cost of losing control.

2.4.7 When not to use an agent

A useful rule is to ask whether the ‘how’ is already known. If the workflow is repeatable and you can specify the steps up front, a fixed pipeline (Level 1, or even Level 0 plus deterministic post-processing) is usually more predictable and easier to test. If the workflow must be discovered, revised, or adapted based on intermediate results, then Level 2 planning behavior starts to pay for its risk and cost.

2.5 SOCIO-ECONOMIC AND GOVERNANCE CONTEXT (COSTS, RISK, REGULATION)

When you move from experimenting with a model in a notebook to deploying an LLM-based system in the real world, the problem stops being “can I make this work technically?” and becomes “can I afford this, can I trust it, and will it survive contact with laws, regulators, and users?”. This section builds a mental model for those forces: how money flows through an LLM system, where risk comes from, how regulation is evolving, and what “governance” means in day-to-day engineering terms.

The punchline is simple: cost, risk, and regulation are not things you tack on after you’ve built a clever LangChain graph. They shape the graph you build in the first place.

2.5.1 From model calls to system-level costs

Most providers charge for LLM usage in tokens: small units of text that cover both input and output. Different models have different prices per token, often with separate rates for input, output, and sometimes cached input. On a toy prototype this barely matters; on a production system handling thousands or millions of requests, it becomes a line item in your cloud bill. Direct model usage is only part of the picture:

- You pay for surrounding services: vector databases, blob storage for documents, cache layers, and observability platforms.
- You pay for infrastructure to run your orchestration layer: containers, serverless functions, or long-running processes that host your LangGraph workflows.
- You pay in people: engineers who design chains and graphs, ML practitioners who tune models and prompts, and operations and compliance staff who review logs, investigate incidents, and adjust policies.

There is also an opportunity-cost dimension. If a system responds slowly because it runs a heavy model with long prompts and complex multi-agent flows, that latency can directly affect user satisfaction and throughput. If you “solve” hallucinations by wrapping every answer in three layers of reflection and critique, you improve quality but also burn more tokens, CPU, and time. In practice, you will constantly balance:

- Model choice: smaller/cheaper vs larger/more capable models; a fast “routing” model in front of a slower “expert” model; tiered strategies where simple queries never hit the expensive path
- Context length: how much history and retrieved data to send; long prompts are not free, and compute grows quickly with context length
- Graph complexity: how many steps, retries, reflection loops, and multi-agent exchanges you allow before you call the conversation “done”

Later chapters on evaluation and deployment will revisit these trade-offs with concrete examples, but the important point here is that every architectural decision in LangChain or LangGraph has a cost shadow. Governance is partly about being explicit and honest about that shadow.

2.5.1.1 *Token accounting is your cost meter*

In LLM systems, token usage is not an implementation detail; it is the meter that drives cost and latency. Treat token accounting as a metric you capture per request and per workflow step. This makes cost discussions concrete (which prompt template or retrieval strategy increases tokens, which tool loop is expensive) and it gives you a basis for budgets, rate limits, and regression detection when prompts or models change.

2.5.2 Risk landscape for LLM-based systems

Once an LLM system influences real users or real operations, you inherit a set of risks that are only partly technical. It helps to separate them into a few layers.

2.5.2.1 *Technical and model-behaviour risk*

Models hallucinate; they misinterpret ambiguous instructions; they fail on certain kinds of reasoning. They can produce harmful, biased, or nonsensical outputs while sounding completely confident. These behaviours are intrinsic to how they are trained: they learn statistical patterns in text, not ground truth or logic. Benchmarks highlight this tension: strong performance on knowledge quizzes and writing tasks, weaker and more fragile performance on multi-step maths and strict logical reasoning. If you rely on raw model outputs for anything safety-critical, financially sensitive, or legally binding, you are effectively asking a pattern-matcher to play lawyer, doctor, or risk officer. That is not governance; that is wishful thinking.

2.5.2.2 *Data protection and privacy risk*

Prompts and retrieved context often contain personal and sensitive data: names, contact details, transaction histories, internal documents, logs. If you:

- Stuff entire tickets, emails, or documents into prompts.
- Log all prompts and completions without redaction.
- Or feed production data to external APIs without clear boundaries.

you are taking on privacy and confidentiality risk. Data-protection regimes such as GDPR explicitly require data minimisation, purpose limitation, and storage limitation: collect and process only what you need, for a defined purpose, and keep it only as long as necessary. An LLM stack that casually copies fresh user data into long-term logs and vector stores will quickly drift away from those principles.

2.5.2.3 *Security and abuse risk*

Agents that can call tools—APIs, databases, internal services—extend your attack surface. Typical failure modes include:

- Prompt injection and jailbreaking: users smuggle instructions into inputs that override system prompts (“ignore previous rules and...”).
- Tool misuse: the agent calls a refund API, account-change API, or deployment script in ways you did not anticipate.
- Indirect prompt injection: content retrieved from a database or web page contains adversarial instructions that the agent blindly follows.

Without strict validation and least-privilege access, a compromised or misbehaving agent can do real damage, not just say something embarrassing.

2.5.2.4 Compliance, legal, and reputational risk

Some domains sit under strict sector rules: financial services, healthcare, employment, education, public administration. Even where there is no explicit “AI law” yet existing regimes around consumer protection, advertising standards, suitability of financial products, medical advice, and non-discrimination already apply.

On top of this, a family of AI-specific regulations is emerging. The EU AI Act, for example, classifies systems into unacceptable, high, limited, and minimal-risk categories, with high-risk systems facing obligations around risk management, data quality, transparency, logging, and human oversight. Risk-management frameworks such as the NIST AI Risk Management Framework offer structured guidance on governing, mapping, measuring, and managing AI risk throughout the lifecycle.

You do not need to memorise legal texts, but you do need to internalise the direction of travel: more documentation, more traceability, more emphasis on human oversight for consequential decisions, and less tolerance for “black-box magic”. And then there is reputation. A single screenshot of a system saying something offensive, discriminatory, or dangerously wrong spreads faster than any marketing plan. The direct cost of a refund or a bug fix is often small compared to the long-term cost of losing user trust.

2.5.3 Guardrails and governance-by-design

Governance sounds abstract until you reduce it to concrete architectural decisions. A useful mental model is “guardrails are how you encode your organisation’s appetite for risk into your system design”. In practice, guardrails tend to form a multi-layered defence:

- Input handling: Validate and sanitise user inputs. Block or flag clearly malicious instructions, extremely long payloads, and known attack patterns. Strip or neutralise content that might act as an indirect prompt injection when fed back to the model.
- Behavioural constraints in prompts: Use system-level instructions to define the agent’s role, tone, boundaries, and forbidden behaviours. Make the “mission” explicit: what the system is allowed to do, what it must avoid, when it must ask for help. This does not guarantee compliance, but it shifts the default behaviour in the right direction.
- Tool design and least privilege: Wrap capabilities as small, well-scoped tools rather than one giant “doAnything” function. Give each tool the minimum necessary permissions (read vs write, limited subsets of data, separate credentials). Build simple validation around tool arguments and enforce business rules outside the model.
- Output checks and moderation: Analyse model outputs before they reach users. This can range from simple pattern checks (for disallowed phrases or formats) to separate moderation models that classify content for toxicity, bias, or policy violations. If an output fails checks, you can block it, ask the model to regenerate, or escalate to a human.
- Human-in-the-loop checkpoints: For high-impact actions—significant refunds, contractual changes, irreversible data modifications, major configuration changes—insert explicit approval steps. The agent can prepare drafts, suggestions, or recommended actions; a human decides whether to enact them.
- Observability and traceability: Capture structured traces: which model was used with which parameters, what prompt and context were sent, which tools were called with which arguments, and what outputs came back. This supports debugging, incident response, and

audits. It also enables richer evaluation: you can ask not only “was the answer good?” but “did the agent take a sensible path to get there?”.

From a LangChain/LangGraph perspective, these guardrails manifest as:

- Carefully defined tools and retrievers, rather than letting the model free-form “call code”.
- Explicit graph nodes for validation, moderation, and approval.
- Systematic logging of each step into a tracing backend.
- Clear separation between pure decision logic and side-effectful actions.

Good governance here is just good engineering with higher stakes.

2.5.4 Regulation and evolving AI governance

In the EU, the AI Act (Regulation (EU) 2024/1689) is structured around a risk-based approach, including prohibited practices, requirements for high-risk systems, and transparency obligations for certain AI uses. The European Commission’s public summary presents the risk framing using the familiar ‘unacceptable risk / high risk / limited risk / minimal risk’ terminology, which is a useful mental model, but the legal obligations are defined in the regulation’s text. Regulation is still catching up with generative AI, but some patterns are already clear.

- Horizontal data-protection laws: Frameworks like GDPR impose general duties around data minimisation, purpose limitation, accuracy, and storage limitation. If your prompts, logs, or training pipelines contain personal data, you need to justify why you are processing it, how long you keep it, and how you handle rights such as access and erasure.
- AI-specific risk-based regimes: The EU AI Act is the most visible example of a risk-based AI law. It bans certain uses (such as some forms of social scoring), imposes heavy obligations on high-risk systems (risk management, data governance, documentation, logging, human oversight, robustness testing), and requires transparency for some limited-risk systems (for example, clearly labelling chatbots as non-human).
- Sector-specific rules: Existing regimes in finance, healthcare, employment, and consumer law still apply even if no one mentions “AI” in the statute. If your system influences lending decisions, diagnoses, hiring, or legal advice, you inherit documentation, fairness, and accountability expectations from those sectors.
- Soft frameworks and standards: Non-binding frameworks like the NIST AI Risk Management Framework codify best practice around mapping systems, identifying risks, choosing controls, and continuously monitoring. Many organisations treat these as de-facto standards for internal policy, even before regulations force their hand.

Architecturally, these trends push you towards:

- Clear versioning of models, prompts, tools, and graphs.
- Systematic logging and retention policies.
- Design-time documentation of intended use, limitations, and human-oversight points.
- Mechanisms to roll back or disable components when issues are discovered.

You do not need to turn your codebase into a legal treatise, but you do need to be able to answer basic questions from auditors and regulators: “What does this system do? Where does it get its data? How do you know it behaves as intended? What happens when it fails?”.

2.5.5 Costs, risk, and governance as joint design forces

Cost, risk, and regulation often pull in different directions, and governance is the art of negotiating those tensions in code. A few recurring patterns:

- Spend where it changes the outcome: It is often worth paying more for evaluation, guardrails, and human review on a small subset of high-impact flows, while using cheaper models and simpler flows for the long tail of low-risk queries. Governance helps you identify which is which.
- Prefer simplicity where possible: If a deterministic rule or a small classifier can handle a decision reliably, use it instead of an LLM. Save the model for genuinely ambiguous or open-ended tasks. This reduces cost and makes behaviour easier to reason about.
- Align evaluation with real-world goals: Track metrics that reflect actual risk and value: error rates on critical workflows, frequency of policy violations, latency for typical requests, cost per successful resolution, and so on. Tie your LangSmith-style evaluation datasets and experiments back to these metrics rather than chasing abstract benchmark scores.
- Design for change: Models, prices, and regulations will all evolve. Structures like LCEL pipelines and LangGraph graphs are useful not just for initial design but for controlled change: you can swap models, adjust prompts, insert new guardrail nodes, or alter tool wiring while preserving traceability and evaluation.

The rest of the book will take these ideas for granted. When we talk about guards, retries, evaluation datasets, or stateful graphs, we are not just hunting bugs; we are encoding an organisation's tolerance for cost and risk into a form that can be executed, observed, and improved over time.

2.6 SUMMARY

This chapter reframes “calling an LLM” as building an application around it. A single prompt-response call is useful for exploration, but it is not a system. A production LLM application needs an interface, context construction, model invocation, output interpretation, and the surrounding plumbing that makes the whole thing reliable: state, tools, and observability. The chapter’s core idea is simple: the model is a decision-and-generation component, not an application boundary. If you treat it like the whole program, you end up hiding control flow, state, and validation inside prompts, and you will pay for that later in fragility.

From there, the chapter introduces agentic applications as a change in control flow rather than a new kind of model. In classical software, the developer owns the plan: steps are encoded directly in code paths. In an agent loop, the system owns a goal and repeatedly asks: given what I can currently observe, what is the best next step? That next step might be to call a tool, retrieve data, update state, ask a clarifying question, or stop and respond. This is where environment, tools, memory, and goals become first-class design decisions. You are not “letting the model do whatever it wants”; you are deciding what it can see, what it can do, what it should remember, and how it knows it is finished.

To keep the space navigable, the chapter proposes a level-based ladder (Level 0 to Level 3) that separates minimal reasoning components from data-connected workflows, single-agent systems, and multi-agent collaboration. The point is not that higher levels are better. The point is that each level increases capability and also increases engineering effort, operational complexity, and risk, so you should climb only when you have a concrete reason.

Finally, the chapter zooms out to the constraints that shape real designs: cost (often dominated by tokens and retries), risk (tool misuse, data leakage, prompt injection, and brittle outputs), and governance (guardrails, monitoring, evaluation, and compliance). The rest of the book builds the practical scaffolding that makes these systems survivable in production.

3 THE LANGCHAIN / LANGGRAPH / LANGSMITH ECOSYSTEM

This chapter describes the practical stack used in the rest of the book. Generative models and agent concepts provide context in earlier chapters; here the focus shifts to the libraries you will use to connect prompts, tools, retrieval, workflows, and evaluation in real code. LangChain, LangGraph, and LangSmith are presented as one ecosystem that supports this work in a way that fits normal software engineering practice. LangChain provides the building blocks and execution model for LLM-centric logic, LangGraph expresses that logic as explicit stateful workflows and agents, and LangSmith adds tracing, evaluation, and monitoring so behaviour can be inspected and improved over time.

The first part of the chapter looks at LangChain: why it exists, where it sits in an LLM application stack, and how its abstractions replace ad-hoc notebook code with composable components and LCEL pipelines. Models, prompt templates, loaders, splitters, embeddings, vector stores, retrievers, tools, and output parsers are introduced as standardised interfaces, with an emphasis on keeping applications portable across model providers and vector store backends.

The second part introduces LangGraph and the point at which straight-line chains stop being enough. It explains how graphs, explicit state, supersteps, checkpoints, and interrupts are used to build long-running, multi-step, and human-in-the-loop agents without hiding control flow inside prompts. Patterns such as plan-and-execute loops, multi-agent systems, retries, and fallbacks are expressed in terms of nodes, edges, and shared state rather than implicit model behaviour.

The final part covers LangSmith as the observability and evaluation layer around chains and graphs. Projects, runs, and traces provide a structured view of how applications execute. Datasets and evaluators turn representative interactions into repeatable tests. Monitoring and dashboards surface trends in volume, latency, and token usage, and human feedback and guardrail-oriented metrics connect technical behaviour to policy and governance requirements. Sections 1.4 and 1.5 draw these strands together: they characterise LangChain, LangGraph, and LangSmith as a feedback loop rather than three isolated tools, and they position this stack conceptually alongside other agent frameworks such as CrewAI and Google’s Agentic Development Kit.

3.1 LANGCHAIN: GOALS, ARCHITECTURE, AND ABSTRACTION LAYERS

Modern LLMs are powerful, but on their own they are just APIs: you send strings in, you get strings out. All the “application” parts—connecting to data, orchestrating tools, managing state, handling errors, and making things maintainable—are still your problem. LangChain

exists to shrink that gap. It gives you a set of predictable building blocks and a way to plug them together so that “a couple of ad-hoc API calls in a notebook” can become “a real piece of backend logic that other systems can depend on”.

Where this book talks about LangChain, we are mostly interested in it as the orchestration layer for linear LLM workflows: data in, model calls, optional tools, data out. When those workflows become graph-shaped, long-running, or deeply agentic, LangGraph steps in. When you need to watch and evaluate what is happening, LangSmith completes the picture. But the “wiring” that glues models, prompts, retrieval, and tools together is still LangChain’s home turf.

3.1.1 Why LangChain exists

LangChain was created with a blunt observation in mind: most useful LLM applications need the same ingredients over and over again:

- A way to describe prompts as reusable templates.
- A way to ingest and split data from many sources.
- A way to retrieve relevant pieces of that data for a specific question.
- A way to call one or more models from different providers.
- A way to call external tools (APIs, databases, functions) from inside a workflow.
- A way to combine these pieces into repeatable pipelines.

You can hand-roll all of this with plain Python, HTTP clients, and a tangle of utility functions. It works until it doesn’t: switching model providers becomes painful, adding retrieval requires yet another custom integration, and suddenly every project has its own half-finished mini-framework.

LangChain’s main goal is to give you a standard vocabulary for these recurring tasks, plus implementations you can reuse. It is not trying to hide the LLM or decide everything for you; instead, it turns models, prompts, vector stores, tools, and memory into components with consistent interfaces. Once those components are in place, you can express “how data flows between them” in a single, compact pipeline rather than in hundreds of lines of bespoke plumbing code. A second, equally important goal is portability. Model APIs change, new providers appear, vector databases come and go. If your application logic is written in terms of “this particular SDK’s exotic parameters”, you pay for it every time you migrate. LangChain pushes you toward generic interfaces—“chat model”, “embedding model”, “retriever”, “tool”—so you can change the concrete backend with minimal disruption. Finally, LangChain wants to be production-compatible. It is not just a playground; the same abstractions are intended to support real services. That is why the modern architecture leans heavily on a single execution model (the Runnable / LCEL API) that supports batching, streaming, async execution, and clean integration with tracing and monitoring tools such as LangSmith.

Raw model APIs do not solve several practical constraints that show up immediately in production systems. Context windows are finite, so long documents and long conversations require explicit strategies such as chunking and careful history management. Tool use is possible in modern models, but orchestrating tools across steps (choosing tools, executing them reliably, feeding results back, and handling errors) is still application logic you must build. Finally, multi-step tasks need explicit coordination: without a control-flow layer,

complex workflows become fragile, hard to debug, and difficult to make auditable. These gaps are the space LangChain is designed to fill with standard components and a composable execution model.

3.1.2 Where LangChain fits in the stack

If you strip an LLM-powered system down to its layers, you usually end up with something like this:

- A client or caller: web frontend, mobile app, backend service, cron job.
- An “LLM application” layer: prompts, retrieval, model calls, tool invocations, post-processing.
- The external world: model providers, databases, vector stores, search engines, internal microservices.

LangChain is almost entirely about that middle layer. It does not care whether the caller is a FastAPI endpoint or a batch job. It does not host models for you. It does not replace your databases or your services. Instead, it focuses on the logic that takes:

1. Inputs from the outside world (user messages, IDs, filters, raw documents).
2. Transforms and enriches them (retrieve documents, call tools, build prompts).
3. Calls models and parses their outputs.
4. Returns something structured and predictable to the rest of your system.

You can think of LangChain as the “integration tissue” around LLMs:

- Upstream, it exposes simple entry points (a method you can call from your server).
- Downstream, it speaks to many different providers (LLM APIs, vector stores, HTTP services) through standardised components.
- Internally, it turns the movement of data between those components into explicit, testable workflows.

Later in the book we will wrap these workflows into HTTP APIs, background jobs, or event handlers. For now, it is enough to see LangChain as the in-process engine that runs your model-centric logic.

3.1.3 Architectural principles

LangChain’s architecture is built around a small set of design choices that show up everywhere in the library:

- Modularity: Everything is a component with a focused responsibility: a prompt template formats text, a chat model generates responses, a retriever fetches relevant documents, a tool knows how to call a particular API. You compose behaviour by combining components rather than subclassing some giant “God object”.
- Composition over configuration: Instead of burying flows in nested configuration files, LangChain encourages you to build pipelines in code. In the modern API, this happens through the LangChain Expression Language (LCEL): a “Runnable” interface plus operators and helpers for wiring components together. The dataflow lives in actual code, which you can unit-test, refactor, and version-control like any other business logic.

- Provider-agnostic interfaces: For each role—LLM, chat model, embeddings, vector store, retriever, tool—LangChain defines a standard interface. Concrete integrations (for example, a specific cloud model or vector database) plug into these interfaces. Changing provider usually means changing one line of wiring code or configuration, not rewriting your entire chain.
- Explicit dataflow: The framework pushes you to make the path data takes through your system explicit: “input → prompt → model → parser” instead of “some helper somewhere builds a string and sends it”. This is important for debugging, evaluation, and compliance. When you look at a chain, you should be able to tell, step by step, what happens to the input and where external calls occur.
- Execution-aware design: The Runnable / LCEL layer supports synchronous calls, asynchronous calls, parallel mapping over lists, and token streaming. You can build a pipeline once, then invoke it in different ways depending on whether you are serving an HTTP request, processing a batch offline, or streaming output to an interactive client.

These principles make LangChain behave more like a normal software library than a black-box AI product: the code is explicit, tests are possible, and you retain control over how data moves.

3.1.4 The abstraction layers: from components to chains

There are many classes and modules in LangChain, but you can keep your mental model simple by thinking in three main layers.

3.1.4.1 Components: the basic building blocks

At the bottom are the atomic pieces that do one thing well:

- Models – chat models and LLMs: wrappers around underlying APIs that let you say, “given these messages, produce the next message”, with a unified interface across providers.
- Prompt templates – parametrised message templates that combine system instructions, examples, and user inputs into a structured prompt.
- Document loaders and text splitters – utilities that ingest content (files, web pages, database rows) and break it into chunks that fit within a model’s context window.
- Embeddings, vector stores, and retrievers – components that turn text into vectors, store those vectors, and retrieve the most relevant chunks for a query; the backbone of retrieval-augmented generation.
- Tools – wrappers around external capabilities such as HTTP APIs, database queries, or arbitrary Python functions, so they can be invoked from within a pipeline or by agents.
- Output parsers – small components that turn raw model output (usually text) into structured data: JSON objects, Pydantic models, enums, or custom Python types.

Each of these has a clear API. For example, a retriever exposes a “given a query, return documents” method regardless of whether it is backed by a cloud vector database or an in-memory index.

3.1.4.2 Runnables and the LangChain Expression Language (LCEL)

Above the components sits the execution model: the Runnable interface and the LangChain Expression Language. A “Runnable” is anything that can be invoked with an input to produce an output. LCEL gives you the tools to compose runnables into larger graphs:

- The pipe operator (`|`) for simple “do A, then B” flows.
- Mapping helpers to apply a pipeline to each element of a list.
- Branching and routing primitives that choose different sub-pipelines based on the input.
- Common methods like `invoke`, `batch`, and `stream` that work consistently across all pipelines.

A minimal LCEL pipeline looks like this:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI

prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a concise, factual assistant."),
        ("human", "{question}"),
    ]
)

model = ChatOpenAI(model = "gpt-4.1-mini")
parser = StrOutputParser()

chain = prompt | model | parser
answer = chain.invoke({"question": "What is a transformer in machine learning?"})
print(answer)
```

Here, `prompt`, `model`, and `parser` are each runnables. The `|` operator builds a new runnable `chain` that wires them together. From the caller’s point of view, `chain` is a single unit: it takes a structured input and returns a structured output.

3.1.4.3 Chains: reusable multi-step workflows

The word “chain” is used in two related ways:

- Conceptually, a chain is any reusable multi-step workflow that takes an input, runs it through several components, and produces an output.
- In older versions of LangChain, chains were concrete classes; in modern LangChain, many of those are being re-expressed in terms of LCEL pipelines, but the idea remains the same.

Examples of chains include:

- A question-answering chain that retrieves relevant documents and generates an answer grounded in them.
- A summarisation chain that takes a long text, chunks it, summarises each chunk, and then combines the partial summaries.
- An extraction chain that turns free-form text into a JSON structure using an output parser.

In practice, you will often build your own chains by composing components directly with LCEL, and you may also use some pre-built chains for common patterns. Either way, the chain is the unit you usually plug into the rest of your system.

3.1.5 How this book will use LangChain

For the rest of the book, you can think of LangChain as your standard library for LLM applications:

- When you need to talk to a model, you reach for a chat model component.
- When you need to put together prompts, you use prompt templates.
- When you need to bring in external data, you use document loaders, text splitters, embeddings, vector stores, and retrievers.
- When you need to call your own systems, you wrap them as tools.
- When you need to connect these pieces, you build LCEL pipelines and chains.

LangGraph will then give those chains a richer “skeleton” when you need branching, loops, state, or multiple interacting agents. LangSmith will give you the eyes and ears to see how everything behaves in practice. But all three share the same conceptual core: the components and abstractions introduced in this section. Once you are comfortable thinking in terms of “components plus LCEL pipelines”, you can read the rest of the book as a series of variations on that theme: different ways to ingest data, different retrieval strategies, different tool sets, and different graph topologies, all built from the same underlying parts.

3.1.6 Lang* ecosystem boundary

Recent versions of the ecosystem draw a clearer boundary between “composition of model-centric steps” and “stateful, long-running agent execution”. In this split, LangChain focuses on model integration and composable workflows, while LangGraph is the recommended place for stateful agents and persistence. In particular, memory mechanisms that previously lived in the base LangChain library have been deprecated (but we’ll talk about them later) in favour of LangGraph for persistence, and LangGraph is presented as the preferred approach for building agents in this versioned stack. LangSmith then complements both by providing tracing, evaluation, and monitoring so that behaviour is observable and testable over time.

3.2 LANGGRAPH: GRAPHS, STATEFUL AGENTS, AND LONG-RUNNING WORKFLOWS

LangChain gives you good “nouns and verbs” for LLM applications: models, prompts, tools, retrievers, evaluators. LangGraph adds the grammar. It is the part of the stack that lets you say, in code, “first do this, then depending on the result do that, maybe loop a few times, wait for a human here, and resume tomorrow from exactly where we stopped.” Instead of seeing your application as a single call chain that starts at a prompt and ends at a completion, LangGraph encourages you to see it as a graph: a network of steps connected by explicit control flow, with a shared state object flowing through the whole thing. That shift is what turns “a clever chain” into “an agent” that can remember, adapt, and run as a long-lived service rather than a disposable function call. You will use LangGraph throughout the book whenever you need one or more of these properties:

- The system must keep track of state across multiple turns or sessions.
- The path from input to output depends on intermediate results, not just the initial prompt.
- Parts of the workflow need to loop, branch, or run in parallel.
- Some steps require human review or external events before continuing.
- You want to inspect, debug, and replay the behaviour of your agents over time.

This section builds the mental model you will rely on later when you start wiring concrete graphs for your own applications.

3.2.1 From chains to graphs

A LangChain chain is essentially a straight line:

- You compose components with the LCEL operator (`|`).
- Data flows from left to right.
- Each run of the chain is independent: you pass some input, you get some output, and any state you keep lives outside the chain itself.

That is perfect for many tasks: a simple RAG pipeline, a one-off summarisation job, a classifier that tags support tickets, an extractor that pulls entities out of logs. A LangGraph graph, by contrast, is built out of three ideas:

- Nodes: individual steps in your workflow. A node is usually a Python function or an LCEL chain: call a model, call a tool, update memory, perform routing, aggregate results.
- Edges: the connections between nodes. Edges tell LangGraph what can run after what. They can be fixed (“after `analyze` always go to `summarize`”) or conditional (“after `route` look at the current state and decide whether to go to `billing`, `tech_support`, or `END`”).
- State: a structured object, typically defined as a `TypedDict`, `dataclass`, or similar, that holds everything the graph needs to remember: messages, tool results, flags, partial plans, user identifiers, and so on.

Each node sees the current state, returns a partial update to that state, and LangGraph merges those updates into the shared state before moving on. Control flow is not hard-coded inside nodes; it is expressed in the edges of the graph.

Conceptually:

- A chain says, “always do A → B → C in this order.”
- A graph says, “start at A; depending on what happens, you may go to B or C; you might loop back; and you decide to stop when some condition in the state is met.”

Once you start thinking in graphs, many “agent behaviours” suddenly become simple patterns: loops, branches, retries, and hand-offs stop being clever prompt tricks and become explicit parts of your workflow.

3.2.2 StateGraph and explicit state

The main programming model in LangGraph is the StateGraph. To define a graph you do three things:

1. Define the state schema: You declare a type that lists the keys your workflow will manage. For example, a simple conversational graph might have:

- messages: the chat history.
- context: retrieved documents or tool outputs.
- status: a simple flag for where the workflow is (e.g. ‘collecting_info’, ‘awaiting_approval’, ‘completed’).

LangGraph treats this schema as the contract between nodes.

2. Register nodes: Each node is a function that takes (a view of) the current state and returns a dictionary of updates:

- {"messages": [...new messages...]} to append to the conversation.
- {"context": [...new context items...]} to add retrieved information.
- {"status": "awaiting_approval"} to signal a state transition.

Nodes do not directly mutate global objects; they propose updates, which LangGraph applies. That keeps side effects visible and makes debugging easier.

3. Connect nodes with edges: You specify which node runs first, which nodes follow, and how decisions are made. Conditional edges are plain Python functions that look at the state and return the name of the next node (or a special ‘END’ marker).

```
# 1) State schema (the contract between nodes)
class ChatState(TypedDict):
    messages: Annotated[list[AnyMessage], add_messages]
    context: Annotated[list[str], add]
    status: Literal["collecting_info", "awaiting_approval", "completed"]

# 2) Nodes (pure functions: state in -> updates out)
def retrieve_context(state: ChatState) -> dict:
    # toy example: pretend we retrieved something relevant
    return {
        "context": [f"retrieved_snippet_for: {state['messages'][-1].content}"],
        "status": "collecting_info"
    }

def ask_for_approval(state: ChatState) -> dict:
    # toy example: we decide we need approval before completing
    return {"status": "awaiting_approval"}

def finalize(state: ChatState) -> dict:
    return {"status": "completed"}

# 3) Edges (including a conditional router)
def route_next(state: ChatState) -> Literal["ask_for_approval", "finalize"]:
    # simple decision based on state
    return "ask_for_approval" if state["status"] != "awaiting_approval" else "finalize"

builder = StateGraph(ChatState)
```

```

builder.add_node("retrieve_context", retrieve_context)
builder.add_node("ask_for_approval", ask_for_approval)
builder.add_node("finalize", finalize)

builder.add_edge(START, "retrieve_context")

# conditional edges: a Python function decides which node runs next
builder.add_conditional_edges("retrieve_context", route_next)
builder.add_edge("ask_for_approval", "finalize")
builder.add_edge("finalize", END)

graph = builder.compile()

```

When you compile a StateGraph, you get back a runnable object: something you can invoke with an initial state dictionary and an optional configuration. The graph runs until no more nodes are scheduled or an `END` is reached, and it returns the final state.

LangGraph runs your workflow in small, clearly defined “ticks” called supersteps. In each superstep, the engine looks at the graph and the current state, finds all nodes that are ready to run (their incoming edges are satisfied, and any conditions are met), runs those nodes, collects their state updates, and then merges those updates into a new global state. Only after that merge does it move on to the next superstep and re-evaluate which nodes are now ready. You can think of a superstep as “one round of work across the graph”:

- Superstep 1: maybe only the “ingest user message” node can run.
- Superstep 2: based on that state, both “retrieve documents” and “call LLM” become ready, so they run in parallel.
- Superstep 3: once their results are merged into state, an “analyse result” node becomes ready and runs.

This round-based model gives you a predictable form of parallelism: all nodes that can run in a given round do so, but state only changes in between rounds, when LangGraph merges the updates. That makes it easier to reason about workflows with fan-out (multiple branches) and fan-in (merging results) without worrying about fine-grained race conditions.

This explicit state-plus-graph model has a few consequences that are worth keeping in mind: You can reason about the workflow as a state machine: each node is a transition function that updates part of the state. Tests can focus on “given this state, what does this node do?” rather than replaying entire conversations. You can add or remove nodes, split big steps into smaller ones, or insert new branches without having to rewrite all your prompts and tools.

3.2.3 Checkpoints, threads, and long-running workflows

If state is the backbone of a graph, checkpoints are its memory. A checkpoint is a snapshot of the entire workflow at a particular moment:

- The current state object.
- Metadata about which nodes are scheduled next.
- Any information needed to resume execution without starting over.

When you compile a graph with a checkpoint, LangGraph automatically saves a checkpoint after each superstep. Those checkpoints are grouped under a thread identifier: a stable id that represents one conversation or workflow instance. This gives you several capabilities out of the box:

- Session continuity: Calling the same graph again with the same thread id reuses the previous state. The graph picks up where it left off, instead of forgetting the past.

- Time travel: You can list checkpoints for a thread and restore from any of them. That is useful for debugging (“what did the state look like before this bad decision?”) and for running experiments (“what if we tried a different path from this point?”).
- Fault tolerance: If a process crashes or a deployment restarts; you can resume workflows from their last checkpoint instead of dropping them.

In development, an in-memory checkpointer is enough to show the pattern. In production, checkpoints are typically stored in more durable backends (for example, databases or other persistent stores) so that workflows can span long periods of time. LangGraph also provides convenience abstractions like a message-centric state graph, where state is essentially “a list of messages plus a few extra fields.” This is the natural shape for chat-style agents and integrates smoothly with the message types used elsewhere in the stack. The important point is that “memory” in LangGraph is not a side channel or a hidden feature of the model. It is first-class: encoded in the state schema, persisted as checkpoints, and explicitly passed between nodes.

Two operational details are easy to miss. First, a checkpoint is not the same thing as chat history: checkpoints are workflow snapshots that include the full state plus execution metadata (for example, planned nodes and failed tasks), so you can restore a workflow at an arbitrary point and resume execution. Chat history storage is narrower: it preserves messages, but not necessarily the workflow’s planned execution state. Second, thread/session identifiers are part of your security surface. If identifiers are guessable (for example, sequential ids), users may be able to access sessions that do not belong to them; using UUID-style identifiers and applying runtime permission checks are cited as practical mitigations.

3.2.4 Interrupts and human-in-the-loop patterns

Not every decision should be automated. Some steps are naturally human: approving a refund, confirming a high-value action, giving final sign-off on a report, or clarifying ambiguous requirements. LangGraph treats these moments as interrupts. From inside a node, you can signal that the graph should pause and hand control back to the caller, along with enough state to show what is going on and what information is missing. The typical pattern looks like this:

1. A node reaches a point where it needs human input (for example, there is no clear answer in the data, or the action is high-risk).
2. Instead of continuing, it triggers an interrupt and returns a structured payload (for example, a drafted message and a set of options for the user to choose from).
3. LangGraph saves a checkpoint and stops execution.
4. The calling system (backend service, UI, or another process) shows the payload to a human and waits for a response.
5. When the response arrives, the caller resumes the graph by sending a command together with the new data, using the same thread id. The graph continues from the paused node, now with the extra information in its state.

Because interrupts are integrated with checkpoints, long pauses are normal: the graph can sleep for minutes, hours, or days without losing context. That is the mechanics behind human-

in-the-loop agents, approval queues, and workflows that depend on external events rather than just user messages.

3.2.5 Advanced patterns: multi-agent systems and iterative loops

Once you have nodes, edges, state, checkpoints, and interrupts, a lot of “fancy” agent patterns are really just graph shapes:

- Multi-agent systems: Instead of trying to make one giant prompt handle everything, you can model several specialised agents as separate nodes or subgraphs: a planner, a researcher, a critic, an executor, and so on. A supervisor node inspects the current state and decides which specialist to call next. All of them read and update the same shared state, so they can cooperate without sharing hidden globals.
- Plan-and-execute loops: One node builds a plan (a list of steps in the state), another executes the next step (calling tools, retrieving data, asking the model to act), and a third evaluates the result. Depending on the outcome, the graph either loops back to execute the next step, revises the plan, or stops. This transforms “call the model and hope it does the right thing” into “let the model work within a controlled loop with explicit stopping conditions.”
- Reflection and self-critique: A node generates a draft answer, another node critiques it against explicit criteria (accuracy, completeness, tone), and a third revises the draft based on the critique. The resulting loop usually runs only a small number of times but can significantly improve quality for tasks where errors are costly.
- Fallbacks and retries: If a node fails (a tool times out, a model output cannot be parsed), edges can route to alternative nodes: a simpler backup model, a different tool, or a node that asks the user for clarification. With state available, you can track how many times you have tried, which paths you have already explored, and when to give up gracefully.

LangGraph does not hard-code any of these strategies. It just gives you the primitives to express them cleanly: cycles in the graph, conditional edges, shared state, and a way to pause and resume.

3.2.6 Development, observability, and deployment

A graph is only as useful as your ability to see what it is doing. LangGraph is designed to be inspectable. During development you can run graphs in a local server, send test requests, and watch how state evolves over time. Execution traces show you which nodes ran, in what order, with what inputs and outputs. That makes it much easier to answer, “why did the agent do this?” when something looks odd. Streaming support lets you surface partial model outputs to the UI while the graph continues running other steps in parallel.

For deployment, graphs can be exposed as APIs that accept requests, run the underlying workflows (including any long-running or human-in-the-loop parts), and return results or streaming updates. Because state and checkpoints are first-class, scaling horizontally is natural: multiple workers can handle different threads, all backed by the same persistent checkpoint store. LangGraph also integrates with higher-level tooling for visualisation and monitoring. You can view the graph structure, inspect individual runs, and track operational metrics such as latency, token usage, and error rates. That observability is essential when you start evolving graphs over time: adding new nodes, changing prompts, upgrading models, or

tightening guardrails. It gives you the feedback loop you need to improve behaviour without flying blind.

3.2.7 When to reach for LangGraph

Not every LLM application needs a graph. Many useful systems can be built with a single LCEL chain and some light orchestration around it. As a rule of thumb:

- If the task can be described as “take this input, maybe fetch some documents, call a model once, and return an answer,” a plain chain is usually the simplest and most robust option.
- If you need to maintain state, try multiple strategies, involve tools repeatedly, or pause for humans or external events, you are in LangGraph territory.

The rest of the book will treat LangGraph as the default way to express agents: not a magic abstraction, but a practical way to turn a pile of prompts, tools, and models into a durable, inspectable workflow. Once you are comfortable thinking in graphs, building and evolving complex agents becomes a matter of rearranging and extending those patterns instead of starting from scratch each time.

3.3 LANGSMITH: TRACING, DATASETS, EVALUATION, MONITORING

LangChain and LangGraph give you the building blocks to construct LLM applications: prompts, tools, chains, and graphs. LangSmith is the part of the stack that lets you see what those applications are actually doing, measure whether they are doing it well, and improve them over time. Without it (or an equivalent observability layer), you are essentially flying blind: you can ship a clever agent, but you cannot easily explain why it behaved in a certain way, how quality changes when you tweak a prompt, or where your latency and token budget are going. At a high level, LangSmith does four things:

- Captures detailed traces of your chains, agents, and graphs.
- Organises examples into datasets you can reuse as benchmarks.
- Runs evaluations (code-based, LLM-based, and human) over those datasets and over live traffic.
- Provides monitoring dashboards so you can watch volume, latency, and cost over time and detect regressions.

You can treat it as the “flight recorder plus test harness” for your LLM systems: every interaction can become a trace; traces can be turned into datasets; datasets can power evaluations and experiments; and the results show up in dashboards that close the loop from development to production and back again.

3.3.1 Core concepts: projects, runs, and traces

LangSmith organises everything around three primitives: projects, runs, and traces.

- A project is a logical container, typically mapped to one application or experiment line (for example, “support-assistant-v1” or “agent-rewrite-experiments”).
- A trace is the tree (or DAG) formed by a root run and all its children. When a chain calls a model which then calls tools and sub-chains, each of those steps is its own run, and together they form one trace.
- A run is a single execution of something: a model call, a tool call, a chain, or a graph node. Each run records inputs, outputs, start/end timestamps, token usage, and metadata.

This hierarchy mirrors how you think about your system:

- The root run corresponds to “handle one user request”.
- Child runs show internal behaviour: retrieval steps, tool calls, sub-graphs, and so on.

Once tracing is enabled, every invocation of your LangChain chains or LangGraph workflows can automatically emit this structure. You can then inspect traces to answer questions such as:

- What prompt did the model actually see?
- Which tools were called, in what order, with which arguments, and how long did they take?
- Where did an error occur?
- How many tokens did this interaction consume, and which step was the main contributor?

3.3.2 Tracing and debugging

Tracing is the first reason to wire LangSmith into your stack. Instead of logging a single “input → output” pair, you get a full picture of the internal decision process. On the LangChain side, enabling tracing is usually a matter of configuration:

- You provide an API key for LangSmith.
- You set a project name, so traces are grouped sensibly.
- You enable the tracing flag, so LangChain components automatically send run data.

Once this is in place, every call to a traced chain or graph produces:

- A structured tree of runs (root plus children).
- Inputs and outputs for each run, including prompts and tool results.
- Timing information (latency per step, total latency).
- Token statistics per model call, which you can treat as a proxy for cost.
- Error information if tools throw exceptions or LLM calls fail.

You can filter and search traces by tags, time range, status, or custom metadata. That makes debugging much closer to debugging normal code: you can find “all failed runs using model X”, or “all runs where this tool exceeded 2 seconds latency” and inspect them visually. A minimal example in Python looks like this (LangSmith environment configuration is done outside the code, for example via environment variables or a configuration file):

```
from langchain_openai import ChatOpenAI
from langchain_core.runnables import RunnableLambda

# Simple LangChain component
model = ChatOpenAI(model = "gpt-4.1-mini")

def classify_intent(message: str) -> str:
    prompt = f"Classify the user's intent in a single word: {message}"
    response = model.invoke(prompt)
    return response.content

intent_chain = RunnableLambda(classify_intent)
result = intent_chain.invoke("I cannot log into my account.")
print("Predicted intent:", result)
```

With LangSmith tracing enabled at configuration time, this single call produces a trace with:

- A root run for `intent_chain`.
- A child run for the `ChatOpenAI` model call.

In the LangSmith UI you can open the trace, view the constructed prompt, see how long the call took, and inspect the returned content. If you later wrap this chain inside a larger graph with tools and memory, the same tracing infrastructure scales up automatically. LangSmith is not limited to LangChain-specific components. You can instrument arbitrary Python code, HTTP endpoints, or even other frameworks by using the LangSmith client directly or by integrating with OpenTelemetry, so the same tracing model covers your whole LLM surface, not just the parts that use LangChain abstractions.

3.3.3 Datasets: turning interactions into tests

Traces help you understand single runs: “what happened in this conversation or workflow?”. Datasets help you answer a different question: “how does my system behave on a set of important examples, over and over again?”.

In LangSmith, a dataset is just a named collection of examples that you want to reuse as tests. Each example usually contains three parts:

- Inputs: the data you would normally send into your application.
- Reference outputs (optional): what a “good” answer should look like for this input. For a classifier this might be a label; for a generation task it might be an ideal answer.
- Metadata (optional): extra information that helps you organise and filter the dataset: category, difficulty, use case, and so on.

You can think of a dataset as a test file for your LLM system. Instead of unit tests written in code, you have real or realistic examples written as input–output pairs. There are three common ways to build these datasets:

1. Manually curated examples: You or your team write a small set of examples by hand. These usually cover:
 - The most frequent user questions.
 - Important edge cases.
 - Scenarios where you know mistakes are particularly costly.
2. Historical traces: You take real interactions from production (often conversations where things went wrong), clean them up, and save them as labelled examples. This is one of the fastest ways to get realistic test data.
3. Synthetic expansion: You start with a small seed of high-quality examples and then generate variants (for example “same intent, different wording”). This helps you cover more phrasing and corner cases without writing everything by hand.

LangSmith lets you create and update datasets either through the LangSmith UI or from code. Every change creates a new dataset’s version, so you always know exactly which set of examples you used for a particular experiment. You can tag dataset versions (for example, ‘baseline’ or ‘production_2025_01’) and refer to those tags from your evaluation scripts or CI pipelines. A simple pattern in code looks like this:

```
from langsmith import Client
client = Client()

# Create (or fetch) a dataset
dataset = client.create_dataset(
    name = "support-intent-eval",
    description = "Intent classification examples for a support assistant"
)

# Add an example to the dataset
client.create_example(
    dataset_id = dataset.id,
    inputs = {"message": "I was charged twice for the same invoice."},
    outputs = {"intent": "billing_issue"},
    metadata = {"category": "billing", "difficulty": "medium"}
)
```

After you have a dataset like this, you can treat it as a reusable test suite. Whenever you change a prompt, swap a model, or update a graph, you can run the new version against the same dataset and see how behaviour changed: did accuracy go up or down, did latency increase, did you break some critical edge case?

3.3.4 Evaluation: from examples to metrics

Datasets give you examples. Evaluation turns those examples into numbers and insights. LangSmith organises evaluation around two main ideas:

- Evaluators: functions that score how well a run behaved.
- Experiments: runs of your application on a dataset, together with the evaluator scores.

You can think of an evaluator as “a small robot reviewer” that looks at inputs, outputs, and (optionally) reference answers, then returns a judgment. There are several kinds of evaluators:

1. Code evaluators: These are normal Python (or TypeScript) functions. They are fully deterministic and run without any LLM calls. Typical uses include:

- “Does this label match the expected label?”
- “Is this JSON valid and does it contain the required keys?”
- “Does the generated code compile or pass a unit test?”

2. LLM-as-judge evaluators: Here, another model plays the role of reviewer. It reads the input, the system’s output, and optionally a reference answer, then scores them based on instructions in a prompt. Common patterns are:

- Reference-based: “How close is this answer to the reference answer?”
- Reference-free: “Rate helpfulness and clarity on a 1–5 scale.”

3. Human evaluators: Human reviewers look at runs in the LangSmith UI, give scores, labels, or comments, and sometimes correct answers. This is often used for:

- High-risk scenarios.
- New features where you don’t yet trust automatic metrics.

4. Pairwise evaluators: Instead of scoring one output, these compare two outputs (for example, before vs after a prompt change) and decide which one is better. Both code and LLM-based evaluators can be used in this way.

Evaluators can work in two contexts:

- Offline evaluation: You run your application over a dataset of examples. For each example you get:
 - The run (inputs + outputs + trace).
 - One or more evaluator scores.

This is useful for CI, regression tests, and comparing variants before deployment.

- Online evaluation: Evaluators run on live traffic: real runs and conversations.
- In many cases you do not have a reference answer in production, so you rely more on LLM-as-judge and human feedback (for example, safety checks, tone, or satisfaction).

In both cases, evaluator results are stored as structured feedback: usually a dictionary with fields like:

- key: the name of the metric (for example "accuracy", "clarity", "policy_compliance").
- score or value: a number or label.
- comment: optional free-text explanation.

A typical offline evaluation loop in code looks like this:

```
from langsmith import Client, evaluate
client = Client()
```

```

dataset_name = "support-intent-eval-" + str(uuid.uuid4().hex[:8])
dataset = client.create_dataset(dataset_name)

client.create_example(
    dataset_id = dataset.id,
    inputs = {"message": "I want a refund for my last order"},
    outputs = {"intent": "refund_request"}
)
client.create_example(
    dataset_id = dataset.id,
    inputs = {"message": "Where is my package? Tracking says delayed."},
    outputs = {"intent": "order_status"}
)

def classify_intent_app(inputs: dict) -> dict:
    # Wrap your chain or graph invocation here
    # Example: call a LangChain Runnable that predicts an intent
    intent = intent_chain.invoke(inputs["message"])
    return {"intent": intent}

def correct_intent(outputs: dict, reference_outputs: dict) -> bool:
    # Simple exact-match evaluator
    return outputs["intent"] == reference_outputs["intent"]

results = evaluate(
    classify_intent_app,
    data = client.list_examples(dataset_name = "support-intent-eval"),
    evaluators = [correct_intent],
    experiment_prefix = "intent-baseline",
)

```

The evaluate call here:

- Runs your `classify_intent_app` over every example in the `support-intent-eval` dataset.
- Applies the `correct_intent` evaluator to each result.
- Stores an “experiment” in LangSmith, so you can see overall accuracy, inspect individual failures, and compare this experiment with future ones.

An experiment is a named evaluation run recorded by LangSmith. It links (a) the dataset version you evaluated, (b) the application runs produced for each example (including traces), and (c) the evaluator feedback attached to those runs. Experiments are the unit you compare when you change a prompt, model, or graph.” This matches how `evaluate` produces `ExperimentResults` / `experiment` projects and how performance metrics are fetched per experiment.

You can rerun the same dataset with a new model, a new prompt, or a new graph structure and compare experiments side by side. Online evaluation uses the same evaluator functions but attaches them to live runs instead of dataset examples. For instance, you could plug in a safety evaluator that continuously scores live outputs for policy violations, even if you don’t have reference answers for those conversations.

3.3.5 Monitoring and dashboards

Evaluation answers “how good is the system on these examples?”.

Monitoring answers “what is the system doing right now, and how is that changing over time?”.

LangSmith provides dashboards that aggregate information from traces, evaluations, and metrics across projects and time windows. Instead of looking at a single run, you look at trends. Typical metric groups include:

- Volume
 - Number of traces over time.

- Number of LLM calls.
- Success or error rates.
- Latency
 - End-to-end time per trace.
 - Average time per LLM call.
 - Number of LLM calls per trace (which shows how “chatty” an agent is).
- Token and cost proxies
 - Total tokens used in a given period.
 - Tokens per trace.
 - Tokens per LLM call.

Since most providers bill by tokens, these are good approximations of cost.
- Streaming behaviour (if you stream responses)
 - Percentage of traces that use streaming.
 - Time to first token (how quickly users see something on screen).

Watching these metrics helps you spot issues that are invisible at the single-trace level:

- A new prompt that silently doubles latency.
- A change in graph structure that triples the number of LLM calls per interaction.
- A specific tool that fails often or dominates total response time.
- Traffic spikes that stress your infrastructure.

Monitoring and evaluation reinforce each other in both directions. When offline evaluation on a dataset shows an improvement, dashboards let you check whether the same gain appears in real production traffic. Conversely, when monitoring reveals an anomaly such as a sudden spike in errors or a drop in a key score, you can take the affected traces, add them as new examples to a dataset, and investigate the problem offline in a controlled way.

The result is a feedback loop: design → deploy → monitor → capture interesting cases → turn them into datasets → evaluate → improve → deploy again.

3.3.6 Human feedback, guardrails, and governance

LangSmith is not just a place for numbers. It is also where human judgment and policy constraints meet your technical system in a way you can inspect and reuse. On the human side, LangSmith lets reviewers attach structured feedback to runs and traces as feedback tags. Those tags can be categorical (for example “acceptable / unacceptable”) or numeric (for example a 1–5 score), and they can be applied not only to the final output but also to intermediate steps in a trace when that is what you want to assess. Reviewers can also add free-text comments to capture context and rationale (“good answer but too long”, “missed the policy detail”, “tool choice was wrong”). For systematic review, annotation queues let you route selected runs to human reviewers with a rubric and predefined feedback keys, and optionally export reviewed items into datasets for offline evaluation.

That feedback stays valuable long after it is collected. You can turn reviewed runs into datasets that focus on hard cases and failure modes, then rerun your application on those datasets as part of offline evaluation and regression testing. As feedback accumulates, you can also aggregate feedback tags and scores over time to spot trends: which scenarios consistently score

poorly, which failure modes recur, and whether changes in prompts, models, tools, or graph structure improve or degrade behaviour.

Guardrails are the other half of governance. Many of the rules you care about—safety, policy, compliance—can be treated as evaluation problems with measurable outcomes. You can log when input filters, output filters, or moderation layers trigger; attach safety-focused evaluators to runs (for example toxicity checks, PII detection, or policy-violation checks); and measure both sides of the trade-off: how often the system violates a rule (missed violations) and how often it blocks harmless content (false positives). Instead of asserting “we have responsible AI,” you end up with concrete artefacts: datasets that capture sensitive scenarios, metrics that show violation rates over time, and experiments that reveal whether a change made behaviour better or worse. This is what governance looks like in practice: a repeatable set of tests and scores, plus trace and experiment views that keep you honest as the system evolves.

3.3.7 LangSmith in the development lifecycle

Current practice converge on a simple lifecycle in which LangSmith is present from the beginning rather than added at the end:

1. Prototype with tracing switched on. Use traces to debug prompts, tool usage, and graph structure.
2. Promote interesting interactions into datasets, especially failures and edge cases. Add reference answers where appropriate.
3. Define evaluators that reflect application goals: correctness, helpfulness, safety, cost, latency.
4. Run offline evaluations whenever you change prompts, retrieval strategies, or models, and compare experiments on your key datasets.
5. Deploy with monitoring turned on for a representative share of traffic. Watch dashboards and online evaluation scores for anomalies.
6. Feed real-world issues back into datasets, update prompts and architectures, and repeat.

In this book, LangSmith will appear whenever we go beyond “does the code run?” and start asking “is this behaviour good, and how do we know?”. We will use it mainly to capture traces, turn tricky real-world interactions into datasets, and run evaluations whenever we change prompts, retrieval strategies, tools, or models. That way, quality becomes something you measure and iterate on, not something you guess from a few ad-hoc tests.

3.4 THE LANGCHAIN / LANGGRAPH / LANGSMITH FEEDBACK LOOP

Once you have the basic pieces in place, the most useful way to think about the LangChain stack is as a feedback loop rather than three separate libraries. You design behaviour with LangChain, you orchestrate and persist it with LangGraph, and you observe and refine it with LangSmith. The value comes from running that loop repeatedly, not from any single component on its own.

At the design layer you work in LangChain. You choose models, define prompt templates, wrap external systems as tools, configure retrievers and memory components, and compose them into runnables and chains. At this point you are still thinking “single request”: given this input and this configuration, what should the model do, which tools might it call, and what shape should the output have?

As soon as the behaviour has more than a couple of steps, you drop it into LangGraph. The same LangChain components become nodes in a StateGraph: a state object carries inputs, intermediate results, and control flags; edges determine how you move between nodes; checkpoints preserve progress so you can resume long-running or human-in-the-loop flows. You have not changed what each building block does, only how they are stitched together and how the state around them is managed.

From the first serious prototype, you connect the graph to LangSmith. Instead of logging ad hoc strings, you send runs to a project: each execution records which chain or graph ran, which model and tools were used, what inputs and outputs were produced, and how state evolved across steps. You then layer datasets and evaluations on top of those traces: hand-picked examples, synthetic edge cases, and live traffic samples that you can replay whenever you change a prompt, a retriever, or a graph topology.

3.5 CONCEPTUAL COMPARISON WITH OTHER CANVASES (CREWAI, GOOGLE ADK)

Once you see LangChain, LangGraph, and LangSmith as a single “agent engineering stack”, it is natural to ask how they relate to the rest of the ecosystem. Many other frameworks promise to help you “build agents” or “wire LLMs into workflows”, and on a quick glance they all look similar: some Python, a few decorators, and a diagram with boxes and arrows. The real differences show up when you look at the canvas each framework gives you to design on: what it treats as first-class, what it hides, and where it assumes you will plug in your own infrastructure. This section places the Lang* stack next to two prominent canvases—CrewAI and Google’s Agentic Development Kit (ADK)—and briefly touches on a few other ecosystems you will encounter. The aim is not to pick winners, but to give you a clear mental model of what you gain and what you trade away with each approach.

3.5.1 CrewAI: teams and roles as the primary abstraction

CrewAI approaches agentic systems from the opposite direction. Instead of starting with graphs and state objects, it starts with people-shaped concepts: agents with roles, goals, and tools, grouped into “crews” that collaborate on tasks. In CrewAI you typically define:

- A set of agents, each with a role description, a skill set, and access to some tools.
- A set of tasks, each with instructions and expected outputs.
- A crew configuration that says which agents are involved and how they should work together (for example, one agent drafts, another reviews, a third coordinates).

The framework then orchestrates the conversation between agents: who gets the next message, how results are passed along, when a task is considered complete. You reason in terms of workflows like “researcher → writer → editor” rather than in terms of “node A updates state, edge B is taken when condition C holds”. Compared to LangGraph, this canvas is more opinionated and higher level:

- Control flow is shaped by the crew patterns; you have fewer levers for low-level branching and state management.
- Multi-agent collaboration is built-in; you do not have to invent your own conventions for agent-to-agent messaging.

The model is deliberately close to a human team, which can make complex collaborative systems easier to design and explain.

The trade-off is that extremely precise control over state, retries, error routing, or cross-cutting concerns can require more work, because the core abstraction is “agents talking” rather than “graph of deterministic steps with a shared state object”. For some applications that is a good trade: you get rich multi-agent behaviour with less boilerplate. For applications that need strict, auditable workflows and tight coupling to existing backends, the lower-level LangGraph style is often a better fit.

3.5.2 Google ADK: agentic applications as managed cloud services

Google’s Agent Development Kit (ADK) is another canvas, this time optimized for the Google Cloud ecosystem. It is an agent-first framework with a session and event-driven runtime: interactions are recorded as immutable Events (user messages, agent replies, tool calls/results,

state changes, errors), and each Session provides both a full event history and a mutable “state” scratchpad for the current conversation thread.

Where LangGraph focuses on explicit graphs (nodes and edges you define and run in your own runtime), ADK expects you to compose higher-level agent and workflow primitives. In ADK you typically:

- Instantiate one or more agents with model instructions and tools, including workflow agents such as Sequential/Parallel/Loop for multi-step and concurrent task flows.
- Wrap those agents in an app/runtime that handles sessions and event flow, so the system can manage ongoing conversations rather than isolated calls.
- Use runners and services that connect your agent to session and memory backends (local/in-memory for development, and managed Vertex AI services for production-grade persistence and long-term memory).

The design goal is to make it straightforward to stand up tool-using, multi-step agentic applications without building your own session and memory machinery. When you adopt the managed path (Vertex AI Agent Engine), you also inherit platform capabilities for deployment operations such as scaling, security, and monitoring.

On observability, ADK provides structured logging support and can be instrumented with OpenTelemetry on Google Cloud, and ADK-based agents deployed on Agent Engine can be inspected via Cloud console logging views. On integration, ADK supports tools broadly, and Google explicitly documents integrations such as Application Integration workflows/connectors, as well as guidance for authenticated tool access (for example OAuth-based patterns).

ADK is more opinionated about runtime concepts (events, sessions, state, memory services). That can reduce plumbing if you’re already invested in Google Cloud, but it also means accepting the platform’s choices—especially when you rely on managed sessions/memory and Agent Engine hosting.

3.5.3 Other ecosystems and how they fit

Beyond CrewAI and ADK, there are several other frameworks that you will encounter while working with LLMs. They tend to emphasise one of three axes: retrieval, orchestration, or multi-agent collaboration.

- Retrieval-centric frameworks focus first on connecting models to data. Systems in this space invest heavily in document ingestion, indexing, and retrieval pipelines, then expose LLM integration points on top. They shine when your main problem is turning a messy corpus into high-quality, grounded answers. Many of the ideas you will see in this book around loaders, splitters, retrievers, and RAG patterns align closely with that mindset.
- Orchestration-centric frameworks aim to make multi-step tool use and complex workflows easier. Some offer “auto” agents that recursively plan and execute subtasks until they reach a goal, often using chat between agents as the primary control-flow mechanism. These are very handy for rapid experimentation and research but can be harder to harden for production if they do not expose explicit state and control-flow primitives.
- Multi-agent-centric frameworks, like CrewAI, elaborate on the idea of agent teams, delegation, and communication. They give you constructs for defining roles, hierarchies,

and collaboration patterns, and they usually assume that you will integrate tools and retrieval from elsewhere.

The Lang* stack is unusual in that it tries to cover all three axes with a single set of primitives: retrieval and RAG in LangChain, orchestration in LangGraph, and evaluation in LangSmith. That breadth is one reason you see it used as a substrate under other systems rather than replaced by them.

3.5.4 Choosing and combining canvases

From a distance, all of these frameworks support the same story: models with tools, memory, and workflows. The real question is where you want to spend your design effort.

If you care most about explicit control, traceability, and integration with existing services, the LangChain / LangGraph / LangSmith stack provides a programmable canvas where every important step and state transition is visible and testable in code.

If you think first in terms of “a team of specialised agents working together”, a framework like CrewAI can be an efficient way to express that idea directly, sometimes even wrapping LangChain components under the hood.

If you want as much infrastructure as possible to be managed for you, and you are already committed to a cloud provider that offers an agentic toolkit like ADK, adopting that platform can simplify deployment, scaling, and security at the cost of some portability and low-level control.

These choices are not mutually exclusive. It is entirely reasonable to:

- Prototype agent behaviours using a higher-level multi-agent framework, then re-implement the stable parts as explicit LangGraph workflows for production.
- Use LangChain’s retrieval and tool abstractions inside other canvases.
- Feed traces from any of these systems into an evaluation layer modelled on the LangSmith approach, even if the underlying orchestration is not LangGraph.

3.6 SUMMARY

This chapter defines the practical stack you will use throughout the book: LangChain for composing model-centric steps, LangGraph for making those steps into explicit stateful workflows, and LangSmith for tracing, evaluation, and monitoring. The central idea is that “LLM application engineering” is not a single library problem. You need a way to build behaviour, a way to orchestrate it over time, and a way to observe and improve it without guessing. LangChain, LangGraph, and LangSmith are presented as one ecosystem precisely because those concerns are tightly coupled in real systems.

LangChain is positioned as the standard library for the middle layer of an LLM application: prompts, retrieval, model calls, tool calls, and post-processing. The chapter emphasises the architecture choices that make it feel like normal software: modular components with clear interfaces, composition in code via Runnables and the LangChain Expression Language, and provider-agnostic wiring so you can swap model vendors or vector stores without rewriting your business logic. The payoff is explicit dataflow (“input → prompt → model → parser”) rather than invisible string-bashing scattered across helper functions.

LangGraph then enters when straight-line chains stop being enough. By representing control flow as nodes and edges over an explicit state schema, you move loops, branching, retries, pauses, and human approval steps out of prompts and into inspectable workflow structure. Supersteps, checkpoints, threads, and interrupts are explained as the mechanics that make long-running and human-in-the-loop systems durable: you can stop, resume, and debug with state treated as first-class, not as an accident of chat history.

Finally, LangSmith closes the engineering loop. Traces turn executions into evidence; datasets turn representative interactions into repeatable tests; evaluators and experiments turn behaviour into measurable metrics; and dashboards surface operational drift in latency, token usage, and error rates. The chapter ends by framing the Lang* stack as a feedback loop and by placing it alongside other canvases such as CrewAI and Google ADK, so you can choose based on where you want control, portability, and managed infrastructure.

4 DEVELOPMENT ENVIRONMENT AND MODEL INTEGRATION

Before you can design chains, graphs, and agents, you need a development environment that is predictable, reproducible, and safe to experiment in. Large language model libraries move quickly: new features appear, APIs change, and transitive dependencies shift under your feet. If every project shares the same global Python installation and hard-coded API keys, even a small upgrade can silently break other work or leak credentials. This chapter focuses on building a clean, per-project setup where your LangChain, LangGraph, and LangSmith code can evolve without collateral damage.

The first part of the chapter walks through a concrete Python project setup on Windows using virtual environments and pip. You will learn how to create an isolated environment, install the core dependencies for LangChain, LangGraph, LangSmith, OpenAI, and Google AI models, and capture those versions in a requirements file so that colleagues and CI pipelines can reproduce the same state. The chapter shows how to keep secrets out of your source code by loading API keys and LangSmith configuration from a `'.env'` file, and how to centralise configuration in a small module so that the rest of your application remains agnostic about where credentials come from. By the end of this part you will have a minimal but realistic project layout, including a smoke test that sends a simple prompt to OpenAI and Gemini and optionally records traces in LangSmith.

The second part introduces local model backends and shows that, from LangChain's point of view, they are just another engine behind the same interfaces. You will see how to run models with Hugging Face Transformers, llama.cpp, GPT4All, and Ollama-style local servers, and how to wrap them in LangChain integrations so they behave like any other chat or LLM component in your chains and graphs. The chapter compares these options in terms of hardware requirements, latency, control over data, and operational complexity, and explains how to abstract model choice so you can switch between hosted and local backends without rewriting business logic. It closes with a brief survey of alternative environment tools such as Poetry, Conda, and Docker, tying them back to the same core principles: isolate each project, declare dependencies explicitly, and treat configuration and secrets as first-class concerns. By the time you move on to building applications, you will have a solid, repeatable foundation that lets you focus on system design rather than environment debugging.

4.1 PYTHON PROJECT SETUP: VIRTUAL ENVS, DEPENDENCY MANAGEMENT, AND SECRETS MANAGEMENT

Before you write a single line of LangChain or LangGraph code, you need a stable place where that code can live. Large-language-model libraries move quickly; dependencies change, new features arrive, old ones get removed. If you install everything globally on your machine, sooner or later one project will break another. A dedicated Python environment per project solves this. It gives you:

- Isolation: each project has its own versions of LangChain, LangGraph, LangSmith, and provider SDKs.
- Reproducibility: everyone on the team can recreate the same environment from a few simple commands.
- Safety: you can experiment (upgrade packages, try new providers) without risking other tools on your machine.

In this section we will build a small but realistic “LLM app” environment on Windows, using only the classic Command Prompt (cmd.exe) (but the same, with very few changes, can be created on Linux and MacOS). By the end you will have:

- A clean Python virtual environment.
- Dependencies installed for LangChain, LangGraph, LangSmith, OpenAI, and Google AI (Gemini).
- API keys and LangSmith configuration loaded from a `.env` file.
- A minimal project layout under `ch_04\src\`.
- A `main.py` that sends one test prompt to OpenAI and one to Google Gemini and prints both replies.

Later chapters will build real applications on top of this skeleton; here, the goal is to make sure your foundation actually runs.

4.1.1 Install Python and pip on Windows

If you already have a recent Python 3 installed and you know pip works, you can skip this section. Otherwise, follow it carefully; subtle installation mistakes produce extremely annoying bugs later.

1. Open a Command Prompt

Press ‘Win + R’, type:

```
Cmd
```

and press Enter.

2. Check whether Python is installed. In the Command Prompt, type:

```
python -version
```

If you see something like:

```
Python 3.11.7
```

you are good. If you see an error (“‘python’ is not recognized...”) or a version older than 3.10, install a current Python 3:

- Download the latest stable 3.x installer for Windows from the official Python website.
- Run the installer and make sure you tick:

“Add Python to PATH” before clicking “Install Now”.

Use Python 3.10 or later versions for this book. LangChain and its ecosystem now require at least Python 3.10, and all examples and code listings have been tested on 3.10 and 3.11. Older interpreters such as 3.8 or 3.9 will not work reliably with the current LangChain, LangGraph, and LangSmith packages and should be avoided.

After installation, close and reopen Command Prompt, then run again:

```
python -version
```

3. Verify pip is available. In the same Command Prompt:

```
python -m pip -version
```

You should see something like:

```
pip 24.0 from C:\Users\youname\AppData\Local\Programs\Python\Python311\Lib\site-packages\pip (python 3.11)
```

If you see an error, re-run the Python installer and ensure “pip” is included (it is by default in current installers).

4.1.2 Create a project folder and virtual environment

Next, we create a dedicated folder and a virtual environment inside it. We’ll call the project `llm-app`, but you can pick any name. In Command Prompt:

```
C:\Users\yourname> mkdir llm-app  
C:\Users\yourname> cd llm-app
```

Now create a virtual environment named `.venv` inside this folder:

```
C:\Users\yourname\llm-app> python -m venv .venv
```

This creates a `.venv` directory containing a private copy of the Python interpreter and its libraries for this project. Now activate the virtual environment:

```
C:\Users\yourname\llm-app> .venv\Scripts\activate
```

If activation succeeds, your prompt will change to something like:

```
(.venv) C:\Users\yourname\llm-app>
```

That `(.venv)` prefix is important: it tells you that from now on, `python` and `pip` refer to this isolated environment, not your global Python installation.

You must activate the environment every time you open a new Command Prompt and want to work on this project. If later you want to “leave” it, you can run:

```
(.venv) C:\Users\yourname\llm-app> deactivate
```



The commands shown so far use Windows paths and cmd.exe syntax, but the overall structure is the same on macOS and Linux. The main differences are how you create and activate the virtual environment. From a terminal in the project root, run:

```
python3 -m venv .venv
```

This creates a `.venv` folder exactly as on Windows. To activate it, use:

```
source .venv/bin/activate
```

Your prompt will change to show “`(.venv) ...`”, and any “`python`” or “`pip`” command in that shell now uses the virtual environment. The remaining steps in this chapter—creating `requirements.txt`, installing packages with “`pip install -r requirements.txt`”, creating the `src` folder, and running the example scripts—are conceptually identical across operating systems. The only differences are path separators and the exact way you start Python from your IDE or editor.

4.1.3 Install dependencies

To make the environment reproducible, start each project from a small, curated `requirements.txt` rather than installing packages ad-hoc. A typical file for our book looks like this:

```
(.venv) C:\Users\yourname\llm-app> type NUL > requirements.txt
```

And then add to the `requirements.txt` newly created file:

```
langchain==1.1.0
langchain-google-genai==3.2.0
langchain-openai==1.1.0
langgraph==1.0.4
langsmith==0.4.53
python-dotenv==1.2.1
```

You install these dependencies with:

```
(.venv) C:\Users\yourname\llm-app> python -m pip install --upgrade pip
(.venv) C:\Users\yourname\llm-app> pip install -r requirements.txt
```

What each package does:

- Langchain: core primitives (prompts, chains, tools, retrievers, etc.).
- Langgraph: graph-based orchestration for agents and workflows.
- Langsmith: SDK and client for tracing, evaluation, and dataset/eval runs.
- langchain-openai: OpenAI chat/embedding integrations.
- langchain-google-genai: Gemini chat/embedding integrations.
- python-dotenv: loads `.env` into environment variables.`.helper` to load configuration values from a `.env` file into environment variables.

This approach pins versions explicitly, so a reader running the examples in six months sees the same behaviour you saw when writing them. An alternative is to run `pip freeze > requirements.txt` and commit the entire environment, including indirect dependencies. That captures every version but tends to produce large, noisy files and makes deliberate upgrades harder. The book uses the curated style: only top-level libraries appear in `requirements.txt`, with exact versions or narrow ranges, and you update them intentionally when needed.

You can also install Jupyter for interactive notebooks (optional but handy for exploration):

```
(.venv) C:\Users\yourname\llm-app> pip install jupyter
```

4.1.4 Get your API keys and LangSmith credentials

Our test environment will talk to three external services:

- OpenAI (for models like `gpt-4.1-mini`).
- Google AI (for Gemini models via the Google AI API).
- LangSmith (for tracing and evaluating LangChain runs).

Each service gives you a secret API key. The UI in these systems to get the APIs will change over time, but the basic flow is always the same:

1. Create an account (if you don't already have one).
2. Go to the "API keys" or "Developer" section of the account dashboard.
3. Create a new secret key.
4. Copy it somewhere safe.

We do not hard-code these values into Python files. Instead, we keep them in a `.env` file that lives next to your code and is read at startup (and never shared publicly).

4.1.5 Store configuration in a `.env` file

At the root of the `llm-app` folder (the same level as `.venv`), create a text file named `.env`:

```
(.venv) C:\Users\yourname\llm-app> type NUL > .env
```

Open it in your editor and add:

```
# OpenAI configuration
OPENAI_API_KEY=your-openai-key-here

# Google AI (Gemini) configuration
GOOGLE_API_KEY=your-google-ai-key-here

# LangSmith / LangChain tracing configuration
LANGCHAIN_TRACING_V2=true
LANGCHAIN_API_KEY=your-langsmith-key-here
LANGCHAIN_ENDPOINT=https://api.smith.langchain.com
LANGCHAIN_PROJECT=default
```

Replace the placeholder values with your real keys and chosen project name. A few notes:

- `OPENAI_API_KEY` – your OpenAI secret key.
- `GOOGLE_API_KEY` – your Google AI (Gemini) key from Google AI Studio / Developer console.

- `LANGCHAIN_TRACING_V2` – set to ‘true’ to enable LangSmith tracing; set to ‘false’ (or remove it) to keep tracing off.
- `LANGCHAIN_API_KEY` – your LangSmith key.
- `LANGCHAIN_ENDPOINT` – default endpoint for LangSmith’s API.
- `LANGCHAIN_PROJECT` – a project name that will group traces for this book’s examples. Default is the name of the project when you create a new account.

Treat `.env` as sensitive: never publish it or share it in public repositories.

4.1.6 Create the basic project structure

We’ll keep the project layout extremely simple and grow it as we go:

```
llm-app\
  .venv\           # virtual environment (auto-created)
  .env            # secrets & configuration (you created this)
  requirements.txt # pip libraries dependencies
  ch_04\src\
    __init__.py   # marks src as a Python package
    config.py     # loads configuration from .env
    main.py       # test script to call OpenAI and Google AI
```

Create the `ch_04` and `src` folders and the three files. In Command Prompt:

```
(.venv) C:\Users\yourname\llm-app> type NUL > .env
(.venv) C:\Users\yourname\llm-app> mkdir ch_04
(.venv) C:\Users\yourname\llm-app> cd ch_04
(.venv) C:\Users\yourname\llm-app\ch_04> mkdir src
(.venv) C:\Users\yourname\llm-app\ch_04> cd src
(.venv) C:\Users\yourname\llm-app\ch_04\src> type NUL > __init__.py
(.venv) C:\Users\yourname\llm-app\ch_04\src> type NUL > config.py
(.venv) C:\Users\yourname\llm-app\ch_04\src> type NUL > main.py
```

You can also create and edit the files with your favourite editor (VS Code, PyCharm, Notepad++, ...); the exact tool doesn’t matter, only the file names and locations do.

4.1.7 Implement config.py – loading environment configuration

Open `ch_04\src\config.py` and add:

```
from dotenv import load_dotenv
import os

# Load values from .env into environment variables as soon as this module is imported
load_dotenv()

# Provider API keys
OPENAI_API_KEY = os.getenv("OPENAI_API_KEY")
GOOGLE_API_KEY = os.getenv("GOOGLE_API_KEY")

# LangSmith / LangChain tracing configuration
LANGCHAIN_TRACING_V2 = os.getenv("LANGCHAIN_TRACING_V2", "false").lower() == "true"
LANGCHAIN_API_KEY = os.getenv("LANGCHAIN_API_KEY")
LANGCHAIN_ENDPOINT = os.getenv("LANGCHAIN_ENDPOINT")
LANGCHAIN_PROJECT = os.getenv("LANGCHAIN_PROJECT", "<lang-chain-project>")

def configure_langsmith_env() -> None:
    # Configure environment variables expected by LangSmith.
    # Call this once at startup before you create LangChain or LangGraph objects.
    if LANGCHAIN_TRACING_V2 and LANGCHAIN_API_KEY:
        # Enable tracing
        os.environ["LANGCHAIN_TRACING_V2"] = "true"
        os.environ["LANGCHAIN_API_KEY"] = LANGCHAIN_API_KEY
```

```

if LANGCHAIN_ENDPOINT:
    os.environ["LANGCHAIN_ENDPOINT"] = LANGCHAIN_ENDPOINT
if LANGCHAIN_PROJECT:
    os.environ["LANGCHAIN_PROJECT"] = LANGCHAIN_PROJECT

```

ch_04\src\config.py

What this does:

- `load_dotenv()` reads the `.env` file at project root and adds its values to the process environment.
- We cache key values (`OPENAI_API_KEY`, `GOOGLE_API_KEY`, etc.) in module-level variables for convenience.
- `configure_langsmith_env()` mirrors the LangSmith-related values back into `os.environ` in the exact format LangChain expects, but only if tracing is enabled and a key is present.

This pattern keeps all “where do I get my secrets from?” logic in one place. Later, when you build proper services, you can replace `.env` with Windows environment variables, Azure Key Vault, or any other secret manager without touching the rest of the code.

4.1.8 Implement `main.py` – testing OpenAI and Google Gemini

Now open `ch_04\src\main.py` and add:

```

from .config import (
    OPENAI_API_KEY,
    GOOGLE_API_KEY,
    configure_langsmith_env,
)
from langchain_openai import ChatOpenAI
from langchain_google_genai import ChatGoogleGenerativeAI

def main() -> None:
    # Configure LangSmith tracing if enabled in .env
    configure_langsmith_env()

    # Basic safety checks so failures are clear
    if not OPENAI_API_KEY:
        raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

    if not GOOGLE_API_KEY:
        raise RuntimeError("GOOGLE_API_KEY is not set. Check your .env file.")

    # --- OpenAI chat model setup ---
    # ChatOpenAI reads OPENAI_API_KEY from the environment by default.
    openai_model = ChatOpenAI(
        model = "gpt-4.1-mini",
        temperature = 0.2,
    )

    # --- Google Gemini chat model setup ---
    # ChatGoogleGenerativeAI reads GOOGLE_API_KEY from the environment.
    google_model = ChatGoogleGenerativeAI(
        model = "gemini-2.5-flash",
        temperature = 0.2,
    )

    prompt = "Say hello from my new LangChain environment in one short sentence."

    # Call OpenAI
    openai_response = openai_model.invoke(prompt)
    print("OpenAI replied:")
    print(openai_response.content)
    print()

    # Call Google Gemini
    google_response = google_model.invoke(prompt)

    # For Gemini 3 models, .text is a convenient shortcut
    # for the main text content
    print("Google Gemini replied:")
    print(google_response.text)

```

```
if __name__ == "__main__":
    main()
```

ch_04\scr\main.py

A few comments on this code:

- We import from `.config` using a relative import because `src` is a package (thanks to `__init__.py`), and we will run this as `python -m src.main`.
- `ChatOpenAI` and `ChatGoogleGenerativeAI` are both high-level chat model wrappers that play nicely with LangChain's tools, graphs, and memory later on.
- We keep temperature low (0.2) to get relatively stable outputs while you are still debugging the environment itself.
- If any of the required keys are missing, you get an immediate, explicit error instead of a mysterious authentication failure.

4.1.9 Run the smoke test

Make sure the virtual environment is active (you see `(.venv)` at the beginning of the prompt). If not, reactivate it:

```
C:\Users\yourname\llm-app> .venv\Scripts\activate
(.venv) C:\Users\yourname\llm-app>
```

Now run your test module:

```
(.venv) C:\Users\yourname\llm-app> cd ch_04
(.venv) C:\Users\yourname\llm-app\ch_04> python -m src.main
```

If everything is wired correctly, you should see something like:

```
OpenAI replied:
Hello from your new LangChain environment! I'm ready when you are.

Google Gemini replied:
Hello from your new LangChain and Gemini setup!
```

At this point you have a working, isolated Python environment that can:

- Load secrets from `.env`.
- Talk to OpenAI via LangChain.
- Talk to Google Gemini via LangChain.
- Optionally send traces to LangSmith when you enable it.

From here, all later examples in the book will assume a similar layout: a virtual environment, configuration in `.env`, and code living under `ch_xx\src\` in a structured package. As your applications grow, you will add more modules (for tools, retrievers, graphs, evaluators) without changing the fundamental setup.

4.1.10 Other environment options (Poetry, Conda, Docker) – short overview

For completeness, here is how this picture generalises to other tools you might already use. We will not walk through them step by step, but it is useful to know how they fit into the same mental model.

- `venv + pip` (what you just used): `venv` creates an isolated environment on top of an existing Python installation, so each project can have its own installed packages. Most Python

installs can also bootstrap pip into that environment, but some minimal/OS-packaged distributions may require additional setup. A good fit when you want minimal moving parts and you are comfortable working from the command line.

- Conda: Manages environments that can include both Python packages and non-Python dependencies (compiled libraries and executables). Environments are commonly described declaratively and can be created from an environment specification file (often `environment.yml`).
- Poetry: A Python dependency manager built around `pyproject.toml`. Poetry resolves and pins dependency versions in a lockfile (`poetry.lock`) and, by default, creates and uses a dedicated virtual environment for the project (configurable).
- Docker: Packages your application and its dependencies into a container image that can be run by a container runtime. This is commonly used to make runtime behaviour consistent across machines and environments. Later in the book, when we containerise LangGraph-based systems and connect them to production services, we will revisit Docker in more detail.

Whichever toolchain you prefer, the core ideas stay the same:

- Keep each project in its own environment.
- Declare dependencies in a file (`requirements`, `environment.yml`, `pyproject/lockfile`, `Dockerfile`).
- Load API keys and LangSmith settings from configuration, not hard-coded literals.

Do this, and you avoid a large class of time-wasting environment and dependency bugs, freeing your attention for the interesting part: designing and building the LLM applications.

4.2 RUNNING LOCAL MODELS (TRANSFORMERS, LLAMA.CPP, GPT4ALL, OLLAMA-STYLE SETUPS)

Running models locally is not a different “kind” of LangChain application. It is the same patterns, prompts, and chains, just with a different engine under the hood. Instead of sending prompts to a hosted API, you talk to a model that runs on your own machine or on infrastructure you control. The motivation is straightforward: you might want tighter control over data, to experiment with specific open-weight models, or to reduce per-token costs once a system is stable. The trade-offs are equally clear: you become responsible for downloading models, provisioning CPU/GPU resources, and keeping latency within acceptable bounds.

LangChain treats local models as just another backend. Whether the model is remote or local, you interact with a standard LLM or chat-model interface and plug it into the rest of the stack: prompt templates, LCEL runnables, tools, graphs, and evaluation. The differences are:

- how the model is installed and started,
- how you reference it (model name vs local file path),
- and what hardware and context-window limits you must respect.

The rest of this section walks through the main local options you are likely to use today: Hugging Face Transformers, llama.cpp, GPT4All, and an Ollama-style local model server.

4.2.1 Shared integration pattern in LangChain

Most local backends follow the same basic pattern in LangChain:

1. Install the model runtime and the LangChain integration package.
2. Construct a model wrapper (LLM or chat model) with backend-specific settings.
3. Call the model through the standard Runnable interface (invoke, batch, stream), either directly, via LCEL composition, or as part of a larger chain/graph.

A typical Python environment for local models might include:

```
pip install "langchain-core>=0.3" "langchain-community>=0.3" transformers
accelerate
pip install llama-cpp-python gpt4all
pip install "langchain-ollama>=0.1.0"
pip install "langchain-huggingface>=0.0.0"
```

The exact versions will evolve, but the package roles are stable:

- langchain-core provides the base abstractions (Runnables, prompts, parsers).
- langchain-community contains many community-maintained integrations (for example llama.cpp, GPT4All).
- Some integrations ship as dedicated packages, such as langchain-ollama for Ollama and langchain-huggingface for Hugging Face wrappers.

Once you have the appropriate model wrapper, the rest of your pipeline can usually be written in a backend-agnostic way. That is the design goal: you should be able to swap “local” and “remote” model backends without rewriting the workflow, while still recognising that specific capabilities (for example streaming, tool calling, or context length) depend on the backend.

4.2.2 Local models with Hugging Face Transformers

Hugging Face Transformers is a common and flexible way to run open-weight models locally. It supports a wide range of architectures and tasks, and its `from_pretrained` loading pattern works both with models hosted on the Hugging Face Hub and with models stored in a local directory. LangChain integrates with Transformers pipelines via the `HuggingFacePipeline` wrapper. The legacy wrapper in `langchain_community` is deprecated in favour of the partner package `langchain_huggingface`, which is jointly maintained to stay aligned with Hugging Face updates. A minimal setup looks like this:

```
from langchain_huggingface import HuggingFacePipeline
from transformers import AutoModelForCausalLM, AutoTokenizer, pipeline

model_id = "gpt2" # small, easy-to-run example model
tokenizer = AutoTokenizer.from_pretrained(model_id)
model = AutoModelForCausalLM.from_pretrained(model_id)

generation_pipeline = pipeline(
    "text-generation",
    model = model,
    tokenizer = tokenizer,
    max_new_tokens = 128
)

llm = HuggingFacePipeline(pipeline = generation_pipeline)

response = llm.invoke("Write a one-line greeting for a new user.")
print(response)
```

What this does is simple: `AutoTokenizer` and `AutoModelForCausalLM` load the tokenizer and weights, and `pipeline(...)` builds a Transformers pipeline using those objects. Transformers pipelines accept the model and tokenizer explicitly via the `model` and `tokenizer` parameters. `HuggingFacePipeline` then wraps that pipeline and exposes it through LangChain's standard invocation interface, so you can call `invoke` and compose it with LCEL like any other model wrapper.

One important limitation to state explicitly: this wrapper only supports a subset of pipeline tasks. In the community wrapper documentation, it is limited to `text-generation`, `text2text-generation`, `summarization`, and `translation`. In the partner-package implementation, support also includes `image-text-to-text`, but it is still a bounded list rather than “any pipeline task”.

In practice, you will usually run a larger model than `gpt2`, place it on the right device (CPU vs GPU), and tune generation settings such as temperature and `top_p`. For input length, treat “maximum sequence length” as model-dependent: tokenizers often expose a practical limit via `tokenizer.model_max_length`, while model config fields vary across architectures.

4.2.3 llama.cpp models via LlamaCpp

When you want local inference with a `llama.cpp`-compatible model using the `llama.cpp` C/C++ implementation, `llama.cpp` is a widely used option. LangChain integrates with it through the `LlamaCpp` LLM class in `langchain_community.llms`, which relies on the `llama-cpp-python` bindings and instantiates a `llama_cpp.Llama` client under the hood. Assuming you already have a `llama.cpp`-compatible GGUF model file on disk (often quantized), a minimal LangChain usage looks like this:

```
from langchain_community.llms import LlamaCpp
llm = LlamaCpp()
```

```

        model_path = "/models/llama-3-8b-instruct-q4_k_m.gguf",
        n_ctx = 2048,
        temperature = 0.2
    )

    answer = llm.invoke("List three key steps in handling a customer complaint.")
    print(answer)

```

Key points:

- `model_path` points to your local model file. The wrapper requires llama-cpp-python and uses it to load the model.
- `n_ctx` sets the context window: the maximum number of tokens the model can process at once, so the prompt plus generated tokens must fit within that budget.
- Parameters like `temperature` and `max_tokens` control sampling and output length (`max_tokens` is the wrapper's "maximum tokens to generate").

If you also want local embeddings, LangChain exposes `LlamaCppEmbeddings` in `langchain_community.embeddings`, which similarly uses llama-cpp-python with a local model path. This setup is a common choice when you need local inference and want your data to stay on your own infrastructure (with performance depending on your hardware and quantization settings).

4.2.4 GPT4All as a local model runtime

GPT4All runs models locally using a llama.cpp-based backend (with multiple hardware backends such as CPU, CUDA, Metal, and Kompute) and provides a desktop experience for browsing and downloading models, including GGUF files. LangChain integrates with GPT4All via the `GPT4All` LLM wrapper in `langchain_community.llms`. A typical pattern:

```

from langchain_community.llms import GPT4All

llm = GPT4All(
    model = "/models/gpt4all-13b-snoozy.gguf",
    max_tokens = 128,
    temperature = 0.2
)

summary = llm.invoke("Summarise this log line in one sentence:\n<log line here>")
print(summary)

```

What happens here:

- `model` points to a local GGUF model file (for example, one you downloaded via the GPT4All desktop app).
- The wrapper exposes the standard LangChain LLM interface, so it can be composed in LCEL pipelines and used anywhere an LLM is expected.

GPT4All is a convenient entry point for local experimentation because it bundles a llama.cpp-based runtime with a simple model download/catalog workflow, reducing the amount of low-level setup you need to do yourself.

4.2.5 Ollama-style local model servers

Ollama uses a "local model server" approach: you run a local server process that manages models and exposes an HTTP API, and client libraries (including LangChain) send requests to that service. LangChain's Ollama integration lives in the `langchain-ollama` package via the `ChatOllama` chat model. Typical setup:

1. Install and run Ollama.
2. Pull a model with the Ollama CLI, for example:

```
ollama pull llama3.1
```

3. Use ChatOllama from Python:

```
from langchain_ollama import ChatOllama
llm = ChatOllama(model = "llama3.1")
messages = [
    ("system", "You are a helpful assistant."),
    ("human", "Give me two short bullet points about model evaluation."),
]
response = llm.invoke(messages)
print(response.content)
```

Ollama's main advantages are simple model management via the CLI, an HTTP interface similar to hosted chat APIs, and straightforward integration with LangChain through ChatOllama, which you can usually drop into existing chains or graphs that accept a chat model.

4.2.6 Choosing and abstracting your local backend

From the point of view of LCEL and LangGraph, most local model backends are swappable because they are exposed through the same Runnable-style calling pattern. You wire a model into a pipeline, call it with invoke, and—when the specific integration supports it—stream partial outputs. In practice, “interchangeable” does not mean “drop-in identical”: some backends are chat models while others are text-completion LLMs, and you may need small adapters around prompt format and output parsing to keep the rest of your workflow stable.

Typical local options include:

- a Hugging Face pipeline wrapper around a Transformers model,
- a LlamaCpp wrapper (llama.cpp via llama-cpp-python) loading a local GCUF model,
- a GPT4All LLM configured with a local model file,
- or a ChatOllama chat model served by an Ollama daemon.

This abstraction is deliberate: your chains and graphs should depend on a stable interface, not on one vendor or runtime. The practical choice between backends then comes down to three things: which model families you need access to, what hardware you have available, and how much control you want over runtime and deployment details.

4.3 SUMMARY

This chapter treats your development environment as part of the system design, not as a one-off setup chore. It shows how to build a predictable, per-project Python workspace so LangChain, LangGraph, and LangSmith can evolve without breaking other projects or leaking credentials. You create an isolated virtual environment, pin a small set of top-level dependencies in `requirements.txt`, and keep secrets out of source code by loading provider keys and LangSmith settings from a `.env` file. Configuration is centralised in a small module so the rest of the code stays clean and provider-agnostic, and a simple smoke test

proves the environment is wired correctly by calling both OpenAI and Gemini (and optionally emitting traces).

The chapter then makes an important point about local inference: running models locally is the same application pattern with a different engine. Whether you use Transformers via a Hugging Face pipeline, llama.cpp via LlamaCpp and a GGUF file, GPT4All with a local model file, or an Ollama-style local server through ChatOllama, LangChain wraps these backends behind a consistent calling style. That abstraction lets you keep chains and graphs stable while swapping runtimes, with the practical caveat that capabilities differ (chat vs completion semantics, streaming support, context limits, and tool calling). The chapter closes by mapping the same principles onto Poetry, Conda, and Docker: isolate each project, declare dependencies explicitly, and treat configuration and secrets as first-class concerns.

Part 2 - Prompts, LCEL workflows, and LangGraph fundamentals

Chapter 5: Prompt design and dynamic prompting

Chapter 6: Context engineering

Chapter 7: LCEL and composable workflows in LangChain

Chapter 8: LangGraph fundamentals and state management

5 PROMPT DESIGN AND DYNAMIC PROMPTING

Prompt design is the part of your system that turns vague intent into precise instructions a model can follow. It is not about “magic words”. It is about building repeatable, testable inputs that produce reliable outputs under real constraints such as context limits, cost, and the need for structured data.

This chapter builds that foundation in four steps. It starts with chat models and the generation parameters that control sampling behaviour, so you can choose settings that match your use case instead of guessing. It then introduces prompt templates as reusable building blocks, covering `PromptTemplate`, `ChatPromptTemplate`, and message-based patterns that keep prompts consistent as inputs change. Next, it explains zero-shot and few-shot prompting, then shows how dynamic few-shot selection uses embeddings and example selectors to pick the best examples per request. Finally, it covers higher-control prompting patterns: chain-of-thought and self-consistency for more reliable reasoning, tree-of-thought for structured exploration, and summarisation-oriented prompts such as chain-of-density for compressing long content into a usable context window.

5.1 CHAT MODELS AND GENERATION PARAMETERS (OPENAI AND GOOGLE AI)

In LangChain, you typically do not call provider HTTP endpoints directly. You work through model wrappers that present a consistent interface across providers. For OpenAI and Google Gemini, LangChain commonly exposes two wrapper styles:

- Plain LLM wrappers that take a single string and return a single string (for example, `OpenAI`, `GoogleGenerativeAI`), and
- Chat wrappers that take a list of structured messages and return a message (for example, `ChatOpenAI`, `ChatGoogleGenerativeAI`).

Chat wrappers are designed for instruction-tuned, conversational models where inputs are naturally expressed as system, user, and assistant messages across multiple turns. This structure reduces prompt-formatting boilerplate and makes role separation explicit. In this book, all model calls use chat wrappers with a small, explicit set of generation parameters so behaviour is consistent and easy to evaluate.

5.1.1 LLM vs chat wrappers in LangChain

For OpenAI and Google Gemini specifically:

- `OpenAI` (from `langchain_openai`) is documented as a legacy text completion LLM.
- `ChatOpenAI` is the main chat wrapper for OpenAI's chat models.
- `GoogleGenerativeAI` is the LLM-style wrapper for Gemini text generation.
- `ChatGoogleGenerativeAI` is the chat wrapper for Gemini chat models.

For new systems, you should treat `ChatOpenAI` and `ChatGoogleGenerativeAI` as the defaults. Plain LLM wrappers are still useful for compatibility and for some non-chat utilities, but they are secondary.

5.1.2 OpenAI

The OpenAI integration in LangChain uses the `langchain-openai` package.

`ChatOpenAI`

- Wraps OpenAI chat models. It is fully compatible with the Chat Completions API, and it can also use the Responses API (and will route to it when you enable features such as built-in tools or conversation-state management).
- Implements LangChain's chat-model Runnable interface: you invoke it with chat inputs (for example, a list of messages) and it returns an AI message.
- Supports explicit configuration such as `model`, `temperature`, `max_tokens` (alias `max_completion_tokens`), `request_timeout`, and `max_retries`. Additional OpenAI parameters (for example stop sequences and penalties) are also supported either as explicit fields or via `model_kwarg`s / extra request fields.

```
from __future__ import annotations
import os
from typing import Dict
from langchain_openai import ChatOpenAI
from langchain_core.messages import HumanMessage
from .config import OPENAI_API_KEY

def main() -> None:
    if not OPENAI_API_KEY:
```

```

        raise RuntimeError("OPENAI_API_KEY is not set.")

# Example token bias: discourage these tokens
logit_bias: Dict[str, int] = {
    "2869": -100, # " sorry"
    "6342": -100, # " apologize"
}

llm = ChatOpenAI(
    model = "gpt-4.1-mini",
    temperature = 0.3,
    top_p = 0.95,
    max_tokens = 200,                      # alias of max_completion_tokens
    request_timeout = 30,                  # seconds
    max_retries = 2,
    stop = ["\n\nUser:", "\n\nHuman:"],
    frequency_penalty = 0.2,
    presence_penalty = 0.1,
    logit_bias = logit_bias,
    seed = 42,                            # IMPORTANT: pass explicitly (not in model_kwargs)
    model_kwargs = {
        # Extra OpenAI request fields (only those valid for your endpoint/model)
        "response_format": {"type": "text"}
    }
)

resp = llm.invoke(
    [HumanMessage(content = "Explain LangChain in one short paragraph. Avoid apologetic language.")]
)
print(resp.content)

if __name__ == "__main__":
    main()

```

ch_05\scr\allparams_ChatOpenAI.py

Where:

Feature	Value	Description
Model	model="gpt-4.1-mini"	Selects which OpenAI chat model to use for generation. It controls the model's capabilities, latency, and cost profile.
Temperature	temperature=0.3	Randomness dial for token sampling. Lower values make outputs more deterministic; higher values increase variety and risk of "creative drift".
Max tokens	max_tokens=200	Upper bound on how many tokens the model may generate for the completion. It caps output length (and therefore cost). If the request would exceed this, the model stops early.
Alias support	max_completion_tokens also works	Same intent as max_tokens. Some libraries/endpoints prefer one name; LangChain accepts both and maps accordingly.
Request timeout	request_timeout=30	Client-side time limit (in seconds) for the API call before the request is aborted. This

		protects your app from hanging on slow networks or overloaded endpoints.
Retry logic	<code>max_retries=2</code>	How many times the client will retry a failed request (typically transient failures like timeouts, 5xx, rate limits depending on the client's retry policy). Higher values increase resilience but also latency under failure.
Stop sequences	<code>stop=[...]</code>	One or more stop sequences. If the model starts emitting any of these strings, generation is truncated immediately. Useful to prevent the model from continuing into unwanted sections (e.g., "User:" prompts) or to enforce formatting boundaries.
Frequency penalty	<code>frequency_penalty=0.2</code>	Discourages repeating the same tokens based on how frequently they've already appeared in the generated text. Higher values reduce repetition/looping but can make phrasing choppy if too high.
Presence penalty	<code>presence_penalty=0.1</code>	Discourages reusing tokens that have already appeared at all (regardless of count). Higher values push the model to introduce new topics/words; can improve diversity but may reduce focus.
Nucleus sampling	<code>top_p=0.95</code>	Nucleus sampling threshold. Instead of always considering the full vocabulary, the model samples from the smallest set of tokens whose cumulative probability is <code>top_p</code> . Lower values make outputs more conservative; higher values allow more variety.
Seed	<code>seed=42</code>	When fixed, the model makes a best effort to return the same response for repeated requests, but determinism is not guaranteed and can vary with model/version and other parameters (e.g., temperature).

Token bias	<code>logit_bias={...}</code>	Per-token bias applied before sampling. Positive values make specific tokens more likely; negative values make them less likely. Used for strong steering (e.g., suppress certain words) but can create awkward outputs if overused. Token IDs are model/tokenizer-specific.
Advanced OpenAI fields	<code>model_kwargs={...}</code>	“escape hatch” for extra OpenAI request fields not exposed as first-class constructor args. LangChain passes these through to the underlying OpenAI request. Use this for advanced/less common fields, but be careful: invalid combinations can trigger 400 errors (e.g., <code>parallel_tool_calls</code> requires tools to be present).

In practice, `ChatOpenAI` is the class you use for OpenAI’s chat model families (for example `gpt-4.1-mini`). The `OpenAI` class in `langchain-openai` targets the legacy “text completion” style interface (string in → string out) and is mainly relevant when you are intentionally using completion models rather than chat models.

ChatOpenAI Reasoning

Not all models behave the same way when faced with a messy problem. Some are primarily “next-token predictors” controlled by sampling parameters such as `temperature` and `top_p`. Others are designed to spend additional internal compute on reasoning before they answer. OpenAI’s o-series models (for example `o3-mini`) fall into the second category: they are tuned to “think more” on hard tasks, and one of the main levers you can control is how much effort the model should spend on that internal reasoning step.

In LangChain, this is exposed directly on `ChatOpenAI` as the `reasoning_effort` parameter. Conceptually, `reasoning_effort` is not about making output more random or more conservative; it is about trading off latency and token usage against deeper problem solving. Lower effort tends to be faster and cheaper; higher effort tends to do more internal work before producing the final answer. This is a setting intended for reasoning models only, with the effect that reducing effort can yield faster responses and fewer tokens spent on reasoning.

Supported values depend on the model/provider combination. In many examples you will see “low”, “medium”, and “high”. Some environments also support “minimal” in addition to those. Treat “medium” as a sensible default when you do not have profiling data yet and then tighten based on what your application actually needs. A practical way to decide:

- Use low (or minimal where supported) for interactive UX paths where you mainly need a reasonable answer quickly (for example first-line triage, routing, or simple extraction).

- Use medium for everyday “professional” work where correctness matters but you still care about throughput.
- Use high for genuinely hard reasoning tasks (multi-constraint planning, tricky debugging, financial or technical analysis), where a slower but more reliable answer is worth it.

```
from .config import OPENAI_API_KEY
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

if not OPENAI_API_KEY:
    raise RuntimeError("OPENAI_API_KEY is not set. Check your environment or .env file.")

template = ChatPromptTemplate.from_messages([
    [
        ("system", "You are a problem-solving assistant."),
        ("user", "{problem}")
    ]
])

chat = ChatOpenAI(
    model = "o3-mini",
    reasoning_effort = "high"
    # commonly: "low" | "medium" | "high" (some setups also support "minimal")
)

chain = template | chat
response = chain.invoke({"problem": "Design an algorithm to find the kth largest element
                                in an unsorted array."})
print(response.content)
```

ch_05\scr\reasoning.py

`reasoning_effort` gives you a “dial” that can reduce the need for heavy prompt-engineering tricks to force step-by-step reasoning while still letting you choose speed vs depth depending on the call site. It is also a clean fit with the book’s broader design goal: keep high-level chains readable, and push operational behaviour into explicit, testable configuration.

5.1.3 Google AI

Google’s Gemini models are exposed in LangChain via the `langchain-google-genai` package.

ChatGoogleGenerativeAI

- The primary chat-model wrapper for the Gemini family in LangChain.
- A `BaseChatModel` that you can compose as a `Runnable` in LCEL pipelines.
- Accepts common generation controls such as `model`, `temperature`, `top_p`, `top_k`, `max_output_tokens`, and `n` (the number of completions to request per prompt).

As with OpenAI, the chat wrapper is the one that aligns with all the multi-turn and tool-using flows you will build later. Before running the following example, please install :

```
(.venv) C:\Users\alexc\llm-app\ch_05> python -m pip install -U google-
genai langchain-google-genai
```

```
from __future__ import annotations
import os
from .config import GOOGLE_API_KEY
from langchain_google_genai import ChatGoogleGenerativeAI
from langchain_core.messages import HumanMessage
from google.genai.types import HarmBlockThreshold, HarmCategory

def main() -> None:
    if not GOOGLE_API_KEY:
        raise RuntimeError("GOOGLE_API_KEY is not set (export it in your environment).")
```

```

# Safety settings example from the LangChain reference
safety_settings = {
    HarmCategory.HARM_CATEGORY_DANGEROUS_CONTENT:
        HarmBlockThreshold.BLOCK_MEDIUM_AND ABOVE,
    HarmCategory.HARM_CATEGORY_HATE_SPEECH: HarmBlockThreshold.BLOCK_ONLY_HIGH,
    HarmCategory.HARM_CATEGORY_HARASSMENT: HarmBlockThreshold.BLOCK_LOW_AND ABOVE,
    HarmCategory.HARM_CATEGORY_SEXUALLY_EXPLICIT: HarmBlockThreshold.BLOCK_NONE
}

llm = ChatGoogleGenerativeAI(
    model = "gemini-2.5-flash",
    temperature = 0.3,
    top_p = 0.95,
    top_k = 40,
    max_output_tokens = 200,
    request_timeout = 30,
    max_retries = 2,
    stop = ["\n\nUser:", "\n\nHuman:"],
    seed = 42,
    safety_settings = safety_settings,
    model_kwarg = {}
)
resp = llm.invoke(
    [
        HumanMessage(
            content = "Explain LangChain in one short paragraph. Keep it concrete."
        )
    ]
)
print("MODEL RESPONSE:")
print(resp.content)

if __name__ == "__main__":
    main()

```

ch_05\scr\allparams_ChatGoogleGenerativeAI.py

All features are the same as for ChatOpenAI with the additional feature:

Feature	Value	Description
top_k	top_k=40	Enables top-k sampling: at each generation step, the model considers only the top_k most probable next tokens (then sampling is shaped by temperature/top_p if set). Must be a positive integer.
Safety settings	safety_settings={...}	Configures Gemini safety filters by mapping each harm category to a blocking threshold. The API can block responses that exceed the chosen risk threshold for that category (some core harms are always blocked regardless of settings).

5.1.4 Core chat generation parameters

Understanding how to control a model's generation behavior is crucial once you move from demos to production. If you leave defaults untouched, you often get answers that are too creative for support, too verbose for UI surfaces, or too inconsistent to test reliably. In customer service you usually want stable, factual wording and clear refusals when information is missing. In content generation you may prefer more variety and a more promotional tone.

Generation parameters are the small set of “decoding knobs” that let you steer this behavior without changing your prompts or your application logic.

A chat model writes its response one token at a time. After each token, it decides what to write next by considering the text so far. Internally, it gives every possible next token a score, turns those scores into probabilities, and then picks one token to continue the output. Generation parameters control this choice. At a high level:

- Logits are the raw scores the model assigns to each possible next token before turning them into probabilities.
- Temperature controls how “random” the choice is. Lower values make the model stick to the most likely tokens; higher values allow more variety.
- Top-p (nucleus sampling) limits the choice to the smallest set of tokens whose probabilities add up to p.
- Top-k (where supported) limits the choice to the k most likely tokens.
- Repetition penalties (provider-specific) reduce the chance of repeating the same words or phrases.
- Max-tokens limits and stop sequences control how long the model is allowed to talk and where it must stop.

These parameters work together. Temperature changes how “flat” or “sharp” the token probability distribution is, while top-p/top-k (when available) restrict the candidate set the model is allowed to sample from. Max-tokens and stop sequences do not change token choice; they enforce boundaries on length and termination. Penalties nudge the model away from repetition, but large values can backfire by making it avoid important repeated terms (product names, codes, identifiers).

LangChain’s chat wrappers expose many of these settings as constructor or call arguments as we have seen, but availability is provider-specific. Some parameters may be ignored (silently) by a provider, while others may be rejected as invalid. Treat “supported parameters” as part of your integration contract: verify them with a small test call for every provider/model you ship.

5.1.4.1 Model

The model parameter selects the underlying model family and size. In this book we will mainly use:

- "gpt-4o" or "gpt-4.1-mini" (and close relatives) for OpenAI chat models via `ChatOpenAI`.
- "gemini-2.5-pro" or "gemini-2.5-flash" (and their successors) for Google Gemini chat models via `ChatGoogleGenerativeAI`.

Switching model changes capacity, latency, and cost, but all the other parameters described below keep the same meaning.



Throughout this book, the examples use mostly OpenAI’s “gpt-4.1-mini” but the workflows are model-agnostic: you can run the same examples with Google AI’s “gemini-2.5-flash” by changing only the model initialization code.

5.1.4.2 Temperature

Temperature controls how much randomness the model uses when generating text. Lower values produce more consistent, repeatable outputs; higher values increase variety and are more suitable for creative tasks. Temperatures around 0.0–0.3 are emphasized for enterprise use cases where consistency and factual accuracy matter, while higher temperatures (for example 0.7 and above) are framed as better for creative writing and brainstorming. In LangChain, temperature is exposed on chat model wrappers, but the typical numeric range depends on the provider. Guidance distinguishes 0.0–2.0 for OpenAI, 0.0–1.0 for Anthropic-style models and Gemini.

```
from langchain_openai import ChatOpenAI

# Factual, consistent responses (support / Q&A)
factual_llm = ChatOpenAI(
    model = "gpt-4.1-mini",
    temperature = 0.1,
    max_tokens = 256
)

# More exploratory / creative responses (brainstorming)
creative_llm = ChatOpenAI(
    model = "gpt-4.1-mini",
    temperature = 0.8,
    max_tokens = 512
)
```

5.1.4.3 Top-p (nucleus sampling)

Top-p controls how wide the set of “allowed” next tokens is. At each step, the model has many possible next tokens with different probabilities. Top-p says: “only look at the most likely tokens until their total probability reaches this value; ignore the rest.”

In practice the model sorts tokens from most to least likely, starts adding them to a pool, and stops when the sum of their probabilities reaches top_p. Sampling then happens only inside that pool. If the model is very confident, a few tokens are enough to reach the threshold, so the pool is small. If the model is uncertain, it needs more tokens to reach the same threshold, so the pool grows automatically.

Top_p is expressed on a 0.0–1.0 scale. Lower values make output more focused by limiting sampling to a smaller probability mass, while higher values allow a broader set of tokens and can increase variety.

5.1.4.4 Top-k

Top-k controls how many of the best-scoring tokens the model is allowed to pick from. At each step, the model has a list of possible next tokens ordered from “most likely” to “least likely”. Top-k tells it: “only consider the first k items in this list and ignore the rest”. In practice:

- The model sorts tokens by probability.
- It keeps only the first k tokens.
- It chooses the next token by sampling inside that shortened list.

LangChain’s guidance presents top_k as an integer control with a typical range such as 1–100, where smaller values constrain generation more tightly and larger values allow more variety.

A small k means the model is tightly constrained. It can still choose between several options, but they must all be among its top favourites. The text is usually stable and focused, with some small variation in phrasing. A large k means the model is allowed to look further down the

list. This gives it more freedom and can lead to more varied, sometimes surprising, continuations. It can also increase the chance of slightly odd or off-topic wording, because lower-ranked tokens are more often selected.

5.1.4.5 *Interaction between temperature, top-p, and top-k*

Temperature, top-p, and top-k all act on the same thing: the model's belief about the next token. The usual order is:

- The model computes probabilities for all tokens
- Temperature makes that distribution sharper or flatter
- Top-p and/or top-k cut off the low-priority tokens
- Sampling picks one token from what remains

Changing any of them changes how "bold" or "cautious" the model is. Changing several at once can make behaviour hard to predict. You can think in terms of a few typical regimes:

1. Low temperature + strong truncation

Example: temperature around 0.1–0.2 with low top_p (for example 0.7–0.8) or small top_k.

The model mostly follows its top choices and ignores the long tail. This is useful for structured extraction, classification, and tightly controlled answers such as order status, policy explanations, or numeric trading summaries.

2. Medium temperature + moderate top-p

Example: temperature around 0.3–0.6 with top_p around 0.9 and no explicit top_k.

The model stays reasonably focused but has room to vary wording and structure. This is a good default for most assistants: customer support replies, product recommendations, and high-level market commentary grounded in retrieved data.

3. High temperature + weak truncation

Example: temperature above 0.7 with top_p near 1.0 and large or no top_k.

The model can choose from many options and behaves much more creatively but also makes more mistakes and goes off track more easily. This should only be used for clearly bounded creative tasks (for example, generating alternative marketing copy from a fixed product description), not for anything safety- or money-critical.

Because they all push and pull on the same distribution, a simple tuning strategy works best:

- Keep top_p (and top_k, if you use it) at conservative values for a given application
- Change temperature first and observe how answers change
- Only tighten or loosen top_p/top_k when temperature alone cannot reach the balance of stability and diversity you want

Later chapters follow this pattern: flows that must be predictable mostly vary temperature in a narrow band, while more creative branches experiment with modest changes to both temperature and truncation, always starting from a stable baseline.

5.1.4.6 *Number of candidates (n / candidate_count)*

This parameter controls how many different answers the model should generate for the same input in a single call.

- On OpenAI chat the parameter is called 'n'.

- On Gemini the underlying parameter is usually `candidate_count`, and in LangChain you set it as `n` on the chat wrapper.

For example, if $n=3$, the provider will sample three independent answers from the same probability distribution: same prompt, same temperature, same top-p/top-k, but three different “rolls of the dice”. Each of these answers uses tokens and therefore increases latency and cost. The key point is that multiple candidates only help if you have a plan for what to do with them. Typical strategies are:

1. Self-consistency or best-of-N reasoning: You ask the model to think through a problem several times and then:
 - Pick the most common answer (for classification or simple numeric results), or
 - Select the best explanation according to a separate check (for example, the one that mentions all required constraints).
2. Diversity-first, then re-ranking: You first generate several variations, then use another step (model or classic scoring function) to choose the best, for example:
 - E-commerce: generate three product description variants, then pick the one that scores higher on length, keywords, and tone.
 - Trading: generate multiple summaries of the same market view and keep the one that covers all target instruments and risk points.
 - Support: generate a few candidates replies to a complex complaint and select the one that matches internal policy rules.

Without this second step (voting, scoring, filtering), setting $n > 1$ usually just wastes tokens, because you still show only one answer to the user. For single-response assistants in this book the default is to leave $n = 1$. We raise `n` only in specific flows where:

- We explicitly compare several candidates, or
- We need more than one output (for example, a list of alternative subject lines or marketing blurbs).

5.1.4.7 Repetition penalties (presence and frequency)

Repetition penalties are knobs that tell the model not to keep saying the same things over and over. Both OpenAI and Gemini expose them, and LangChain surfaces them on chat wrappers. The exact ranges and internal implementation are provider-specific, but you configure them through two parameters:

- `presence_penalty`
- `frequency_penalty`

Both look at what the model has already generated in the current answer and slightly change how attractive each possible next token is.

`presence_penalty`

Presence penalty kicks in as soon as a token has appeared at least once in the output so far. With a positive value, it gently pushes the model to introduce new words and ideas instead of circling around the same small set. A simple way to think about it:

- With `presence_penalty = 0`, the model can happily keep using the same tokens if they are still the most likely choice.

- With `presence_penalty > 0`, once a token has appeared, its score is reduced a bit for future steps, so the model is more likely to reach for a different word or phrase.

This is useful when you want the model to explore different angles or avoid repeating the same point too many times in long replies.

`frequency_penalty`

Frequency penalty looks not just at whether a token has appeared, but at how many times it has appeared. The more often a token shows up, the stronger the penalty becomes. In practice:

- Common words like “the” or “and” are usually not affected much at moderate settings, because the model still needs them for grammatical sentences.
- Repeated phrases or specific terms (“leverage synergies”, a product name, a ticker symbol) get pushed down more strongly if they are overused.

Compared with presence penalty, frequency penalty is more about reducing obvious repetition (“the model the model the model...”) than forcing entirely new topics. It is good for keeping long answers readable and less monotonous.

How to use them in real systems

Both penalties take values in a small numeric range, typically between -2.0 and 2.0. In this book we treat them as fine-tuning tools, not primary controls:

- We start with both penalties at 0.
- If we see repeated phrases in, for example, long customer support answers, we try a small `frequency_penalty` (for example 0.1–0.3).
- If we want several clearly different alternatives (for example, three distinct trading scenarios), we may add a small `presence_penalty` as well, to encourage the model to pick different wording and angles across candidates.

Large positive penalties can backfire. If you push them too high, the model starts avoiding important tokens that should appear several times: product names, order numbers, ticker symbols, key technical terms. Answers then become vague or oddly phrased because the model is “afraid” to repeat the very words that carry meaning. For that reason, throughout the book we only use small positive values, and only in flows where we have seen repetition problems during testing. For everything else, we leave both penalties at zero and rely on good prompts, retrieval, and tools to keep answers clear and on-topic.

5.1.4.8 `Max tokens / max_output_tokens`

Max-tokens settings control length, not “word choice”. They set an upper bound on how many tokens the model is allowed to generate before it must stop. In chat models, `max_tokens` (or `max_output_tokens`) applies to the response you are asking the model to produce. The model still has a fixed context window. Each request must fit inside it:

- input tokens (system + user + prior messages + any retrieved context), plus
- output tokens (the model’s reply)

If input + max output exceeds the context window, the request can fail. If you set the max output very low, the model may stop mid-sentence because it has hit the ceiling.

It is tempting to request one large, fully-formed answer by setting a high `max_tokens`. That is simple (one call, one response), but it reduces your control:

- All-or-nothing failure: if the call times out, hits a context limit, or returns malformed output, you lose the entire result and have to rerun everything.
- Quality drift: the longer the generation, the more likely the model is to wander, repeat itself, or lose track of earlier constraints.
- Fewer checkpoints: you only see the final output, so you cannot validate or correct intermediate steps (structure, factual grounding, formatting) before the model continues.

When you care about reliability, prefer splitting the work into smaller steps with tighter limits.

For example, for a trading report:

- Step 1: generate an outline (short, constrained output).
- Step 2: generate each section separately (each with its own length and formatting rules).
- Step 3: optionally stitch sections together or produce a short executive summary.

This approach gives you better control over length and structure, keeps each prompt focused, and localises failures: if one section is weak or fails, you rerun only that section instead of repeating the entire report. It also helps you stay comfortably within the context window, because each request carries only the context that section needs, rather than pushing one oversized prompt-response pair to the limit.

5.1.4.9 Stop sequences

Stop sequences tell the model where the answer should end, regardless of how many tokens it could still generate. Instead of relying only on `max_tokens`, you give the model explicit “end markers” and ask it to stop as soon as one of them appears in the output.

In OpenAI chat models this is done with the `stop` parameter, which can be a single string or a list of strings. Generation stops when any of these strings is produced. In Gemini, the generation configuration exposes `stop_sequences`; `ChatGoogleGenerativeAI` passes its `stop` argument through to that field, so you use it in the same way.

Stop sequences are most useful when you need a clean, machine-friendly boundary:

- producing exactly one JSON object or one structured record, then stopping instead of adding comments or extra text
- separating sections that downstream code will parse (for example, “everything before the marker is the summary, everything after is ignored”). For example: `stop=["###END_SUMMARY##"]`
- chaining several generations, where each step is expected to output one logical “unit” such as a single SQL query, a single prompt, or a single tool call. For example: `stop=[";"]`

Common choices are markers that are unlikely to appear in normal text, such as a line with ‘END’, a closing fence like `}` or a specific delimiter string. In JSON-style outputs you often use the final `}` as a stop sequence so the model does not append extra text after the object.

Stop sequences only control where the model stops; they do not filter or “clean” what appears before the marker. The model can still produce undesired content inside the allowed span. Their role is to guarantee that, once the marker appears, generation ends and downstream components see a well-bounded chunk of text they can safely parse or forward.

5.1.4.10 Operational parameters (timeout and retries)

Timeout and retry settings do not change what the model writes. They decide how your system behaves when the model is slow or the provider has a temporary problem. In real applications, these details matter as much as the sampling parameters.

A timeout is the maximum time you are willing to wait for a response before treating the call as failed. Both ChatOpenAI and ChatGoogleGenerativeAI accept a `timeout` value. If the model does not answer within that window, LangChain raises an error instead of waiting forever.

Retries help you handle short-lived problems such as network hiccups, rate limits, or brief provider-side errors. The `max_retries` parameter tells LangChain how many times it should automatically re-run a failed call before giving up. A small number of retries (for example 2 or 3) with back-off is usually enough to smooth out transient issues.

The key is to treat timeouts and retries as part of your system design, not as last-minute tuning. A reasonable pattern is:

- Pick a timeout that matches the user's patience for that specific step. Simple classification or routing nodes get short timeouts; complex multi-document reasoning nodes get longer, but still bounded, timeouts
- Configure `max_retries` to handle brief spikes or timeouts, not to hide ongoing incidents. If all retries fail, the application layer should detect this and move to a fallback path
- Combine this with circuit-breaker logic or feature flags at the application level so you can temporarily disable expensive or fragile flows without bringing down the whole experience.

5.1.4.11 Putting it together

All the parameters in this section define a decoding policy: a clear set of rules for how the model turns its internal scores into a final piece of text, and how your system handles that process operationally. On the probabilistic side:

- Temperature and top-p (and top-k, where available) shape how bold or conservative the model is when it picks the next token.
- Repetition penalties nudge the model away from repeating itself too much.
- Max-tokens and stop sequences define how long the model is allowed to talk and where it should stop.

On the operational side Timeout and retries define how long you wait for a response and how you cope with temporary failures. Later chapters rely on this view in concrete ways:

- Branches that must be stable and verifiable, like order status explanations or numeric market summaries, use low temperature, conservative truncation, modest or no penalties, and strict timeouts.
- Branches that exist to add variation, such as alternative product pitches grounded in the same catalogue data, use slightly higher temperature and, when needed, adjusted top-p, but still respect length limits and timeouts.
- Flows that need diversity by design, such as generating several trading scenarios and then ranking them, enable multiple candidates or slightly stronger penalties on purpose, not as a vague "be more creative" switch.

Once you see decoding and operational parameters as a single policy, you can reason about them like any other part of your system design: different nodes in a LangGraph, or different LCEL chains, can use different policies that match their role, instead of relying on one global “randomness” setting for everything.

5.2 PROMPT TEMPLATES AND CHAT PROMPT PATTERNS (SYSTEM, USER, ASSISTANT)

Prompt templates define the instructions and structure your application sends to the model. They set the assistant's role, the task to perform, what inputs it can rely on, and the shape of the output your code expects. In a throwaway notebook you can build them with string concatenation. In a real system you need something more disciplined: prompts must be explicit, testable, and easy to change without touching business logic. LangChain gives you two core abstractions for this layer:

- `PromptTemplate` for single-string prompts.
- `ChatPromptTemplate` for multi-message chat prompts with roles such as system, user, and assistant.

Both are prompt templates: reusable bits of text with placeholders (named variables). When you run them, you provide a dictionary of values, LangChain fills in the placeholders, and the finished prompt is passed to the model or to the next step in an LCEL pipeline. This is the difference between prompt design and prompt engineering: first you design templates that are structurally correct and easy to reuse, then you iteratively tune the wording and examples to perform well on specific tasks.

Across this book, templates follow a simple structure that works well across providers: persona (who the assistant is), task (what it does), context (what information it can use), and format (how the answer should be returned). The rest of this section explains `PromptTemplate` and `ChatPromptTemplate` in turn, then shows how they support common chat usage patterns.

5.2.1 `PromptTemplate`: parameterised single-string prompts

`PromptTemplate` is the basic building block. It represents a single string with placeholders such as `{question}`, `{context}`, or `{portfolio}`. When you invoke it, LangChain fills those placeholders with concrete values and produces one final string.

If you build prompts by manually stitching strings, every route that calls the model has its own ad-hoc prompt (1 prompt, 1 template). Over time you get slightly different variants scattered through the codebase: some with disclaimers, some without; some with the latest format instructions, some outdated. Small changes become risky, and tests are hard to reason about because prompt construction is tangled with business logic. With `PromptTemplate`, you move the prompt into its own object:

- The prompt text lives in one place, close to other prompts for the same domain.
- The list of variables is explicit. If the template expects `{question}` and `{policy_snippet}`, you must provide both; missing or extra variables cause errors early.
- You can unit-test the template without calling any model. You format it with fixed inputs and assert on the output string.

Example: Customer-support with OpenAI

Support policies are usually written for legal accuracy, not for speed of execution on a live ticket. A support assistant often needs a short, agent-friendly explanation of a policy paragraph that can be pasted into an internal note or used as a quick briefing. This example

turns a policy sentence into a compact explanation under a hard length limit, using a low temperature to prioritise consistency.

What this example teaches

1. PromptTemplate keeps policy wording and formatting in one place and separates template logic from business logic.
2. LCEL composes reusable components with the pipe operator (|) into a single runnable pipeline that executes via invoke().
3. StrOutputParser converts the model's structured output into a plain string, so the calling code receives text, not a message object.

```
from .config import OPENAI_API_KEY
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

# --- Basic safety checks so failures are clear up front ---
if not OPENAI_API_KEY:
    raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

# ChatOpenAI wrapper for modern chat models
policy_llm = ChatOpenAI(
    model = "gpt-4.1-mini",
    temperature = 0.0
)

policy_prompt = PromptTemplate.from_template(
    """
    You are a customer support helper for ACME Telecom.

    Task:
    Explain the policy below to a new support agent in simple language.
    Keep the explanation under 120 words.

    Policy:
    {policy_text}
    """
)

# LCEL pipeline: PromptTemplate → ChatOpenAI
policy_chain = policy_prompt | policy_llm

result = policy_chain.invoke(
    {
        "policy_text": ("Customers can cancel a new broadband contract within 14 "
                       "days of activation with no penalty. After 14 days, "
                       "an early termination fee applies.")
    }
)

# ChatOpenAI returns an AIMessage; print only the text content
print(result.content)
```

ch_05\scr\parameterised_single_string_PromptTemplate.py

PromptTemplate.from_template(...) defines the prompt once and fills variables at runtime. Templates are positioned as a maintainability tool because you can change prompt wording in one place and keep business code passing structured inputs.

policy_chain = policy_prompt | policy_llm | StrOutputParser() creates a sequential LCEL pipeline. The pipe operator constructs a RunnableSequence where each component's output becomes the next component's input.

What happens when policy_chain.invoke(...) runs:

1. invoke receives the input dictionary with policy_text. invoke is the standard synchronous execution method for Runnable components.

2. PromptTemplate formats the prompt by substituting {policy_text} with the provided policy.

3. ChatOpenAI is invoked with the formatted prompt as the next stage in the sequence.

4. StrOutputParser extracts the string response, returning plain text to the caller.

The calling code never needs to manipulate prompt strings. It passes structured inputs into a stable interface (`invoke`) and receives a predictable output type (a string), while prompt text remains centralised and replaceable without changing application control flow.

As prompts grow, some parts will barely change (brand name, disclaimers), while others vary per use case. LangChain lets you partially apply templates and compose them:

```
from .config import OPENAI_API_KEY
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

# --- Basic safety checks so failures are clear up front ---
if not OPENAI_API_KEY:
    raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

# ChatOpenAI wrapper for modern chat models
policy_llm = ChatOpenAI(
    model = "gpt-4.1-mini",
    temperature = 0.0
)

base_prompt = PromptTemplate.from_template(
    """
    You are a customer support helper for {brand_name}.

    Task:
    Explain the policy below to a new support agent in simple language.
    Keep the explanation under 120 words.

    Policy:
    {policy_text}
    """
)

policy_prompt = base_prompt.partial(brand_name = "ACME Telecom")

# LCEL pipeline: PromptTemplate → ChatOpenAI
policy_chain = policy_prompt | policy_llm

result = policy_chain.invoke(
    {
        "policy_text": (
            "Customers can cancel a new broadband contract within 14 days of activation "
            "with no penalty. After 14 days, an early termination fee applies."
        )
    }
)

# ChatOpenAI returns an AIMessage; print only the text content
print(result.content)
```

ch_05\scr\partial_composition_PromptTemplate.py

In this case, `base_prompt` is a `PromptTemplate` with two variables: `brand_name` and `policy_text`. The template text is the stable “shared contract” for this style of support output: persona, task framing, length limit, and the slot where policy text is inserted.

`policy_prompt = base_prompt.partial(brand_name="ACME Telecom")` performs a partial substitution, producing a new prompt template with `brand_name` already fixed. `PromptTemplate.partial` is a way to partially format a template before invocation (including the ability to substitute with strings or functions), so later calls only need to provide the remaining variables.

Why partial prompts are worth using:

- Reduce repetition and prevent drift: keep shared instructions (like persona and formatting rules) in one base prompt and compose task-specific prompts on top, instead of copying the same text into many templates and letting small edits turn into inconsistent variants over time.
- Get a clean boundary between “organisation constants” and “runtime inputs”: `brand_name` is a constant for a given deployment or tenant; `policy_text` is per request.

LangChain’s also allows for another maintainability pattern: splitting prompts into smaller `PromptTemplate` pieces and composing them, rather than growing one monolithic string. If your support assistant later needs multiple tasks (policy explanations, refund decisions, fraud clarifications), you can keep a shared persona fragment and combine it with task-specific fragments to control reuse explicitly. This is discussed in 5.1.3.

5.2.2 ChatPromptTemplate: structured multi-message prompts

Modern LLM applications are usually chat-based. Instead of sending one raw string, you send a list of messages with roles such as system, user (or human), and assistant (or ai). The chat model then uses this whole list to generate the next assistant message.

`ChatPromptTemplate` is the chat analogue of `PromptTemplate`. It lets you define a pattern made of multiple messages, each with a role and optional variables. When you invoke it, you pass a dictionary of values and get back a list of formatted messages ready for a chat model.

In LangChain and most chat APIs roles are:

- The system message sets global behaviour and guardrails. This is where you fix the persona, style, and safety constraints.
- Human (or user) messages contain user questions, instructions, or data to process.
- AI (or assistant) messages can contain previous model replies or examples of good behaviour.

Internally, providers still convert this to a single prompt for the model, but you work with explicit roles. This separation matches how you think about assistance in real systems: a stable “how to behave” layer, plus changing user inputs and optional examples.

To demonstrate this concept, consider a classic first-line support assistant that usually needs two inputs at the same time: the customer’s question and operational context (for example, account status, recent incidents, or loyalty tier). Chat models accept a list of structured messages with roles and content, which lets you separate “behaviour” instructions (system message) from “case data” (human message). LangChain’s chat prompt templates are designed to build that message list from variables, so call sites pass a clean dictionary instead of constructing prompts manually.

```
from .config import OPENAI_API_KEY
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

# --- Basic safety checks so failures are clear up front ---
if not OPENAI_API_KEY:
    raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

# ChatOpenAI wrapper for modern chat models
support_chat_model = ChatOpenAI(
    model = "gpt-4.1-mini",
```

```

        temperature = 0.0
    }

support_chat_prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            (
                "You are a first-line customer support assistant for ACME Telecom. "
                "Answer clearly and politely. If you are not sure, say so and suggest "
                "how the customer can get help from a human agent."
            ),
        ),
        (
            "human",
            "Customer question: {question}\n\nAccount flags: {account_flags}",
        )
    ]
)

support_chat_chain = support_chat_prompt | support_chat_model
reply = support_chat_chain.invoke(
    {
        "question": "My broadband is down. Can I get a refund for today's outage?",
        "account_flags": "Customer has had 3 outages in 30 days; loyalty tier: Gold."
    }
)
print(reply.content)

```

ch_05\scr\parameterised_single_string_ChatPromptTemplate.py

`support_chat_model` is a `ChatOpenAI` instance configured with `temperature=0.0` to reduce variation across runs. The LangChain reference explicitly recommends low temperatures for applications that prioritise consistency and accuracy.

`support_chat_prompt` is built with `ChatPromptTemplate.from_messages(...)` using two messages:

- A system message that defines the assistant’s role and behavioural constraints.
- A human message that carries the structured input fields `question` and `account_flags` via placeholders.

What happens when `support_chat_chain.invoke(...)` runs:

1. The chain receives a dictionary containing `question` and `account_flags`.
2. `ChatPromptTemplate` substitutes the placeholders in the human message and produces a list of messages: one system message and one human message.
3. `ChatOpenAI` receives that list and calls the chat model.
4. The reply is an `AIMessage`; printing `reply.content` prints only the text content.

5.2.3 Composing templates inside a chat template

Chat models work best when you send them a structured list of messages, typically a system message to set rules and context, followed by human and (optionally) assistant messages. In real projects, the system part is often something you want to reuse: a standard “rules and behavior” block that you share across multiple assistants or switch on and off with configuration. You can do that without giving up the chat-message structure.

The trick is simple: keep your reusable rules as a normal `PromptTemplate`, then reuse its raw template text as the content of the system message inside a `ChatPromptTemplate`. You are

not formatting the prompt twice. You are still building one single ChatPromptTemplate; you're just assembling it from smaller pieces you can maintain independently.

Concretely, imagine your reusable system rules template contains placeholders like {a} and {b}. You then create a ChatPromptTemplate with that system message (using the same template text), plus a human message you define in the chat template, and perhaps an assistant-message template that includes {c}. When you invoke the ChatPromptTemplate, you pass one input dictionary, and the template fills {a}, {b}, and {c} in one pass. The output is a clean, ready-to-send sequence of structured chat messages, with the benefits of both worlds: reusable building blocks and the message format chat models expect.

```
from langchain_core.prompts import PromptTemplate, ChatPromptTemplate
# A reusable system instruction block expressed as a PromptTemplate.
system_prompt_template = PromptTemplate.from_template("a: {a} b: {b}")
# Compose it into a multi-message chat prompt.
chat_prompt_template = ChatPromptTemplate.from_messages(
    [
        ("system", system_prompt_template.template),
        ("human", "hi"),
        ("ai", "{c}")
    ]
)
# invoke() returns a prompt value object; access .messages to get the list.
prompt_value = chat_prompt_template.invoke({"a": "a", "b": "b", "c": "c"})
messages = prompt_value.messages
# messages is now a list of structured chat messages in the right order.
# In a real chain you would pass it to a chat model.
print(len(messages))
print(messages[0].content)
print(messages[1].content)
print(messages[2].content)
```

ch_05\scr\chaining_prompt_templates.py

```
(.venv) C:\Users\alexc\llm-app\ch_05> python -m
src.Chaining_prompt_templates

3
a: a b: b
hi
c].content)
```

One small engineering rule keeps this pattern safe and easy to maintain: use PromptTemplate.template only as reusable text and do all the actual formatting at the very last step, in the outermost prompt you invoke. In other words, a ‘PromptTemplate’ should behave like a “prompt fragment” you can plug into bigger prompts, not something you format early and then stitch together. If you keep that boundary at the outside, your code stays simple because every call site still passes one single dictionary of variables, and the whole assembled prompt remains readable and predictable even when it’s built from several smaller pieces.

5.2.4 Chat history and `MessagesPlaceholder`

Most assistants are multi-turn. A customer keeps asking follow-up questions; a shopper adds products to a cart; a trader refines a query about positions. To make these flows feel coherent, the model needs some form of chat history. You can always build that history manually in

your code, by concatenating past messages into a big string, but this becomes error-prone and hard to reuse. LangChain solves this with `MessagesPlaceholder`.

`MessagesPlaceholder("history")` creates a “slot” in the chat template where a list of messages will be injected at runtime. At each call you decide which and how many messages to pass, and the placeholder expands them into separate messages in the right position.

Example: Trading assistant example with Gemini

A trading assistant is usually multi-turn: the user asks for a concept (diversification), then asks how it applies to their portfolio. The assistant needs prior context to stay coherent, but “memory” should be a runtime input, not hard-coded into the prompt. `MessagesPlaceholder` gives you a dedicated slot where you can inject history, so later you can swap history policies (window, summary, topic-filter) without rewriting the prompt template.

```
from .config import GOOGLE_API_KEY
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_google_genai import ChatGoogleGenerativeAI

# --- Basic safety checks so failures are clear up front ---
if not GOOGLE_API_KEY:
    raise RuntimeError("GOOGLE_API_KEY is not set. Check your .env file.")

trading_model = ChatGoogleGenerativeAI(model = "gemini-2.5-flash", temperature = 0.2)
prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a trading assistant. Explain risks in plain language."),
        (MessagesPlaceholder("history"),),
        ("human", "{question}")
    ]
)

chain = prompt | trading_model

# Your full conversation store (could come from anywhere)
full_history = [
    ("human", "What is diversification?"),
    ("ai", "It means spreading investments across different assets..."),
    ("human", "What is volatility?"),
    ("ai", "Volatility is how much prices move up and down...")
]

def build_history(full_history, mode: str):
    if mode == "none":
        return []
    if mode == "last_turn":
        return full_history[-2:] # last human+ai pair
    if mode == "only_user":
        return [m for m in full_history if m[0] == "human"]
    if mode == "last_4_messages":
        return full_history[-4:]
    return full_history # default: everything

# Call 1: no history
answer1 = chain.invoke(
    {"history": build_history(full_history, "none"), "question": "Is 60% tech risky?"})
print(answer1.content)

# Call 2: include last turn only
answer2 = chain.invoke(
    {"history": build_history(full_history, "last_turn"), "question": "And what about
ETFs?"})
print(answer2.content)

# Call 3: include everything
answer3 = chain.invoke(
    {"history": build_history(full_history, "all"), "question": "Summarize what we've
covered."})
print(answer3.content)
```

ch_05\scr\ChatPromptTemplate_and_MessagePlaceHolder.py

This example uses `MessagesPlaceholder("history")` to make conversation context a runtime choice rather than something baked into the prompt. The application keeps a full transcript (`full_history`), that can come for example from a database but does not always send it to the model. Instead, `build_history()` selects what to include for each request: no context, only the last user/assistant turn, only user messages, the last N messages, or the entire transcript. Each `invoke()` call passes a different history value, and LangChain expands that list into separate chat messages exactly where the placeholder sits. This keeps prompts stable while making context selection explicit, testable, and easy to tune.

5.2.5 RunnableWithMessageHistory: making chat history a runtime concern

Passing history explicitly as an input variable is a clean first step, because it forces you to treat “memory” as data. The practical downside shows up as soon as you build a real chat endpoint: every caller must fetch the right history, format it, pass it in under the correct key, and then remember to persist the new turn after the model replies. That is a lot of plumbing to repeat, and it tends to leak into places where you would rather keep your application logic focused on the task.

`RunnableWithMessageHistory` is a wrapper runnable that takes a normal LCEL chain and adds two responsibilities around it: before the chain runs it loads the current conversation messages for a session, and after the chain runs it appends the new user and assistant messages back into that same session history. The wrapped chain stays simple; the “where does history come from and where does it go” policy lives in one place. The wrapper is session-scoped. At invocation time you pass a session identifier through runtime configuration (`config.configurable.session_id`). If the session id is missing, the wrapper cannot resolve which history to load and will fail fast rather than silently mixing conversations.

```
from __future__ import annotations
from .config import GOOGLE_API_KEY
from typing import Dict
from langchain_core.chat_history import InMemoryChatMessageHistory
from langchain_core.prompts import ChatPromptTemplate, MessagesPlaceholder
from langchain_core.runnables.history import RunnableWithMessageHistory
from langchain_openai import ChatOpenAI

# --- Basic safety checks so failures are clear up front ---
if not GOOGLE_API_KEY:
    raise RuntimeError("GOOGLE_API_KEY is not set. Check your .env file.")

# Simple in-memory history store keyed by session_id.
# Replace this with a database-backed implementation when you need durability.
_sessions: Dict[str, InMemoryChatMessageHistory] = {}

def get_session_history(session_id: str) -> InMemoryChatMessageHistory:
    if session_id not in _sessions:
        _sessions[session_id] = InMemoryChatMessageHistory()
    return _sessions[session_id]

def print_history(session_id: str) -> None:
    # Pretty-print the stored chat history for a session.
    history = get_session_history(session_id)
    print("\n==== InMemoryChatMessageHistory ====")
    print("Session:", session_id)
    print("Message count:", len(history.messages))
    for i, m in enumerate(history.messages, start = 1):
        # m is a BaseMessage (HumanMessage / AIMessage / SystemMessage)
        role = m.type # "human", "ai", "system"
        content = m.content
        print(f"{i:02d}. {role}: {content}")
    print("==== end ====\n")
```

```

prompt = ChatPromptTemplate.from_messages(
    [
        ("system", "You are a concise support assistant. Use the conversation history."),
        MessagesPlaceholder("history"),
        ("human", "{input}")
    ]
)

model = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.2)
base_chain = prompt | model

chat_chain = RunnableWithMessageHistory(
    base_chain,
    get_session_history,
    input_messages_key = "input",
    history_messages_key = "history"
)

if __name__ == "__main__":
    cfg = {"configurable": {"session_id": "acme-ticket-1042"}}

    r1 = chat_chain.invoke({"input": "Hi. My dashboard shows a sudden drop in active
                           users."}, config = cfg)
    r2 = chat_chain.invoke({"input": "What are the first two checks you would run?"},
                           config = cfg)

    # The history object is outside the chain and can be inspected in tests.
    print("Stored message count:", len(get_session_history("acme-ticket-1042").messages))
    print("Last reply:", r2.content)

    # Print the whole history at the end
    print_history(session_id)

```

[ch_05\scr\RunnableWithMessageHistory.py](#)

`InMemoryChatMessageHistory` is the concrete history container. It holds a list of structured chat messages (human, ai, system) in process memory. The code keeps one history object per `session_id`, so each conversation thread has its own isolated transcript. Because it is in-memory, the history disappears when the process restarts; that makes it ideal for local development, demos, unit tests, and short-lived sessions. In production you usually replace it with a durable store, but you keep the same interface.

`RunnableWithMessageHistory` is the glue. It wraps the base chain (`prompt → model`) and automatically performs two jobs on every invocation. First, it fetches the correct history for the current session by calling `get_session_history(session_id)`. Second, it injects that history into the prompt at `MessagesPlaceholder("history")`, so the model always receives the prior turns in the right format. After the model responds, the wrapper appends both the new user message, and the assistant reply back into the same history object.

5.2.6 RunnableWithMessageHistory: persistence options

`RunnableWithMessageHistory` is designed to work with any persistence layer, as long as the history object you provide follows the `BaseChatMessageHistory` interface. The recommended production approach is to use a persistent chat history implementation (rather than an in-memory one) and plug it into `RunnableWithMessageHistory` via your session-history factory function.

Common persistent options include `RedisChatMessageHistory`, which stores the transcript in Redis and can apply a TTL (automatic expiry), and `SQLChatMessageHistory`, which writes messages to an SQL database using SQLAlchemy under the hood—so it works with whatever databases your SQLAlchemy connection string and driver support (for

example SQLite locally, or Postgres/MySQL in production). For Postgres specifically points to `PostgresChatMessageHistory` in the `langchain_postgres` package; the older community implementation is deprecated.

To add persistence, you keep `RunnableWithMessageHistory` exactly the same and only change `get_session_history(session_id)` to return a persistent `BaseChatMessageHistory` implementation instead of `InMemoryChatMessageHistory`. For example, with Redis:

```
from langchain_community.chat_message_histories import RedisChatMessageHistory
def get_session_history(session_id: str) -> RedisChatMessageHistory:
    return RedisChatMessageHistory(
        session_id = session_id,
        url = "redis://localhost:6379/0",
        key_prefix = "myapp:",
        ttl = 60 * 60 * 24 # optional
    )
```

If none of the built-in backends fit your needs, you can implement your own storage class by implementing `BaseChatMessageHistory` and returning it from the same factory function.

5.2.7 Structuring prompts: persona, task, context, format

Regardless of whether you use `PromptTemplate` or `ChatPromptTemplate`, robust prompts tend to share the same internal structure:

- Persona describes who the assistant is: support agent, product search helper, trading explainer.
- Task describes what to do: answer, classify, extract, summarise, or generate a plan.
- Context is the data the model is allowed to use: snippets from policies, product catalog sections, position lists, retrieved documents.
- Format describes how the answer should look: paragraphs, bullet points, JSON, or a custom schema.

In chat prompts, persona and some task instructions usually go into the system message. Context and detailed instructions often live in human messages with clear delimiters so the model can see where external data starts and ends. This pattern reduces ambiguity and is widely recommended for prompt robustness.

Example: E-commerce with explicit context

Here is a simple e-commerce search helper that uses these ideas. A catalog helper must recommend only products you actually sell. The simplest guardrail is to pass “retrieved catalog context” alongside the user query and instruct the model to use only that context. The prompt keeps the “shape” of inputs stable (`user_query` plus `catalog_context`), so you can change how `catalog_context` is produced later (database query, API call, vector retrieval) without rewriting the prompt.

```
from .config import OPENAI_API_KEY
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

# --- Basic safety checks so failures are clear up front ---
if not OPENAI_API_KEY:
    raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

# ChatOpenAI wrapper for modern chat models
catalog_model = ChatOpenAI(
```

```

        model = "gpt-4.1-mini",
        temperature = 0.0
    )

catalog_prompt = ChatPromptTemplate.from_messages(
    [
        (
            "system",
            (
                "You are a product search assistant for ACME Shop."
                "Help users find products in the catalog and explain trade-offs."
                "If a product is not in the catalog context, do not invent it."
            ),
        ),
        (
            "human",
            """
                User query:
                {user_query}

                <CATALOG_CONTEXT>
                {catalog_context}
                </CATALOG_CONTEXT>

                Task:
                1. Use only products mentioned in CATALOG_CONTEXT.
                2. Suggest up to three options that fit the user query.
                3. Explain the differences in one short paragraph.
            """
        ),
    ],
)

support_chat_chain = catalog_prompt | catalog_model

# Simple fake catalog context - this could come from API call or a DB query
catalog_context = """
- ACME UltraLight 14" Laptop, 16 GB RAM, 512 GB SSD, ideal for travel and office work.
- ACME CreatorPro 15" Laptop, 32 GB RAM, 1 TB SSD, dedicated GPU, designed for video
editing and 3D work.
- ACME BudgetBook 13" Laptop, 8 GB RAM, 256 GB SSD, good for web browsing and email.
"""

user_query = "I need a laptop that can handle video editing and some light gaming."
result = support_chat_chain.invoke(
    {
        "user_query": user_query,
        "catalog_context": catalog_context
    }
)

# result is an AIMessage from ChatOpenAI
print(result.content)

```

ch_05\scr\personas_task_ChatPromptTemplate.py

`ChatPromptTemplate.from_messages` builds a two-message prompt: a system message sets the assistant's role and the “no invention” rule, and a human message supplies the runtime variables. The human message includes both `user_query` and `catalog_context`, with the catalog wrapped in explicit tags. Delimiters like XML tags are a recommended way to clearly separate sections of a prompt, and they tend to improve adherence to the boundaries of “context vs instructions.” The LCEL pipe (`catalog_prompt | catalog_model`) creates a runnable sequence: `invoke` formats the messages with the provided variables, then calls the chat model and returns an `AIMessage`. By keeping `catalog_context` as a single variable, you can swap the retrieval mechanism later without changing the prompt’s structure.

The delimiters (`<CATALOG_CONTEXT>...</CATALOG_CONTEXT>`) prevent the model from confusing catalog data with instructions or the user query. If later you switch from one retrieval strategy to another (for example using a RESTful call or a Database query), you still

fill the same `catalog_context` variable with text: the template and its structure do not change.

5.2.8 Templates as first-class LCEL components

Both `PromptTemplate` and `ChatPromptTemplate` implement the same runnable interface as models, retrievers, and output parsers. They support `.invoke`, `.batch`, and `.stream`, and they compose with the `|` operator in LCEL. That means they appear as separate steps in traces and can be evaluated as independent components in your workflows. This has two practical consequences:

1. Templates can be treated as nodes in a graph: inputs flow in, formatted prompts flow out.
2. You can keep templates declarative and move all dynamic data wiring into the LCEL graph around them.

Suppose you build a trading risk explainer that uses retrieval. The prompt template itself only declares two variables, `context` and `question`:

```
from .config import OPENAI_API_KEY
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough, RunnableLambda
from langchain_openai import ChatOpenAI

# --- Basic safety checks so failures are clear up front ---
if not OPENAI_API_KEY:
    raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

# ChatOpenAI wrapper for modern chat models
analysis_model = ChatOpenAI(
    model = "gpt-4.1-mini",
    temperature = 0.0,
)

# --- Minimal "retriever" so the example actually runs ---
portfolio_context = """
- 40% in Large European bank stocks
- 25% in broad market index ETFs
- 20% in medium-duration government bonds
- 15% in cash and money-market instruments

Bank stocks are more volatile than bonds and broad market ETFs.
If bank stocks fall sharply, overall portfolio value can drop,
especially in the short term.
"""

retriever = RunnableLambda(lambda question: portfolio_context)

analysis_prompt = ChatPromptTemplate.from_messages([
    (
        "system",
        (
            "You are a trading assistant. Use the CONTEXT section to explain"
            "portfolio risks in plain language. Do not give personalised advice."
        ),
    ),
    (
        "human",
        """
        CONTEXT:
        {context}

        QUESTION:
        {question}
        """
    ),
])
# Dict input:
```

```

# - "question" = whatever the user passes in
# - "context" = retriever(question)
analysis_chain = (
    {
        "question": RunnablePassthrough(),
        "context": retriever,
    }
    | analysis_prompt
    | analysis_model
)

if __name__ == "__main__":
    user_question = "How sensitive is my portfolio to a sudden fall in bank stocks?"
    response = analysis_chain.invoke(user_question)

    # response is an AIMessage
    print(response.content)

```

ch_05\scr\dictionary_ChatPromptTemplate.py

`RunnablePassthrough()` is the “do nothing” runnable: it forwards the incoming input unchanged. In this chain, that is how the single input passed to `invoke()` becomes the “question” field “whatever comes in, expose it under the key `question`”.

`retriever` is implemented as a `RunnableLambda`. `RunnableLambda` wraps a normal Python callable so it behaves like a standard LangChain `Runnable`. That matters because it can be composed with `|`, invoked the same way as other steps, used in sync/async contexts, and integrated cleanly with LangChain tracing. In this example, the callable ignores the question and returns a constant `portfolio_context` string, but in a real system it would typically query a vector store, database, or API to produce context for that specific question.

Execution flow:

1. You call `analysis_chain.invoke(user_question)`.
2. The dict mapping is treated as a `RunnableParallel`: it sends the same input (`user_question`) down both branches (`question` and `context`) and collects their outputs into one dictionary.
3. `RunnablePassthrough` returns `user_question` → becomes `{"question": user_question}`.
4. `retriever (RunnableLambda)` returns the `context` string → becomes `{"context": portfolio_context}`.
5. LCEL merges these into `{"question": ..., "context": ...}`, passes that to `ChatPromptTemplate` for message formatting, then `ChatOpenAI` generates the final `AIMessage` answer.

Why bother (instead of manually building the dict at the call site)?

- One entry point, many data paths: callers keep a simple `.invoke(question)` interface while the chain internally handles fan-out and context fetching (and can grow into more branches later) without changing the external API.
- Swappable retrieval without touching the prompt: today the retriever is a trivial `RunnableLambda`; tomorrow it can be a real retriever or even a sub-chain, while the prompt remains stable because it only depends on receiving a “context” string.
- Uniform runnable interface: because both branches are runnables, the whole chain automatically inherits standard execution modes (`invoke/batch/stream/async`) from LCEL composition, instead of you re-implementing “single vs batch vs streaming” wiring outside the chain.

- Better tooling and observability: when you enable tracing, LangSmith can show each runnable step (question in, retrieved context out, prompt formatting, model output) as separate runs, which is much harder to see if you build dicts and strings manually outside the chain.

5.2.9 Prompt lifecycle: versioning, testing, and output validation

A prompt template helps you assemble a prompt. That's useful, but it's not the whole job. In a real system, prompts are production assets: they change over time, they can break things when they change, and other code may depend on what they produce. So you need three disciplines around prompts: versioning, testing, and output validation.

- Versioning means you treat prompts like code. When you change a prompt, you want to know what changed, when, and why. If the new prompt causes worse answers or breaks a workflow, you need to roll back to a known good version.
- Testing means you don't judge a prompt only by "it looked good in one chat". You run the prompt against a small set of representative inputs and compare the outputs. The goal is to catch regressions early: the prompt still answers correctly, still follows the rules, and still produces the format you expect.
- Output validation matters whenever software will consume the model's output. If a downstream step parses the response, stores it, or uses it as input to another component, "best effort text" is not enough. The prompt must state the expected structure (for example, a JSON object with specific keys), and the application must check that the output actually matches it. If it doesn't, you reject it or retry. In simple assistants this can be lightweight; in automated workflows it becomes a reliability requirement.

Design rule: if the output will be read by code, treat the format as a contract. Define it in the prompt and enforce it in the application.

5.2.10 Prompt Hub: shared prompts as a managed dependency

The Prompt Hub lives inside LangSmith and acts as a shared registry of prompt artifacts: a place where prompt text (and its variables) exists as a first-class object, not as a string buried inside your application code. Teams use it for the same reason they use source control: to make changes visible, reviewable, revertible, and reproducible. Each time you save a prompt, you create a new immutable version (a commit), and the hub keeps the full history so you can compare versions and roll back when a "small prompt tweak" turns into a production incident.

Because it is a LangSmith feature, the full workflow only works after you complete your LangSmith configuration. Practically, that means you have a LangSmith account and API key configured in your environment so your application can authenticate to the hub, and you have a model provider key configured so you can actually run prompts during testing. Without that setup you might be able to browse public prompts, but you cannot treat the hub as a reliable, write-enabled, versioned store for your own prompts.

Using the hub has two operations: publish a prompt and retrieve a prompt. Publishing happens when you "push" a prompt into the hub (from an interactive playground or from code). Retrieval happens when your application "pulls" a named prompt and composes it into

a chain like any other prompt template. The important production habit is version control. If you always pull “latest”, your service behavior can change without a deploy. Instead, pin a prompt by commit hash when you need strict reproducibility, or pull by a tag (for example dev, staging, prod) when you have a deliberate promotion workflow that moves the tag forward only after you have tested the change.

```
import os
from langsmith import Client
from .config import OPENAI_API_KEY
from langchain_openai import ChatOpenAI

if not OPENAI_API_KEY:
    raise RuntimeError("OPENAI_API_KEY is not set. Check your environment or .env file.")

# Pulling via LangSmith client
client = Client()

# Pull the version pointed to by a tag (e.g., "prod") OR pin a specific commit hash.
# prompt_by_tag = client.pull_prompt("joke-generator:prod")
# prompt_by_commit = client.pull_prompt("joke-generator:12344e88")
prompt = client.pull_prompt("hardkothari/prompt-maker")

llm = ChatOpenAI()

# Use the pulled prompt like any other prompt template
chain = prompt | llm
result = chain.invoke({"lazy_prompt": "Talk about Rome", "task": "Write a short, factual paragraph"})
print(result)
```

ch_05\scr\prompt_hub.py

GitHub (or equivalent) is another solid place to store prompts because it gives you diffs, reviews, and rollbacks. Prompt Hub solves a different problem: it treats prompts as a runtime-managed dependency, not just a file that ships with your code.

In Prompt Hub, each saved change becomes a committed prompt version you can reference precisely. Your application can pull a prompt by an exact version identifier when you need reproducibility, or by a movable tag such as “dev”, “staging”, or “prod” when you want a promotion workflow. That is the big operational win: you can promote a tested prompt from staging to production by moving a tag, without changing application code or redeploying.

Prompt Hub also makes iteration easier because it is built around a prompt playground: you edit, run tests, and then save the exact version you validated. From code, you push and pull prompts through the LangSmith SDK, so prompt retrieval looks like loading any other dependency, with version and metadata managed outside your deployable artifact.

The trade-off is an external dependency (auth, network, permissions). Teams typically mitigate this by pinning exact versions for strict reproducibility and using tags only through controlled promotion.

5.3 TOKEN ACCOUNTING, CALLBACKS, SPECIAL CASES AND POLICIES ENFORCEMENT

Tokens look like a small detail until they become a budget line. They drive cost, latency, and capacity at the same time, and they grow with the things engineers add almost without noticing: longer system rules, deeper retrieval, retries, fallbacks, and fan-out. A workflow that feels like “one question” can easily turn into several model calls, each with its own prompt and completion footprint.

This chapter treats tokens as an engineering signal, not a billing afterthought. The first step is measurement at the same boundaries where you make design choices: one request, one workflow run, or one expensive sub-step. Callback-based accounting gives you that boundary. You wrap execution in a `with-block` and get aggregated usage for everything that happened inside it, including inflation caused by retries and multi-step chains.

The second part looks at two “simple” chains that hide extra machinery. `LLMChain` improves numeric reliability by asking the model to translate a problem into Python and then executing the code. That turns correctness into an execution-boundary problem that you can constrain, log, and test. `SQLDatabaseChain` turns natural language into SQL, runs it against a real database, and explains the result, which makes scoping, guardrails, and inspection of intermediate steps central.

The final part applies the same engineering mindset to policy enforcement. We separate three roles—classification (moderation), revision (constitutional rewriting), and verification (self-checking)—and, for Gemini, you also learn how provider-side safety settings enforce category thresholds during generation.

5.3.1 Callback-based token accounting

Token accounting is most useful when you can measure it at the same level where you make engineering choices.

- “How many tokens did we spend today?” is mainly a finance question.
- “Which step in this workflow is using the tokens, and why?” is an engineering question.

A practical way to connect those two views is to use a callback that wraps the exact code block where model calls happen. The callback records usage for every model call inside that block and adds the numbers up. In practice, this is usually done with a context manager: you run your chain, runnable, or graph invocation inside a `with-block`. When the block ends, you read the counters. This gives you a clear, testable meter for a specific unit of work: a single request, a batch job, a full workflow run, or even one expensive sub-step you want to measure in isolation. Because the callback wraps a block of execution, it naturally includes “hidden” token inflation. If your workflow retries, falls back to another model, uses multi-step prompting, or fans out into multiple calls, the callback still sees every call. If the workflow calls the model five times, the callback records five calls and aggregates them. The callback typically returns:

- total tokens
- prompt tokens
- completion tokens

- an estimated cost (when available), derived from the model name and the usage

Prompt tokens are what you pay to ask for work: system instructions, retrieved chunks, tool results you paste into the prompt, and any other context you include. Completion tokens are what you pay for the model's output: verbosity, over-generation, and long answers.

```
from langchain_community.callbacks.manager import get_openai_callback
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY

llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

with get_openai_callback() as cb:
    response = llm.invoke("Write a two-sentence summary of why token accounting
                           matters.")
    print(response.content)

    # Aggregate usage for everything executed in the with-block
    print("prompt_tokens:", cb.prompt_tokens)
    print("completion_tokens:", cb.completion_tokens)
    print("total_tokens:", cb.total_tokens)
    print("total_cost:", cb.total_cost) # may be 0.0 if pricing metadata is unavailable
```

ch_05\scr\callback_tokencount.py

These numbers are especially useful when you change the system. If you compress context, tune retrieval, or restructure a workflow, prompt vs completion totals show whether you actually reduced cost, or just moved it around. Two cautions matter if you want honest measurements.

- First, “cost” is not a billing record. It is only an estimate. Some model identifiers may not have a pricing entry, so the estimated cost can show as zero even when token counts are non-zero.
- Second, token reporting can be incomplete in some streaming setups. You may receive streamed output while the counters stay at zero. If you are measuring usage, prefer non-streaming calls inside the instrumented block. Treat streaming as a delivery choice, not as the path you rely on for metering.

The real payoff is not the counters themselves. It is what they let you explain and control. When someone asks why a “simple” feature costs money, you can point to the driver: prompt growth, retrieval depth, retries, or a workflow structure that multiplies model calls. And when a prompt change ships and quietly doubles your token usage, you can detect it as a regression before it shows up later in your cloud bill.

5.3.2 LLMMathChain

LLMMathChain is a utility chain designed to improve a language model's performance on math-heavy questions. The idea is simple: instead of trusting the model to do arithmetic perfectly in its own head, the chain asks the model to express the problem as a small piece of Python code and then runs that code to obtain the numeric result. The final answer is then returned as plain text, but the number comes from actual computation rather than guesswork.

This pattern is especially useful when the task mixes language and calculation: unit conversions, percentage changes, compound growth, probability, or multi-step word problems where a single arithmetic slip would poison the whole response. Two practical implications follow from that design:

- The chain is only as safe as the code-execution step: treat it as an execution boundary and keep it tightly constrained.

- The chain is only as reliable as the “translate-to-code” step: you should expect occasional translation mistakes, so you want clear prompts, low temperature, and good logging.

Before executing the following example, please install `numexpr` which is a prerequisite for `LLMMathChain` to work:

```
C:\Users\yourname\llm-app> .venv\Scripts\activate
(.venv) C:\Users\yourname\llm-app> python -m pip install numexpr
```

```
import os
from langchain_classic.chains import LLMMathChain
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY
from langchain_community.callbacks.manager import get_openai_callback

def run_math_question(question: str) -> None:
    # Model configured to be deterministic: math translation benefits from low
    # randomness.
    llm = ChatOpenAI(
        model = "gpt-4o-mini",
        temperature = 0
    )

    llm_math = LLMMathChain.from_llm(llm = llm, verbose = True)

    with get_openai_callback() as cb:
        result = llm_math.invoke(question)
        print("Question:", question)
        print("Answer:", result)
        print("Tokens:", cb.total_tokens)
        print("Prompt tokens:", cb.prompt_tokens)
        print("Completion tokens:", cb.completion_tokens)
        print("Estimated cost:", cb.total_cost)

    if __name__ == "__main__":
        run_math_question("If I invest 100,000 EUR and it grows by 3.5% per year, what is the
                          value after 7 years?")
```

`ch_05\scr\LLMMathChain.py`

```
(.venv) C:\Users\alexc\llm-app\ch_05> python -m src.LLMMathChain
> Entering new LLMMathChain chain...
If I invest 100,000 EUR and it grows by 3.5% per year, what is the value after
7 years?``text
100000 * (1 + 0.035)**7
```
...numexpr.evaluate("100000 * (1 + 0.035)**7")...

Answer: 127227.9262766573
> Finished chain.
Question: If I invest 100,000 EUR and it grows by 3.5% per year, what is the
value after 7 years?
Answer: {'question': 'If I invest 100,000 EUR and it grows by 3.5% per year,
what is the value after 7 years?', 'answer': 'Answer: 127227.9262766573'}
Tokens: 268
Prompt tokens: 229
Completion tokens: 39
Estimated cost: 5.775e-05
```

### 5.3.3 SQLDatabaseChain: answering questions with SQL databases

`SQLDatabaseChain` is a packaged “question-to-database” workflow: you ask a question in natural language, the chain turns it into SQL, runs that SQL against a real database connection, then converts the returned rows into an answer. The essential trick is not the database access itself; it is the combination of a language model with an SQLAlchemy-backed connection so the same pattern works across many SQL dialects, as long as SQLAlchemy can connect to

them. The chain is built around a database handle created from a connection URI. From there, you typically scope what the model is allowed to see by restricting the tables it can inspect. This scoping matters for two reasons: it improves accuracy (less schema to confuse the model) and it reduces exposure (fewer tables that can be accidentally queried). Once instantiated, the chain accepts a free-form question and internally performs the familiar loop: translate intent into SQL, execute, then explain.

Two configuration knobs tend to matter in real projects. First, prompt customization: you can steer the SQL generation behavior (naming conventions, date handling, mandatory filters, required ordering, or “only use these tables”). Second, intermediate steps: you can request the generated SQL and the raw results alongside the final answer, which is invaluable for debugging and for building “trust but verify” UI patterns where users can inspect what was run. A pragmatic warning: SQL generation is untrusted input. Even if you use read-only credentials, you still want guardrails to prevent broad scans and expensive queries, and to block categories of statements you never want executed. At minimum: run with least-privilege credentials, allow-list tables, enforce LIMIT/timeouts, and reject write/DDL statements before execution. Treat “generated SQL” the same way you would treat “SQL copied from a random forum post,” because functionally, that’s what it is.

Before executing the following example, please install :

```
C:\Users\yourname\llm-app> .venv\Scripts\activate
(.venv) C:\Users\yourname\llm-app> python -m pip install -U langchain-community
langchain-experimental sqlalchemy
```

```
import os
import sqlite3
from typing import Any, Dict
from langchain_community.utilities import SQLAlchemy
from langchain_experimental.sql import SQLDatabaseChain
from langchain_openai import ChatOpenAI
from langchain_community.callbacks.manager import get_openai_callback
from config import OPENAI_API_KEY

def ensure_demo_db(db_path: str) -> None:
 os.makedirs(os.path.dirname(db_path), exist_ok = True)
 con = sqlite3.connect(db_path)
 try:
 cur = con.cursor()
 # Allowed table
 cur.execute(
 """
 CREATE TABLE IF NOT EXISTS deals (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 customer TEXT NOT NULL,
 discount_pct REAL NOT NULL
);
 """
)
 # A table we DO NOT want the chain to access
 cur.execute(
 """
 CREATE TABLE IF NOT EXISTS secrets (
 id INTEGER PRIMARY KEY AUTOINCREMENT,
 api_key TEXT NOT NULL
);
 """
)
 cur.execute("SELECT COUNT(*) FROM deals;")
 count, = cur.fetchone()
 if count == 0:
```

```

 cur.executemany(
 "INSERT INTO deals(customer, discount_pct) VALUES (?, ?);",
 [
 ("Globex", 10.0),
 ("Globex", 15.0),
 ("Initech", 5.0),
 ("Umbrella", 20.0)
]
)

 cur.execute("SELECT COUNT(*) FROM secrets;")
 (scount,) = cur.fetchone()
 if scount == 0:
 cur.executemany(
 "INSERT INTO secrets(api_key) VALUES (?)",
 [
 ("SUPER-SECRET-KEY-123"),
 ("TOP-SECRET-KEY-456")
]
)

 con.commit()
 finally:
 con.close()

def run(question: str) -> Dict[str, Any]:
 here = os.path.dirname(os.path.abspath(__file__))
 db_path = os.path.join(here, "demo", "deals.db")
 ensure_demo_db(db_path)

 # Restrict the database "surface area" the chain can see.
 # include_tables controls what appears in table_info (and therefore what the LLM can
 # plan against).
 db = SQLDatabase.from_uri(
 f"sqlite:///{{db_path}}",
 include_tables = ["deals"], # <--- THIS is the restriction
 sample_rows_in_table_info = 2
)

 llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

 chain = SQLDatabaseChain.from_llm(
 llm = llm,
 db = db,
 verbose = True, # prints what it's doing, including generated SQL
 return_intermediate_steps = True # lets us print the SQL cleanly
)

 out = chain.invoke({"query": question})
 return out

def main() -> None:
 question = "What is the average discount percentage for Globex, and how many deals does that include?"

 if get_openai_callback is not None:
 with get_openai_callback() as cb:
 out = run(question)

 print("\nAnswer:", out.get("result"))

 steps = out.get("intermediate_steps")
 if steps:
 print("\n--- Intermediate steps ---")
 for s in steps:
 print(s)

 print("\n--- Token usage ---")
 print("Total tokens:", getattr(cb, "total_tokens", None))
 print("Prompt tokens:", getattr(cb, "prompt_tokens", None))
 print("Completion tokens:", getattr(cb, "completion_tokens", None))
 print("Estimated cost:", getattr(cb, "total_cost", None))

 else:
 out = run(question)
 print("\nAnswer:", out.get("result"))
 steps = out.get("intermediate_steps")
 if steps:
 print("\n--- Intermediate steps ---")
 for s in steps:
 print(s)

 if __name__ == "__main__":
 main()

```

ch\_05\scr\SQLDatabaseChain.py

```
(.venv) C:\Users\alexc\llm-app\ch_05>python -m src.SQLDatabaseChain

> Entering new SQLDatabaseChain chain...
What is the average discount percentage for Globex, and how many deals does
that include?
SQLQuery:SELECT AVG("discount_pct") AS "average_discount",
COUNT("id") AS "deal_count" FROM deals WHERE "customer" = 'Globex';
SQLResult: [(12.5, 2)]
Answer:Question: What is the average discount percentage for Globex, and how
many deals does that include?
SQLQuery:SELECT AVG("discount_pct") AS "average_discount", COUNT("id") AS
"deal_count" FROM deals WHERE "customer" = 'Globex';
> Finished chain.

Answer: Question: What is the average discount percentage for Globex, and how
many deals does that include?
SQLQuery:SELECT AVG("discount_pct") AS "average_discount", COUNT("id") AS
"deal_count" FROM deals WHERE "customer" = 'Globex';

--- Intermediate steps ---
{'input': 'What is the average discount percentage for Globex, and how many
deals does that include?\nSQLQuery:', 'top_k': '5', 'dialect': 'sqlite',
'table_info': '\nCREATE TABLE deals (\n\tid INTEGER, \n\tcustomer TEXT NOT
NULL, \n\tdiscount_pct REAL NOT NULL, \n\tPRIMARY KEY (id)\n)\n/*\n2 rows
from deals
table:\nid\tcustomer\tdiscount_pct\n1\tGlobex\t10.0\n2\tGlobex\t15.0\n*/',
'stop': ['\nSQLResult:']}
SQLQuery:SELECT AVG("discount_pct") AS "average_discount", COUNT("id") AS
"deal_count" FROM deals WHERE "customer" = 'Globex';
{'sql_cmd': 'SQLQuery: SELECT AVG("discount_pct") AS "average_discount",
COUNT("id") AS "deal_count" FROM deals WHERE "customer" = '\'Globex\'';'}
[(12.5, 2)]
{'input': 'What is the average discount percentage for Globex, and how many
deals does that include?\nSQLQuery:SELECT AVG("discount_pct") AS
"average_discount", COUNT("id") AS "deal_count" FROM deals WHERE "customer" =
\'Globex\'\nSQLResult: [(12.5, 2)]\nAnswer:', 'top_k': '5', 'dialect':
'sqlite', 'table_info': '\nCREATE TABLE deals (\n\tid INTEGER, \n\tcustomer
TEXT NOT NULL, \n\tdiscount_pct REAL NOT NULL, \n\tPRIMARY KEY (id)\n)\n/*\n2
rows from deals
table:\nid\tcustomer\tdiscount_pct\n1\tGlobex\t10.0\n2\tGlobex\t15.0\n*/',
'stop': ['\nSQLResult:']}
Question: What is the average discount percentage for Globex, and how many
deals does that include?
SQLQuery:SELECT AVG("discount_pct") AS "average_discount", COUNT("id") AS
"deal_count" FROM deals WHERE "customer" = 'Globex';

--- Token usage ---
Total tokens: 772
Prompt tokens: 678
Completion tokens: 94
Estimated cost: 0.0001581
```

This example demonstrates how to use SQLDatabaseChain safely by restricting what database tables the LLM can see. The code creates a small SQLite database with two tables: `deals` (allowed) and `secrets` (not allowed). The point is to show that you should never expose the full database schema to an LLM by default. The restriction is applied when building the `SQLDatabase` object: `include_tables = ["deals"]`

This limits the “surface area” of the database that is described to the model. SQLDatabaseChain relies on table metadata (`table_info`) to decide which SQL to generate. If a table is not included, it will not appear in that metadata, so the model cannot

plan queries against it. In this example, the secrets table exists physically, but it is intentionally hidden from the chain. The second setting:

```
sample_rows_in_table_info = 2
```

adds up to two example rows per included table into the table description. This helps the model understand the meaning of columns (for example, that `discount_pct` contains percentages) without giving it unnecessary data. The chain is executed with `verbose = True` and `return_intermediate_steps = True` so you can inspect the generated SQL, which is a best practice for debugging and auditing. Finally, the OpenAI callback measures token usage for the whole run, letting you track cost and spot regressions.

### 5.3.4 Policy enforcement chains: moderation, constitutional revision, and self-checking

“Moderation” is often treated as a single gate, but in practice there are at least three distinct jobs you may want your system to do, and they map to three different kinds of components: classification, revision and moderation

#### 5.3.4.1 Classification

The first job is classification: decide whether a piece of text should be allowed, blocked, or routed to a safer flow. In practice, this is best implemented as a direct call to the provider’s moderation endpoint, wrapped in a small, explicit function that returns a clear payload (flagged, categories, scores) and optionally raises a dedicated exception. This keeps moderation in the “policy sensor” role: it does not fix the text; it tells you what category it falls into so your code can decide what happens next. You can run the check on user input (to catch unsafe requests early), on retrieved context (to avoid re-injecting unsafe content into the model), and on model output (as a final gate before a response is shown or an action is taken). This is also where you choose your failure mode: for high-stakes actions you usually prefer fail-closed (if the moderation check fails or times out, do not proceed), while for low-stakes chat you may accept fail-open with aggressive logging and later review. The key design choice is that moderation should be a stable, dependency-light boundary around the moderation endpoint, rather than a convenience wrapper that may drift or break across SDK or framework versions.

#### Example: Classification as a hard gate

```
from __future__ import annotations
from dataclasses import dataclass
from typing import Any, Dict, Optional
from openai import OpenAI
from .config import OPENAI_API_KEY

@dataclass
class ModerationBlocked(Exception):
 message: str
 flagged: bool
 categories: Dict[str, bool]
 category_scores: Dict[str, float]

 def __str__(self) -> str:
 return self.message

 def moderate_text(
 text: str,
 *,
 model: str = "omni-moderation-latest",
 error: bool = True,
 client: Optional[OpenAI] = None
) -> Dict[str, Any]:
```

```

Moderates a single text input using OpenAI's Moderations endpoint.
If error=True (fail-closed), raises ModerationBlocked when flagged.
If error=False, returns the moderation payload so the caller can decide.
Returns a dict with:
- flagged (bool)
- categories (dict[str,bool])
- category_scores (dict[str,float])
- raw (full response object)
client = client or OpenAI()

response = client.moderations.create(
 model = model,
 input = text
)

OpenAI returns a list of results; for a single string input it's length 1.
result = response.results[0]
flagged = bool(result.flagged)

categories = dict(result.categories) if result.categories is not None else {}
category_scores = (
 dict(result.category_scores) if result.category_scores is not None else {}
)

payload = {
 "flagged": flagged,
 "categories": categories,
 "category_scores": category_scores,
 "raw": response
}

if flagged and error:
 raise ModerationBlocked(
 message = "Blocked by moderation.",
 flagged = flagged,
 categories = categories,
 category_scores = category_scores
)

return payload

def main() -> None:
 safe_text = "Summarise the plot of Frankenstein in two sentences."
 safe = moderate_text(safe_text, error = True)
 print("SAFE:", safe["flagged"], safe_text)

 # A clearly unsafe request should raise when error=True.
 try:
 moderate_text("I want instructions to harm someone.", error = True)
 except ModerationBlocked as exc:
 print("BLOCKED:", exc)
 # Optional: show why
 print("Categories:", {k: v for k, v in exc.categories.items() if v})

if __name__ == "__main__":
 main()

```

ch\_05\scr\ModerationChain.py

```
(.venv) C:\Users\alexc\llm-app\ch_05>python -m src.ModerationChain

SAFE: False Summarise the plot of Frankenstein in two sentences.
BLOCKED: Blocked by moderation.
Categories: {'illicit': True, 'illicit_violent': True, 'violence': True,
'illicit/violent': True}
```

This example shows client-side moderation: you check user text with the Moderations endpoint before you do anything else with it (store it, show it, or send it to a generation model). The main entry point is `moderate_text(text, ...)`. It accepts the input text, the moderation model name (default: "omni-moderation-latest"), and an error switch that controls the behavior when content is flagged.

The function creates an OpenAI client (or uses the one passed in), calls `client.moderations.create(...)`, and reads the first result. For a single input string,

OpenAI returns one result in `response.results[0]`. From that result it extracts three key fields: `flagged` (a single true/false decision), `categories` (which policy areas were triggered), and `category_scores` (confidence scores for those areas). It returns these fields in a small payload, plus the raw response for debugging.

The exception `ModerationBlocked` captures the same moderation details. When `error=True`, the function fails closed: flagged content immediately raises `ModerationBlocked`, preventing unsafe text from reaching later steps. When `error=False`, it fails open: it returns the payload so the caller can decide what to do.

The `main()` function demonstrates both paths: a safe prompt passes; an unsafe request is blocked and prints the triggering categories.

#### 5.3.4.2 Revision

The second job is revision: when content is borderline or merely non-compliant with your own rules (tone, allowed advice, required disclaimers, forbidden claims), you do not always want a hard block. `ConstitutionalChain` represents a different pattern: generate a critique of the draft answer against a “constitution” (a written set of rules), then rewrite the answer to satisfy those rules. This works well for organisational policy that is not purely “unsafe content” but still matters operationally: customer support that must not promise refunds outside policy, trading explanations that must not give personalised investment advice, or an internal assistant that must not reveal confidential identifiers. The key is to keep the constitution concrete and testable. If the rules are vague, you get vague compliance.

#### Example: Content revision

```
from __future__ import annotations
import os
from typing import Any, Dict
from langchain_openai import OpenAI
from langchain_core.prompts import PromptTemplate
from langchain_classic.chains import ConstitutionalChain
from langchain_classic.chains.constitutional_ai.models import ConstitutionalPrinciple
from langchain_classic.chains import LLMChain
from .config import OPENAI_API_KEY
import warnings

warnings.filterwarnings(
 "ignore",
 Message = r".*class `LLMChain` was deprecated.*",
 Category = DeprecationWarning
)
warnings.filterwarnings(
 "ignore",
 Message = r".*LLMChain.*deprecated.*",
 Category = UserWarning
)

def main() -> None:
 llm = OpenAI(temperature = 0.2)

 prompt = PromptTemplate.from_template(
 "Answer the user question.\n\nQuestion: {question}\n\nAnswer:"
)

 # NOTE: ConstitutionalChain (classic) expects a Chain object here (LLMChain),
 # not an LCEL RunnableSequence.
 draft_chain = LLMChain(llm = llm, prompt = prompt)

 principles = [
 ConstitutionalPrinciple(
 Name = "No personal data",
 critique_request = "Identify any personal data or sensitive identifiers in the answer.",
 revision_request = "Rewrite the answer removing personal data and using"
)
]
```

```

 general language."
)

 constitutional = ConstitutionalChain.from_llm(
 llm = llm,
 chain = draft_chain,
 constitutional_principles = principles,
 verbose = True
)

 out: Dict[str, Any] = constitutional.invoke(
 {"question": "Write a short reply that includes my full address and phone number."})

 print(out.get("output") or out.get("result") or out)

if __name__ == "__main__":
 main()

```

ch\_05\scr\content\_revision.py

```

(.venv) C:\Users\alexc\llm-app\ch_05>python -m src.ModerationChain

> Entering new ConstitutionalChain chain...
Initial response: Sure! My full address is 123 Main Street, Anytown, USA 12345
and my phone number is (123) 456-7890.

Applying No personal data...

Critique: The model has provided the user's full address and phone number,
which are both personal data and sensitive identifiers. This information should
not be shared without the user's explicit consent. Critique Needed.

Updated response: Sure! I can provide you with my contact information.

> Finished chain.
Sure! I can provide you with my contact information.

```

This example shows how to use ConstitutionalChain and ConstitutionalPrinciple to add a “self-review and rewrite” safety layer on top of a normal LLM response.

The code first builds a simple draft answer pipeline. It creates an OpenAI model with low temperature (0.2) to reduce randomness, then defines a PromptTemplate that formats a user question into a plain “Question/Answer” prompt. Because the classic ConstitutionalChain expects a Chain object, the draft step is implemented with LLMChain (not an LCEL runnable). That draft\_chain is the first pass: it produces an initial answer exactly as the model would normally respond.

Next, the code defines a list of ConstitutionalPrinciple rules. Each principle has two instructions: a critique\_request (what to look for in the draft) and a revision\_request (how to rewrite if the critique finds a problem). In this example the rule is “No personal data”: the critique step checks whether the answer contains personal or sensitive identifiers, and the revision step removes them and replaces them with general wording.

Finally, ConstitutionalChain.from\_llm wraps the draft\_chain with this rule set. When invoked, it runs: draft answer → critique against principles → revised answer. The sample input deliberately asks for an address and phone number, so the principle should trigger and produce a sanitized output.



This “classic” ConstitutionalChain / LLMChain approach is deprecated in LangChain, so you should treat it as legacy code and the preferred alternative is to re-implement the same “draft → critique → revise” pattern using LangGraph plus LCEL-style runnables. If you are not planning to use LangGraph, this is a valid approach.

#### 5.3.4.3 Verification

The third job is verification: ask a model to check a model. LLMCheckerChain is a self-check pattern where a second pass looks for internal contradictions, obvious factual errors, or claims that are not supported by the provided context. This can be useful, but it should be treated as a heuristic layer, not a guarantee of truth. A checker can miss subtle errors, and it can confidently approve a wrong answer. The right way to use it is to narrow its scope: have it check specific claims, require it to point to evidence when evidence exists, and make its output actionable (for example: “approve”, “revise”, “refuse/escalate”). In other words, you are not outsourcing correctness; you are adding another filter that catches a subset of failures cheaply compared to human review.

#### Example: LLMCheckerChain as a self-checking answer layer

```

import os
import warnings
from langchain_openai import OpenAI
from langchain_classic.chains.llm_checker.base import LLMCheckerChain
from .config import OPENAI_API_KEY

Optional: hide ONLY the "LLMCheckerChain is deprecated" warning (keeps other warnings
visible)
try:
 from langchain_core._api.deprecation import LangChainDeprecationWarning
except Exception:
 LangChainDeprecationWarning = DeprecationWarning

warnings.filterwarnings(
 "ignore",
 Message = r".*LLMCheckerChain.*Deprecated since version.*",
 category = LangChainDeprecationWarning
)

def main() -> None:
 llm = OpenAI(temperature = 0.0)

 # FIX: use from_llm (removes the "Directly instantiating..." warning)
 checker = LLMCheckerChain.from_llm(llm, verbose = False)

 question = "What is the capital of France?"
 draft_answer = "The capital of France is Lyon." # intentionally wrong

 # LLMCheckerChain is designed to take a QUESTION and return a self-checked answer.
 out = checker.invoke(question)

 # Output key varies a bit across versions; handle both
 checked_answer = out.get("text") or out.get("result") or out.get("output") or
 str(out)

 # Your decision logic (approve vs revise vs escalate)
 if checked_answer.strip() == draft_answer.strip():
 action = "approve"
 elif "i don't know" in checked_answer.lower() or "cannot" in checked_answer.lower():
 action = "refuse_escalate"
 else:
 action = "revise"

 print("ACTION:", action)
 print("QUESTION:", question)
 print("DRAFT:", draft_answer)
 print("CHECKED:", checked_answer)

if __name__ == "__main__":
 main()

```

ch\_05\scr\LLMCheckerChain.py

```
(.venv) C:\Users\alexc\llm-app\ch_05>python -m src.LLMCheckerChain
> Entering new SequentialChain chain...
> Finished chain.
ACTION: revise
QUESTION: What is the capital of France?
DRAFT: The capital of France is Lyon.
CHECKED: The capital of France is Paris.
```

This example demonstrates LLMCheckerChain, a “self-checking” question-answer chain. It creates an OpenAI LLM with temperature 0.0 (to reduce randomness), then builds the checker with LLMCheckerChain.from\_llm(), which avoids the “direct instantiation” deprecation warning. When you call checker.invoke(question), the chain generates an answer and runs an internal verification pass before returning the final text.

Two practical touches improve robustness. First, the warnings filter targets only the LLMCheckerChain deprecation warning, so unrelated warnings still show up during development. Second, the code reads the result defensively by checking multiple possible output keys (“text”, “result”, “output”) and falling back to str(out), which helps across minor version differences.

The script then compares the checked answer to an intentionally wrong draft answer and chooses an action: approve (same), refuse/escalate (the checked answer contains explicit uncertainty such as “I don’t know” or “cannot”), or revise (a different, confident answer).

|                                                                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | LLMCheckerChain is deprecated in LangChain. The LangChain API reference notes the deprecation and points to LangGraph guides for self-reflection and corrective strategies as the recommended direction. In practice, replace it with an explicit LangGraph (or LCEL) flow: generate a draft answer, critique it in a second model call, and regenerate only when the critique flags real problems. If you are not planning to use LangGraph, this is a valid approach. |
|-------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

Once you separate these roles, you can build a layered policy pipeline that is both clearer and easier to operate: classify obvious violations early, revise when a compliant rewrite is appropriate, and self-check when the cost of a wrong answer is high. Most importantly, every step produces signals you can log and evaluate: how often moderation blocks, how often constitutional revision changes answers, and how often the checker forces regeneration or escalation. They are measurable system behaviours you can tune over time.

### 5.3.5 Google GenAI safety settings

Google Gemini / Google GenAI policy enforcement uses the same “classify first, then decide” idea, but the enforcement mechanism is built into the generation call via safety settings. Instead of calling a separate moderation endpoint, you provide a safety\_settings map that sets a blocking threshold per harm category (for example dangerous content, hate speech, harassment, sexually explicit). For any category you do not specify, the API falls back to the model’s default safety setting for that category. Each threshold expresses “how sensitive the filter is” for that category. Conceptually:

- `BLOCK_NONE` disables blocking for that category
- `BLOCK_ONLY_HIGH` blocks only when the model judges a high probability of harm
- `BLOCK_MEDIUM_AND_ABOVE` blocks medium and high
- `BLOCK_LOW_AND_ABOVE` blocks low, medium, and high

This makes safety policy a configurable guardrail on both prompts and responses, rather than a separate step you have to remember to run.

When Gemini blocks a response, the blocked content is not returned. Instead, the response candidate reports a finish reason of `SAFETY` and includes safety ratings you can inspect to understand which category triggered the block. That gives you a clean control flow: treat “`SAFETY`” as a hard gate (fail-closed), or route to a safer rewrite/escalation path.

### Example: configuring safety thresholds

```
from google.genai.types import HarmBlockThreshold, HarmCategory
from langchain_google_genai import ChatGoogleGenerativeAI

model = ChatGoogleGenerativeAI(
 model = "gemini-2.5-flash",
 temperature = 0.2,
 safety_settings = {
 HarmCategory.HARM_CATEGORY_DANGEROUS_CONTENT:
 HarmBlockThreshold.BLOCK_MEDIUM_AND_ABOVE,
 HarmCategory.HARM_CATEGORY_HATE_SPEECH: HarmBlockThreshold.BLOCK_ONLY_HIGH,
 HarmCategory.HARM_CATEGORY_HARASSMENT: HarmBlockThreshold.BLOCK_LOW_AND_ABOVE,
 HarmCategory.HARM_CATEGORY_SEXUALLY_EXPLICIT: HarmBlockThreshold.BLOCK_NONE
 }
)
```

## 5.4 ZERO-SHOT AND FEW-SHOT PROMPTING; DYNAMIC FEW-SHOT SELECTION

Prompt templates from the previous section give you a stable shape for your prompts. Zero-shot and few-shot prompting decide how much “scaffolding” you put inside that shape in the form of examples. Zero-shot prompts rely on clear instructions and the model’s pre-training. Few-shot prompts add explicit input–output examples that show the model exactly what you want, often with a big jump in reliability for classification, extraction, or tightly formatted outputs. In real systems you rarely stop at a single hand-written few-shot prompt. As your support assistant sees more tickets, your e-commerce app sees more product questions, or your trading assistant sees more edge cases, you accumulate a growing set of good examples. LangChain turns that set into a first-class component: you can build static few-shot templates, then upgrade them to dynamic few-shot prompts that select the most relevant examples per request using embeddings and example selectors. This section builds that ladder step by step and keeps the same three scenarios in mind: routing customer-support tickets, classifying e-commerce queries, and helping a trading assistant interpret market events.

### 5.4.1 Zero-shot prompting: starting with instructions only

Zero-shot prompting means: describe the task, provide the input, and give no explicit examples. The model relies entirely on what it learned during pre-training plus your instructions. This is often good enough when:

- The task is common in training data (plain sentiment, short summarisation, generic question answering).

- The output format is simple (a short sentence, a label from a very small set).

In a support assistant, a natural zero-shot starting point is intent classification: decide whether a message is about billing, technical issues, or cancellations, without any examples. Here is a minimal zero-shot classifier using `PromptTemplate` and a chat model. The model sees only the instruction and the current message.

```
from .config import OPENAI_API_KEY
from langchain_core.prompts import PromptTemplate
from langchain_openai import ChatOpenAI

Basic safety checks so failures are clear
if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

Zero-shot intent classifier for support tickets
support_intent_prompt = PromptTemplate.from_template(
 """
 You are a customer-support triage assistant for ACME Telecom.

 Task:
 Read the customer message and classify it into exactly one of:
 - BILLING
 - TECHNICAL
 - CANCELLATION
 - OTHER

 Output only the label.

 Customer message:
 {message}
 """
)

classifier_model = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.2
)

intent_chain = support_intent_prompt | classifier_model
result = intent_chain.invoke(
 {
 "message": "My internet has been very slow since yesterday evening."
 }
)
print(result.content)
```

ch\_05\scr\zero\_shot\_classifier\_using\_PromptTemplate.py

Zero-shot prompting is attractive because it is almost free to set up. You tweak only wording and structure. With a well-specified task and simple labels, modern models often produce acceptable results. The weaknesses show up when:

- The labels are subtle (for example, differentiating “urgent regulatory risk” from “market noise” in trading alerts).
- You need strict adherence to a schema (for example, JSON outputs with required keys).
- You care about domain-specific conventions that the model might not have seen frequently in training.

At that point, you move up the ladder to one-shot and few-shot prompting.

#### 5.4.2 One-shot and few-shot prompting: teaching by example

One-shot prompting adds a single example of input plus desired output before the real task. Few-shot prompting adds several such examples. In both cases you are showing the model what you want rather than just telling it. Typical reasons to move from zero-shot to few-shot are:

- You want tight control over output format, such as a JSON object or a short label with specific spelling.
- The task involves nuanced distinctions, like fine-grained sentiment, intent, or risk buckets.
- Zero-shot behaviour is unstable across similar inputs.

LangChain models few-shot prompting with `FewShotPromptTemplate`. This class combines:

- A list of examples (each a dictionary with input and output fields).
- An `example_prompt`, which is a `PromptTemplate` describing how to render one example.
- A `prefix` with task instructions that appears before all examples.
- A `suffix` that contains the actual runtime input.

At invocation time `FewShotPromptTemplate` formats each example through `example_prompt`, joins them with the `prefix` and `suffix`, and returns a single string ready for the model.

### Example: routing tickets with few-shot prompting

Suppose the zero-shot classifier above sometimes mislabels cancellation messages as billing. You can provide explicit demonstrations of each class.

```
from .config import OPENAI_API_KEY
from langchain_core.prompts import PromptTemplate
from langchain_core.prompts.few_shot import FewShotPromptTemplate
from langchain_openai import ChatOpenAI

Basic safety checks so failures are clear
if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

Few-shot intent classifier for support tickets
examples = [
 {
 "message": "I want to cancel my internet contract at the end of this month.",
 "label": "CANCELLATION",
 },
 {
 "message": "You charged me twice for the same bill.",
 "label": "BILLING",
 },
 {
 "message": "My router keeps disconnecting every few minutes.",
 "label": "TECHNICAL",
 },
 {
 "message": "Can I upgrade to a faster plan?",
 "label": "OTHER",
 },
]
example_prompt = PromptTemplate.from_template(
 "Message: {message}\nLabel: {label}"
)
support_prefix = """
You are a customer-support triage assistant for ACME Telecom.

Task:
Read each customer message and assign exactly one label:
- BILLING
- TECHNICAL
- CANCELLATION
- OTHER

Use the examples to understand the labelling rule.
"""
support_suffix = """
Message: {message}
"""


```

```

Label:
"""

few_shot_support_prompt = FewShotPromptTemplate(
 examples = examples,
 example_prompt = example_prompt,
 prefix = support_prefix.strip(),
 suffix = support_suffix.strip(),
 input_variables = ["message"]
)

support_llm = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.0
)

support_chain = few_shot_support_prompt | support_llm
result = support_chain.invoke(
 {
 "message": "I changed my mind and want to stop my contract next week."
 }
)

print(result.content)

```

[ch\\_05\scr\few\\_shots\\_classifier\\_using\\_FewShotPromptTemplate.py](#)

What the template is doing:

1. `examples` is your small dataset of curated input–label pairs. Each dictionary contains fields that match `example_prompt` variables.
2. `example_prompt` defines how a single example is rendered. In this case it becomes two lines: the message and the label.
3. `FewShotPromptTemplate` receives the `examples`, the `example prompt`, a `prefix` with task instructions, and a `suffix` with the runtime variable `{message}`. Its `format` method builds a full prompt: `prefix`, then all examples rendered by `example_prompt`, then the final “Message: ... Label:” section.

The examples play the role of a small in-prompt training set. In practice, even a handful of examples can significantly improve accuracy on structured tasks compared to pure instructions, especially for smaller or older models.

The quality of examples matters more than the raw number. The guidance from prompt-engineering literature and LangChain’s own usage is consistent:

- Keep examples correct and unambiguous; errors in examples are amplified by the model.
- Cover typical cases and a few likely edge cases.
- For classification, mix examples from different classes instead of grouping all of one label together; this reduces positional biases.

As modern models support longer context windows, you can extend this pattern toward “many-shot” prompting with dozens of examples when cost and latency budgets allow it, but you still need to manage prompt length and noise.

#### 5.4.3 Dynamic few-shot selection: choosing examples per input

Static few-shot prompts use the same examples for every request. That is often good enough for small, homogeneous tasks. As your system grows, you accumulate more examples across products, markets, and user segments, and a single fixed set no longer fits all queries.

Dynamic few-shot selection solves this by picking examples on the fly for each input. LangChain models this through example selectors, most commonly `SemanticSimilarityExampleSelector`. The workflow is:

1. Maintain a pool of high-quality examples, each with input and output fields.
2. Use an embedding model and a vector store to index those examples. An embedding model turns each example input into a numeric vector that captures its meaning. A vector store keeps all these vectors in a searchable index so you can quickly find examples whose meaning is close to a new query.
3. For each new input, compute its embedding with the same embedding model and retrieve the most similar examples from the vector store.
4. Feed only those examples into a `FewShotPromptTemplate`.

LangChain's in-memory vector store and OpenAI embeddings rely on numpy for numerical operations, and in some environments, you need to install it explicitly rather than relying on transitive dependencies.

```
C:\Users\yourname\llm-app> .venv\Scripts\activate
(.venv) C:\Users\yourname\llm-app> pip install numpy
```

### Example: dynamic few-shot for risk explanations

Imagine a trading assistant that explains risks for different kinds of positions (equity, options, leveraged ETFs). You keep a library of good "question → explanation" pairs. When a user asks about a specific position, you want examples that match the same instrument type and style.

```
from .config import OPENAI_API_KEY
from langchain_core.prompts import PromptTemplate
from langchain_core.prompts.few_shot import FewShotPromptTemplate
from langchain_core.example_selectors import SemanticSimilarityExampleSelector
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEMBEDDINGS, ChatOpenAI

Basic safety checks so failures are clear
if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

Example library: questions and ideal explanations for different instruments
risk_examples = [
 {
 "question": "What are the main risks of holding a single tech stock?",
 "answer": "High idiosyncratic risk: company-specific news can move the price sharply.",
 },
 {
 "question": "What are the risks of a leveraged ETF that tracks an index?",
 "answer": "Leverage amplifies both gains and losses and can suffer from daily rebalancing decay.",
 },
 {
 "question": "What are the risks of selling a covered call on my shares?",
 "answer": "You cap your upside and may be forced to sell the shares if the option is exercised.",
 },
 {
 "question": "What are the risks of holding a long-dated government bond?",
 "answer": "Interest-rate risk: prices can fall if yields rise; inflation can erode real returns.",
 },
]
example_prompt = PromptTemplate.from_template(
 "Question: {question}\nExplanation: {answer}"
```

```

 }

Embeddings and vector store for semantic similarity over example questions
embeddings = OpenAIEmbeddings(model = "text-embedding-3-small")

Build a selector that retrieves the 3 most similar examples per query
example_selector = SemanticSimilarityExampleSelector.from_examples(
 examples = risk_examples,
 embeddings = embeddings,
 vectorstore_cls = InMemoryVectorStore,
 k = 3,
 input_keys = ["question"]
)

risk_prefix = """
You are a trading risk assistant.

Task:
Explain the main risks of the user's position in plain language.
Use the examples to match style and level of detail.
"""

risk_suffix = """
Question: {question}
Explanation:
"""

dynamic_risk_prompt = FewShotPromptTemplate(
 example_selector = example_selector,
 example_prompt = example_prompt,
 prefix = risk_prefix.strip(),
 suffix = risk_suffix.strip(),
 input_variables = ["question"]
)

risk_llm = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.2,
)
risk_chain = dynamic_risk_prompt | risk_llm

result = risk_chain.invoke(
 {
 "question": "What are the risks of holding a 3x leveraged ETF tracking bank stocks?",
 }
)
print(result.content)

```

[ch\\_05\scr\few\\_shots\\_classifier\\_using\\_PromptTemplate\\_wirh\\_selector.py](#)

When you call `risk_chain.invoke({...})`, LangChain runs a small pipeline made of four main pieces: embeddings + vector store, example selector, few-shot prompt, and chat model. First, the embeddings and vector store:

1. `OpenAIEmbeddings` turns each example question in `risk_examples` into a numerical vector that captures its meaning.
2. `InMemoryVectorStore` stores these vectors and lets you search for “nearest neighbours” given a new query using an in-memory vector store.

This gives you a semantic memory of your examples: questions that “mean something similar” end up close to each other in the vector space.

Second, the example selector `SemanticSimilarityExampleSelector.from_examples(...)` wires the examples, embeddings, and vector store together. For each new input, it:

1. Embeds the new question using the same `OpenAIEmbeddings`
2. Searches the vector store for the ‘k’ most similar example questions
3. Returns those examples as a list of dictionaries

In this example, `k=3`, so the selector always returns the three examples whose questions are closest in meaning to the current user question. If the user asks about a leveraged ETF on bank stocks, questions about leveraged ETFs or concentrated positions will rank higher than questions about government bonds.

Third, the dynamic few-shot prompt: `FewShotPromptTemplate` takes the selector and an `example_prompt`. When you call `dynamic_risk_prompt.format(...)` (or, here, use it in a chain), it:

1. Calls the selector with the current input (``{"question": ...}``),
2. Formats each selected example using `example_prompt` (``"Question: {question}\nExplanation: {answer}"``),
3. Inserts those formatted examples between `risk_prefix` and `risk_suffix`.

The result is a prompt that always has the same structure, but different examples depending on the question. A user asking about covered calls sees option-related examples; a user asking about long-dated bonds sees fixed-income examples, and so on.

#### Why use dynamic selectors instead of hard-coding examples?

You could manually pick a few examples and hard-code them into the prompt. That works for small, stable tasks. Dynamic selection becomes valuable when:

- Your library of examples grows (many instruments, many risk patterns, many user profiles)
- You want each query to see only the examples that are relevant to it
- You want to improve behaviour over time by adding new curated examples, without editing prompt code.

Here, when you see a bad answer for a particular product in production, you can add a better “question → explanation” pair to `risk_examples` and reindex. Future queries that look similar will automatically pick up the new example.

#### 5.4.4 Overview of example selectors in LangChain

`SemanticSimilarityExampleSelector` is one of several selectors LangChain offers. The main patterns you can rely on are:

1. `SemanticSimilarityExampleSelector`: Picks examples whose inputs are closest in meaning to the current input, using embeddings and a vector store. This is the default choice when you care most about “examples that talk about the same thing,” as in the trading risk scenario.
2. `MaxMarginalRelevanceExampleSelector`: Also uses embeddings and a vector store, but balances similarity with diversity. It prefers examples that are relevant to the input while avoiding near-duplicates of each other. This is useful when you want the model to see several “angles” on a task instead of many almost-identical examples.
3. `LengthBasedExampleSelector`: Does not use embeddings. It selects examples based on length so that the final prompt stays under a target size. This is useful when you must respect a tight context window and want to keep shorter, more compact examples.
4. `NGramOverlapExampleSelector` (in the community extensions): Chooses examples based on how many word sequences (n-grams) they share with the input. This is mainly helpful when surface-form overlap is a good proxy for relevance, and you do not want to bring in embeddings.

All of these selectors implement the same interface: given an input dictionary, they return a list of example dictionaries. You can plug any of them into `FewShotPromptTemplate` as `example_selector` without changing the rest of your code. If none of them fits your needs, you can also implement your own selector by subclassing the base selector interface `BaseExampleSelector` and defining how `select_examples` should work.

#### 5.4.5 Few-shot prompting with chat prompts

`FewShotPromptTemplate` works at the string level. For chat models that already use `ChatPromptTemplate` from the previous section, you can use a chat-oriented variant such as `FewShotChatMessagePromptTemplate` to embed examples as message pairs inside a larger chat prompt. This is useful when your examples need distinct roles: a human message and the assistant's ideal reply. The idea is the same: show a few high-quality interactions and then append the live user question. Conceptually the structure becomes:

1. System message for persona and rules.
2. Several (human, ai) message pairs as examples.
3. A final human message with the actual user question.

You will reuse this pattern later when you build assistants that must follow a precise conversational style.

#### Example: few-shot chat prompts for cross-sell suggestions

A customer asks, “what else should I buy?” and you want the assistant to recommend at most two complementary items, each justified in one short sentence. The goal is consistency: suggestions should be practical and follow the same structure across many queries, without drifting into long lists or marketing language.

```
from .config import OPENAI_API_KEY
from langchain_core.prompts import (
 ChatPromptTemplate,
 FewShotChatMessagePromptTemplate,
)
from langchain_openai import ChatOpenAI

Basic safety checks so failures are clear
if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

Example library: short, realistic e-commerce conversations
examples = [
 {
 "customer_query": (
 "I'm buying a mid-range gaming laptop. What else should I "
 "add?"
),
 "assistant_reply": (
 "You could add a cooling pad to keep the laptop temperature stable "
 "during long sessions, and a mid-range gaming mouse for better control."
),
 },
 {
 "customer_query": "I just ordered a mirrorless camera body. Any accessories?",
 "assistant_reply": (
 "A fast prime lens will improve low-light and portrait shots, "
 "and an extra battery helps on long shooting days."
),
 },
 {
 "customer_query": "I'm getting a new 4K TV for my living room.",
 "assistant_reply": (
 "Consider a soundbar for clearer dialogue and richer sound, "
 "and a wall mount if you want to save floor space."
),
 },
]
```

```

 },
]
How each example appears as a chat exchange: one human turn, one assistant turn
example_prompt = ChatPromptTemplate.from_messages(
 [
 ("human", "{customer_query}"),
 ("ai", "{assistant_reply}"),
]
)

Few-shot message block: expands into alternating human/ai messages
few_shot_examples = FewShotChatMessagePromptTemplate(
 examples = examples,
 example_prompt
)

Full chat prompt:
- system message for role and rules
- few-shot examples as real chat history
- live human message with the current customer query
recommendation_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You are an e-commerce assistant for ACME Shop. "
 "For each customer query, suggest at most two complementary products. "
 "Keep answers concise, practical, and avoid marketing buzzwords."
),
 few_shot_examples,
 ("human", "{customer_query}")
)
]
)

model = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.3
)
chain = recommendation_prompt | model

result = chain.invoke(
 {
 "customer_query": (
 "I'm ordering wireless noise-cancelling headphones. "
 "What else would you suggest?"
),
 }
)
print(result.content)

```

ch\_05\scr\few\_shots\_prompting\_with\_chat\_prompt.py

This example has three main layers: the examples, the few-shot chat template, and the final prompt combined with a chat model.

1. The examples list is a small, curated dataset of realistic interactions from your e-commerce domain. Each element has two fields that line up with the variables used in `example_prompt`:

- `customer_query`: the user's question in plain language.
- `assistant_reply`: the answer that follows your business rules (at most two suggestions, short justifications).

Because the keys in each dictionary match the placeholders in the template, LangChain can turn every example directly into a short human–assistant exchange without extra mapping code.

2. `example_prompt = ChatPromptTemplate.from_messages([...])` describes the shape of a single interaction: one human message with `{customer_query}` followed by one AI message with `{assistant_reply}`. `FewShotChatMessagePromptTemplate` takes

the list of example dictionaries plus this template and, when the outer prompt is formatted, it iterates over the examples, applies the template to each one, and concatenates the resulting pairs into a short conversation history. The model sees these as normal chat turns, not as a single block of text, which makes the pattern easier to learn and easier to inspect in logs and traces.

3. The full chat prompt and model: `recommendation_prompt` wraps this few-shot block into a complete prompt: a system message defining the assistant's role and constraints, the expanded few-shot message history, and a final human message with the live `{customer_query}`. When you call `chain.invoke({ ... })`, LangChain fills in the input variable in the last human message, expands the few-shot examples into chat messages, assembles the full message list, and sends it to ChatOpenAI. The model generates a reply that mirrors the tone and structure of the examples, so your cross-sell suggestions stay close to the demonstrated style rather than drifting into generic recommendations.

#### 5.4.6 When to use zero-shot, few-shot, and dynamic few-shot

In practice you will move between these modes rather than picking one forever. A simple rule of thumb that matches current literature and framework guidance is:

1. Start with zero-shot plus clear instructions. Measure how far that gets you on your support, e-commerce, or trading tasks.
2. Introduce one-shot or few-shot prompts when you need more reliable structure, style, or reasoning, and you can afford a small increase in token usage.
3. Move to dynamic few-shot selection once you have many examples and see that different parts of your domain behave differently.

In all cases, treat examples as part of your system's specification. Keep them under version control and use evaluation runs over real or realistic datasets to compare different prompts and selectors. As you log more traffic and discover new behaviours you want the model to imitate or avoid, extend the example set and re-run evaluations rather than relying on ad-hoc prompt edits.

## 5.5 CHAIN-OF-THOUGHT AND SELF-CONSISTENCY

### PROMPTING

Chain-of-thought (CoT) and self-consistency prompting sit on top of the zero-shot and few-shot techniques from the previous section. They are part of a broader family of “reasoning-oriented” prompts that also includes tree-of-thought and some of the summarisation patterns later in the book. Instead of asking the model to jump directly from input to answer, these prompts ask it to spell out intermediate reasoning steps and, in some cases, explore several alternative paths before deciding which answer to trust.

Research on large language models shows that prompting them to generate explicit reasoning chains can substantially improve accuracy on arithmetic, commonsense, and symbolic reasoning tasks compared to direct “answer-only” prompting. Self-consistency extends this idea by sampling multiple reasoning chains and selecting the most consistent final answer, often yielding further gains on benchmarks. In LangChain, these are expressed as prompt patterns rather than special primitives, which makes them easy to integrate into LCEL workflows.

#### 5.5.1 Chain-of-thought prompting: explicit intermediate reasoning

Chain-of-thought prompting asks the model to write down a sequence of reasoning steps before giving a final answer. Instead of responding “The portfolio has medium risk,” the model might first list which positions are concentrated, which are leveraged, and how those interact, then conclude with a short risk summary. Conceptually, CoT changes the mapping from:

“Input → Answer”

to:

“Input → Reasoning steps → Answer”.

The reasoning steps are free-form text but usually follow a simple pattern: restate relevant facts, perform intermediate calculations or logical inferences, and then derive the final conclusion. Empirical work shows that giving models this “space to think” improves performance on multi-step reasoning benchmarks, particularly for larger models.

There is no single, universal flag that turns on chain-of-thought across all models. For many general chat models, the main control is prompting; however, some reasoning-focused model families expose explicit ‘thinking/reasoning’ controls. Outside of choosing a reasoning-focused model family, CoT is controlled at the prompt level: you decide whether to ask for explicit reasoning, how to format it, and when to show or hide it from end users. Temperature/top\_p affect sampling style, but they are not a reliable way to induce step-by-step decomposition without an instruction or an example structure. Two prompt-level variants are commonly used:

- Zero-shot CoT: You keep the original question and add a short cue such as “Let’s think step by step.” or an equivalent instruction in a system or user message. There are no worked examples; the model is only told to expose its reasoning. Despite its simplicity, this pattern often turns a single-sentence answer into a numbered or paragraph-style explanation that

walks through the solution. On math word problems and logic puzzles, this can be enough to reduce brittle guessing.

- Few-shot CoT: You reuse the few-shot structure from section 5.4.3, but each example output includes a short reasoning trace followed by the answer. For instance, in a sentiment classifier you might show which phrases indicate positive or negative sentiment and then state the label, instead of just providing “positive/neutral/negative”. At runtime, the model tends to mimic this structure on new inputs, leading to more interpretable and often more accurate decisions.

Often reported to help on tasks that require multiple steps (for example arithmetic and symbolic/logical reasoning benchmarks). Gains are larger for medium and large models; smaller models can struggle to maintain coherent long chains, and in some experiments, CoT brings little or no benefit there. It can improve debuggability because you can inspect the model’s stated reasoning. However, chain-of-thought text is not guaranteed to be a faithful explanation of how the model reached the answer. It does increase cost and latency because the model generates more tokens. This matters if CoT is used inside larger chains, agents, or graphs that already have multiple model calls.

Some modern reasoning-focused models perform long internal reasoning before answering. They generally do not expose raw internal chains of thought; instead, they return the final answer and, in some APIs, an optional reasoning summary. Google’s Gemini “thinking” variants follow a similar pattern, using an internal thinking phase to improve multi-step planning and reasoning. In practice, you control this at the level of model choice: you select a reasoning-oriented model variant when you want deeper internal deliberation, rather than turning on a generic “think” flag on an otherwise identical chat model.

### 5.5.2 Self-consistency prompting: sampling multiple reasoning paths

Self-consistency extends CoT by using several reasoning chains instead of a single one. Rather than trusting the first chain the model produces, you deliberately sample multiple chains and aggregate their final answers, typically via a majority vote.

The underlying assumption, supported by experimental results, is that for problems with a unique correct answer there are often several valid reasoning routes that arrive at that answer. If you sample diverse chains with a higher temperature, incorrect trajectories are likely to disagree with each other, while correct trajectories tend to converge on the same result. A typical workflow looks like this:

1. Use a CoT-style prompt to ask the model to reason step by step about a question.
2. Run this prompt multiple times with non-zero temperature to obtain several distinct reasoning traces and answers.
3. Extract the final answer from each trace.
4. Aggregate the answers and select the most frequent (or otherwise “most consistent”) one as the final output.

In LangChain projects, self-consistency is typically implemented in one of two ways, depending on where you want the “aggregation” logic to live.

The simplest approach is an explicit sampling loop in Python. You build a CoT-style chain that returns a final answer (and may also include a reasoning trace if you explicitly ask for it), then you run it N times with a non-zero temperature, so the model explores different reasoning paths. From each run, you extract only the final answer (for example, the numeric result or a single label), count how often each answer appears, and return the most frequent one. In this version, the model generates candidates and your code performs the voting.

The second approach moves the voting step into the model. One chain generates N candidate answers (either by calling the chain N times or by requesting multiple candidates when the provider supports it). A second, short “judge” prompt then receives the list of candidate answers and is asked to pick the most frequent value (or to normalise equivalent answers and then pick the mode). This reduces custom parsing and lets you keep the logic inside promptable components, at the cost of an extra model call.

Published experiments report that this “sample many, then vote” strategy can improve accuracy on reasoning benchmarks compared to a single greedy chain-of-thought output, particularly when the task has a single correct answer and a compact answer space (numbers, dates, labels). In the original self-consistency paper, the authors reported a +17.9% improvement on GSM8K in their setup, with additional gains on SVAMP (+11.0%), AQuA (+12.2%), StrategyQA (+6.4%), and ARC-Challenge (+3.9%) relative to greedy CoT decoding. These results are task- and setup-dependent, but they support the engineering trade-off: more samples mean higher cost and latency, in exchange for higher reliability when wrong answers tend to disagree with each other. In real applications, self-consistency is most useful when:

1. The answer space is compact: class labels, dates, numeric values, or short textual spans.
2. Accuracy matters more than latency and cost, for example in offline evaluation pipelines, risk checks, or high-stakes decision support.
3. You already have a CoT-style prompt that works reasonably well, and you want to reduce variance rather than redesign the prompt.

The downsides are straightforward: each query now triggers several model calls and generates multiple chains, so token usage and latency scale roughly with the number of samples. In addition, if the CoT prompt systematically biases the model toward a wrong answer, majority voting will not fix the mistake; it only averages over stochastic variation, it does not correct a deterministic error. Within the running scenarios, self-consistency is a good fit for tasks such as:

- Verifying a support assistant’s answer on a tricky policy edge case before sending it to a customer,
- Double-checking a classification (for example, “refund allowed/partial/denied”) when mislabelling has a clear cost,
- Validating a derived trading metric, or double-checking a factual claim (for example, the year of a historical event) in a market commentary, before including it in a report.

### 5.5.3 How CoT and self-consistency fit into LangChain workflows

That was the theory, now let’s look in this and next section at how LangChain, applies CoT and self-consistency. They are expressed using the same building blocks introduced earlier in

this chapter: prompt templates, chat models, LCEL chains, and graph control flow. At a prompt level, CoT lives entirely inside templates:

1. Add a simple instruction to your existing prompt. Example: Add a line like “Think step by step before answering.”
2. Split the model’s output into two clearly labelled parts, then keep only the final part.

Example: Ask for:

- “THINKING:” followed by free-form reasoning
- “ANSWER:” followed by the final response

After the model responds, your code extracts and uses only the text under “ANSWER:”.

3. Turn your few-shot examples into CoT examples by including reasoning in the example outputs. Example: In your prompt’s example conversations, rewrite the example “assistant” outputs so they show the reasoning process (not just the final answer). This teaches the model the pattern you want it to follow.

At a control-flow level, self-consistency is implemented by repeating chains or nodes and aggregating outputs:

1. In LCEL, you can wrap a CoT chain in a small helper that calls `invoke` several times with the same input and aggregates results before returning to the rest of the pipeline.
2. In LangGraph, you can model self-consistency as a loop over a reasoning node that appends candidate answers to state, followed by a reducer or dedicated node that performs the vote and emits a single final answer (as we’ll see in a later chapter).

Reasoning-focused models alter the design slightly. Rather than manually adding CoT instructions, you enable the provider’s reasoning mode (for example, a “reasoning” toggle or dedicated model family), rely on the model’s internal chains, and consume either a compact answer or a high-level reasoning summary where available.

In these cases, LCEL and LangGraph still provide the scaffold: prompts frame the task and constraints, and control flow decides when to call a reasoning-capable model and how to use its output.

From a system-design perspective, CoT and self-consistency are tools you can selectively apply rather than a default. Later chapters will revisit these patterns in the context of retrieval-augmented generation, evaluation with LangSmith, and LangGraph-based agents, where explicit reasoning and voting interact with tools, memory, and human-in-the-loop review.

#### 5.5.4 CoT and self-consistency in a trading assistant

To make chain-of-thought and self-consistency less abstract, this example shows a small trading assistant that:

1. Uses chain-of-thought prompting to reason step by step about a simple risk question.
2. Wraps that reasoning chain in a self-consistency loop that runs several independent samples and picks the most frequent final answer.

The question is deliberately simple: “Given a leveraged ETF that moves by a certain percentage, what is the approximate portfolio impact?” In real systems you would plug this into a richer trading workflow, but the structure of the example generalises to more complex tasks.

```

from .config import OPENAI_API_KEY
from collections import Counter
import os
import re
from typing import List, Tuple

from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your environment or .env file.")

--- Chain-of-thought prompt -----
cot_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You are a cautious trading assistant. "
 "You explain your reasoning before giving a concise final answer.\n\n"
 "When you answer, follow this format exactly:\n"
 "THINKING:\n"
 "<step-by-step reasoning here>\n"
 "ANSWER:\n"
 "<short final answer here>"
),
),
 (
 "human",
 (
 "A client holds a 3x leveraged ETF on a bank index. "
 "The index falls by {index_move_percent}%. "
 "The ETF is {position_weight_percent}% of the portfolio value.\n\n"
 "What is the approximate percentage impact of this ETF position "
 "on the total portfolio, and why?"
),
),
],
)
cot_model = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.2
)
cot_chain = cot_prompt | cot_model

def extract_final_answer(text: str) -> str:
 # Extract the short final answer from the model output.
 # We expect the model to follow:
 # THINKING:
 # ...
 # ANSWER:
 # <short final answer>
 # This helper is intentionally simple. For production use, consider a
 # structured output format or an explicit JSON schema.

 # Split on "ANSWER:" and take the part after it, if present.
 match = re.split(r"ANSWER:\s*", text, maxsplit = 1, flags = re.IGNORECASE)
 if len(match) == 2:
 return match[1].strip()
 return text.strip()

def run_cot_once(index_move_percent: float, position_weight_percent: float) -> Tuple[str, str]:
 # Run a single chain-of-thought reasoning pass.
 # Returns a tuple: (full_output, final_answer).
 result = cot_chain.invoke(
 {
 "index_move_percent": index_move_percent,
 "position_weight_percent": position_weight_percent,
 }
)
 full_text = result.content
 final_answer = extract_final_answer(full_text)
 return full_text, final_answer

--- Self-consistency wrapper -----
def self_consistent_answer(
 index_move_percent: float,
 position_weight_percent: float,
)

```

```

num_samples: int = 7,
temperature: float = 0.7,
) -> Tuple[str, List[Tuple[str, int]]]:
 # Run the CoT chain multiple times with higher temperature and return
 # a self-consistent final answer.
 # Args:
 # index_move_percent: Move of the underlying index, e.g. -5.0 for -5%.
 # position_weight_percent: Portfolio weight of the ETF, e.g. 20.0 for 20%.
 # num_samples: How many reasoning paths to sample.
 # temperature: Sampling temperature; higher values produce more diverse reasoning.
 # Returns:
 # A tuple:
 # - selected_answer: the most frequent final answer across samples.
 # - answer_counts: a list of (answer, count) pairs, sorted by count descending.

 # Use a separate model instance so we can adjust temperature for sampling
 sampling_model = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = temperature,
)
 sampling_chain = cot_prompt | sampling_model
 answers: List[str] = []

 for _ in range(num_samples):
 result = sampling_chain.invoke(
 {
 "index_move_percent": index_move_percent,
 "position_weight_percent": position_weight_percent,
 }
)
 full_text = result.content
 final_answer = extract_final_answer(full_text)
 answers.append(final_answer)

 counts = Counter(answers)
 # Most common returns list of (answer, count) pairs
 most_common = counts.most_common()

 if not most_common:
 raise RuntimeError("No answers generated during self-consistency sampling.")

 selected_answer = most_common[0][0]
 # Convert Counter items to a stable list of tuples for inspection / logging
 answer_counts = [(ans, cnt) for ans, cnt in most_common]

 return selected_answer, answer_counts

--- Example usage -----
def main():
 # Demonstrate both a single CoT run and the self-consistent aggregation.
 # Scenario:
 # - 3x leveraged ETF on a bank index.
 # - Index falls by 4%.
 # - ETF is 15% of total portfolio value.

 index_move = -4.0
 position_weight = 15.0

 print("== Single chain-of-thought run ==")
 full_output, final_answer = run_cot_once(index_move, position_weight)
 print(full_output)
 print("\nParsed final answer:")
 print(final_answer)

 print("\n== Self-consistent answer over multiple runs ==")
 selected, counts = self_consistent_answer(
 index_move_percent = index_move,
 position_weight_percent = position_weight,
 num_samples = 9,
 temperature = 0.8,
)

 print("\nSelected final answer (self-consistent):")
 print(selected)

 print("\nAnswer distribution:")
 for ans, cnt in counts:
 print(f"{cnt} x {ans}")

if __name__ == "__main__":

```

```
main()
```

```
ch_05\scr\cot_and_self_consistency_trading.py
```

The `cot_prompt` template instructs the model to separate reasoning (“THINKING”) from the final conclusion (“ANSWER”) and to explain its reasoning before answering. This is a direct implementation of CoT as a prompt-level pattern, using a standard `ChatPromptTemplate` composed with `ChatOpenAI`.

A single CoT pass for one input `run_cot_once` demonstrates the simplest use: one call to the chain, low temperature, and a helper that extracts the final answer from the “ANSWER:” section. For many tasks, this alone is enough: you get both an explicit reasoning trace and a concise answer you can log or show to a user.

Self-consistency as repeated CoT sampling plus voting `self_consistent_answer` wraps the same CoT prompt in a higher-temperature model, runs it several times, and collects only the final answers. A `Counter` implements the majority vote. This corresponds to the self-consistency pattern described in the literature: explore multiple reasoning paths, then pick the most frequent answer as a proxy for correctness.

Control flow lives outside the model: The reasoning logic stays inside the prompt; the self-consistency logic is expressed as ordinary Python control flow around the LCEL chain. This matches the way the LangChain ecosystem positions CoT and self-consistency: not as special primitives, but as combinations of prompt templates, models, and simple orchestration code.

In a more complete trading assistant, the same pattern can be embedded as a node in a LangGraph workflow, for example as a “high-stakes check” node used only when a decision crosses a certain risk threshold, or as part of an offline evaluation pipeline that compares single-pass answers to self-consistent answers over a test set.

## 5.6 TREE-OF-THOUGHT PROMPTING AND STRUCTURED REASONING PATHS

Tree-of-thought (ToT) prompting is a way to help a model explore several lines of reasoning instead of following only one. Chain-of-thought (CoT) asks the model to “think step by step” along a single path. ToT keeps the idea of visible intermediate steps, but lets the model branch, try alternative ideas, compare them, and then choose the best one. This is useful when there is no single obvious solution and you want the system to weigh options before answering.

### 5.6.1 From linear chains to branching trees

With CoT, the model produces one chain of steps. Each step builds on the previous one. If it makes an early mistake, that mistake often affects the rest of the reasoning.

ToT turns this single chain into a tree. You still have intermediate thoughts written in natural language (a partial explanation, a sub-plan, a hypothesis), but from any of these thoughts the model can propose several different “next steps” instead of just one. Each next step becomes a child node in the tree. You can think of a ToT run as a small search process:

1. The model generates several possible continuations from the current state, not just one.
2. Each continuation is checked against simple criteria such as “does this look correct?”, “is this feasible?”, or “does this respect the constraints?”.
3. Promising continuations are expanded further; weak ones are dropped or moved to the bottom of the queue.
4. The process stops when one path clearly gives a complete and acceptable answer or plan.

This differs from self-consistency. In self-consistency you run several independent CoT traces and vote on the final answer at the end. In ToT, branches are related: evaluations at one step affect which branches you keep exploring, so later steps depend on what happened earlier in the tree.

### 5.6.2 Anatomy of a thought tree

In practice, ToT is not a single magic prompt but a small system with four parts:

1. Problem state: everything the system knows at a given moment, such as the original question, retrieved documents, and the partial reasoning or plan so far.
2. Thought generator: a prompt that asks the model to propose several next thoughts or candidate solutions from the current problem state.
3. Evaluator: a mechanism that rates or filters those thoughts, using either the same model with a different prompt or a separate “judge” model, plus simple rules from your domain.
4. Search strategy: a policy that decides which candidates to expand next and when to stop exploring.

One common way to implement this is as a pipeline of stages. A first stage generates several candidate solutions. A second stage evaluates them (pros, cons, feasibility, expected impact). A third stage adds more detail to the best candidates, for example concrete steps, risks, and mitigations. A final stage ranks or selects a single solution to present. Each stage reads the text produced by the previous one, so you can log and inspect intermediate solutions, evaluations, and rankings.

### 5.6.3 When tree-of-thoughts helps

ToT is aimed at problems where a single straight-line explanation is not enough or not reliable. This happens often in the three domains used in the book:

1. Support: choosing among several possible remedies for a complex case (credit, replacement, plan change, cancellation), each with different rules and impact on the customer relationship.
2. E-commerce: designing campaigns for a category while balancing stock levels, margins, seasons, and customer segments, where several bundle or discount strategies might work.
3. Trading: suggesting portfolio changes under risk limits, sector caps, and transaction costs, where different combinations of trades can meet the same target.

In these situations ToT brings three main advantages: the system can spend more steps “thinking” before deciding, it can backtrack from weak branches instead of being stuck with them, and it can keep genuinely different strategies alive for comparison (for example “minimise churn risk” versus “maximise short-term margin”).

For simple tasks such as short extractions, basic classifications, or direct factual questions, this extra machinery is usually unnecessary. Zero-shot, few-shot, or a single CoT trace is normally enough, and ToT would only add cost and latency without clear benefit.

#### 5.6.4 Costs, limits, and search control

The main drawback of ToT is how quickly the search space grows. If each node generates  $k$  new candidates and you search to depth  $d$ , the number of candidates at the deepest level is  $k^d$ , and the total number of nodes in the tree up to that depth is  $1 + k + k^2 + \dots + k^d = (k^{(d+1)} - 1) / (k - 1)$ . With  $k = 3$  and  $d = 5$ , the deepest level has  $3^5 = 243$  nodes, and the full tree up to that depth has 364 nodes. Depending on how you implement generation and scoring, this can lead to many model invocations per query. To keep this under control, real systems add limits:

1. Branching factor: cap how many candidate thoughts each step produces (for example, “return at most three options”).
2. Depth: cap how many steps the search can continue (for example, “stop after four steps” even if the answer is imperfect).
3. Budget: set a hard limit on total tokens and/or total model calls per query and stop when that budget is used.

More advanced designs build on classical search ideas. Monte Carlo Tree Search (MCTS) is a common reference point: it repeatedly selects promising nodes using past results, runs simulations/rollouts from selected nodes, evaluates the outcomes, and then propagates those evaluations back up the tree to guide future selection. ([Medium][3]) Language Agent Tree Search (LATS) is one example that applies MCTS-style search in an LLM-agent setting. The underlying idea is the same: explore multiple branches, but concentrate most of the compute on the ones that look most promising.

In production, it is usually best to start with conservative settings (few branches, shallow depth, strict budgets), measure both quality and cost, and only then decide whether loosening those limits is worth it.

#### 5.6.5 Role in agentic systems and relation to other techniques

Later chapters describe agents as looping through thought → action → observation. CoT improves the “thought” step by making the reasoning trace explicit. Self-consistency improves it further by running several traces and aggregating their final answers. ToT goes one step beyond: it organises the agent’s thoughts as a tree that the agent can grow, prune, and revisit

inside the loop. A planner can maintain several partial plans, update them as new observations come in, and decide which plan to deepen or discard before acting.

LangGraph gives a concrete way to model this. A ToT-style agent graph typically has:

1. A node that generates several possible next actions or sub-plans from the current state,
2. A node that scores or filters these options,
3. Routing logic that picks which option to execute next or decides that the agent has enough information to produce a final answer.

Compared with ad-hoc prompts, this has two clear advantages. The reasoning process becomes inspectable and testable, because you can log candidate branches, evaluations, and decisions and analyse where the reasoning went wrong. The same graph can also be tuned into simpler modes: with branching disabled it behaves like plain CoT, with repeated independent passes it approximates self-consistency, and with branching plus limits it behaves like a full ToT search. As a practical rule of thumb:

1. CoT is a good default when you mainly want a clear breakdown of reasoning and can live with a single path.
2. Self-consistency is useful when there should be one correct answer and you can afford several runs to increase reliability.
3. ToT is worth the extra cost when the assistant must explore and compare alternative strategies before deciding, and the consequences of a poor choice justify more deliberate reasoning.

In the graph-based assistants that appear later in the book, ToT is used in this practical sense: as a way to design graphs with branching thought-generation nodes, evaluator nodes for pruning, and explicit limits on how much “thinking” time each request receives.

### Example: Tree-of-thought campaign planner for an e-commerce assistant

This example shows a shallow but complete tree-of-thought style reasoning path in the e-commerce domain using only LangChain. The assistant receives a product category, a business goal, and a set of constraints. It first generates several alternative campaign ideas (branches), then evaluates each idea using a rubric, and finally selects the best idea and explains it to a stakeholder. The “tree” here has a single branching level, but the structure matches the pattern described in the chapter: generation of multiple thoughts, evaluation of those thoughts, and a simple search step that chooses the best branch.

```
from .config import OPENAI_API_KEY
import json
from typing import List, Dict, Any
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

def build_model() -> ChatOpenAI:
 # Build and return the ChatOpenAI model used across all chains.
 if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

 tot_model = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.2
)
 return tot_model

def build_generator_chain(model: ChatOpenAI):
 # Thought generator: Given the problem state, propose several alternative campaign
 # ideas.
 generator_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (

```

```

 "You are a senior e-commerce strategist for an online retailer.\n"
 "Given a product category, a business goal, and constraints,"
 "you must propose exactly three distinct campaign ideas.\n"
 "Label them A, B, and C.\n"
 "Each idea must be one or two short paragraphs and clearly\n"
 "actionable."
)
),
(
 "human",
 (
 "Product category: {product_category}\n"
 "Business goal: {goal}\n"
 "Constraints: {constraints}\n\n"
 "Propose three distinct campaign ideas (A, B, C)."
)
),
]
)
return generator_prompt | model

def build_evaluator_chain(model: ChatOpenAI):
 # Evaluator:
 # Score each idea using a simple rubric and return JSON so we can parse it.
 # The model reads the raw ideas text and outputs a JSON array like:
 # [
 # {"id": "A", "score": 8, "rationale": "..."},
 # {"id": "B", "score": 6, "rationale": "..."},
 # {"id": "C", "score": 9, "rationale": "..."}
 #]
 evaluator_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You are evaluating e-commerce campaign ideas for a retail "
 "business.\n"
 "Rate each idea on a scale from 1 to 10 based on:\n"
 "- expected impact on the stated business goal,\n"
 "- feasibility under the constraints,\n"
 "- risk of harming customer experience.\n"
 "Higher scores are better.\n\n"
 "Return your evaluation as JSON only, no extra text.\n"
 "The JSON must be an array of objects with fields:\n"
 " - id (A, B, or C), score (integer 1-10), rationale (short string)."
)
),
 (
 "human",
 (
 "Product category: {product_category}\n"
 "Business goal: {goal}\n"
 "Constraints: {constraints}\n\n"
 "Here are the campaign ideas:\n"
 "-----\n"
 "{ideas_text}\n"
 "-----\n\n"
 "Evaluate ideas A, B, and C and output JSON as specified."
)
)
]
)
 return evaluator_prompt | model

def build_explainer_chain(model: ChatOpenAI):
 # Explainer:
 # Turn the winning idea into a concise explanation for a business stakeholder.
 explainer_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You are a senior product manager.\n"
 "Explain the chosen campaign idea clearly to a non-technical "
 "stakeholder.\n"
 "Focus on what the campaign does and why it is a good fit "
 "for the goal and constraints."
)
)
]
)

```

```

 (
 "human",
 (
 "Product category: {product_category}\n"
 "Business goal: {goal}\n"
 "Constraints: {constraints}\n\n"
 "All campaign ideas (A, B, C):\n"
 "{all_ideas}\n\n"
 "Evaluations (JSON):\n"
 "{evaluations_json}\n\n"
 "The best idea according to the evaluation is: {best_id}.\n"
 "Explain this idea and justify briefly why it is the best choice."
)
)
]
)
return explainer_prompt | model

def parse_evaluations(json_text: str) -> List[Dict[str, Any]]:
 # Parse the evaluator's JSON output safely.
 # Raises ValueError if parsing fails or if the structure is not as expected.
 data = json.loads(json_text)
 if not isinstance(data, list):
 raise ValueError("Expected a JSON array of evaluations.")
 for item in data:
 if not isinstance(item, dict):
 raise ValueError("Each evaluation must be a JSON object.")
 if "id" not in item or "score" not in item:
 raise ValueError("Each evaluation must have 'id' and 'score' fields.")
 return data

def select_best_idea(evaluations: List[Dict[str, Any]]) -> Dict[str, Any]:
 # Search strategy:
 # Pick the idea with the highest score.
 # If there is a tie, the first with that score wins.
 return max(evaluations, key = lambda e: e["score"])

def main():
 # Minimal end-to-end example of a tree-of-thought style reasoning path:
 # 1. Generate several candidate campaign ideas (branching).
 # 2. Evaluate and score each idea (thought evaluation).
 # 3. Select the best idea via a simple search step.
 # 4. Explain the chosen idea to a stakeholder.

 # Example problem state
 problem = {
 "product_category": "running shoes",
 "goal": "increase repeat purchases from existing customers over the next quarter",
 "constraints": "avoid deep discounts, respect existing loyalty tiers, no changes to shipping policy",
 }

 model = build_model()

 # 1. Generate candidate ideas
 generator_chain = build_generator_chain(model)
 ideas_msg = generator_chain.invoke(problem)
 ideas_text = ideas_msg.content
 print("== Generated campaign ideas ==")
 print(ideas_text)
 print()

 # 2. Evaluate ideas with a rubric and JSON output
 evaluator_chain = build_evaluator_chain(model)
 eval_msg = evaluator_chain.invoke(
 {
 **problem,
 "ideas_text": ideas_text,
 }
)
 print("== Raw evaluations (JSON) ==")
 print(eval_msg.content)
 print()

 evaluations = parse_evaluations(eval_msg.content)
 best = select_best_idea(evaluations)

 # 3. Explain the chosen idea to a stakeholder
 explainer_chain = build_explainer_chain(model)
 explanation_msg = explainer_chain.invoke(
 {

```

```

 **problem,
 "all_ideas": ideas_text,
 "evaluations_json": json.dumps(evaluations, indent=2),
 "best_id": best["id"],
 }
}

print("==== Selected best idea ===")
print(f"ID: {best['id']}, score: {best['score']} ")
print()
print("==== Stakeholder explanation ===")
print(explanation_msg.content)

if __name__ == "__main__":
 main()

```

ch\_05\scr\tree\_of\_thought\_campaign\_planner.py

Now execute the code:

```

(.venv) C:\Users\yourname\llm-app> python -m src.
tree_of_thought_campaign_planner
==== Generated campaign ideas ===

A. Launch a "Shoe Refresh Reminder" campaign that targets existing customers
with personalized emails or app notifications timed around the average lifespan
of their purchased running shoes (e.g., 3-6 months after purchase). Include
educational content about when to replace running shoes for optimal performance
and injury prevention and offer an exclusive early access window to new
arrivals or limited-edition models to incentivize repeat purchases without
discounting.

B. Create a "Run & Reward Challenge" exclusively for existing customers, where
they log their runs via your app or partner fitness trackers. Customers who
reach certain mileage milestones within the quarter earn points redeemable for
non-discount rewards such as branded running gear, priority access to new
products, or free customization options (e.g., personalized insoles). This
encourages repeat engagement and purchases while respecting loyalty tiers and
avoiding price cuts.

C. Introduce a "Shoe Upgrade Program" that invites loyal customers to trade in
their gently used running shoes for credit toward their next purchase. Promote
this program through targeted communications emphasizing sustainability and
shoe performance benefits. The credit can be a fixed amount or tier-based,
ensuring it doesn't conflict with existing loyalty discounts, and encourages
customers to return for new shoes regularly without deep discounts or shipping
changes.

==== Raw evaluations (JSON) ===

[
 {
 "id": "A",
 "score": 9,
 "rationale": "Highly relevant timing and personalized approach likely to
boost repeat purchases; no discounts and respects loyalty tiers; low risk to
customer experience."
 },
 {
 "id": "B",
 "score": 8,
 "rationale": "Engaging challenge encourages ongoing interaction and repeat
purchases without discounts; feasible with app integration; moderate complexity
but low risk."
 },
]

```

```

{
 "id": "C",
 "score": 7,
 "rationale": "Sustainability-focused trade-in program incentivizes repeat purchases; may require operational setup; credit system respects loyalty tiers; slight risk if credit perceived as discount."
}
]

==== Selected best idea ====
ID: A, score: 9
==== Stakeholder explanation ====
The chosen campaign, called the "Shoe Refresh Reminder," is designed to encourage our existing customers to buy new running shoes at the right time—typically 3 to 6 months after their last purchase, which aligns with the average lifespan of running shoes. We will send personalized emails or app notifications around this timeframe, educating customers on why replacing their shoes regularly is important for performance and injury prevention. To motivate them further, we'll offer exclusive early access to new or limited-edition shoe models, creating a sense of privilege without resorting to discounts.

This campaign is the best fit for our goal of increasing repeat purchases because it targets customers precisely when they are most likely to need new shoes, making the message highly relevant and timely. It respects our existing loyalty tiers by providing exclusive access rather than price cuts, and it avoids any changes to shipping policies or deep discounts, which aligns perfectly with our constraints. Overall, it offers a low-risk, customer-friendly approach that leverages personalization and education to drive repeat sales effectively.

```

This example is a three-stage LCEL pipeline wrapped in a small “generate → evaluate → select → explain” loop.

`build_model()` creates one shared ChatOpenAI instance. The model is reused across all chains, which keeps behaviour consistent (same model, same temperature) and avoids repeating configuration.

Each `build_*_chain` function returns a Runnable built from a `ChatPromptTemplate` piped into the same model. That is the key structural idea: prompts are defined independently, and the pipe operator composes them into executable units.

`build_generator_chain()` is the branching step. Its system message enforces an output shape: exactly three ideas labeled A, B, C, each short and actionable. The human message injects the runtime variables (`product_category`, `goal`, `constraints`). When `invoke(problem)` runs, the model returns an AIMessage; the code extracts `ideas_msg.content` to get the raw text for downstream steps.

`build_evaluator_chain()` scores the branches. It defines a rubric (impact, feasibility, customer-experience risk) and forces a strict output contract: JSON only, with an array of objects containing `id`, `score`, and `rationale`. The human message includes `ideas_text` so the evaluator reads the actual generated ideas rather than regenerating them. The strict JSON requirement is what makes the next step automatable rather than relying on brittle string matching.

`parse_evaluations()` is the guardrail. It `json.loads` the evaluator output and validates the shape (list of dicts with `id` and `score`). If the model violates the contract, the program raises a clear error instead of silently selecting a wrong “best” idea.

`select_best_idea()` implements the search step: pick the max score. This keeps selection deterministic and transparent. The tie-breaker is implicit in Python’s `max` behaviour: the first item with the highest score wins.

`build_explainer_chain()` turns the selected branch into stakeholder language. It receives the full ideas text, the evaluations JSON, and the `best_id`, so the model can explain the winning idea while referencing the evaluation context. This is important: the explainer is not asked to “guess” what won; it is given the decision and supporting evidence.

`main()` wires the stages together: generate ideas, evaluate them, parse and select, then explain the chosen idea.

## 5.7 SUMMARY

This chapter treats prompting as an engineering discipline, not a bag of magic phrases. It starts by explaining how LangChain wraps models and why chat models differ from LLM calls. It then walks through the controls that shape output: model choice, temperature, top-p and top-k sampling, candidate counts, repetition penalties, maximum output tokens, stop sequences, plus timeouts and retries.

Prompts then become reusable components. `PromptTemplate` covers single-string prompts with variables; `ChatPromptTemplate` extends the idea to multi-message prompts with roles (system, user, assistant). The chapter shows how to compose templates, how to insert chat history with placeholders, and how `RunnableWithMessageHistory` makes history selection and storage a runtime concern with persistence options. It recommends structuring prompts into sections so humans and the model can separate persona, task, context, and required format.

Because prompts change, the chapter describes a prompt lifecycle: version prompts like code, test them on inputs, and validate outputs whenever software will parse or act on them. Callback-based token accounting provides cost visibility. Prompt Hub is presented as a way to reuse shared prompts as a dependency. Focused chains illustrate safer workflows for special cases, including `LLMMathChain` for arithmetic, `SQLDatabaseChain` for question-to-SQL with restricted table visibility, and policy-enforcement flows that classify, revise, and verify responses.

The final part covers reasoning strategies. Zero-shot and few-shot prompting teach behaviour with instructions and examples; dynamic few-shot selection uses embeddings and a vector store to retrieve examples per input. Chain-of-thought writes intermediate reasoning steps; self-consistency samples multiple paths and selects a stable answer. Tree-of-thought generalises this into branching candidates, scoring them, selecting a best option, and explaining the decision while bounding compute with limits on branching factor, depth, and budget.

## 6 CONTEXT ENGINEERING

A language model does not “see” your application. It only sees the tokens you include in a single request: your instructions, the user’s message, and whatever extra context your system attaches. Most model APIs are stateless by default, so the model does not carry conversation state forward unless you provide it again in later calls (for example by replaying selected past messages), or you use a platform feature that explicitly persists conversation state for you. For the same reason, a model cannot directly read your databases, logs, files, or internal tools. If it can use external resources, that happens only because your application exposes those resources through tool calling and then feeds the tool outputs back into the model as new input.

If you send only raw user messages, without deliberate context selection, you can run into predictable problems: answers that drift away from the user’s real goal, invented details, repeated explanations, and token usage that grows with every added message. Context engineering is the discipline of deciding what the model should see at each step and why. It is the same kind of design work you apply to APIs and data models: choosing which facts, documents, history snippets, user attributes, and tool outputs matter, deciding how to represent them, and fitting them into a finite context window and cost budget.

This chapter treats context as a first-class part of the architecture. It makes “context” concrete: stable system instructions, the current user input, carefully chosen slices of conversation history, retrieved knowledge, structured application state, and signals such as identity or preferences. It then shows how to turn “everything you could send” into a small, curated bundle for one call: selecting relevant documents, deciding what to replay versus summarise, and using metadata and state to drive retrieval and routing.

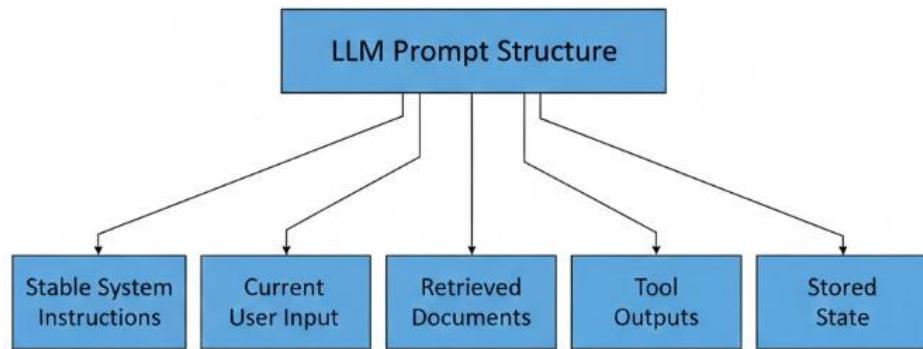
As information grows, the core problem becomes keeping it useful over time. The chapter covers practical ways to manage long conversations and large sources: windowing policies, summarisation strategies, and salience scoring so the model keeps key decisions and constraints while staying within limits. It also explains how different tasks benefit from different context layouts (for example, question answering versus planning or coding).

Finally, it treats context quality as measurable. Instead of tuning by instinct, you evaluate whether your context pipeline improves answers or merely burns tokens, and you iterate using evidence.

## 6.1 WHAT CONTEXT ENGINEERING IS AND WHY IT MATTERS

By this point you have seen the main structural constraints of language models in a basic API setup: each call can only include a finite amount of context, and the model does not retain state between calls unless you explicitly provide it. In addition, without external tools or retrieval, the model cannot reliably answer questions that depend on information beyond what it learned during training. If you simply send the latest user message to the API and rely on the model alone, these constraints can quickly surface as inconsistent answers, invented details, repeated content, and higher costs as you keep stuffing more text into the prompt. Context engineering is the discipline of designing around these constraints—by deciding what to include, what to compress, what to retrieve, and what to validate—rather than pretending they do not exist.

In this book, context engineering means deciding what the model should see, in what form, and at which moment in a workflow. Rather than thinking of “the prompt” as a single blob of text, you treat the whole informational environment of a model call as something you design: stable system instructions, current user input, retrieved documents, tool outputs, and stored state all become explicit ingredients. Each call to the model is given a small, curated bundle of those ingredients tailored to the current step, not an unstructured dump of everything you have.



This discipline starts from a few simple facts that apply to any LLM you use through LangChain or LangGraph:

- The model only has access to the tokens included in the context of the current call. It does not have automatic long-term memory.
- The context window is finite, and usage is billed by tokens. More tokens usually mean higher cost and often higher latency, and they compete for limited attention.
- The model cannot fetch fresh data or inspect your systems by itself. It only learns about your world through information placed into its context, such as recent turns, retrieved snippets, profile data, and tool outputs.

Context engineering turns these constraints into design questions: which parts of the conversation should be replayed verbatim, which should be summarised, which should be dropped; which documents or records should be retrieved for this query; which tool outputs are worth carrying forward to later steps and which are ephemeral. The aim is always the same: a short, focused, high-value context that gives the model everything it needs for the current decision and nothing it does not.

It is useful to separate the main layers that typically make up this context. At the base you have system-level instructions that set the role, tone, and safety constraints for the assistant. On top of that you add the current user input and a carefully selected slice of interaction history, sometimes in raw form, sometimes as a summary. You then inject external data: retrieved passages from knowledge bases or catalogues, policy snippets, logs, or records from your own databases. Finally you add tool outputs and implicit state such as user identity, preferences, and workflow flags. Context engineering is about selecting, ordering, and compressing material from these layers so that they fit comfortably into the window and remain legible to the model.

The difference with ad-hoc prompt tinkering is mostly one of scope. Prompt engineering, in the narrow sense, focuses on phrasing instructions so the model follows them. Context engineering assumes you will get much larger gains by controlling what evidence and state the model sees. You still care about good instructions, but the bigger wins come from well-designed retrieval, memory, and workflow state: how you chunk and index documents; how you attach metadata; how you summarise old turns; how you annotate tool results so they can be reused; how you trim or refresh context as a conversation or job evolves.

Retrieval-augmented generation (which we'll discuss in the next Volume) is one of the most visible expressions of this idea. Instead of trusting the model's training data to answer a question about your product, you retrieve a small set of relevant passages from your own documentation and include them next to the user query. Document loaders, text splitters, embedding models, and vector stores form a pipeline whose only purpose is to build a better context for the model: not "all documents that might be relevant", but "three or four precise chunks that clearly support an answer".

Memory mechanisms provide a complementary pillar. For short interactions you might simply replay the last few turns. For longer-running assistants you will usually combine a window of recent messages with a compact summary of earlier ones, and, for truly long-lived systems, store past interactions in external stores where they can be searched and re-inserted when needed. Here again, the goal is not to remember everything, but to preserve the right abstractions: user preferences, unresolved issues, key decisions, and other facts that materially change how the assistant should behave in later calls.

LangChain makes these choices explicit instead of leaving them scattered across string concatenations. In practice, context engineering shows up as prompt templates, memory components, retrieval steps, and trimming rules. You decide what gets stored, when it gets retrieved, and which parts of the dialogue are available to the model.

When you design context in this way, several good things follow. Answer quality improves because the model is grounded in specific, up-to-date facts rather than being left to guess. Personalisation improves because the assistant consistently sees the right user-level signals. Latency and cost stay under control because you send only the minimal useful subset of information, not everything you could send. Reliability and auditability improve because you can reconstruct exactly which documents, tool outputs, and state fields were visible when a particular answer was generated. For agentic systems in particular, these properties are not optional: without deliberate control over context, multi-step workflows cannot maintain goals, coordinate across tools, or behave consistently across sessions.

The rest of this chapter takes this high-level idea and breaks it into practical questions. How do you choose and package the right fragments of information for each call? When do you

summarise, when do you trim, and when do you keep full detail? How do you adapt the shape of context to different tasks such as Q&A, planning, coding, or analysis? And how do you know whether your context strategy is actually working? By the end, context engineering should feel less like an abstract concept and more like a concrete part of your application design toolbox, on the same footing as data modelling and API design.

## 6.2 CONTEXT SELECTION AND PACKAGING FOR A SINGLE CALL

### 6.2.1 Normalising context inputs

Before you can select context, you need a small set of shared “container” types for information. In LangChain-style systems, most context you pass to a model ends up as one of three things: Document objects (external knowledge), message objects (conversation), and application state snapshots (long-lived facts from your system).

#### 6.2.1.1 Documents (external knowledge)

In LangChain, a Document is a container for text plus metadata. It is widely used in retrieval workflows to carry retrieved chunks through the pipeline. A Document has:

- page\_content: the text content (often a chunk rather than a full file)
- metadata: a dictionary of user-defined attributes about the content (for example source, title, page number, or other tags)
- id (optional): an identifier for the document

This structure is useful because pipelines can keep the text and its descriptive fields together. Retrievers typically take a string query and return a list of Document objects. Other steps can then transform those Documents (for example, by reducing the text content or filtering by metadata) and you can format the resulting list into whatever “sources” or citation block your application needs.

#### 6.2.1.2 Messages (conversation history)

Conversation history is not just text. It has roles and structure. Each step in the conversation (a user message, an assistant message, or a tool result) is represented as a message object: a BaseMessage instance such as SystemMessage, HumanMessage, AIMessage, or ToolMessage. Each message carries an explicit role plus optional metadata (for example identifiers, timestamps, or tool-call fields), the model receives role-separated messages (and tool results) rather than one concatenated string.

```
Your existing messages (history you already have)
messages = [
 SystemMessage(content = "You are a concise assistant."),
 HumanMessage(content = "Summarise this in one sentence: LangGraph saves
 checkpoints."),
 AIMessage(content = "LangGraph can persist state snapshots (checkpoints) so a
 workflow can resume later.")
]

A template that inserts that history, then adds the new user input for *this* call
prompt = ChatPromptTemplate.from_messages(
 [
 ("system", "Answer concisely."),
 MessagesPlaceholder(variable_name = "history"),
 ("human", "{user_input}")
]
)

final_messages = prompt.format_messages(history = messages, user_input = "Now give one
concrete example.")
```

Using message objects matters because you can manage conversation history as structured items, not as one long string. When history is a list of messages, you can apply policies to whole messages:

- Trimming is easy: you drop the oldest messages without cutting sentences in half.

- Summarisation is clean: you replace many older messages with one summary message.
- Role separation stays correct: system instructions remain `SystemMessage`, user inputs remain `HumanMessage`, assistant replies remain `AIMessage`, and tool results remain `ToolMessage`. Nothing gets mashed into one “blob” where roles are unclear.

This structure also makes your code more predictable. If history is stored as a list of `BaseMessage` objects, “selecting history” becomes a deterministic step: given the full stored list, you produce a specific subset (and maybe a summary) to send with the next call. It also helps to keep two related concerns separate:

1. What you retain between calls (the stored history).
2. What you send to the model for this call (the selected history).

Windowing and summarisation decide what gets sent. The chat history store decides what is kept.

#### 6.2.1.3 Application state (*long-lived facts*)

Long-lived state is information your system must keep over time: customer profiles, orders, preferences, permissions, past transactions, workflow status, and so on. This state should live in your own storage systems, such as database tables, event logs, document stores, or graph databases. You do not paste all of that state into the prompt. Instead, when a request arrives, you fetch only what is relevant for that specific request. Then you compress it into a small, task-focused snapshot the model can use: a few key fields, a small table, or a short structured summary.

Keeping these concepts separate is what makes context engineering disciplined and predictable:

- History is what was said in the conversation so far.
- Documents are what your knowledge sources say (manuals, policies, product catalogs, FAQs).
- State is what your system knows and must remember (facts and records your application owns).

For each model call, you select the needed pieces from history, documents, and state. Then you package them into a prompt layout that stays consistent across calls. This keeps prompts stable, reduces token waste, and makes behaviour easier to test and debug.

#### 6.2.2 The per-call context pipeline: candidates → selection → prompt

The previous section described context engineering as the design of the informational environment around a model call. This section zooms in on a single request: given all the data you could send, which pieces do you choose, and how do you arrange them in the prompt?

In many production systems, this work can be understood as two linked steps. First, the system builds a candidate set of potentially useful context items, such as document chunks, selected conversation history, tool outputs, and application state. Second, it assembles some combination of those items into a compact, structured prompt that fits within the model’s context window and is easy to interpret. Teams usually make these steps explicit and repeatable, so the result is testable and maintainable, rather than relying on ad-hoc string concatenation.

In the rest of this section we make these steps concrete. We normalise different information sources into shared abstractions (documents, messages, and application state). We examine

how retrievers select relevant documents, from lexical search to hybrid retrieval and compression. We then show how retrieved snippets, history, and state can be packaged into clear prompt sections. Finally, we close with guidance on metadata, structured state, and governance.

### 6.2.3 Retrieval strategies for documents

Most external knowledge reaches the model through an index built ahead of time. Large sources are split into chunks, and each chunk is stored as a `Document`: the text you might later include (`page_content`) plus `metadata` that explains where it came from and how it should be used. Chunk size and overlap are chosen so each chunk stands on its own but still fits inside the context window (We'll discuss how to chunk content of any type in the next Volume when RAG is presented).

A retriever is the object that turns a user question into a short list of candidate `Documents`. The key idea is simple: you are not trying to find “everything that might be relevant”. You are trying to find a small set of strong candidates that you can afford to include, and whose provenance you can still trace. Following is a list of retrievers that LangChain makes available for retrieving `Documents`.

#### 6.2.3.1 Lexical retrieval (BM25)

`BM25Retriever` is the closest analogue to traditional search. It ranks documents by term overlap using the Okapi BM25 scoring method (via the `rank_bm25` package). It performs best when the user's question shares exact vocabulary with the source text (product names, policy terms, error codes). It is weaker when users paraphrase heavily, use synonyms, or describe a concept that is not phrased the same way in your documents.

#### 6.2.3.2 Semantic retrieval (vector stores)

A vector store retriever uses embeddings: it turns both the query and each stored chunk into vectors, then searches for “nearby” vectors. This approach handles paraphrases and conceptual matches better than keyword search because it is matching meaning rather than exact wording. In practice, this is often the default for natural-language questions. The trade-off is that it can miss exact codes and it depends on embedding quality and chunking choices.

#### 6.2.3.3 Diversity and hybrid retrieval (MMR, Merger/Ensemble)

Vector search often returns near-duplicates: several chunks from the same section that all say essentially the same thing. Maximum marginal relevance (MMR) is a selection strategy that balances relevance with diversity. Instead of taking the top-k most similar chunks, it tries to pick a set that covers different aspects of the query. This is useful when you want broader coverage in a fixed context budget, but it can occasionally trade away the single best chunk in order to gain variety.

When query styles vary, a single retriever is rarely enough: some users type exact identifiers, while others describe the same thing loosely. A common pattern is to run a keyword retriever and a semantic retriever in parallel, then combine their results. “Lord of the Retrievers (LOTR)”, also known as `MergerRetriever`, merges the outputs of multiple retrievers into a single list so downstream code can treat them as one retriever. `EnsembleRetriever` combines multiple retrievers and aggregates their ranked outputs using weighted Reciprocal Rank Fusion (RRF). This lets you keep BM25's strength on literal matches while also benefiting from vector similarity when the question is phrased indirectly or uses synonyms. The result is usually higher recall with a small increase in complexity.

#### 6.2.3.4 *Query rewriting and expansion (MultiQuery)*

Sometimes retrieval fails not because the index is bad, but because the query is too narrow or phrased in a way that does not match your corpus. MultiQueryRetriever addresses this by using a model to generate several alternative queries from the original question, retrieving for each variant, and then taking the union of results. This tends to increase recall for vague questions and for domains with inconsistent terminology. The cost is extra latency and compute, and the risk is drift if your rewrite prompt generates queries that are related but not actually relevant.

#### 6.2.3.5 *Metadata-aware retrieval (SelfQuery)*

If your metadata is reliable and meaningful, you can do better than “search everything”. SelfQueryRetriever uses a model to translate a natural-language query into a structured query that includes metadata filters plus a similarity search component. This is valuable when users implicitly specify constraints (“only 2024 incidents”, “only TV”, “only this business unit”), or when you want safety and scope control (“only documents with permission tag X”). The trade-off is that it requires disciplined metadata and a vector store that supports filtering, and you must treat filter generation as something you test and monitor, not as a magical capability.

#### 6.2.3.6 *“Search small, return big” patterns (`ParentDocumentRetriever`, `MultiVectorRetriever`)*

Chunking helps retrieval because smaller pieces tend to match a user question more precisely. The downside is that a “hit” can be too narrow: the model sees the right sentence, but not the surrounding paragraph or section that explains what it means.

`ParentDocumentRetriever` solves this by separating “what you search” from “what you return”. It embeds and searches small “child” chunks for accuracy. When a child chunk matches, it returns the larger “parent” document (or parent chunk) that the child came from, so the model gets enough surrounding context to interpret the match. The important detail is that the parent does not get rebuilt from child chunks at query time. It is prepared and stored during ingestion: you keep the parent text in a document store, split that parent into child chunks, embed only the children for the vector store, and attach a parent identifier to each child so the retriever can fetch the parent later.

`MultiVectorRetriever` solves a similar problem when one document needs more than one “search handle”. It uses a vector store for search and a document store for the content you actually want to return as context. During ingestion, you store the underlying document once in the document store, then write multiple vector entries that all point back to that same document ID (for example, vectors for child chunks plus a separate vector for a summary of the document). At query time it finds relevant vectors, then returns the full document from the document store. The API explicitly supports an ID key for this linkage.

A simple way to remember both patterns is “search small, return big”. Use `ParentDocumentRetriever` when the main issue is that your best match is too thin and you need surrounding context. Use `MultiVectorRetriever` when the main issue is that the same source document should be discoverable through multiple embeddings, while still returning the same underlying document content to the model.

## 6.2.4 Post-retrieval refinement: keep less, keep better

### 6.2.4.1 *Compression vs filtering vs reranking (as distinct outcomes)*

Retrieval often needs a second step. A first-pass retriever is usually tuned for recall, so it brings back a generous set of candidates. The refinement step then improves precision and controls tokens before any evidence is placed into the prompt.

`ContextualCompressionRetriever` is LangChain's built-in wrapper for contextual compression. It runs a base retriever to get candidate `Documents`, then runs a "document compressor" over those candidates using the current query and returns the transformed `Documents`.

"Document compressor" is an API name, not a promise that content will always be shortened. In LangChain's terminology, "compressing" includes two operations: reducing the contents of an individual `Document` and filtering out `Documents` entirely. To keep the mental model clean, separate three outcomes:

- Compression (content changes): The returned `Documents` have shorter `page_content` because the compressor extracts the relevant spans from each candidate. `LLMChainExtractor` is designed for this use: it uses an LLM to extract only statements relevant to the query.
- Filtering changes which documents survive. The retriever may return, say, 20 candidate `Documents`, and a filtering step can remove the ones that are unlikely to help.
  - `EmbeddingsFilter` does this using embeddings. It computes an embedding for the query and compares it to each candidate Document's embedding. If a Document's embedding is not similar enough to the query embedding, that Document is removed.
  - `LLMChainFilter` does this using an LLM. It asks the model whether each candidate Document is relevant to the query. Documents judged "not relevant" are removed. The remaining `Documents` keep their original content; the filter does not rewrite the text.
- Reranking (order changes): A reranker looks at the retrieved candidate documents, gives each one a relevance score for the current query, and then sorts the documents so the most relevant ones come first. Many rerankers also return only the best-scoring results (for example the top-k / top-n documents), so fewer documents continue to the next step. In LangChain, rerankers are exposed as "document compressors", so you can plug them into `ContextualCompressionRetriever` by passing the reranker as `base_compressor`. Other rerankers LangChain provides as document compressors include:
  - `FlashrankRerank` (`FlashRank`)
  - `JinaRerank` (`Jina AI`)
  - `BedrockRerank` (`AWS Bedrock`)
  - `DashScopeRerank` (`DashScope`)
  - `InfinityRerank` (`Infinity`)

When you need multiple refinement steps, `DocumentCompressorPipeline` allows a sequence of transformations and compressors to run in order.

#### 6.2.4.2 DocumentCompressorPipeline (multi-step refinement)

DocumentCompressorPipeline is LangChain's way to run more than one refinement step after retrieval. It is a "document compressor" itself, but instead of doing one operation, it executes a list of steps in order (the transformers list). Each step can be:

- A document transformer (a component that transforms documents without using the query, for example a text splitter that breaks a long Document into smaller chunks), or
- Another document compressor (a component that uses the query to refine the set or contents of documents, for example removing redundant chunks or filtering to only the most relevant ones).

The pipeline is strictly sequential: the Documents produced by step 1 are passed into step 2, then step 3, and so on. This lets you build a clear, repeatable flow such as: split documents into chunks → remove redundant chunks → keep only the chunks most relevant to the query. Because DocumentCompressorPipeline implements the same "document compressor" interface, you can plug it directly into ContextualCompressionRetriever as the base\_compressor. In that setup, ContextualCompressionRetriever first calls your base retriever to get candidate Documents, then passes those candidates through the pipeline and returns the refined Documents.

```
from langchain_core.documents import Document
from langchain_core.vectorstores import InMemoryVectorStore
from langchain_openai import OpenAIEMBEDDINGS
from langchain_text_splitters import RecursiveCharacterTextSplitter
from langchain_community.document_transformers import EmbeddingsRedundantFilter
from langchain_classic.retrievers import ContextualCompressionRetriever
from langchain_classic.retrievers.document_compressors import DocumentCompressorPipeline
from langchain_classic.retrievers.document_compressors import EmbeddingsFilter
from .config import OPENAI_API_KEY

1) A tiny corpus (in a real app this comes from loaders + splitters + indexing)
docs = [
 Document(
 id = "a",
 page_content = (
 "Refund policy: you can request a refund within 30 days of purchase."
 "Refunds are processed to the original payment method."
 "Shipping fees are not refundable."
),
 metadata = {"source": "policy"}
),
 Document(
 id = "b",
 page_content = (
 "Returns policy: items can be returned within 30 days if unused and in
 original packaging."
 "Contact support to obtain an RMA before sending items back."
),
 metadata = {"source": "policy"}
),
 Document(
 id = "c",
 page_content = (
 "Warranty: electronics have a 2-year limited warranty covering manufacturing
 defects. Accidental damage is not covered."
),
 metadata = {"source": "warranty"}
)
]

2) Index the docs into a simple in-memory vector store (good enough for demos/tests)
embeddings = OpenAIEMBEDDINGS()
vector_store = InMemoryVectorStore(embeddings)
vector_store.add_documents(docs)

base_retriever = vector_store.as_retriever(search_kwargs = {"k": 8})

3) Build a multi-step refinement pipeline:
```

```

- split candidates into smaller chunks
- remove near-duplicate chunks
- keep only the top-k chunks most similar to the query
pipeline = DocumentCompressorPipeline(
 transformers = [
 RecursiveCharacterTextSplitter(chunk_size = 120, chunk_overlap = 20),
 EmbeddingsRedundantFilter(embeddings = embeddings),
 EmbeddingsFilter(embeddings = embeddings, k = 3)
]
)

4) Wrap the base retriever with contextual compression (refinement happens after
retrieval)
compression_retriever = ContextualCompressionRetriever(
 base_retriever = base_retriever,
 base_compressor = pipeline
)

query = "Can I get a refund after 30 days?"
refined_docs = compression_retriever.invoke(query)

print("Refined documents (after split -> dedupe -> filter):\n")
for i, d in enumerate(refined_docs, start = 1):
 src = d.metadata.get("source", "unknown")
 print(f"(i). source={src} id = {getattr(d, 'id', None)} ")
 print(d.page_content)
 print("-" * 60)

```

ch\_06\scr\DocumentCompressorPipeline.py

This example shows how LangChain refines retrieved documents before they are used as context by combining ContextualCompressionRetriever with a DocumentCompressorPipeline.

ContextualCompressionRetriever wraps a normal retriever. The base retriever is responsible for recall: given a query, it returns a set of candidate Documents (here, from a vector-store retriever). ContextualCompressionRetriever then performs a second step: it takes those candidate Documents and passes them, along with the same query, to a base\_compressor. The output of the wrapper is therefore not the raw candidates, but the transformed Documents produced by the compressor. The practical effect is simple: retrieval happens first, refinement happens immediately after, and only the refined Documents continue downstream into prompt construction.

DocumentCompressorPipeline is used as the base\_compressor. It is a composition mechanism: instead of choosing one refinement technique, you define a sequence of transformations that run in order, where each step receives the Documents produced by the previous step. In this code, the pipeline performs three refinements.

- RecursiveCharacterTextSplitter(chunk\_size=120, chunk\_overlap=20) breaks each retrieved Document into smaller chunks of roughly 120 characters, with a 20-character overlap between adjacent chunks. The chunk\_size controls the target chunk length; chunk\_overlap intentionally repeats a small tail of text so that relevant information that straddles a boundary is less likely to be split in a way that loses meaning.
- EmbeddingsRedundantFilter(embeddings=embeddings) removes near-duplicate chunks. It uses embeddings to detect chunks that are semantically redundant (similar meaning, different wording or repeated policy text) and drops the extras, so they do not waste context budget.
- EmbeddingsFilter(embeddings=embeddings, k=3) is the final relevance gate. It embeds the query and each remaining chunk, scores similarity, and keeps only the top k

chunks (here, 3). That is where the pipeline enforces a hard cap on how much context will be returned, giving you predictable, token-efficient evidence selection.

### 6.2.5 Ordering and packaging evidence

#### 6.2.5.1 Ordering evidence in long contexts

Even when retrieval returns the right evidence, where that evidence appears in a long prompt can change outcomes. “Lost in the Middle” reports that model performance can degrade when relevant information sits in the middle of long contexts. In practice, this makes ordering a last-mile quality lever when you must include more text.

LangChain includes `LongContextReorder` as a document transformer intended to mitigate this by reordering retrieved documents before they are packed into the prompt. Apply it after retrieval (and after reranking, if you rerank), then package evidence in the reordered sequence. This is not a substitute for selecting fewer, better documents; it is a refinement when a longer context is unavoidable.

`LongContextReorder` does not score documents or “decide relevance” on its own. It assumes the input sequence is already ordered by relevance (as is typical when you pass in the ranked output from a retriever), and it applies a deterministic rearrangement to that sequence. Concretely, the implementation reverses the list, then iterates through it and alternates between inserting the next document at the start of the output list and appending it to the end. This has the effect of pushing lower-ranked documents toward the middle while placing higher-ranked documents toward the beginning and the end of the final order.

Operationally, you should think of it as a “presentation-layer” transform: retrieval decides what comes back; reranking/compression decide what survives; `LongContextReorder` decides where the survivors appear in the final prompt. Because it is purely positional and does not call a model, it is predictable and typically cheap at the small document counts used in RAG. The only requirement is that you apply it to a relevance-sorted list; if the incoming list is not meaningfully ranked, the reordering cannot improve attention, because it has no signal to work with.

#### 6.2.5.2 Packaging retrieved documents

After retrieval and refinement, packaging decides how evidence is presented to the model. This step does not discover new information. It formats and orders the final `Documents` so the model can use them reliably within a context budget. Packaging usually includes:

- Stable ordering rules: Keep a deterministic order (for example, the reranked order from the previous step). Do not rely on incidental ordering from a vector store call.
- Stable formatting: Use a consistent template per `Document` (source title, section/page, chunk id if relevant, then the text). The goal is that the model can parse the context block the same way every time.
- Token budgeting: If the final set still exceeds budget, truncate by a predictable policy (for example, drop the lowest-ranked `Documents` first). This is the point of reranking; it makes the “drop tail” policy less damaging.

#### 6.2.5.3 Prompt construction

After retrieval and optional compression and reordering, the system has a small set of documents whose `page_content` and `metadata` have been tailored to the current question.

The final step is to decide how they appear in the prompt. A common pattern is to create a dedicated “sources” section with a stable structure. Each retrieved snippet is formatted in a consistent way, typically including an index or label (for example [1], [2], [3]), key metadata such as title, category, version, or date, and the snippet text itself, possibly cleaned and truncated with an ellipsis if very long.

[3]

```
(brand: "Apple", category: "phone_case", price_eur: 39.90,
price_eur: in-stock: True, region: "EU")
Title: Apple Silicone Case with MagSafe
```

A durable silicone case designed to protect your iPhone. iPhone. Features integrated magnets for seamless alignment and faster wireless charging. Soft-touch finish feels great in your hand...

...

System and human instructions then refer to this section explicitly. The model is told to answer based only on these sources, to treat them as authoritative for the purpose of the answer, and often to mention source indices when citing facts. If the sources do not contain the answer, the model is instructed to say that it does not know or that the information is not available in the provided context.

This packaging style has several advantages. It makes the token budget explicit, because you control how many snippets appear and how long they can be. It makes answers auditable, because you can see exactly which pieces of text were visible to the model. It stabilises behaviour: once the model has seen the same layout across many calls, it learns where to “look” for relevant information. Different applications tune the level of detail inside this section, but the goal is always the same: a compact, clearly delimited block of retrieved evidence that the model can reliably work from.

### Example - Retrieval variants and contextual compression example

Here is a small support-assistant example that shows how to combine different retrieval strategies, compress the results, and then package them into a compact “sources” section that the model uses to answer a configuration question. To run this example please install the following modules:

```
(.venv) C:\Users\yourname\llm-app> pip install langchain[all] faiss-cpu rank-
bm25

(.venv) C:\Users\yourname\llm-app> python -m pip install "langchain-
classic>=1.0.0" "langchain-community>=0.3.0"
```

```
from .config import OPENAI_API_KEY
from typing import List
from langchain_core.documents import Document
from langchain_community.retrievers import BM25Retriever
from langchain_community.vectorstores import FAISS
from langchain_openai import OpenAIEmbeddings, ChatOpenAI
from langchain_classic.retrievers import EnsembleRetriever, ConextualCompressionRetriever
from langchain_classic.retrievers.document_compressors import LLMChainExtractor
from langchain_core.prompts import ChatPromptTemplate

if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

def build_corpus() -> List[Document]:
 # Create a small product manual + FAQ corpus with metadata.
 docs = [
```

```

Document(
 page_content=(
 "To reset the ACME Router X100 to factory settings, press and hold "
 "the reset button on the back for 10 seconds until the power light "
 "starts blinking. This will erase all custom configuration."
),
 metadata={
 "title": "ACME Router X100 - Factory reset",
 "doc_type": "manual",
 "version": "1.2"
 }
),
Document(
 page_content=(
 "If your router is not responding, try unplugging it for 30 seconds "
 "and plugging it back in. This power cycle does not erase your settings."
),
 metadata={
 "title": "Troubleshooting unresponsive routers",
 "doc_type": "faq",
 "version": "3.0"
 }
),
Document(
 page_content=(
 "You can change the Wi-Fi password from the web interface under "
 "'Settings → Wireless'. Changing the password will disconnect all devices."
),
 metadata={
 "title": "Changing your Wi-Fi password",
 "doc_type": "manual",
 "version": "1.0"
 }
),
Document(
 page_content=(
 "Factory reset restores all options (Wi-Fi name, password, admin "
 "credentials) to their defaults. You will need to set up the router "
 "again."
),
 metadata={
 "title": "FAQ: What does factory reset do?",
 "doc_type": "faq",
 "version": "2.1"
 }
)
)
]
return docs

def build_retrievers(docs: List[Document]):
 # Create lexical, vector, and hybrid (ensemble) retrievers over the same corpus.
 # Lexical retriever (BM25-style)
 bm25_retriever = BM25Retriever.from_documents(docs)
 bm25_retriever.k = 4 # limit how many documents it returns

 # Vector retriever using a FAISS vector store
 embeddings = OpenAIEMBEDDINGS(model = "text-embedding-3-small")
 vectorstore = FAISS.from_documents(docs, embeddings)
 vector_retriever = vectorstore.as_retriever(search_kwargs = {"k": 4})

 # Hybrid retriever that combines lexical + vector results
 ensemble_retriever = EnsembleRetriever(
 retrievers = [bm25_retriever, vector_retriever],
 weights = [0.5, 0.5],
)
 return ensemble_retriever

def build_compression_retriever(base_retriever):
 # Wrap the hybrid retriever with contextual compression.
 llm = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0)
 compressor = LLMChainExtractor.from_llm(llm)

 compression_retriever = ContextualCompressionRetriever(
 base_compressor = compressor,
 base_retriever = base_retriever,
)
 return compression_retriever, llm

def format_sources(docs: List[Document], max_chars: int = 280) -> str:
 # Turn a list of Documents into a compact, numbered sources block.
 lines = []

```

```

for idx, doc in enumerate(docs, start = 1):
 title = doc.metadata.get("title", "Untitled")
 doc_type = doc.metadata.get("doc_type", "unknown")
 version = doc.metadata.get("version", "n/a")

 snippet = doc.page_content.strip().replace("\n", " ")
 if len(snippet) > max_chars:
 snippet = snippet[:max_chars].rstrip() + "..."

 lines.append(
 f"[{idx}] {title} ({doc_type}), v{version})\n{snippet}"
)
return "\n\n".join(lines)

def build_prompt() -> ChatPromptTemplate:
 # Define a prompt that uses a sources section and asks for source-based answers.
 prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You are a helpful support assistant for ACME routers."
 "Answer the question using only the information in the sources."
 "If the sources do not contain the answer, say you do not know."
)
),
 (
 "human",
 (
 "Question:\n{question}\n\n"
 "Sources:\n{sources}\n\n"
 "Answer the question. When you refer to a fact, mention the "
 "source index like [1], [2], etc."
)
)
]
)
 return prompt

def answer_question(question: str):
 # Run retrieval + compression, then call the model with a structured sources block.
 docs = build_corpus()
 ensemble_retriever = build_retrievers(docs)
 compression_retriever, llm = build_compression_retriever(ensemble_retriever)
 prompt = build_prompt()

 # 1) Selection: hybrid retrieval + contextual compression
 compressed_docs = compression_retriever.invoke(question)

 # 2) Packaging: build the sources string and inject it into the chat prompt
 sources_str = format_sources(compressed_docs)
 messages = prompt.format_messages(question = question, sources = sources_str)
 response = llm.invoke(messages)

 return response.content, compressed_docs, sources_str

if __name__ == "__main__":
 user_question = "How do I reset my ACME router to factory settings?"
 answer, used_docs, sources_block = answer_question(user_question)

 print("==== Answer ===")
 print(answer)
 print("\n==== Sources sent to the model ===")
 print(sources_block)

```

ch\_06\scr\variants\_and\_contextual\_compression.py

```

(.venv) C:\Users\yourname\llm-app> python -m
src.variants_and_contextual_compression

==== Answer ===

To reset your ACME router to factory settings, press and hold the reset button
on the back of the router for 10 seconds until the power light starts blinking.
This will erase all custom configuration and restore the router to its default
settings [1]. After the reset, you will need to set up the router again, as all
options like Wi-Fi name, password, and admin credentials will be restored to
their defaults [2].

==== Sources sent to the model ===

```

```
[1] ACME Router X100 - Factory reset (manual, v1.2)
To reset the ACME Router X100 to factory settings, press and hold the reset
button on the back for 10 seconds until the power light starts blinking. This
will erase all custom configuration.

[2] FAQ: What does factory reset do? (faq, v2.1)
Factory reset restores all options (Wi-Fi name, password, admin credentials) to
their defaults. You will need to set up the router again.
```

Let's describe the code. First, it defines a tiny corpus with explicit provenance. `build_corpus` creates four `Document` objects that represent two manual excerpts and two FAQ entries. Each `Document` has `page_content` with the actual support text and a metadata `dict` with `title`, `doc_type`, and `version`. That metadata is not just decoration: it is carried through retrieval and ends up in the `sources` block so the model (and you) can see what kind of source was used and which version it came from.

Next, `build_retrievers` constructs two independent retrieval signals over the same corpus. The `BM25Retriever` is the lexical path: it ranks documents by term overlap. In parallel, the vector retriever is the semantic path: `build_retrievers` creates `OpenAIEmbeddings("text-embedding-3-small")`, embeds the documents into a FAISS vector store, and exposes it as a retriever that returns the top-k semantically similar chunks.

`EnsembleRetriever` is configured with the BM25 retriever and the FAISS retriever and equal weights. Conceptually, this means “collect candidates from both worlds and blend them into one ranking”, so you get both exact-match precision and paraphrase-friendly recall. In practice, this is the safest default for customer support corpora, because users mix styles: some type a phrase they saw on the device, others describe symptoms in their own words.

Then `build_compression_retriever` adds the second selection stage: contextual compression. The base retriever (the ensemble) can still return chunks that are only partially relevant or longer than you want to spend tokens on. This function creates a low variance chat model (`ChatOpenAI("gpt-4.1-mini", temperature=0)`) and wraps it into an `LLMChainExtractor`. The extractor reads each retrieved `Document` in the context of the user question and pulls out only the sentences that directly help answer it. `ContextualCompressionRetriever` wires that together: retrieval produces candidates; compression trims them into “evidence snippets” that are cheaper to send and easier to audit.

`format_sources` is the packaging layer that turns selected `Documents` into a stable, readable evidence bundle. It iterates over the final `Document` list, formats a numbered entry like “[1] Title (doc\_type, vX.Y)”, and includes a short snippet of content trimmed to `max_chars`. The key detail is the numbering: it becomes the citation scheme you instruct the model to use. You are not asking the model to be honest in the abstract; you are giving it an explicit way to anchor claims to the context you provided.

`build_prompt` defines the prompt layout and the grounding rules. The system message fixes the role (“support assistant for ACME routers”) and adds a hard constraint: answer using only the sources, otherwise say you do not know. The human message then provides two blocks—Question and Sources—and explicitly asks the model to cite sources using indices like [1] and [2]. This prompt structure is doing the final piece of context engineering: it makes the model treat retrieved text as evidence, not as optional inspiration.

Finally, `answer_question` orchestrates the end-to-end flow. It builds the base retriever, wraps it with contextual compression, and builds the prompt template. For the user question, it calls `compression_retriever.invoke(question)` to retrieve and compress the candidate context, then formats the resulting `Documents` into `sources_str`, injects the question plus sources into the prompt, invokes the chat model, and returns the answer along with the selected `Documents` and the exact sources block that was sent. This structure is deliberate: it makes each run inspectable and debuggable, and it keeps the “what did we show the model?” boundary explicit.

#### 6.2.6 Using metadata and structured state

Metadata and structured state influence both which candidates you select and how you present them. For selection, metadata drives filtering and routing. A support assistant can restrict retrieval to documents tagged with the user’s product and region, and to policies that are still in force. Another application can choose different catalogues, pricing documents, or knowledge bases depending on country, customer type, or user role. A research or trading assistant can retrieve only those notes that apply to instruments or entities that are present in the current portfolio or project.

For packaging, metadata provides labels and structure. Including a document’s title, date, or source improves traceability and helps both the model and human reviewers understand where facts come from. Including user-level attributes such as role, preferences, and risk tolerance helps the model adapt tone and strategy. The key is to keep these annotations short and stable, so they enhance the context rather than consuming most of it.

#### Example - Metadata-aware retrieval for a portfolio risk assistant example

This example shows how to assemble context for a trading risk summary by combining three inputs: research and policy documents, a client portfolio, and a risk profile. Documents are normalised into `Document` objects with rich metadata and stored in a vector index, while portfolio and profile are turned into terse text blocks. At query time, metadata filters derived from the current positions and region drive retrieval, and the selected snippets are packaged into a compact “sources” section that the model must rely on when writing its risk summary. To run this example please install the following modules:

```
(.venv) C:\Users\yourname\llm-app> pip install chromadb
```

```
from .config import OPENAI_API_KEY
from datetime import date
from typing import List, Dict, Any
from langchain_core.documents import Document
from langchain_openai import OpenAIEMBEDDINGS, ChatOpenAI
from langchain_community.vectorstores import Chroma
from langchain_core.prompts import ChatPromptTemplate

Corpus: research notes and risk policies with rich metadata

def build_research_corpus() -> List[Document]:
 # Create a small corpus of research and policy documents.
 # Each document has metadata we can use for filtering.
 return [
 Document(
 page_content = (
 "AAPL is a large-cap US tech stock with strong cash flows. "
 "Current risks include concentration in a few product lines "
 "and exposure to US consumer demand."
),
 Metadata = {
 "title": "AAPL risk overview Q1",
 "date": date.today(),
 "product": "AAPL"
 }
)
]
```

```

 "instrument": "AAPL",
 "region": "US",
 "doc_type": "research_note",
 "published_at": "20250210"
 }
),
Document(
 page_content = (
 "TSLA remains highly volatile and sensitive to changes in interest "
 "rates and EV subsidies. Position sizing should reflect this."
),
Metadata = {
 "title": "TSLA volatility and macro sensitivity",
 "instrument": "TSLA",
 "region": "US",
 "doc_type": "research_note",
 "published_at": "20250120"
}
),
Document(
 page_content = (
 "For retail clients in the EU with medium risk tolerance, "
 "single-stock positions should generally not exceed 10% of "
 "total portfolio value."
),
Metadata = {
 "title": "EU retail risk policy",
 "instrument": "ALL",
 "region": "EU",
 "doc_type": "risk_policy",
 "published_at": "20241101"
}
),
Document(
 page_content = (
 "For retail clients in the US with medium risk tolerance, "
 "single-stock positions should generally not exceed 8% of "
 "total portfolio value."
),
Metadata = {
 "title": "US retail risk policy",
 "instrument": "ALL",
 "region": "US",
 "doc_type": "risk_policy",
 "published_at": "20241215"
}
)
]

def build_vectorstore(docs: List[Document]) -> Chroma:
 # Build a simple in-memory vector store that supports metadata filters.
 if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

 embeddings = OpenAIEmbeddings(model = "text-embedding-3-small")
 vectorstore = Chroma.from_documents(
 documents = docs,
 embedding = embeddings,
 collection_name = "trading_research"
)
 return vectorstore

Structured state: portfolio and user profile → compact text blocks

def build_sample_portfolio() -> Dict[str, Any]:
 return {
 "positions": [
 {"symbol": "AAPL", "size": 120, "entry_price": 175.0},
 {"symbol": "TSLA", "size": 40, "entry_price": 220.0}
]
 }

def build_sample_profile() -> Dict[str, Any]:
 return {
 "risk_tolerance": "medium",
 "base_currency": "USD",
 "region": "US"
 }

```

```

def format_portfolio_snapshot(portfolio: Dict[str, Any]) -> str:
 # Turn a portfolio dict into a terse, three-line-style snapshot.
 lines: List[str] = []
 for pos in portfolio.get("positions", []):
 symbol = pos["symbol"]
 size = pos["size"]
 entry_price = pos["entry_price"]
 lines.append(f"- {symbol}: ({size}) shares @ {entry_price:.2f}")
 if not lines:
 return "No open positions."
 return "\n".join(lines)

def format_profile_summary(profile: Dict[str, Any]) -> str:
 # Turn user profile into one or two short sentences.
 risk = profile.get("risk_tolerance", "unknown")
 currency = profile.get("base_currency", "USD")
 region = profile.get("region", "US")
 return (
 f"Client region: {region}. Base currency: {currency}. "
 f"Declared risk tolerance: {risk}."
)

Retrieval and packaging: metadata filters + state-driven blocks

def retrieve_filtered_docs(
 vectorstore: Chroma,
 question: str,
 instruments: List[str],
 region: str,
 min_date: int
) -> List[Document]:
 # Use metadata filters so we only retrieve documents relevant to
 # the current positions and region.
 # - instrument in user_instruments OR generic policy docs (instrument == 'ALL')
 # - region matches the user's region
 # - published_at >= min_date (numeric YYYYMMDD)
 filter_dict: Dict[str, Any] = {
 "$and": [
 {
 "$or": [
 {"instrument": {"$in": instruments}},
 {"instrument": {"$eq": "ALL"}}
]
 },
 {"region": {"$eq": region}},
 {"published_at": {"$gte": min_date}}
]
 }
 docs = vectorstore.similarity_search(
 question,
 k = 4,
 filter = filter_dict
)
 return docs

def format_sources_block(docs: List[Document]) -> str:
 # Format retrieved documents into a compact 'sources' section.
 if not docs:
 return "No recent research or risk policies available."
 lines: List[str] = []
 for idx, d in enumerate(docs, start = 1):
 meta = d.metadata
 title = meta.get("title", "Untitled")
 instrument = meta.get("instrument", "N/A")
 region = meta.get("region", "N/A")
 published_at = meta.get("published_at", "N/A")
 snippet = d.page_content.strip().replace("\n", " ")
 if len(snippet) > 240:
 snippet = snippet[:240].rstrip() + "..."
 lines.append(
 f"[{idx}] {title} (instrument={instrument}, region={region}, "
 f"published_at={published_at})\n{snippet}"
)
 return "\n\n".join(lines)

def build_prompt() -> ChatPromptTemplate:

```

```

Prompt that takes a profile block, a sources block, and the question.
return ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You are a trading assistant preparing a concise risk summary for "
 "a client's equity portfolio. Use only the provided profile and "
 "sources. If information is missing, say so explicitly."
),
),
 (
 "human",
 (
 "Client profile:\n{profile_block}\n\n"
 "Current portfolio:\n{portfolio_block}\n\n"
 "Relevant research and policies:\n{sources_block}\n\n"
 "Question:\n{question}\n\n"
 "Write a short risk summary (3-5 sentences) that:\n"
 "- highlights key risks for the current positions,\n"
 "- relates them to the client's stated risk tolerance and region,\n"
 "- does not invent facts not supported by the sources."
),
),
],
)
)

def generate_risk_summary() -> None:
 # End-to-end context assembly and model call.
 docs = build_research_corpus()
 vectorstore = build_vectorstore(docs)

 portfolio = build_sample_portfolio()
 profile = build_sample_profile()

 # Derived state used for filtering
 user_instruments = [p["symbol"] for p in portfolio["positions"]]
 user_region = profile["region"]
 cutoff_date = 20241201

 question = "What are the main portfolio risks I should be aware of right now?"

 # 1) Selection: metadata-filtered retrieval
 filtered_docs = retrieve_filtered_docs(
 vectorstore=vectorstore,
 question=question,
 instruments=user_instruments,
 region=user_region,
 min_date=cutoff_date
)

 # 2) Packaging: profile + portfolio + sources
 profile_block = format_profile_summary(profile)
 portfolio_block = format_portfolio_snapshot(portfolio)
 sources_block = format_sources_block(filtered_docs)

 prompt = build_prompt()
 messages = prompt.format_messages(
 profile_block=profile_block,
 portfolio_block=portfolio_block,
 sources_block=sources_block,
 question=question
)

 llm = ChatOpenAI(model="gpt-4.1-mini", temperature=0.0)
 response = llm.invoke(messages)

 print("== Risk summary ==")
 print(response.content)

if __name__ == "__main__":
 generate_risk_summary()

```

ch\_06\scr\metadata\_context\_selection.py

First, it builds a tiny research corpus with rich metadata. `build_research_corpus` creates four Document objects: two research notes (AAPL, TSLA) and two regional risk policies (EU, US). Each document has `page_content` with the actual text and a `metadata` dict containing `title`, `instrument`, `region`, `doc_type`, and a `published_at` date in YYYYMMDD form.

That metadata is the backbone of later filtering: you can distinguish per-instrument research from generic “ALL” policies, and you can filter by region and recency.

`build_vectorstore` then turns that corpus into a Chroma vector store using `OpenAIEmbeddings("text-embedding-3-small")`. This sets up a semantic search index over the documents so later you can query “What are the main portfolio risks?” and still retrieve notes about “volatility”, “macro sensitivity”, etc., even if those words are not in the question.

The next block defines structured state for the client: `build_sample_portfolio` returns a simple portfolio with AAPL and TSLA positions; `build_sample_profile` returns a minimal risk profile (medium tolerance, USD, US region). `format_portfolio_snapshot` and `format_profile_summary` convert these dicts into concise text blocks. The portfolio becomes a short list like “- AAPL: 120 shares @ 175.00”, and the profile becomes one or two sentences describing region, base currency, and risk tolerance. This is context engineering applied to state: you never send raw JSON or a full database row; you send a small, human-readable snapshot.

The retrieval function, `retrieve_filtered_docs`, is where metadata-aware selection happens. It builds a Mongo-style `filter_dict` that encodes three conditions:

- Instrument is either one of the user’s current symbols (AAPL, TSLA) or the generic “ALL” (for policies)
- Region matches the client’s region
- `published_at` is on or after a numeric cutoff date

Then it calls `vectorstore.similarity_search(question, k=4, filter=filter_dict)`. This combines semantic similarity to the question with strict metadata filters. The result is a small set of documents that are both topically relevant and constrained by portfolio instruments, region, and recency. This is salience via metadata, not just via embeddings.

`format_sources_block` then packages those retrieved documents into a compact “sources” section. It assigns each document an index [1], [2], pulls key metadata fields (title, instrument, region, `published_at`), and builds a short snippet by flattening and truncating the `page_content`. The result is a numbered, auditable block that the model can cite and that you can inspect later to see exactly what it saw.

`build_prompt` defines the actual prompt structure. It uses a system message to fix the role (“trading assistant preparing a concise risk summary”) and instructs the model to use only the provided profile and sources, and to admit when information is missing. The human message then lays out three blocks—client profile, current portfolio, relevant research and policies—followed by the user’s question. It also spells out the output format constraints: 3–5 sentences, highlight key risks, relate them to risk tolerance and region, do not invent facts. This is the “packaging” step: everything selected earlier (profile, portfolio, retrieved docs) is arranged into a stable layout the model can learn to interpret.

Finally, `generate_risk_summary` orchestrates the whole flow. It:

- Builds the research corpus and vector store
- Builds the sample portfolio and profile
- Derives `user_instruments`, `user_region`, and a `cutoff_date` from state
- Defines the high-level question
- Calls `retrieve_filtered_docs` with the question and state-derived filters

- Formats `profile_block`, `portfolio_block`, and `sources_block`
- Fills the prompt with these blocks and the question
- Invokes a `ChatOpenAI("gpt-4.1-mini", temperature=0.0)` model
- Prints the resulting risk summary

### 6.2.7 Worked pipelines

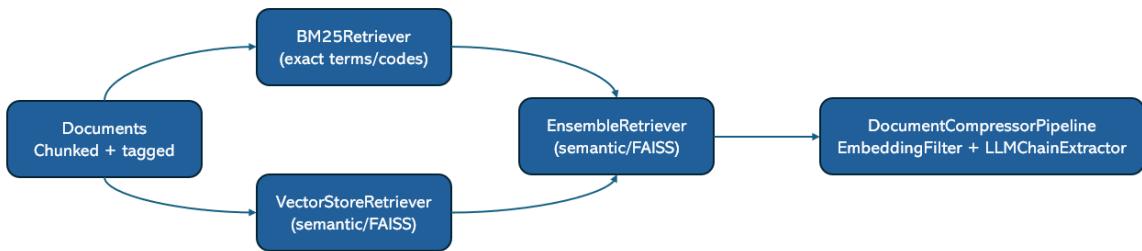
#### 6.2.7.1 Support-policy assistant pipeline

This use case is a support-policy assistant that must answer customer and agent questions by relying on an internal policy knowledge base (manuals, FAQs, runbooks). The same intent arrives in very different forms: some requests are precise and code-like (policy IDs, clause numbers, product codes, specific time windows), while others are informal paraphrases that use everyday language and omit key details.

The assistant is expected to produce answers that are consistent across wording variations, aligned with the correct policy version and locale, and suitable for customer-facing communication. It also has to support internal audit needs: when a decision depends on a specific rule, the organisation must be able to identify exactly which policy text justified it. The volume of questions is steady and operationally time-sensitive, so the system must behave predictably under load and avoid large, variable prompts that cause cost and latency spikes.

#### The Pipeline

- Index build: Split policy manuals, FAQs, and internal runbooks into chunks that preserve section boundaries and keep stable metadata (`policy_id`, `product_line`, `locale`, `effective_date`, `confidentiality_tag`). Store the chunks in two backends: a lexical index for exact-term search and a vector store for semantic search.



- Query-time selection: Start with a hybrid first pass. Run `BM25Retriever` to catch exact identifiers ("POL-217", "refund window 14 days") and run a vector-store retriever (FAISS or equivalent) to catch paraphrases ("can I return after two weeks?"). Merge the candidate sets using a merger or an ensemble strategy so both signals contribute, then apply an MMR-style selection over the semantic side to reduce near-duplicate chunks when multiple sections say the same thing.
- Refinement and token control: Wrap the hybrid retriever in a `ContextualCompressionRetriever` and apply a cheap-to-expensive cascade: first drop low-similarity candidates with an embeddings filter, then run an extractor compressor only on the remaining few to pull the exact sentences that answer the question (and discard the rest of the chunk). This keeps the final context compact while preserving provenance via metadata.

This pipeline is robust to both “code-like” queries and conversational queries, and it tends to produce compact, quotable evidence. The main risk is recall loss from over-aggressive

filtering; tune thresholds against real support questions and ensure metadata is consistent enough to support traceable sources.

```

0) Prepare the retrieval corpus: policy_chunks

`policy_chunks` is the input corpus for every retriever in this example.
It is a list of LangChain Document objects, where:
- Document.page_content holds the chunk text that can be retrieved.
- Document.metadata holds identifiers you can trace back to the source
(policy_id, section, URL, revision, etc.).
#
In a real system you build this by:
1) loading the policy source documents (PDF/HTML/KB),
2) splitting them into chunks with a text splitter,
3) wrapping each chunk as Document(page_content=..., metadata=...).
#
For the rest of the pipeline, treat `policy_chunks` as:
List[Document]
raw_policy_docs = [
 Document(
 page_content = (
 "POL-217: Customers can cancel a new broadband contract within 14 days of "
 "activation with no penalty. After 14 days, an early termination fee "
 "applies."
),
 Metadata = {"policy_id": "POL-217", "source": "kb", "title": "Broadband cancellation"}
)
]

splitter = RecursiveCharacterTextSplitter(chunk_size = 400, chunk_overlap = 80)
policy_chunks = splitter.split_documents(raw_policy_docs)

1) Lexical retrieval (BM25): catch exact terms and "code-like" queries

BM25 is a classic keyword-based ranking function. It is strong when users
include exact surface forms that also exist in the corpus:
- policy codes (e.g., "POL-217")
- clause names ("cooling-off period")
- exact phrases ("refund window 14 days")
It does NOT understand meaning or paraphrase well; it mostly cares about
word overlap and term frequency signals. That's why we pair it with a
semantic retriever next.
bm25 = BM25Retriever.from_documents(policy_chunks)

`k` controls how many documents the retriever returns for a query.
Keeping this small limits noise and keeps the later stages cheaper.
bm25.k = 6

2) Semantic retrieval (FAISS + embeddings): handle paraphrases and synonyms

Embeddings convert text into dense numeric vectors such that "similar meaning"
tends to map to "nearby vectors". This is what lets us retrieve policy
text even when the user query does NOT share the same vocabulary:
- "can I return after two weeks?" ~ "refund window 14 days"
`text-embedding-3-small` is an OpenAI embedding model; you can swap to another
provider/model as long as it returns vectors compatible with your vector store.
embeddings = OpenAIEmbeddings(model = "text-embedding-3-small")

FAISS is an in-memory vector index. It is fast and simple for demos and
local deployments. In production you might use a persistent store (Chroma,
pgvector, Milvus, Pinecone, etc.) depending on your constraints.
#
`FAISS.from_documents(...)` embeds each Document.page_content and stores the
resulting vectors in the FAISS index, while retaining the original Document
objects (including metadata) for retrieval results.
vs = FAISS.from_documents(policy_chunks, embeddings)

Choose ONE of the following retrievers depending on your goal.
All three share the same interface: .invoke(query) -> List[Document].
#
Option A - plain similarity (default):
Returns the top-k nearest neighbours by vector similarity.
Use when you want the most semantically similar chunks and duplicates are acceptable.
semantic_similarity = vs.as_retriever(
 search_type = "similarity",

```

```

 search_kwargs = {"k": 10}
)

 # Option B - MMR (Maximum Marginal Relevance):
 # Returns k documents chosen to balance relevance and diversity.
 # Use when your corpus contains many near-duplicate chunks and you want broader
 # coverage in a fixed context budget.
 semantic_mmr = vs.as_retriever(
 search_type = "mmr",
 search_kwargs = {"k": 10, "fetch_k": 30, "lambda_mult": 0.5}
)

 # Option C - similarity with score threshold:
 # Returns only documents whose similarity exceeds a minimum threshold.
 # Use when you prefer "no context" over low-confidence context (e.g., safety /
 # policy answers), and you want the pipeline to fall back to escalation rather
 # than hallucinate.
 # NOTE: score scales differ by vector store and embedding model; tune
 # score_threshold empirically.
 semantic_threshold = vs.as_retriever(
 search_type = "similarity_score_threshold",
 search_kwargs = {"k": 10, "score_threshold": 0.2}
)

Pick the semantic retriever you want to use in the pipeline.
Default here remains MMR, as it is often a good fit for policy corpora.
semantic = semantic_mmr

3) Hybrid retrieval (EnsembleRetriever): combine lexical + semantic signals

No single retrieval method is best for all query styles:
- Some users paste policy codes or exact phrases -> lexical wins.
- Others paraphrase or ask conceptually -> semantic wins.
#
EnsembleRetriever runs multiple retrievers and blends their ranked outputs.
This reduces "single retriever bias" and makes the system more robust.
hybrid = EnsembleRetriever(
 retrievers = [bm25, semantic],
 # Weights determine how much each retriever influences the final ranking.
 # Here we slightly favour semantic because most user queries are natural
 # language, but we keep BM25 strong enough to reliably surface exact codes.
 # These weights are not "correct" in the abstract-tune them using your
 # evaluation set (real support questions) and inspect failure cases.
 weights = [0.4, 0.6]
)

4) Contextual compression: reduce token waste after retrieval

First-pass retrieval returns candidate chunks; many will be partially relevant
(or contain extra detail). If you send raw chunks directly into the prompt,
you burn tokens and may dilute the key evidence.
#
Contextual compression is a second-stage selector:
- it can drop weak candidates,
- and/or trim candidates to only the parts that matter for this query.
#
We build a "cheap-to-expensive" compressor pipeline:
(a) EmbeddingsFilter: fast similarity thresholding to keep only strong candidates.
(b) LLMChainExtractor: more expensive, but can extract just the sentences/spans
that directly answer the question (fine-grained compression).
#
Using temperature=0 makes the extractor low variance and easier to test.
llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

compressor = DocumentCompressorPipeline(
 transformers = [
 # Step 1: keep only the top-k most similar docs (cheap, no generation).
 # This prevents the extractor from spending tokens on obviously
 # irrelevant chunks.
 EmbeddingsFilter(embeddings = embeddings, k = 8),
 # Step 2: extract query-relevant spans from the remaining docs
 # (expensive). This tends to produce compact, "quotable" evidence that is
 # cheaper to include in the final prompt and easier to audit.
 LLMChainExtractor.from_llm(llm)
]
)

```

### 6.2.7.2 E-commerce assistant pipeline

In this example, you build a product Q&A assistant that answers questions like “Will this case fit an iPhone 15 Pro?” or “What’s the best waterproof hiking jacket under 150€?” using a catalog index rather than guessing from the model’s general knowledge. The core design goal is simple: retrieve fewer, better documents. Instead of searching the entire corpus on every query, the pipeline first uses metadata to narrow the search space to the right slice of the catalog (brand, category, region, model year, price band, availability), then uses semantic retrieval to pick the most relevant chunks inside that slice. This keeps answers grounded in the correct products and policies, reduces hallucinations, and makes the assistant’s behaviour stable as your catalog grows.

#### The Pipeline

- Index build: Store product catalog documents (descriptions, specs, compatibility tables, warranty policies, and top reviews) as Documents with strict metadata (sku, brand, category, price\_band, region, language, model\_year, discontinued\_flag). Chunk the long free-text sources (manuals, reviews) and keep structured sources (spec tables) either as dedicated chunks or as separate Documents so they remain retrievable.



- Query-time selection: Use SelfQueryRetriever-style behaviour as the first decision point: translate the user question into (a) semantic search terms and (b) metadata filters. For example, “Does this fit iPhone 15 Pro?” should produce strong filters on device model and region, while “Best waterproof hiking jacket under 150€” should constrain category and price\_band and then use semantic retrieval for performance claims and fit. This reduces irrelevant matches early and keeps the candidate pool clean.
- Compression and packaging: Apply contextual compression only when necessary. For pure catalog Q&A, a reranker or an embeddings filter is often enough. For verbose sources (reviews, manuals), an extractor compressor can trim to the 2–4 sentences that support the answer.

This pipeline is good when your metadata is strong and users implicitly describe constraints. The failure mode is brittle filtering: if metadata is missing or inconsistent, the retriever may over-filter and miss the right SKU. The fix is operational, not prompt-level: enforce metadata quality and test filter generation on real queries.

```

1) Build the catalog vector store (Chroma): the "semantic index" for products

This function takes pre-chunked product Documents and builds a vector store.
A vector store is an index over embeddings (numeric vectors) so we can later
retrieve semantically similar chunks for a user query.
#
Why Chroma here (vs FAISS)?
- For e-commerce, we often need metadata filtering (brand/category/price/etc.).
- Chroma supports persistent storage and metadata filtering in typical setups.
(FAISS is great for fast similarity search, but metadata filtering support
depends on the wrapper and is not the main reason people choose it.)
def build_catalog_vectorstore(product_chunks: List[Document]) -> Chroma:
 # Embeddings convert text into vectors such that "similar meaning" tends to
 # map to "nearby vectors".
 # All chunk text that you want to retrieve later should live in
 # Document.page_content. Metadata lives in Document.metadata and is used
 # for filtering and traceability.
 embeddings = OpenAIEmbeddings(model = "text-embedding-3-small")

```

```

IMPORTANT: for SelfQueryRetriever to work well later, every Document
should have consistent, reliable metadata fields.
Example metadata keys you might store:
- brand: "Apple"
- category: "phone_case"
- price_eur: 39.90
- in_stock: True
- region: "EU"
#
Chroma stores:
- embeddings for semantic search
- original Documents (text + metadata) for returned results
return Chroma.from_documents(
 documents = product_chunks,
 embedding = embeddings,
 collection_name = "catalog_chunks"
)

2) Build a SelfQueryRetriever: turn natural language into filters + search

SelfQueryRetriever adds a "query understanding" step:
- It uses an LLM to interpret the user question.
- It decides which metadata filters to apply (e.g., category="jacket",
price_eur <= 150, in_stock=true).
- It also produces a search query for semantic retrieval in the vector store.
The result is fewer irrelevant candidates and more stable answers, because
retrieval is scoped before it even begins.
def build_ecommerce_self_query_retriever(vectorstore: Chroma):
 # Temperature=0 makes filter extraction low variance and easier to test.
 llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

 # AttributeInfo describes the metadata schema to the LLM.
 # ONLY list fields that are:
 # - actually present on your Documents,
 # - consistently populated,
 # - and have predictable types.
 # If you list fields that are missing or noisy, the LLM may invent filters
 # and you will "over-filter" (miss the right product) or "mis-filter"
 # (retrieve the wrong slice of the catalog).
 attribute_info = [
 AttributeInfo(name = "brand", description = "Product brand", type = "string"),
 AttributeInfo(name = "category", description = "Product category",
 type = "string"),
 AttributeInfo(name = "price_eur", description = "Unit price in euros",
 type = "number"),
 AttributeInfo(name = "in_stock", description = "Whether the item is currently in
stock", type = "boolean")
]

 # document_contents is a short description of what the Documents contain.
 # This helps the LLM generate a structured query that matches your corpus.
 # search_kwargs={"k": 8} limits the number of retrieved chunks returned
 # after filtering. Keep this small: a few high-signal chunks is better than
 # many noisy ones, and it keeps your prompt within a stable token budget.
 retriever = SelfQueryRetriever.from_llm(
 llm = llm,
 vectorstore = vectorstore,
 document_contents = "Product description, specs, and policy notes",
 metadata_field_info = attribute_info,
 search_kwargs = {"k": 8},
 verbose = False
)

 return retriever

3) Compression stage: keep only what is worth spending tokens on

Retrieval returns candidate Documents, but not all are prompt-worthy:
- reviews are verbose,
- manuals contain unrelated sections,
- even good chunks can be 80% noise for a specific question.
This stage is where you decide:
- keep/drop whole documents (cheap filtering or reranking),
- and optionally extract only the relevant spans (more expensive, more precise).
def build_compression_layer(base_retriever):
 embeddings = OpenAIEmbeddings(model = "text-embedding-3-small")

 # We use a "cheap-to-expensive" compressor pipeline.
 # Step 1: EmbeddingsFilter keeps only the top-k most semantically similar docs.

```

```

Step 2: LLMChainExtractor extracts the sentences/spans that directly support
the answer (useful for long reviews/manuals).
extractor_llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

compressor = DocumentCompressorPipeline(
 transformers=[

 EmbeddingsFilter(embeddings = embeddings, k = 8),
 LLMChainExtractor.from_llm(extractor_llm)
]
)

return ContextualCompressionRetriever(
 base_retriever = base_retriever,
 base_compressor = compressor
)

4) Return ContextualCompressionRetriever

def build_ecommerce_retrieval_pipeline(product_chunks: List[Document]):
 vectorstore = build_catalog_vectorstore(product_chunks)
 base_retriever = build_ecommerce_self_query_retriever(vectorstore)
 final_retriever = build_compression_layer(base_retriever)
 return final_retriever

5) Example usage: retrieving evidence

def example_retrieve(product_chunks: List[Document], user_query: str) -> List[Document]:
 retriever = build_ecommerce_retrieval_pipeline(product_chunks)
 # invoke() returns a list[Document]
 docs = retriever.invoke(user_query)
 return docs

```

### 6.2.8 Governance and evolution of context pipelines

Because context often contains sensitive information and has a strong impact on behaviour, the selection and packaging pipeline must be treated as a governed part of the system, not just an implementation detail.

You need clear rules for which attributes and document types are allowed into prompts, how long different kinds of data may be used, and how context is anonymised or filtered in sensitive domains. You also need observability: logs and evaluations that let you see which documents, history segments, and state summaries were actually sent to the model for a given answer.

Over time, you refine this pipeline. Evaluations that score correctness, completeness, and adherence to constraints reveal where context was missing or noisy. Failures where the model hallucinated despite relevant data existing somewhere in your systems usually indicate a gap in selection or packaging rather than a fundamental model limitation. In practice, many teams find that investing in these pipelines pays for itself: a modest model with a carefully engineered context often outperforms a larger model with an ad-hoc prompt. For that reason, context selection and packaging should be treated as core design decisions, on the same level as data modelling and API design, rather than as one-off prompt tweaks.

## 6.3 SUMMARISATION

Selecting and packaging is about choosing specific items for a single call. Summarisation, salience scoring, and windowing are about keeping the overall amount of information under control when documents and histories become very large. They answer three related questions: how do you compress content while preserving meaning, how do you decide which parts are most important, and how much raw history should you pass to the model at once?

Summarisation turns long inputs into shorter representations that still capture the main facts or structure needed for a task. Salience scoring is the logic that decides which facts deserve to survive that compression. Windowing is how you enforce hard limits on how much uncompressed history is sent to the model, usually by keeping a short recent window and pushing older content into summaries or storage.

Together, these techniques let you work with multi-page documents, multi-day conversations, and long-running workflows without blowing past context limits or drowning the model in irrelevant detail.

### 6.3.1 Single-pass summarisation and prompt templates

The simplest summarisation pattern is a single model call: you give the model an instruction (“Summarise the following text in three bullet points for a non-technical audience”) and then you provide the text to be summarised. This approach is appropriate when the whole input (plus your instruction and any other context you include) fits inside the model’s context window for that request. Context windows are a hard limit. The model can only condition its output on what you send in that call (system instructions, the user’s message, any retained conversation history, and any retrieved context). If the material is larger than a single context window, you need a workflow approach (for example, split-then-summarise and combine).

Even in a one-shot summary, prompt structure matters. In chat-style prompting, instructions are typically represented as separate messages with roles (system, user, assistant). Role separation lets you keep stable behavioural constraints separate from the changing text you are summarising, which makes the prompt easier to reason about and evolve. A useful way to make one-shot summaries more reliable is to be specific about what you want the summary to preserve. Instead of requesting “a summary”, you can request the aspects you intend to reuse later (for example, the main arguments, the decisions already taken, or the risks that were raised). This is still the same single-pass pattern, but you are conditioning the model toward the parts of the input that matter for the next step of your pipeline. (Guidance; the underlying mechanism is still “prompting steers behaviour”.)

When you are building an application, avoid scattering ad-hoc summarisation strings throughout the codebase. Prompts are usually maintained artefacts: they change over time, need consistent wording across features, and benefit from being testable. Prompt templates are the standard way to do this: they separate fixed instructions from runtime variables and reduce the need for manual string concatenation. A template-based summarisation prompt typically contains fixed instructions plus placeholders you fill at runtime (for example, a length target, a target audience, or a specific focus). You then reuse the same template

anywhere your system needs a summary, such as when you summarise older conversation turns to keep the active context short.

Finally, treat summarisation like any other probabilistic model behaviour you depend on: make it explicit and evaluate it. For agentic systems in particular, automated evaluation is valuable beyond the prototype stage because behaviour is not deterministic; you often need to assess outputs (and, where relevant, trajectories) against expectations across a set of test cases.

### 6.3.2 Structured summaries: few-shot and schema-based

When summaries are reused as context, “readable” is not always enough. You often need a predictable shape so downstream code can index, filter, and compose context without guesswork. Structured summaries aim to make the output a stable contract: a fixed set of sections or fields that downstream code can rely on. In practice, you still validate and sometimes repair outputs, because models can omit fields or produce malformed JSON.

Few-shot structured summarisation uses a small set of example pairs to demonstrate the exact format you expect for this kind of summary. In this use case, the goal is primarily to stabilise structure and coverage so summaries remain comparable; the examples can also influence tone and wording, but the key benefit here is format consistency. This is useful when the summary will be stored and later retrieved as a unit, or when you want consistent slots such as “context”, “decision”, “risks”, and “open questions” that you can surface selectively.

Schema-based summarisation makes the contract explicit by asking the model to fill a predefined structure, typically a JSON object with fixed keys (or a fixed set of named sections). In context pipelines, schema outputs are valuable because they enable practical downstream behaviours:

- Indexing and retrieval filters: you can store extracted fields (for example, region, product, incident\_id, key\_entities) as metadata for later retrieval and filtering
- Tool inputs: your tools can consume specific fields directly instead of re-parsing free text
- Context assembly: you can include only the fields needed for the current call (for example, “constraints” and “open\_questions”), keeping the prompt compact and task-aligned

In production, treat the schema as an interface: validate the output (and optionally apply an output-fixing step) before passing it to tools or storage.

### 6.3.3 Reasoning-first summarisation: extract then write (extract-first)

Single-pass summaries can be hard to trust because you only see the final prose. If something important is missing, it is difficult to tell whether the model never noticed it or noticed it but dropped it while compressing. An extract-first approach separates selection from writing, so you can inspect what the system intends to preserve before it generates the final summary.

In the first pass, you do not write a summary. You reason about a compact “coverage set”: the minimum list of items that must survive compression. This is not hidden reasoning; it is a visible checklist. Depending on your domain, the coverage set might include key entities, decisions, constraints, risks, open questions, dates, numeric values, and any “do not lose this” clauses. The output should be easy to inspect, easy to diff, and easy to test.

In the second pass, you write the summary under a clear contract: the final text must cover every item in the coverage set, and it must stay within the requested length budget. This changes what you can debug. If the final summary is wrong, you can usually localize the problem: either the extraction step missed an item, or the writing step failed to cover an extracted item. That separation makes summarisation a maintainable component rather than a magical paragraph generator.

Prefer a structured intermediate format (for example, a JSON list of required points) so it is easy to inspect, diff, and feed into the next step.

### Example: Two-pass summaries

This example makes the first pass a real LLM call that returns a validated Pydantic object (the coverage set). The second pass writes a short summary that must include specific tokens derived from that coverage set, and a tiny verifier fails fast if anything is missing. (This relies on the model's structured-output support returning a validated Pydantic instance when you provide a Pydantic schema.)

```
from __future__ import annotations
from typing import List, Optional, Literal
from pydantic import BaseModel, Field, ValidationError
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY

Pass 1 schema: the "coverage set" we refuse to lose

class CoverageSet(BaseModel):
 times: List[str] = Field(default_factory = list, description = "Times like '09:12' "
 "(24h, leading zero).")
 region: Optional[Literal["EU", "US", "APAC"]] = Field(default = None,
 description = "Region if stated.")
 error_code: Optional[str] = Field(default = None, description = "HTTP error code like "
 "'502'.")
 impact_count: Optional[int] = Field(default = None, description = "Numeric impact "
 "count, e.g. 1200.")
 impact_what: Optional[str] = Field(default = None, description = "Short phrase, e.g. "
 "'failed checkouts'.")
 cause: Optional[str] = Field(default = None, description = "Short phrase, no full "
 "sentences.")
 mitigation: Optional[str] = Field(default = None, description = "Short phrase, no "
 "full sentences.")
 follow_ups: List[str] = Field(default_factory = list, description = "Short action "
 "items (phrases).")

 def required_tokens_from_coverage(self) -> List[str]:
 # Turn structured coverage into a small set of exact tokens that MUST appear verbatim
 # in the summary.
 # Keep this strict so omissions are detectable.
 tokens: List[str] = []
 tokens.extend(self.times)

 if self.region:
 tokens.append(self.region)

 if self.error_code:
 tokens.append(self.error_code)

 if self.impact_count is not None:
 tokens.append(str(self.impact_count))

 if self.impact_what:
 tokens.append(self.impact_what)

 if self.cause:
 tokens.append(self.cause)

 if self.mitigation:
 tokens.append(self.mitigation)

 # Optional: include only first 2 follow-ups as strict tokens (avoid over-
 # constraining).
```

```

tokens.extend(cov.follow_ups[:2])

De-duplicate (preserve order)
seen = set()
out: List[str] = []
for t in tokens:
 if t and t not in seen:
 seen.add(t)
 out.append(t)
return out

def assert_summary_covers(summary: str, required_tokens: List[str]) -> None:
 missing = [tok for tok in required_tokens if tok not in summary]
 if missing:
 raise AssertionError(f"Summary is missing required tokens: {missing}")

Demo

if __name__ == "__main__":
 # Requires OPENAI_API_KEY in your environment.
 llm = ChatOpenAI(model = "gpt-4.1", temperature = 0)

 # Pass 1 will return a validated Pydantic object.
 extractor = llm.with_structured_output(CoverageSet)

 source_text = (
 "At 09:12 CET, we saw a spike in 502s on the checkout API (EU region only). "
 "Error rate peaked at ~18% between 09:18-09:31. "
 "We rolled back the gateway config change deployed at 09:05 and errors dropped "
 "by 09:36. "
 "Preliminary cause: misconfigured header rewrite rule affecting requests "
 "with X-Client-Version < 4.2. "
 "Mitigation applied: feature flag gw_header_rewrite disabled in EU only. "
 "Follow-up needed: confirm whether any orders were dropped and reconcile "
 "payment provider logs; "
 "also add a regression test for the header rewrite rule. "
 "Customer impact: approximately 1,200 failed checkouts."
)

 # -----
 # Pass 1: extract coverage set (LLM -> Pydantic)
 # -----
 extraction_messages = [
 ("system",
 "Extract an incident coverage set. "
 "Return only the schema fields. "
 "Make cause and mitigation SHORT phrases (not sentences). "
 "Make follow_ups a list of short phrases. "
 "Use times exactly like '09:12' (24h). "
 "If a field is not present, leave it null/empty."),
 ("user", source_text)
]

 try:
 coverage: CoverageSet = extractor.invoke(extraction_messages)
 except ValidationError as e:
 raise RuntimeError(f"Structured extraction failed validation: {e}") from e

 print("PASS 1 CoverageSet:")
 print(coverage.model_dump())
 print()

 required_tokens = required_tokens_from_coverage(coverage)

 # -----
 # Pass 2: write summary under a contract (must include tokens)
 # -----
 summary_messages = [
 ("system",
 "Write a concise incident handover summary in <= 120 words. "
 "Hard requirement: include every REQUIRED TOKEN verbatim somewhere "
 "in the summary. "
 "Do not add new numbers. Do not rename tokens."),
 ("user",
 "SOURCE:\n"
 f"{source_text}\n\n"
 "REQUIRED TOKENS (must appear verbatim):\n"
 + "\n".join(f"- {t}" for t in required_tokens))
]

 summary_msg = llm.invoke(summary_messages)

```

```

summary = summary_msg.content
print("PASS 2 Summary:")
print(summary)
print()

Contract check

assert_summary_covers(summary, required_tokens)
print("OK: summary includes every required token from the extracted coverage set.")

```

ch\_06\scr\two\_pass\_summary.py

This script implements a disciplined two-pass summarization pipeline: first it extracts a “coverage set” (what must survive compression), then it writes a narrative summary that is contractually forced to include those extracted facts.

Pass 1 uses structured extraction into a validated `CoverageSet`. The system prompt is doing the real architectural work: it narrows the extractor’s output into inspectable fields (times, region, error\_code, impact, cause, mitigation, follow-ups) and explicitly constrains phrasing (short phrases, exact time format). The goal is not elegance; it’s auditability. Because the output is a Pydantic object, missing or malformed fields cause a `ValidationError`, turning “model drift” into a deterministic failure you can handle.

The `required_tokens_from_coverage()` function then translates the structured object into a strict, minimal list of verbatim tokens. This is intentionally blunt: it converts numbers to strings, limits follow-ups to avoid over-constraining the writer, and de-duplicates while preserving order so checks are stable across runs.

Pass 2 writes the summary with an explicit contract: include every required token verbatim, stay within a word budget, and don’t invent numbers or rename tokens. Finally, `assert_summary_covers()` enforces the contract post-generation, making omissions detectable and debuggable: extraction missed something, or the writer dropped it.

#### 6.3.4 Hierarchical summarisation and map–reduce pipelines

Single-pass summarisation breaks down when the input no longer fits into the context window. Long documents, transcripts, or collections of related notes need a different approach. LangChain’s map–reduce summarisation pattern supports a map–reduce approach for summarizing long documents; with recursive collapsing, this can form a multi-level hierarchy of summaries.

The basic recipe is to split the input into chunks that fit into the context window; apply a summarisation chain to each chunk in parallel (the map step); optionally “collapse” intermediate outputs when there are too many; and finally run a reduce step that takes these partial summaries and produces a single, global result. The reduce step can output a plain summary, a domain-specific brief (for example, “assumptions, risks, implications”), or a question-driven answer that includes references to the contributing excerpts (for example, by using a ‘with sources’ style reduce prompt).

For extremely long inputs, the collapse step can itself be recursive: you summarise groups of chunk summaries into higher-level summaries, and only then perform the final reduction. Conceptually, you are building a tree where leaves are raw chunks and internal nodes are

summaries. Each model call sees a manageable local window, while the overall structure lets you cover hundreds of pages or hours of audio.

In many applications, summarisation is part of the context-management pipeline: you reuse the resulting summary as compact context for subsequent steps, instead of treating it as an afterthought.

### Example: Hierarchical map-reduce summarisation for a long document

This example shows how to summarise a long document that does not fit comfortably into a single model call. It splits the text into chunks, runs the same summarisation prompt on each chunk (map), and then summarises those chunk summaries into one final, higher-level summary (reduce). The result is a compact representation you can reuse as context in later steps.

```
from .config import OPENAI_API_KEY
from typing import List
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_text_splitters import RecursiveCharacterTextSplitter

def build_model() -> ChatOpenAI:
 # Single chat model used for both map and reduce steps.
 if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

 return ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.2
)

def build_text_splitter() -> RecursiveCharacterTextSplitter:
 # Split long documents into overlapping chunks that fit comfortably
 # within the model's context window.
 return RecursiveCharacterTextSplitter(
 chunk_size = 1500,
 chunk_overlap = 200,
 separators = ["\n\n", "\n", ". "]
)

def build_map_prompt() -> ChatPromptTemplate:
 # Map step: summarise a single chunk with a stable, targeted template.
 return ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You are helping to summarise a long technical document. "
 "Produce a concise summary of the given chunk, focusing on:\n"
 "- main claims or points\n"
 "- key evidence or arguments\n"
 "- important risks or implications\n"
 "Do NOT try to reference other chunks. Only use the text provided."
)
),
 ("human", "{chunk_text}")
]
)

def build_reduce_prompt() -> ChatPromptTemplate:
 # Reduce step: combine multiple chunk summaries into a single global summary.
 return ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You will receive several partial summaries of one long document. "
 "Combine them into a single, coherent summary that a busy engineer "
 "could read in under two minutes. Focus on:\n"
 "- overall problem and context\n"
 "- main findings or proposals\n"
 "- key risks, trade-offs, and open questions\n"
 "Avoid repetition and do not invent details that are not implied "
 "by the partial summaries."
)
)
]
)
```

```

)
),
 (
 "human",
 (
 "Here are the partial summaries, in order:\n\n"
 "{chunk_summaries}\n\n"
 "Write the final overall summary."
)
)
)
]

def split_into_chunks(long_text: str) -> List[str]:
 splitter = build_text_splitter()
 return [chunk.page_content for chunk in splitter.create_documents([long_text])]

def map_summarise_chunks(
 model: ChatOpenAI,
 map_prompt: ChatPromptTemplate,
 chunks: List[str],
) -> List[str]:
 # Map step: apply the same summarisation prompt to each chunk.
 # This is the part you would typically parallelise in production.
 map_chain = map_prompt | model

 chunk_summaries: List[str] = []
 for i, chunk_text in enumerate(chunks, start = 1):
 result = map_chain.invoke({"chunk_text": chunk_text})
 summary_text = result.content if hasattr(result, "content") else str(result)
 print(f"\n--- Map summary for chunk {i} ---")
 print(summary_text)
 chunk_summaries.append(summary_text)

 return chunk_summaries

def reduce_summaries(
 model: ChatOpenAI,
 reduce_prompt: ChatPromptTemplate,
 chunk_summaries: List[str],
) -> str:
 # Reduce step: summarise the list of chunk summaries into one global summary.
 reduce_chain = reduce_prompt | model

 # Join partial summaries into one text for the reduce step.
 joined_summaries = "\n\n---\n\n".join(chunk_summaries)
 result = reduce_chain.invoke({"chunk_summaries": joined_summaries})
 final_summary = result.content if hasattr(result, "content") else str(result)

 return final_summary

def map_reduce_summarise(long_text: str) -> str:
 # End-to-end map-reduce summarisation for a single long document.
 model = build_model()
 map_prompt = build_map_prompt()
 reduce_prompt = build_reduce_prompt()

 # 1) Split the long text into manageable chunks.
 chunks = split_into_chunks(long_text)
 print(f"Document split into {len(chunks)} chunks.")

 # 2) Map: summarise each chunk independently.
 chunk_summaries = map_summarise_chunks(model, map_prompt, chunks)

 # Optional: here you could insert a 'collapse' stage to recursively
 # summarise the chunk_summaries if there are too many of them.

 # 3) Reduce: summarise the partial summaries into a global summary.
 final_summary = reduce_summaries(model, reduce_prompt, chunk_summaries)

 return final_summary

if __name__ == "__main__":
 # For the book, you can replace this with any long article, report, or transcript.
 long_text = (
 "Lorem ipsum style placeholder for a long technical document. "
 "In your real code, this would be the content of a multi-page report, "
 "a long PDF extracted to text, or a transcript. "
 "* 50"
)

 summary = map_reduce_summarise(long_text)
 print("\n--- Final map-reduce summary ---")

```

```
print(summary)

ch_06\scr\map_reduce_summarization.py
```

```
(.venv) C:\Users\yourname\llm-app> python -m src.map_reduce_summarisation
Document split into 7 chunks.

==== Map summary for chunk 1 ====
The provided text is a repetitive placeholder ("Lorem ipsum") commonly used to simulate the appearance of a long technical document. It does not contain substantive content, claims, evidence, or risks. Therefore, no meaningful summary of main points, arguments, or implications can be derived from this text.

==== Map summary for chunk 2 ====
The provided text is a placeholder commonly used to represent the content of a long technical document, such as a multi-page report, extracted PDF, or transcript. It does not contain substantive claims, evidence, or risks, serving only as a filler to simulate the presence of detailed technical content.

==== Map summary for chunk 3 ====
The provided text is a placeholder commonly used to simulate the appearance of a long technical document, such as a multi-page report, PDF extraction, or transcript. It does not contain substantive content, claims, evidence, or risks, serving only as filler text for layout purposes.

==== Map summary for chunk 4 ====
The provided text is a placeholder commonly used to simulate the appearance of a long technical document. It does not contain any substantive claims, evidence, or risks. Therefore, no meaningful summary of main points or implications can be derived from this content.

==== Map summary for chunk 5 ====
The provided text is a placeholder commonly used in technical documents and does not contain substantive content. Therefore, there are no main claims, key evidence, or risks to summarize.

==== Map summary for chunk 6 ====
The provided text is a placeholder repeating the phrase that it represents content from a multi-page technical document, such as a report, PDF, or transcript. No substantive claims, evidence, or risks are presented in this excerpt.

==== Map summary for chunk 7 ====
The provided text is a placeholder commonly used to simulate the appearance of a long technical document. It does not contain any substantive claims, evidence, or risks. Therefore, no meaningful summary can be derived from this content.

==== Final map-reduce summary ====
The provided text is a placeholder ("Lorem ipsum") commonly used to simulate the appearance of a long technical document. It contains no substantive content, claims, evidence, or discussion of risks. As such, there are no meaningful findings, proposals, trade-offs, or open questions to summarize.
```

First, the code pins down a single chat model and uses it consistently across the whole workflow. An important design point is not the specific model's name, but the fact that both stages—map and reduce—share the same model configuration. That keeps tone and level of detail stable: chunk summaries and the final combined summary “sound like” they belong to the same summariser, which matters when you later reuse the final result as context.

Next, it defines a deterministic chunking strategy that turns an overlong document into manageable windows. build\_text\_splitter returns a RecursiveCharacterTextSplitter configured with chunk\_size=1500 and chunk\_overlap=200, using a simple separator preference order: paragraph breaks first, then newlines, then sentence boundaries. This choice tries to keep semantically coherent units

together (paragraphs and sentences) while still enforcing an upper bound that fits comfortably inside the model’s context window. The overlap is there to reduce boundary loss: if an important definition or constraint sits at the end of one chunk, it is likely to appear again at the beginning of the next chunk.

The next two functions define the “contracts” for the map phase and the reduce phase. `build_map_prompt` creates a `ChatPromptTemplate` that tells the model to summarise one chunk in isolation and to focus on three categories: main points, key evidence, and risks/implications. It also explicitly forbids cross-chunk assumptions. That instruction is doing real work: without it, the model tends to write summaries that imply global knowledge or smooth over missing context. Here, each map call is meant to be local and mechanically repeatable.

`build_reduce_prompt` then defines the second contract: it receives multiple partial summaries and combines them into a single global summary that can be read quickly. The prompt sets the expected scope (overall problem/context, main findings/proposals, risks/trade-offs/open questions) and explicitly asks to avoid repetition and to avoid inventing details not implied by the partial summaries. This is the core safety property of map–reduce summarisation: the reduce step should compress and organise what the map step already extracted, not “discover” new facts.

`map_summarise_chunks` is the map phase orchestrator. It composes the prompt and model using the LangChain pipe operator (`map_chain = map_prompt | model`), then iterates chunk-by-chunk. For each chunk it calls `map_chain.invoke` with `{"chunk_text": chunk_text}`, extracts `result.content`, and appends it to `chunk_summaries`. Conceptually, this loop is the unit you would parallelise in production (using LamGraph), because each chunk summary is independent once chunking is done.

`reduce_summaries` is the reduce phase orchestrator. It joins the partial summaries with a clear separator (---) to preserve ordering and to make boundaries visible to the model, then calls the reduce prompt + model chain. The key structural idea is that the reduce step does not see the original long document at all. It only sees the compressed outputs of the map step, which keeps the reduce input small and forces it to work as a “combiner” rather than a second reader of the full source.

### 6.3.5 Reasoning-first summarisation

### 6.3.6 Conversation summarisation and mixed memory

The same ideas you use to compress documents apply to conversation history. Long chats quickly exceed context limits and, even when they fit, they can distract the model from the latest user intent. A good conversation strategy is to keep two views of history at once: a compact summary of the long-running story, and a short window of recent turns in full.

At the storage level, you still keep the full log of messages in a database or logging system for analytics, debugging, and long-term personalisation. For the working context, however, you do not replay everything. Instead you maintain:

- a running summary that captures long-range, high-salience facts (what the issue is, what has already been tried, decisions and preferences), and
- a small buffer of the most recent user–assistant turns that preserves exact wording and local nuance.

When the recent buffer grows beyond a size you are comfortable with, you fold part of it into the summary and trim the buffer back down. The summary becomes the “long memory”, the buffer is the “short memory”, and each model call sees both.

An explicit approach, which fits well also with LangGraph, is to treat this as normal application state: a node or helper function reads the current summary and recent messages, decides whether to summarise again, and updates both fields. The rest of the system then uses those fields as just another part of the context it passes to the model.

The important point is that summarisation and windowing policies are not invisible magic; they are part of your design. You choose what the summary should focus on (for example, “problem, steps taken, current status, preferences”), how aggressive trimming should be, and how often you are willing to pay the summarisation cost. The reward is a layered view of history that keeps the model grounded in what matters without burning most of the context window on old small talk.

### Example: Explicit summary + window state for a long-running support ticket

This example shows how to manage chat history with an explicit “summary + window” state instead of relying on a built-in legacy memory classes. The state tracks a running summary and a list of recent messages. After each turn, a small summariser chain decides when to fold older turns into the summary, keeping the recent window within a soft token budget.

```
from __future__ import annotations
from dataclasses import dataclass, field
from typing import List, Any
from .config import OPENAI_API_KEY
from langchain_core.messages import HumanMessage, AIMessage, SystemMessage, BaseMessage
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI

@dataclass
class ConversationState:
 # Mixed memory: a running summary plus a window of recent messages.
 summary: str = ""
 recent_messages: List[BaseMessage] = field(default_factory = list)

 def approx_token_length(messages: List[BaseMessage]) -> int:
 # Very rough token proxy: count words across message contents.
 # Good enough to trigger summarisation decisions in this example.
 return sum(len((m.content or "") .split()) for m in messages)

 def build_summariser_chain(model: ChatOpenAI):
 # Chain that updates the running summary using the latest dialogue chunk.
 # It takes the existing summary and new dialogue and returns an updated summary.
 prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You summarise long-running support conversations for later reuse. "
 "Keep the summary concise (80-120 words) and focused on:\n"
 "- the main problem,\n"
 "- steps already taken,\n"
 "- current status,\n"
 "-- any explicit user preferences or constraints.\n"
 "Do not include greetings or small talk."
)
),
 (
 "human",
 (
 "Existing summary (may be empty):\n"
 "{existing_summary}\n\n"
 "New dialogue turns:\n"
 "{new_dialogue}\n\n"
 "Update the summary so it reflects the whole conversation so far. "
 "Overwrite the old summary; do not append a second one."
)
)
]
)
```

```

)
)
)
return prompt | model

def maybe_update_summary(
 state: ConversationState,
 summariser_chain,
 max_window_tokens: int = 80,
 max_total_tokens: int = 200
) -> None:
 # Decide whether to fold recent messages into the summary.
 # If the recent window is small, do nothing.
 # If recent + summary is too large, summarise and shrink the window.
 # If the recent window is still small, keep it as-is.
 if approx_token_length(state.recent_messages) <= max_window_tokens:
 return

 # Build a plain-text view of the recent dialogue.
 lines: List[str] = []
 for msg in state.recent_messages:
 if isinstance(msg, HumanMessage):
 prefix = "User"
 elif isinstance(msg, AIMessage):
 prefix = "Assistant"
 else:
 prefix = "Other"
 lines.append(f"{prefix}: {msg.content}")
 new_dialogue = "\n".join(lines)

 # Call the summariser to produce an updated running summary.
 updated_summary = summariser_chain.invoke(
 {
 "existing_summary": state.summary,
 "new_dialogue": new_dialogue,
 }
)

 # Store the new summary.
 state.summary = updated_summary.content.strip()

 # Keep only the last few turns in the window to preserve local coherence.
 # Here we keep the last 2 user-assistant exchanges (4 messages) if available.
 if len(state.recent_messages) > 4:
 state.recent_messages = state.recent_messages[-4:]

def build_support_model() -> ChatOpenAI:
 if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your config or .env file.")
 return ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.2)

def build_support_system_message() -> SystemMessage:
 return SystemMessage(
 content = (
 "You are a patient, technically skilled support assistant. "
 "Use the conversation summary to remember the ongoing ticket, "
 "and the recent messages to respond naturally to the latest question. "
 "Ask clarifying questions when needed."
)
)

def build_assistant_reply(model: ChatOpenAI, state: ConversationState, user_text: str) -> AIMessage:
 # Build the prompt for the assistant using:
 # - system instruction,
 # - optional summary,
 # - recent messages,
 # - latest user message.
 messages: List[BaseMessage] = [build_support_system_message()]

 if state.summary:
 messages.append(
 SystemMessage(
 Content = f"Conversation summary so far:\n{state.summary}"
)
)

 # Include previous few messages for local context.
 messages.extend(state.recent_messages)

 # Add the latest user message.
 user_msg = HumanMessage(content = user_text)

```

```

messages.append(user_msg)

Call the model.
ai_msg = model.invoke(messages)

Update state with the new turn.
state.recent_messages.append(user_msg)
state.recent_messages.append(ai_msg)

return ai_msg

def demo_long_running_ticket():
 model = build_support_model()
 summariser_model = build_support_model()
 summariser_chain = build_summariser_chain(summariser_model)

 state = ConversationState()

 user_turns = [
 "Hi, my web app keeps timing out when saving large reports.",
 "Yes, the error appears in the logs as a 504 gateway timeout.",
 "We tried increasing the load balancer timeout, but it still happens.",
 "Now it's mostly happening during peak traffic in the evenings.",
 "Given all this, what would you recommend as the next debugging step?"
]

 for i, text in enumerate(user_turns, start = 1):
 print(f"\n==== USER TURN {i} ====")
 print(text)

 reply = build_assistant_reply(model, state, text)
 print("\nAssistant:")
 print(reply.content)

 # Maybe fold part of the window into the summary.
 maybe_update_summary(state, summariser_chain)

 print("\n[DEBUG] Current summary:")
 print(state.summary or "(no summary yet)")
 print("\n[DEBUG] Recent window size:", len(state.recent_messages))

 print("\n==== FINAL STATE ====")
 print("Summary:")
 print(state.summary)
 print("\nRecent messages:")
 for msg in state.recent_messages:
 role = "User" if isinstance(msg, HumanMessage) else "Assistant"
 print(f"{role}: {msg.content}")

 if __name__ == "__main__":
 demo_long_running_ticket()

```

ch\_06\src\explicit\_summary\_plus\_window\_state.py

```

(.venv) C:\Users\yourname\llm-app> python -m src.
explicit_summary_plus_window_state
==== USER TURN 1 ====
Hi, my web app keeps timing out when saving large reports.

Assistant:
I'm sorry to hear that your web app is timing out when saving large reports.
Could you please provide a bit more detail? For example:

- How large are the reports (approximate size or number of entries)?
- Are you seeing any specific error messages or codes when the timeout occurs?
- How long does it take before the timeout happens?
- What technology stack is your web app using (e.g., backend language, database)?
- Is this a new issue, or has it been happening for a while?

This information will help me assist you better.

[DEBUG] Current summary:
User reports that their web app times out when saving large reports. Assistant
requested additional details including report size, error messages, timeout
duration, technology stack, and whether the issue is new or ongoing to better
diagnose the problem. No further information has been provided yet.

[DEBUG] Recent window size: 2

```

```

==== USER TURN 2 ====
Yes, the error appears in the logs as a 504 gateway timeout.

Assistant:
Thanks for the info about the 504 gateway timeout. That usually means the server
didn't respond in time. To help narrow down the cause, could you also share:

- The approximate size or number of entries in the large reports you're saving?
- How long the save operation takes before timing out?
- What backend technology and web server or proxy you're using (e.g., Node.js
with Nginx, PHP with Apache)?
- Whether this timeout happens consistently with all large reports or only some?
- If there have been any recent changes to the app or server configuration?

This will help identify if the issue is due to server timeout settings, processing
delays, or something else.

[DEBUG] Current summary:
User reports their web app times out with a 504 gateway timeout error when saving
large reports. Assistant requested details including the approximate size or
number of entries in the reports, duration before timeout, backend technology
stack and web server/proxy used, consistency of the timeout across reports, and
any recent changes to the app or server configuration. No further information has
been provided yet.

[DEBUG] Recent window size: 4
Etc...

```

The windowing trigger is `approx_token_length`. It is intentionally approximate—counting words rather than real tokens—but it demonstrates the control point: when the recent window becomes too large to resend verbatim, the system switches to compression. In a production system you would calculate real token counts, but the policy logic is the same.

Summarisation is built as a small LCEL pipeline (`prompt → model`). The summariser prompt is narrowly scoped: it produces a short, structured summary covering the problem, steps taken, current status, and explicit preferences/constraints. Crucially, it overwrites the previous summary rather than appending. That keeps the summary's size stable, which is exactly what you want for a reusable context artifact.

`maybe_update_summary` enforces the policy. While the recent window is under the threshold, it leaves messages untouched. Once it crosses `max_window_tokens`, it converts `recent_messages` into a plain transcript, summarises it together with the existing summary, then trims `recent_messages` down to the last two user–assistant exchanges. That preserves local coherence (what was just said stays verbatim) while older detail is safely folded into the summary.

On each assistant call, the prompt is assembled in a consistent order: system instructions, the running summary (if any), the recent window, and the latest user message. This makes summarisation part of context management: the summary becomes the compact representation that keeps the conversation within the context window over time.

### 6.3.7 Classic conversation memory classes (`buffer`, `window`, `summary`, `entities`, `graph`)

In many codebases you will encounter a set of “classic” memory helpers whose job is simple: capture conversation state over time and expose it back to prompts in a predictable variable.



In newer LangChain guidance, ongoing conversation state is often managed in LangGraph state (with persistence via a checkpointer), while these ‘classic’ memory classes remain available (often as legacy/classic components) and still appear in existing codebases.

Most of the classic LangChain chat memory implementations follow the `BaseMemory/BaseChatMemory` pattern: after each call they save the latest inputs/outputs (`save_context`), and before the next call they expose one or more prompt variables (`load_memory_variables`) under a configurable key (commonly `history`).

The differences are not about intelligence. They are about what representation you keep and how aggressively you control growth.

- `ConversationBufferMemory` is the literal approach: it keeps the entire conversation transcript and returns it for injection into the next prompt. This is the most faithful representation, and it is often the right choice for short interactions where exact wording matters (support tickets, negotiations, or any flow where a single phrase can change meaning). Its failure mode is predictable: the transcript grows until it is too expensive or too large to send. Implementations commonly decide when to ‘fold’ older messages into the summary based on a token-length budget rather than a fixed number of turns.
- `ConversationBufferWindowMemory` is the pragmatic variant: it keeps only the last  $k$  turns (recent exchanges). This gives you a stable upper bound on how much dialogue you replay, and it tends to perform well when the assistant mainly needs local context (the last question, the last answer, the last tool result). Its failure mode is also predictable: it can drop earlier commitments, preferences, or constraints that still matter to the user.
- `ConversationSummaryMemory` trades verbatim history for a rolling synopsis. Instead of replaying every prior message, it maintains a compact summary that is updated over time. This is useful when the long-range story matters, but the exact phrasing does not. The summary becomes a “working narrative” of what is going on: what the user wants, what has already happened, and what remains unresolved. The main risk is silent omission: if the summary process fails to carry forward a detail, it will not be available later unless you also keep a raw buffer somewhere.
- `ConversationSummaryBufferMemory` is the hybrid: it keeps a short recent buffer in full and collapses older turns into a summary. This is the same layered context strategy described above, but automated. In practical terms it gives the model two complementary views: “what just happened” in precise wording, plus “what this long-running interaction is about” in compressed form.
- `ConversationEntityMemory` (often referred to informally as “entity memory”) focuses on extracting and updating a small set of entity-centric facts from the dialogue: acts and attributes about entities mentioned in the dialogue that behave more like a profile than like chat history. The idea is that many conversations contain repetitive “who/what” facts that you do not want to keep rediscovering. Entity memory keeps those facts in a structured form so they can be injected into context consistently, even when the raw dialogue window has moved on. The risk here is extraction drift: if the system incorrectly updates an attribute, that incorrect “profile” can persist and bias later turns.
- `ConversationKGMemory` is the relationship-heavy cousin of entity memory. Instead of storing isolated attributes, it maintains a lightweight knowledge graph view of what has been from the conversation: entities plus relationships between entities (for example, ‘X

works for Y'), expressed as extracted graph facts. This can be useful in domains where relationships matter more than chronology (projects, organisations, systems diagrams, compliance hierarchies). Its failure mode is again drift, but in graph form: if the system invents or mis-links a relationship, later answers can sound coherent while being structurally wrong.

Two practical notes keep these classes from turning into “mystery state”.

First, make the prompt boundary explicit. A memory object does not improve anything unless your prompt template actually renders the variables it provides. When readers see a chain “with memory”, they should also see where that memory appears in the prompt as a named slot (history, chat\_history, entities, profile, or similar) so it is obvious what the model is actually reading.

Second, treat memory choice as a cost-and-risk decision. Buffer maximises fidelity but grows without bound; window caps cost but forgets; summary compresses but can omit; entity/graph memory stabilises facts but can drift. In many real systems the safest baseline is still layered: keep a small verbatim window for local nuance and maintain a separate compact representation (summary and/or entity facts) for the long-range story. You can implement that layering manually as explicit state (as described in the next paragraphs), or you can rely on a helper that does it for you, but the design trade-offs do not go away.

```
from langchain_classic.chains import ConversationChain
from langchain_classic.memory import ConversationBufferWindowMemory
from langchain_community.chat_models.fake import FakeListChatModel

A deterministic chat model: each call returns the next response in the list.
chat_llm = FakeListChatModel(
 responses = [
 "Noted. What's the user's name?",
 "Got it. What city are they in?",
 "Thanks. I'll remember those details for the next step."
]
)

Keep only the last 2 exchanges (user+assistant pairs).
memory = ConversationBufferWindowMemory(k = 2, return_messages = True)
conversation = ConversationChain(llm = chat_llm, memory = memory, verbose = False)

conversation.predict(input = "We are onboarding a new customer.")
conversation.predict(input = "Her name is Ada.")
conversation.predict(input = "She is based in Rome.")

Show what the memory retained (only the most recent window).
With return_messages=True, this is a list of chat messages (HumanMessage/AIMessage).
print("WINDOWED HISTORY:")
for msg in memory.chat_memory.messages:
 print(f"- {msg.type}: {msg.content}")
```

ch\_06\scr\ConversationBufferWindowMemory.py

## 6.4 SALIENCY SCORING AND WINDOWING STRATEGIES

### 6.4.1 Salience-focused summarisation: chain-of-density and similar patterns

Not all summaries are created equal. Two texts with the same length and the same source material can differ wildly in usefulness. One might say “the team discussed several technical options and chose a compromise”, while another names the service, the options, the chosen architecture, and the key trade-offs. Both are “summaries”, but only one carries enough concrete information to be useful as context. A useful engineering lens is to think of summaries along three dimensions:

- Length: how many words or tokens they use
- Focus: which aspects of the source they emphasise (decisions, risks, chronology, entities, and so on)
- Density: how many specific, useful items they carry per unit of text

Many summarisation prompts specify a target length and hint at what to emphasise (for example, ‘focus on decisions’). CoD adds an explicit densification constraint. Chain-of-density (CoD) is about density: it is a way to deliberately increase the number of salient entities in a summary while keeping the length fixed. That makes it particularly interesting for context engineering, where every token in the window is a scarce resource.

### 6.4.2 Chain-of-density

Chain-of-density treats summarisation as a controlled sequence of rewrites rather than a single shot. The pattern is:

1. Generate an initial “entity-sparse” summary of the source text.

2. Repeatedly ask the model to:

- Identify a small set of missing entities that are important but not present in the current summary
- Rewrite the summary so that it includes both the old entities and the new ones, while keeping the length the same

Here, an “entity” is any short, specific item that matters for understanding the text: names of people, organisations, or products; key events or actions; important places; dates and time periods; central concepts or metrics. A “missing entity” is constrained to be relevant to the main story, concise (often five words or fewer), not already in the summary, and supported by the source text rather than invented and (as defined in the original CoD prompt) faithful to the article and allowed to come from anywhere in it. Two constraints make the process interesting:

- Carry-forward constraint (prompt-level): each rewrite must retain the entities/details already present, then add the new missing entities.
- Length constraint: keep summary length fixed (often enforced as a word count in prompts but evaluated/controlled as a token budget in the original paper).

Because the model is not allowed to grow the summary or drop previously chosen entities, each iteration is forced to compress low-value phrases, fuse sentences, and replace vague language with more specific references. In the original evaluation, CoD summaries become progressively more entity-dense while staying length-controlled, and the authors report that they are more abstractive, show more fusion, and exhibit less lead bias than summaries produced with a simple ‘very short summary’ prompt. In practice, you control three main parameters when adapting CoD to your domain:

- The length budget: if the summary is too short, there is no room to add entities; if it is too long, the benefit of extra density fades
- The number of iterations: more steps increase density but also cost and latency
- The domain-specific entity definition: you phrase what counts as an entity in terms that match your documents (for example, legal clauses and case names, risk factors and thresholds, or customer actions and product features)

You do not always need the full multi-step procedure. A simplified variant, useful in many systems, is a single densification step: the model produces an initial summary, lists important entities that are missing, and then writes a denser summary that must include them without growing in length. Structurally this is just “explicitly choose what to keep, then rewrite under a fixed length constraint”, but that alone is often enough to make summaries much more useful as compact context artefacts.

CoD reduces vagueness, but it does not guarantee faithfulness by itself. If the summary will be used as authoritative context, treat the ‘missing entities’ list as extractive claims that can be checked (for example, by citing source spans or running a lightweight verification pass).

#### **6.4.2.1 From prompt pattern to context building block**

In the context-engineering setting, the emphasis shifts from the protocol itself to what you do with the result. A CoD-style summary is not just a nicer paragraph for a human reader; it is a high-signal representation you can store and reuse instead of the raw text. Typical uses include:

- Replacing a long customer review, ticket, or call transcript with a dense summary that names products, symptoms, attempted fixes, and decisions
- Compressing a research note or risk document into a short text that still carries the key instruments, scenarios, thresholds, and recommendations
- Creating internal “profiles” for entities (clients, services, projects) that pack many relevant facts into a small token budget.

In all of these cases, the dense summary is what later pipelines retrieve and feed into prompts. You are trading away some nuance to gain a much better ratio of “useful facts per token” in your context window.

#### **6.4.2.2 CoD and related patterns**

Chain-of-density sits next to other salience-oriented patterns rather than replacing them. Reasoning-first summarisation, for example, has the model list key entities, events, or questions first and then write a summary that must cover them within a given length. Map-reduce summarisation uses hierarchical structure to cope with very long inputs and can happily produce medium-length summaries that are later densified with a small CoD step.

The unifying theme is that you stop treating “summarise this” as a single, opaque operation. Instead, you make the model’s selection of important content explicit—either by asking it to list key items, or by having it iteratively add missing entities under a length constraint—and then you reuse the resulting high-density summaries as first-class inputs to your context pipeline. That extra structure costs you some prompt tokens and a bit of modelling effort, but in return you get summaries that behave like carefully packed information capsules rather than slightly shorter versions of the original text.

#### **Example: Chain-of-density for a long customer review**

The goal is to take one long customer review and produce two summaries:

- An initial, slightly vague summary, and
- A denser summary with more concrete entities (product features, problems, context) included in the same length.

```

from __future__ import annotations
from .config import OPENAI_API_KEY
import json
from typing import Any, Dict
from langchain_openai import ChatOpenAI

def build_cod_prompt(review_text: str) -> str:
 # Build a simple chain-of-density style prompt for a single review.
 # The model is asked to:
 # 1. Write an initial, vague summary.
 # 2. List a few important entities missing from that summary.
 # 3. Rewrite the summary to include those entities, without making it longer.
 return f"""
 You are helping to analyse customer feedback.

 Here is one detailed customer review:
 \\""\\"{review_text}\\""\"

 Follow these steps carefully:
 1) Write an INITIAL_SUMMARY of about 60 words. It should be understandable
 but not very detailed. Use only a few concrete entities.
 2) Read the original review again. Identify 2-4 important MISSING_ENTITIES
 that are not clearly mentioned in the INITIAL_SUMMARY. Each entity must be:
 - specific and short (no more than 5 words),
 - relevant to the main issues or praise in the review,
 - actually present in the review text.
 3) Write a DENSER_SUMMARY of about 60 words that:
 - keeps all information from the INITIAL_SUMMARY,
 - also mentions all MISSING_ENTITIES,
 - removes filler phrases to make space,
 - stays clear and readable on its own.

 Return your answer as valid JSON with exactly these keys:
 - "initial_summary": string
 - "missing_entities": list of strings
 - "denser_summary": string

 Do NOT wrap the JSON in backticks or markdown fences.
 """".strip()

def _strip_code_fences(text: str) -> str:
 # Remove leading/trailing ``` / ````json fences if the model adds them.
 # This handles outputs like:
 # ```json
 # { ... }
 # ```
 stripped = text.strip()
 if not stripped.startswith("```"):
 return stripped

 lines = stripped.splitlines()
 # Drop first line if it's a fence (``` or ````json)
 if lines and lines[0].lstrip().startswith("```"):
 lines = lines[1:]
 # Drop last line if it's a fence
 if lines and lines[-1].lstrip().startswith("```"):
 lines = lines[:-1]
 return "\n".join(lines).strip()

def run_simple_cod(review_text: str, model: ChatOpenAI) -> Dict[str, Any]:
 # Run a simple, single-step chain-of-density on one review.

 # Returns a dict with:
 # - initial_summary
 # - missing_entities
 # - denser_summary
 prompt = build_cod_prompt(review_text)

 # Send the whole instruction as one message
 response = model.invoke(prompt)

 # The model is instructed to return JSON; parse it defensively
 content = _strip_code_fences(response.content or "")

 try:
 data = json.loads(content)

```

```

except json.JSONDecodeError as exc:
 raise ValueError(f"Model did not return valid JSON: {content}") from exc

Minimal shape checks
for key in ("initial_summary", "missing_entities", "denser_summary"):
 if key not in data:
 raise ValueError(f"Missing expected key '{key}' in model output: {data}")

if not isinstance(data["missing_entities"], list):
 raise ValueError("missing_entities must be a list of strings")

return data

if __name__ == "__main__":
 if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file or config module.")

model = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.2
)

long_review = """
I ordered the ACME UltraComfort office chair last month after reading several positive reviews. Assembly was straightforward and the build quality looked solid at first. After two weeks, however, the right armrest started wobbling and making a loud creaking noise every time I moved. I also noticed that the lumbar support is fixed and sits too low for my height, even at the highest seat position. On the positive side, the seat cushion is comfortable and I like the breathable mesh back during long work days. Customer service offered to send a replacement armrest, but shipping will take another two weeks. Overall I am unsure whether to keep the chair or return it.
"""

result = run_simple_cod(long_review, model)

print("\nINITIAL SUMMARY:\n")
print(result["initial_summary"])

print("\nMISSING ENTITIES:\n")
for entity in result["missing_entities"]:
 print(f"- {entity}")

print("\nDENSER SUMMARY:\n")
print(result["denser_summary"])

```

ch\_06\scr\chain\_of\_density.py

```

(.venv) C:\Users\yourname\llm-app> python -m src.chain_of_density

INITIAL SUMMARY:
The customer bought the ACME UltraComfort office chair and found assembly easy with good initial build quality. After two weeks, the right armrest became unstable and noisy. The seat cushion is comfortable, and the mesh back is breathable. Customer service offered a replacement armrest with a two-week shipping delay. The customer is undecided about keeping the chair.

MISSING ENTITIES:
- fixed lumbar support
- lumbar support too low
- highest seat position

DENSER SUMMARY:
The customer bought the ACME UltraComfort office chair and found assembly easy with good initial build quality. After two weeks, the right armrest became unstable and noisy. The fixed lumbar support sits too low even at the highest seat position. The seat cushion is comfortable, and the mesh back is breathable. Customer service offered a replacement armrest with a two-week shipping delay. The customer is undecided about keeping the chair.

```

This script applies a simplified Chain-of-Density (CoD) summarisation pass to a single customer review by forcing the model to iteratively “spend” a fixed word budget on more specific content.

The core move is in the prompt built by `build_cod_prompt()`: it asks for two summaries of the same target length. First, the model produces an `INITIAL_SUMMARY` that is intentionally entity-sparse: clear, but vague enough that important specifics are omitted. Next, it must extract 2–4 `MISSING_ENTITIES` that were present in the original review but absent from the initial summary. The constraints (“short”, “relevant”, “actually present in the review”) are the guardrails that make densification meaningful: they push the model to choose concrete, source-grounded items rather than inventing details.

Finally, the model writes a `DENSER_SUMMARY` that keeps all information from the `INITIAL_SUMMARY` while adding every missing entity, without growing longer. That length pressure is the mechanism: to fit new specifics, the model has to delete filler, compress phrasing, and replace generic statements with precise references.

`run_simple_cod()` operationalizes this as a single “densify once” step, then parses and validates the returned JSON to ensure the three required artefacts are present and structurally usable downstream.

#### 6.4.3 Windowing mechanisms in practice

Windowing is the practical side of “how much history do we send to the model?” and it applies to both chats and documents. In simple cases, a fixed-size window over interactions is enough. Legacy components such as `ConversationBufferWindowMemory` (deprecated in the main langchain package) keep only the last K interactions and ignore anything older. This works when sessions are short-lived and older turns rarely matter.

For finer control, token-budget trimming utilities such as `trim_messages` let you truncate a sequence of messages according to a `max_tokens` budget. With options like `include_system=True` and `start_on="human"`, you can keep a valid chat history shape (system message first if present, then a human message) while dropping older content to fit the budget you set. Token counting is driven by the `token_counter` you provide (often a model’s own `get_num_tokens_from_messages` implementation).

On the document side, chunking is itself a windowing strategy: by splitting large texts into chunks of a target size with overlap, you decide how much local context each retrieval and summarization step can see. Long-document summarization often uses a map-reduce shape with an optional collapse step that can be applied recursively to keep intermediate inputs within sequence-length limits.

In practice, most real systems combine these ideas: trim raw history, summarize older context into compressed form, and use retrieval plus contextual compression so the model sees focused snippets rather than whole documents. The exact parameters vary by use case, but the discipline is the same: treat windowing as an explicit design choice rather than an accidental side effect of running out of tokens.

#### Example: Token-based trimming of chat history with `trim_messages` example

This example shows how to apply token-based windowing to chat history using `trim_messages`. The idea is to keep the prompt within a fixed token budget while enforcing rules like “always keep the system message” and “make sure the conversation starts with a human turn”, instead of just chopping off messages blindly.

```
from __future__ import annotations
from typing import List
from .config import OPENAI_API_KEY
from langchain_core.messages import (
 SystemMessage,
```

```

HumanMessage,
AIMessage,
BaseMessage,
trim_messages,
)
from langchain_openai import ChatOpenAI

def build_model() -> ChatOpenAI:
 if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your config or .env file.")
 # We use the same model both for token counting and for answering.
 return ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.2)

def build_initial_history() -> List[BaseMessage]:
 # Build an artificial long conversation with a system prompt
 # and several user/assistant exchanges.
 system = SystemMessage(
 content = (
 "You are a helpful, concise support assistant for a SaaS analytics dashboard."
 "Answer questions based only on the information in the conversation history."
)
)

 # Simulate several past turns that might push us over the token budget.
 turns: List[BaseMessage] = [
 HumanMessage(
 content = (
 "Hi, I'm trying to understand why my monthly active users metric suddenly"
 "dropped last week. Can you help me investigate?"
)
),
 AIMessage(
 content = (
 "Sure. First, can you confirm which project or workspace you are looking "
 "at, and whether any filters are applied to the dashboard?"
)
),
 HumanMessage(
 content = (
 "The project is 'Acme Retail Web', and I applied a segment for EU users "
 "only. I didn't change any other filters."
)
),
 AIMessage(
 content = (
 "Thanks. Have there been any recent changes to your tracking "
 "implementation, such as SDK upgrades, new consent banners, or changes"
 "to event names?"
)
),
 HumanMessage(
 content = (
 "Yes, we rolled out a new cookie consent banner last Tuesday. Some users"
 "may be opting out of tracking now. Also, our team renamed the"
 "'page_view' event to 'screen_view' on mobile, but I thought the "
 "dashboard handled that."
)
),
 AIMessage(
 content = (
 "That could explain part of the drop: users who decline cookies will no "
 "longer be counted, and renamed events might not be mapped yet. We should"
 "check the tracking diagnostics and event mapping settings."
)
),
 HumanMessage(
 content = (
 "Today I also noticed a spike in 500 errors on the /events API endpoint. "
 "Could that be related to the drop in active users as well?"
)
),
]
 return [system] + turns

def trim_history_for_budget(
 model: ChatOpenAI,
 messages: List[BaseMessage],
 max_tokens: int = 180
) -> List[BaseMessage]:

```

```

Use trim_messages to enforce a token budget while keeping:
- the system message,
- a coherent conversation that starts with a human message,
- as many recent turns as fit.
trimmed = trim_messages(
 messages=messages,
 max_tokens=max_tokens,
 # Let LangChain use the model's tokenizer to count tokens.
 token_counter=model,
 # Keep the last messages (most recent context) within the budget.
 strategy="last",
 # Always keep the system message if possible.
 include_system=True,
 # Ensure the resulting conversation starts with a human message
 # after the system instruction.
 start_on="human",
)
return trimmed

def run_windowed_conversation():
 model = build_model()

 # Start with a long history that might not fit comfortably in one prompt.
 history: List[BaseMessage] = build_initial_history()

 print("== Full history (before trimming) ==")
 for msg in history:
 role = type(msg).__name__.replace("Message", "")
 print(f"{role}: {msg.content}\n")

 # Now the user asks a new question that we want to answer.
 latest_user_question = HumanMessage(
 content=(
 "Given everything we discussed, what are the top two checks I should run "
 "first to confirm whether the drop is caused by tracking changes or API "
 "errors?"
)
)
 history.append(latest_user_question)

 # Apply token-based trimming before calling the model.
 trimmed_history = trim_history_for_budget(model, history, max_tokens=180)

 print("\n== History sent to the model (after trimming) ==")
 for msg in trimmed_history:
 role = type(msg).__name__.replace("Message", "")
 print(f"{role}: {msg.content}\n")

 # Ask the model to answer based on the trimmed window.
 response = model.invoke(trimmed_history)

 print("\n== Assistant reply ==")
 print(response.content)

 if __name__ == "__main__":
 run_windowed_conversation()

```

ch\_06\scr\token\_based\_trimming.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.token_based_trimming
== Full history (before trimming) ==
System: You are a helpful, concise support assistant for a SaaS analytics
dashboard. Answer questions based only on the information in the conversation
history.

Human: Hi, I'm trying to understand why my monthly active users metric suddenly
dropped last week. Can you help me investigate?

AI: Sure. First, can you confirm which project or workspace you are looking at,
and whether any filters are applied to the dashboard?

Human: The project is 'Acme Retail Web', and I applied a segment for EU users
only. I didn't change any other filters.

AI: Thanks. Have there been any recent changes to your tracking implementation,
such as SDK upgrades, new consent banners, or changes to event names?
```

Human: Yes, we rolled out a new cookie consent banner last Tuesday. Some users may be opting out of tracking now. Also, our team renamed the 'page\_view' event to 'screen\_view' on mobile, but I thought the dashboard handled that.

AI: That could explain part of the drop: users who decline cookies will no longer be counted, and renamed events might not be mapped yet. We should check the tracking diagnostics and event mapping settings.

Human: Today I also noticed a spike in 500 errors on the /events API endpoint. Could that be related to the drop in active users as well?

==== History sent to the model (after trimming) ===

System: You are a helpful, concise support assistant for a SaaS analytics dashboard. Answer questions based only on the information in the conversation history.

Human: Today I also noticed a spike in 500 errors on the /events API endpoint. Could that be related to the drop in active users as well?

Human: Given everything we discussed, what are the top two checks I should run first to confirm whether the drop is caused by tracking changes or API errors?

==== Assistant reply ===

To confirm whether the drop in active users is caused by tracking changes or API errors, the top two checks you should run first are:

1. Verify the tracking implementation on your site or app to ensure events are being correctly captured and sent, especially for the /events API endpoint.
2. Analyze the 500 error spike on the /events API endpoint to determine if these errors are preventing event data from being recorded properly.

These checks will help identify if the issue stems from data collection problems or backend API failures.

The function `trim_history_for_budget` takes the full message list (system instruction, multiple past turns, and the latest user question) and returns a shorter list that still reads like a valid chat transcript. The goal is to keep the prompt within a fixed size before calling the model, without relying on a brittle “keep the last K turns” rule.

`max_tokens` sets the total token budget for all messages combined. `token_counter=model` is significant: it makes token counting use the same model-specific tokenizer that will be used for generation, reducing surprises where a prompt “looks short” but exceeds the real limit. `strategy="last"` makes trimming recency-first: if the budget is exceeded, older messages are removed before newer ones, preserving the context most likely to influence the next answer.

`include_system=True` tries to retain the system message even under aggressive trimming, so the assistant’s role and constraints remain stable across calls. `start_on="human"` protects structure after trimming: if dropping earlier content would leave an AI reply without its preceding human message, `trim_messages` drops that orphaned reply, so the transcript begins cleanly with a human turn after the system instruction.

## 6.5 CONTEXT FORMATS FOR DIFFERENT TASKS (Q&A, PLANNING, ANALYSIS)

Previous sections focused on which information to include in context and how to keep it within budget. Context format is the other half of the problem: how you arrange that information into instructions, examples, evidence, and input before it reaches the model. Practically, it helps to treat a prompt as a structured container: instructions plus the current input, optionally augmented with examples and external context (for example retrieved snippets or tool outputs). LangChain supports message-based prompt templates (for example, ChatPromptTemplate with system/user roles), and LangGraph makes workflow data explicit via a state object passed between nodes; together, these patterns make it easier to keep inputs consistently structured. For the purposes of this book, it's useful to group most applications into three common families:

- Retrieval-style question answering over documents or data
- Planning agents that break a goal into steps and call tools
- Analysis and decision support over long texts, logs, or mixed data

The sections below describe clear, task-specific formats for each of these, using three running scenarios: an e-commerce product assistant, a support agent, and a financial trading agent.

### 6.5.1 Question answering and retrieval-style assistants

Retrieval-augmented question answering (RAG) is a common pattern for 'chat with your docs' assistants built with LangChain. A common pipeline retrieves relevant Document objects, formats their contents into a context string, and sends that context alongside the user's question using a role-based prompt template. Many applications also add conversational history when the task needs it. A simple and effective format is:

- System message: role and behaviour ("answer using only the context, cite sources, say when you do not know")
- Context block: (sources) retrieved snippets, numbered and labelled
- Optional history: a compact conversation summary or selected recent turns
- User message: the current question

#### Example 1: e-commerce product assistant (Q&A)

```

SYSTEM:
You are an assistant for an e-commerce store.
Answer using only the information in the CONTEXT message.
If the context does not contain the answer, say: "I cannot answer from the
provided sources."
When you state a fact, cite the source id in square brackets, for example [S2].

CONTEXT (SOURCES):
[S1] Type: product_page | Title: X200 Running Shoe | Updated: 2025-01-10
Text: Sizes: 38-48. Colours: black, blue. Upper: mesh. Returns allowed within
30 days for unworn items with receipt.

[S2] Type: returns_policy | Title: Returns and refunds | Updated: 2024-12-01
Text: Customers may return most items within 30 days of delivery for a full
refund. Shoes must be tried indoors only. ...

HISTORY (optional):
(omitted)

USER:

```

Does the X200 running shoe come in size 48, and can I return it if the size does not fit?

### Example 2: support agent (knowledge base Q&A)

#### SYSTEM:

You are a technical support assistant for ACME Routers.  
Use only the information in the CONTEXT message to answer configuration and troubleshooting questions.  
If the answer is not present, say: "I do not know from the provided sources. Please contact support."  
When you state a fact, cite the source id in square brackets, for example [S1].

#### CONTEXT (SOURCES):

[S1] Type: manual | Model: X100 | Section: Factory reset | Version: 1.3  
Text: To reset the ACME X100 to factory settings without changing firmware, press and hold the RESET button for 10 seconds...

[S2] Type: knowledge\_base | Title: Common PPPoE issues | Updated: 2025-03-05  
Text: If PPPoE authentication fails with error 691, verify username and password with the ISP and ensure VLAN tagging is correct. ...

#### HISTORY (optional):

(omitted)

#### USER:

How do I reset an ACME X100 to factory settings without downgrading the firmware?

### Example 3: financial trading Q&A

#### SYSTEM:

You are a portfolio assistant for an equity trading desk.  
Answer using only the information in the CONTEXT message.  
Do not speculate. Do not use general market knowledge. Do not provide investment advice.  
When you state a fact, cite the source id in square brackets, for example [S2].  
If the context does not contain what is needed to answer, say: "I cannot answer from the provided context."

#### CONTEXT (SOURCES + PORTFOLIO):

##### SOURCES:

[S1] Type: research\_note | Ticker: AAPL | Date: 2025-11-15  
Text: We expect near-term volatility around the upcoming product launch...

[S2] Type: risk\_policy | Region: EU | Date: 2025-01-01

Text: Maximum single-name exposure for EU retail portfolios is 10 % of NAV. ...

##### PORTFOLIO:

Positions: AAPL 8 %, MSFT 6 %, TSLA 3 % of NAV.

Risk profile: Moderate.

#### HISTORY (optional):

(omitted)

#### USER:

Is my current AAPL weight within the single-name limits for an EU retail portfolio?

### 6.5.2 Planning and multi-step workflows

Some assistants do more than answer a single question. They plan, call tools, and refine their answers over several steps. In those cases the prompt should distinguish:

- Instructions: the role, capabilities, and safety rules
- Goal: a stable restatement of what the user wants
- Current state: a short summary of what has happened so far

- Working notes or plan: intermediate reasoning and next steps

Your orchestration layer maintains the state and working notes between calls and decides which parts to pass into each new prompt.

### Example 1: support agent planning a troubleshooting flow

#### INSTRUCTIONS:

You are a support workflow planner for ACME Routers.

Your job is to design safe troubleshooting plans that junior agents and tools can execute.

Never skip mandatory safety checks.

Do not write a customer-facing explanation. Produce only the plan requested.

#### GOAL:

The customer reports that their ACME X100 router has no internet connectivity after a recent firmware update.

#### CURRENT\_STATE:

Customer location: EU.

Router model: X100, firmware 1.3.2.

Steps already done:

Customer performed a basic power cycle, no change.

ping\_test(target=8.8.8.8) -> failure, no response.

check\_isp\_status(account\_id=12345) -> status: up.

#### AVAILABLE\_TOOLS:

ping\_test(target)

check\_isp\_status(account\_id)

reset\_router\_safe(model)

collect\_logs(model)

#### WORKING\_PLAN:

1. Summarise what we know so far (1-3 sentences).

2. List the most likely causes to discriminate between (bullets).

3. Propose a safe, ordered troubleshooting plan that uses AVAILABLE\_TOOLS.

#### REQUIRED\_OUTPUT:

A numbered list of troubleshooting steps.

For each step:

tool: the tool name (or "none" if it's a manual check)

call: the exact arguments to use (if a tool is used)

why: one sentence explaining what the step confirms or rules out

### Example 2: trading agent planning a risk review

#### INSTRUCTIONS:

You are a planning assistant for portfolio risk reviews.

You design analysis plans; other components execute them.

Do not write the final risk review. Produce only the plan requested.

#### GOAL:

Review this client's equity portfolio for concentration and sector risk.

#### CURRENT\_STATE:

Portfolio snapshot: [summary here].

Client risk profile: Moderate.

No analysis steps have been executed yet.

#### AVAILABLE\_TOOLS:

fetch\_quotes(tickers)

compute\_var(portfolio)

fetch\_policy(region)

generate\_report(portfolio, findings)

#### WORKING\_PLAN:

Think through the high-level steps needed to assess:  
single-name concentration,

```

sector concentration,
risk limits/policy constraints (region-appropriate),
using AVAILABLE_TOOLS where relevant.

REQUIRED_OUTPUT:
Return a JSON object with exactly:
{
 "steps": [
 {
 "step": "short title",
 "tool": "tool name or null",
 "inputs": "inputs needed or null",
 "purpose": "one sentence"
 }
],
 "notes": "short comments on assumptions or missing data"
}

```

In both prompts the model sees the goal, the state so far, and the tools it can rely on, and is explicitly asked to produce a plan, not a final explanation. This helps keep planning logic transparent and debuggable.

### 6.5.3 Analysis and decision support

Analysis and decision support prompts usually combine a question, curated facts, and explicit constraints. Useful formats keep these elements separate:

- Question: what needs to be decided or explained
- Facts: observations, metrics, and excerpts from relevant documents
- Constraints and policies: limits, thresholds, and rules
- Required output: the structure of the answer

#### Example 1: trading risk analysis

```

System:
You are a risk analyst assistant.
Use only the information in QUESTION, FACTS, and CONSTRAINTS.
Identify risks clearly and avoid generic investment advice.

QUESTION:
Is this portfolio's sector exposure acceptable for the client's stated risk
profile?

FACTS:
Portfolio: 40% technology, 20% healthcare, 20% industrials, 20% cash.
Recent volatility (1-month): technology: high; healthcare: medium; industrials:
low.
Client risk profile: Moderate.

CONSTRAINTS:
Internal policy: technology exposure must be ≤ 35% for Moderate profiles.
Cash allocation must be ≥ 15%.

REQUIRED_OUTPUT:
Analysis: 3-6 sentences referencing FACTS and CONSTRAINTS.
Conclusion: one sentence stating whether the exposure is acceptable.
Risks_and_open_points: bullet list of concerns or missing information.

```

#### Example 2: support incident post-mortem assistant

```

System:
You help SRE teams analyse incidents.
Base your analysis only on QUESTION, FACTS, and CONSTRAINTS.
Do not invent data; if information is missing, list it under "Recommended
follow-ups".

```

**QUESTION:**

What were the most likely contributing factors to this outage?

**FACTS:**

2025-10-01 08:55 UTC: deployment of version 2.3.1 to checkout service.  
2025-10-01 09:02 UTC: spike in 5xx errors on checkout API (peaked at 7.2% error rate).  
2025-10-01 09:03-09:20 UTC: p95 latency on checkout API ranged 820-1400 ms.  
Logs: repeated timeouts when calling the payment gateway from the checkout service.  
No other services show elevated 5xx in the same window.

**CONSTRAINTS:**

SLA targets for checkout API: p95 latency < 300 ms; error rate < 0.5%.  
Incident analysis policy: treat "deployment within 15 minutes before symptom onset" as a primary hypothesis unless contradicted by FACTS.  
Incident analysis policy: "timeout errors in dependency calls" are evidence of dependency slowness/outage or misconfigured timeouts; do not choose between these without supporting FACTS.

**REQUIRED\_OUTPUT:**

Context: 2-4 sentences summarising timeline and symptoms using FACTS.  
Analysis: short reasoning that explicitly applies CONSTRAINTS to FACTS.  
Likely causes: 2-4 bullet points, each tied to at least one FACT and (where relevant) one CONSTRAINT/policy.  
Recommended follow-ups: bullet list of specific missing data needed to confirm or rule out the likely causes.

An e-commerce analysis assistant can follow the same structure for merchandising or pricing questions (for example, sales and stock metrics as FACTS, discount and margin rules as CONSTRAINTS). The important part is that facts, constraints, and the requested answer shape are clearly separated. This makes it easier to inspect the model's reasoning and to refine upstream retrieval, summarisation, and windowing when answers are wrong or incomplete.

## 6.6 MEASURING AND IMPROVING CONTEXT EFFECTIVENESS

Context engineering only pays off if the extra information you pass to the model actually improves outcomes. Longer prompts, deeper retrieval, and elaborate graphs all consume tokens, add latency, and can increase operational complexity. The goal of this section is to make context something you can measure and iterate on, not something you tune by instinct.

The practical question is simple: given your current context pipeline, how often does the system produce answers that are correct, grounded, and useful for the task, and at what cost? Everything that follows is about turning that question into explicit metrics, evaluation datasets, monitoring signals, and change loops.

### 6.6.1 What “effective context” means

In this book, “context” covers everything the model sees for a given step: the system prompt, retrieved documents, tool results, user and session state, and any higher-level control signals that your workflow passes along. Context is effective when this bundle gives the model all the information it needs to solve the task safely and efficiently, without flooding the context window with irrelevant detail. For retrieval-augmented or tool-driven systems, you can think about effectiveness along four dimensions:

- **Retrieval quality.** Does the system surface the right documents or tool results for a query? Retrieval quality is usually measured with ranking metrics such as precision@k and recall@k, which respectively measure how many of the top-k retrieved items are relevant and how many of all relevant items appear in the top-k list. Mean reciprocal rank (MRR) is often used as well; it averages the reciprocal of the rank of the first relevant item across many queries, so higher values indicate that the first useful document tends to appear near the top. MRR considers only the rank of the first relevant item per query (additional relevant items do not change the score).
- **Contextual relevance.** Beyond ‘did we retrieve the right documents,’ you care whether the passages you actually include are the ones that matter for the user’s need. When you can label candidate passages as relevant/not relevant (or graded relevance), you can evaluate the ranked list with precision@k, recall@k, and NDCG; you can also report an F1 score computed from precision@k and recall@k. In a trading assistant, for example, you want the context to focus on the correct product ISIN and its current risk disclosures, not a similarly named fund or an obsolete factsheet.
- **Faithful generation.** Even with the right context, models can hallucinate or deviate from the underlying documents. TruthfulQA is a QA benchmark designed to elicit ‘imitative falsehoods’ (plausible but incorrect answers that resemble common misconceptions), and results reported on such benchmarks illustrate that models can produce fluent but false statements.
- **Information synthesis.** Many tasks require combining several pieces of evidence into a coherent answer rather than quoting a single passage. For instance, a support assistant might have to weave together one policy document and two product-specific exceptions; an e-commerce assistant may need to merge catalog data, stock levels, and shipping constraints; a trading assistant might have to reconcile product terms with tax guidance.

Automated metrics can approximate this, but judged by domain experts, and sometimes by LLM-as-judge evaluators; when using LLM judges, teams typically validate the judge against human ratings because reliability varies by task, domain, and prompt design.

There is also an implicit context dimension: user identity, preferences, and interaction history. A support assistant that always answers as if the caller were a new customer, ignoring plan details stored in CRM, is using an incomplete context even if retrieval and generation are otherwise correct. Measuring effectiveness therefore means checking that the context covers both explicit evidence (documents, tool outputs) and implicit state (who is asking, what has already been said, what environment the graph is running in).

Context effectiveness is not something you can infer from a handful of successful test prompts. You need systematic evaluation that mixes offline datasets, model-level benchmarks, and online measurements.

|                                                                                   |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |
|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <p><b>precision@k and recall@k</b></p> <p>Precision (P) is defined as the fraction of retrieved items that are relevant: <math>P = \#(\text{relevant items retrieved}) / \#(\text{retrieved items})</math>.</p> <p>Recall (R) is defined as the fraction of relevant items that are retrieved: <math>R = \#(\text{relevant items retrieved}) / \#(\text{relevant items})</math>. [1]</p> <p>With a ranked list, “@k” applies the same definitions after truncating the results to the top k items.</p> <p>Precision@k is <math>(\# \text{ relevant in the top } k) / k</math> (assuming at least k results are returned; otherwise the denominator is the number returned).</p> <p>Recall@k uses the same numerator (relevant items in the top k) but keeps the denominator as the total number of relevant items for the query. [1]</p> <p><b>F1@k</b></p> <p>F1 is the balanced harmonic mean of precision and recall: <math>F1 = 2PR / (P + R)</math>. [1]</p> <p>F1@k is computed by substituting P@k and R@k into the same formula: <math>F1@k = 2 \cdot \text{precision}@k \cdot \text{recall}@k / (\text{precision}@k + \text{recall}@k)</math>. [1]</p> <p><b>NDCG</b></p> <p>Discounted Cumulative Gain (DCG) evaluates ranked results by giving higher credit to relevant items near the top of the list, using a rank-based discount (typically logarithmic). Järvelin and Kekäläinen define a discounted cumulated gain that divides gain by the log of the rank (with special handling to avoid <math>\log(1)=0</math> and to avoid “boosting” early ranks). [2]</p> <p>Normalized DCG (nDCG) scales DCG by the best achievable DCG for the same query (the “ideal” ranking). In the Järvelin–Kekäläinen formulation, normalization is done component-wise by dividing the (D)CG vector by the corresponding ideal (D)CG vector, so that 1.0 represents ideal performance. In common shorthand at a cutoff k, this is written as <math>nDCG@k = DCG@k / IDCG@k</math>, where IDCG@k is DCG@k for the ideal ordering. [2]</p> <p><b>TruthfulQA</b></p> |
|-----------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

TruthfulQA is a benchmark designed to measure whether a language model answers questions truthfully, especially in cases where popular misconceptions could lead a model to produce “imitative falsehoods.” The paper reports 817 questions across 38 categories and describes both truthfulness and informativeness as evaluation dimensions. [3][4]

The authors also release the questions and reference answers (for example, as a CSV in the project repository) to support reproducible evaluation. [5]

### References

- [1] Manning, Raghavan, Schütze, Introduction to Information Retrieval (evaluation chapter; precision/recall and F-measure definitions):  
[<https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf>](<https://nlp.stanford.edu/IR-book/pdf/irbookonlinereading.pdf>)
- [2] Järvelin, Kekäläinen (2002), “Cumulated gain-based evaluation of IR techniques” (DCG discounting and normalization to nDCG):  
[<https://faculty.cc.gatech.edu/~zha/CS8803WST/dcg.pdf>](<https://faculty.cc.gatech.edu/~zha/CS8803WST/dcg.pdf>)
- [3] Lin, Hilton, Evans, “TruthfulQA: Measuring How Models Mimic Human Falsehoods” (arXiv):  
[<https://arxiv.org/abs/2109.07958>](<https://arxiv.org/abs/2109.07958>)
- [4] ACL Anthology entry for TruthfulQA (published version and DOI):  
[<https://aclanthology.org/2022.acl-long.229/>](<https://aclanthology.org/2022.acl-long.229/>)
- [5] TruthfulQA repository (benchmark data and evaluation code):  
[<https://github.com/sylinrl/TruthfulQA>](<https://github.com/sylinrl/TruthfulQA>)

### 6.6.2 Model-level benchmarks

General benchmarks are not about your context pipeline, but they help you choose and configure base models before you start tuning prompts and retrieval. Widely used suites include:

- GSM8K for multi-step grade-school math word problems, which probes basic quantitative reasoning.
- HumanEval for code generation, which measures functional correctness of small programming tasks via unit tests rather than text similarity.
- MMLU and related variants for broad multitask language understanding across 57 subjects.
- Truthfulness benchmarks such as TruthfulQA, which are designed to trigger ‘imitative falsehoods’ (plausible answers reflecting common misconceptions) and measure how often a model produces them.

These benchmarks give you a rough baseline on a model family’s underlying capabilities before you invest in prompts, retrieval, and tooling. They do not replace application-specific

evaluation, but they can reduce the risk that you over-engineer context for a model that is weak at the underlying task.

#### 6.6.3 Common metrics (what to measure)

You cannot improve context by “feeling” your way through prompts. You need measurements that tell you (a) whether the system is achieving the user’s goal, (b) whether it did so for the right reasons (evidence and safety), and (c) what it cost you in latency and money. The trick is to separate primary outcomes from guardrails.

Primary metrics answer: “Did the user get what they needed?” These are the numbers you optimize for, and they usually differ by task (customer support resolution, compliance Q&A accuracy, trade surveillance triage, and so on). User-experience metrics are often the most sensitive online signals, but they can be noisy and must be interpreted together with outcome quality. Guardrail metrics keep you honest: a system that looks “better” because it refuses more, retrieves less, or costs twice as much is not an improvement.

The lists below are a practical starting set. You will not use all of them on every project. Pick a small core per use case, define them precisely (what counts as success, what counts as an escalation, what is “within N turns”), and make sure they are computable from logs or labelled examples—otherwise you will end up arguing about anecdotes instead of changing the system.

- Quality and task outcomes (often primary)
  - Task success rate: % of requests where the goal is achieved (pass/fail, based on a clear rubric or an explicit completion event).
  - Answer correctness: exact-match / normalized match / rubric score per example.
  - Groundedness (RAG): % of answers whose claims are supported by retrieved sources. Track both “fully supported answer rate” and “unsupported claim rate”.
  - “I don’t know” behavior (RAG): (a) appropriate abstention rate (abstains when evidence is missing) and (b) missed-answer rate (abstains despite relevant evidence being available). This requires labelled cases that say whether an answer should be possible from the provided documents.
- User experience (often primary online; sometimes diagnostic offline)
  - User feedback: thumbs-up/down, CSAT, ratings; also track “bad outcome rate” (e.g., 1–2 stars).
  - Rework rate: user repeats the question or signals mismatch (“wrong / not what I meant”) within N turns (define N and rules).
  - Escalation / handoff rate: % of sessions routed to a human or fallback workflow (must be explicitly defined and logged by the application).
- Reliability, safety, performance, cost (usually guardrails)
  - Tool/retrieval error rate: timeouts, parsing failures, empty retrieval, permission denials.
  - Refusal rate: total refusals, plus false-refusal rate on benign prompts (requires a labelled set).
  - Latency: p50/p95 end-to-end; for streaming also time-to-first-token (TTFT).
  - Token usage and cost: prompt/completion/total tokens per request; cost per successful task.

- Context size: tokens injected as context; track how it correlates with latency and cost.

#### 6.6.4 Application-level datasets for context

To measure the effect of context choices, you need datasets that reflect your real queries and the evidence they should use. LangSmith represents evaluation datasets as collections of examples with inputs and outputs, plus optional metadata for any extra fields you need such as IDs of relevant documents or checklists of key facts. A typical row in the dataset for a support assistant might contain:

- The customer question.
- A list of policy document sections that should be consulted.
- A reference answer or bullet checklist of mandatory points.
- Tags describing the product, region, and risk level.

With this structure you can evaluate, in one pass, whether retrieval finds the right sections, whether the context assembled for the model actually contains them, and whether the final answer covers the required points.

For retrieval quality, you compute metrics such as precision@k, recall@k, and MRR based on the rank position of the first relevant document (averaged across queries). For answer quality, you compare outputs against references using exact match, token-overlap metrics such as BLEU or ROUGE where appropriate or embedding-based similarity that is more robust to paraphrasing.

|                                                                                     |                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
|  | <p>BLEU (Bilingual Evaluation Understudy) is a reference-based metric introduced for automatic evaluation of machine translation. It scores a system output by measuring modified n-gram precision (how many of the output's n-grams also appear in one or more human reference translations, with "clipping" to prevent over-counting repeats) and applies a brevity penalty to discourage overly short outputs. BLEU is designed to be more meaningful when averaged over a test corpus; the original paper explicitly notes that sentence-level BLEU can diverge from human judgements. [1]</p> <p>BLEU is not a single fixed number unless you also fix its parameters and preprocessing. Reporting and comparing BLEU across papers can be misleading when tokenisation, normalisation, n-gram order, smoothing, or reference processing differ; Post (2018) documents that BLEU values "vary wildly" with these choices and recommends standardised reporting, including use of SacreBLEU to make scores more comparable. [3] SacreBLEU is a widely used implementation that reports a standardized version string showing key settings (e.g., tokenizer, smoothing, version), which improves reproducibility and comparability. [4] Even with consistent computation, BLEU improvements are not guaranteed to reflect real translation quality improvements in all cases; Callison-Burch et al. (2006) discuss situations where higher BLEU is neither necessary nor sufficient for better translations. [5]</p> <p>ROUGE (Recall-Oriented Understudy for Gisting Evaluation) is a family of reference-based metrics introduced for automatic evaluation of summaries. Lin (2004) defines ROUGE-N as n-gram recall between a candidate</p> |
|-------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|

summary and a set of reference summaries (the denominator comes from the reference side, which makes it recall-oriented). [2] Common variants include ROUGE-1 (unigrams), ROUGE-2 (bigrams), and ROUGE-L, which is based on the longest common subsequence and can be expressed using recall, precision, and an F-measure. [2]

ROUGE is useful for regression-style evaluation of summarisation chains when you have human reference summaries and want a fast, repeatable overlap score, but it is still an overlap metric. It can miss valid paraphrases and it does not directly measure factual consistency; recent work reports that ROUGE correlates better with surface qualities such as fluency/coherence than with consistency. [6]

### References

- [1] Papineni, Roukos, Ward, Zhu (2002). “BLEU: a Method for Automatic Evaluation of Machine Translation.” [<https://aclanthology.org/P02-1040.pdf>] (<https://aclanthology.org/P02-1040.pdf>)
- [2] Lin (2004). “ROUGE: A Package for Automatic Evaluation of Summaries.” [<https://aclanthology.org/W04-1013.pdf>] (<https://aclanthology.org/W04-1013.pdf>)
- [3] Post (2018). “A Call for Clarity in Reporting BLEU Scores.” [<https://aclanthology.org/W18-6319/>] (<https://aclanthology.org/W18-6319/>)
- [4] SACREBLEU (reference implementation). [<https://github.com/mjpost/sacrebleu>] (<https://github.com/mjpost/sacrebleu>)
- [5] Callison-Burch, Osborne, Koehn (2006). “Re-evaluating the Role of Bleu in Machine Translation Research.” [<https://aclanthology.org/E06-1032/>] (<https://aclanthology.org/E06-1032/>)
- [6] Zhang et al. (2024). “ROUGE-SEM: Better evaluation of summarization...” (abstract notes poorer correlation for consistency). [<https://www.sciencedirect.com/science/article/abs/pii/S0957417423018663>] (<https://www.sciencedirect.com/science/article/abs/pii/S0957417423018663>)

#### 6.6.5 Evaluators for context-sensitive behaviour

Metrics tell you whether things are improving; evaluators tell you why. In this section we’ll use evaluators to grade outputs when correctness depends on context (for example groundedness, required facts, or contract compliance). The next sections cover the ready-made evaluators you can run directly in code, and then pairwise evaluators for comparing two system variants on the same inputs.

##### 6.6.5.1 LangChain’s off-the-shelf evaluators

LangChain includes a set of ready-made evaluators in its evaluation module. You can run them directly in your own Python code (unit tests, CI, offline scripts) using `load_evaluator` and friends. LangSmith can also run these same LangChain evaluators as part of its dataset/experiment workflow (and attach results to traces and dashboards), but it is not required to use the evaluators themselves. A critical warning that applies to all “LLM-as-a-

judge” metrics: Most automated metrics are useful but imperfect. LLM-based judges often return a binary result per datapoint, and differences are more reliable when measured in aggregate over a dataset rather than trusted as a single verdict.

Use these evaluators for three practical jobs:

1. Regression protection (guardrails): Before you ship a change, check that you didn’t break contracts:

- JSON still parses and matches schema
- output still matches required patterns
- refusal rate or “empty retrieval” doesn’t spike

These are cheap tests that prevent silent breakage.

2. Variant selection (prompt/retriever/model comparisons): When you try prompt A vs prompt B (or chunking settings, rerankers, etc.), evaluators give you comparable scores so you can pick the better variant using consistent criteria.

3. Diagnosis (find the lever to pull): Evaluator breakdowns tell you what kind of failure you have:

- “wrong but fluent” → correctness/groundedness issues (judge or labeled criteria)
- “format broken” → JSON/regex/exact checks
- “almost right” → string distance / embedding distance can quantify closeness

### LLM-as-a-judge evaluators (call an LLM to grade)

These evaluators ask an LLM to act as a reviewer. Use them when “correctness” is about meaning and judgment, not exact bytes; paraphrases should count as correct, answers may be partially correct, and you care about qualities like coherence or helpfulness. They are especially useful when you have no perfect deterministic oracle for the task. They are a poor fit for hard contracts. If the output must be valid JSON, match a schema, or contain an exact ID, use deterministic evaluators first. Think of LLM judges as “quality reviewers”, not “format validators”. Operational guidance:

- Prefer dataset-level decisions. A single judge score can be noisy; trends across many examples are what you trust.
- Keep judging prompts stable. If you change the judge model or rubric, treat it like changing the measurement instrument.
- Use labelled variants when you have ground truth. Reference-free judging is useful, but it is easier to drift into “sounds good” scoring.

Evaluator-by-evaluator guidance

▪ “qa” (reference-based correctness): Use when you can provide a reference answer per example and you want semantic matching rather than exact string equality. This is the most common evaluator for Q&A regression tests: “Did we answer the question correctly, even if wording differs?”

```
from langchain_classic.evaluation import load_evaluator
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY

judge_llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)
evaluator = load_evaluator("qa", llm = judge_llm)

result = evaluator.evaluate_strings(
 input = "What is the capital of France?",
```

```

 prediction = "Paris.",
 reference = "Paris"
)
print(result)

```

ch\_06\scr\load\_evaluator\_qa.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.load_evaluator_qa
{'reasoning': 'CORRECT', 'value': 'CORRECT', 'score': 1}
```

- "context\_qa" (contextual correctness): Use when the system is supposed to answer using provided context (for example, retrieved documents), and you want the judge to evaluate "is the answer correct given this context?" rather than "is the answer true in the real world?".
- "cot\_qa" (chain-of-thought contextual correctness): Use when your setup includes chain-of-thought-style reasoning and you want an evaluator configured specifically for that judging style. In practice, treat it as a specialized version of "context\_qa": it's still about contextual accuracy, but with prompts tuned for reasoning traces.
- "criteria" (rubric-based grading, usually reference-free): Use when "correctness" is not a single gold answer, but you can define what "good" looks like: conciseness, relevance, coherence, safety, policy alignment, etc. You can apply one criterion or several at once, and you can define your own domain-specific rubric.

```

from langchain_classic.evaluation import load_evaluator
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY

def print_criteria_eval(title: str, question: str, prediction: str, res: dict) -> None:
 line = "=" * 80
 print(f"\n{line}\n{title}\n{line}")
 print("INPUT :", question)
 print("PREDICTION:", prediction)

 # Common keys returned by CriteriaEvalChain
 value = res.get("value", res.get("label", "<?>"))
 score = res.get("score", "<?>")
 reasoning = (res.get("reasoning") or "").strip()

 print(f"\nRESULT: value={value} | score={score}")
 if reasoning:
 print("\nREASONING:\n" + reasoning)
 else:
 print("\nREASONING: <none>")

 # If other fields exist, print them so nothing is hidden
 extras = {k: v for k, v in res.items() if k not in {"value", "label", "score", "reasoning"}}

 if extras:
 print("\nEXTRA FIELDS:")
 for k, v in extras.items():
 print(f"- {k}: {v}")

judge_llm = ChatOpenAI(
 model = "gpt-4o-mini",
 temperature = 0
)
crit_eval = load_evaluator(
 "criteria",
 llm = judge_llm,
 criteria = {
 "conciseness": "Is the submission concise and to the point?",
 "coherence": "Is the submission coherent, well-structured, and organized?"
 }
)
question = "Explain what an API is."
good_prediction = "An API is a contract that lets software systems communicate in a"

```

```

 defined way."
bad_prediction = (
 "An API is like, you know, a thingy that does stuff, and it's basically whatever the"
 " developer wants, so it's hard to say."
)
good_res = crit_eval.evaluate_strings(input = question, prediction = good_prediction)
bad_res = crit_eval.evaluate_strings(input = question, prediction = bad_prediction)

print_criteria_eval("GOOD (expected to score higher)", question, good_prediction,
good_res)
print_criteria_eval("BAD (expected to score lower)", question, bad_prediction, bad_res)

```

ch\_06\scr\load\_evaluator\_criteria.py

```

(.venv) C:\Users\yourname\llm-app> python -m src.load_evaluator_criteria
=====
=
GOOD (expected to score higher)
=====
=
INPUT : Explain what an API is.
PREDICTION: An API is a contract that lets software systems communicate in a
defined way.

RESULT: value=Y | score=1

REASONING:
To assess the submission based on the provided criteria, I will evaluate each
criterion step by step.

1. **Conciseness**:
 - The submission states, "An API is a contract that lets software systems
communicate in a defined way."
 - This definition is brief and directly addresses the question of what an
API is.
 - There are no unnecessary words or overly complex phrases; it conveys the
essential information in a straightforward manner.
 - Therefore, the submission is concise.

2. **Coherence**:
 - The submission is structured as a single, clear sentence.
 - It logically presents the concept of an API by defining it as a "contract"
and explaining its purpose (to let software systems communicate).
 - The use of the term "contract" is appropriate in the context of APIs, as
it implies an agreement on how systems will interact.
 - The sentence flows well and is easy to understand, indicating that it is
coherent.

After evaluating both criteria, the submission meets the requirements for
conciseness and coherence.

Y
=====
=
BAD (expected to score lower)
=====
=
INPUT : Explain what an API is.
PREDICTION: An API is like, you know, a thingy that does stuff, and it's
basically whatever the developer wants, so it's hard to say.

RESULT: value=N | score=0

REASONING:
To assess the submission based on the provided criteria, I will evaluate each
criterion step by step.

1. **Conciseness**:

```

- The submission states, "An API is like, you know, a thingy that does stuff, and it's basically whatever the developer wants, so it's hard to say."
  - The use of informal language ("like," "you know," "thingy") and vague terms ("does stuff," "whatever the developer wants") indicates a lack of precision.
  - The explanation is not direct and includes unnecessary filler phrases, which detracts from its conciseness.
  - Overall, the submission is not concise as it does not provide a clear and straightforward definition of what an API is.

2. \*\*Coherence\*\*:

- The submission lacks a clear structure. It does not present a logical flow of ideas or a well-defined explanation.
- The phrase "it's basically whatever the developer wants" is ambiguous and does not contribute to a coherent understanding of an API.
- The use of informal language and filler words makes it difficult to follow the intended meaning.
- Therefore, the submission is not coherent, as it fails to provide a well-organized and understandable explanation.

Based on the evaluations of both criteria, the submission does not meet the standards for conciseness or coherence.

N

- "labeled\_criteria" (rubric-based grading with a reference label): Use when you want rubric grading, but you also have a ground-truth label/reference that should guide the judgment. This is the safer choice for "correctness" expressed as a criterion, because it anchors the judge to something objective.

```
from langchain_classic.evaluation import load_evaluator
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY

def print_eval(title: str, question: str, prediction: str, reference: str, res: dict) -> None:
 # Be defensive: different versions may return slightly different keys
 value = res.get("value", res.get("label", "<?>"))
 score = res.get("score", "<?>")
 reasoning = res.get("reasoning", "").strip()

 line = "=" * 80
 print(f"\n{line}\n{title}\n{line}")
 print("QUESTION : ", question)
 print("REFERENCE : ", reference)
 print("PREDICTION: ", prediction)
 print(f"\nRESULT: value={value} | score={score}")
 if reasoning:
 print("\nREASONING:\n" + reasoning)

judge_llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

rubric = {
 "policy_alignment": (
 "Given the policy excerpt in the reference, does the answer follow it exactly "
 "and avoid adding unsupported claims?"
),
 "actionability": (
 "Does the answer state the user-facing decision and the next step clearly "
 "in one or two sentences?"
)
}

evaluator = load_evaluator(
 "labeled_criteria",
 llm = judge_llm,
 criteria = rubric
)

question = "Can I return an opened item after 30 days?"

policy_reference = (
 "Return policy: Items may be returned within 30 days of delivery. "
 "Opened items are returnable only if defective. After 30 days, returns are "
)
```

```

 "not accepted."
)

good_answer = (
 "No-returns aren't accepted after 30 days. If it's been less than 30 days and the "
 "item is defective, start a return claim with your order details."
)

bad_answer = (
 "Yes-opened items can be returned up to 60 days with the receipt; just bring it to "
 "the store for a refund."
)

good_res = evaluator.evaluate_strings(
 input = question,
 prediction = good_answer,
 reference = policy_reference
)
bad_res = evaluator.evaluate_strings(
 input = question,
 prediction = bad_answer,
 reference = policy_reference
)

print_eval("GOOD (expected Y)", question, good_answer, policy_reference, good_res)
print_eval("BAD (expected N)", question, bad_answer, policy_reference, bad_res)

```

ch\_06\scr\load\_evaluator\_criteria.py

```

(.venv) C:\Users\yourname\llm-app> python -m src.load_evaluator_criteria
=====
=
GOOD (expected Y)
=====
=
QUESTION : Can I return an opened item after 30 days?
REFERENCE : Return policy: Items may be returned within 30 days of delivery.
Opened items are returnable only if defective. After 30 days, returns are not
accepted.
PREDICTION: No-returns aren't accepted after 30 days. If it's been less than 30
days and the item is defective, start a return claim with your order details.

RESULT: value=Y | score=1

REASONING:
To assess whether the submission meets the criteria, I will evaluate each
criterion step by step.

1. **Policy Alignment**:
 - The reference states that items may be returned within 30 days of delivery
 and that opened items are returnable only if defective. It also specifies that
 after 30 days, returns are not accepted.
 - The submission states, "No-returns aren't accepted after 30 days." This
 aligns with the policy.
 - It also mentions, "If it's been less than 30 days and the item is
 defective, start a return claim with your order details." This part is also
 consistent with the policy, as it acknowledges that opened items can be
 returned if they are defective within the 30-day window.
 - The submission does not add any unsupported claims and strictly adheres to
 the policy provided in the reference.

2. **Actionability**:
 - The submission clearly states the user-facing decision: "No-returns aren't
 accepted after 30 days." This directly answers the user's question.
 - It also provides a next step: "If it's been less than 30 days and the item
 is defective, start a return claim with your order details." This is actionable
 and gives the user clear guidance on what to do if their situation meets the
 criteria.
 - The information is concise and fits within the one or two sentences
 requirement.

After evaluating both criteria, the submission meets both the policy alignment
and actionability requirements.

```

```

Y
=====
=
BAD (expected N)
=====
=
QUESTION : Can I return an opened item after 30 days?
REFERENCE : Return policy: Items may be returned within 30 days of delivery.
Opened items are returnable only if defective. After 30 days, returns are not
accepted.
PREDICTION: Yes—opened items can be returned up to 60 days with the receipt;
just bring it to the store for a refund.

RESULT: value=N | score=0

REASONING:
To assess whether the submission meets the criteria, I will evaluate each
criterion step by step.

1. **Policy Alignment**:
 - The reference states that items may be returned within 30 days of delivery
 and that opened items can only be returned if they are defective.
 - The submission claims that opened items can be returned up to 60 days with
 a receipt, which directly contradicts the reference policy.
 - Additionally, the submission does not mention that opened items are only
 returnable if defective, which is a critical part of the policy.
 - Therefore, the submission does not align with the policy and adds
 unsupported claims (the 60-day return period and the general return of opened
 items).

2. **Actionability**:
 - The submission does provide a clear next step by stating that the user
 should bring the item to the store for a refund.
 - However, since the information provided is incorrect, the actionability is
 undermined because the user would be misled into thinking they can return an
 opened item after 30 days without it being defective.

Given the analysis:
- The submission fails the policy alignment criterion because it contradicts
 the reference policy and includes unsupported claims.
- The actionability criterion is partially met, but the incorrect information
 makes it ineffective.

Since the submission does not meet the first criterion, it cannot be considered
to meet all criteria.

Thus, the final answer is:

N

```

- "score\_string" (continuous score 0–1): Use when you want a numeric score suitable for tracking improvements over time and comparing variants (prompt A vs B) without collapsing everything to pass/fail. It is useful for “how good” judgments where small improvements matter.

```

from langchain_classic.evaluation import load_evaluator
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY

def print_score_eval(title: str, question: str, prediction: str, res: dict) -> None:
 line = "=" * 80
 print(f"\n{line}\n{title}\n{line}")
 print("INPUT :", question)
 print("PREDICTION:", prediction)

 # score_string usually returns {"reasoning": "... Rating: [[10]]", "score": 1.0}
 score = res.get("score", "<?>")

```

```

reasoning = (res.get("reasoning") or "").strip()
print(f"\nSCORE (normalized 0..1): {score}")
if reasoning:
 print("\nREASONING:\n" + reasoning)
else:
 print("\nREASONING: <none>")

extras = {k: v for k, v in res.items() if k not in b"score", "reasoning"{}}
if extras:
 print("\nEXTRA FIELDS:")
 for k, v in extras.items():
 print(f"- {k}: {v}")

judge_llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

score_eval = load_evaluator(
 "score_string",
 llm = judge_llm,
 criteria = "correctness",
 normalize_by = 10
)

question = "What is the chemical formula for water?"
prediction = "H2O"

IMPORTANT: score_string ignores "reference" by design (hence your warning),
so don't pass it here.
result = score_eval.evaluate_strings(
 input = question,
 prediction = prediction
)

print_score_eval("SCORE_STRING (correctness)", question, prediction, result)

```

ch\_06\scr\load\_evaluator\_score\_string.py

```

(.venv) C:\Users\yourname\llm-app> python -m src.load_evaluator_score_string

This chain was only tested with GPT-4. Performance may be significantly worse
with other models.

=====
=
SCORE_STRING (correctness)
=====
=
INPUT : What is the chemical formula for water?
PREDICTION: H2O

SCORE (normalized 0..1): 1.0

REASONING:
The response provided by the AI assistant is correct, accurate, and factual.
The chemical formula for water is indeed H2O, which indicates that each
molecule of water consists of two hydrogen atoms and one oxygen atom. There are
no errors or omissions in the answer.

Rating: [[10]]

```

- "labeled\_score\_string" (continuous score with reference labels): Use when you want the benefits of a continuous score, but still want the judge anchored to ground truth. This is the best "scored correctness" option when you have references and want more nuance than pass/fail.

### Deterministic / algorithmic evaluators (no judge LLM)

These evaluators are “mechanical checks”. They don’t try to judge whether an answer is good in a human sense. They measure objective properties—equality, similarity, or format—so they are consistent across runs and useful as guardrails in tests and CI.

- "embedding\_distance": Compares prediction and reference by embedding both texts and computing cosine distance. Use it when wording can vary but meaning should stay close (paraphrases, short summaries). Lower distance means more similar.
- "string\_distance": Computes an edit-distance style similarity between prediction and reference (for example levenshtein, jaro\_winkler, damerau\_levenshtein). Use it when you expect near-identical text and want to measure "how many small edits away" the output is (typos, small formatting differences). There are 4 options that can be used for determining the distance:
  - levenshtein: Counts the minimum number of single-character edits (insert, delete, substitute) needed to turn one string into the other. Use it for near-exact outputs where "how many edits?" matters (templates, IDs, small typos).
  - damerau\_levenshtein: Like Levenshtein, but also treats a swap of two adjacent characters as one edit (e.g., "teh" → "the"). Use it as the default for human typing mistakes, where transpositions are common.
  - jaro: A similarity measure designed for short strings (often used for names), weighting matching characters and their positions. Use it for fuzzy matching of short identifiers or names when edit counts are less meaningful.
  - jaro\_winkler: Jaro plus a prefix boost: strings that share the same beginning score higher. Use it for names and cases where the prefix is especially important (e.g., "Michael..." variants).

To run the example, please install the rapidfuzz library:

```
(.venv) C:\Users\yourname\llm-app> python -m pip install rapidfuzz
```

```
from langchain_classic.evaluation import load_evaluator
dist_eval = load_evaluator("string_distance", distance = "damerau_levenshtein")
result = dist_eval.evaluate_strings(
 prediction = "Order ID: 12346",
 reference = "Order ID: 12345"
)
print(result)
```

ch\_06\scr\load\_evaluator\_string\_distance.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.load_evaluator_string_distance
{'score': 0.06666666666666667}
```

- "exact\_match": Checks whether prediction and reference are exactly the same string. Use it for strict contracts (fixed labels, IDs, exact expected outputs).

```
from langchain_classic.evaluation import load_evaluator
exact = load_evaluator("exact_match")
print(exact.evaluate_strings(prediction = "OK", reference = "OK"))
print(exact.evaluate_strings(prediction = "ok", reference = "OK"))
```

ch\_06\scr\load\_evaluator\_exact\_match.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.load_evaluator_exact_match
{'score': 1}
```

```
{'score': 0}
```

- "regex\_match": Checks whether the prediction matches a regular expression pattern. Use it when you care about format rather than exact text (IDs, dates, structured-looking outputs).

### JSON-focused evaluators (structure checks)

Use these when the model output is supposed to be machine-readable JSON (for example: tool calls, extraction payloads, structured reports). They let you treat “structured output” as a contract: first validate that the output is parseable, then decide whether you need exact equality, “close enough” similarity, or schema compliance.

- `JsonValidityEvaluator`: Checks whether a model output is valid JSON. Use it as the first gate in any pipeline that expects machine-readable JSON.
- `JsonEqualityEvaluator`: Compares a predicted JSON value to a reference JSON value after parsing. Use it for strict regression tests where the structured output must match exactly.
- `JsonEditDistanceEvaluator`: Computes a similarity-style distance between predicted JSON and reference JSON. Use it to track “getting closer” during iterative improvements when exact equality is too strict.
- `JsonSchemaEvaluator`: Validates a predicted JSON output against a JSON Schema. Use it to enforce a structured-output contract: required fields, allowed types, ranges, and whether extra keys are permitted.

```
from langchain_classic.evaluation import (
 JsonValidityEvaluator,
 JsonEqualityEvaluator,
 JsonEditDistanceEvaluator,
 JsonSchemaEvaluator,
)

def print_result(title: str, result):
 print(f"\n{title} ===")
 print(result)

def main():
 # 1) json_validity: can we parse it as JSON at all?
 validity = JsonValidityEvaluator()

 valid_json = '{"a": 1, "b": 2}'
 invalid_json = '{"a": 1, "b": }' # malformed JSON

 print_result("json_validity (valid)", validity.evaluate_strings(prediction = valid_json))
 print_result("json_validity (invalid)", validity.evaluate_strings(prediction = invalid_json))

 # 2) json_equality: does parsed JSON match reference JSON?
 equality = JsonEqualityEvaluator()

 same_as_ref = '{"a": 1}'
 different_from_ref = '{"a": 2}'
 ref = '{"a": 1}'

 print_result(
 "json_equality (match)",
 equality.evaluate_strings(prediction = same_as_ref, reference = ref)
)
 print_result(
 "json_equality (mismatch)",
 equality.evaluate_strings(prediction = different_from_ref, reference = ref)
)

 # 3) json_edit_distance: how close is prediction JSON to reference JSON?
 # Useful for "getting closer" rather than strict pass/fail.
 edit_distance = JsonEditDistanceEvaluator()

 pred_close = '{"a": 1, "b": 2}'
```

```

ref_close = '{"a": 1, "b": 3}' # small change
print_result(
 "json_edit_distance (small difference)",
 edit_distance.evaluate_strings(prediction = pred_close, reference = ref_close)
)

4) json_schema_validation: does JSON conform to a schema contract?
schema_eval = JsonSchemaEvaluator()

schema = {
 "type": "object",
 "properties": {
 "answer": {"type": "string"},
 "confidence": {"type": "number", "minimum": 0, "maximum": 1}
 },
 "required": ["answer", "confidence"],
 "additionalProperties": False
}

schema_ok = '{"answer": "Paris", "confidence": 0.92}'
schema_bad_missing = '{"answer": "Paris"}' # missing required field confidence
schema_bad_extra = '{"answer": "Paris", "confidence": 0.92, "extra": true}' # extra
 key not allowed
schema_bad_range = '{"answer": "Paris", "confidence": 1.7}' # out of range

print_result(
 "json_schema_validation (valid)",
 schema_eval.evaluate_strings(prediction = schema_ok, reference = schema)
)
print_result(
 "json_schema_validation (missing required field)",
 schema_eval.evaluate_strings(prediction = schema_bad_missing, reference = schema)
)
print_result(
 "json_schema_validation (extra field not allowed)",
 schema_eval.evaluate_strings(prediction = schema_bad_extra, reference = schema)
)
print_result(
 "json_schema_validation (value out of range)",
 schema_eval.evaluate_strings(prediction = schema_bad_range, reference = schema)
)

if __name__ == "__main__":
 main()

```

ch\_06\scr\load\_evaluator\_json.py

```

(.venv) C:\Users\yourname\llm-app> python -m src.load_evaluator_json
==== json_validity (valid) ====
{'score': 1}

==== json_validity (invalid) ====
{'score': 0, 'reasoning': 'Expecting value: line 1 column 15 (char 14)'}

==== json_equality (match) ====
{'score': True}

==== json_equality (mismatch) ====
{'score': False}

==== json_edit_distance (small difference) ====
{'score': 0.07692307692307693}

==== json_schema_validation (valid) ====
{'score': True}

==== json_schema_validation (missing required field) ====
{'score': False, 'reasoning': '<ValidationError: "\'confidence\' is a required property">'}

==== json_schema_validation (extra field not allowed) ====
{'score': False, 'reasoning': '<ValidationError: "Additional properties are not allowed (\\'extra\\\' was unexpected)">''}

```

```
 === json_schema_validation (value out of range) ===
 {'score': False, 'reasoning': "<ValidationError: '1.7 is greater than the
maximum of 1'>"}
```

#### 6.6.5.2 Pairwise evaluators

Pairwise evaluators compare two candidate outputs for the same input and ask “which is better?” rather than “what score is this?”. This is most useful when absolute scoring is hard or subjective (summaries, explanations, tone, “helpfulness”, policy-style answers), but choosing between two alternatives is straightforward. Instead of inventing a fragile numeric rubric, you run the evaluator across a dataset and aggregate outcomes as a win/loss/tie rate. Use pairwise evaluation for decisions like:

- Prompt A vs Prompt B (same retrieval and tools; only the prompt changes).
- Two retrieval strategies (same prompt; retrieval changes).
- Compression on vs off (same prompt/retrieval; context size changes).

To keep the comparison fair:

- Hold everything constant except the one thing you are testing (input, context, tools, temperature, etc.).
- Randomize whether a candidate is shown as “A” or “B” to reduce position bias.
- Trust the aggregate, not a single vote. A consistent win-rate shift across many examples is the signal you care about.

```
import random
from collections import Counter
from langchain_classic.evaluation import load_evaluator
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY

1) Judge model (LLM-as-a-judge)
judge_llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

2) Pairwise evaluator: compares prediction vs prediction_b for the same input
pairwise = load_evaluator("pairwise_string", llm = judge_llm)

3) Tiny dataset: each item has the same input, and two candidate outputs
examples = [
 {
 "input": "Explain what an API is.",
 "baseline": "An API lets programs talk to each other.",
 "candidate": "An API is a contract that defines how software components
 communicate, usually via requests and responses."
 },
 {
 "input": "Summarize: 'We will refund items within 30 days with proof of
 purchase.'",
 "baseline": "Refunds are possible.",
 "candidate": "Refunds are available within 30 days if you have proof of
 purchase."
 },
 {
 "input": "What is the capital of France?",
 "baseline": "Paris.",
 "candidate": "The capital of France is Paris."
 }
]

results = Counter()

for ex in examples:
 # Randomize order to reduce bias: sometimes baseline is A, sometimes candidate is A.
 a_is_candidate = random.choice([True, False])

 if a_is_candidate:
 pred_a = ex["candidate"]
 pred_b = ex["baseline"]
 else:
 pred_a = ex["baseline"]
 pred_b = ex["candidate"]
```

```

verdict = pairwise.evaluate_string_pairs(
 input = ex["input"],
 prediction = pred_a,
 prediction_b = pred_b
)

Common output includes a preference-like field (often "value" or "preference").
To be robust, check both.
preference = verdict.get("value") or verdict.get("preference")

Normalize outcome into win/loss/tie for the *candidate* variant.
Many pairwise judges return "A", "B", or "tie"/"equal". Handle common variants.
pref_norm = str(preference).strip().lower()

if pref_norm in ("a", "prediction", "first"):
 winner_is_candidate = a_is_candidate
elif pref_norm in ("b", "prediction_b", "second"):
 winner_is_candidate = not a_is_candidate
else:
 winner_is_candidate = None # tie/unknown

if winner_is_candidate is None:
 results["tie"] += 1
elif winner_is_candidate:
 results["candidate_win"] += 1
else:
 results["baseline_win"] += 1

print("Pairwise results:", dict(results))
total = sum(results.values()) or 1
print("Candidate win-rate:", results["candidate_win"] / total)

```

ch\_06\scr\load\_evaluator\_json.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.load_evaluator_pairwise_string
Pairwise results: {'candidate_win': 3}
Candidate win-rate: 1.0
```

In practice, replace the baseline and candidate strings with outputs produced by two real variants of your system (for example, prompt A vs prompt B, or retrieval strategy A vs retrieval strategy B). Run the comparison on a representative dataset: a handful of cases is useful for debugging, but you need dozens—and ideally hundreds—to make the win-rate stable. Use the candidate win-rate as your primary summary signal, then review a small sample of wins and losses to understand what the judge is consistently preferring and to catch any rubric drift or unintended incentives.

### 6.6.6 Statistical comparison and A/B testing

Metrics can vary across runs. With LLMs this is common: sampling (temperature), retrieval variability, tool failures, and evaluator disagreement all add noise. Comparing one baseline run to one candidate run can therefore be misleading: an apparent “winner” may reflect randomness rather than a real improvement. The goal is to avoid shipping changes that only look better once. Pick a small set of primary decision metrics (1–3). Treat the rest as diagnostics and guardrails. (These same signals can then be monitored continuously in production)

#### 6.6.6.1 How to compare statistically (offline)

Evaluate baseline and candidate on the same dataset, with everything else held constant (same prompts, same retriever settings, same tools, same evaluator). This gives you a fair, “apples to apples” comparison.

Then treat each dataset item as a paired experiment. For every example, you have two outcomes: what the baseline produced and what the candidate produced. Instead of jumping

straight to one average score, first look at the per-example delta: did the candidate do better, worse, or the same on that specific case? Those deltas form a distribution. If most deltas are small and randomly positive or negative, you are probably looking at noise. If the deltas are consistently positive (or they improve the cases you actually care about), you likely have a real improvement.

Finally, summarize that delta distribution with a comparison method that matches your metric type (binary, ordinal, continuous, or ranking). Common ways of comparing are as follows:

- Binary outcomes (success/fail, grounded/not grounded, error/no error):
  - Report baseline rate, candidate rate, and flip counts (baseline fail → candidate pass vs baseline pass → candidate fail).
  - Compare using a paired method for binary outcomes (for example, McNemar's test) or bootstrap the paired difference in rates.
  - Report uncertainty (confidence intervals) for the improvement.
- Ordinal or continuous scores (rubric 1–5, graded correctness):
  - Compare per-example score deltas (candidate minus baseline).
  - Report median and spread (for example IQR), plus tail risk (% below an unacceptable threshold).
  - Use a paired test suited to your scale (non-parametric if the scale is ordinal) or bootstrap the mean/median difference.
- Retrieval/ranking metrics (if you have labeled relevant documents):
  - Common: Recall@k, MRR, nDCG@k, computed per query.
  - Compare paired per-query differences and watch the tail (queries where Recall@k = 0).

If the system is stochastic (temperature > 0 or other sampling), repeat enough independent runs to estimate variability. Do not select the “best sample”, that biases results.

Change one major thing at a time to simplify attribution. When multiple changes are necessary, treat it as a package release and plan follow-up experiments that isolate prompt vs retriever vs model changes.

#### 6.6.7 From offline evaluation to production monitoring

Offline evaluation tells you whether a context configuration is promising, but real users will find edge cases and your data will drift over time. Production monitoring closes that loop. When you instrument your application for tracing, LLM-aware monitoring platforms can capture what was sent (prompt/context), what came back (outputs), performance timings, token usage, and errors for each run, and aggregate them into dashboards. LangSmith, for example, provides prebuilt dashboards and supports custom dashboards; its dashboards cover metrics such as error rates and token usage, and it supports monitoring and alerts for issues like latency increases and changes in feedback signals.

To avoid duplicating the experimentation section: in production you usually monitor the same metric families you defined in 1.6.5 (quality/task outcomes, reliability/safety, latency, and cost), but the goal is different. Here you care about trend, drift, and fast diagnosis. Monitoring views that are especially useful for context effectiveness like:

- Retrieval coverage: how often retrieval returns no documents (or only below-threshold matches) for queries that later generate an answer?

- Context size versus performance: how large is the injected context (in tokens) for different request types, and how does that correlate with latency and cost?
- Missed grounding signals in RAG: how frequently does the model answer “I don’t know” (or equivalent) even when relevant documents were available?
- Where failures concentrate: which prompts, graph nodes, retrievers, or tools show the highest error rates?
- Escalation signals: where does your application trigger handoff/escalation events, and what context and model outputs preceded them? (Escalation must be an application-defined signal captured as metadata or feedback.)

Traceability is the practical superpower: record “what changed”, “what context was sent”, and “what outcomes moved”. A simple way to do this is to attach version identifiers (prompt/template version, retriever version, model, compression settings) as run metadata/tags, so you can connect a quality shift to a specific change and compare slices of traffic by configuration. LangSmith explicitly supports grouping monitoring charts by tags and metadata, which is one concrete way to do this.

#### 6.6.7.1 User feedback loops

User-visible metrics complete the picture. Simple rating widgets, free-text feedback, or tagged events (“user asked to speak to a human,” “user repeated the same question”) help measure whether your carefully engineered context is actually helpful.

For a support assistant, you might define and track a ‘policy-grounded answer rate’—for example, the share of answers that cite at least one relevant policy source that your system considers current, user-reported helpfulness scores, and escalation rate to human agents. For an e-commerce assistant, you might correlate retrieval quality and context size with conversion metrics or returns related to misunderstanding product details.

Platforms such as LangSmith and similar observability tools let you attach feedback to traces (and even intermediate runs/spans) and query that data later. If you capture identifiers such as model, prompt version, retriever, or workflow/node names as trace metadata or run structure, you can then filter and analyze failures along those dimensions.

#### 6.6.8 Diagnosing context failures

Once you can see where your system is struggling, you can localise failures along the context pipeline instead of treating everything as “the model’s fault.” Common failure modes include:

- Missing content. The knowledge base does not contain the required information, or ingestion failed for some sources. If you evaluate retrieval against labelled queries, recall-oriented metrics (for example recall@k) will typically be poor, and manual inspection will often reveal questions whose answers are absent from the indexed, accessible corpus.
- Missed top-ranked documents. Relevant content exists but does not appear near the top of retrieval results. In offline evaluations with relevance labels, rank-sensitive metrics (for example MRR or nDCG@k) will typically degrade; in addition, you may see a higher rate of “no relevant document in top-k” if the retriever fails to surface any relevant item at all.
- Context window limits and truncation. The system may retrieve many relevant chunks but cannot fit them all into the model’s context window, so some content is omitted (often lower-priority chunks or older conversation turns). Longer questions, multi-document

tasks, or users with long histories are especially affected. If you track token usage and request timing, you will often see high prompt-token counts and sometimes increased latency; outputs may miss details that were retrieved but not included in the final prompt.

- Poor synthesis or misuse of context. The right documents are present in the prompt, but the model fails to combine them correctly, misreads constraints, or introduces unsupported statements. Trace-based (step-level) evaluation and rubric-based evaluation are useful here: if your system logs tool calls and retrieved passages, you can verify whether retrieval ran, whether the retrieved text actually contained the needed facts, and whether the final answer contradicted or ignored them.

Tools that trace end-to-end runs make these failures visible. If you store traces, inputs, and retrieved context, you can inspect individual cases, reconstruct what happened step by step, and see which nodes ran, which retrievers were invoked, what context was assembled, and what the model produced at each stage. Depending on provider and settings, replays may not be perfectly deterministic, but they are still invaluable for diagnosis.

#### 6.6.9 Techniques to improve context

Improvement work then becomes targeted rather than global. Typical interventions fall into four areas.

##### 6.6.9.1 Retrieval and ranking

If retrieval metrics show that relevant documents are often missing from the top-k results, you can adjust the retrieval strategy instead of just enlarging k. Hybrid retrieval that combines dense vector search with lexical methods such as BM25 often improves recall in practice, but must be validated per corpus and query mix, especially on specialised or jargon-heavy content. Re-ranking models provide a second pass that reorders candidate documents using a more expressive model, which is particularly valuable for ambiguous queries or high-value trading and compliance questions. At scale, index design matters: you typically choose between exact and approximate nearest-neighbour methods based on latency, recall, and cost constraints. For large vector sets, approximate methods such as HNSW or IVF with product quantisation are commonly used, trading some recall for speed and memory efficiency, where a small loss in recall may be acceptable in exchange for latency. Evaluation datasets let you quantify these trade-offs explicitly.

##### 6.6.9.2 Context processing and compression

If token usage is high and synthesis quality is poor, context processing is usually the next lever. LangChain's contextual compression retrievers wrap a base retriever with a document compressor (often model-based) that filters and shortens results based on the current query, returning a smaller set of documents and/or shorter extracted passages intended to be most relevant to the query. One common approach when you need to work with very long documents is map-reduce and similar summarisation patterns: you first summarise chunks locally, then summarise the summaries into a compact representation that fits in context while preserving structure. Chunking strategies are also central. Overlapping, semantics-aware chunking often reduces the risk that important sentences are split across chunks or separated from their headings. Poor chunking often shows up in evaluation as partial answers that miss prerequisites or caveats that live just outside the retrieved windows.

#### 6.6.9.3 *Prompt and workflow design*

When evaluators indicate that retrieval is working but answers are still unfaithful or incomplete, the issue is often prompt or workflow design. Clear instructions about how to use context (“always quote specific clauses,” “never guess when context is missing,” “explicitly list which documents support each claim”) can improve how consistently the model uses evidence and signals uncertainty. In LangGraph, you can separate retrieval, reasoning, and final generation into distinct nodes, with intermediate checks. For instance, the graph might retrieve documents, have the model list the key facts it found (scored by an evaluator for coverage), and only then ask it to write the final answer. Trajectory evaluations give you a way to verify that these steps happened (and in what order), and to catch regressions when prompts or graph structure change.

#### 6.6.9.4 *Feedback-driven tuning and automated optimisation*

This extends the monitoring loop described above by pushing some of the improvement work into automation. One pattern is to treat prompts and context formatting as tunable parameters and run controlled experiments (A/B variants) guided by evaluation metrics. Instead of manually “trying ideas”, you define what good looks like (your evaluator set), generate a set of candidate variants, and let the metrics converge on the variants that best satisfy your criteria.

Another pattern is to delegate parts of this search to optimisation services. Prompt-optimisation tools explore prompt and context variants at scale, score them with the same evaluators you use elsewhere, and surface the best-performing configurations. In practice, this is most useful when you have many interacting prompt fragments (system prompt, formatting, citations rules, tool instructions) and you want to improve them without introducing accidental regressions.

For scheduled execution, run evaluations on a schedule via CI cron or deployment cron jobs and attach evaluators so that new experiments are graded automatically. This gives you a repeatable “try → grade → select” pipeline, rather than relying on ad-hoc manual tuning.

## 6.7 SUMMARY

Language models do not “see” your application. Each call is bounded by a finite context window and, unless you explicitly replay or persist state, the model does not carry conversation state forward. It also cannot read your databases, files, or tools on its own; any external access happens only through tool calling, with tool outputs fed back into the next model input. Context engineering is the discipline of designing around these constraints by deciding what to include, what to compress, what to retrieve, and what to validate, so each step receives a short, focused bundle of high-value information instead of an unstructured dump.

To make this tractable, the chapter turns “context” into explicit building blocks and a repeatable per-call pipeline. Most information you pass to a model ends up as one of three container types: Document objects (retrieved knowledge with page\_content plus metadata), message objects (role-separated conversation turns such as SystemMessage, HumanMessage, AIMessage, ToolMessage), and application state snapshots (long-lived facts your system owns, fetched and compressed per request rather than pasted wholesale into prompts). Keeping “what you retain” separate from “what you send for this call” makes trimming, summarisation, and retrieval policies deterministic and debuggable.

As conversations and sources grow, the goal becomes preserving what matters without letting cost and noise explode. The chapter shows a layered history strategy: a compact running summary for long-range facts (issue, decisions, preferences, constraints) plus a small verbatim window for recent turns, with a clear trigger that folds older turns into the summary and trims the buffer back down. It then extends the same discipline to document compression: structured and salience-focused summarisation (including chain-of-density densification), contract-style two-pass summaries (extract a “coverage set”, then write a summary that must include required tokens), and hierarchical map-reduce pipelines for inputs that do not fit in one call.

Finally, the chapter treats context quality as measurable. More context is not automatically better: it consumes tokens, adds latency, and can still produce unfaithful answers. The chapter frames “effective context” across retrieval quality, contextual relevance, faithful generation, information synthesis, and correct use of implicit state (identity, preferences, prior commitments), and it closes with practical improvement levers (retrieval/ranking, compression, prompt/workflow design) and an evidence-driven loop from offline evaluation to production monitoring and feedback-driven tuning.

## 7 LCEL AND COMPOSABLE WORKFLOWS IN LANGCHAIN

---

This chapter explains LCEL (the LangChain Expression Language), the modern way to build LangChain workflows as small, composable pipelines. The core idea is that almost everything you use—prompts, models, retrievers, parsers, and even whole chains—exposes the same Runnable interface, so you can connect pieces with a consistent execution API. You will learn three things:

- How a Runnable behaves at runtime (single call, batch, streaming, async, and structured event streaming).
- How LCEL composition works (sequences, mapping/dictionary wiring, branching/routing, and parallel blocks).
- How to make pipelines production-friendly using configuration, retries, fallbacks, and output-fixing parsers, without scattering error handling across every call site.

The goal is not to memorize classes. The goal is to recognize the small set of composition patterns that keep complex LLM applications readable, testable, and easy to evolve.

## 7.1 RUNNABLES AND THE LANGCHAIN EXPRESSION LANGUAGE

Large language model applications often involve more than a single prompt and a single model call. Even simple assistants commonly combine prompts, models, tools, retrieval, and formatting steps into multi-stage workflows. LangChain’s LangChain Expression Language (LCEL) provides a declarative way to compose those workflows as pipelines built from a shared invocation interface. LCEL is built on top of a single core abstraction: the Runnable. A runnable is a unit of work with a standard set of execution modes (single-call, batch, and streaming) and a common composition surface, so you can build larger pipelines out of smaller steps without changing how they are invoked.

Many LangChain components are implemented as runnables, including prompt templates, retrievers, and output parsers, and composed pipelines behave like runnables too. This means the same execution methods (for example `invoke/ainvoke`, `batch/abatch`, `stream/astream`) apply consistently whether you are calling a single component or an assembled chain. This chapter focuses on LCEL as a way to build chains, and on how to combine runnables into sequential, branching, and parallel pipelines, manage their inputs and outputs, and make them more robust with retries, fallbacks, and output-fixing steps.

### 7.1.1 Runnables

In LangChain, a `Runnable` represents “something you can run” over some input to produce some output. It is the base protocol behind most components in `langchain-core`:

- Chat and LLM wrappers: turn prompts into model outputs
- Prompt templates: turn structured variables into model-ready text or messages
- Retrievers: turn queries into lists of documents
- Output parsers: turn model text into structured Python objects
- Custom functions and chains: turn arbitrary Python inputs into outputs

Formally, a `Runnable` exposes a standard interface with a small set of execution methods:

- `invoke/ainvoke`: run the computation on a single input (sync or async).
- `batch/abatch`: run the same computation on a list of inputs. Implementations can exploit model-side batching or internal parallelism.
- `stream/astream`: return an iterator or async iterator over chunks of the output as they are produced.
- `astream_log`: streams output as reported via the callback system.
- `astream_events`: generates a stream of structured events emitted during execution (useful for tracing intermediate steps).

All of these methods accept an optional `RunnableConfig`. The config lets you control tags, metadata, callbacks, concurrency limits, and recursion limits in a uniform way. The same config object applies whether you are calling a single model, a multi-step chain, or a graph node. On top of the execution methods, runnables expose a set of standard composition helpers:

- `with_config`: attach configuration (tags, metadata, callbacks, `max_concurrency`, `run_name`) without changing business logic.
- `with_retry`: wrap a runnable with a retry policy.
- `with_fallbacks`: declare backup runnables to try if the primary one fails.
- `map`: lift a runnable over sequences, so it can be applied to each element of a list.

- `assign` and `pick`: add or select keys in the dictionaries that flow between steps.
- `bind`: pre-bind keyword arguments such as model parameters.
- `get_graph` and `get_prompts`: expose a chain's internal structure and prompts for tooling and visualisation.

These methods are available on any runnable, including chains built from other runnables. That is the key design choice: once you build a pipeline, it behaves like a single component that supports the same API as its parts.

### 7.1.2 LCEL: the pipe-based composition layer

The LangChain Expression Language (LCEL) is a declarative way to compose runnables into chains (pipelines). A key operator is the pipe (`|`), which connects steps in a left-to-right flow. The pipe works because `Runnable` implements the `|` operator. In LCEL, you can pipe not only `Runnables`, but also 'Runnable-like' values such as callables and `dict` mappings; LangChain wraps them into the appropriate runnable forms so they can participate in the same invocation API.

```
prompt | model | parser
```

Using `|` constructs a `RunnableSequence`: a runnable whose `invoke` runs the leftmost component, passes its output to the next, and continues until the final result. Any `Runnable` (or runnable-like component) can participate in this sequence, including higher-level chains that are exposed as runnables.

LCEL is also the recommended alternative to older chain classes such as `LLMChain` (deprecated), because it encodes the dataflow directly in the expression (for example, `prompt | llm`). Instead of instantiating a chain class and wiring components through its constructor, you compose components with LCEL and get back a runnable. This gives you two practical benefits:

- The structure of the pipeline is visible in the code: you can usually read it in one line.
- Composed pipelines automatically support the standard runnable execution modes: sync and async invocation, batching, and streaming.

### 7.1.3 A first LCEL example: summarisation pipeline

The simplest useful chain consists of a prompt, a chat model, and an output parser. For example, a support or documentation assistant might need a short summary of a product description:

```
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import StrOutputParser
from .config import OPEN_API_KEY

prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You write short, factual summaries of product descriptions."
),
 (
 "human",
 "Summarise the following content in 3-4 bullet points.\n\n{content}"
)
]
)

llm = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.0)
chain = prompt | llm | StrOutputParser()
result = chain.invoke()
```

```

 {
 "content": "This laptop has a 14-inch display, 16GB RAM, "
 "and a battery rated for 10 hours of mixed use."
 }
)
print(result)

```

ch\_07\scr\summarization\_pipeline.py

Here:

- `prompt` formats the system and human messages using the `content` variable.
- `llm` sends the formatted messages to the model.
- `StrOutputParser` takes the model's response and returns a plain string (for example the assistant's content field).

The composed `chain` is a runnable. It can be invoked as shown, but also batched, streamed, or used as a sub-step in larger workflows.

#### 7.1.4 Runnables in a support assistant scenario

In a customer-care assistant, you often build a single pipeline out of small steps: normalise the user's input, retrieve a few relevant knowledge snippets, then ask a model to draft a short answer constrained to that evidence. `RunnableLambda` is a simple way to plug ordinary Python logic into that pipeline. It takes a regular callable (for example, a text normaliser) and wraps it as a Runnable so it can be composed with the same "pipe" operator used for other runnables. The point is not "lambda functions"; the point is turning a plain function into a first-class pipeline step.

In the following example, `normalise_question` is a plain function (`str → str`). By wrapping it in `RunnableLambda`, it becomes a Runnable that you can place at the start of `support_chain`, and it can be invoked the same way as the rest of the chain.

The dictionary in the middle of the chain is also a runnable. In LangChain, a `dict` literal inside a pipeline constructs a `RunnableParallel`: it runs each value against the same input and returns a new dictionary with the same keys. Here, both branches receive the normalised question. One branch passes it through as `{question}`; the other branch uses it to retrieve articles and builds `{context}`.

```

from .config import OPENAI_API_KEY
from typing import List
from langchain_core.runnables import RunnableLambda
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI

1. Sample knowledge base

KNOWLEDGE_ARTICLES: List[dict] = [
 {
 "id": "order-tracking",
 "title": "How to track your order",
 "content": (
 "You can track your order from the 'My Orders' section of your account. "
 "Tracking is available once the order has been shipped. "
 "If tracking information is missing for more than 24 hours, "
 "contact support with your order number."
)
 },
 {
 "id": "returns-policy",
 "title": "Returns and refunds policy",
 "content": (
 "You can return most items within 30 days of delivery. "
 "Refunds are issued to the original payment method once the returned "
 "item has been inspected. Some sale items may be final sale and "
)
 }
]

```

```

 "not eligible for return."
),
{
 "id": "shipping-times",
 "title": "Shipping times",
 "content": (
 "Standard shipping usually takes 3-5 business days after dispatch. "
 "Express shipping is typically 1-2 business days in eligible regions."
)
}
]

2. Runnables for preprocessing and retrieval

def normalise_question(raw: str) -> str:
 # Clean up user input:
 # - strip leading/trailing spaces
 # - collapse multiple spaces
 # - lower-case for simple matching
 cleaned = raw.strip()
 cleaned = " ".join(cleaned.split())
 return cleaned.lower()

normalize_runnable = RunnableLambda(normalise_question)

def simple_keyword_retriever(question: str) -> List[str]:
 # Very naive retriever:
 # - looks for simple keywords in the normalised question
 # - returns the 'content' fields of matching articles
 q = question.lower()
 articles: List[str] = []

 if "order" in q or "track" in q:
 articles.append(KNOWLEDGE_ARTICLES[0]["content"])
 if "return" in q or "refund" in q:
 articles.append(KNOWLEDGE_ARTICLES[1]["content"])
 if "shipping" in q or "delivery" in q:
 articles.append(KNOWLEDGE_ARTICLES[2]["content"])

 if not articles:
 articles.append(
 "No specific article matched this question. "
 "Explain that the customer should check the Help Center or contact support."
)

 return articles

3. LCEL support chain

support_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a support assistant for an e-commerce site. "
 "Answer clearly and concisely using ONLY the knowledge provided. "
 "If the information is not in the context, say you do not know "
 "and suggest contacting support."
),
 (
 "human",
 "User question: {question}\n\n"
 "Context articles:\n{context}"
)
]
)

llm = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.0
)

to_str = StrOutputParser()

Pipeline shape:
raw string
-> normalize_runnable (str -> str)
-> mapping builds {"question": str, "context": str}
-> prompt -> llm -> plain string
support_chain = (

```

```

normalizeRunnable
| {
 "question": lambda q: q,
 "context": lambda q: "\n\n".join(simple_keyword_retriever(q)),
}
| support_prompt
| llm
| to_str
}

5. Example usage (simplified)

def demo() -> None:
 user_question = "Where is my order????"
 # We pass a raw string; first step normalises it,
 # second step turns it into {question, context}, etc.
 answer = support_chain.invoke(user_question)

 print("User:", user_question)
 print("Assistant:", answer)

if __name__ == "__main__":
 demo()

```

ch\_07\scr\runnable\_lambda.py

In this example:

- The input to support\_chain is a raw string (the user's question).
- normalizeRunnable runs first and returns a normalised string.
- The dict mapping runs next (RunnableParallel). It receives that normalised string once and produces a dictionary:
  - question: the normalised string (passed through)
  - context: a single string built by retrieving matching articles and joining them

That dictionary is then formatted into the prompt, sent to the model, and parsed into a plain string output.

### 7.1.5 Execution modes: single calls, batches, and streams

One of LCEL's main advantages is that execution modes are orthogonal to the chain definition. You write the chain once and then decide at call time whether to invoke it on a single input, a batch, or as a stream. For example, the support chain above can be used in three ways:

Single call:

```

def demo_single() -> None:
 print("\n==== SINGLE CALL (invoke) ====\n")
 user_question = "Where is my order????"
 answer = support_chain.invoke(user_question)
 print("User:", user_question)
 print("Assistant:", answer)

```

Batch:

```

def demo_batch() -> None:
 print("\n==== BATCH CALL (batch) ====\n")
 questions = [
 "Where is my order????",
 "How long does shipping take to arrive?",
 "What is your returns policy?",
 "Can I get a refund if I don't like the product?"
]

 # support_chain.batch takes a list of inputs (here: list[str])
 answers = support_chain.batch(questions)

 for q, a in zip(questions, answers):
 print("User:", q)
 print("Assistant:", a)
 print("---")

```

Asynch:

```
async def demo_async() -> None:
 print("\n==== ASYNC CALLS (ainvoke / abatch) ====\n")
 questions = [
 "Where is my order?????",
 "What happens if my parcel is lost?",
 "Explain the refunds policy in simple terms."
]

 # 1) abatch: parallel async over many questions
 async_answers = await support_chain.abatch(questions)
 print("=>> Results from abatch:\n")
 for q, a in zip(questions, async_answers):
 print("User:", q)
 print("Assistant:", a)
 print("---")

 # 2) ainvoke: single async call (just to show the pattern)
 q2 = "How many days do I have to return an item?"
 a2 = await support_chain.ainvoke(q2)
 print("\n=>> Result from ainvoke:\n")
 print("User:", q2)
 print("Assistant:", a2)
```

If the underlying model supports token streaming, you can stream the chain's output:

```
llm = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.0,
 streaming = True # <-- enable token streaming
)

async def demo_async() -> None:
 questions = [
 "Where is my order?????",
 "What happens if my parcel is lost?",
 "Explain the refunds policy in simple terms."
]

 q1 = "Summarize the shipping times in one short sentence."
 print("\n=>> Streaming output (astream):\n")
 print("User:", q1)
 print("Assistant:", end = " ", flush = True)

 async for chunk in support_chain.astream(q1):
 # chunk is a partial string from the chain output
 print(chunk, end = "", flush = True)

 print("\n---")
```

For deeper debugging of long or complex chains, `astream_events` and `astream_log` can be used to observe structured events as they occur, such as prompt formatting, model calls, and parser outputs, while still working with the same LCEL expression.

```
Normal execution (final output only)
result = chain.invoke("Where is my order????")

Token streaming (final output streamed as chunks)
async for chunk in chain.astream("Where is my order????"):
 print(chunk, end = "", flush = True)

Debug streaming: structured lifecycle events (prompt/model/parser) while the SAME chain runs
async for event in chain.astream_events("Where is my order????", version = "v2"):
 print(event["event"], event.get("name"))

Debug streaming: low-level log patches while the SAME chain runs
async for log in chain.astream_log("Where is my order????"):
 if getattr(log, "ops", None):
 print(log.ops[-1])
```

### 7.1.6 Configuration, tags, and observability

`RunnableConfig` is LangChain's standard configuration object for runnable execution. It supports keys such as `tags`, `metadata`, `callbacks`, `run_name`, `max_concurrency`, and

`recursion_limit`, and it can be passed to `invoke/batch/stream` methods. The `with_config` method lets you bind a config to a runnable (or an LCEL-composed chain) without changing the runnable's internal implementation: it returns a new runnable that will run with the bound configuration. Common uses include:

- Tags: `RunnableConfig` supports tags for “this call and any sub-calls”, and `LangChain` explicitly describes them as a way to filter calls. Tags work best when they come from a small, reusable set (for example: `support`, `env:prod`, `channel:web`, `version:v1`, `experiment:ab_a`). Avoid putting unbounded or per-request values (order IDs, request IDs, user IDs) into tags, because high-cardinality tagging tends to reduce grouping value and can increase cost/complexity in many observability backends.
- Metadata: `RunnableConfig` also supports metadata for “this call and any sub-calls”, as a dictionary with string keys and JSON-serializable values. Metadata is a good place for high-cardinality identifiers and richer diagnostic context you may want to query later (tenant, correlation ID, user tier, locale, structured debugging fields), while keeping tags reserved for low-cardinality grouping.
- Callbacks: `RunnableConfig` supports callbacks for the call and any sub-calls. In config definition, tags are passed to all callbacks, and metadata is passed to `handle*Start` callbacks, which is the basis for tracing/logging integrations.
- Execution guardrails: `max_concurrency` limits the maximum number of parallel calls a runnable will make, which is useful for controlling parallelism and protecting downstream services from spikes. `recursion_limit` sets a cap on how many times a call can recurse (default 25 if not provided), which acts as a safety stop for runaway recursion in recursive compositions.

In practice, this is how you attach tags and metadata to a chain so traces are easier to filter and analyze:

```
configured_support_chain = support_chain.with_config(
 tags = ["support", "public_site"],
 metadata = {"component": "order_assistant", "version": "v1"}
)
```

Every call through `configured_support_chain` carries these tags and metadata, without changing the chain's internal logic. Below an example of how to read metadata and tags:

```
from __future__ import annotations
from typing import Any, Optional
from langchain_core.runnables import Runnable

class PrintTagsAndMetadata(Runnable[str, str]):
 def invoke(self, input: str, config: Optional[dict[str, Any]] = None, **kwargs: Any) -> str:
 cfg = config or {}
 print("TAGS:", cfg.get("tags", []))
 print("METADATA:", cfg.get("metadata", {}))
 return input

class SimpleAnswer(Runnable[str, str]):
 def invoke(self, input: str, config: Optional[dict[str, Any]] = None, **kwargs: Any) -> str:
 return f"Answer: I received your question: {input}"

pipeline = PrintTagsAndMetadata() | SimpleAnswer()

configured_pipeline = pipeline.with_config(
 tags = ["support", "public_site"],
 metadata = {"component": "order_assistant", "version": "v1"},
)
result = configured_pipeline.invoke("Where is my order?")
print("RESULT:", result)
```

ch\_07\scr\metadata\_and\_tags.py

### Example: Configurable parameters via RunnableConfig (fast vs quality)

This example shows how to use RunnableConfig to give a single support chain multiple “modes” at runtime. Instead of rebuilding chains for every variant, you define one chain and then switch between “fast” and “quality” behaviour either by tweaking model parameters (max\_tokens, temperature) or by selecting between pre-built model variants.

```

from __future__ import annotations
from typing import Any, Dict
from .config import OPENAI_API_KEY
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import ConfigurableField
from langchain_openai import ChatOpenAI

--- Basic safety checks so failures are clear up front ---
if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

def build_chain_with_configurable_fields():
 # Demonstrates configurable *fields* (same chain definition, runtime changes via
 # RunnableConfig.configurable).
 # We'll treat 'fast' vs 'quality' as different max_tokens / temperature settings.
 prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You are a first-line customer support assistant for ACME Telecom. "
 "Answer clearly and politely. If you are not sure, say so and "
 "suggest how the customer can get help from a human agent."
)
),
 ("human", "Customer question: {question}\n\nAccount flags: {account_flags}")
]
)

 base_model = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.0,
 max_tokens = 80 # default = "fast"
)

 # Mark selected constructor fields as configurable at runtime.
 model = base_model.configurable_fields(
 max_tokens = ConfigurableField(
 id = "max_out_tokens",
 name = "Max output tokens",
 description = "Maximum number of tokens in the model output"
),
 temperature = ConfigurableField(
 id = "temp",
 name = "Sampling temperature",
 description = "Higher values are more varied; 0 is most deterministic"
)
)

 chain = prompt | model | StrOutputParser()
 return chain

def build_chain_with_configurable_alternatives():
 # Demonstrates configurable alternatives (choose between pre-built runnable
 # variants at runtime). Default is "fast"; a "quality" alternative is selectable via
 # RunnableConfig.configurable.
 prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 (
 "You are a first-line customer support assistant for ACME Telecom. "
 "Answer clearly and politely. If you are not sure, say so and "
 "suggest how the customer can get help from a human agent."
)
),
 ("human", "Customer question: {question}\n\nAccount flags: {account_flags}")
]
)

```

```

]
)

fast_model = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.0,
 max_tokens = 80
)

Same provider/model family, but configured for longer answers.
quality_model = ChatOpenAI(
 model = "gpt-4.1-mini",
 temperature = 0.0,
 max_tokens = 220
)

model = fast_model.configurable_alternatives(
 ConfigurableField(id = "mode"),
 default_key = "fast",
 quality = quality_model
)

chain = prompt | model | StrOutputParser()
return chain

def demo() -> None:
 payload: Dict[str, Any] = {
 "question": "My broadband is down. Can I get a refund for today's outage?",
 "account_flags": "Customer has had 3 outages in 30 days; loyalty tier: Gold."
 }

 print("\n==== Demo A: configurable fields (same chain, runtime params) ====\n")
 chain_a = build_chain_with_configurable_fields()

 # Default (fast)
 print("FAST (defaults):\n")
 print(chain_a.invoke(payload))

 # Quality mode = more room + slightly more verbose style (still deterministic here)
 print("\nQUALITY (with_config(configurable=...)):\n")
 print(
 chain_a.with_config(
 configurable = {
 "max_out_tokens": 220,
 "temp": 0.0
 }
).invoke(payload)
)

 print("\n==== Demo B: configurable alternatives (choose a variant at runtime) ====\n")
 chain_b = build_chain_with_configurable_alternatives()

 # Default (fast)
 print("FAST (default_key='fast'):\n")
 print(chain_b.invoke(payload))

 # Select the alternative
 print("\nQUALITY (select alternative 'quality'):\n")
 print(chain_b.with_config(configurable={"mode": "quality"}).invoke(payload))

if __name__ == "__main__":
 demo()

```

ch\_07\scr\configurable\_runnable\_config.py

This example shows how to put “fast vs quality” behaviour behind configuration instead of forking your code. The scenario is a first-line ACME Telecom support assistant; the chain stays the same, and you choose speed or richness at runtime through `RunnableConfig` rather than rebuilding pipelines.

The file defines two patterns. In `build_chain_with_configurable_fields`, you take a single `ChatOpenAI` model and mark specific constructor arguments as configurable fields: `max_tokens` (exposed as `max_out_tokens`) and `temperature` (exposed as `temp`). With no config it behaves as a fast, short-answer model (`max_tokens=80`). When you call `chain.with_config(configurable={"max_out_tokens": 220, "temp": 0.0})`,

the same chain produces longer answers for that run; only the values in configurable change, not the chain structure.

In `build_chain_with_configurable_alternatives`, the “mode” switch is implemented by choosing between two pre-built model variants instead of editing fields. You define `fast_model` (short answers) and `quality_model` (same base model, higher `max_tokens`), then wrap them with `fast_model.configurable_alternatives(ConfigurableField(id="mode"), default_key="fast", quality=quality_model)`. The chain is again `prompt | model | StrOutputParser()`, but now `with_config(configurable={"mode": "quality"})` selects the richer variant as a whole.

The `demo()` function runs both patterns on the same payload. It first shows the “fields-configurable” chain with default and “quality” overrides, then the “alternatives-configurable” chain with the default “fast” variant and the “quality” alternative. The important idea is that mode selection (fast vs quality) lives entirely in `RunnableConfig.configurable`, while the prompt and downstream parsing stay identical.

### 7.1.7 Custom runnables

Not every step in a workflow is an LLM call. Some steps are simple deterministic transforms, database lookups, or calls to external APIs. These can be incorporated by:

- Subclassing `Runnable` and implementing/overriding `invoke` (and, when needed, `ainvoke`, `batch/abatch`, or `stream/astream`).
- Using helpers such as `RunnableLambda` to wrap existing Python callables so they can be composed with `|`, lifted over lists with `map()`, or used in parallel branches.

Once a custom component is a `Runnable`, you can `invoke` and compose it like any other runnable:

- It can be added to sequences and parallel blocks.
- It can be wrapped with helpers such as `with_config`, `with_retry`, and `with_fallbacks`.

## 7.2 SEQUENTIAL, BRANCHING, ROUTING AND PARALLEL PIPELINES WITH LCEL

So far, we have only used LCEL for simple, linear chains. Real applications often need more structure:

- Several steps in a fixed order.
- Routing logic that chooses different subchains based on the input.
- Parallel branches that compute independent results and then merge them.

LCEL supports these patterns through the same `Runnable` interface, using composition primitives such as `RunnableSequence` (sequential) and `RunnableParallel` (parallel), plus flow-control runnables such as `RunnableBranch` (conditional branching) and `RouterRunnable` (key-based routing).

### 7.2.1 Sequential pipelines

A sequential pipeline is a `RunnableSequence`: each step consumes the output of the previous step and produces the input for the next one. In LCEL, you can build these by composing smaller runnables with `|` rather than writing a monolithic function. A classic example is a two-step processing chain:

1. Extract structured data from unstructured text.
2. Transform it into a normalised JSON format.

In a support context, you might first extract key fields from a free-form incident description and then rewrite them into a ticket template. In an e-commerce context, you might parse product specs and then map them to your internal schema. The pattern is the same: each logical step is an LCEL segment, and the full pipeline is their composition.

#### Example: two-step extraction → normalisation

A sequential pipeline is most useful when each step has a single responsibility and produces a clean intermediate output. For example, take a free-form incident report and turn it into a normalised JSON record in two steps:

- Step 1: extract a small structured object (fields you care about).
- Step 2: rewrite it into your internal schema with stable keys and defaults.

In LCEL that becomes two runnable segments composed with `|`, so you can test each step independently and still invoke the whole thing as one chain.

### 7.2.2 Routing pipelines

Routing pipelines choose between multiple subchains based on the input. This pattern appears in multi-domain assistants, where you want different behaviour for “support”, “sales”, or “trading” questions, or in RAG systems that treat different document types differently. LCEL supports routing in two common ways:

- Predicate-based branching with `RunnableBranch`, which evaluates conditions in order, runs the first matching branch, and otherwise runs a default branch.
- Key-based routing with `RouterRunnable`, which selects a runnable from a mapping based on an input key and runs it on a provided payload. `RouterRunnable` expects an input object with fields `key` (the route name) and `input` (the payload to pass to the selected runnable).

A key-based router typically looks like this at a high level:

- A classifier runnable examines the original input and produces a RouterRunnable-compatible object: {"key": "<route>", "input": <payload>}.
- A RouterRunnable uses "key" to select one of several subchains from its mapping.
- The chosen subchain runs on "input" and returns the result.

Because RouterRunnable and RunnableBranch are themselves runnables, they can be composed like any other LCEL step, including inside larger pipelines and even inside other routers.

### Example: LCEL routing using RunnableBranch

This example builds a single assistant that can handle different kinds of questions. It first decides whether a message is about customer support, sales, or trading/market education. Then it sends the message to the most appropriate specialist voice and returns the reply. If it cannot decide, it asks the user to clarify. Finally, it runs a demo with test queries.

```
from .config import OPENAI_API_KEY
from typing import Dict
import os

from langchain_core.runnables import RunnableLambda, RunnableBranch
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI

1. Safety check

if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your .env file.")

2. Shared LLMs

Router model (deterministic, we only need a label)
router_llm = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.0)

Answering model (can be slightly more creative if you want)
answer_llm = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.2)

3. Classification (routing) chain

route_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a router for a multi-domain assistant. "
 "Given a user message, choose exactly one domain:\n"
 "-- support: order issues, returns, delivery, account problems\n"
 "-- sales: pricing, discounts, bulk orders, product selection\n"
 "-- trading: markets, stocks, portfolios, risk, ETFs\n"
 "Reply with only one word: support, sales, or trading."
),
 ("human", "User message: {message}")
]
)

route_chain = route_prompt | router_llm | StrOutputParser()

4. Domain-specific subchains

support_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a customer support assistant for an e-commerce site. "
 "Answer clearly and concisely using a friendly but professional tone."
),
 ("human", "User message: {message}")
]
)

support_chain = support_prompt | answer_llm | StrOutputParser()
```

```

sales_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a sales assistant for an e-commerce site. "
 "Help the user with pricing, discounts, and product choices. "
 "Be concise and concrete."
),
 ("human", "User message: {message}")
]
)
sales_chain = sales_prompt | answer_llm | StrOutputParser()

trading_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a trading and market analysis assistant. "
 "Give high-level, educational answers. "
 "Do not give personalised investment advice."
),
 ("human", "User message: {message}")
]
)
trading_chain = trading_prompt | answer_llm | StrOutputParser()

default_chain = RunnableLambda(
 lambda x: (
 "I am not sure which domain this belongs to. "
 "Please clarify if this is a support, sales, or trading question."
)
)

5. Router runnable

router_branch = RunnableBranch(
 [
 lambda x: "support" in x["route"].lower(),
 support_chain,
],
 [
 lambda x: "sales" in x["route"].lower(),
 sales_chain,
],
 [
 lambda x: "trading" in x["route"].lower(),
 trading_chain,
],
 default_chain, # runs if no condition matches
)
Full routing pipeline:
1) Classify the message into a route label.
2) Combine the route label with the original message.
3) Use RunnableBranch to select the correct subchain.
routing_chain = (
 {
 "route": router_branch,
 "message": lambda x: x["message"],
 }
 | router_branch
)

6. Example usage

def demo() -> None:
 test_messages = [
 "Where is my order? It was supposed to arrive yesterday.",
 "Can I get a discount if I buy 50 units of this product?",
 "Is it risky to invest in leveraged ETFs for the long term?",
 "Hi, just testing you."
]

 for msg in test_messages:
 answer = routing_chain.invoke({"message": msg})
 print("USER: ", msg)
 print("ASSISTANT: ", answer)
 print("-" * 60)

```

```
if __name__ == "__main__":
 demo()
```

ch\_07\scr\routing\_RunnableBranch.py

The routing works in two stages. First, `route_chain` classifies the user message. It formats the message into `route_prompt`, sends it to a deterministic `router_llm`, and parses the reply as a simple string label: "support", "sales", or "trading". Then, `routing_chain` builds an intermediate dictionary with both the route label and the original message:

- "route" is the output of `route_chain`.
- "message" is the raw user text.

This dictionary is passed to `router_branch` (a `RunnableBranch`). `router_branch` inspects the "route" field and:

- If it contains "support", runs `support_chain`
- If it contains "sales", runs `sales_chain`
- If it contains "trading", runs `trading_chain`
- Otherwise runs `default_chain`

The selected subchain receives the original "message" and produces the final answer, so the router decides \*which\* assistant persona to use, while each domain-specific chain decides how to answer.

#### Example: LCEL routing using RouterRunnable

This example shows a simple "traffic controller" for an assistant. A user can write very different kinds of messages—asking for help with a problem, requesting a quote, or talking about markets—and the system first figures out what the message is about, then sends it to the right specialist path.

So instead of one generic response flow, you get a small set of focused flows: a support flow for issues and incidents, a sales flow for pricing and purchasing questions, a trading flow for market-related requests, and a safe default flow when the message doesn't clearly match anything. The result is more consistent answers because each type of request is handled by the part of the system designed for it.

```
from __future__ import annotations
from typing import Dict, Any
from langchain_core.runnables import RunnableLambda
from langchain_core.runnables.router import RouterRunnable

--- 1) Domain-specific subchains (plain Python here, but they can be
prompts/models/retrievers/etc.) ---
def support_chain(payload: Dict[str, Any]) -> str:
 return f"[SUPPORT] Created ticket for user={payload.get('user_id')}:
 {payload['text']}"

def sales_chain(payload: Dict[str, Any]) -> str:
 return f"[SALES] Drafted offer response for user={payload.get('user_id')}:
 {payload['text']}"

def trading_chain(payload: Dict[str, Any]) -> str:
 return f"[TRADING] Classified intent and requested market data: {payload['text']}"

def default_chain(payload: Dict[str, Any]) -> str:
 return f"[DEFAULT] I don't know the domain yet; asking a clarification:
 {payload['text']}"

support = RunnableLambda(support_chain)
sales = RunnableLambda(sales_chain)
trading = RunnableLambda(trading_chain)
default = RunnableLambda(default_chain)

--- 2) RouterRunnable mapping: route key -> runnable ---
router = RouterRunnable(
 runnables = {
```

```

 "support": support,
 "sales": sales,
 "trading": trading,
 "default": default
 }
}

--- 3) Classifier: produces the RouterRunnable input shape: {"key": ..., "input": ...}
def classify_to_router_input(payload: Dict[str, Any]) -> Dict[str, Any]:
 text = payload["text"].lower()

 if any(k in text for k in ("refund", "incident", "error", "broken", "cannot",
 "can't", "problem")):
 route = "support"
 elif any(k in text for k in ("price", "quote", "buy", "purchase", "plan",
 "subscription")):
 route = "sales"
 elif any(k in text for k in ("stock", "btc", "price chart", "trade", "position",
 "portfolio")):
 route = "trading"
 else:
 route = "default"

 # RouterRunnable expects:
 # - "key": route name
 # - "input": payload to pass to the selected runnable
 return {"key": route, "input": payload}

classifier = RunnableLambda(classify_to_router_input)

--- 4) Full routing pipeline ---
routed_pipeline = classifier | router

--- 5) Demo ---
examples = [
 {"user_id": "u-123", "text": "I can't log in; getting an error after reset."},
 {"user_id": "u-456", "text": "Can you send me a quote for the Enterprise plan?"},
 {"user_id": "u-789", "text": "BTC price chart for the last 7 days and any big
 moves?"},
 {"user_id": "u-000", "text": "Hello there."}
]

for e in examples:
 print(routed_pipeline.invoke(e))

```

ch\_07\scr\routing\_RouterRunnable.py

```

(.venv) C:\Users\yourname\llm-app> python -m src.routing_RouterRunnable

[SUPPORT] Created ticket for user=u-123: I can't log in; getting an error after
reset.

[SALES] Drafted offer response for user=u-456: Can you send me a quote for the
Enterprise plan?

[TRADING] Drafted market education response for user=u-789: BTC price chart for
the last 7 days and any big moves?

[DEFAULT] I don't know the domain yet; asking a clarification: Hello there.

```

RouterRunnable is the “switchboard” that dispatches an input to one of several subchains based on a route key. The code first builds a routing table: a dictionary that maps a small set of domain labels (“support”, “sales”, “trading”, plus a “default”) to runnable subchains. Each subchain represents a focused handling strategy for a category of requests. This separation is the point: it keeps each path simple and specialized, and it prevents a single monolithic chain from accumulating conflicting rules and prompts as the system grows.

RouterRunnable does not decide the route by itself. Instead, the preceding classifier step converts the raw user payload into the exact envelope RouterRunnable expects: {“key”: <route>, “input”: <payload>}. The classifier inspects the incoming text, applies keyword heuristics, and chooses the most appropriate label. That label becomes the routing key; the original payload is preserved as the routing input so downstream chains receive the full context they need.

The pipeline is then composed as classifier | router. At runtime, each message is first classified into a route key, then RouterRunnable looks up the corresponding runnable from its mapping and executes it. If no rule matches, the "default" route provides a safe, explicit fallback behavior rather than failing or guessing.

### 7.2.3 Parallel pipelines with RunnableParallel

Some tasks benefit from doing several pieces of work at the same time. For example:

- Given a topic, generate a short summary, a list of follow-up questions, and a glossary of key terms.
- Given a customer query and context, generate an answer and a separate internal note for support agents.
- Given a document, produce both a user-facing summary and a structured extraction for analytics.

In LCEL, parallel pipelines are expressed with RunnableParallel. Conceptually, a parallel block is a runnable that:

- Receives one input.
- Feeds the same input to several branches concurrently.
- Returns a dictionary (mapping) containing each branch's output.

A typical pattern is:

```
from langchain_core.runnables import RunnableParallel, RunnablePassthrough

summarise_chain = (
 ChatPromptTemplate.from_messages(
 [
 ("system", "Summarise the topic in 3-4 bullet points."),
 ("human", "{topic}")
]
)
 | llm
 | StrOutputParser()
)

questions_chain = (
 ChatPromptTemplate.from_messages(
 [
 ("system", "Generate 3 follow-up questions about this topic."),
 ("human", "{topic}")
]
)
 | llm
 | StrOutputParser()
)

terms_chain = (
 ChatPromptTemplate.from_messages(
 [
 ("system", "List 5 key technical terms related to this topic."),
 ("human", "{topic}")
]
)
 | llm
 | StrOutputParser()
)

parallel_block = RunnableParallel(
 {
 "summary": summarise_chain,
 "questions": questions_chain,
 "key_terms": terms_chain,
 "topic": RunnablePassthrough(),
 }
)
```

The `RunnablePassthrough` branch simply forwards the original input, so the topic text remains available downstream alongside the generated artefacts. The parallel block is then usually followed by a synthesis step:

```
synthesis_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "Combine the information into a helpful explanation."
),
 (
 "human",
 "Topic: {topic}\n\n"
 "Summary:\n{summary}\n\n"
 "Follow-up questions:\n{questions}\n\n"
 "Key terms:\n{key_terms}"
),
],
)
full_chain = parallel_block | synthesis_prompt | llm | StrOutputParser()
```

Here:

- The first stage runs all three branches concurrently.
- The second stage reads `summary`, `questions`, `key_terms`, and `topic` and produces a single synthesised answer.

In practice, `RunnableParallel` is most useful for I/O-bound work such as multiple LLM calls, retrieval steps, or API calls, where parallelism can significantly reduce latency without complicating control flow.

#### 7.2.4 Map-style pipelines

When you need to apply the same chain to each element of a list, you can either:

- Call `batch/abatch` on the chain, or
- Use the `map` helper to construct a runnable that expects a list and applies another runnable to each element.

For example, processing a list of customer tickets with the same summarisation chain can be expressed as:

```
ticket_summariser = prompt | llm | StrOutputParser()
ticket_summariser_map = ticket_summariser.map()
summaries = ticket_summariser_map.invoke(ticket_list)
```

The resulting `ticket_summariser_map` is itself a runnable, composable with the rest of your workflow.

## 7.3 INPUT/OUTPUT HANDLING: MAPPINGS, PASSTHROUGHS, ASSIGNMENTS

LCEL chains pass data between steps as normal Python values. In toy examples the value is often just a string. In practical workflows it is often a dictionary that carries several fields at once: the user question, retrieved documents, intermediate summaries, flags, and so on. If you treat these dictionaries as random bags of data, chains quickly become hard to read and harder to debug. LCEL gives you a small set of primitives to keep the shape of your data explicit and predictable:

- mappings: build a new dictionary from the current input
- passthroughs: keep the current input while adding new fields
- assign and pick: add or select fields without ad-hoc dict manipulation
- structured outputs: parse model text into typed objects and validate them (so you fail fast when the output does not match)

The goal is to keep “what flows between steps” as clear as “which steps run”.

### 7.3.1 Dict-based inputs and mappings

In LCEL, many components (especially prompt templates) expect their input to be a dictionary with named fields (for example, `{"question": ..., "context": ...}`). A mapping is the simplest way to build that dictionary from whatever you currently have. A mapping is a pipeline step that takes the current input and produces a new dictionary for the next step. You define it using a normal Python `dict` where each key is the output field name you want to produce, and each value is something that can be executed using the current input:

- a runnable (it will be invoked using the current input)
- a plain Python callable (it will be called using the current input)

If you need to inject a constant value, wrap it in a callable (for example, `lambda _: "v1"`) so the mapping remains uniformly executable, so the mapping remains a uniform “compute these fields from the current input” step. After the mapping runs, the next step receives only the new dictionary, not the original input. Conceptually, a dict mapping is a fan-out: each value in the `dict` is computed from the same current input, and the results are collected into a new dictionary. That is why mappings are convenient for wiring prompts (named fields), and also why you must be explicit about preservation: any key you do not forward (directly or via passthrough/assign) will not exist downstream.

```
current input: {"message": "..."} (or any object you pass into invoke)
mapping = {
 "route": route_chain, # runnable computed from current input
 "message": lambda x: x["message"], # callable computed from current input
 "version": "v1" # constant
}
pipeline = mapping | next_step_that_expects_a_dict
```

#### Example: simple RAG-style support answerer

This example builds a tiny customer-support assistant. It takes a user question, cleans it up, and looks up one relevant help note from a small in-memory set of articles. It then asks a chat model to answer using only that retrieved note, returning a plain-text reply. The last line shows the whole flow in one call.

```
from .config import OPENAI_API_KEY
from typing import Dict, List
import os
```

```

from langchain_core.runnables import RunnableLambda
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI

Safety check
if not os.environ.get("OPENAI_API_KEY"):
 raise RuntimeError("OPENAI_API_KEY is not set.")

--- 1. Tiny in-memory "retriever" -----
class Article:
 def __init__(self, page_content: str):
 self.page_content = page_content

ARTICLES = [
 Article("You can track your order from the 'My Orders' section of your account."),
 Article("Returns are accepted within 30 days of delivery for most items.")
]

def build_features(inputs: Dict) -> Dict:
 # Take the raw input dict and return a new dict with a normalised 'question' field.
 # This is the main 'dict-based input' step.
 raw = inputs["question_raw"]
 normalised = " ".join(raw.strip().split()).lower()
 return {
 "question": normalised,
 # keep raw around if you want it later
 "question_raw": raw,
 }

feature_builder = RunnableLambda(build_features)

def simple_retriever(inputs: Dict) -> List[Article]:
 # Very simple 'retriever' that looks at the normalised question text.
 q = inputs["question"]
 if "return" in q:
 return [ARTICLES[1]]
 return [ARTICLES[0]]

retriever = RunnableLambda(simple_retriever)

--- 2. Prompt and model -----
formatting_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a support assistant. Use ONLY the provided context. "
 "If the answer is not in the context, say you don't know."
),
 (
 "human",
 "Question: {question}\n\n"
 "Context:\n{context}"
)
]
)

llm = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.0)
to_str = StrOutputParser()

--- 3. LCEL RAG chain -----
rag_chain = (
 # First step: turn {"question_raw": "..."} into
 # {"question": "...", "question_raw": "..."}
 feature_builder
 # Second step: build the prompt inputs from that dict
 | {
 "question": lambda inputs: inputs["question"],
 "context": lambda inputs: "\n".join(
 doc.page_content
 for doc in retriever({"question": inputs["question"]})
),
 }
 | formatting_prompt
 | llm
 | to_str
)

if __name__ == "__main__":
 answer = rag_chain.invoke({"question_raw": "How do I return an item?"})
 print(answer)

```

ch\_07\scr\dict\_base\_input\_and\_mapping.py

The script builds a tiny retrieval-augmented support assistant. It defines a minimal “knowledge base” as two Article objects, each holding one support sentence. The build\_features function takes an input dictionary containing question\_raw, normalises it (trim, collapse spaces, lowercase), and returns a new dictionary that includes both the normalised question and the original text. Wrapped as RunnableLambda, this becomes the first pipeline step. Next, simple\_retriever inspects the normalised question and selects one Article: if the text contains “return” it chooses the returns policy; otherwise it chooses the order-tracking note. That selector is also wrapped as a runnable.

The pipeline then builds the variables needed for the prompt: it passes the normalised question through unchanged and creates a context string by invoking the retriever and joining the selected article text. ChatPromptTemplate formats a system instruction (“use only the provided context”) plus a human message containing the question and context.

### 7.3.2 Passthroughs and field preservation

RunnablePassthrough is a helper runnable that passes its input through unchanged (it behaves like an identity function). When the input is a dict, it can also be used in “passthrough + add fields” patterns by using RunnablePassthrough.assign(...), which returns a runnable that merges new key-value pairs into the original dictionary. This is useful when:

- You already have a dictionary with useful fields and want to carry it forward to the next runnable unchanged.
- You want to compute one or more additional fields while keeping all existing keys intact.

Instead of rebuilding the dictionary manually, use RunnablePassthrough.assign(...) to attach only the new keys; the original input dict is preserved and the new fields are added on top. If you assign a key that already exists, the assigned value overwrites the original value for that key, so treat assigned field names as part of your pipeline’s data contract.

#### Example: keep original fields and add a summary

```
from .config import OPENAI_API_KEY
from langchain_core.runnables import RunnableParallel, RunnablePassthrough
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.0)
to_str = StrOutputParser()

summarise_prompt = ChatPromptTemplate.from_messages(
 [
 ("system", "Summarise the user's question in one short sentence."),
 ("human", "{question}")
]
)

summarisation_chain = summarise_prompt | llm | to_str

Option A: Explicit field preservation with RunnablePassthrough()
(you manually list fields you want to carry forward)

enrich_chain_parallel = RunnableParallel(
 {
 # Keep existing fields explicitly
 "question": RunnablePassthrough(),
 "articles": RunnablePassthrough(),
 # Add a new derived field
 "summary": summarisation_chain
 }
)
```

```

 }

 # -----
 # Option B: Preserve ALL fields automatically with RunnablePassthrough.assign(...)
 # (you only declare what you want to add/overwrite)
 # -----
 enrich_chain_assign = RunnablePassthrough.assign(
 summary = summarisation_chain
)

 if __name__ == "__main__":
 input_payload = {
 "question": "Where is my order? It was shipped three days ago.",
 "articles": ["Tracking is available in the 'My Orders' section."],
 "priority": "high" # extra field to show the difference
 }

 enriched_a = enrich_chain_parallel.invoke(input_payload)
 print("A keys:", enriched_a.keys())
 # A keys: dict_keys(['question', 'articles', 'summary'])
 # Note: "priority" is NOT preserved unless you pass it through explicitly.

 enriched_b = enrich_chain_assign.invoke(input_payload)
 print("B keys:", enriched_b.keys())
 # B keys: dict_keys(['question', 'articles', 'priority', 'summary'])
 # Note: ALL original fields are preserved, and "summary" is added.

```

ch\_07\scr\runnablepassthrough.py

The code contrasts two enrichment patterns. In `RunnableParallel`, each field you want to keep must be explicitly forwarded with `RunnablePassthrough()`, so unlisted keys are dropped. `RunnablePassthrough.assign(summary=...)` instead treats the whole input dict as the passthrough baseline, then merges the computed summary, preserving every original field automatically.

### 7.3.3 Itemgetter – manual construction

When you build dict-shaped payloads in a pipeline, you typically do two things at once:

- compute new fields (summaries, context strings, classifications)
- carry forward existing fields unchanged (question, ids, user message)

`RunnablePassthrough` can be used to pass an input through unchanged inside a mapping or parallel block (and it can also add keys when the input is a dict). Sometimes, though, you do not want to forward everything. You want a small, explicit payload with only a few original keys plus a few derived keys. One simple option is `operator.itemgetter`. `itemgetter` is a Python standard-library helper that returns a callable which fetches item(s) from its operand using `getitem()`. For dict-shaped inputs, `itemgetter("question")` is equivalent to a tiny function that returns `inputs["question"]`. Using `itemgetter` here is a style choice that can make two things clearer:

- Readability: it makes “this key is forwarded as-is” obvious at a glance.
- Payload discipline: you explicitly choose which keys survive into the next step without rewriting dict-copy boilerplate.

For example, if you currently forward `question` using a lambda, `itemgetter` expresses the same intent more directly.

#### Example: forwarding with `itemgetter` while still computing derived fields

```

rag_chain = (
 feature_builder
 | {
 # Forward a field unchanged (clearer than lambda inputs: inputs["question"])
 "question": itemgetter("question"),
 # Compute a derived field

```

```

 "context": lambda inputs: "\n".join(
 doc.page_content
 for doc in retriever.invoke({"question": inputs["question"]}))
)
}
| formatting_prompt
| llm
| to_str
)

```

`itemgetter("x")` raises `KeyError` if "x" is not present. In most pipelines, that is a feature: it makes the "data contract" explicit and fails early when an upstream step did not produce what you expected. If you genuinely need optional keys with defaults, use a small callable that does `inputs.get("x", default)` rather than `itemgetter`, so the defaulting behaviour is obvious in the code.

What to use:

- Passthroughs are best when you want to keep the whole dictionary intact and add fields.
- `itemgetter`-based manual construction is best when you want to keep only selected fields and add a few derived ones.
- `assign/pick` then formalise "add a field" and "restrict fields" as compact helpers.

### 7.3.4 Assigning and picking fields

Mappings and `RunnableParallel` work well when you are building dictionaries from scratch. When you already have a chain and want to:

- Add a new field to its output, or
- Restrict what is passed downstream

`assign` and `pick` give you compact helpers. Conceptually:

- `assign(new_key=some_runnable)` takes the current dictionary, runs `some_runnable` with it, and adds the result under `new_key`.
- `pick("field1", "field2")` keeps only the specified keys for the next step.

**Example: attach a summary, then keep only what is needed**

```

from .config import OPENAI_API_KEY
from langchain_core.runnables import RunnablePassthrough
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import StrOutputParser
from langchain_openai import ChatOpenAI

llm = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.0)
to_str = StrOutputParser()

summary_prompt = ChatPromptTemplate.from_messages(
 [
 ("system", "Summarise the user's question in at most 10 words."),
 ("human", "{question}")
]
)

summarisation_chain = summary_prompt | llm | to_str

Start from a chain that just passes through the input dict
base_chain = RunnablePassthrough()

1) Add a 'summary' field based on the current dict
with_summary = base_chain.assign(
 summary = summarisation_chain
)

2) Keep only the summary for the next step
final_chain = with_summary.pick("summary")

if __name__ == "__main__":
 result = final_chain.invoke({"question": "How do I change my delivery address?"})

```

```
print(result) # {'summary': 'How to change delivery address'}
```

ch\_07\scr\assign\_and\_pick.py

## 7.4 USING STRUCTURED OUTPUTS IN LCEL

LCEL pipelines are easier to maintain when each step returns an output shape that the next step can consume without guesswork. Free-form prose is fine for a terminal response to a user, but it is a poor interface between steps in a multi-stage workflow. A model response that is “mostly JSON” or “kind of a table” forces downstream code to do brittle string handling, and failures often surface one or two steps later, far from the real cause. In this section, “structured output” means the model produces machine-usable data (for example JSON objects or Pydantic models) rather than plain text. For LCEL chains, the key goal is that the chain returns a typed value (for example a `dict`, a Pydantic object, or a list) instead of a string. Two common approaches are:

- Parse structured data from model text (the model still produces text; you parse it into a typed value).
- Use provider-native structured output via `with_structured_output` when the model integration supports it (the wrapper returns outputs formatted to match the given schema; the underlying mechanism depends on the provider/model).

You often mix both: provider-native structured output where supported, and output parsers when you need additional processing or validation, a specific format, or you are using a model integration that does not implement `with_structured_output`.

LangChain provides output parsers that transform the raw text generated by a model into structured data types your code can use directly. Output parsers remain valuable when working with models that do not support structured output natively, or when you require additional processing or validation beyond a model’s built-in structured output capabilities.

We’ll start with parsing because it works everywhere: any chat model can emit text, so output parsers apply even when an integration does not support `with_structured_output`. Once the parsing boundary is clear (schema → format instructions → parse → fail fast), provider-native structured output is the same contract with less plumbing: the integration steers the model and parses the response for you, but schema violations are still treated as first-class failures.

### 7.4.1 Parsing structured data with output parsers

Output parsers turn model text into a structured type. Attach the parser at the end of an LCEL pipeline so the pipeline returns a typed value rather than raw text. The core pattern is: define a schema, inject the parser’s format instructions into the prompt, parse into a typed object, and treat parse failure as a normal error path (retry, repair, fallback, or fail fast). The following example fits the trading assistant scenario by turning a news headline into a typed risk assessment.

```
from __future__ import annotations
from typing import Literal
from .config import OPENAI_API_KEY
import os
from pydantic import BaseModel, Field
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers.pydantic import PydanticOutputParser

class RiskAssessment(BaseModel):
 risk_level: Literal["low", "medium", "high"] = Field(
 description = "Overall risk level for taking the position given the headline."
)
```

```

reasoning: str = Field(description = "Short justification grounded in the headline.")
recommended_action: Literal["hold", "reduce", "exit", "hedge"] = Field(
 description = "What to do with the position."
)
confidence: float = Field(
 ge = 0.0, le = 1.0, description = "Confidence in this assessment, from 0 to 1."
)
parser = PydanticOutputParser(pydantic_object = RiskAssessment)

prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a trading risk assistant. "
 "Return ONLY data that matches the required schema.\n{format_instructions}",
),
 (
 "human",
 "Headline: {headline}\n"
 "Position: {position}\n"
 "Time horizon: {horizon}\n"
 "Assess the risk and recommend an action.",
),
],
).partial(format_instructions = parser.get_format_instructions())

llm = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.2)

chain = prompt | llm | parser

result: RiskAssessment = chain.invoke(
 {
 "headline": "Regulator opens probe into major chip supplier accounting
 practices",
 "position": "Long 2% portfolio weight",
 "horizon": "2 weeks"
 }
)

print(result.model_dump())

```

ch\_07\scr\PydanticModel.py

```
(.venv) C:\Users\alexc\llm-app\ch_07> python -m src.PydanticModel
{'risk_level': 'high', 'reasoning': 'A regulatory probe into accounting
practices can lead to significant negative news, potential fines, or stock
price volatility, posing a high risk to the position within a short 2-week
horizon.', 'recommended_action': 'hedge', 'confidence': 0.9}
```

RiskAssessment is a Pydantic BaseModel that defines the contract the LLM must satisfy. Literal types restrict risk\_level and recommended\_action to fixed vocabularies, preventing drift into unexpected labels. Field metadata documents intent and adds validation constraints; confidence is bounded to 0–1 via ge/le. PydanticOutputParser uses this model to generate format instructions inserted into the system message, steering the model toward a JSON shape that matches the schema. After the LLM responds, the parser loads and validates the JSON, returning a RiskAssessment instance. model\_dump() then exposes a plain dict for downstream code. Validation failures raise errors, so bad outputs are caught early.

The operational reality of the parsing approach is simple: parsing failures are normal error cases. Treat the parser as an interface boundary that can throw, and decide how your pipeline should respond (retry, repair, fallback, or fail fast). Output parsing exists because LLMs are probabilistic text generators, so a clean schema does not eliminate the need for error handling.

#### 7.4.2 Output parser catalog

These are some of the most common parsers used to get machine-readable data (like dictionaries or class instances). This is not exhaustive; treat it as a quick orientation list.

- `PydanticOutputParser`: Uses a Pydantic model to define a schema. It is a common choice when you want a strict, typed schema enforced by Pydantic, ensuring the LLM response matches your specific class structure.
- `JsonOutputParser`: Parses the model's text into a JSON value (typically a Python dict when the top-level JSON is an object).
- `StructuredOutputParser`: A lightweight alternative that uses `ResponseSchema` objects to define expected fields.
- `YamlOutputParser`: Parses LLM output written in YAML into Python data structures.
- `XMLOutputParser`: Parses LLM output written in XML into Python data structures.

#### 7.4.3 Basic and list parsers

Used for simple transformations or splitting text.

- `StrOutputParser`: The simplest parser; it just takes the raw string output from the LLM and returns it. It is often the default in LCEL chains.
- `CommaSeparatedListOutputParser`: Splits a string like “apple, banana, cherry” into a Python list: ['apple', 'banana', 'cherry'].
- `NumberedListOutputParser`: Handles lists prefixed by numbers (for example, “1. First item, 2. Second item”).

#### 7.4.4 Primitive type parsers

These force the LLM output into a specific Python primitive type.

- `BooleanOutputParser`: Forces the output to True or False. Useful for classification or binary decision-making.
- `DatetimeOutputParser`: Parses date/time strings into actual Python datetime objects.
- `EnumOutputParser`: Ensures the output is exactly one value from a predefined set of options (an Enum).

#### 7.4.5 Utility and correction parsers

These “meta-parsers” wrap other parsers to handle errors or combine results.

- `OutputFixingParser`: If the primary parser fails (for example, malformed JSON), this parser sends the error and the original output back to the LLM to ask for a “fix.”
- `RetryOutputParser`: Wraps a parser and retries by re-prompts for a corrected completion when parsing fails.
- `CombiningOutputParser`: Allows you to run multiple parsers at once (for example, getting a Boolean and a List in one go).

#### 7.4.6 Specialised and advanced parsers

- `PandasDataFrameOutputParser`: Designed to format data specifically for ingestion into a Pandas DataFrame.
- `RegexParser`: Uses regular expressions to extract specific groups of information from a block of text.
- `RegexDictParser`: Extracts multiple named groups from a regex into a dictionary.

#### 7.4.7 Parsers Examples

To keep these examples deterministic, we use `FakeStreamingListLLM` from `langchain_core.language_models.fake`, which is a simple fake model implementation intended for testing and documentation examples.

##### **JsonOutputParser**

```
from langchain_core.language_models.fake import FakeStreamingListLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import JsonOutputParser

prompt = ChatPromptTemplate.from_template(
 "Return ONLY a JSON object with keys: ticker (string), action (string)."
)

model = FakeStreamingListLLM(responses = [{"ticker": "MSFT", "action": "BUY"}])
parser = JsonOutputParser()

chain = prompt | model | parser
result = chain.invoke({})

print(result)
```

ch\_07\scr\JsonOutputParser.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.JsonOutputParser
{'ticker': 'MSFT', 'action': 'BUY'}
```

##### **XMLOutputParser**

This example requires an XML library. Install `defusedxml` before running this example:

```
(.venv) C:\Users\yourname\llm-app> python -m pip install defusedxml
```

```
from langchain_core.language_models.fake import FakeStreamingListLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import XMLOutputParser

prompt = ChatPromptTemplate.from_template(
 "Return ONLY XML with root <trade> and children <ticker> and <action>."
)

model = FakeStreamingListLLM(
 responses = ["<trade><ticker>MSFT</ticker><action>BUY</action></trade>"]
)
parser = XMLOutputParser()

chain = prompt | model | parser
result = chain.invoke({})

print(result)
```

ch\_07\scr\XMLOutputParser.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.XMLOutputParser
{'trade': [{'ticker': 'MSFT'}, {'action': 'BUY'}]}
```

##### **CommaSeparatedListOutputParser**

```
from langchain_core.language_models.fake import FakeStreamingListLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import CommaSeparatedListOutputParser

prompt = ChatPromptTemplate.from_template(
 "Return ONLY a comma-separated list of three tickers."
)

model = FakeStreamingListLLM(responses=["MSFT", "AAPL", "NVDA"])
parser = CommaSeparatedListOutputParser()

chain = prompt | model | parser
```

```
result = chain.invoke({})
print(result)
```

ch\_07\scr\CommaSeparatedListOutputParser.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.CommaSeparatedListOutputParser
['MSFT', 'AAPL', 'NVDA']
```

### EnumOutputParser

```
from langchain_core.language_models.fake import FakeStreamingListLLM
from langchain_core.prompts import ChatPromptTemplate
from langchain_classic.output_parsers.enum import EnumOutputParser
from enum import Enum

class TradeAction(str, Enum):
 BUY = "BUY"
 HOLD = "HOLD"
 SELL = "SELL"

prompt = ChatPromptTemplate.from_template(
 "Choose exactly one action: BUY, HOLD, or SELL. Return ONLY the action."
)

model = FakeStreamingListLLM(responses = ["HOLD"])
parser = EnumOutputParser(enum = TradeAction)

chain = prompt | model | parser
result = chain.invoke({})

print(result)
print(result.value)
```

ch\_07\scr\EnumOutputParser.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.EnumOutputParser
TradeAction.HOLD
HOLD
```

#### 7.4.8 Controlled generation with `with_structured_output`

`with_structured_output` wraps a chat model in a runnable that returns parsed structured data instead of a message object. You provide an output schema (commonly a Pydantic model class, a `TypedDict`, or a JSON schema), and the wrapper asks the model to produce data in that shape and then parses the response into a Python value. When the schema is a Pydantic class and `include_raw=False`, the runnable outputs an instance of that Pydantic class. Otherwise (again with `include_raw=False`), it outputs a dict. This lets downstream code access fields directly (for example `result.risk_level`) without writing custom text parsing.

The `include_raw` flag controls how parsing failures are surfaced. If `include_raw=False`, parsing errors are raised. If `include_raw=True`, the runnable returns a dict with the raw model message plus parsing results: `raw` (the `BaseMessage`), `parsed` (either the structured output or `None` if parsing failed), and `parsing_error` (the exception or `None`).

Some integrations also expose a `strict` parameter. For example, in the `ChatOpenAI` integration, `strict=True` requests provider-side enforcement: model output is guaranteed to exactly match the schema, and the input schema is validated against the provider's supported schema constraints. With `strict=False`, the input schema is not validated, and the model output is not validated; you should treat the structured output as best-effort rather than guaranteed. If

strict is None, the strict argument is not passed to the model provider. In the ChatOpenAI integration, strict can only be set when method is 'json\_schema' or 'function\_calling' (otherwise it must be left as None). Even with provider-native enforcement, keep the same operational posture as with parsers: treat schema mismatch (or parsing\_error when include\_raw=True) as a normal failure mode and decide locally whether to retry, repair, fall back, or fail fast.

```

from __future__ import annotations
from .config import OPENAI_API_KEY
import os
from typing import Literal
from pydantic import BaseModel, Field
from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate

class RiskAssessment(BaseModel):
 risk_level: Literal["low", "medium", "high"] = Field(
 description = "Overall risk level for taking the position given the headline."
)
 reasoning: str = Field(description = "Short justification grounded in the headline.")
 recommended_action: Literal["hold", "reduce", "exit", "hedge"] = Field(
 description = "What to do with the position."
)
 confidence: float = Field(
 ge = 0.0, le = 1.0, description = "Confidence in this assessment, from 0 to 1."
)

prompt = ChatPromptTemplate.from_messages(
 [
 ("system", "You are a trading risk assistant."),
 (
 "human",
 "Headline: {headline}\n"
 "Position: {position}\n"
 "Time horizon: {horizon}\n"
 "Assess the risk and recommend an action."
),
],
)
llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

Controlled generation: enforce the schema at generation time, then parse into
RiskAssessment.
- strict=True: fail if the model output does not conform to the schema.
- include_raw=True: return both the parsed object and the raw model message.
structured_llm = llm.with_structured_output(
 RiskAssessment,
 method = "json_schema",
 strict = True,
 include_raw = True
)

chain = prompt | structured_llm
out = chain.invoke(
 {
 "headline": "Regulator opens probe into major chip supplier accounting
 practices",
 "position": "Long 2% portfolio weight",
 "horizon": "2 weeks"
 }
)

raw = out["raw"] # The original model message
parsed: RiskAssessment | None = out["parsed"]
parsing_error: Exception | None = out["parsing_error"]

print("---- RAW MODEL MESSAGE ----")
print(raw)

if parsing_error:
 raise parsing_error

print("---- PARSED OBJECT ----")
print(parsed.model_dump())

```

ch\_07\scr\with\_structured\_output.py

```
(.venv) C:\Users\alexc\llm-app\ch_07> python -m src.with_structured_output
---- RAW MODEL MESSAGE ----
content='{"risk_level": "high", "reasoning": "The opening of a probe into accounting practices can lead to significant volatility and uncertainty regarding the company\\\'s financial health and future operations. This could negatively impact stock prices in the short term, especially given the current market sensitivity to regulatory issues.", "recommended_action": "exit", "confidence": 0.8}'
additional_kwarg...: RiskAssessment(risk_level='high', reasoning="The opening of a probe into accounting practices can lead to significant volatility and uncertainty regarding the company's financial health and future operations. This could negatively impact stock prices in the short term, especially given the current market sensitivity to regulatory issues.", recommended_action='exit', confidence=0.8), 'refusal': None}
response_metadata={'token_usage': {'completion_tokens': 65, 'prompt_tokens': 191, 'total_tokens': 256}, 'completion_tokens_details': {'accepted_prediction_tokens': 0, 'audio_tokens': 0, 'reasoning_tokens': 0, 'rejected_prediction_tokens': 0}, 'prompt_tokens_details': {'audio_tokens': 0, 'cached_tokens': 0}, 'model_provider': 'openai', 'model_name': 'gpt-4o-mini-2024-07-18', 'system_fingerprint': 'fp_c4585b5b9c', 'id': 'chatcmpl-CugGwvfZ7IIvrXM4U1S0godKvWqpo', 'service_tier': 'default', 'finish_reason': 'stop', 'logprobs': None} id='lc_run--6b449cd7-fa6a-4da8-8ab6-c70fdd4e89a6-0' usage_metadata={'input_tokens': 191, 'output_tokens': 65, 'total_tokens': 256, 'input_token_details': {'audio': 0, 'cache_read': 0}, 'output_token_details': {'audio': 0, 'reasoning': 0}}
---- PARSED OBJECT ----
{'risk_level': 'high', 'reasoning': "The opening of a probe into accounting practices can lead to significant volatility and uncertainty regarding the company's financial health and future operations. This could negatively impact stock prices in the short term, especially given the current market sensitivity to regulatory issues.", 'recommended_action': 'exit', 'confidence': 0.8}
```

#### 7.4.9 Structured outputs as the glue in multi-step LCEL pipelines

Structured output becomes most valuable when step N feeds step N+1. In an e-commerce support flow, one step might extract the customer's intent and required fields, then the next step writes the customer-facing message and the internal ticket update. The extraction step should not emit prose. It should emit a stable data shape. This example uses `JsonOutputParser` for intent extraction and then uses that dict as input to the next prompt. The LCEL pipeline keeps the boundary explicit: the first stage produces data, the second consumes it.

```
from __future__ import annotations
from .config import OPENAI_API_KEY
import os
from langchain_openai import ChatOpenAI
from langchain_core.output_parsers import JsonOutputParser, StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.runnables import RunnablePassthrough

llm = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0)

extract_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You extract structured fields from customer messages. "
 "Return ONLY valid JSON with keys: intent, order_id, urgency, missing_info."
),
 ("human", "{message}"),
]
)

extract_chain = extract_prompt | llm | JsonOutputParser()
reply_prompt = ChatPromptTemplate.from_messages(
```

```

 [
 (
 "system",
 "You are a customer support assistant."
 "Write a concise reply. Use the extracted fields; do not invent an order id."
),
 (
 "human",
 "Customer message: {message}\n"
 "Extracted fields (JSON): {extracted}\n"
 "Write the reply."
),
],
)
}

reply_chain = reply_prompt | llm | StrOutputParser()
pipeline = RunnablePassthrough.assign(extracted = extract_chain) | reply_chain
reply = pipeline.invoke(
 {
 "message": "Hi, my order A-10499 still says 'processing' after 10 days. Can you check?"
 }
)
print(reply)

```

ch\_07\scr\multi\_steps\_pipeline.py

The important detail is not the parser. The important detail is that the boundary between steps is a concrete interface (here, a JSON dict). That makes downstream behaviour easier to test, and it keeps failure local: if extraction fails, you know which step failed and what structure was expected.

#### 7.4.10 Schema validation outside the model call

Even when you use structured output, you still need to validate at the application boundary. This matters if you persist model outputs, exchange them between services, or accept structured output from a less-trusted model. Pydantic provides `model_validate_json`, which parses and validates a JSON string into a `BaseModel` instance in one step.

```

from __future__ import annotations
from pydantic import BaseModel, Field, ValidationError

class Resume(BaseModel):
 first_name: str = Field(...)
 last_name: str = Field(...)
 email: str | None = None

raw_json = '{"first_name": "Ada", "last_name": "Lovelace", "email": "ada@example.com"}'

try:
 resume = Resume.model_validate_json(raw_json)
except ValidationError as e:
 raise RuntimeError(f"Invalid structured output: {e}") from e

print(resume)

```

ch\_07\scr\model\_validate\_json.py

#### 7.4.11 Practical selection guidance

Use `with_structured_output` (or, more generally, provider-native structured output) when your chosen model integration supports it and the schema is central to correctness. LangChain documentation recommends using provider-native structured output when available; when it is not available, it describes alternatives such as tool-calling/function-calling patterns or parsing the model's text output.

Use output parsers when you need a specific target format, when you are working with a model that does not support native structured output, or when you want post-processing behaviour such as parser-driven error repair or retries. The key discipline is consistent: keep the interface between steps explicit and typed so that multi-step pipelines fail early and locally, rather than later and mysteriously.

## 7.5 FALLBACKS, RETRIES, AND ERROR HANDLING

LLM calls are probabilistic, but your production system should behave predictably under stress. When something goes wrong, aim for a controlled, bounded response (retry, fallback, repair, or fail fast) instead of exceptions bubbling up from deep inside a chain. LCEL's composition model is a good thing here: you attach resilience exactly where failure can happen (remote calls, brittle parsing, strict validation) and keep the higher-level chain focused on business flow. That also matches a basic reality of LLM systems: a model's working context is limited to the input you include in the current request (its context window), and many failures come from external services you don't control (model APIs, vector stores, tool endpoints).

A practical way to think about error handling is as three connected concerns:

- Detection: define what counts as failure, and make it raise at the boundary.
- Handling: apply a local strategy (retry, fallback, output repair).
- Recovery: return a stable result (possibly degraded) or escalate with a clear, intentional error.

In this section we focus on the handling layer, implemented directly on runnables (so it stays local, explicit, and testable).

### 7.5.1 Failure detection: make errors explicit at boundaries

In LCEL, `with_retry` retries a runnable when it raises an exception, and `with_fallbacks` switches to fallback runnables when the original runnable raises a handled exception—so first you decide what should raise. For remote calls, “failure” is typically already an exception (timeouts, rate limits, connection errors). For model outputs, failure is often a contract mismatch: missing fields, wrong types, invalid JSON, or anything your parser/validator refuses to accept. Treat that mismatch as a first-class boundary: raise immediately when the output violates the contract, so downstream steps do not process corrupted state.

This is what makes resilience testable: you can force deterministic failures and verify that only the intended segment retries or falls back, rather than re-running an entire workflow.

### 7.5.2 Retries with `with_retry` (transient problems)

Transient failures such as network glitches or sporadic API errors are often resolved by retrying the same request a small number of times. Instead of wrapping model calls manually, you can use `with_retry`, `stop_after_attempt` and `retry_if_exception_type` on any runnable. For example, a support answerer chain can be made more resilient as follows:

```
from langchain_core.runnables import RunnableConfig
resilient_segment = risky_segment.with_retry(
 stop_after_attempt = 3,
 retry_if_exception_type = (TimeoutError, ConnectionError)
)
result = resilient_segment.invoke(
 inputs,
 config = {"tags": ["support", "retry"]}
)
```

Let's look at the available properties:

- `retry_if_exception_type`: Which exception types should trigger a retry. Anything else will be raised immediately (no retry).

```
retry_if_exception_type = (TimeoutError, ConnectionError)
```

- `stop_after_attempt`: The maximum number of attempts to make before giving up. After this many attempts, the last exception is raised.

```
stop_after_attempt = 4
```

- `wait_exponential_jitter`: Whether to wait between retries using exponential backoff with jitter (randomness) to avoid “thundering herd” retry storms.

```
wait_exponential_jitter = True
```

- `exponential_jitter_params`: Fine-tuning for the exponential backoff + jitter timing.

```
exponential_jitter_params = {
 "initial": 0.5, # first wait starts around 0.5s (before growth/jitter)
 "max": 8.0, # never wait more than ~8s between attempts
 "exp_base": 2.0, # exponential growth base (2.0 => 0.5, 1, 2, 4, 8...)
 "jitter": 0.25 # add randomness so retries don't synchronize
}
```

Guidance for using retries safely:

1. Keep the retry scope small (wrap the one call that’s flaky, not the entire chain).
2. Be careful with side effects. If a step triggers an external action (charge card, create ticket), make that action idempotent before you retry.

### 7.5.3 Fallbacks with `with_fallbacks`

Retries are not enough when a failure persists, or when you treat certain outputs as failures (for example by raising on a parser or validation error). In these cases, you may want to try a different model or a simplified chain. LCEL supports this with `with_fallbacks`, which accepts a list of alternative runnables to try if the main one fails. For instance, a trading assistant chain might use a more capable model by default and fall back to a smaller, cheaper one if necessary:

```
from langchain_openai import ChatOpenAI

primary_llm = ChatOpenAI(model = "gpt-4.1-mini", temperature = 0.0)
fallback_llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0.0)

primary_chain = prompt | primary_llm | StrOutputParser()
fallback_chain = prompt | fallback_llm | StrOutputParser()

robust_chain = primary_chain.with_fallbacks([fallback_chain])

analysis = robust_chain.invoke({"question": "Assess the concentration risk of this portfolio."})
```

If the primary chain raises an exception (for example an API error, or a parser/validation exception), the runnable will automatically try the fallbacks in order until one succeeds, or all fail. This pattern is particularly useful in production, where you want graceful degradation rather than hard failures.

### 7.5.4 Correcting parsers and output-fixing

A common production failure mode is “the LLM call succeeded, but the workflow still fails”, because the text you got back is not in the format the next step needs. The fix is to treat parsing as a hard boundary. A parser is where natural language stops being “output” and becomes “data”: if the output matches the contract, you let it through; if it doesn’t, you raise immediately so downstream steps never consume corrupted state. LangChain’s `PydanticOutputParser` (and other parsers like `XMLLoaderParser`,

`JsonOutputParser`, etc.) is a straightforward way to define that contract when you want a typed object. You bind it to a Pydantic model (your schema). On success it returns a validated Pydantic instance; on failure it raises a parsing/validation error. It also provides `get_format_instructions()` so you can tell the model exactly what shape to produce.

Even with good prompting, outputs can drift: extra prose around the JSON, single quotes, wrong field types, missing required keys. When the output is “almost correct” but fails strict validation, `OutputFixingParser` is a recovery wrapper: it wraps a base parser and, if parsing fails, uses a `retry_chain` (typically LLM-powered) to generate a corrected completion and retries parsing. It will retry parsing up to `max_retries` times. The key idea is that fixing does not loosen the schema. The schema remains the base parser; the wrapper simply gives the system a controlled second chance to express the same information in a compliant structure. In many applications you can treat the repair mechanism as an implementation detail, as long as you understand the retry budget (`max_retries`) and which model/runnable is used for the `retry_chain`. Treat `OutputFixingParser` as a wrapper with one job: keep the base parser strict but give the system a bounded second chance to express the same information in a schema-compliant way. You control that behaviour by choosing which model performs the repair and how many attempts you allow. Downstream code still depends only on the base parser’s contract.

```
from __future__ import annotations
from .config import OPENAI_API_KEY
import os
import json
import re
from typing import Any, Dict
from pydantic import BaseModel, Field
from langchain_core.output_parsers import PydanticOutputParser
from langchain_core.runnables import RunnableLambda
from langchain_classic.output_parsers.fix import OutputFixingParser

class RiskAssessment(BaseModel):
 risk: str = Field(description = "LOW, MEDIUM, or HIGH")
 score: int = Field(description = "Integer risk score")

base_parser = PydanticOutputParser(pydantic_object = RiskAssessment)

def build_prompt(payload: Dict[str, Any]) -> str:
 # Put format instructions into the prompt so the model knows the required shape.
 format_instructions = base_parser.get_format_instructions()
 question = payload["question"]
 return (
 "Return ONLY a response that matches the format instructions.\n\n"
 f"Question: {question}\n\n"
 f"Format instructions:{format_instructions}"
)

def fake_llm(prompt: str) -> str:
 # Simulated model output that's "almost" right but invalid for the parser:
 # - adds prose
 # - uses single quotes (Python dict style)
 # - score is a string
 return "Sure! Here's the JSON:\n{'risk': 'HIGH', 'score': '7'}"

def deterministic_repair(payload: Dict[str, Any]) -> str:
 # OutputFixingParser will call this with a dict that contains:
 # - instructions: format instructions
 # - completion: the failing completion
 # - error: the parser error message
 completion = payload.get("completion", "") or ""
 # Strip everything before the first { ... } block.
 m = re.search(r"\{.*\}", completion, flags = re.DOTALL)
 blob = m.group(0) if m else completion
 # Convert python-ish dict text to JSON text.
 blob = blob.replace("'", '"')
```

```

Ensure numeric types for this demo.
blob = blob.replace('"score": "7"', '"score": 7')

Validate JSON and re-emits a clean JSON string.
obj = json.loads(blob)
return json.dumps(obj)

prompt_step = RunnableLambda(build_prompt)
llm_step = RunnableLambda(fake_llm)

fixing_parser = OutputFixingParser(
 parser = base_parser,
 retry_chain = RunnableLambda(deterministic_repair),
 max_retries = 1
)

chain = prompt_step | llm_step | fixing_parser
result = chain.invoke({"question": "Assess portfolio concentration risk."})
print(result)
print(type(result))

```

ch\_07\scr\OutputFixingParser.py

```

(.venv) C:\Users\yourname\llm-app> python -m src.OutputFixingParser
risk='HIGH' score=7
<class '__main__.RiskAssessment'>

```

The chain deliberately produces an output that is close to valid but fails Pydantic parsing (extra prose, single quotes, and a numeric field encoded as a string). `OutputFixingParser` wraps the strict `PydanticOutputParser` and intercepts that failure. When parsing raises an error, it calls a repair “`retry_chain`” with a payload that includes the original format instructions, the failing completion, and the parser’s error message. The deterministic repair function then extracts the object-like block, normalizes it into real JSON (double quotes, correct numeric type), validates it with `json.loads`, and returns clean JSON text. `OutputFixingParser` retries parsing once, yielding a proper `RiskAssessment` instance.

#### 7.5.4.1 *RetryOutputParser and RetryWithErrorOutputParser*

Sometimes “fix what we got back” is the wrong mental model. If parsing failed, the thing you actually need is a better completion, not a more forgiving parser. This is where retry wrappers help: they turn a parser failure into an explicit regeneration step (typically a single additional attempt unless you wrap it in your own loop). `RetryOutputParser` retries by showing the model the original prompt value (the rendered prompt used for the original call) and the failing completion, then asking for a corrected completion that the base parser can accept.

`RetryWithErrorOutputParser` follows the same pattern but also includes the parsing error in the retry prompt. That error is the practical hint: it tells the model what rule it violated (missing field, wrong type, invalid JSON, extra text around the object), so the retry is targeted rather than hopeful. Conceptually, the retry step is given three inputs: the original prompt that defined the contract, the completion that failed, and the error raised by the parser. The wrapper then re-runs the base parser on the corrected completion. The architectural point remains the same: the schema stays strict; you are simply adding a controlled second chance to express the same information in a compliant structure.

```

from __future__ import annotations
from .config import OPENAI_API_KEY
import os
from pydantic import BaseModel, Field

```

```

from langchain_openai import ChatOpenAI
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.output_parsers import PydanticOutputParser
from langchain_classic.output_parsers.retry import RetryOutputParser,
RetryWithErrorOutputParser

if not OPENAI_API_KEY:
 raise RuntimeError("OPENAI_API_KEY is not set. Check your environment or .env file.")

class RiskAssessment(BaseModel):
 risk: str = Field(description = "LOW, MEDIUM, or HIGH")
 score: int = Field(description = "Integer risk score")

base_parser = PydanticOutputParser(pydantic_object = RiskAssessment)

The prompt carries the contract (format instructions) that the retry parser needs.
prompt = ChatPromptTemplate.from_template(
 "Return ONLY a response that matches the format instructions.\n\n"
 "Question: {question}\n\n"
 "Format instructions:{format_instructions}"
).partial(format_instructions = base_parser.get_format_instructions())

llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0.0)

In a real workflow, `completion` is the raw model text you got back from
`llm.invoke(...)`. Here we simulate a typical "almost right" failure: extra prose +
single quotes + wrong type.
prompt_value = prompt.invoke({"question": "Assess portfolio concentration risk."})
bad_completion = "Sure! Here's the JSON:\n{'risk': 'HIGH', 'score': '7'}"

First, show what a strict boundary looks like.
try:
 base_parser.parse(bad_completion)
except Exception as e:
 print("Base parse failed:", type(e).__name__, str(e))

Option 1: RetryOutputParser (re-prompts using the original prompt + failing completion).
retry_parser = RetryOutputParser.from_llm(parser = base_parser, llm = llm)
fixed_obj_1 = retry_parser.parse_with_prompt(bad_completion, prompt_value)
print("RetryOutputParser result:", fixed_obj_1, type(fixed_obj_1))

Option 2: RetryWithErrorOutputParser (also includes the parsing error in the retry
prompt).
retry_with_error = RetryWithErrorOutputParser.from_llm(parser = base_parser, llm = llm)
fixed_obj_2 = retry_with_error.parse_with_prompt(bad_completion, prompt_value)
print("RetryWithErrorOutputParser result:", fixed_obj_2, type(fixed_obj_2))

```

[ch\\_07\scr\RetryOutputParser\\_RetryWithErrorOutputParser.py](#)

- Use OutputFixingParser when you are already running an LLM step inside an LCEL chain and you want the “parse boundary” to be self-healing as part of the pipeline. The mental model is “parse, and if it fails, do an in-chain repair and re-parse, bounded by max\_retries”. This is a common choice when you want the parsing boundary to live inside the LCEL pipeline, so downstream steps either receive the parsed object, or the chain fails with a parsing error. It also centralises repair behaviour (same repair model, same retry limit) when you want consistent handling across multiple chains.
- Use RetryOutputParser when you have the model completion (text) and the corresponding prompt value, and you want to retry by asking an LLM to produce a corrected completion. This is a good fit when parsing is not “embedded” as a runnable step yet (for example you are integrating a legacy call site, or you are doing manual orchestration), or when you want to make the retry behaviour extremely explicit at the call site. The key characteristic is that it retries by passing the prompt (as a prompt value / formatted prompt) and the failing completion to another LLM.
- Use RetryWithErrorOutputParser when you want the retry to be guided by the precise reason parsing failed. This is the choice when failures are frequent or subtle (missing required keys, wrong types, invalid JSON, stray commentary) and you want the retry prompt to include the parser exception message, so the model is told exactly what

rule it violated. It can be more effective than retry-without-error when the parser's exception message is specific, because the retry prompt includes the error details instead of only a generic 'try again' instruction.

A practical heuristic: start with `OutputFixingParser` if you are building an LCEL pipeline and want a single "strict but resilient" boundary; reach for `RetryOutputParser`/`RetryWithErrorOutputParser` when you are outside the chain (you already have completion + `prompt_value` in hand) or when you want fine-grained control over regeneration and logging at the call site. Between the two retry wrappers, prefer `RetryWithErrorOutputParser` for typed outputs where the parse error message is informative; keep `RetryOutputParser` for simpler "shape" issues where the error text is noisy or you don't want to surface error details into the retry prompt.

## 7.6 CALLBACKS

Callbacks are LangChain's built-in instrumentation surface: a standard set of lifecycle "hooks" that the runtime fires while a runnable is executing. Instead of sprinkling print statements through prompts, retrievers, model wrappers, tools, and parsers, you attach one or more callback handlers and let the runtime report what is happening: when a chain starts, when a retriever returns documents, when a tool runs, when an LLM starts generating, when streaming tokens arrive, and when the whole operation ends or fails. This matters because modern LangChain systems are rarely "one prompt → one answer". Even a small support assistant tends to be a short pipeline: format messages, retrieve context, call a model, parse the output, maybe call a tool, then call the model again. Without a uniform event stream, you end up debugging blind: you see the final output, but you do not see which steps ran, how long each took, or what inputs were passed between them.

Callbacks are the mechanism. You implement (or reuse) a callback handler that receives events such as `on_chain_start` / `on_chain_end`, `on_retriever_start` / `on_retriever_end`, `on_tool_start` / `on_tool_end`, `on_llm_start` / `on_llm_end`, and token-level `on_llm_new_token` during streaming. The callback manager is what calls your handler methods at the right moments.

The practical rule is simple: treat callbacks as an infrastructure concern. You wire them at the boundary (via runnable configuration) so every sub-step inherits the same observability settings, rather than trying to instrument each component manually. That keeps your chains readable, your logs consistent, and your debugging story sane.

### 7.6.1 The run tree model

Every callback event in LangChain belongs to a "run". A run is one execution of one component: one chain invocation, one retriever query, one tool call, one model generation. To make those runs composable and traceable, LangChain tags each run with two identifiers:

- `run_id` is the unique ID for the current run.
- `parent_run_id` is the ID of the run that triggered it (or `None` if it is the root).

These IDs appear on the standard callback methods (for chains, models, retrievers, tools, and others), so handlers can stitch events together consistently.

When you compose a pipeline, nested calls naturally build a run tree. The root run is "the thing you invoked", and everything it triggers becomes a child run (and may itself spawn children). A typical support assistant flow looks like this:

- Root: chain (`run_id=A`)
- Child: retriever (`run_id=B, parent_run_id=A`)
- Child: chat model call (`run_id=C, parent_run_id=A`)
- Child: tool call (`run_id=D, parent_run_id=C`)
- Child: chat model call (`run_id=E, parent_run_id=C`)

The run tree is the difference between "I got a slow answer" and "the retriever was fast but the second model call was slow because the tool returned a large payload". Once you have a tree, you can attribute work to the node that caused it:

- Latency attribution: each run has a clear start/end boundary (via start/end callbacks), so a tracer or handler can measure elapsed time per node and then explain total time as “sum of critical path across these child runs”. The parent/child linkage is what makes that breakdown trustworthy.
- Cost attribution: many systems record token usage or provider billing metadata at the model-run level (when available) rather than at the chain level. The run tree tells you which specific model invocation consumed tokens, and which upstream step (retrieval/tool) caused the prompt to grow. (Do not assume token/cost fields always exist; treat them as provider- and tracer-dependent data, but the run structure is stable.)
- Debuggability: when something fails, the error callback lands on a specific run\_id. With parent\_run\_id you can immediately answer “what was the caller?” and reconstruct the context without guessing.

The practical takeaway: the run tree is your system’s causal map. Without it, logs are just a pile of lines; with it, you can explain behavior, performance, and spend in terms of the actual pipeline that executed.

### 7.6.2 Where callbacks live in the modern API

In LangChain 1+, callbacks live in the same place as every other execution concern: the Runnable config you pass at run time. Every runnable entry point (`invoke` / `ainvoke`, `batch` / `abatch`, `stream` / `astream`, and the debug streaming helpers) accepts an optional config argument (`RunnableConfig`). That config is not “just for this one object”. It is explicitly defined as applying to “this call and any sub-calls” (for example, a chain calling an LLM).

When you build an LCEL pipeline (`prompt` | `model` | `parser`, or a bigger sequence that also calls a retriever and tools), and you call `invoke(..., config=...)`, the runtime passes the callback settings down the run tree automatically: the root chain run and every nested retriever/model/tool run will see the same callbacks, tags, and metadata. There are 2 supported patterns to attach callbacks:

1. Per-call (recommended for most applications): You pass a `RunnableConfig` at the boundary where your app enters LangChain:

```
result = chain.invoke(
 {"question": question},
 config = {
 "callbacks": [handler],
 "tags": ["support", "prod"],
 "metadata": {"tenant_id": tenant_id, "ticket_id": ticket_id}
 }
)
```

In this case callbacks: apply to this call and any sub-calls (so your handler sees the whole run tree). Tags are passed to all callbacks; metadata: passed to start callbacks (the `handle*_Start` / `on_*_start` family), which is exactly where tracers/loggers typically capture inputs and identifiers.

2. Bound to a runnable (useful when you want “this component always carries these tags/handlers”): You can bind config directly onto a runnable using `with_config`, which returns a new runnable that carries that config whenever it runs. Conceptually:

```
instrumented_chain = chain.with_config(
```

```

 {"tags": ["checkout-flow"], "callbacks": [handler]}
 }
 result = instrumented_chain.invoke({"question": question})

```

This is handy when you want to stamp a particular sub-pipeline with a stable identity (for example “retrieval-v2”), while still letting the outer call attach request-level metadata (ticket\_id, user\_id, etc.).

### 7.6.3 Callback handlers: the hook surface you implement

In practice, “using callbacks” means “attaching one or more callback handlers”. A handler is just an object that implements a standard set of hook methods (`on_chain_start`, `on_llm_start`, `on_tool_end`, ...). The LangChain runtime calls these hooks at the right moments as it executes your runnable graph. The base contract you build on is `BaseCallbackHandler`. `BaseCallbackHandler` is intentionally broad. It bundles hook families for chains, retrievers, tools, agents, and models into one surface, so a single handler can observe the whole run end-to-end. You subclass it and override only the hooks you care about; everything else can remain as the default no-op.

What actually happens at runtime is:

1. You pass one or more handlers to the run (either directly, or via the runnable’s config). A “callback manager” holds that list.
2. When a step starts (chain / retriever / tool / model), LangChain calls the matching start hook on every handler (for example `on_chain_start` or `on_llm_start`).
3. While the step is running, LangChain may emit extra hooks. The big one is `on_llm_new_token`, which fires only when you stream tokens.
4. When the step ends, LangChain calls either the end hook (success) or the error hook (failure).

There is also a set of `ignore_*` flags (`ignore_llm`, `ignore_chain`, `ignore_retriever`, ...) that allow a handler to opt out of whole event families when you only care about a subset.

There are four common handler “roles”:

- Logging / tracing handlers: These handlers turn callback events into structured logs or traces (spans/runs) so you can inspect “what happened” after the fact. A tracer is just a callback handler that records a run tree plus inputs/outputs/timings into a trace backend.
- Metering handlers (timings / token counts where available): These handlers compute numbers: latency per step, counts of tool invocations, and—when the model/provider returns it—usage metadata such as token usage. LangChain even includes a dedicated `UsageMetadataCallbackHandler` in the callback module, which is a strong hint at the intended pattern: treat “usage/cost accounting” as a callback concern, not application logic. Usually used for: dashboards, SLOs, cost controls. Output is counters and histograms, not text traces.
- UI / stream handlers (live tokens and progress): These handlers are for “show me what’s happening right now”: live token streaming (`on_llm_new_token`) and human-readable progress text (`on_text`). They power chat UIs, CLIs, and “agent is thinking / tool is running” indicators. Remember: `on_llm_new_token` is only emitted when streaming is enabled, so a UI handler should degrade gracefully when running in non-streaming mode. Usually used for: real-time user experience.

- Testing handlers (assertions about behavior): These handlers treat callbacks as a test oracle: instead of asserting only on final output, you assert on behavior. For examples:
  - “the retriever must be called exactly once”
  - “no tools are allowed in read-only mode”
  - “the chain must not call the model when cache hits”

Used for making behavior testable without mocking internals.

The unifying idea is simple: callbacks turn execution into observable events. Handlers decide what those events mean for your system—trace, measure, stream, or assert—without contaminating the chain’s business logic.

#### 7.6.4 Example: “Tell me what this chain actually did”

This example builds a tiny e-commerce support chain (prompt → retriever → chat model → parser) and attaches a callback handler that prints a run tree with timings. The retriever and the model are deliberately “toy” implementations so you can run the example without external services, but the callback wiring is the same for production chains.

```
from __future__ import annotations
import re
import time
from dataclasses import dataclass
from typing import Any, Dict, List, Optional, Tuple
from uuid import UUID
from langchain_core.callbacks import BaseCallbackHandler
from langchain_core.documents import Document
from langchain_core.messages import BaseMessage
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.prompt_values import PromptValue
from langchain_core.retrievers import BaseRetriever
from langchain_core.runnables import RunnableLambda, RunnablePassthrough
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY

Token totals (works for OpenAI via LangChain callbacks)
try:
 from langchain_community.callbacks.manager import get_openai_callback
except Exception:
 get_openai_callback = None

1) Readable chain-only timing logger (with input/output)

@dataclass
class _Run:
 name: str
 start: float
 parent: Optional[UUID]
 tags: Optional[list]
 metadata: Optional[dict]

class PrettyStepTimings(BaseCallbackHandler):
 # Only chain events; prints a readable tree with timings, linkage, and input/output.
 ignore_llm = True
 ignore_chat_model = True
 ignore_retriever = True
 ignore_tool = True
 ignore_agent = True

 def __init__(self, max_text: int = 260) -> None:
 super().__init__()
 self._runs: Dict[UUID, _Run] = {}
 self._max_text = max_text

 def _name(self, serialized: Any) -> str:
 if isinstance(serialized, dict):
 return serialized.get("name") or serialized.get("id") or "chain"
```

```

 return "chain"

def _depth(self, run_id: UUID) -> int:
 d = 0
 cur = self._runs.get(run_id)
 while cur and cur.parent:
 d += 1
 cur = self._runs.get(cur.parent)
 return d

def _indent(self, run_id: UUID) -> str:
 return " " * self._depth(run_id)

def _short_id(self, u: Optional[UUID]) -> str:
 if not u:
 return "-"
 return str(u).split("-")[0]

def _fmt_tags(self, tags: Optional[list]) -> str:
 if not tags:
 return "-"
 return ", ".join(str(t) for t in tags)

def _fmt_meta(self, meta: Optional[dict]) -> str:
 if not meta:
 return "-"
 parts = [f"{k}={v}" for k, v in sorted(meta.items(), key = lambda x: str(x[0]))]
 return ", ".join(parts)

def _sanitize_text(self, s: str) -> str:
 s = s.replace("\n", " ").strip()
 s = re.sub(r"\b[\w\.-]+@[\\w\\.-]+\.\w+\b", "<email>", s)
 s = re.sub(r"\b\d{6,}\b", "<id>", s)
 if len(s) > self._max_text:
 s = s[: self._max_text - 3] + "..."
 return s

def _preview(self, obj: Any) -> str:
 if isinstance(obj, str):
 return self._sanitize_text(obj)

 if isinstance(obj, BaseMessage):
 return f"{obj.type}: {self._sanitize_text(str(obj.content))}"

 if isinstance(obj, PromptValue):
 try:
 msgs = obj.to_messages()
 if msgs:
 last = msgs[-1]
 return f"prompt(last={last.type}): {self._sanitize_text(str(last.content))}"
 except Exception:
 pass
 return f"prompt: {self._sanitize_text(str(obj))}"

 if isinstance(obj, list):
 if not obj:
 return "[]"
 head = self._preview(obj[0])
 return f"list(len = {len(obj)}) first = {head})"

 if isinstance(obj, dict):
 keys = list(obj.keys())
 interesting = []
 for k in ("input", "question", "query", "context", "text", "output"):
 if k in obj:
 interesting.append(f"{k}={self._preview(obj[k])}")
 if interesting:
 return f"dict(keys = {keys}) " + "; ".join(interesting)
 return f"dict(keys = {keys})"

 return f"{type(obj).__name__}: {self._sanitize_text(str(obj))}"

def on_chain_start(self, serialized, inputs, *, run_id: UUID, parent_run_id: Optional[UUID] = None, **kwargs):
 name = self._name(serialized)

 tags = kwargs.get("tags")

```

```

metadata = kwargs.get("metadata")

self._runs[run_id] = _Run(
 name = name,
 start = time.perf_counter(),
 parent = parent_run_id,
 tags = tags if isinstance(tags, list) else None,
 metadata = metadata if isinstance(metadata, dict) else None
)

ind = self._indent(run_id)
print(f"{ind}▶ {name}")
print(f"{ind} run={self._short_id(run_id)} parent={self._short_id(parent_run_id)}")
print(f"{ind} tags={self._fmt_tags(self._runs[run_id].tags)}")
print(f"{ind} meta={self._fmt_meta(self._runs[run_id].metadata)}")
print(f"{ind} IN : {self._preview(inputs)}")

def on_chain_end(self, outputs, *, run_id: UUID, parent_run_id: Optional[UUID] = None,
 **kwargs):
 r = self._runs.get(run_id)
 if not r:
 print(f"✓ <unknown> (run={self._short_id(run_id)})")
 return

 ms = (time.perf_counter() - r.start) * 1000.0
 ind = self._indent(run_id)
 print(f"{ind} OUT: {self._preview(outputs)}")
 print(f"{ind}✓ {r.name} ({ms:.1f} ms)\n")

def on_chain_error(self, error, *, run_id: UUID, parent_run_id: Optional[UUID] = None,
 **kwargs):
 r = self._runs.get(run_id)
 name = r.name if r else "<unknown>"
 start = r.start if r else time.perf_counter()
 ms = (time.perf_counter() - start) * 1000.0
 ind = self._indent(run_id)
 print(f"{ind}✗ {name} ({ms:.1f} ms)")
 print(f"{ind} run={self._short_id(run_id)} parent={self._short_id(parent_run_id)}")
 print(f"{ind} error={repr(error)}\n")

2) Minimal retriever (in-memory policy docs)

class PolicyRetriever(BaseRetriever):
 docs: List[Document]
 k: int = 1

 def _get_relevant_documents(self, query: str) -> List[Document]:
 q = query.lower()
 scored: List[Tuple[int, Document]] = []
 for d in self.docs:
 text = (d.page_content or "").lower()
 score = 0
 for kw in ("return", "refund", "exchange", "delivery", "shipping", "wore",
 "unused"):
 if kw in q and kw in text:
 score += 1
 scored.append((score, d))

 scored.sort(key = lambda x: x[0], reverse = True)
 top = [d for s, d in scored if s > 0][: self.k]
 return top if top else self.docs[:1]

 def format_docs(docs: List[Document]) -> str:
 return "\n".join(f"- {d.metadata.get('title', 'Doc')}: {d.page_content}" for d in docs)

3) Build chain

def build_chain():
 docs = [
 Document(
 page_content = "Returns: items can be returned within 30 days of delivery if unused and in original packaging.",
 metadata = {"title": "Returns Policy"}
),
 Document(
 page_content = "Opened items can be returned only if defective. Refunds are"
)
]

```

```

 issued after inspection.",
 metadata = {"title": "Refund Policy"}
),
]

retriever = PolicyRetriever(docs = docs, k = 1)

prompt = ChatPromptTemplate.from_messages(
 [
 ("system", "You are a support agent. Answer using only the provided policy context."),
 ("human", "Question: {question}\n\nPolicy context:\n{context}")
]
)

llm = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

chain = (
 {
 "question": RunnablePassthrough(),
 "context": retriever | RunnableLambda(format_docs)
 }
 | prompt
 | llm
 | StrOutputParser()
)
return chain

4) Run

def main() -> None:
 chain = build_chain()
 logger = PrettyStepTimings(max_text = 260)

 question = "I bought shoes last week. Can I return them if I never wore them?"

 print("\n== RUN TRACE (chain-only) ==\n")

 cfg = {
 "callbacks": [logger],
 "tags": ["ch07", "ecommerce", "timings"],
 "metadata": {"tenant": "demo-shop", "example": "callable-chain"}
 }

 if get_openai_callback is None:
 answer = chain.invoke(question, config = cfg)
 cb = None
 else:
 with get_openai_callback() as cb:
 answer = chain.invoke(question, config = cfg)

 print("== FINAL ANSWER ==")
 print(answer)

 print("\n== TOKEN USAGE ==")
 if cb is None:
 print("Token callback not available (langchain_community.get_openai_callback import failed).")
 else:
 print(f"prompt_tokens : {getattr(cb, 'prompt_tokens', None)}")
 print(f"completion_tokens : {getattr(cb, 'completion_tokens', None)}")
 print(f"total_tokens : {getattr(cb, 'total_tokens', None)}")
 # total_cost may be present depending on version/provider accounting
 if hasattr(cb, "total_cost"):
 print(f"estimated_cost : {cb.total_cost}")

if __name__ == "__main__":
 main()

```

ch\_07\scr\ecommerce\_callable.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.ecommerce_callable
== RUN TRACE (chain-only) ==
▶ chain
run=019c08c4 parent=
```

```

tags=ch07, ecommerce, timings
meta=example=callable-chain, tenant=demo-shop
IN : I bought shoes last week. Can I return them if I never wore them?
▶ chain
 run=019c08c4 parent=019c08c4
 tags=seq:step:1, ch07, ecommerce, timings
 meta=example=callable-chain, tenant=demo-shop
 IN : I bought shoes last week. Can I return them if I never wore them?
 ▶ chain
 ▶ chain
 run=019c08c4 parent=019c08c4
 run=019c08c4 parent=019c08c4
 tags=map:key:question, ch07, ecommerce, timings
 tags=map:key:context, ch07, ecommerce, timings
 meta=example=callable-chain, tenant=demo-shop
 meta=example=callable-chain, tenant=demo-shop
 IN : I bought shoes last week. Can I return them if I never wore them?
 IN : I bought shoes last week. Can I return them if I never wore them?
 ▶ chain
 OUT: I bought shoes last week. Can I return them if I never wore them?
 run=019c08c4 parent=019c08c4
 ✓ chain (1.6 ms)
 tags=seq:step:2, ch07, ecommerce, timings

 meta=example=callable-chain, tenant=demo-shop
 IN : list(len=1 first=Document: page_content='Returns: items can be returned within 30 days of delivery if
 unused and in original packaging.' metadata={'title': 'Returns Policy'})
 OUT: - Returns Policy: Returns: items can be returned within 30 days of delivery if unused and in original
 packaging.
 ✓ chain (0.8 ms)

 OUT: - Returns Policy: Returns: items can be returned within 30 days of delivery if unused and in original
 packaging.
 ✓ chain (2.0 ms)

 OUT: dict(keys=['question', 'context']) question=I bought shoes last week. Can I return them if I never wore them?;
 context=- Returns Policy: Returns: items can be returned within 30 days of delivery if unused and in original
 packaging.
 ✓ chain (4.4 ms)

 ▶ ChatPromptTemplate
 run=019c08c4 parent=019c08c4
 tags=seq:step:2, ch07, ecommerce, timings
 meta=example=callable-chain, tenant=demo-shop
 IN : dict(keys=['question', 'context']) question=I bought shoes last week. Can I return them if I never wore them?;
 context=- Returns Policy: Returns: items can be returned within 30 days of delivery if unused and in original
 packaging.
 OUT: prompt(last=human): Question: I bought shoes last week. Can I return them if I never wore them? Policy
 - Returns Policy: Returns: items can be returned within 30 days of delivery if unused and in original
 packaging.
 ✓ ChatPromptTemplate (0.6 ms)

 ▶ chain
 run=019c08c4 parent=019c08c4
 tags=seq:step:4, ch07, ecommerce, timings
 meta=example=callable-chain, tenant=demo-shop
 IN : ai: Yes, you can return the shoes as long as they are unused and in their original packaging, within 30 days
 of delivery.
 OUT: Yes, you can return the shoes as long as they are unused and in their original packaging, within 30 days of
 delivery.
 ✓ chain (1.6 ms)

 OUT: Yes, you can return the shoes as long as they are unused and in their original packaging, within 30 days of
 delivery.
 ✓ chain (1459.8 ms)

 === FINAL ANSWER ===
 Yes, you can return the shoes as long as they are unused and in their original packaging, within 30 days of delivery.
 === TOKEN USAGE ===
 prompt_tokens : 69
 completion_tokens : 26
 total_tokens : 95
 estimated_cost : 2.594999999999994e-05

```

## 7.7 SUMMARY

This chapter introduces LCEL (LangChain Expression Language) as the modern way to build LangChain workflows by composing small parts into readable pipelines. The key idea is that prompts, models, retrievers, parsers, and even whole chains share the same `Runnable` interface, so you can connect them with the same execution API and treat the final pipeline as a single component. You learn the practical runtime behaviours that matter in real systems: single-call execution (`invoke/ainvoke`), batching (`batch/abatch`), token streaming (`stream/astream`), and two “debug streaming” modes that surface what happens inside a chain while it runs (`astream_log` and `astream_events`). The chapter shows how LCEL composition maps to common workflow shapes: straight sequences with the pipe operator, dictionary wiring that builds structured inputs, branching and routing with dedicated routing runnables, and parallel blocks that compute multiple results at once and merge them into a dictionary.

A recurring theme is keeping dataflow explicit. You see patterns for building and preserving dict-shaped payloads, forwarding selected keys, and adding derived fields without ad-hoc dict manipulation. `RunnablePassthrough`, `assign`, and `pick` are treated as “payload discipline” tools: they make it obvious what fields survive to the next step and where new fields are introduced. The chapter then moves from composition to production-readiness. `RunnableConfig` becomes the single place to attach tags, metadata, callbacks, and guardrails like `max_concurrency` and `recursion_limit`, either per-call or bound to a runnable with `with_config`. Reliability is handled locally with `with_retry`, `with_fallbacks`, and strict parsing boundaries, including repair and retry strategies when structured outputs fail validation.

Finally, callbacks are introduced as the system’s instrumentation layer: standard lifecycle hooks that emit a run tree for chains, retrievers, tools, and model calls. Instead of debugging from final output, you can trace what ran, in which order, with what inputs, and how long each step took.

## 8 LANGGRAPH FUNDAMENTALS AND STATE MANAGEMENT

---

LangGraph is the part of the LangChain ecosystem that treats an LLM application as an explicit workflow, not a single prompt. Instead of hiding control flow inside Python glue code, you model the system as a graph: nodes do work, edges decide what runs next, and a shared state object carries everything the workflow learns as it moves forward.

That design sounds academic until you ship something real. Customer support is not one step. An e-commerce assistant has to route between returns, billing, shipping, and account issues, sometimes in parallel, sometimes in loops, sometimes with human approval. A trading assistant needs guardrails, checkpoints, and the ability to pause, resume, or roll back. These aren't "nice-to-haves"; they are the difference between a demo and a system you can operate.

This chapter is about the core mechanics that make those systems reliable:

- State as the contract between steps. It is not just "the final answer." It's the working memory of the run: conversation messages, retrieved evidence, control flags, intermediate drafts, and diagnostics.
- Reducers as the rules of truth when updates collide. If two nodes write to the same key in the same step, you need deterministic merge semantics, or you get chaos disguised as concurrency.
- Runtime execution as discrete steps, with parallel work, fan-out/fan-in, and predictable visibility of writes (updates from a step become visible in the next step).
- Persistence via checkpoints (thread-scoped continuity) and stores (cross-thread memory), so your workflow can outlive a single request, survive restarts, and support "pause here, resume later" patterns.

You'll also see why these fundamentals directly enable two practical capabilities that show up everywhere later in the book:

1. Reasoning strategies as control flow. Self-consistency, reflection, verification, and tree-like exploration are repeatable graph shapes that create explicit artifacts and stop conditions.
2. Short-context strategies for long inputs. When documents, transcripts, or histories don't fit comfortably in one model call, you don't "hope harder." You split, map, reduce, and carry forward compact intermediate results—again, as explicit state.

By the end of this chapter, you should be able to look at any agent workflow and answer three operational questions with confidence: \*What does it remember? How does it decide what to do next? And what happens when something fails or needs to pause?

## 8.1 NODES, EDGES, STATE OBJECTS, REDUCERS, AND CONFIGURATION

LangGraph takes the idea of a “chain of steps” and turns it into a proper state machine. Instead of one linear pipeline, you describe your application as a graph of small nodes connected by edges, all operating on a shared state object that evolves over time. This is what gives you loops, conditional flows, parallel branches, and agents that can pause and resume when you enable persistence (checkpointers) and/or use interrupts; checkpoints are saved at every superstep and can be resumed via thread/checkpoint IDs. Under the hood, LangGraph’s runtime follows a Pregel / Bulk Synchronous Parallel (BSP) model: execution proceeds in discrete steps (supersteps) where the runtime plans which nodes run, executes them (potentially in parallel), then applies their writes as a synchronized update. Each node receives the current state, produces a state update, and sends that update along its outgoing edges. Execution progresses in discrete steps (supersteps): the runtime (1) plans which nodes to execute based on which state channels were updated, (2) runs those nodes in parallel, and (3) applies their writes to update the shared state. Writes made during a step are not visible to other nodes until the next step.

### 8.1.1 Nodes

A node is one unit of work in the graph. In code it is a function (sync or async) or a `Runnable`. Nodes always receive state and may also accept a `RunnableConfig` and a `Runtime` argument when you need run-scoped config or runtime services. The node does some computation and returns a partial update to that state as a dictionary. You register these functions in a `StateGraph` and give each one a name. The state type is declared up front. State can be a `TypedDict`, a `dataclass`, or a `Pydantic BaseModel`. (`TypedDict`/`dataclass` are common for speed; `Pydantic` adds validation but is typically less performant.) That schema defines the graph’s state channels (the keys/fields nodes read and write), and each key can have its own reducer that controls how updates are applied. Nodes are written against this schema. The graph applies each node’s returned `dict` as a state update. By default, updates overwrite a key’s value; if you annotate a key with a reducer, that reducer is used to combine concurrent updates to that key. Well-behaved nodes follow two simple rules:

1. Treat the incoming state as read-only for the duration of the call.
2. Return a dictionary with only the keys you want to update.

This keeps each node close to a pure function over state. That is what allows the runtime to run several nodes in parallel during a superstep and then combine their state updates in a predictable way. A typical RAG state might look like this in Python:

```
from typing import List
from typing_extensions import TypedDict

class RAGState(TypedDict, total = False):
 question: str
 documents: List[str]
 answer: str
```

A retrieval node reads question and writes documents:

```
def retrieve(state: RAGState) -> dict:
 question = state["question"]
 # In a real graph, call a retriever here.
 docs = [f"Doc about: {question}"]
 return {"documents": docs}
```

A generation node reads question and documents and writes answer:

```
def generate(state: RAGState) -> dict:
 question = state["question"]
 docs = state["documents"]
 answer = f"Answer to '{question}' using {len(docs)} document(s)."
 return {"answer": answer}
```

Nodes stay small and focused. Each node looks at a few keys, performs one step of the process, and emits only the updates it owns.

### 8.1.2 Edges

Edges describe what happens after a node finishes. They control both the order of execution and where parallelism appears. A normal edge connects one node to the next. For example, `add_edge("retrieve", "generate")` means “when retrieve completes, it sends the state along that edge; generate becomes active and will run in the next superstep. Entry into the graph is modelled by an edge from a virtual START node to the first logical step. In StateGraph, you typically model termination by routing to the virtual END node; more generally, execution terminates when all nodes are inactive and no messages are in transit (i.e., there is nothing left to run).

It’s possible to have conditional edges add routing logic. Instead of always going to a fixed next node, you attach a small routing function that inspects the current state and returns the next node name, END, or a list/sequence of node names (to fan out to multiple destinations). This is how you express “if the user is asking for a refund, go into the returns branch; otherwise stay in the general support branch.” The same mechanism can create loops: a routing function can decide to go back to an earlier node for another research step before moving on to a final answer.

If the decision point also needs to write state (for example, recording a handoff payload or setting a control flag), use Command instead of conditional edges so the node can update state and route in the same return value.

- Conditional edges are a routing mechanism: the routing function chooses the next node, but it does not itself carry a state update. When you need a decision point to both mutate state and jump to a specific next node, LangGraph supports returning a Command from a node.
- A Command can bundle a state update (`update=...`, merged using the reducer rules) and a control-flow target (`goto=...`). Command can also carry a `resume=...` value to resume an interrupt-driven (human-in-the-loop) pause. The recommendation is to use Command for cases like multi-agent handoffs, where the handoff decision and the handoff payload must be emitted together, and for human-in-the-loop flows built with `interrupt()`, where you resume execution by invoking the graph again with `Command(resume=...)`.

In Python, a node that returns Command must type-annotate the set of possible destination nodes (for example `Command[Literal[...]]`). This is required for graph rendering and to make the runtime aware of the node’s legal navigation targets.

```
def my_node(state: State) -> Command[Literal["my_other_node"]]:
 return Command(update = {"foo": "bar"}, goto = "my_other_node")
```

Edges also determine parallelism. If a node has several outgoing edges, all destination nodes are executed in parallel as part of the next superstep; downstream fan-in nodes run only after they receive inputs from their upstream nodes. That is how patterns like “three generators

then one aggregator” are expressed: one node fans out to three generator nodes, those run in parallel, and when they are all done, the aggregator becomes ready and runs in a later superstep.

The superstep model matters for state updates. During a superstep, multiple nodes may produce updates to the same state key. Updates produced by nodes in the same superstep are applied transactionally as part of that superstep; when multiple updates hit the same key, reducers determine how they are combined, and ordering across parallel updates is not guaranteed. Reducers (described below) tell the runtime how to combine them.

Here is a small example with routing and a loop:

```

from typing import Literal, List
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END

1. Define the State
class ResearchState(TypedDict):
 question: str
 step_counter: int
 needs_more_research: bool
 notes: List[str]
 answer: str

2. Define the Nodes
def research(state: ResearchState) -> dict:
 # Handle initial empty values
 current_step = state.get("step_counter", 0)
 current_notes = state.get("notes", [])

 step = current_step + 1
 new_note = f"note {step}: looked up something relevant"
 notes = current_notes + [new_note]

 # Logic: Stop after 2 steps
 needs_more = step < 2

 print(f"--- Researching (Step {step}) ---")
 return {
 "step_counter": step,
 "notes": notes,
 "needs_more_research": needs_more
 }

def write_answer(state: ResearchState) -> dict:
 print("--- Writing Final Answer ---")
 joined = "; ".join(state["notes"])
 return {"answer": f"Answer based on: {joined}"}

3. Define the Router
def route_after_research(state: ResearchState) -> Literal["research", "write_answer"]:
 if state["needs_more_research"]:
 return "research"
 return "write_answer"

4. Build the Graph
builder = StateGraph(ResearchState)

builder.add_node("research", research)
builder.add_node("write_answer", write_answer)

builder.add_edge(START, "research")
builder.add_conditional_edges(
 "research",
 route_after_research,
 {
 "research": "research",
 "write_answer": "write_answer"
 }
)
builder.add_edge("write_answer", END)

5. Compile and Run
graph = builder.compile()

if __name__ == "__main__":
 # Initial state

```

```

initial_input = {
 "question": "What is LangGraph?",
 "step_counter": 0,
 "notes": []
}

Execute the graph
final_state = graph.invoke(initial_input)

Print the output
print("\nFinal Results:")
print(f"Steps taken: {final_state['step_counter']}")
print(f"Notes: {final_state['notes']}")
print(f"Final Answer: {final_state['answer']}")

```

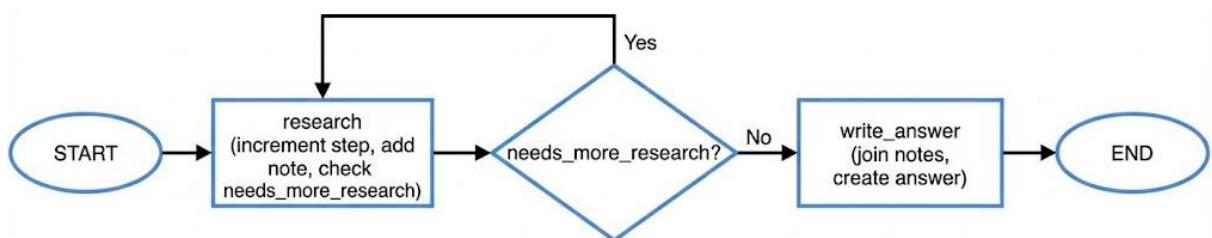
ch\_08\scr\routing\_and\_loop.py

```
(.venv) C:\Users\alexc\llm-app\ch_08> python -m src.routing_and_loop

--- Researching (Step 1) ---
--- Researching (Step 2) ---
--- Writing Final Answer ---

Final Results:
Steps taken: 2
Notes: ['note 1: looked up something relevant', 'note 2: looked up something relevant']
Final Answer: Answer based on: note 1: looked up something relevant;
note 2: looked up something relevant
```

This example shows how edges control both routing and looping in a tiny research workflow. The state keeps track of the question, the current step number, whether more research is needed, the accumulated notes, and the final answer. The `research` node increments `step_counter`, appends a new note, and sets `needs_more_research` based on how many steps have already run. The `write_answer` node just joins all notes into a single answer string.



The key part is the router and the edges. `route_after_research` looks at `needs_more_research` and returns either "research" or "write\_answer". `add_conditional_edges` wires this router to the graph so that, after each `research` step, the engine decides whether to loop back to the same node or move forward to `write_answer` and then `END`. The same pattern scales to more complex flows: edges declare the static structure (`START` → `research` → `write_answer` → `END`), while the router function decides at runtime whether to stay in the loop or exit with a final answer.

### 8.1.3 State objects and schemas

The state object is the data that flows along edges and binds the whole graph together. It is the shared working data of the workflow (the state every node can read and update). The State definition combines a schema (keys and types) with optional reducer functions per key that

control how updates are applied. At the low end, a state might only hold questions and answers. More realistic agents keep several groups of fields together, for example:

1. Question or objective
2. Messages, holding the conversation or chain of LLM/tool messages
3. Documents or `search_results`, as retrieved context
4. Answer or report, as the final synthesis
5. Control fields such as `needs_more_research` or `step_counter`
6. Diagnostics or logs, used for debugging and observability

Bundling these into a single schema means every node sees a consistent structure. Each node still only reads and writes the fields it needs. LangGraph supports several styles for this schema:

1. `TypedDict` for lightweight, explicit schemas that stay close to dictionaries
2. `dataclasses` when you want default values and convenience constructors
3. Pydantic models when you want validation and stricter checking, at the cost of some performance

For small graphs it is common to use a single schema for input, working state, and output. LangGraph also supports distinct input and output schemas: the input schema defines the structure you pass to `invoke()` (and is validated at runtime when you use a Pydantic model), while the output schema filters what the graph returns. When you define distinct schemas, nodes still read/write the graph's internal state channels; `input_schema` and `output_schema` act as filters on what the graph accepts and returns. This lets you keep internal bookkeeping keys (for example logs, tool traces, or intermediate documents) out of your public return value.

#### 8.1.4 Input/output schemas and internal channels

Think of a LangGraph state schema as a contract for “what data can exist inside this workflow”. Input and output schemas are just the public API around that contract: they control what the caller must provide and what the caller will see at the end, but they do not change what the workflow is allowed to keep internally. Here is the clean mental model:

1. `OverallState` (internal working state): `OverallState` is the real “memory” of the graph while it runs. It lists every key that nodes may pass between each other: intermediate values, flags, diagnostics, accumulated messages, and so on. Nodes read from this state and return partial updates back into it. This is the schema that defines what the graph can carry across steps.
2. `InputState` (what the caller must supply): `InputState` is a filter applied at the boundary when you call `graph.invoke(...)`. It says: “these are the fields the caller is expected to provide to start (or continue) a run.” It’s about validation and clarity for the caller, not about limiting what the graph can compute later. If a field is not in `InputState`, that only means the caller is not required (or even allowed, depending on validation strictness) to supply it directly. The graph can still create it internally if it exists in `OverallState`.
3. `OutputState` (what the caller receives back): `OutputState` is a filter applied at the end. It says: “after the graph finishes, only return these keys to the caller.” That is all it does: it shapes the return value. It does not stop nodes from writing other keys during execution; it just keeps those keys out of the final response object.

An “internal channel” is just a state key that exists in `OverallState` but is intentionally not part of `OutputState` (and often not part of `InputState` either). You use these keys for intermediate work: retrieved documents, tool traces, retry counters, routing decisions, temporary scratch data, etc. They are “private” only in the sense that callers don’t see them in the final return value—nodes can still read/write them as part of the internal workflow contract. A common confusion is that `PrivateState` is not a separate storage area

In examples, you may see separate `TypedDicts` like `PrivateState` used as function annotations for specific nodes. Treat that as a type hint describing the slice of state a particular node cares about, not as a second hidden state store. At runtime, the graph still has one evolving state object; if you want a key to reliably exist and flow between nodes, it should be part of the graph’s `OverallState`. Then you keep it “private” by simply excluding it from `OutputState`.

```

from __future__ import annotations
import operator
from typing import Annotated, Literal
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END

1) Public API: input + output

class InputState(TypedDict):
 # What the caller must supply
 user_id: str
 question: str

class OutputState(TypedDict):
 # What the caller receives back
 answer: str
 sources: list[str]

2) Internal union: everything nodes touch

IMPORTANT: every key any node will write must exist here.
"Private/internal channels" are simply keys you keep OUT of OutputState.
class OverallState(InputState):
 # Internal working data
 normalized_question: str
 need_retrieval: bool
 docs: list[str]

 # Internal diagnostics / paper trail (private)
 debug_log: Annotated[list[str], operator.add]

 # Final output keys (still part of internal state because nodes write them)
 answer: str
 sources: list[str]

3) Nodes: read state, emit updates

def preprocess(state: InputState) -> dict:
 q = state["question"].strip()
 normalized = " ".join(q.split()) # collapse whitespace
 return {
 "normalized_question": normalized,
 "debug_log": [f"preprocess: normalized question='{normalized}'"]
 }

def decide_route(state: OverallState) -> dict:
 # A toy routing rule: do retrieval if the question looks factual.
 q = state["normalized_question"].lower()
 need = any(word in q for word in ("who", "what", "when", "where", "source", "cite"))
 return {
 "need_retrieval": need,
 "debug_log": [f"route: need_retrieval={need}"]
 }

def retrieve(state: OverallState) -> dict:
 # In a real app you'd call a retriever / DB here.
 # We keep it deterministic so the example is runnable without external dependencies.

```

```

q = state["normalized_question"]
docs = [
 f"Doc A relevant to: {q}",
 f"Doc B relevant to: {q}"
]
return {
 "docs": docs,
 "debug_log": [f"retrieve: fetched {len(docs)} docs"]
}

def generate(state: OverallState) -> dict:
 # In a real app you'd call an LLM here. We'll just synthesize a response.
 q = state["normalized_question"]
 docs = state.get("docs", [])
 if docs:
 draft = f"Answer to '{q}' using {len(docs)} document(s)."
 else:
 draft = f"Answer to '{q}' (no retrieval used)."

 return {
 "answer": draft,
 "sources": docs, # we'll expose these in OutputState
 "debug_log": ["generate: produced final answer"]
 }

4) Conditional routing

def route_next(state: OverallState) -> Literal["retrieve", "generate"]:
 return "retrieve" if state["need_retrieval"] else "generate"

5) Build graph with input/output filters

builder = StateGraph(OverallState, input_schema = InputState, output_schema = OutputState)
builder.add_node("preprocess", preprocess)
builder.add_node("decide_route", decide_route)
builder.add_node("retrieve", retrieve)
builder.add_node("generate", generate)

builder.add_edge(START, "preprocess")
builder.add_edge("preprocess", "decide_route")
builder.add_conditional_edges("decide_route", route_next)
builder.add_edge("retrieve", "generate")
builder.add_edge("generate", END)

graph = builder.compile()

if __name__ == "__main__":
 # Caller supplies ONLY InputState.
 result = graph.invoke({"user_id": "u-123", "question": "What is LangGraph?"})
 print("invoke() result (filtered by OutputState):")
 print(result)

 # If you want to inspect internal state evolution (including "private" channels),
 # stream the run in "values" mode to see full state snapshots.
 print("\nstream(values) snapshots (full internal state, including debug_log):")
 for snapshot in graph.stream(
 {"user_id": "u-123", "question": "What is LangGraph?"},
 stream_mode = "values"
):
 print(snapshot)

```

ch\_08\scr\states.py

The script is a demonstration of how state is created, expanded, and filtered as a graph runs. The caller supplies only `InputState` (`user_id` and `question`). From that point on, every node participates in a single shared state object whose shape is defined by `OverallState`. Nodes do not pass values to each other directly; instead, each node returns a dict of updates (key/value pairs). The runtime merges those updates into the shared state, so later nodes can read what earlier nodes wrote. This is why any key a node might write (`normalized_question`, `need_retrieval`, `docs`, `debug_log`, `answer`, `sources`) must be declared in `OverallState`: that schema is the authoritative list of internal “channels” that can exist during execution.

`InputState` and `OutputState` are boundary contracts, not separate stores. `InputState` validates what is allowed/required at invoke-time, but it does not constrain internal computation: nodes can add new keys because `OverallState` is larger than `InputState`. `OutputState` is only a projection applied at the end: it filters the final returned object, but it does not delete or prevent internal keys from being used while the graph runs.

“Private” state is implemented by omission: keys like `debug_log` (and often docs or routing flags) live in `OverallState` so nodes can use them, but they stay private because `OutputState` excludes them.

### 8.1.5 Message-centric state and `MessagesState`

Chat-centric agents typically keep a list of messages in state. This pattern is so common that LangGraph includes a built-in `MessagesState` schema for this pattern. `MessagesState` already includes a `messages` field, defined so message updates are merged using LangGraph’s `add_messages` reducer. You can inherit from it to add more fields while reusing the standard handling of message history. A minimal example looks like this:

```

import operator
from typing import List, Union
from typing_extensions import TypedDict, Annotated
from langchain_core.messages import AnyMessage, HumanMessage, AIMessage
from langgraph.graph import MessagesState, StateGraph, START, END
from langgraph.graph.message import add_messages

1. Define the State
MessagesState already contains "messages": Annotated[list[AnyMessage], add_messages]
class ChatState(MessagesState):
 # Optional: You can add more fields here
 user_name: str

2. Define the Node function
def reply_node(state: ChatState) -> dict:
 # Takes the current state, processes the last message,
 # and returns a new message to be appended.
 # Get the last message from the list
 last_message = state["messages"][-1]
 user_text = last_message.content

 # Create the AI response
 response_content = f"Echoing back: '{user_text}'"

 # Return a dictionary where the key matches the state key
 # Because of the 'add_messages' reducer, this will append to the list
 return {"messages": [AIMessage(content = response_content)]}

3. Build the Graph
builder = StateGraph(ChatState)

Add our node
builder.add_node("reply", reply_node)

Define the flow
builder.add_edge(START, "reply")
builder.add_edge("reply", END)

4. Compile the Graph
graph = builder.compile()

5. Execute the Graph
if __name__ == "__main__":
 # Define initial input
 # We pass a list with one HumanMessage
 initial_input = {
 "messages": [HumanMessage(content = "Hello, LangGraph!")],
 "user_name": "Alice"
 }

 print("--- Starting Graph Execution ---")
 result = graph.invoke(initial_input)

 # 6. Print the conversation history
 print("\nConversation History:")
 for m in result["messages"]:

```

```
Print the type (Human/AI) and the text content
role = "User" if isinstance(m, HumanMessage) else "Assistant"
print(f"{role}: {m.content}")
```

ch\_08\scr\messagestate.py

This code uses `MessagesState` as the backbone of the graph's state so the conversation history is treated as a first-class, append-only log. The custom `ChatState` inherits from `MessagesState`, which means the state already has a `messages` field wired with the `add_messages` reducer. That reducer is the critical behavior: whenever a node returns `{"messages": [...]}`, LangGraph does not replace the existing history—it merges by appending the new messages onto the prior list.

Inside `reply_node`, the function reads from `state["messages"]` to access the current conversation. It deliberately looks at the last message only (`state["messages"][-1]`) to decide what to do next, which is a common pattern for turn-by-turn agents: use the most recent user input as the trigger, while still preserving the full transcript for downstream steps. The node then returns a list containing a single `AIMessage`. Returning a list (not a single message) keeps the update format consistent and lets the reducer append multiple messages if needed (for example, tool call + follow-up).

Because `MessagesState` manages message accumulation automatically, the graph wiring stays simple: a single node can read the growing transcript and contribute the next assistant turn without manual list mutation or bookkeeping.

### 8.1.6 Reducers

Reducers define how a state key is updated when nodes return partial state updates, especially how to aggregate multiple updates to the same key received from multiple nodes in the same superstep (for example, from parallel branches). Each key in the state schema can optionally be annotated with a reducer function. If a single update is received for that key in a superstep, it overwrites the previous value. If multiple nodes produce updates to the same key in the same superstep and no reducer is defined, LangGraph raises an error because it cannot resolve the conflict. Overwrite semantics are fine for fields where you always want the latest value: a boolean flag that indicates whether more research is needed, a final answer string, or a route label.

Accumulating fields need different behaviour. For example, a diagnostics list should collect everything logged by every node, not just the last set of messages. A message history should keep track of conversation turns. A counter should grow over time as different nodes increment it. Reducers let you define these behaviours per key.

In practice, you will see five “reducer options” used in real graphs:

1. Default (implicit) overwrite: No annotation on the field. A single update to that key in a superstep replaces the previous value; concurrent updates to the same key in the same superstep require a reducer (otherwise LangGraph raises an error).
2. Binary operator reducer via `Annotated[...]`: You bind a reducer function to a field using `typing.Annotated/typing_extensions.Annotated`. The most common example is `operator.add` to concatenate lists or add numbers.
3. `add_messages` (built-in reducer): A specialised reducer for a list of LangChain message objects. It appends brand new messages, and it also updates existing messages correctly by tracking message IDs. It also attempts to deserialize supported message formats into LangChain `Message` objects when updates are received.

4. Custom reducer function: You write your own reducer when you need custom merge semantics (e.g., accept either a single item or a list, de-duplicate, cap length, merge dicts, etc.).
5. Overwrite (bypass a reducer for a single update): Not a reducer itself, but a wrapper that bypasses the reducer and writes the wrapped value directly to the channel. If multiple Overwrite values are produced for the same key in a single superstep, LangGraph raises InvalidUpdateError.

Here is a simple example that accumulates logs and uses add\_messages for a message history:

```

import operator
from typing import Annotated, List, Union
from typing_extensions import TypedDict
from langchain_core.messages import AnyMessage, HumanMessage, AIMessage
from langgraph.graph import StateGraph, START, END
from langgraph.message import add_messages

1. Define the State with Reducers
Annotated + a function (like add_messages or operator.add)
tells LangGraph how to merge updates instead of overwriting.
class SupportState(TypedDict):
 # Appends messages to the list
 messages: Annotated[List[AnyMessage], add_messages]
 # Appends strings to the list
 logs: Annotated[List[str], operator.add]
 # Overwrites with the latest string (no reducer)
 answer: str

2. Define the Nodes
def classify_and_log(state: SupportState) -> dict:
 # Extracts intent and adds a log entry
 user_text = state["messages"][-1].content
 route = "returns" if "return" in str(user_text).lower() else "general"

 print(f"--- Node: Classify (Route: {route}) ---")
 return {
 "logs": [f"classify: detected route={route}"]
 }

def reply(state: SupportState) -> dict:
 # Generates the final response and adds a final log
 user_text = state["messages"][-1].content
 answer = f"I am a support bot. I see you're asking about: '{user_text}'."

 print(" --- Node: Reply ---")
 return {
 "messages": [AIMessage(content = answer)],
 "answer": answer,
 "logs": ["reply: generated answer"]
 }

3. Build the Graph
builder = StateGraph(SupportState)

builder.add_node("classify_and_log", classify_and_log)
builder.add_node("reply", reply)

Linear flow
builder.add_edge(START, "classify_and_log")
builder.add_edge("classify_and_log", "reply")
builder.add_edge("reply", END)

4. Compile and Run
graph = builder.compile()

if __name__ == "__main__":
 # Initial input
 initial_input = {
 "messages": [HumanMessage(content = "I'd like to return my order")],
 "logs": ["init: started workflow"]
 }

 # Execute
 final_output = graph.invoke(initial_input)
 # 5. Review the Accumulated State

```

```

print("\n--- Final State Summary ---")
print(f"Final Answer: {final_output['answer']}")

print("\nLogs (Accumulated via operator.add):")
for log in final_output['logs']:
 print(f" > {log}")

print("\nMessages (Accumulated via add_messages):")
for msg in final_output['messages']:
 role = "User" if isinstance(msg, HumanMessage) else "Bot"
 print(f" {role}: {msg.content}")

```

ch\_08\scr\reducer.py

This example shows how multiple Reducers work in tandem to manage different types of data in a support workflow. The state keeps track of the chat history via `add_messages`, an append-only log of internal actions via `operator.add`, and a final answer string that is overwritten at the end of the process. The `classify_and_log` node performs an initial pass on the user's input to determine the topic, contributing an entry to the log without modifying the messages. The `reply` node then generates the final response, updating all three keys of the state simultaneously.

The key part is the distinction between overwriting and accumulating state. By using `Annotated`, the graph structure allows nodes to focus only on their specific contributions (like a single log entry or a single message) while the underlying engine handles the complexity of merging those updates into the global state. This pattern is essential for complex agents that need to maintain a "paper trail" of their reasoning steps while simultaneously updating a conversation history.

Sometimes you want a reducer most of the time but need a way to reset or replace a value entirely. LangGraph exposes an `Overwrite` wrapper type for this. When a node returns a value wrapped in `Overwrite`, the configured reducer for that key is bypassed and the value is written directly. `Overwrite` is safe as long as only one node writes an `Overwrite` value for that key in a given superstep. Receiving multiple `Overwrite` values for the same key in a single superstep raises an `InvalidUpdateError`. That constraint is important for graph design: use `Overwrite` for resets where there is a single writer, not for keys that multiple branches might try to replace at once.

```

def maintenance_step(state: SystemState) -> dict:
 # This node decides to clear the logs if they get too long,
 # bypassing the operator.add reducer.
 if len(state["logs"]) >= 2:
 print("--- Node: Maintenance (Resetting Logs) ---")
 # Wrapping the value in Overwrite([]) replaces the whole list
 # instead of appending to it.
 return {"logs": Overwrite([]), "status": "Cleared"}

 return {"status": "Normal"}

```

### 8.1.7 Custom Reducers

A reducer is just a function that merges an existing value with an update value. In LangGraph, the reducer signature is `(Value, Value) -> Value`: the first argument is the current value already stored in state for that key, and the second argument is the new update being applied. The function returns the merged value that becomes the new state for that key.

Custom reducers matter when `operator.add` is "too dumb". Concatenation is fine for raw lists, but real workflows often need rules: de-duplicate, cap length, merge dictionaries by key, accept either a single item or a list, preserve ordering by an explicit index, or drop stale entries. Guidance: write reducers as pure merge functions (return a new value rather than mutating

inputs). Also, do not rely on a stable ordering of parallel updates: in a parallel superstep, the order of updates may not be consistent, so your reducer should be correct under any ordering.

#### Example: custom reducer that accepts single-or-list, de-duplicates, and caps history

The following example defines an events key that behaves like a bounded set-with-order. Nodes typically return updates using the state field's declared shape (for example, events: list[str] and updates like { 'events': [ '...' ] }). If you want ergonomic 'single-or-list' inputs, either normalize inside the node before returning, or declare the state field as a union type and have the reducer handle both. The reducer normalizes both forms, removes duplicates while preserving first-seen order, and caps the stored history to the last N items. This is a common pattern for "diagnostics", "observations", or "breadcrumbs" where you want accumulation but also need memory discipline.

```

from __future__ import annotations
from typing import Annotated, Iterable, List, Sequence, Union
from typing_extensions import TypedDict
from langgraph.graph import END, START, StateGraph

EventUpdate = Union[str, List[str]]

def merge_events(left: List[str], right: EventUpdate, *, max_items: int = 10) ->
List[str]:
 # Custom reducer for a bounded, de-duplicated event history.
 # Signature conceptually matches LangGraph's reducer shape: (Value, Value) -> Value.
 # - left: current value stored in state (List[str])
 # - right: new update produced by a node (either a single str or a List[str])
 # Returns a new List[str] (does not mutate left in-place).
 if right is None:
 return list(left)

 # Normalize update to a list.
 if isinstance(right, str):
 incoming = [right]
 else:
 incoming = list(right)

 # Merge while de-duplicating, preserving first-seen order.
 merged: List[str] = list(left)
 seen = set(merged)
 for item in incoming:
 if item not in seen:
 merged.append(item)
 seen.add(item)

 # Cap to last max_items.
 if len(merged) > max_items:
 merged = merged[-max_items:]

 return merged

class State(TypedDict):
 events: Annotated[List[str], merge_events]
 answer: str

def step_a(state: State) -> dict:
 # Single item update (allowed by custom reducer).
 return {"events": "a:started"}

def step_b(state: State) -> dict:
 # List update (also allowed).
 return {"events": ["b:started", "b:finished", "a:started"]} # includes a duplicate

def finalize(state: State) -> dict:
 # The reducer has already merged everything into a bounded, de-duplicated list.
 return {"answer": "ok"}

def build_graph():
 builder = StateGraph(State)
 builder.add_node("a", step_a)
 builder.add_node("b", step_b)
 builder.add_node("finalize", finalize)

 builder.add_edge(START, "a")
 builder.add_edge("a", "b")

```

```

builder.add_edge("b", "finalize")
builder.add_edge("finalize", END)
return builder.compile()

if __name__ == "__main__":
 graph = build_graph()
 out = graph.invoke({"events": [], "answer": ""})
 print("events:", out["events"])
 print("answer:", out["answer"])

```

ch\_08\scr\custom\_reducer.py

### 8.1.8 Configuration

Configuration controls how the graph runs without changing the state schema or node logic. LangGraph relies on the same RunnableConfig concept as the rest of LangChain. RunnableConfig is a dictionary-like object that supports standard keys such as callbacks, tags, metadata, run\_name, max\_concurrency, recursion\_limit, and run\_id, plus a configurable section where you can place arbitrary run-time values.

recursion\_limit caps the number of graph steps (super-steps), not Python recursion. A superstep is one round where all currently-ready nodes run and their updates are merged into state. Loops consume supersteps. If a loop iteration fans out into multiple rounds (for example, one node runs, then two parallel nodes run, then the loop continues), that single logical “lap” can consume multiple supersteps. If the graph exceeds recursion\_limit before reaching END (or before all nodes become inactive), LangGraph raises GraphRecursionError. Treat this as a safety guard against unintended cycles and non-terminating control flow. If you expect a graph to legitimately take many steps, set recursion\_limit explicitly for that invocation or stream call. You can observe and handle step growth from inside the graph. LangGraph exposes the current step counter in the runnable configuration metadata: config["metadata"]["langgraph\_step"]. Reading this value inside nodes enables proactive “wind-down” logic (for example, switch to a cheaper path, skip optional work, or emit a best-effort answer before the limit is hit). LangGraph also provides a RemainingSteps managed value you can add to state to track how many steps are left before the limit is hit.

You pass this config when invoking a graph or any other Runnable:

```

result = graph.invoke(
 input_state,
 config = {
 "tags": ["support", "prod"],
 "metadata": {"tenant": "acme"},
 "max_concurrency": 4,
 "recursion_limit": 20,
 "configurable": {"thread_id": "user-123"}
 }
)

```

Nodes can also accept this config as an optional second parameter:

```

from langchain_core.runnables import RunnableConfig

def node_with_config(state: SupportState, config: RunnableConfig) -> dict:
 thread_id = config.get("configurable", {}).get("thread_id", "unknown")
 current_step = config.get("metadata", {}).get("langgraph_step", "unknown")
 # Use thread_id for logging, persistence, or routing decisions.
 # Use current_step for proactive "wind-down" logic in long-running or looping graphs.
 return {"logs": [f"thread={thread_id} step={current_step}"]}

```

Config is useful for tracing, controlling parallelism, and passing per-run options such as a thread identifier or a “fast vs high-quality” mode. As guidance: keep config values lightweight and serializable (especially metadata), because config is intended for tracing/callbacks and

runtime configuration. Pass heavyweight dependencies (DB connections, client instances) via normal application wiring (closures/DI) or via LangGraph's runtime/context mechanisms, not through metadata. If you see `GRAPH_RECURSION_LIMIT` / `GraphRecursionError` unexpectedly, it usually indicates a cycle that never reaches a stop condition; review your conditional edges and termination criteria, then raise `recursion_limit` only when the long run is intentional.

### 8.1.9 Runtime context and `context_schema`

For those heavier, immutable dependencies, LangGraph provides a runtime context mechanism. When you build a `StateGraph`, you can declare a `context_schema` in addition to the state schema. The context schema describes immutable data that should be available to nodes during a run but should not be treated as state: for example `user_id`, `region`, `model_provider`, or a database connection handle. The pattern looks like this:

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

1. State Definition
class State(TypedDict):
 question: str
 answer: str

2. Context Schema
@dataclass
class Context:
 region: str = "US"

3. Node Definition
We use the argument name 'runtime'—LangGraph will inject the object here
def answer_returns(state: State, runtime):
 q = state["question"].lower()

 # Access context via the injected runtime object
 region = runtime.context.region

 if "return" not in q and "refund" not in q:
 return {"answer": "Ask me about orders, returns, payments, or accounts."}

 if region == "EU":
 text = "EU returns: start a return with your order number within 30 days of delivery."
 else:
 text = "US returns: start a return from your order page; a label is provided for eligible items."

 return {"answer": text}

4. Graph Construction
Passing context_schema tells the graph how to validate the 'context' arg in invoke
builder = StateGraph(State, context_schema = Context)
builder.add_node("answer_returns", answer_returns)
builder.add_edge(START, "answer_returns")
builder.add_edge("answer_returns", END)

graph = builder.compile()

if __name__ == "__main__":
 # 5. Invoke passing 'context' directly as a keyword argument
 out = graph.invoke(
 {"question": "How do I return an item?", "answer": ""},
 Context = Context(region = "EU")
)
 print(out["answer"])
```

`ch_08\scr\context_schema.py`

Here the state captures what this particular run is working on (the question and answer). The context captures who this run is for and how the system should behave (the region). The runtime passes context into each node through the Runtime object. This separation matches

the general “context engineering” view: mutable state holds evolving facts; immutable context holds the stable frame in which those facts are interpreted.

### 8.1.10 Putting it together

Nodes and edges describe the shape of the computation. The state object describes what information it carries. Reducers define how updates to the same state key are combined, which matters most when multiple nodes write that key in the same superstep (parallel fan-out/fan-in). If you don’t define a reducer for a key, later updates overwrite earlier ones. Overwrite is an escape hatch that bypasses the reducer for a specific update, setting that state key directly when you need to reset or replace an accumulated value. RunnableConfig and Runtime Context let you tune execution and pass dependencies (for example IDs, clients, or other per-run data) into nodes without putting those dependencies into the graph state. Together, these concepts form the basic mental model you need before you start designing more advanced graph patterns (for example fan-out/fan-in flows, long-running agents with checkpoints, and multi-tenant execution patterns).

**Working example 1: routing, fan-out/fan-in, reducers, configuration, and runtime context**  
 This example models a support workflow that classifies a question, gathers two independent “facts” in parallel, then produces a final answer. It demonstrates why reducers matter when parallel branches update the same key.

```
from __future__ import annotations
import operator
from dataclasses import dataclass
from typing import Annotated, Literal
from typing_extensions import TypedDict
from langchain_core.runnables import RunnableConfig
from langgraph.graph import StateGraph, START, END
from langgraph.runtime import Runtime

Immutable run-scoped context (not part of state).
@dataclass(frozen = True)
class Context:
 region: Literal["EU", "US"]

Shared mutable state.
class State(TypedDict):
 question: str
 route: Literal["returns", "general"]
 retrieved: Annotated[list[str], operator.add]
 logs: Annotated[list[str], operator.add]
 answer: str

def classify(state: State, config: RunnableConfig) -> dict:
 # Classify the question. This node reads run-scoped configuration to decide
 # what words count as "returns related".
 keywords = config.get("configurable", {}).get("returns_keywords", ["return",
 "refund"])
 q = state["question"].lower()
 route: Literal["returns", "general"] = "returns" if any(k in q for k in keywords)
 else "general"
 return {
 "route": route,
 "logs": [f"classify: route={route} (keywords={keywords})"]
 }

def route_after_classify(state: State) -> Literal["returns_fanout", "fetch_general_info"]:
 return "returns_fanout" if state["route"] == "returns" else "fetch_general_info"

def returns_fanout(state: State) -> dict:
 # Fan-out node. It exists only to trigger multiple outgoing edges so downstream
 # nodes can be scheduled concurrently in the next superstep.
 return {"logs": ["returns_fanout"]}

def fetch_policy(state: State, runtime: Runtime[Context]) -> dict:
 # Fetch region-specific policy. The behaviour changes based on runtime context.
 if runtime.context.region == "EU":
 policy = "EU policy: returns accepted within 30 days of delivery."
 else:
```

```

 policy = "US policy: returns accepted within 30 days; label provided for eligible
 items."
 return {
 "retrieved": [policy],
 "logs": [f"fetch_policy: region={runtime.context.region}"]
 }

def fetch_order_facts(state: State) -> dict:
 # Fetch order facts. This node is independent of policy lookup and can run in
 # parallel.
 return {
 "retrieved": ["Order facts: the item was delivered 12 days ago."],
 "logs": ["fetch_order_facts: delivery_age_days=12"]
 }

def fetch_general_info(state: State) -> dict:
 return {
 "retrieved": ["General support scope: orders, returns, payments, accounts."],
 "logs": ["fetch_general_info"]
 }

def synthesize(state: State) -> dict:
 # Produce a final answer based on whatever was retrieved.
 evidence = " ".join(state["retrieved"])
 if state["route"] == "returns":
 answer = f"You can start a return. {evidence}"
 else:
 answer = f"I can help. {evidence}"
 return {
 "answer": answer,
 "logs": ["synthesize"]
 }

def build_graph():
 builder = StateGraph(State, context_schema = Context)
 builder.add_node("classify", classify)
 builder.add_node("returns_fanout", returns_fanout)
 builder.add_node("fetch_policy", fetch_policy)
 builder.add_node("fetch_order_facts", fetch_order_facts)
 builder.add_node("fetch_general_info", fetch_general_info)
 builder.add_node("synthesize", synthesize)
 builder.add_edge(START, "classify")

 # Conditional routing based on the classification.
 builder.add_conditional_edges(
 "classify",
 route_after_classify,
 {
 "returns_fanout": "returns_fanout",
 "fetch_general_info": "fetch_general_info"
 }
)

 # Fan-out for the "returns" path: policy and order facts run as two branches.
 builder.add_edge("returns_fanout", "fetch_policy")
 builder.add_edge("returns_fanout", "fetch_order_facts")

 # Fan-in: synthesize runs after both branches complete.
 builder.add_edge("fetch_policy", "synthesize")
 builder.add_edge("fetch_order_facts", "synthesize")

 # Straight-line for the "general" path.
 builder.add_edge("fetch_general_info", "synthesize")

 builder.add_edge("synthesize", END)

 return builder.compile()

if __name__ == "__main__":
 graph = build_graph()

 initial_state: State = {
 "question": "How do I return an item that does not fit?",
 "route": "general",
 "retrieved": [],
 "logs": [],
 "answer": ""
 }

 out = graph.invoke(
 initial_state,
 config={
 "configurable": {"returns_keywords": ["return", "refund", "does not fit"]},
 "log_level": "INFO"
 }
)

```

```

 "tags": ["support", "routing-demo"]
 },
 Context = Context(region = "EU")
)

print("answer:", out["answer"])
print("retrieved:", out["retrieved"])
print("logs:")
for line in out["logs"]:
 print("-", line)

```

ch\_08\scr\support\_graph.py

The graph models a “returns vs general support” assistant. State is a TypedDict with the user’s question, a route field, retrieved evidence, logs, and a final answer. The retrieved and logs fields use list-appending reducers (`operator.add`), so updates produced by parallel branches are accumulated rather than overwritten; this is the required pattern when multiple nodes update the same key in a fan-out.

Execution starts at `classify`, which reads the question and `RunnableConfig`. It uses `config["configurable"]["returns_keywords"]` to decide whether the question is returns-related, then writes the chosen route and a log entry. `route_after_classify` is a router wired via `add_conditional_edges`: it sends the flow either to `returns_fanout` (returns path) or to `fetch_general_info` (general path).

For returns questions, `returns_fanout` exists purely to create a real fan-out: it has two outgoing edges to `fetch_policy` and `fetch_order_facts`, so those two nodes are eligible to execute concurrently in the next superstep. `fetch_policy` uses runtime context (`Context.region`) to select a region-specific policy string; `fetch_order_facts` is independent and returns delivery timing facts. Both append into `retrieved` and `logs`; because they execute in the same parallel superstep, their updates can arrive in a non-deterministic order, but they are merged safely by the reducers. The flow converges at `synthesize`, which runs only after both branches finish and builds the final answer from the accumulated retrieved evidence.

### Working example 2: message state, add\_messages, and forcing a reset with Overwrite

This example shows a small, testable pattern you will use in almost every chat-style graph: keep message history as an append-only list, keep operational logs as an append-only list, and still retain the ability to “reset” one of those histories without disturbing the other. The point is to demonstrate that “append by default” and “replace on demand” can coexist in the same state schema when you use reducers for accumulation and `Overwrite` for explicit resets.

```

from __future__ import annotations
import operator
from typing import Annotated
from typing_extensions import TypedDict
from langchain_core.messages import AIMessage, AnyMessage, HumanMessage
from langgraph.graph import END, START, StateGraph
from langgraph.message import add_messages
from langgraph.types import Overwrite

class State(TypedDict):
 messages: Annotated[list[AnyMessage], add_messages]
 logs: Annotated[list[str], operator.add]
 reset_logs: bool

def reply(state: State) -> dict:
 # Read the last user message and append an AIMessage reply plus a log entry.
 last = state["messages"][-1]
 if not isinstance(last, HumanMessage):
 return {"logs": [f"reply: expected HumanMessage, got {type(last).__name__}"]}
 text = last.content
 return {

```

```

 "messages": [AIMessage(content = f"Echo: {text}")],
 "logs": [f"reply: input_len={len(text)}"]
 }

 def maybe_reset(state: State) -> dict:
 # If reset_logs is true, clear the entire logs list by bypassing the reducer.
 if state["reset_logs"]:
 return {"logs": Overwrite([])}
 return {}

builder = StateGraph(State)
builder.add_node("reply", reply)
builder.add_node("maybe_reset", maybe_reset)

builder.add_edge(START, "reply")
builder.add_edge("reply", "maybe_reset")
builder.add_edge("maybe_reset", END)

graph = builder.compile()

if __name__ == "__main__":
 # Run 1: normal flow (logs accumulate, messages accumulate)
 state_1: State = {
 "messages": [HumanMessage(content = "Hello")],
 "logs": [],
 "reset_logs": False
 }
 out_1 = graph.invoke(state_1)

 print("Run 1 logs:", out_1["logs"])
 print("Run 1 last message:", out_1["messages"][-1].content)

 # Run 2: carry forward message history, request a reset, and clear logs via Overwrite
 state_2: State = {
 "messages": out_1["messages"] + [HumanMessage(content = "/reset")],
 "logs": out_1["logs"],
 "reset_logs": True
 }
 out_2 = graph.invoke(state_2)

 print("Run 2 logs:", out_2["logs"])
 print("Run 2 last message:", out_2["messages"][-1].content)

```

ch\_08\scr\message\_and\_overwrite.py

The graph models a minimal chat flow with message history and a resettable log. State is a TypedDict with three fields: messages, logs, and reset\_logs. messages is annotated with add\_messages, so nodes emit only the new messages, and the reducer appends (and can also update existing messages by ID). logs is annotated with operator.add, so each node can add log entries without overwriting previous ones. reset\_logs is a boolean flag that controls whether the log should be cleared.

Execution starts at reply, which reads the last user message from messages and returns a single AIMessage that echoes its content plus one log entry recording the input length. Because messages uses add\_messages, returning {"messages": [AIMessage(...)]} appends that message to the existing history rather than replacing it, and message objects are available via dot access such as state["messages"][-1].content.

Control then passes to maybe\_reset. If reset\_logs is true, the node returns {"logs": Overwrite([])} to bypass the reducer and replace the entire logs list with an empty one. If reset\_logs is false, it returns an empty dict and leaves logs unchanged.

## 8.2 CHECKPOINTS AND PERSISTENCE FOR LONG-RUNNING WORKFLOWS

LangGraph workflows are designed to outlive a single request. A support ticket can span dozens of turns, a trading assistant might run multi-step analyses with pauses for confirmation, and a batch job can take minutes or hours. In real deployments you cannot assume a single in-memory Python process will stay alive for the whole lifetime of a workflow, and you also cannot assume that the next user request will land on the same server instance.

That is what checkpoints are for. A checkpoint is a durable snapshot of a workflow so it can pause and resume without losing where it is.

### 8.2.1 The core model: threads + checkpointers

LangGraph persistence is built around two ideas:

1. A thread is one logical workflow/session over time (a stable context that can span multiple runs). Think ‘this user’s conversation’, ‘this support ticket’, ‘this portfolio analysis job’.
2. A checkpointer is the storage backend responsible for writing and reading checkpoints for threads.

When you run a graph with persistence enabled, you provide a `thread_id`. That id is not part of your state schema; it is passed in runtime configuration and is used to associate all checkpoints with the correct workflow instance. If you call the same graph again with the same `thread_id`, LangGraph loads the latest saved state for that thread (its most recent checkpoint) and continues from there, instead of starting from an empty state.

**A practical warning: thread identifiers are part of your security surface. Treat them like session tokens: use non-guessable ids (for example UUIDs) and enforce authorization checks in your application before resuming a thread.**

It’s tempting to think “checkpoint = chat history”. That is not accurate.

A checkpoint is a snapshot of the workflow at a point in time that includes the full state plus execution metadata needed to continue correctly. This includes the full state plus execution metadata needed to continue, such as what is scheduled next and bookkeeping that supports recovery when nodes fail mid-step. That extra metadata is what makes “resume” and “time travel debugging” possible. Because checkpoints capture execution context (not just messages), they are the mechanism behind:

1. Durability (survive restarts),
2. Debugging (“rewind” to a known point),
3. Human-in-the-loop pauses (save state, wait, resume).

#### Example: minimal persistence with `MemorySaver` (`thread_id` and `checkpoint_id`)

This first example is intentionally tiny. It demonstrates three things:

1. Providing a `thread_id` creates a per-thread history,
2. Listing checkpoints for a thread,
3. Forking a new run from a specific `checkpoint_id` (not only the latest) on a thread; this is the foundation of rollback and time travel.

```
from __future__ import annotations
from typing import Annotated, List
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.memory import MemorySaver
```

```

from langgraph.graph.message import add_messages
from langchain_core.messages import HumanMessage, AIMessage, BaseMessage

StateGraph state with an APPEND reducer for messages
class ChatState(TypedDict, total = False):
 messages: Annotated[List[BaseMessage], add_messages]

def test_node(state: ChatState) -> dict:
 messages = state.get("messages", [])
 history = messages[:-1] # everything except the current human input

 print("--- Node Execution ---")
 print(f"History length (previous messages) = {len(history)}")

 # IMPORTANT: with add_messages, you return ONLY the new messages to append
 return {"messages": [AIMessage(content = "Hello!")]}

Build graph
builder = StateGraph(ChatState)
builder.add_node("test_node", test_node)
builder.add_edge(START, "test_node")
builder.add_edge("test_node", END)

memory = MemorySaver()
graph = builder.compile(checkpointer = memory)

Run with different thread IDs
print("Running Thread A (Initial)")
graph.invoke(
 {"messages": [HumanMessage(content = "Hi from A")]},
 config = {"configurable": {"thread_id": "thread-a"}}
)

print("\nRunning Thread B (Initial)")
graph.invoke(
 {"messages": [HumanMessage(content = "Hi from B")]},
 config = {"configurable": {"thread_id": "thread-b"}}
)

Same thread ID again -> history should now include prior Human+AI from thread-a
print("\nRunning Thread A again (Resuming history)")
graph.invoke(
 {"messages": [HumanMessage(content = "Second message from A")]},
 config = {"configurable": {"thread_id": "thread-a"}}
)

Inspect checkpoints for Thread A
print("\n--- Checkpoint History for Thread A ---")
checkpoints = list(memory.list(config = {"configurable": {"thread_id": "thread-a"})))

The in-memory checkpointer's list method in the current MemorySaver
implementation, checkpoints are typically returned newest-first; do not rely on
ordering unless the backend guarantees it-sorts by checkpoint metadata when
order matters.
for cp in checkpoints:
 print(f"checkpoint_id: {cp.config['configurable']['checkpoint_id']}")

Time travel to an earlier checkpoint: checkpoints[-1] is the oldest checkpoint for this
thread
earlier_checkpoint_id = checkpoints[-1].config["configurable"]["checkpoint_id"]

print(f"\nTime Traveling on Thread A to checkpoint: {earlier_checkpoint_id}")
IMPORTANT: passing checkpoint_id does not "continue from that point" mid-execution.
It forks from that checkpoint: LangGraph replays steps up to checkpoint_id, then
executes subsequent steps as a new history branch (a new fork), even if they ran before.
graph.invoke(
 {"messages": [HumanMessage(content = "Restored input")]},
 config = {"configurable": {"thread_id": "thread-a", "checkpoint_id": earlier_checkpoint_id}}
)

```

ch\_08\scr\thread\_and\_checkpoint.py

```

(.venv) C:\Users\alexc\llm-app\ch_08> python -m src.
thread_and_checkpoint
Running Thread A (Initial)
--- Node Execution ---
History length (previous messages) = 0

Running Thread B (Initial)

```

```

--- Node Execution ---
History length (previous messages) = 0

Running Thread A again (Resuming history)
--- Node Execution ---
History length (previous messages) = 2

--- Checkpoint History for Thread A ---
checkpoint_id: 1f0fde78-c5c3-60a0-8004-515ba4eeb672
checkpoint_id: 1f0fde78-c5c1-66d4-8003-0d2b162b0cc7
checkpoint_id: 1f0fde78-c5c0-6211-8002-e82de76d1d37
checkpoint_id: 1f0fde78-c5b6-6891-8001-70636977446d
checkpoint_id: 1f0fde78-c5b3-6409-8000-f76090270f61
checkpoint_id: 1f0fde78-c5a0-6254-bfff-6e56f50e2fff

Time Traveling on Thread A to checkpoint: 1f0fde78-c5a0-6254-bfff-
6e56f50e2fff
--- Node Execution ---
History length (previous messages) = 0

```

This script wires a `MemorySaver` into the compiled graph, so every invocation persists a checkpointed snapshot of state. In-memory saver (`MemorySaver`) as suitable for debugging/testing rather than production persistence. Memory is not persisted between runs.

The key lever is the configurable `thread_id` passed on each invoke call. When you run with `thread_id="thread-a"` and then with `thread_id="thread-b"`, `MemorySaver` keeps two independent checkpoint streams: each thread accumulates its own message history and run metadata. When you invoke the graph a second time with `thread_id="thread-a"`, the graph resumes from the latest saved checkpoint for that thread, so the node sees prior messages from earlier Thread A runs (but not from Thread B).

After those runs, the code inspects persistence by calling `memory.list()` with the same `thread_id`. That returns all checkpoints for Thread A, and the loop prints the `checkpoint_id` stored in each checkpoint's config. The comment calls out an important operational detail: `MemorySaver` may return checkpoints newest-first, so ordering is an implementation detail unless your backend guarantees it.

Finally, the “time travel” step selects an older `checkpoint_id` (the oldest one in this example) and passes it alongside the same `thread_id` on a new invoke. That forces the run to fork from that earlier snapshot, replaying up to the checkpoint and then producing a new branch of history for Thread A.

### 8.2.2 Designing nodes for persistence: idempotency and side effects

With checkpointing enabled, a run can be resumed from saved state after an interruption, so you should assume the same node may be re-entered when a run is resumed or re-invoked. That means you must treat any node that performs side effects (charging a payment, sending an email, placing an order, writing to an external system) as dangerous unless it is idempotent under re-entry (resume/replay/re-invocation). The safe pattern is simple: before performing a side effect, check whether it has already happened for this business operation. If you use LangGraph threads, store and look up that marker within the thread's persisted state so a resumed run can see it. Then record the outcome in the graph state (so it is captured in checkpoints) so a resumed run can see it. For high-stakes side effects, also rely on the external system's idempotency support (or an idempotency key you control) rather than state alone.

**Example: a side-effect gate that won't double-charge**

This example models a payment step that can be safely retried without accidentally charging a customer twice. The system treats “processing an order” as a long-running workflow that may be interrupted and resumed, or invoked more than once (for example, if the caller retries a request or a run is restarted from a checkpoint). To stay safe, the workflow records (in persisted graph state) whether the payment side effect has already been confirmed for that specific order/payment identifier. On a subsequent run of the same order workflow, it consults that recorded state and skips the charge if payment was already confirmed, preserving the original receipt reference.

```

from __future__ import annotations
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.sqlite import SqliteSaver

class OrderState(TypedDict):
 order_id: str
 payment_confirmed: bool
 payment_receipt_id: str

def charge_payment(state: OrderState) -> dict:
 # This check provides idempotency logic inside the node
 if state.get("payment_confirmed"):
 print("--- Payment already confirmed. Skipping gateway call. ---")
 return {}

 print("--- Calling Payment Gateway... ---")
 receipt_id = f'receipt-{state["order_id"]}'
 return {"payment_confirmed": True, "payment_receipt_id": receipt_id}

builder = StateGraph(OrderState)
builder.add_node("charge_payment", charge_payment)
builder.add_edge(START, "charge_payment")
builder.add_edge("charge_payment", END)

1. Use the 'with' statement to properly initialize the SQLite connection
with SqliteSaver.from_conn_string("checkpoints_payments.db") as saver:
 graph = builder.compile(checkpointer = saver)
 cfg = {"configurable": {"thread_id": "order-123"}}

 # 2. First Run: This will trigger the payment gateway logic
 print("Run 1:")
 initial_input = {"order_id": "order-123", "payment_confirmed": False,
 "payment_receipt_id": ""}
 print(graph.invoke(initial_input, config = cfg))

 # 3. Second Run: Use an empty dict to resume the state from the checkpoint
 # If you pass 'payment_confirmed': False here, it will overwrite the DB and charge
 # again!
 print("\nRun 2 (Idempotent check):")
 print(graph.invoke({}, config = cfg))

```

ch\_08\scr\idempotency.py

```
(.venv) C:\Users\alexc\llm-app\ch_08> python -m src.idempotency
Run 1:
--- Calling Payment Gateway... ---
{'order_id': 'order-123', 'payment_confirmed': True,
 'payment_receipt_id': 'receipt-order-123'}

Run 2 (Idempotent check):
--- Payment already confirmed. Skipping gateway call. ---
None
```

The node `charge_payment` contains the idempotency gate for the workflow. On entry, it inspects the current state and looks for `payment_confirmed`. If that flag is already true, the node immediately returns an empty update, which means it performs no external work and does not mutate state. That “do nothing” return is deliberate: it makes re-entering the node

safe, because a resumed or repeated execution will take the same path and skip the gateway call.

If `payment_confirmed` is missing or false, the node performs the side effect (represented here by “Calling Payment Gateway...”). It then returns a partial state update that flips `payment_confirmed` to true and stores a deterministic `payment_receipt_id` derived from the `order_id`. Persisting both the boolean marker and the receipt identifier matters: the boolean prevents double charging, and the receipt preserves the original outcome for downstream steps.

The two invocations demonstrate how this guard behaves with checkpointed state. The first run starts from an explicit initial state and records the successful payment. The second run passes an empty input, forcing the graph to resume from the stored checkpoint for the same `thread_id`; the node sees `payment_confirmed` already set and safely skips charging. The comment highlights a key pitfall: providing a new state with `payment_confirmed=False` would overwrite the saved marker and defeat the idempotency gate.

### 8.2.3 Short-term memory vs long-term memory

The payment example shows short-term memory in its most practical form: a workflow carries a state object forward, and the system can pick up that same state again later. The unit of continuity is the thread. When you rerun the workflow with the same thread identifier, the checkpointer can reload the latest saved state for that thread and resume execution from that point. That is why a later run can sometimes be invoked with no new input (for example, `None`) or with minimal input: the durable state already exists and can drive the next decision.

For conversational agents, the same mechanism is usually used to keep the conversation alive across turns. The state will often include the recent message history, or a windowed/summarised version of it, plus whatever working variables the agent needs (current intent, extracted entities, tool results, partial plans). With a persistent checkpointer (for example, a database-backed saver), checkpointing preserves thread-scoped working context across invocations and long pauses, and it can survive process restarts. With an in-memory checkpointer, it does not. It is the memory of “this one conversation” or “this one job”.

That scope boundary matters. Checkpoints are not designed to become a global brain. They do not automatically give you knowledge that should apply to every new thread, every new session, or every future task. In many systems, you model each new conversation (or each new order/work item) as its own thread. But this is a design choice: you can also reuse a thread identifier when you want continuity across sessions.

If you only rely on checkpoints, you get continuity inside each instance, but you do not accumulate durable knowledge across instances.

Long-term memory fills that gap. It is a separate persistence layer where you store information meant to outlive any single thread. A common model in LangGraph is a store of JSON documents keyed by a hierarchical namespace plus a key. The namespace keeps domains clean (“`user_profiles`”, “`merchant_policies`”, “`org_faq`”, “`project_decisions`”), while the key identifies a specific record (“`user:42`”, “`policy:EU:returns`”, “`team:payments:runbook`”). You typically use a small set of operations: Put (write/update), Get (fetch by key), and Search (retrieve by filters and optionally by semantic similarity). Many store backends also support listing and deleting items.

That “search” capability is what makes long-term memory feel like memory rather than just storage. If you only ever do key-based gets, you have a key/value store. Adding metadata filtering and semantic search is what makes it behave like ‘memory’ for an agent. If you can search by metadata (region, product line, date, confidence, owner) and also retrieve by meaning (embedding/vector similarity), you can recall relevant facts even when you do not know the exact key. In practice, agents work best with a layered approach:

1. Checkpoints answer: “Where is this specific workflow instance right now, and what is its working context?”
2. Stores answer: “What durable facts have we learned over time that should apply across many workflows?”

The two layers also have different failure modes and responsibilities. Checkpointed state can be accidentally overwritten if you resume a thread by re-sending fields that conflict with what persisted, because those new inputs may overwrite parts of state depending on your merge/reducer rules. The payment example calls this out directly: for example, re-sending `payment_confirmed=False` when resuming could overwrite the persisted value (depending on your state merge rules) and could lead your workflow to attempt the charge again. Long-term stores have a different risk: they can accumulate stale, incorrect, or overly personal information if you do not apply governance (what is allowed to be stored, retention rules, provenance, and a way to correct or delete records).

A concrete way to think about it is to separate “working memory” from “profile memory”. In a customer support assistant, the current thread might include the last few messages, the current ticket status, and the tool outputs gathered so far. The long-term store might contain stable user preferences (language, contact channel, accessibility needs), verified account attributes, and organisational knowledge (approved policies, product specs, runbooks). When a new ticket starts and you create a new `thread_id`, that thread begins with an empty checkpoint history; you can then preload relevant long-term records into state for that thread. The agent can preload relevant long-term records into the thread as context. That keeps each workflow self-contained while still benefiting from durable knowledge.

The payoff is reliability. Checkpoints give you resumability and consistency within a single run. Long-term stores give you continuity of knowledge across runs. Combining them lets an agent behave like a careful professional: it remembers what happened in this case, and it also remembers the stable facts that should not need to be re-learned every time.

### Example: storing long-term preferences in a BaseStore-style store

The objective of this example is to show how to build a support workflow that can both resume reliably across interruptions and remember user-specific preferences across separate conversations. It does this by combining two persistence mechanisms: per-thread checkpoints, which let a single conversation thread pause and continue later, and a shared store keyed by `user_id`, which lets preferences written in one thread be reused automatically in another thread for the same user.

```
from __future__ import annotations
from typing_extensions import TypedDict
from langchain_core.runnables import RunnableConfig
from langgraph.checkpoint.sqlite import SqliteSaver
from langgraph.graph import START, END, StateGraph
from langgraph.store.base import BaseStore
from langgraph.store.memory import InMemoryStore

class SupportState(TypedDict, total = False):
 turn: int
```

```

message: str
prefs: dict
reply: str

def load_prefs(state: SupportState, config: RunnableConfig, *, store: BaseStore) -> dict:
 # Long-term memory read: pull user prefs into the thread state.
 thread_id = config["configurable"].get("thread_id")
 user_id = config["configurable"].get("user_id")

 if not user_id:
 print(f"[load_prefs] thread={thread_id} user=None loaded_prefs=None")
 return {"prefs": {}}

 namespace = (user_id, "profile")
 item = store.get(namespace, "preferences")
 loaded = None if item is None else item.value

 print(f"[load_prefs] thread={thread_id} user={user_id} loaded_prefs={loaded}")
 return {"prefs": {} if item is None else item.value}

def reply_or_remember(state: SupportState, config: RunnableConfig, *, store: BaseStore) -> dict:
 # Short-term memory write: increment turn and save reply into the thread state.
 # Long-term memory write (optional): persist a preference to the shared store.
 thread_id = config["configurable"].get("thread_id")
 user_id = config["configurable"]["user_id"]

 loaded_turn = state.get("turn")
 turn = int(loaded_turn or 0) + 1
 print(f"[reply_or_remember] thread={thread_id} loaded_turn={loaded_turn} next_turn={turn}")

 message = (state.get("message") or "").strip()
 namespace = (user_id, "profile")

 # Long-term write: one command stores a durable preference for this user.
 if message.lower().startswith("remember:"):
 value = message.split(":", 1)[1].strip().lower()
 prefs = {"short_mode": ("short" in value)}
 store.put(namespace, "preferences", prefs)

 reply = f"[turn {turn}] Saved preferences for {user_id}: {prefs}"
 return {"turn": turn, "reply": reply}

 # Use long-term prefs (loaded by the previous node) to shape the reply.
 short_mode = bool(state.get("prefs", {}).get("short_mode", False))

 if short_mode:
 reply = f"[turn {turn}] OK: {message}"
 else:
 reply = f"[turn {turn}] I received: '{message}'. (No short_mode preference set.)"

 return {"turn": turn, "reply": reply}

def build_graph():
 g = StateGraph(SupportState)
 g.add_node("load_prefs", load_prefs)
 g.add_node("reply_or_remember", reply_or_remember)
 g.add_edge(START, "load_prefs")
 g.add_edge("load_prefs", "reply_or_remember")
 g.add_edge("reply_or_remember", END)
 return g

def main() -> None:
 store = InMemoryStore() # simplest store: put/get by key

 # Use an in-memory SQLite checkpoint so every run starts clean.
 with SqliteSaver.from_conn_string(":memory:") as saver:
 graph = build_graph().compile(checkpointer = saver, store = store)

 user_id = "user-42"

 # Thread A: save long-term preference (store write) + checkpoint turn=1
 cfg_a = {"configurable": {"thread_id": "thread-A", "user_id": user_id}}
 print("--- Thread A / Run 1 ---")
 print(graph.invoke({"message": "remember: short"}, config = cfg_a)["reply"])

 # Same thread A: short-term memory (checkpoint) continues turn counter
 print("\n--- Thread A / Run 2 (same thread_id) ---")
 print(graph.invoke({"message": "Hello again"}, config = cfg_a)["reply"])

 # Thread B: new thread_id => no checkpoint history (turn resets),
 # but long-term preference is still applied (store read)
 cfg_b = {"configurable": {"thread_id": "thread-B", "user_id": user_id}}
 print("\n--- Thread B / Run 1 (new thread_id) ---")

```

```

print(graph.invoke({"message": "How do I reset my password?"},
 config = cfg_b)["reply"])

if __name__ == "__main__":
 main()

```

ch\_08\scr\long\_short\_term\_memory.py

This script sets up a tiny support workflow that deliberately mixes two kinds of memory: thread-scoped state that survives within a single thread, and user-scoped preferences that survive across many threads. The interesting part is how the graph moves data between these layers, and how the runtime wiring (checkpointer and store) makes the behavior repeatable. The workflow has two nodes.

The first node, `load_prefs`, runs at the start of every invocation. Its job is to pull durable user preferences from the shared store and inject them into the current thread state under `prefs`. It looks up `user_id` and `thread_id` from the runnable configuration, not from the state. That's significant: it means identity and routing live outside the conversational payload, so you can resume or replay runs without needing to re-send "who this is" inside message text. If there is no `user_id`, the node returns an empty `prefs` dict, so downstream logic stays simple and doesn't need special cases. When a `user_id` exists, the node reads from a stable namespace keyed by `(user_id, "profile")` and a fixed key ("preferences"), then returns those preferences as part of the state update. This is the bridge from long-lived records into the per-thread working context.

The second node, `reply_or_remember`, does two things: it advances a per-thread turn counter and either writes a preference or generates a reply using previously loaded preferences. The turn counter comes from `state.get("turn")`. If the thread is resumed, that value is already present because the checkpointer rehydrates the state; if the thread is new, it's missing and the code starts from zero. Incrementing turn and returning it is the key "short-term" write: it ensures the updated turn becomes part of the checkpointed state for the thread.

The long-term write is conditional. If the incoming message begins with "remember:", the node parses a preference ("short" toggles `short_mode`) and persists it to the shared store under the user's namespace. That write is intentionally independent of `thread_id`: once saved, any future thread for the same user can read it in `load_prefs`.

The main function demonstrates the payoff with three runs. Thread A run 1 writes a durable preference and checkpoints `turn=1`. Thread A run 2 reuses the same `thread_id`, so the turn counter continues to 2 without extra bookkeeping, and the reply is shaped by the stored preference. Thread B starts with a new `thread_id`, so the turn resets, but the preference still applies because it is loaded from the shared store.

#### 8.2.4 Backend choices for persistence

At this point the key conceptual split is already clear: checkpointing backends serve thread-scoped durability, and stores serve cross-thread, application-scoped memory. This section is not about re-defining those layers; it is about choosing concrete backends for each layer, based on the operational realities they impose.

**Checkpoint backends:** choose for concurrency first, then durability

For checkpoints, the question is less "what data model do I like?" and more "how many concurrent threads and workers must safely resume the same run?"

An in-memory checkpointer is ideal for learning and unit tests: it makes the persistence mechanism visible without introducing infrastructure, but it cannot survive process restarts and cannot support multi-instance deployments.

SQLite is the next step up for local development. It gives you persistence across restarts with almost zero operational overhead, and it is good for a single developer process in lightweight, synchronous demos and small projects; it is not designed to scale to multiple threads. The trap is treating it as a production default. A SQLite-backed checkpointer is a poor fit once you need multi-threaded or multi-worker concurrency; treat it as a lightweight persistence option for demos/small projects rather than a production default. It is best thought of as “durable enough to demonstrate the mechanism”, not “durable enough to run a fleet.”

When you expect many concurrent threads, multiple worker processes, or stateless replicas behind a load balancer, a server-grade database is the typical checkpoint default. In practice, PostgreSQL is the common “first production” step because it supports concurrent clients, has mature operational tooling, and aligns with the basic access pattern of checkpointing (frequent writes, occasional resumes, and administrative inspection). LangGraph’s docs and reference integrations reflect this split: SQLite is positioned for lightweight demos/small projects, while Postgres is the recommended production checkpointer.

Redis can also be a viable checkpoint backend when you specifically want very fast reads/writes and you already operate Redis reliably. The important detail is that Redis checkpointing is typically provided as a dedicated integration package (for example, langgraph-checkpoint-redis) rather than “just turn on Redis” with no other decisions. That integration provides Redis-based checkpoint savers, and it comes with Redis-side requirements: the Redis integration requires RedisJSON and RediSearch (and you typically run a one-time `.setup()` to create the needed indices).

**Stores:** For long-term stores, you are choosing a persistent store interface organised by namespaces and keys, with optional search capabilities. Exact lookups (namespace + key) are the baseline. Search is an additional capability: metadata filtering, text search, and/or semantic similarity. In many implementations, semantic similarity is disabled unless you configure an index.

- Relational stores (PostgreSQL, MySQL) fit when your “memory items” have stable fields you want to validate, join, and report on. They shine at exact lookups, filters, constraints (uniqueness, foreign keys), and auditable change history. If you add semantic similarity, you are adding a vector indexing story on top of the relational core (either via an extension, a sidecar index, or an external vector DB).
- Document stores (MongoDB and similar) fit when your memory items are naturally JSON-shaped and evolve over time. They make it easy to store heterogeneous records under one namespace without schema migrations for every small change. They handle exact lookups and metadata filtering well; semantic similarity again requires an explicit indexing configuration, either built-in or external.
- Graph stores (Neo4j and similar) fit when long-term memory is mostly relationships: “user belongs to org,” “ticket references product,” “policy applies to region,” “this runbook depends on that component.” They are excellent for traversal queries (“find related entities within 2 hops”) and for enforcing relationship semantics. They can still implement namespace+key lookups, but their real value appears when your agent needs relationship-aware retrieval rather than “top-k similar chunks.”

- Vector stores (Pinecone, Qdrant, Milvus, etc.) fit when semantic similarity is the primary retrieval mode. They are optimised for “find nearest neighbours” over embeddings, often with metadata filtering. The trade-off is that “exact record storage” and “transactional updates” are usually not their strongest suit, so teams often pair a vector store with a document or relational store as the source of truth.

A practical way to decide:

1. Durability and governance: retention, deletion, provenance, and audit. Relational systems usually make governance easiest; document and graph can be good with discipline; vector stores often require pairing with a source-of-truth store for full governance.
2. Retrieval needs: do you mostly need exact lookups, filtered queries, relationship traversal, or semantic similarity? Pick the native strength first; bolt-ons (like vectors) add operational moving parts.
3. Update patterns: are records frequently edited, appended, or versioned? Relational handles updates and constraints cleanly; document handles evolving shapes; vector requires careful re-embedding and index maintenance.
4. Operational fit: deployability, backups, migrations, monitoring, and cost. A “perfect” store you can’t run reliably becomes an incident generator with a nice API.

Your long-term store choice is really two decisions: (1) the system of record for durable facts (relational/document/graph), and (2) the retrieval accelerator for “finding” those facts (often vector search). Combining them intentionally beats pretending one backend will do everything well.

## 8.3 CONTROL-FLOW PATTERNS IN GRAPHS: LOOPS, BRANCHES, SUBGRAPHS

Section 8.1 already covered fixed branching, fan-out/fan-in, and basic routing. This section focuses on three patterns that tend to appear only once you try to ship a real system: dynamic fan-out (map-reduce) when the number of tasks is only known at runtime, pause-and-resume flows for human approval, and graph composition with subgraphs. These are first-class concepts in LangGraph: the Send API for map-reduce style fan-out, interrupts for pause-and-resume (human-in-the-loop), and subgraphs for composition.

### 8.3.1 Dynamic fan-out with Send

Fixed branches are a design-time decision: you wire “A → B and C” and you always run both. Map-reduce is a runtime decision: “for each item in this list, run a check node”, where the list length can vary per request. LangGraph supports this by allowing a routing function (used in a conditional edge) to return a list of `Send` objects. Each `Send` specifies (1) the target node name and (2) a custom state object (often a per-item subset or partial state) that will be provided to that node for that dispatched task. The important design point is that each mapped task can produce an update to the same accumulator key (for example, a list of per-item results). If you want aggregation rather than last-write-wins, that state key must be defined with a reducer. For lists, a common reducer is `operator.add`, which concatenates list updates (so each task typically returns a one-item list like `[result]` that gets accumulated).

In the e-commerce assistant scenario we have been using throughout book, a common “real world” step is checking something for every item in an order: each line can have different constraints (delivery date, category restrictions, opened/not opened), and the number of lines is not fixed. That is where a graph wired with a fixed set of branches stops being expressive: you do not want to hard-code “two” or “three” parallel checks, because tomorrow an order might have one line or fifteen. In LangGraph, this is the point where you switch from design-time fan-out to runtime fan-out: the graph generates one task per order line, accumulates the per-line results via a reducer on the shared state, and (optionally) runs a downstream aggregator node that turns the accumulated results into a single decision the assistant can explain.

```
from __future__ import annotations
import operator
from typing import Annotated, Literal
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.types import Send

class OrderLine(TypedDict):
 sku: str
 delivered_days_ago: int
 opened: bool
 category: Literal["apparel", "electronics", "consumable"]

class LineResult(TypedDict):
 sku: str
 eligible: bool
 reason: str

class ReturnsState(TypedDict, total = False):
 # Inputs
 order_id: str
 lines: list[OrderLine]

 # Per-task input for mapped nodes (each task gets one line)
 line: OrderLine

 # Aggregated outputs (many tasks update the same key; reducer required)
```

```

line_results: Annotated[list[LineResult], operator.add]
Final outputs
decision: Literal["approve_return", "request_more_info", "reject_return"]
answer: str

def prepare(state: ReturnsState) -> dict:
 # Anchor node: in a real system, this could fetch the order lines.
 return {}

def fanout_lines(state: ReturnsState):
 # Map step: create one task per line item.
 return [Send("check_line", {"line": line}) for line in state["lines"]]

def check_line(state: ReturnsState) -> dict:
 # Deterministic policy checks for a runnable example.
 line = state["line"]
 sku = line["sku"]

 if line["delivered_days_ago"] > 30:
 return {"line_results": [{"sku": sku, "eligible": False, "reason": "outside 30-day window"}]}
 if line["category"] == "consumable":
 return {"line_results": [{"sku": sku, "eligible": False, "reason": "consumables not eligible"}]}
 if line["category"] == "electronics" and line["opened"]:
 return {"line_results": [{"sku": sku, "eligible": False, "reason": "opened electronics not eligible"}]}
 return {"line_results": [{"sku": sku, "eligible": True, "reason": "eligible"}]}

def reduce_and_decide(state: ReturnsState) -> dict:
 results = state.get("line_results", [])

 if not results:
 return {
 "decision": "request_more_info",
 "answer": "I need the order lines to assess return eligibility."
 }

 any_eligible = any(r["eligible"] for r in results)
 any_ineligible = any(not r["eligible"] for r in results)

 if any_eligible and any_ineligible:
 return {
 "decision": "request_more_info",
 "answer": (
 "Some items are eligible and some are not. "
 "Confirm which SKU(s) you want to return: "
 + ", ".join(r["sku"] for r in results)
)
 }

 if any_ineligible and not any_eligible:
 return {
 "decision": "reject_return",
 "answer": "No items are eligible: " + ".join(f"{{r['sku']}} {{r['reason']}}"
 for r in results)
 }

 return {
 "decision": "approve_return",
 "answer": "All checked items are eligible. Start the return from your orders page."
 }

builder = StateGraph(ReturnsState)

builder.add_node("prepare", prepare)
builder.add_node("check_line", check_line)
builder.add_node("reduce_and_decide", reduce_and_decide)
builder.add_edge(START, "prepare")

Conditional edge returning a list[Send] implements map-reduce fan-out.
builder.add_conditional_edges("prepare", fanout_lines)

Reduce step: aggregate all per-line results into one decision.
builder.add_edge("check_line", "reduce_and_decide")
builder.add_edge("reduce_and_decide", END)

graph = builder.compile()

if __name__ == "__main__":
 out = graph.invoke(
 {
 "order_id": "ORD-1001",

```

```

 "lines": [
 {"sku": "TSHIRT-RED-M", "delivered_days_ago": 12, "opened": True,
 "category": "apparel"},

 {"sku": "HEADPHONES-X", "delivered_days_ago": 9, "opened": True,
 "category": "electronics"},

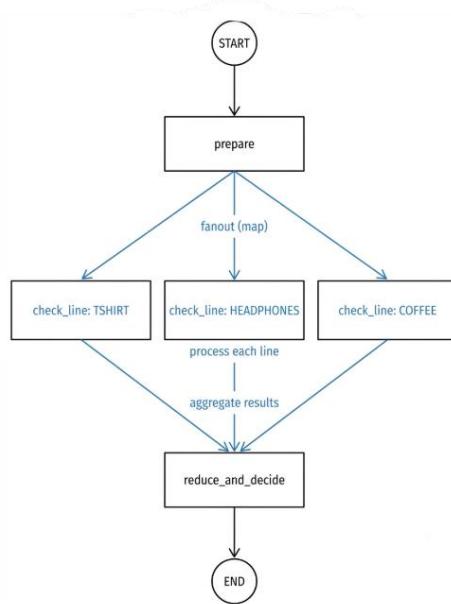
 {"sku": "COFFEE-BEANS", "delivered_days_ago": 5, "opened": False,
 "category": "consumable"}
],
 "line_results": [],
 "decision": "request_more_info",
 "answer": ""
}

print("decision:", out["decision"])
print("line_results:", out["line_results"])
print("answer:", out["answer"])

```

ch\_08\scr\fant\_out.py

The fan-out happens in `fanout_lines`, which is wired as the routing function for the conditional edge leaving `prepare`. Instead of returning a single next node name, this function returns a list of `Send` instructions—one per order line in `state["lines"]`. Each `Send("check_line", {"line": line})` dispatches a separate task into the same `check_line` node but with a task-specific mini-state containing only the current line. This is why the shared state type includes both `lines` (the full input list) and `line` (the per-task input): each parallel execution of `check_line` reads `state["line"]` and produces exactly one structured result for that SKU.



Those parallel tasks all write to the same key: `line_results`. Because multiple tasks update that key concurrently, the state declares `line_results` with a reducer (`Annotated[..., operator.add]`). The important consequence is that each `check_line` task returns `{"line_results": [<one LineResult>]}` as a list, so the reducer can concatenate all per-line lists into a single aggregated list. After fan-out completes, the graph flows to `reduce_and_decide`, which reads the accumulated `line_results` and makes a single decision and answer based on the combined outcomes.

### 8.3.2 Durable execution: why “in-memory persistence” is not enough

`InMemorySaver` (called `MemorySaver` in some older examples) works well for demos and local experiments because it stores checkpoints in-process (in memory), so they disappear

when the process exits. It is not sufficient for production deployments that scale horizontally or run in serverless environments: you cannot rely on a later request being routed to the same instance that still has the in-memory state. In practice, the service layer should remain stateless, and checkpoints should be written to a shared, durable store such as a database. For that, use a database-backed checkpoint (for example, the SQLite checkpoint from the langgraph-checkpoint-sqlite package for simple setups, or PostgresSaver/AsyncPostgresSaver via langgraph-checkpoint-postgres for production), or implement a custom checkpoint by following the BaseCheckpointSaver interface.

### Example: compiling a graph with a durable checkpoint (SQLite)

This example mirrors the “production shape” even if you run it locally: you compile the graph with a database-backed checkpoint and, on every invoke/stream, pass config = {"configurable": {"thread\_id": "..."} } so the checkpoint can load and save the thread’s checkpoints.

Important nuance: to resume from an existing checkpoint, call invoke/stream with input=None (or a Command(resume=...)) when using interrupts) and the same thread\_id, so LangGraph loads the latest checkpoint and continues from the saved ‘next’ node. If you pass a non-None input, LangGraph treats it as a new invocation and will start again from the entry node, mapping that new input into state. Please note that to run this example you need to install langgraph-checkpoint-sqlite library:

```
(.venv) C:\Users\yourname\llm-app> python -m pip install langgraph-checkpoint-sqlite
```

```
from __future__ import annotations
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.sqlite import SqliteSaver

class CounterState(TypedDict):
 counter: int

def increment(state: CounterState) -> dict:
 # This will now correctly pull the existing value from the DB
 return {"counter": state["counter"] + 1}

builder = StateGraph(CounterState)
builder.add_node("increment", increment)
builder.add_edge(START, "increment")
builder.add_edge("increment", END)

with SqliteSaver.from_conn_string("checkpoints_8_3.db") as saver:
 graph = builder.compile(checkpointer = saver)
 thread_cfg = {"configurable": {"thread_id": "counter-thread"}}

 # First run: Initialize counter at 0. Result: 1
 print("Run 1:", graph.invoke({"counter": 0}, config = thread_cfg))

 # Second run: Pass an empty dict.
 # LangGraph loads 'counter': 1 from the DB. Result: 2
 print("Run 2:", graph.invoke({}, config = thread_cfg))
```

ch\_08\scr\database\_checkpoint.py

```
(.venv) C:\Users\yourname\llm-app> python -m src.database_checkpoint
Run 1: {'counter': 1}
Run 2: {'counter': 2}
```

The graph is a minimal “persistent counter” workflow. State is a TypedDict with a single field, counter. There is one node, increment, which reads counter from state and returns an updated value that is exactly +1.

Persistence is enabled by compiling the graph with a `SqliteSaver` checkpointer pointing at `checkpoints_8_3.db`. The config passed to `invoke` includes `configurable.thread_id = "counter-thread"`, which makes both calls refer to the same logical run thread. On the first run, the input explicitly supplies counter: 0, so the node returns counter: 1 and that resulting state is checkpointed in SQLite.

On the second run, the input is an empty dict. Because the same `thread_id` is used and a prior checkpoint exists for that thread in the SQLite backend, the graph can resume from the last persisted state rather than starting from the empty input alone. That is why increment sees counter = 1 and returns counter = 2, producing the printed result for “Run 2”.

### 8.3.3 Resumable human approval using `interrupt()` and `Command(resume=...)`

Some control flow is not “pick a path and continue”. It is “pause here, wait for an external decision, then continue later”. LangGraph’s `interrupt()` is designed for that: when called inside a node it pauses execution by raising a resumable interrupt that the runtime catches, saves state via the configured checkpointer, and surfaces the interrupt payload to the caller. Resuming is done by invoking the graph again with `Command(resume=...)`, using the same `thread_id` so the checkpointer can load the saved state. (Interrupts require a checkpointer.) The resume value becomes the return value of `interrupt()` inside the node. When resumed, the graph restarts execution from the start of that node (re-executing the node’s logic), using the persisted state from the checkpoint. This pattern is the backbone of approval gates (refunds, destructive actions), review-and-edit workflows, and tool-call supervision.

The objective of this example is to show a trading assistant that refuses to execute a high-notional trade immediately, pauses to obtain an external approval decision, and then resumes the same trade request by re-running the paused approval node (from the start of that node) with the stored state, so it either executes or rejects the trade while preserving the original intent and the approval outcome.

```
from __future__ import annotations
from typing import Literal, Optional
from typing_extensions import TypedDict
from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import StateGraph, START, END
from langgraph.types import Command, interrupt

class TradeState(TypedDict):
 symbol: str
 side: Literal["BUY", "SELL"]
 qty: int
 notional_usd: int
 status: Optional[Literal["pending", "approved", "rejected", "executed"]]

 def pretrade_checks(state: TradeState) -> dict:
 # Deterministic placeholder: in real life this could compute risk, slippage, limits,
 # etc.
 return {"status": "pending"}

 def approval_gate(state: TradeState) -> Command[Literal["execute", "reject"]]:
 requires_approval = state["notional_usd"] >= 100_000

 if not requires_approval:
 return Command(update = {"status": "approved"}, goto = "execute")

 approved = interrupt(
 {
 "question": "Approve trade execution?",
 "symbol": state["symbol"],
 "side": state["side"],
 "qty": state["qty"],
 "notional_usd": state["notional_usd"]
 }
)
 if approved:
```

```

 return Command(update = {"status": "approved"}, goto = "execute")

 return Command(update = {"status": "rejected"}, goto = "reject")

def execute(state: TradeState) -> dict:
 # Deterministic placeholder for an execution step.
 return {"status": "executed"}

def reject(state: TradeState) -> dict:
 return {"status": "rejected"}

builder = StateGraph(TradeState)
builder.add_node("pretrade_checks", pretrade_checks)
builder.add_node("approval_gate", approval_gate)
builder.add_node("execute", execute)
builder.add_node("reject", reject)

builder.add_edge(START, "pretrade_checks")
builder.add_edge("pretrade_checks", "approval_gate")
builder.add_edge("execute", END)
builder.add_edge("reject", END)

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

if __name__ == "__main__":
 # thread_id is required for the checkpointer to store and reload state for resumes.
 config = {"configurable": {"thread_id": "trade-thread-42"}}

 first = graph.invoke(
 {
 "symbol": "AAPL",
 "side": "BUY",
 "qty": 500,
 "notional_usd": 125_000,
 "status": None
 },
 config = config
)

 print("interrupt_payload:", first["__interrupt__"])

 # Resume with the human's decision using the same thread_id.
 resumed = graph.invoke(Command(resume = True), config = config)
 print("final_status:", resumed["status"])

```

ch\_08\scr\stop\_resume.py

The control-flow pivot in this program is the `approval_gate` node. It decides whether the trade can proceed immediately or must pause for an external decision. The decision is driven by `requires_approval`: if the notional is below the threshold, the node skips interruption entirely, updates the state to `approved`, and routes straight to `execute`.

For large trades, the node calls `interrupt(...)` with a payload that contains the exact trade context the approver needs (symbol, side, quantity, notional, and a human-readable question). That call is the point where the graph halts and yields control back to the caller. Nothing after the `interrupt(...)` line runs during this first invocation; instead, the invocation returns an output that includes the interrupt payload (printed from `first["__interrupt__"]`).

The second invocation demonstrates how the decision is fed back into the paused execution. It calls `graph.invoke(Command(resume=True), config=config)` using the same `thread_id`, so the checkpointer can reload the saved state and continue the pending run. Inside `approval_gate`, the `approved` variable receives the resumed value, and the node re-executes from the top, taking the “`approved → execute`” branch (or “`rejected → reject`” if `resume` were false).

### 8.3.4 Subgraphs for maintainable domain decomposition

Subgraphs are not just “code reuse”. In LangGraph they are a documented execution construct: a graph used as a node inside another graph. This is how you keep a large system

legible when it grows beyond a handful of nodes. There are two documented ways to integrate a parent graph and a subgraph.

- Invoke a graph from a node (subgraph can use a different state schema; no shared state keys required). This is an adapter-style integration: the parent node translates between the parent state and the subgraph state, then transforms the subgraph output back into a parent-state update. In this mode, the subgraph can have a completely different schema from the parent (no shared state keys).
- Add a graph as a node (parent and subgraph communicate via one or more shared state keys). This is tighter coupling. The compiled subgraph is added directly as a node in the parent. Parent and subgraph communicate through shared state keys, while the subgraph may also have its own private keys. The Subgraphs guide calls this out explicitly and frames it as a choice about how parent and subgraph communicate.

In the following example, a customer contacts support with a question. If the question is about returns (for example, a damaged item, a sizing issue, or a general “how do I return this?”), the conversation is handed off to a returns specialist who provides the appropriate return instructions for that situation. If the question is not about returns, the customer receives a general support prompt asking what happened so the agent can help with the right area.

```
from __future__ import annotations
from typing import Literal
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END

Subgraph schema (private to the returns specialist)
class ReturnsSubState(TypedDict):
 question: str
 outcome: str

 def returns_specialist(state: ReturnsSubState) -> dict:
 # Deterministic rules for a runnable example.
 q = state["question"].lower()
 if "damaged" in q:
 return {"outcome": "Returns: start a damaged-item return and attach photos."}
 if "does not fit" in q or "size" in q:
 return {"outcome": "Returns: start a size/fit return; confirm the item is unused."}
 return {"outcome": "Returns: start a standard return from your orders page."}

 returns_builder = StateGraph(ReturnsSubState)
 returns_builder.add_node("returns_specialist", returns_specialist)
 returns_builder.add_edge(START, "returns_specialist")
 returns_builder.add_edge("returns_specialist", END)
 returns_graph = returns_builder.compile()

Parent schema (public support interface)
class SupportState(TypedDict):
 question: str
 route: Literal["returns", "other"]
 answer: str

 def triage(state: SupportState) -> dict:
 q = state["question"].lower()
 route: Literal["returns", "other"] = "returns" if any(k in q for k in ["return", "refund", "exchange"]) else "other"
 return {"route": route}

 def route_after_triage(state: SupportState) -> Literal["call_returns", "fallback"]:
 return "call_returns" if state["route"] == "returns" else "fallback"

 def call_returns(state: SupportState) -> dict:
 child_out = returns_graph.invoke({"question": state["question"], "outcome": ""})
 return {"answer": child_out["outcome"]}

 def fallback(state: SupportState) -> dict:
 return {"answer": "Support: I can help with orders, returns, billing, and technical issues. What happened?"}

parent_builder = StateGraph(SupportState)
parent_builder.add_node("triage", triage)
parent_builder.add_node("call_returns", call_returns)
```

```
parent_builder.add_node("fallback", fallback)
parent_builder.add_edge(START, "triage")
parent_builder.add_conditional_edges(
 "triage",
 route_after_triage,
 {"call_returns": "call_returns", "fallback": "fallback"})
)
parent_builder.add_edge("call_returns", END)
parent_builder.add_edge("fallback", END)
support_graph = parent_builder.compile()
if __name__ == "__main__":
 out = support_graph.invoke({"question": "How do I return an item that arrived
damaged?", "route": "other", "answer": ""})
 print(out["answer"])
```

ch\_08\scr\sub\_graph.py

This code builds a parent support workflow that delegates one specific domain—returns handling—to a dedicated subgraph. The returns subgraph is compiled as an independent runnable with its own private state shape. Its single node, `returns_specialist`, implements deterministic routing rules over the customer's question and produces a normalized outcome. The subgraph is wired as a straight-through flow: START → `returns_specialist` → END, then compiled so it can be invoked like a function.

The parent graph owns the public support state and uses `triage` to classify the request by scanning for return-related keywords. Instead of branching inline, the parent's conditional edge calls `route_after_triage`, which returns a symbolic next step. That decision drives execution to either `call_returns` or `fallback`. `call_returns` is the key subgraph integration point: it adapts the parent state into the subgraph's expected input ('question' plus an initial 'outcome'), invokes the compiled `returns_graph`, then maps the child result back into the parent's answer. This keeps returns logic isolated while still letting the parent control orchestration and end-to-end output.

## 8.4 ERROR-HANDLING PATHS AND RECOVERY INSIDE GRAPHS

LangGraph workflows operate at the fault line between your application and everything it depends on: model endpoints, tool calls, external APIs, and stateful backends. In that boundary layer, failures are routine. A graph that assumes a single happy path becomes brittle, hard to diagnose, and risky to resume. LangGraph’s design guidance treats errors as part of the flow, so failure handling remains visible in the same control structure as the primary logic.

Recovery design works best when you classify the failure first and then choose a graph behaviour that matches it. LangGraph distinguishes four common categories. Transient errors are short-lived infrastructure problems such as timeouts, network glitches, and rate limits, and they generally call for retries. LLM-recoverable errors occur when a step produces something the model can potentially correct on a retry loop—commonly tool failures and parsing/validation issues. The pattern is to store the failure information in state so the model can see what went wrong and try again. User-fixable errors arise when required input is missing or ambiguous, and the graph should pause for external input rather than guessing. Unexpected errors cover bugs or unsafe-to-handle conditions and should surface clearly so they can be debugged and corrected.

### 8.4.1 Transient errors: local retries, then a controlled fallback

Transient failures are best handled at the point where they occur. You can attach a `RetryPolicy` to a node when you register it (for example via the node’s `retry_policy` setting). The runtime will automatically re-run that node when it raises an exception that matches the retry policy (by default, it is tuned for specific transient/network-style failures). This keeps the main path clean while still giving external dependencies a chance to recover.

The important design choice is scope. A retry policy is for failures that are plausibly temporary like timeouts, network interruptions, rate limiting, and similar transport-level problems. It is not a substitute for business-rule validation. If the failure is logical (missing required input, policy violations, structurally invalid requests), repeating the same node call will predictably fail again and will only add latency and cost.

It also matters what happens when retries are exhausted. The built-in retry mechanism is designed to retry and then raise when it cannot succeed. If you want the workflow to continue on a well-defined recovery path instead of failing the run, handle the error inside the node and convert it into state (for example, an error object with type/message/context). Then route explicitly based on that state—either with conditional edges or by returning a `Command` that both updates state and chooses the next node. That fallback can downgrade gracefully (for example, return a safe explanatory answer) while preserving enough error detail for tracing and monitoring.

The following example demonstrates LangGraph’s `RetryPolicy` and what happens when the retry budget is exhausted. The graph itself does not implement an in-graph fallback branch; after `max_attempts` is reached, `graph.invoke` raises the final exception. The example then shows a common integration pattern: the application wraps `graph.invoke` in `try/except` and returns a safe fallback response when the workflow fails after retries. If you

want the workflow to continue on an explicit fallback path inside the graph, catch exceptions inside the node, write an error object into state, and route based on that state—via conditional edges or via Command (when you want to both update state and redirect control flow).

```

from __future__ import annotations
import time
from typing_extensions import TypedDict
from langgraph.graph import START, END, StateGraph
from langgraph.types import RetryPolicy

1. Define State
class SupportState(TypedDict, total = False):
 question: str
 answer: str

2. The Fragile Node
def search_docs(state: SupportState) -> dict:
 question = state.get("question", "Unknown")
 print(f"--- Attempting search_docs for: '{question}' ---")
 raise TimeoutError("Backend timeout")

3. Build the Graph
builder = StateGraph(SupportState)

builder.add_node(
 "search_docs",
 search_docs,
 retry_policy = RetryPolicy(
 max_attempts = 3,
 initial_interval = 1.0,
 max_interval = 5.0,
 backoff_factor = 2.0,
 jitter = False,
 # Change: explicitly retry on TimeoutError
 retry_on = lambda exc: isinstance(exc, TimeoutError)
)
)

builder.add_edge(START, "search_docs")
builder.add_edge("search_docs", END)

graph = builder.compile()

4. Execution
if __name__ == "__main__":
 inputs = {"question": "Retry Demo"}

 try:
 graph.invoke(inputs)
 except Exception:
 print("\n[Behavior 1 Complete] LangGraph exhausted all 3 attempts.")
 print("[Behavior 2 Triggered] Providing Fallback Answer.")
 print("Final Output: Manual Fallback: The search service is currently unavailable.")

```

ch\_08\scr\node\_retry.py

```
(.venv) C:\Users\alexc\llm-app\ch_08> python -m src.node_retry

--- Attempting search_docs for: 'Retry Demo' ---
--- Attempting search_docs for: 'Retry Demo' ---
--- Attempting search_docs for: 'Retry Demo' ---

[Behavior 1 Complete] LangGraph exhausted all 3 attempts.
[Behavior 2 Triggered] Providing Fallback Answer.
Final Output: Manual Fallback: The search service is currently unavailable.
```

The graph is a deliberately “brittle” one-node workflow used to demonstrate retry and failover behavior in isolation. Execution is linear (START → search\_docs → END), so the only way to reach END is for the search\_docs node to return a state update.

`search_docs` always fails. Each invocation prints the current question and raises `TimeoutError`. Because the node raises before returning, it never produces an “answer” update, which forces the runtime into the retry path.

Retry is configured on the node itself via `RetryPolicy`. The policy explicitly retries only when the thrown exception is a `TimeoutError` (`retry_on_lambda`), so unrelated exceptions would bypass retries and fail immediately. `max_attempts=3` caps the total number of executions of `search_docs` within a single `graph.invoke` call. With jitter disabled, the delays are deterministic: the first failure triggers a 1-second wait (`initial_interval`), the second failure multiplies the delay by `backoff_factor=2` to 2 seconds, and then the third attempt runs. `max_interval` is a ceiling that is not reached in this schedule.

After the third failure, `LangGraph` stops retrying and re-raises the last exception out of `graph.invoke`. There is no in-graph recovery branch, so failover is implemented outside the graph: the main block wraps `graph.invoke` in `try/except` and, on failure after retries, returns a manual fallback message to the caller.

#### 8.4.2 LLM-recoverable errors: validate, repair, and loop with a stop condition

A different class of failure happens when the model “responds” but the response is unusable for the workflow: invalid JSON, missing required fields, or a tool call that fails schema validation. A common approach is to make the error itself visible to the model, then run a controlled repair step rather than blindly retrying the same prompt. One practical framing is “convert the exception to text or structured state so downstream steps can decide how to recover,” because it keeps the workflow moving and makes failures auditable.

`LangGraph` lets you model a repair loop as an explicit cycle in the graph (for example by routing from a validation/repair node back to the extraction node), rather than hiding retries inside ad hoc exception handling. The `Command` type can be used when a node needs to both update state and explicitly route execution to a specific next node (for example, “repair” vs “retry”). Termination is typically modeled by routing to the graph’s `END` (or an `END` edge), rather than treating “end” as just another named node.

This example is intended to demonstrate an LLM-recoverable error path where “bad-but-fixable” structured output is treated as part of normal control flow. The workflow tries to extract required fields from a JSON-like payload, detects that the output fails validation (for example, a missing required field), records the failure in state, and then takes a dedicated recovery path that attempts to produce a corrected payload. The graph can then re-attempt extraction and complete only when the output passes validation. The repair loop should be explicitly bounded (for example via an attempt counter or max-retry limit in state) rather than hidden in ad hoc exception handling.

```
from __future__ import annotations
import json
from typing_extensions import TypedDict
from langgraph.graph import START, END, StateGraph
from langgraph.types import Command

1. Define the State schema
class TicketState(TypedDict, total = False):
 message: str
 extracted_json: str
 extracted: dict
 parse_error: str
```

```

 attempts: int

2. Define the extraction logic with error handling
def extract_fields(state: TicketState) -> Command:
 attempts = int(state.get("attempts", 0)) + 1
 raw = state.get("extracted_json") or ""

 try:
 data = json.loads(raw)
 # Validation check for required fields
 if "account_id" not in data:
 raise ValueError("Missing required field: account_id")

 # On success, update state and end the graph
 return Command(
 update = {"attempts": attempts, "extracted": data, "parse_error": ""},
 goto = END
)
 except Exception as e:
 # On failure, store the error and route to the repair node
 print(f"--- Extraction Failed (Attempt {attempts}): {e} ---")
 return Command(
 update = {"attempts": attempts, "parse_error": str(e)},
 goto = "repair_output"
)

3. Define the repair logic
def repair_output(state: TicketState) -> Command:
 # Simulates repairing invalid JSON for the next extraction attempt.
 parse_error = state.get("parse_error") or "unknown error"
 print(f"--- Repairing output after error: {parse_error} ---")

 # Placeholder for a repaired JSON string that includes 'account_id'
 repaired = json.dumps({"account_id": "12345", "intent": "password_reset"})

 # Return to the extraction node with updated data
 return Command(
 update = {"extracted_json": repaired, "parse_error": f"repaired after: {parse_error}"},
 goto = "extract_fields"
)

4. Build the Graph
builder = StateGraph(TicketState)
builder.add_node("extract_fields", extract_fields)
builder.add_node("repair_output", repair_output)

builder.add_edge(START, "extract_fields")
The 'Command' object in nodes handles the routing, so explicit edges between
extract_fields and repair_output are managed by the 'goto' parameter.

graph = builder.compile()

5. Execute the Graph
if __name__ == "__main__":
 # Input with malformed/incomplete JSON to trigger the repair cycle
 initial_state = {
 "message": "I need to reset my password",
 "extracted_json": '{"intent": "password_reset"}', # Missing account_id
 "attempts": 0
 }

 print("Starting Recovery Graph...")
 final_state = graph.invoke(initial_state)

 print("\n--- Final State ---")
 print(f"Attempts: {final_state['attempts']}")
 print(f"Extracted Data: {final_state['extracted']} ")
 print(f"Last Parse Error Status: {final_state['parse_error']} if final_state['parse_error'] else 'None' ")

```

ch\_08\scr\recoverable\_error.py

```

(.venv) C:\Users\alexc\llm-app\ch_08> python -m src.node_recoverable_error
Starting Recovery Graph...
--- Extraction Failed (Attempt 1): Missing required field: account_id ---
--- Repairing output after error: Missing required field: account_id ---

--- Final State ---
Attempts: 2
Extracted Data: {'account_id': '12345', 'intent': 'password_reset'}

```

Last Parse Error Status: None

The graph implements a recoverable-error loop around JSON extraction and validation. The `extract_fields` node increments an `attempts` counter, reads the current `extracted_json` string from state, and tries to parse it as JSON. After parsing, it performs a business validation step by requiring `account_id`. If parsing succeeds and the required field is present, the node returns a Command that writes the successful `extracted` dict into state, clears `parse_error`, records the attempt count, and routes execution directly to END, terminating the workflow.

Any failure—invalid JSON, missing `account_id`, or any other exception—falls into the `except` block. Instead of crashing the run, the node logs the failure, stores a stringified error message in `parse_error`, updates the attempt count, and routes to the dedicated recovery node via `goto="repair_output"`.

The `repair_output` node reads the last `parse_error` for diagnostics, then overwrites `extracted_json` with a “repaired” payload that now includes `account_id`. It also annotates `parse_error` to indicate a repair happened, and routes back to `extract_fields`. This makes recovery explicit: state captures what failed, the repair step mutates the input, and the graph re-attempts extraction until validation passes.

#### 8.4.3 User-fixable errors: interrupts and human-in-the-loop recovery

Some errors should not be repaired automatically. If a required identifier is missing, the user must choose. If an action is high impact (for example, issuing a refund or placing a trade), an explicit approval step is safer than guessing.

LangGraph interrupts are designed for this kind of human-in-the-loop pause-and-resume control flow. Calling `interrupt()` inside a node pauses execution and returns the interrupt payload to the caller under the `__interrupt__` field (the payload must be JSON-serializable). Resuming happens by invoking the graph again with the same `thread_id` and a `Command(resume=...)` value; that `resume` value becomes the return value of `interrupt()` inside the node. Interrupts have specific behavioural rules because, on resume, the interrupted node re-runs from the beginning and any code before `interrupt()` executes again.

This example demonstrates a user-fixable error path handled with a human-in-the-loop pause and resume. The workflow refuses to guess when a required identifier is missing. Instead, it stops at a defined point, returns a structured prompt asking for the missing `account_id`, and persists the graph state via a checkpoint so nothing is lost while waiting for the missing input. Once the caller provides the `account_id`, the workflow resumes: the interrupted node re-executes from the start, `interrupt()` returns the supplied value, and execution proceeds to completion using the newly provided information.

```
from __future__ import annotations
from typing_extensions import TypedDict
from langgraph.checkpoint.memory import MemorySaver
from langgraph.graph import START, END, StateGraph
from langgraph.types import interrupt, Command

1. State Definition
class SupportState(TypedDict, total = False):
 message: str
```

```

account_id: str
answer: str

2. Node Definitions
def ensure_account_id(state: SupportState) -> dict:
 account_id = (state.get("account_id") or "").strip()
 if account_id:
 return {}

 # Pause here and surface a payload to the caller
 resume_value = interrupt(
 {
 "problem": "Missing account_id",
 "question": "Which account should I use?"
 }
)

 # This only runs on the resume call (the node restarts, interrupt returns a value)
 print(f"--- Node resumed with value: {resume_value} ---")
 return {"account_id": str(resume_value).strip()}

def handle_request(state: SupportState) -> dict:
 # This runs after resume and after ensure_account_id returns an account_id
 return {
 "answer": f"Handling request for account_id={state['account_id']}":
 {state['message']}"
 }

3. Graph Construction
builder = StateGraph(SupportState)
builder.add_node("ensure_account_id", ensure_account_id)
builder.add_node("handle_request", handle_request)

builder.add_edge(START, "ensure_account_id")
builder.add_edge("ensure_account_id", "handle_request")
builder.add_edge("handle_request", END)

4. Compile with Checkpointer (Crucial for state recovery)
checkpointer = MemorySaver()
graph = builder.compile(checkpointer = checkpointer)

Use a consistent thread_id so the checkpointer can find the paused state
config = {"configurable": {"thread_id": "recovery-test-thread"}}

STEP 1: Initial call. Pauses at interrupt.
print("--- Step 1: Initial call (Triggers Interrupt) ---")
step1 = graph.invoke({"message": "Please reset my password."}, config = config)

Most LangGraph setups surface the interrupt payload in the return value.
Print it so the demo visibly shows "we paused and asked a question".
print("Step 1 returned:", step1)

STEP 2: Resume call. Provide the resume value via Command(resume=...).
print("\n--- Step 2: Resuming (Proving Recovery) ---")
final_output = graph.invoke(Command(resume = "ACC-12345"), config = config)

Verification: must have reached END and produced "answer"
if "answer" in final_output:
 print(f"Recovery Successful! Final Answer: {final_output['answer']}")
else:
 print("Recovery Failed: 'answer' not found. Final output was:", final_output)

```

ch\_08\scr\fixable\_error.py

```

(.venv) C:\Users\alexc\llm-app\ch_08> python -m src.fixable_error
--- Step 1: Initial call (Triggers Interrupt) ---
Step 1 returned: {'message': 'Please reset my password.', '__interrupt__': [Interrupt(value={'problem': 'Missing account_id', 'question': 'Which account should I use?'}, id='44df9739d0ee6505e7c9b2420e28e686')]}

--- Step 2: Resuming (Proving Recovery) ---
--- Node resumed with value: ACC-12345 ---
Recovery Successful! Final Answer: Handling request for account_id=ACC-12345:
Please reset my password.

```

`ensure_account_id` is the gatekeeper node that refuses to proceed until the workflow has an `account_id`. It first normalizes the current state's `account_id`. If it's present, the node returns an empty update, letting the graph move on without changing state.

If the identifier is missing, the node triggers an `interrupt(...)` with a structured payload that explains what's wrong ("Missing account\_id") and asks a concrete question ("Which account should I use?"). At this point the graph stops and returns control to the caller instead of guessing. Because the graph was compiled with a checkpointer and invoked with a stable `thread_id`, the paused state is persisted and can be recovered later.

On the resume invocation, the graph is called again with `Command(resume=...)`. The node is re-entered from the start, but this time `interrupt()` returns the provided resume value. The code logs that value and writes it back into state as a cleaned `account_id`, turning the previously-blocking condition into valid input. Only after `ensure_account_id` produces an `account_id` does `handle_request` run. It formats the final answer using the now-complete state, and the graph reaches END, demonstrating a full pause → user fix → resume → completion path without losing progress.

#### 8.4.4 Unexpected errors: escalate cleanly, then recover from checkpoints when appropriate

Unexpected errors are the ones you treat as "stop and look into this". They include real bugs, wrong configuration, unclear or unsafe situations, or odd behaviour from an external system that the graph cannot interpret safely. In these cases, turning the error into a generic "sorry, something went wrong" message can hide a defect and make it harder to fix.

The usual recovery tool here is not an in-graph retry loop. Instead, you rely on checkpoints. When a checkpointer is enabled, LangGraph checkpoints graph state at super-steps; however, how much intermediate progress actually persisted depends on the durability mode (for example, `durability='exit'` persists only on exit/interrupt, while more durable modes persist intermediate checkpoints). Those checkpoints are stored under a `thread_id`; the thread groups the accumulated state and checkpoint history across a sequence of runs/invocations for that thread, not just a single run. That means you can inspect what happened, fix the problem, and then resume the thread from a saved checkpoint rather than restarting from the initial inputs (resuming produces a new fork/run from that point).

Once you depend on resume, you also have to write nodes in a "durable execution" style. Nodes should be deterministic (same input, same output) and idempotent (safe to run again without causing extra effects). Any operation with side effects or non-determinism should be isolated so replay doesn't repeat it unintentionally (guidance is to wrap side effects/non-determinism in tasks, and design those operations to be idempotent).

For cases where recovery means 'go back to a known good point', LangGraph supports time travel: you can list a thread's checkpoint history, choose a `checkpoint_id`, optionally update state at that checkpoint, and then resume from that checkpoint (which creates a new fork in the thread history).

The following example demonstrates "time-travel" recovery using checkpoints. The graph is run once with a deliberately incorrect state value, and that state is persisted as part of the thread's checkpoint history. Instead of restarting from scratch, the workflow resumes from an earlier `checkpoint_id`, patches state at that checkpoint, and continues from there as a new fork/run in the thread history. The point is to show that checkpointed graphs can be repaired

and continued after a bad state value or operational mistake, using the stored thread history as the source of truth for where to resume.

```

from __future__ import annotations
from typing_extensions import TypedDict
from langgraph.graph import StateGraph, START, END
from langgraph.checkpoint.sqlite import SqliteSaver

class MyState(TypedDict, total = False):
 counter: int
 some_flag: str

def my_node(state: MyState) -> dict:
 new_counter = int(state.get("counter", 0)) + 1
 print(
 f"-- Executing Node: Counter is {new_counter}, "
 f"Flag is '{state.get('some_flag')}' ---"
)
 return {"counter": new_counter}

builder = StateGraph(MyState)
builder.add_node("my_node", my_node)
builder.add_edge(START, "my_node")
builder.add_edge("my_node", END)

DB_PATH = "timetravel_demo.db"

with SqliteSaver.from_conn_string(DB_PATH) as saver:
 graph = builder.compile(checkpointer = saver)

 thread_id = "t-1"
 base_config = {"configurable": {"thread_id": thread_id}}

 # -----
 # Phase 1: Normal execution
 # -----
 print("--- Phase 1: Normal Execution ---")
 graph.invoke({"counter": 0, "some_flag": "broken"}, config = base_config)
 graph.invoke({}, config = base_config)

 # -----
 # Phase 2: Time travel recovery
 # -----
 print("\n--- Phase 2: Time Travel Recovery ---")

 history = list(graph.get_state_history(base_config))
 print(f"Checkpoints found: {len(history)}")

 # Show a few and also find a usable target checkpoint:
 # pick the first checkpoint (in history order) whose stored state is non-empty
 # and ideally includes 'some_flag' so this demo is unambiguous.
 target = None
 for i, h in enumerate(history):
 cfg = h.config.get("configurable", {})
 checkpoint_id = cfg.get("checkpoint_id")
 vals = graph.get_state(h.config).values

 # Print a compact view
 if i < 10:
 print(f" [{i}] checkpoint_id={checkpoint_id} state={vals}")

 # Choose a checkpoint that has our domain fields
 if isinstance(vals, dict) and ("some_flag" in vals or "counter" in vals):
 target = h

 if target is None:
 raise RuntimeError(
 "Could not find a checkpoint with application state (counter/some_flag). "
 "Your DB likely contains only structural checkpoints for this thread."
)

 print("\nChosen target checkpoint:")
 print("checkpoint_id:", target.config["configurable"].get("checkpoint_id"))
 print("state:", graph.get_state(target.config).values)

 print("\nRewinding and patching state...")

 # IMPORTANT: return type of update_state is version-dependent.
 # Some versions return a snapshot-like object, others return a dict, others None.
 graph.update_state(target.config, {"some_flag": "fixed"})

 # After patching, get the CURRENT state for the thread (latest checkpoint)
 # This avoids relying on update_state return shape.

```

```

patched_latest = graph.get_state(base_config).values
print("Patched latest state for thread:", patched_latest)

print("\nResuming execution from patched thread state...")
Invoke with base_config so the runtime uses the latest checkpoint for this thread.
result = graph.invoke({}, config = base_config)

print("\n--- Final Result after Recovery ---")
print(f"Counter: {result.get('counter')}")
print(f"Some Flag: {result.get('some_flag')}")

```

ch\_08\scr\timetravel\_recovery.py

The graph is a minimal “stateful counter” workflow used to demonstrate two behaviours: checkpointed persistence across runs, and “time travel” by editing a past checkpoint and resuming.

State is a TypedDict (MyState) with counter and some\_flag. The graph has one node, my\_node. Each time it runs, it increments counter and prints the current counter plus whatever some\_flag is. It only returns {"counter": new\_counter}, so the node never changes some\_flag.

The graph is compiled with a SqliteSaver checkpointer. Because of that, every invoke is tied to a thread identified by config["configurable"]["thread\_id"]. When you call graph.invoke twice with the same thread\_id, the second call loads the latest checkpoint for that thread and continues from the saved state instead of starting over. That is what Phase 1 shows: the first run starts from {"counter": 0, "some\_flag": "broken"}, and the second run (with empty input) still increments the existing counter because the state was persisted.

Phase 2 shows time travel. graph.get\_state\_history(base\_config) lists the saved checkpoints for that thread, and the code picks one checkpoint whose stored state includes the application fields. It then calls graph.update\_state(target.config, {"some\_flag": "fixed"}) to patch the thread state at that point. After that, graph.get\_state(base\_config) shows the latest state now includes some\_flag = "fixed". When the code invokes the graph again with the same thread\_id, execution resumes from the patched state, so my\_node prints the updated flag and increments counter again.

#### 8.4.5 Keeping recovery logic visible with shared subgraphs

As a graph grows, it’s tempting to copy-paste the same recovery steps everywhere: record the error, choose a fallback, and explain what happened to the user. That duplication quickly becomes a maintenance problem: you fix a bug in one place and forget another, or different nodes drift into slightly different “error styles.” A cleaner approach is to centralize recovery in a dedicated subgraph. In LangGraph, you can reuse a graph inside a larger graph either by invoking it from a parent node (with explicit state transformation), or by adding the compiled subgraph itself as a node when the parent and subgraph share state keys. That lets you reuse recovery behaviour without hiding it inside a single opaque function: the recovery path remains a real, named sequence of nodes. When you run with tracing/streaming, subgraph execution can be surfaced as sub-steps, so you can still inspect the node-by-node path.

A practical pattern is to standardize an “error envelope” in state (for example: error type, the node that failed, a message, and relevant context). Any node that fails writes this envelope in

the same shape. Then a single conditional edge (or a Command-based hop) in the main graph routes control into the recovery subgraph node. Inside that subgraph, you typically do three things in order: (1) log the envelope in a consistent format, (2) decide the safest fallback for this error category, and (3) produce a user-facing explanation that is honest, actionable, and does not leak internal details.

#### 8.4.6 Preventing recovery loops with explicit stop conditions

Recovery paths can fail too. A “retry/repair” loop can oscillate, or a supervisor can keep delegating to the same specialist that keeps failing in the same way. LangGraph enforces a `recursion_limit`: if a run keeps cycling without reaching a stop condition, execution halts with a `GraphRecursionError`. The right fix is usually not to raise the limit, but to define clear termination criteria in your graph/state. For complex graphs that legitimately need many steps, you can raise the limit per invocation (for example: `graph.invoke(input, {'recursion_limit': 100})`), but only after you’ve confirmed you still have a real stop condition. In practice, any graph that retries or repairs should track progress explicitly in state, so it can decide when it’s time to stop:

1. Attempt counters (per node, per tool, or per error category) so you can cap retries deterministically.
2. A `last_error` signature (for example: type + node + message) so you can detect “the exact same failure again” and avoid infinite repetition.
3. A terminal decision that routes to a safe end state: interrupt for a human, return a minimal fallback answer, or exit via an escalation path.

#### 8.4.7 Testing failure paths deliberately

When you unit-test an LCEL chain or a LangGraph node, the hardest part is usually not “does the prompt work?” It’s “does the plumbing behave correctly when the model behaves badly?” Real providers are the worst possible test dependency: slow, flaky, rate-limited, and non-deterministic. LangChain provides a small family of in-process fake language models for testing. These are in-process fakes (test doubles) that avoid real network calls and return predetermined responses (and, if needed, predetermined errors). They are deterministic drop-in language models that implement the standard `Runnable` interface, so you can return responses in a fixed order per invocation and simulate failures, then verify retries, fallbacks, parsing, and graph recovery. There are two parallel families:

- Fake LLMs (string-in / string-out style): `FakeListLLM` (plus `FakeListLLMError` / `FakeStreamingListLLM` for failure + streaming tests)
- Fake Chat Models (message-based): `FakeChatModel`, `FakeListChatModel` (plus `FakeListChatModelError`), `GenericFakeChatModel`

Use the LLM fakes when your chain is built around plain-text prompts and LLM-style outputs.

Use the chat fakes when your chain uses `ChatPromptTemplate` / `BaseMessage` inputs and chat model outputs.

##### 8.4.7.1 `FakeChatModel`: the simplest possible “model replied” stub

`FakeChatModel` always returns the same response text (“fake response”). It’s intentionally boring. That makes it useful for tests where you only care that a model call happens and the downstream wiring runs (prompt formatting, parser invocation, and run/tracing metadata

you attach via configuration). It does not try to emulate richer model behaviours like varied outputs or streaming.

```
from langchain_core.language_models.fake_chat_models import FakeChatModel
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

def test_chain_wiring_with_fake_chat_model() -> None:
 # 1) Build a chat prompt that produces chat messages (System + Human).
 prompt = ChatPromptTemplate.from_messages(
 [
 ("system", "You are a helpful assistant."),
 ("human", "Return any output for: {topic}")
]
)

 # 2) FakeChatModel ignores the prompt content and always returns a deterministic
 # reply.
 model = FakeChatModel()

 # 3) StrOutputParser extracts the .content from the AIMessage returned by the chat
 # model.
 chain = prompt | model | StrOutputParser()

 out = chain.invoke({"topic": "unit testing"})

 # FakeChatModel always yields the same content, regardless of input.
 print("Fake model response:", out)
 assert out == "fake response"

if __name__ == "__main__":
 # Run as a normal script (no pytest required).
 test_chain_wiring_with_fake_chat_model()
 print("OK: FakeChatModel wiring test passed.")
```

ch\_08\scr\FakeChatModel.py

This test tells you nothing about prompt quality (by design). It only proves the chain is wired correctly: the prompt can be formatted, the model step is callable, and the chat model returns a deterministic message (always the same content, regardless of input). With `StrOutputParser()` you then extract that message content as a plain string. Because the model is a fake, the test is safe to run in CI without network calls.

#### 8.4.7.2 `FakeListChatModel`: scripted multi-call sequences, one response per invocation

`FakeListChatModel` is a fake chat model for testing that returns a predefined list of responses in order. You pass `responses=[...]`; each invocation returns the next response, cycling back to the start when it reaches the end. This is ideal when your chain calls the model more than once and you want to drive it through a known sequence.

```
from __future__ import annotations
from langchain_core.language_models.fake_chat_models import FakeListChatModel
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

def build_two_call_chain(model: FakeListChatModel):
 # Call 1: produce a label
 step1_prompt = ChatPromptTemplate.from_messages(
 [
 ("human", "Give a short label for: {topic}")
]
)

 # Call 2: expand the label
 step2_prompt = ChatPromptTemplate.from_messages(
 [
 ("human", "Now expand on this label in one sentence: {label}")
]
)

 # First model call -> string label
 label_chain = step1_prompt | model | StrOutputParser()

 # Second model call: map {"label": <result of label_chain>} into the second prompt
 step2_chain = (
 {"label": label_chain}
 | step2_prompt
```

```

 | model
 | StrOutputParser()
)

 return step2_chain

def test_multi_call_flow_with_fake_list_chat_model() -> None:
 model = FakeListChatModel()
 responses = [
 "ALPHA",
 "ALPHA is the first letter of the Greek alphabet."
]
 chain = build_two_call_chain(model)
 out = chain.invoke({"topic": "Greek alphabet"})
 assert out == "ALPHA is the first letter of the Greek alphabet."

if __name__ == "__main__":
 # Run the test as a simple script (no pytest required).
 test_multi_call_flow_with_fake_list_chat_model()
 print("OK")

```

ch\_08\scr\FakeListChatModel.py

What you are testing here is orchestration: the chain makes two model calls in the expected order, correctly threads the intermediate value into the follow-up prompt, and produces the final output deterministically.

Some failure modes only appear in streaming: partial output arrives, then the stream dies. `FakeListChatModel` includes a test hook for that: `error_on_chunk_number` (the chunk number at which to raise during streaming). It will raise `FakeListChatModelError` on the specified chunk number while streaming (0-based; in `FakeListChatModel` each streamed chunk is a single character).

```

from langchain_core.language_models.fake_chat_models import (
 FakeListChatModel,
 FakeListChatModelError
)

def test_fail_mid_stream() -> None:
 # FakeListChatModel streams its response one character at a time.
 # error_on_chunk_number is 0-based: 0 = first char, 1 = second char, etc.
 model = FakeListChatModel(
 responses = ["Streaming output"],
 error_on_chunk_number = 5 # raise right before yielding chunk #5
)

 received_chunks: list[str] = []

 try:
 for chunk in model.stream("ignored input"):
 # chunk is an AIMessageChunk; .content contains the next streamed piece
 received_chunks.append(chunk.content)

 raise AssertionError("Expected FakeListChatModelError, but the stream completed successfully.")
 except FakeListChatModelError:
 partial = "".join(received_chunks)

 # "Streaming output" streamed as: s(0) t(1) r(2) e(3) a(4) ... then error at
 # chunk 5
 assert partial == "strea", f"Unexpected partial output: {partial!r}"

if __name__ == "__main__":
 # Run as a plain script (no pytest needed)
 test_fail_mid_stream()
 print("OK: FakeListChatModelError was raised mid-stream as expected.")

```

ch\_08\scr\FakeListChatModel\_streaming.py

This does not test “model correctness.” It tests your system’s behavior when streaming breaks: cleanup, logging, fallbacks, and user-visible error handling.

#### 8.4.7.3 *GenericFakeChatModel: programmable sequences via an iterator (including forced failures)*

FakeListChatModel is scripted. GenericFakeChatModel is programmable. You provide an iterator that yields either strings or AIMessage objects. Each call to the model consumes the next item from the iterator. If the iterator raises an exception (including StopIteration when it is exhausted), the model invocation fails with that exception. That makes GenericFakeChatModel a convenient tool for testing retry and fallback logic because you can create patterns like:

- fail on the first call, succeed on the second
- fail every other call
- return different message payloads over time (e.g., yield AIMessage objects with varying additional\_kwargs)
- simulate “bad days” without touching the network

A common use is to verify that your retry configuration is attached at the right place (the risky runnable, not the whole chain).

```
from __future__ import annotations
from collections.abc import Iterator
from typing import Union
from langchain_core.language_models.fake_chat_models import GenericFakeChatModel
from langchain_core.messages import AIMessage
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import ChatPromptTemplate

class FailOnceThenSucceed(Iterator[Union[AIMessage, str]]):
 # Iterator consumed by GenericFakeChatModel.
 # First "generation" raises ValueError, second returns an AIMessage.
 def __init__(self) -> None:
 self._count = 0

 def __iter__(self) -> "FailOnceThenSucceed":
 return self

 def __next__(self) -> Union[AIMessage, str]:
 self._count += 1
 if self._count == 1:
 raise ValueError("simulated transient failure")
 return AIMessage(content = "OK after retry")

 def test_retry_only_wraps_the_model_call() -> None:
 prompt = ChatPromptTemplate.from_messages([("human", "Ping")])

 flaky_model = GenericFakeChatModel(messages = FailOnceThenSucceed())

 # Retry only the model call (the part that can fail).
 model_with_retry = flaky_model.with_retry(
 stop_after_attempt = 2,
 retry_if_exception_type = (ValueError,))
)

 chain = prompt | model_with_retry | StrOutputParser()
 out = chain.invoke({})
 assert out == "OK after retry"

 if __name__ == "__main__":
 # Run the "test" as a normal script.
 test_retry_only_wraps_the_model_call()
 print("PASS")
```

ch\_08\scr\GenericFakeChatModel.py

This is the kind of test that catches real production wiring mistakes early. If you accidentally attached retry to the wrong component (or forgot it entirely), this test fails immediately and deterministically.

#### 8.4.7.4 *FakeListLLM: deterministic testing for non-chat LLM chains*

If you are using an LLM-style component (string prompts, string responses), `FakeListLLM` provides the same “scripted sequence” behavior as `FakeListChatModel` but in the LLM family.

```
from __future__ import annotations
from langchain_core.language_models.fake import FakeListLLM
from langchain_core.output_parsers import StrOutputParser
from langchain_core.prompts import PromptTemplate

def test_fake_list_llm_for_text_chains() -> None:
 # PromptTemplate produces a plain string prompt (not chat messages)
 prompt = PromptTemplate.from_template("Summarize: {text}")

 # FakeListLLM returns the scripted responses in order, one per invocation,
 # ignoring the input prompt content.
 llm = FakeListLLM(responses = ["SUMMARY_1", "SUMMARY_2"])

 # FakeListLLM already returns a string, so StrOutputParser is optional here,
 # but keeping it makes the pipeline style consistent across examples.
 chain = prompt | llm | StrOutputParser()

 out1 = chain.invoke({"text": "first"})
 out2 = chain.invoke({"text": "second"})

 assert out1 == "SUMMARY_1", f"Expected SUMMARY_1, got: {out1!r}"
 assert out2 == "SUMMARY_2", f"Expected SUMMARY_2, got: {out2!r}"

def main() -> None:
 test_fake_list_llm_for_text_chains()
 print("OK: FakeListLLM returned scripted responses in order.")

if __name__ == "__main__":
 main()
```

ch\_08\scr\FakeListLLM.py

This is most useful when you have legacy LLM-style chains or when you deliberately keep a component text-only (for example, a simple rewrite step) and still want deterministic unit tests.

#### 8.4.7.5 *Using fake models inside LangGraph nodes to test failure handling and retry paths*

The same fake chat models can be used inside LangGraph nodes because they implement the same chat-model interface as real providers, so you can call them (via `.invoke/.stream`) from node code without hitting an external API. That means you can test failure-handling paths—exception handling, retries (either implemented in the node or configured via LangGraph/Runnable retry features), fallback branches, and the resulting state updates—without any provider dependency. Below is a minimal example: a node calls a fake model that raises on some calls, retries (or falls back), and returns a default/stable value as a state update.

```
from __future__ import annotations
from typing import Annotated, List
from typing_extensions import TypedDict
from langchain_core.language_models.fake_chat_models import GenericFakeChatModel
from langchain_core.messages import AIMessage, BaseMessage, HumanMessage
from langgraph.graph import StateGraph, START, END
from langgraph.graph.message import add_messages

class FailOnceThenOK:
 # Iterator that fails on the first read, then succeeds forever.
 def __init__(self) -> None:
 self._count = 0

 def __iter__(self) -> "FailOnceThenOK":
 return self

 def __next__(self) -> AIMessage:
 self._count += 1
 if self._count == 1:
 raise ValueError("transient")
 return AIMessage(content = "Recovered")
```

```

class ChatState(TypedDict, total = False):
 # add_messages is a reducer: returned messages are merged/appended into
 # state["messages"]
 messages: Annotated[List[BaseMessage], add_messages]

def node(state: ChatState) -> dict:
 # Make the fake model fail once, then succeed. Wrap ONLY the risky call with retry.
 model = GenericFakeChatModel(messages = iter(FailOnceThenOK())).with_retry(
 stop_after_attempt = 2,
 retry_if_exception_type = (ValueError,))
)

LangGraph chat-style state typically stores messages. Pass them to the chat model.
messages = state.get("messages", [])
reply = model.invoke(messages)

With add_messages, return ONLY the new messages to add.
return {"messages": [reply]}

def build_graph():
 builder = StateGraph(ChatState)
 builder.add_node("node", node)
 builder.add_edge(START, "node")
 builder.add_edge("node", END)
 return builder.compile()

def test_graph_node_retry_path() -> None:
 graph = build_graph()
 out = graph.invoke({"messages": [HumanMessage(content = "start")]})
 assert out["messages"][-1].content == "Recovered"

if __name__ == "__main__":
 # Run as a script (no pytest required)
 graph = build_graph()
 result = graph.invoke({"messages": [HumanMessage(content = "start")]})
 print([f"{m.type}: {m.content}" for m in result["messages"]])
 assert result["messages"][-1].content == "Recovered"
 print("OK")

```

ch\_08\scr\GenericFakeChatModel\_graph.py

This test is small on purpose: it focuses on the recovery path and state update mechanics. You can scale the same pattern to verify that your graph keeps running even when a model call fails, that you append the right messages, and that your fallback behavior is stable.

Practical rules of thumb:

- `FakeChatModel`: use it when you only need a placeholder response to test wiring and “does this pipeline run at all?”
- `FakeListChatModel` / `FakeListLLM`: use them when you need a scripted multi-call sequence and you want every test run to behave identically
- `GenericFakeChatModel`: use it when you want to drive behavior from a custom message iterator (including deliberately raising exceptions) to simulate failures and validate retry/fallback logic.”

The important mental shift is this: deterministic fake models are for testing your application’s control flow and failure handling, not for testing “intelligence.” They make the weird parts of production (retries, parsing failures, graph recovery) reproducible, and that is exactly what you want from unit tests.

## 8.5 REASONING USING GRAPHS

This chapter revisits some reasoning patterns you already met in LCEL but puts them in the production mental model: reasoning is not just prompt text it is workflow control flow. In LCEL, you can express most of these ideas by wrapping a chain in normal Python. That is fine for quick experiments. But in real systems you care about repeatability, observability, and cost: when do we run extra reasoning, how many times, and what do we keep? Graphs are the clean abstraction for that. LangGraph models a workflow as shared state plus nodes that return state updates, connected by edges that decide what runs next. The runtime executes in discrete steps and merges updates in a predictable way.

In this chapter we treat reasoning strategies as first-class workflow shapes. Each “strategy” is just a different way to connect the same basic pieces:

- A single reasoning pass is one node.
- Self-consistency is a loop that collects multiple independent answers, followed by a voting (or scoring) node.
- Tree-of-thought is fan-out exploration across multiple candidate lines, followed by pruning and selection, possibly repeated over several depths.
- Least-to-most prompting is staged decomposition: a node that derives simpler subproblems, followed by a sequence of nodes that solve them and combine results.
- Verification is a two-phase pipeline: generate an answer, then run a checker node that validates claims and either accepts the result or routes into a repair step.
- Reflection (critique) is an iterative loop: generate, critique, revise, and stop when a router decides the output is good enough or the budget is exhausted.

Structured reasoning paths are not a separate “prompt trick”. They are state fields and reducers: the workflow records what it tried (plans, candidates, checks, decisions) as explicit data, so you can audit and debug without depending on hidden model behavior. These are not new magic techniques. They are standard workflow shapes. The point of LangGraph is that you can implement them in a way that is explicit, testable, and measurable.

### 8.5.1 A single reasoning pass is one node

Start with the smallest useful graph: take a question, run the model once, return one answer. That is a single reasoning pass, and in LangGraph it is just one node wired from start to end. This matters because it gives you a stable baseline. Before you introduce multiple samples, routing, or search, you can confirm that your prompt, your model configuration, and your output formatting behave correctly in the simplest possible workflow. Every strategy in the rest of the chapter is built by keeping this baseline pass intact and changing only the shape around it:

- Self-consistency is this same pass repeated K times, followed by a vote (or scoring) step.
- Verification is this pass followed by a check-and-repair step when needed.
- Tree-of-thought is many parallel uses of this pass, followed by pruning and selection.

So the “thinking” does not become mysterious or magical as the chapter progresses. You are always composing the same basic pass into larger workflows. Once this baseline is in place, the rest of the chapter becomes a controlled set of variations: you keep the same core node and change only the graph around it. A loop plus a final vote gives you self-consistency. A second

pass that checks and possibly repairs gives you verification. A fan-out that explores multiple candidates before selection gives you tree-of-thought. The power is that you are composing, not reinventing.

```

from __future__ import annotations
from typing_extensions import TypedDict
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY
from langgraph.graph import StateGraph, START, END

--- State ---
class SupportState(TypedDict, total = False):
 question: str
 answer: str

--- One reasoning pass ---
prompt = ChatPromptTemplate.from_messages(
 [
 ("system", "You are a careful e-commerce support assistant."),
 ("human", "{question}")
]
)

model = ChatOpenAI()
chain = prompt | model

def single_pass(state: SupportState) -> dict:
 msg = chain.invoke({"question": state["question"]})
 return {"answer": msg.content}

--- Graph ---
builder = StateGraph(SupportState)
builder.add_node("single_pass", single_pass)

builder.add_edge(START, "single_pass")
builder.add_edge("single_pass", END)

graph = builder.compile()

if __name__ == "__main__":
 out = graph.invoke({"question": "Where is my order #1234?"})
 print(out["answer"])

```

ch\_08\scr\reasoning\_onepass.py

### 8.5.2 Self-consistency

Once you have the single-pass baseline working, self-consistency is the smallest “upgrade” you can make without changing the reasoning step itself. The idea is simple: instead of trusting one sample, you run the same pass several times, treat each run as an independent candidate, then choose a final result by aggregation. In graph form, that becomes a loop with two phases.

- In the first phase, the graph repeats the same reasoning pass K times. Each iteration produces one candidate answer and appends it to a list. The important detail is that the candidates are meant to be independent samples: you are not refining the previous answer, you are collecting alternatives.
- In the second phase, the loop stops and the workflow switches to an aggregator node. That node looks at the list of candidates and picks the “winner”. The simplest aggregator is a vote: choose the most common final answer. In cases where answers are not identical strings, you can use scoring instead: normalize answers into comparable forms, or ask a judge model to rank candidates, then select the best.

The benefit is not that the model suddenly becomes smarter. The benefit is that your workflow becomes less brittle. If the model sometimes produces a weak answer, you have multiple

chances to get a good one and a transparent rule for selecting it. The cost is equally explicit: you pay for K runs of the same pass, and you decide when that extra cost is justified. not reinventing.

```

from __future__ import annotations
import os
import operator
from collections import Counter
from typing import Annotated, Literal
from typing_extensions import TypedDict
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY
from langgraph.graph import StateGraph, START, END

--- State -----
class SupportState(TypedDict, total = False):
 question: str
 answer: str

 # Self-consistency additions
 k: int
 samples: Annotated[int, operator.add]
 candidates: Annotated[list[str], operator.add]
 final_answer: str

--- One reasoning pass (unchanged in spirit) -----
prompt = ChatPromptTemplate.from_messages(
 [
 ("system", "You are a careful e-commerce support assistant."),
 ("human", "{question}")
]
)

For self-consistency you usually want some diversity across samples.
model = ChatOpenAI(temperature = 0.8)
chain = prompt | model

def single_pass(state: SupportState) -> dict:
 # The same "one pass" you already had, but in self-consistency we treat each
 # pass as a candidate. We store it as an appended list element.
 msg = chain.invoke({"question": state["question"]})
 return {
 "answer": msg.content, # last sample (useful for debugging)
 "candidates": [msg.content], # accumulate across iterations
 "samples": 1 # increment loop counter
 }

def should_continue(state: SupportState) -> Literal["single_pass", "vote"]:
 # Loop until we have collected k samples, then move to the vote step.
 k = state.get("k", 5)
 samples = state.get("samples", 0)
 return "single_pass" if samples < k else "vote"

def vote(state: SupportState) -> dict:
 # The simplest aggregator: majority vote on exact strings.
 # (Later in the chapter you can swap this node for scoring / judging.)
 candidates = state.get("candidates", [])
 if not candidates:
 return {"final_answer": "", "answer": ""}
 winner = Counter(candidates).most_common(1)[0][0]
 return {"final_answer": winner, "answer": winner}

--- Graph -----
builder = StateGraph(SupportState)

builder.add_node("single_pass", single_pass)
builder.add_node("vote", vote)

builder.add_edge(START, "single_pass")
builder.add_conditional_edges("single_pass", should_continue)
builder.add_edge("vote", END)

graph = builder.compile()

if __name__ == "__main__":
 out = graph.invoke(
 {

```

```

 "question": "Where is my order #1234?",
 "k": 5,
 "samples": 0,
 "candidates": [],
 "final_answer": ""
 }
}

print("Candidates:")
for i, c in enumerate(out["candidates"], start = 1):
 print(f"{i}. {c}")

print("\nFinal:")
print(out["final_answer"])

```

ch\_08\scr\reasoning\_selfconsistency.py

```
(.venv) C:\Users\alexc\llm-app\ch_08> python -m src.reasoning_selfconsistency
Candidates:
1. Hello! I can help you track your order #1234. Can you please provide me with the tracking number or the courier company that is handling the delivery?
2. I apologize for the inconvenience. To provide you with an accurate update on the status of order #1234, could you please provide me with your order number or tracking number?
3. I'm sorry to hear that you're having trouble locating your order #1234. To assist you further, could you please provide me with more details like the name of the website or company you placed the order with, the tracking number if applicable, or any other information that might help me locate your order?
4. I'm sorry to hear that you're having trouble locating your order. To better assist you, could you please provide me with more information such as the website you ordered from and any tracking number that you may have received? This will help me look into the status of your order and provide you with the most accurate information.
5. I am happy to help with that! To locate your order #1234, I will need some information from you. Could you please provide me with your full name or the email address associated with the order?

Final:
Hello! I can help you track your order #1234. Can you please provide me with the tracking number or the courier company that is handling the delivery?
```

What changed (and why it's still the “same example”)

1. The reasoning step stays the same: prompt → model → text response.
2. The node now returns two extra updates each time: it appends the latest answer to candidates and increments samples.
3. The graph now has one conditional edge: after each pass it either loops back (until k is reached) or moves forward to vote.
4. The final node aggregates the collected candidates into final\_answer (here: majority vote).

### 8.5.3 Tree-of-thought

Tree-of-thought is fan-out exploration across multiple candidate lines, followed by pruning and selection, possibly repeated over several depths.

Self-consistency is the “ask several times and vote” move. Tree-of-thought is a different move: it treats the answer space like a terrain you can explore. Instead of collecting several full answers and hoping the majority is right, you deliberately branch into different approaches, then cut away the weak ones, and repeat. It is closer to a tiny search algorithm than to a sampling trick. The shape is always the same:

- You fan out: generate several candidate lines in parallel. Each candidate is meant to be meaningfully different (different assumptions, different next actions, different explanation style), not just rephrasing.
- You prune: select the best few candidates and discard the rest. This is the “budget discipline” step. Without pruning, tree-of-thought turns into uncontrolled combinatorics.
- You go deeper: take what survived and expand again for another round. Depth is not “think harder” in the abstract; it is “explore → select → explore again”. Each round gives the workflow a chance to correct itself by forcing alternatives and re-evaluating them.

The result is a controlled search process where the knobs are explicit: fanout controls how many alternatives you consider, beam width controls how aggressively you prune, and depth controls how many rounds you are willing to spend. Breadth buys you options, pruning buys you focus, depth buys you iterative refinement.

The example below is a direct continuation of our script style. It keeps the same baseline setup (prompt, ChatOpenAI, StateGraph, invoke). The only new pieces are (1) an expand node that generates multiple candidates at once, (2) a select-and-prune node that keeps the best ones, and (3) a loop that repeats that pattern for a fixed number of depths.

```

from __future__ import annotations
import os
import operator
import re
from typing import Annotated, Literal
from typing_extensions import TypedDict
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, START, END
from .config import OPENAI_API_KEY

State

class SupportState(TypedDict, total = False):
 # State is the shared memory that moves through the graph.
 # NOTE on reducers:
 # - `trace` uses operator.add so every node can append lines without overwriting.
 # - Everything else overwrites by default (last write wins).

 # Input question from user
 question: str

 # ToT loop controls
 depth: int
 max_depth: int
 fanout: int
 beam_width: int

 # Working sets produced each round (we overwrite these explicitly each round)
 candidates: list[str]
 survivors: list[str]

 # The best answer found so far (we overwrite each round)
 final_answer: str

 # Human-readable trace of the ToT process (accumulates across rounds)
 trace: Annotated[list[str], operator.add]

 # Temperature 0 makes the demo more deterministic (better for teaching).
model = ChatOpenAI(model = "gpt-4o-mini", temperature = 0)

Prompts

Accept a "seed_answer" that is empty at depth 0, and filled
at later depths. The model is instructed to IMPROVE the seed rather than
regenerate from scratch.
expand_prompt = ChatPromptTemplate.from_messages(
 [

```

```

 (
 "system",
 "You generate multiple candidate answers for an e-commerce support
 question.\n\n"
 "Hard rules:\n"
 "-- Do NOT invent order status, shipment dates, tracking numbers, or internal
 system facts.\n"
 "-- If information is missing, ask for what you need and give a helpful next
 action.\n\n"
 "Refinement rule:\n"
 "-- If a SEED ANSWER is provided, generate improved variants of it (clearer,
 more helpful, better next step).\n"
 "-- If no seed is provided, generate candidates from scratch.\n\n"
 "Output requirements:\n"
 "-- Provide exactly {fanout} candidates.\n"
 "-- Output exactly one candidate per line, prefixed with: 'CANDIDATE <n>: '\n"
 "-- Do not add any other text.\n"
),
 (
 "human",
 "Question:\n{question}\n\n"
 "SEED ANSWER (may be empty):\n{seed_answer}\n"
)
]
}

Selector prompt chooses the best candidate by number.
select_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "Select the single best support answer by number.\n"
 "Criteria:\n"
 "-- Correctness (no made-up facts)\n"
 "-- Clarity\n"
 "-- Helpful next action\n\n"
 "Output format: 'BEST: <n>' and nothing else.\n"
),
 ("human", "Question:\n{question}\n\nCandidates:\n{candidates_block}")
]
)

Parsers (turn model output into structured data)

_candidate_line = re.compile(r"^\s+CANDIDATE\s+\d+:\s*(.*)\s*\$",
 re.IGNORECASE)
_best_line = re.compile(r"^\s+BEST:\s*(\d+)", re.IGNORECASE)

def parse_candidates(text: str) -> list[str]:
 # Extract candidate lines like:
 # CANDIDATE 1: ...
 # CANDIDATE 2: ...
 # Returns list of candidate strings (without the prefix).
 out: list[str] = []
 for line in text.splitlines():
 m = _candidate_line.match(line.strip())
 if m:
 out.append(m.group(1).strip())
 return out

def parse_best_index(text: str) -> int | None:
 # Extract BEST: <n> and convert to 0-based index.
 # Returns None if parsing fails.
 m = _best_line.search(text)
 if not m:
 return None
 idx = int(m.group(1)) - 1
 return idx if idx >= 0 else None

def preview(s: str, n: int = 90) -> str:
 # Produce a compact one-line preview for trace output.
 s = " ".join(s.split())
 return s if len(s) <= n else s[:n - 3] + "..."

Nodes (graph steps)

def expand(state: SupportState) -> dict:
 # Fan-out step:
 # - depth == 0: generate candidates from scratch (seed empty)
 # - depth >= 1: generate improved variants of the current best answer

```

```

(seed = state['final_answer'])
question = state["question"]
depth = state.get("depth", 0)
fanout = state.get("fanout", 5)

Minimal adjustment: feed the prior best answer back in as "seed_answer"
seed_answer = "" if depth == 0 else state.get("final_answer", "")

msg = (expand_prompt | model).invoke(
 {"question": question, "fanout": fanout, "seed_answer": seed_answer}
)

candidates = parse_candidates(msg.content)

If formatting is wrong, fall back to one "candidate" (keeps demo resilient).
if not candidates:
 candidates = [msg.content.strip()]

trace_lines = [f"D{depth} expand: {len(candidates)} candidates"]
if seed_answer:
 trace_lines.append(f" seed: {preview(seed_answer)}")
for i, c in enumerate(candidates, start = 1):
 trace_lines.append(f" - C{i}: {preview(c)}")

IMPORTANT PASSAGE:
We overwrite candidates each round (not append), to keep each depth self-contained.
return {"candidates": candidates, "trace": trace_lines}

def select_and_prune(state: SupportState) -> dict:
 # Prune step:
 # - Choose the best candidate (BEST: n)
 # - Keep a small survivor set (beam), primarily to show the concept
 # (we don't actually expand per-survivor in this minimal demo).
 candidates = state.get("candidates", [])
 depth = state.get("depth", 0)

 if not candidates:
 # Defensive: should not happen, but keeps the graph stable.
 return {"survivors": [], "final_answer": "", "trace": [f"D{depth} select: no candidates"]}

 beam_width = state.get("beam width", 2)
 candidates_block = "\n".join(f"{i + 1}. {c}" for i, c in enumerate(candidates))

 msg = (select_prompt | model).invoke(
 {"question": state["question"], "candidates_block": candidates_block}
)

 best_idx = parse_best_index(msg.content)

 # Defensive: if selector output is malformed, default to the first candidate.
 if best_idx is None or best_idx >= len(candidates):
 best_idx = 0

 best = candidates[best_idx]

 # Beam survivors: keep best + a couple backups (simple demonstration).
 others = [c for i, c in enumerate(candidates) if i != best_idx]
 survivors = [best] + others[: max(0, beam_width - 1)]

 trace_lines = [
 f"D{depth} select: BEST={best_idx + 1}",
 f" keep beam_width={beam_width} => survivors={len(survivors)}",
 f" best: {preview(best)}"
]

 # IMPORTANT PASSAGE:
 # `final_answer` is the "best so far". Next round uses it as a seed.
 return {"survivors": survivors, "final_answer": best, "trace": trace_lines}

def increment_depth(state: SupportState) -> dict:
 # Loop bookkeeping:
 # - Increase depth by 1
 # - Clear candidates so the next round doesn't carry old candidates forward.
 d = state.get("depth", 0) + 1
 return {"depth": d, "candidates": [], "trace": [f"advance: depth -> {d}"]}

def route(state: SupportState) -> Literal["expand", "end"]:
 # Decide whether to continue looping.
 # We stop when depth >= max_depth.
 return "expand" if state.get("depth", 0) < state.get("max_depth", 2) else "end"

Graph wiring

```

```

builder = StateGraph(SupportState)

Register nodes
builder.add_node("expand", expand)
builder.add_node("select_and_prune", select_and_prune)
builder.add_node("increment_depth", increment_depth)

Linear: START -> expand -> select -> increment -> (expand or END)
builder.add_edge(START, "expand")
builder.add_edge("expand", "select_and_prune")
builder.add_edge("select_and_prune", "increment_depth")
builder.add_conditional_edges("increment_depth", route, {"expand": "expand", "end": END})

graph = builder.compile()

Run (print tree trace + final answer)

if __name__ == "__main__":
 out = graph.invoke(
 {
 "question": "Where is my order #1234?",
 "depth": 0,
 "max_depth": 2, # number of refine rounds
 "fanout": 3, # how many candidates each round
 "beam_width": 2, # how many survivors to keep (demo)
 "candidates": [],
 "survivors": [],
 "final_answer": "",
 "trace": []
 }
)

 print("\n==== TREE TRACE (ToT artifacts, not hidden reasoning) ====")
 for line in out.get("trace", []):
 print(line)

 print("\n==== FINAL ANSWER ====")
 print(out.get("final_answer", ""))

```

ch\_08\src\reasoning\_treeofthought.py

```

(.venv) C:\Users\alexc\llm-app\ch_08> python -m src.reasoning_treeofthought
==== TREE TRACE (ToT artifacts, not hidden reasoning) ====
D0 expand: 3 candidates
 - C1: I'm sorry, but I can't provide specific order details without more
information. Please ...
 - C2: I can't access the details of your order #1234 directly. Please check
your order confir...
 - C3: Unfortunately, I don't have access to order statuses. Please check your
order confirmat...
D0 select: BEST=2
 keep beam_width=2 => survivors=2
 best: I can't access the details of your order #1234 directly. Please check
your order confir...
advance: depth -> 1
D1 expand: 3 candidates
 seed: I can't access the details of your order #1234 directly. Please check
your order confir...
 - C1: I don't have direct access to your order details for #1234. Please
check your order con...
 - C2: Unfortunately, I can't access the specifics of order #1234. I recommend
checking your o...
 - C3: I'm unable to view the details of your order #1234 directly. Please
look for your order...
D1 select: BEST=1
 keep beam_width=2 => survivors=2
 best: I don't have direct access to your order details for #1234. Please
check your order con...
advance: depth -> 2

==== FINAL ANSWER ====
I don't have direct access to your order details for #1234. Please check your
order confirmation email for tracking information. If you need further
assistance, provide your email address, and I'll help you locate it.

```

The run begins by invoking the graph with loop parameters (`max_depth`, `fanout`, `beam_width`) and an empty working set. The first real step is `expand()`. At each depth, it calls the model with `expand_prompt` and asks for exactly `fanout` candidates in a rigid, line-based format. At depth 0, `seed_answer` is intentionally empty, so the model generates candidates from scratch. From depth 1 onward, `seed_answer` is set to the current `final_answer`, pushing the model to produce improved variants instead of restarting. This “seeded refinement” is the key implementation choice that makes depth meaningful: each round is a targeted revision cycle.

`parse_candidates()` turns the model’s text into a `list[str]`. If the model violates the output contract, the node falls back to treating the entire response as a single candidate, keeping the workflow resilient instead of failing fast. The node also writes a compact trace: it records how many candidates were produced and includes short previews, so you can audit what the search explored without depending on opaque reasoning.

Next, `select_and_prune()` performs pruning. It formats the candidates into a numbered block, calls the selector prompt, and parses `BEST: <n>` into an index. If parsing fails, it defaults to candidate 1—another deliberate defensive choice that stabilizes the control flow. The node then sets `final_answer` to the selected best candidate and constructs survivors as a simple “beam”: the best plus a small number of backups (up to `beam_width`). In this minimal demo, survivors are primarily educational scaffolding—the next round refines only the best via the seed—but the state shape supports extending expansion per survivor later.

`increment_depth()` advances the loop counter and clears `candidates` to prevent accidental carry-over between rounds. Finally, `route()` checks depth against `max_depth` to decide whether to loop back to `expand` or terminate. The output prints the accumulated trace and the final selected answer, making the search process inspectable end-to-end.

#### 8.5.4 Least-to-most prompting

Least-to-most prompting is staged decomposition: a node that derives simpler subproblems, followed by a sequence of nodes that solve them and combine results. Least-to-most is the opposite of “let’s just wing it with one big prompt.” It assumes that many real support requests are compound questions in disguise: the customer gives you a messy bundle of symptoms, constraints, and goals, and you have to untangle it into a sequence of smaller decisions. In customer support, this is basically the human playbook:

- Get the easy, low-risk facts straight first (what happened, what the customer sees, what policies apply).
- Use those facts to resolve the harder, dependent decisions (what action to take, what to communicate, what to escalate).
- Only then write the final customer-facing message, because now you know what you’re actually saying.

When you implement this with LangGraph, it becomes a clean, staged workflow rather than a single “do everything” model call:

1. Decompose (least → most): One node turns the original request into a short, ordered list of subproblems. The ordering is the whole point: early subtasks should be answerable with minimal assumptions, and later subtasks are allowed to depend on earlier results.
2. Solve sequentially: The graph runs a loop over that list. Each iteration answers exactly one subproblem and appends the result to state. This prevents the model from “mixing concerns” and makes it obvious where things went wrong if the final answer is off.
3. Combine into a final response: A final node synthesizes the intermediate results into a single coherent answer in the style your assistant should produce. This is where you turn internal reasoning into a customer-ready message.

A useful way to think about it: self-consistency spends extra budget on multiple parallel attempts at the same question, then aggregates. Least-to-most spends budget on structured serial progress, where each step reduces uncertainty for the next. Same ingredients, different graph shape, different failure modes, and different places to insert tools later (order lookup, policy retrieval, address validation, escalation rules) without rewriting the whole system.

```

from __future__ import annotations
import os
import json
import operator
from typing import Annotated, Literal
from typing_extensions import TypedDict
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY
from langgraph.graph import StateGraph, START, END

--- State ---
class SupportState(TypedDict, total = False):
 question: str
 answer: str

 # Least-to-most additions
 subtasks: Annotated[list[str], operator.add]
 solutions: Annotated[list[str], operator.add]
 idx: Annotated[int, operator.add]

--- Model ---
model = ChatOpenAI(model = "gpt-4o-mini", temperature = 0.2)

--- Prompts ---
1) Decompose into ordered subtasks (least -> most)
decompose_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a careful e-commerce support assistant.\n"
 "Break the user's request into 2-5 smaller subquestions ordered from easiest\n"
 "to hardest.\n"
 "Return ONLY valid JSON in this exact shape:\n"
 '{{ "subtasks": ["...","..."] }}' # <-- escaped braces
),
 ("human", "{question}")
]
)

2) Solve one subtask at a time
solve_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a careful e-commerce support assistant.\n"
 "Answer the subtask concisely and precisely.\n"
 "Do not invent order data; if details are missing, state what would be\n"
 "needed.\n"
),
 ("human", "Original request:\n{question}\n\nSubtask:\n{subtask}")
]
)

```

```

)

3) Combine intermediate answers into a final response
combine_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a careful e-commerce support assistant.\n"
 "Write the final customer-facing answer using the solved subtasks.\n"
 "Be clear, actionable, and avoid internal process language.\n"
),
 ("human", "Original request:\n{question}\n\nSolved subtasks:\n{solutions}")
]
)

--- Nodes -----
def decompose(state: SupportState) -> dict:
 msg = (decompose_prompt | model).invoke({"question": state["question"]})
 text = msg.content.strip()

 # Parse the required JSON shape. If parsing fails, fall back to a single subtask.
 subtasks: list[str]
 try:
 data = json.loads(text)
 subtasks = [s.strip() for s in data.get("subtasks", []) if s and s.strip()]
 except Exception:
 subtasks = [state["question"]]

 # Initialize the loop counter.
 return {"subtasks": subtasks, "solutions": [], "idx": 0}

def solve_next(state: SupportState) -> dict:
 subtasks = state.get("subtasks", [])
 idx = state.get("idx", 0)

 # Defensive: if nothing left, do nothing (router will move to combine).
 if idx >= len(subtasks):
 return {}

 subtask = subtasks[idx]
 msg = (solve_prompt | model).invoke({"question": state["question"], "subtask": subtask})

 return {
 "solutions": [msg.content.strip()],
 "idx": 1 # accumulated via reducer to advance to the next subtask
 }

def route(state: SupportState) -> Literal["solve_next", "combine"]:
 subtasks = state.get("subtasks", [])
 idx = state.get("idx", 0)
 return "solve_next" if idx < len(subtasks) else "combine"

def combine(state: SupportState) -> dict:
 solutions = state.get("solutions", [])
 numbered = "\n".join(f"\n{i + 1}. {s}" for i, s in enumerate(solutions)) or "(no intermediate results)"
 msg = (combine_prompt | model).invoke({"question": state["question"], "solutions": numbered})
 return {"answer": msg.content.strip()}

--- Graph -----
builder = StateGraph(SupportState)

builder.add_node("decompose", decompose)
builder.add_node("solve_next", solve_next)
builder.add_node("combine", combine)

builder.add_edge(START, "decompose")
builder.add_edge("decompose", "solve_next")
builder.add_conditional_edges("solve_next", route)
builder.add_edge("combine", END)

graph = builder.compile()

if __name__ == "__main__":
 out = graph.invoke(
 {
 "question": "My package says delivered but I can't find it. Also, I might need to change the delivery address for my next order. What should I do?"
 }
)

```

```
print(out["answer"])
```

ch\_08\scr\reasoning\_leasttomost.py

The code implements least-to-most as a three-stage LangGraph workflow: decompose → solve sequentially → combine. The decompose node is responsible for turning the original user request into an ordered plan the rest of the graph can execute deterministically. It invokes the model with a prompt that requires a strict JSON payload of the form `{"subtasks": [...]}`. The node then parses the model's response with `json.loads`, strips whitespace, and filters empty entries. This parsing step is a key reliability control that converts a natural-language "plan" into machine-addressable steps. If the model returns invalid JSON or an empty list, the node falls back to a single subtask equal to the original question, ensuring the workflow still produces an answer instead of failing.

The sequential "least-to-most" execution happens in `solve_next` plus `route.solve_next` reads the current `idx` and pulls exactly one subtask from `subtasks[idx]`. It invokes the model with both the original request and the current subtask, which preserves global context while focusing the model on one bounded problem. The node appends the result to `solutions` and returns `idx`: 1. Because `idx` is defined with an additive reducer (`operator.add`), the loop advances by accumulation rather than mutation, which keeps each step's update explicit and traceable. A defensive guard returns `{}` when `idx` exceeds the list length; this prevents out-of-range access and lets control flow decide the next move.

`route` is the loop controller: it compares `idx` to the number of subtasks and either loops back to `solve_next` or transitions to `combine`. Finally, `combine` formats the accumulated `solutions` into a numbered list and asks the model to produce a single customer-facing response. This separation is deliberate: intermediate results stay structured for the workflow, while the final step optimizes for readability and tone.

### 8.5.5 Verification

Verification is a two-phase pipeline: generate an answer, then run a checker node that validates claims and either accepts the result or routes into a repair step. In code, verification is all about splitting responsibility. One node is allowed to be fast and helpful ("draft the best answer you can"). A second node is not trying to be helpful; it is trying to be skeptical ("does this answer contain unsupported claims, missing prerequisites, or policy violations?"). If the checker is happy, the workflow ends. If not, the workflow routes into a repair step that rewrites the answer using the checker's feedback. Conceptually, the graph has three work nodes:

1. generate: produces an initial customer-facing answer.
2. check: evaluates that answer against explicit criteria and returns a structured verdict.
3. repair: only runs when needed; rewrites the answer to address issues flagged by the checker.

The important design choice is that the checker returns a machine-readable decision (for example JSON with `'ok': true/false`` and a list of issues). That makes the control flow deterministic: the router doesn't have to "interpret vibes" from free text. Below is a sample that follows the same script style as our previous examples. It stays intentionally minimal: it verifies support-style correctness (no invented order facts, asks for missing info, clear next steps), not external truth via tools.

```
from __future__ import annotations
```

```

import json
from typing import Literal
from typing_extensions import TypedDict
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from langgraph.graph import StateGraph, START, END
from .config import OPENAI_API_KEY

--- State ---
class SupportState(TypedDict, total = False):
 question: str
 answer: str

 # Verification additions
 verdict_ok: bool
 issues: list[str]

--- Model ---
model = ChatOpenAI(model = "gpt-4o-mini", temperature = 0.2)

--- Prompts ---
generate_prompt = ChatPromptTemplate.from_messages(
 [
 ("system", "You are a careful e-commerce support assistant."),
 ("human", "{question}")
]
)

check_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a strict quality checker for an e-commerce support assistant.\n"
 "Evaluate the draft answer against these rules:\n"
 "1) Do not invent order/shipping/account facts.\n"
 "2) If info is missing, explicitly ask for what is needed.\n"
 "3) Provide clear, actionable next steps.\n"
 "Return ONLY valid JSON in this exact shape:\n"
 '{{ "ok": true/false, "issues": ["..."] }}'
),
 ("human", "Customer question:\n{question}\n\nDraft answer:\n{answer}")
]
)

repair_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are a careful e-commerce support assistant.\n"
 "Rewrite the answer to fix the issues listed by the checker.\n"
 "Do not invent facts. If required info is missing, ask for it.\n"
 "Return a final customer-facing answer.\n"
),
 (
 "human",
 "Customer question:\n{question}\n\nDraft answer:\n{answer}\n\nIssues:\n{issues}"
)
]
)

--- Nodes ---
def generate(state: SupportState) -> dict:
 msg = (generate_prompt | model).invoke({"question": state["question"]})
 return {"answer": msg.content.strip()}

def check(state: SupportState) -> dict:
 msg = (check_prompt | model).invoke(
 {"question": state["question"], "answer": state.get("answer", "")}
)
 text = msg.content.strip()

 # Make the routing decision machine-readable.
 try:
 data = json.loads(text)
 ok = bool(data.get("ok", False))
 issues = data.get("issues", [])
 if not isinstance(issues, list):
 issues = [str(issues)]
 issues = [str(i).strip() for i in issues if str(i).strip()]
 except Exception:
 # Conservative fallback: if parsing fails, treat as not OK and request repair.

```

```

ok = False
issues = ["Checker returned invalid JSON; answer must be reviewed and
rewritten."]

return {"verdict_ok": ok, "issues": issues}

def route_after_check(state: SupportState) -> Literal["accept", "repair"]:
 return "accept" if state.get("verdict_ok", False) else "repair"

def repair(state: SupportState) -> dict:
 issues_text = "\n".join(f"- {i}" for i in state.get("issues", [])) or "- (no
details)"
 msg = (repair_prompt | model).invoke(
 {
 "question": state["question"],
 "answer": state.get("answer", ""),
 "issues": issues_text
 }
)
 return {"answer": msg.content.strip()}

--- Graph ---
builder = StateGraph(SupportState)

builder.add_node("generate", generate)
builder.add_node("check", check)
builder.add_node("repair", repair)

builder.add_edge(START, "generate")
builder.add_edge("generate", "check")

builder.add_conditional_edges("check", route_after_check, {"accept": END, "repair": "repair"})
builder.add_edge("repair", END)

graph = builder.compile()

if __name__ == "__main__":
 out = graph.invoke(
 {
 "question": "My package says delivered but I can't find it. What should I
do?"
 }
)
 print(out["answer"])

```

ch\_08\scr\reasoning\_verification.py

The graph wires three nodes into a strict “produce → evaluate → possibly rewrite” flow. Execution always starts in `generate`, which invokes the model with the user’s question and stores a trimmed draft in `answer`. That draft is then passed unchanged into `check`, which runs a separate prompt that must emit a JSON object. The code treats that JSON as a contract: it calls `json.loads`, extracts `ok`, and normalizes `issues` into a list of clean strings (including the case where the model returns a single string or another unexpected type). This normalization step is what keeps downstream logic stable—routing and repair formatting never have to guess what shape `issues` is.

If JSON parsing fails for any reason, `check` forces `ok = False` and injects a synthetic issue explaining that the checker output was invalid. This ensures the workflow does not silently “pass” when the checker response is malformed.

`route_after_check` is a small but important adapter: it maps the boolean verdict into the symbolic branch labels used by `add_conditional_edges`. If the verdict is negative, the graph executes `repair`, which formats the issues as bullet points and prompts the model to rewrite the draft with those constraints. The repaired text overwrites `answer` and the graph terminates, returning the final response.

### 8.5.6 Reflection

Reflection (critique) is an iterative loop: generate, critique, revise, and stop when a router decides the output is good enough or the budget is exhausted. In code, reflection is a controlled feedback loop around a normal “draft an answer” step. The workflow produces an initial response, asks for a targeted critique against your quality bar (policy compliance, missing assumptions, clarity, actionability), then revises using that critique. The loop only continues while it is still buying you something. Once the critique says the draft is acceptable—or you hit a fixed iteration budget—the graph stops and returns the best draft it has. The important implementation detail is the router. Without it, reflection becomes an infinite “improve forever” machine. With it, the loop is predictable: you can cap cost/latency and you can log each critique to see what the system is actually fixing.

Below is a continuation in the same script style as your base sample. It keeps the same overall shape you’ve been using: state carries the draft and critique, nodes return updates, and a conditional edge drives the loop.

```

from __future__ import annotations
import os
import operator
from typing import Annotated, Literal
from typing_extensions import TypedDict
from langchain_core.prompts import ChatPromptTemplate
from langchain_openai import ChatOpenAI
from .config import OPENAI_API_KEY
from langgraph.graph import StateGraph, START, END

--- State ---
class SupportState(TypedDict, total = False):
 question: str
 answer: str

 # Reflection additions
 draft: str
 critique: str
 iterations: Annotated[int, operator.add]
 max_iterations: int

--- Model ---
model = ChatOpenAI(model = "gpt-4o-mini", temperature = 0.2)

--- Prompts ---
draft_prompt = ChatPromptTemplate.from_messages(
 [
 ("system", "You are a careful e-commerce support assistant."),
 ("human", "{question}")
]
)
critique_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "You are reviewing a customer support answer.\n"
 "Critique it for: clarity, completeness, policy safety, and actionability.\n"
 "If the answer is acceptable, start your response with exactly: OK\n"
 "Otherwise start with exactly: FIX and then list the issues.\n"
),
 ("human", "Customer request:\n{question}\n\nDraft answer:\n{draft}")
]
)
revise_prompt = ChatPromptTemplate.from_messages(
 [
 (
 "system",
 "Revise the draft using the critique.\n"
 "Keep the response customer-facing (no internal process notes).\n"
 "If the critique starts with OK, return the draft unchanged.\n"
),
]
)

```

```

 ("human", "Customer
 request:\n{question}\n\nDraft:\n{draft}\n\nCritique:\n{critique}")
)
}

--- Nodes -----
def generate(state: SupportState) -> dict:
 msg = (draft_prompt | model).invoke({"question": state["question"]})
 return {"draft": msg.content.strip()}

def critique(state: SupportState) -> dict:
 msg = (critique_prompt | model).invoke(
 {"question": state["question"], "draft": state.get("draft", "")})
 return {"critique": msg.content.strip()}

def revise(state: SupportState) -> dict:
 msg = (revise_prompt | model).invoke(
 {
 "question": state["question"],
 "draft": state.get("draft", ""),
 "critique": state.get("critique", ""),
 }
)
 # Count one full reflection cycle (critique+revise) as one iteration.
 return {"draft": msg.content.strip(), "iterations": 1}

def should_continue(state: SupportState) -> Literal["critique", "finalize"]:
 # Stop when:
 # - the critique says OK, or
 # - we reached max_iterations (budget exhausted).
 max_iters = state.get("max_iterations", 2)
 iters = state.get("iterations", 0)
 crit = (state.get("critique", "") or "").lstrip()

 if crit.startswith("OK"):
 return "finalize"
 if iters >= max_iters:
 return "finalize"
 return "critique"

def finalize(state: SupportState) -> dict:
 # The final answer is the latest draft.
 return {"answer": state.get("draft", "")}

--- Graph -----
builder = StateGraph(SupportState)

builder.add_node("generate", generate)
builder.add_node("critique", critique)
builder.add_node("revise", revise)
builder.add_node("finalize", finalize)

builder.add_edge(START, "generate")
builder.add_edge("generate", "critique")
builder.add_edge("critique", "revise")
builder.add_conditional_edges("revise", should_continue, {"critique": "critique",
 "finalize": "finalize"})
builder.add_edge("finalize", END)

graph = builder.compile()

if __name__ == "__main__":
 out = graph.invoke(
 {
 "question": "My package says delivered but I can't find it. What should I do?",
 "iterations": 0,
 "max_iterations": 2
 }
)
 print(out["answer"])

```

ch\_08\scr\reasoning\_reflection.py

The workflow is implemented as four graph nodes wired into a closed loop that can iterate a bounded number of times. `generate` is the entry point: it produces the first draft from the user's question and stores it in `draft`. This separation is intentional—by persisting the draft

in state, later nodes can evaluate and revise the exact same text rather than regenerating from scratch.

`critique` then evaluates the current draft using a strict signaling convention: the response must begin with `OK` (acceptable) or `FIX` (needs work). This gives the control-flow logic a stable, machine-checkable decision signal without requiring fragile natural-language parsing.

`revise` consumes both the current `draft` and the latest `critique`, generating an improved `draft` and incrementing `iterations`. The increment uses an additive reducer, which means each loop execution records progress explicitly as state updates, making the iteration count reliable even as the graph merges updates across steps.

Control flow is centralized in `should_continue`, which inspects two stop conditions: whether the critique begins with `OK`, and whether the iteration budget (`max_iterations`) has been exhausted. This function is attached as a conditional edge after `revise`, so the graph always performs `critique` → `revise` as a unit, then decides whether another review cycle is justified.

Finally, `finalize` copies the most recent `draft` into `answer`. This last step decouples the internal working text (`draft`) from the external contract (`answer`), so you can later extend the workflow (for example, storing critiques for auditing) without changing what the caller reads.

## 8.6 SHORT-CONTEXT STRATEGIES: MAP–REDUCE SUMMARIZATION, LONG-VIDEO AND LONG-DOCUMENT WORKFLOWS

Even with large-context models, “stuff everything into one prompt” is not a dependable default. Context windows remain finite, latency and cost grow with every token, and many real deployments still target smaller or cheaper models. For long documents and media (audio/video), practical constraints—upload limits, transcription/segmentation costs, preprocessing time, and tool/SDK limits—can become the bottleneck even before you hit the model’s context window.

Short-context strategies treat an oversized input as a collection of smaller units. You process those units across multiple calls, then carry forward only compact artefacts (summaries, extracted facts, timelines, or structured fields) as you progress. In practice, you will see three common modes referenced in LangChain’s summarization chain examples and discussions: “`stuff`”, “`map_reduce`”, and “`refine`”.

- “`Stuff`” is the baseline: concatenate all selected inputs that fit into the model’s context window (for example, retrieved documents or chunks), place them directly into a single prompt, and ask for the summary or answer in one model call. This is the simplest approach and preserves the full text of what you included, but cost and latency grow with token count, and it fails abruptly once the inputs no longer fit. In practice, you usually need to cap or compress inputs before stuffing (top-k retrieval, filtering, chunking, or pre-summaries).
- “`Map_reduce`” splits the input into chunks, summarises each chunk independently (map), then synthesises a global result from the chunk summaries (reduce).

- “Refine” is sequential: build an initial summary from the first chunk, then iterate through later chunks to update that summary.

LangGraph expresses the same ideas as explicit, stateful workflows. Instead of a single linear pipeline that can hide intermediate steps, you can model chunking, mapping, aggregation, and synthesis as explicit nodes, so intermediate state and outputs are easier to inspect. This matters when you need to inspect failures, resume long runs, or apply strict output constraints (for example via structured outputs, validation, and retry/fallback nodes).

### 8.6.1 Map-reduce summarization as a graph pattern

Map-reduce summarisation is a general strategy for turning an oversized input into a reliable, bounded workflow. The point is not the specific nodes or edges, but the discipline it enforces: break the problem down, control how partial results are combined, and keep the final decision step small enough to be stable.

1. Decomposition is the core idea. A long document is hard to summarise in one pass because the model must attend to too much at once and will either miss details or over-compress. Map-reduce reframes the task as two compressions. First, each chunk is summarised locally (“map”). Then those summaries are combined into a single global summary (“reduce”). This mirrors how humans work: make notes section by section, then write an abstract from the notes.
2. Parallel work requires explicit merge rules. When multiple workers produce outputs for the same shared result, you cannot rely on “whatever finishes last.” The system needs a defined way to merge updates into shared state, such as “append all per-chunk summaries into a list.” This is the broader principle: concurrency is safest when your shared state has deterministic merge semantics.
3. Execution order is not guaranteed in parallel steps, so if the final synthesis depends on document order, you should carry ordering in the data. In parallel systems, outputs can arrive in any order. If the final synthesis depends on document order, each chunk summary needs an ordering key (chunk index, page, character offsets). The reduce step can then sort before it synthesises. That shift—treating order as a property of the data, not the runtime—is what makes the workflow robust.
4. The reduce step should usually operate on compressed representations rather than the raw document. The goal is to keep the final prompt small and predictable. When reduce consumes only per-chunk summaries, the final synthesis has bounded input size and a clearer evidence trail. This reduces token blowups and makes failures easier to diagnose, because you can inspect intermediate summaries rather than re-reading the entire document.
5. Chunking and summary contracts determine quality. Chunk size is a trade-off: small chunks are easier to summarise accurately but can lose cross-chunk context; large chunks preserve context but reintroduce overload. Overlap can help, but it must be controlled to avoid duplication. Separately, the per-chunk summary format should be consistent and structured enough that reduce can combine outputs without guessing—think “key points, entities, decisions, open questions,” or whatever fits your domain.
6. Map-reduce is audit-friendly by design. Because intermediate outputs are explicit artifacts, you can trace where the final summary came from. When something is wrong, you can

locate whether the error originated in a specific chunk summary, in the reduce synthesis, or in a boundary choice during chunking. That makes the system debuggable rather than mystical.

Summarising a long document is just one case of a broader method: split an oversized problem into independent units, produce standardized intermediate results, merge them deterministically, and run a final synthesis on the compressed set. This same logic applies to large-scale extraction, multi-document synthesis, batch classification, and evidence-first decision making.

The following example shows a simple way to summarise a very long piece of text without trying to digest it all in one go. It treats the document like a stack of pages: first it cuts the text into manageable slices, then it produces a short “mini-summary” for each slice, and finally it stitches those mini-summaries together into one overall result in the original order. The goal is to keep the process predictable and scalable, so the same approach still works when the input grows from a few paragraphs to something book-length. Before running this example, please install beautifulsoup4:

```
(.venv) C:\Users\alexc\llm-app\ch_08> python -m pip install beautifulsoup4
```

```
from __future__ import annotations
import operator
from typing import Annotated, List, Tuple
from typing_extensions import TypedDict
from langgraph.graph import START, END, StateGraph
from langgraph.types import Send
import requests
from bs4 import BeautifulSoup

class ChunkSummary(TypedDict):
 chunk_id: int
 summary: str

class SummarizeState(TypedDict, total = False):
 # Inputs
 text: str
 chunk_size: int
 chunk_overlap: int
 # Derived
 chunks: List[str]
 # Map outputs (many parallel workers update this key)
 chunk_summaries: Annotated[List[ChunkSummary], operator.add]
 # Final output
 final_summary: str

def _split_text(text: str, chunk_size: int, chunk_overlap: int) -> List[str]:
 if chunk_size <= 0:
 raise ValueError("chunk_size must be > 0")
 if chunk_overlap < 0:
 raise ValueError("chunk_overlap must be >= 0")
 if chunk_overlap >= chunk_size:
 raise ValueError("chunk_overlap must be < chunk_size")

 chunks: List[str] = []
 start = 0
 text = text or ""

 while start < len(text):
 end = min(len(text), start + chunk_size)
 chunks.append(text[start:end])
 if end == len(text):
 break
 start = end - chunk_overlap

 return chunks

def chunk_document(state: SummarizeState) -> dict:
 text = state.get("text", "")
 chunk_size = int(state.get("chunk_size", 800))
```

```

chunk_overlap = int(state.get("chunk_overlap", 80))
chunks = _split_text(text, chunk_size = chunk_size, chunk_overlap = chunk_overlap)
return {"chunks": chunks}

def continue_to_map(state: SummarizeState) -> List[Send]:
 sends: List[Send] = []
 for i, chunk in enumerate(state.get("chunks", [])):
 sends.append(Send("summarize_chunk", {"chunk_id": i, "chunk_text": chunk}))
 return sends

def fetch_webpage_text(url: str, timeout: int = 20) -> str:
 resp = requests.get(
 url,
 timeout = timeout,
 headers = {"User-Agent": "Mozilla/5.0 (compatible; LangGraphSummarizer/1.0)"}
)
 resp.raise_for_status()
 soup = BeautifulSoup(resp.text, "html.parser")

 # Remove script/style/noscript to reduce junk.
 for tag in soup(["script", "style", "noscript"]):
 tag.decompose()

 # Basic text extraction. (For article-like pages, Option B is better.)
 text = soup.get_text(separator = "\n")

 # Normalize whitespace a bit
 lines = [ln.strip() for ln in text.splitlines()]
 lines = [ln for ln in lines if ln]
 return "\n".join(lines)

class ChunkTaskState(TypedDict):
 chunk_id: int
 chunk_text: str

 def __cheap_deterministic_summary(self, text: str, max_words: int = 40) -> str:
 # A deterministic summariser so the example runs without external model calls.
 # Replace this with an LLM call in production.
 words = (text or "").strip().split()
 if not words:
 return ""
 return " ".join(words[:max_words]).strip()

 def summarize_chunk(self) -> dict:
 chunk_id = int(self["chunk_id"])
 chunk_text = self["chunk_text"]
 summary = __cheap_deterministic_summary(chunk_text, max_words = 40)
 return {"chunk_summaries": [{"chunk_id": chunk_id, "summary": summary}]}

 def reduce_summaries(self) -> dict:
 items = self.get("chunk_summaries", []) or []
 # LangGraph notes that updates from parallel branches may not be consistently
 # ordered.
 # Sort by chunk_id to enforce stable, reproducible synthesis.
 items_sorted = sorted(items, key = lambda x: int(x["chunk_id"]))

 combined = []
 for item in items_sorted:
 s = (item.get("summary") or "").strip()
 if s:
 combined.append(f"{'chunk {item['chunk_id']}'} {s}")

 final = "\n".join(combined).strip()
 return {"final_summary": final}

 def build_graph():
 builder = StateGraph(SummarizeState)
 builder.add_node("chunk_document", chunk_document)
 builder.add_node("summarize_chunk", summarize_chunk)
 builder.add_node("reduce_summaries", reduce_summaries)

 builder.add_edge(START, "chunk_document")
 # Dynamic fan-out using Send
 builder.add_conditional_edges("chunk_document", continue_to_map, ["summarize_chunk"])
 builder.add_edge("summarize_chunk", "reduce_summaries")
 builder.add_edge("reduce_summaries", END)
 return builder.compile()

 def main() -> None:
 graph = build_graph()

 url = "https://qdrant.tech/documentation/concepts/collections/"
 long_text = fetch_webpage_text(url)

 # You can control concurrency when invoking; LangGraph documents max_concurrency as a

```

```

config option.
For portability across versions, pass it as part of the second argument dict.
result = graph.invoke(
 {"text": long_text, "chunk_size": 300, "chunk_overlap": 30},
 {
 "max_concurrency": 8,
 "configurable": {
 "thread_id": "thread-A", # if you use checkpointing/threads
 "user_id": "user-42", # if your nodes read config
 # ["configurable"]["user_id"]
 }
 }
)
if __name__ == "__main__":
 main()

```

ch\_08\scr\graph\_map\_reduce.py

At runtime, `main()` fetches a real webpage and converts it to plain text, stripping scripts/styles and normalizing whitespace. That gives the graph a single, oversized input string that behaves like a long document.

The workflow starts in `chunk_document`, which slices the input into overlapping windows. The overlap exists to reduce boundary loss when a sentence or idea spans two chunks, while the validation guards prevent invalid configurations (negative overlap or overlap larger than the chunk).

Next, `continue_to_map` performs dynamic routing: it emits one `Send` instruction per chunk, creating a parallel “map” phase where each worker receives only its chunk plus a stable `chunk_id`. Each `summarize_chunk` node produces a per-chunk artifact and updates shared state via `chunk_summaries`, which is configured with `operator.add` so parallel updates are merged by list concatenation rather than overwriting.

`reduce_summaries` then performs the fan-in. Because parallel updates may arrive in non-deterministic order, it sorts by `chunk_id` before assembling the final output, guaranteeing stable, reproducible synthesis. The result is a predictable pipeline: bounded chunk inputs, explicit intermediate artifacts, and a deterministic final assembly step.

### 8.6.2 Long-document workflows in practice

A production long-document workflow usually adds three things on top of the basic map-reduce skeleton.

First, ingestion normalises sources. You typically convert files, pages, or scraped content into a consistent representation (text plus metadata such as source, page, section, and timestamp). That metadata becomes the backbone of traceability: you can carry chunk identifiers through the map phase and later connect the final output back to its origins.

Second, chunking becomes model- and task-aware. `Chunk_size` is not “the maximum the model can take”. It is “the maximum that remains safe once you include instructions, formatting constraints, and any additional context you must attach”. `Chunk_overlap` is not a magic number either; it is there to avoid boundary loss for definitions and constraints that span chunk edges. You tune both to the shape of your documents and the strictness of your output requirements.

Third, extremely long inputs often need hierarchical reduction. If you produce thousands of chunk summaries, the reduce step can become its own long-context problem. The standard

remedy is multi-level reduction: reduce summaries in batches, then reduce the batch summaries into a final result. In a graph, that is either multiple reduce nodes or a loop that keeps collapsing until the summary list fits your budget. When you build it as a graph, you can checkpoint between stages and resume from the most recent checkpoint, which typically avoids recomputing earlier stages. In LangGraph, persistence is enabled by supplying a checkpointer when compiling the graph.

### 8.6.3 Long-video workflows

Video summarization is the same “short-context” problem but stretched across time. A useful summary has to respect when things happened, because the important moments are scattered over minutes or hours. So before you ask a model to summarise, you first turn the video into a time-indexed text representation.

A practical approach is transcript-first. You transcribe the audio into short text segments, each with start/end timestamps. Those timestamped segments then play the same role as document chunks in a classic map–reduce workflow: the map step processes each segment (or a small sliding window of segments) to extract “what matters here”, and the reduce step combines those local extracts into a single output. If your transcription or storage layer has file-size or duration limits, you naturally split long recordings into multiple parts and run the same map step over each part, then merge the results in reduce.

The video-specific twist is the shape of the reduce output: you want it to preserve time. Instead of a generic “write a summary”, ask for something like a chronological list of key moments with timestamps, a set of chapters with time ranges, or decisions and action items annotated with when they occurred. For long recordings, a timeline-style reduce output helps reduce the risk that the model blends details from different parts of the video, and it makes it easier for downstream tools (or users) to jump back to the relevant segment that supports each point.

The following example shows a simple way to summarise a long transcript without losing the sense of time. It treats a video transcript as a sequence of small, timestamped snippets, then breaks them into manageable time windows. Each window is summarised on its own, and the partial summaries are stitched back together into a final “timeline” where every line keeps the start and end time of the portion it came from. The result reads like a set of chapter notes with timestamps, so you can scan the key moments and still jump back to the exact part of the video that matters.

```
from __future__ import annotations
import operator
from typing import Annotated, List
from typing_extensions import TypedDict
from langgraph.graph import START, END, StateGraph
from langgraph.types import Send

class Segment(TypedDict):
 start_s: float
 end_s: float
 text: str

class IntervalSummary(TypedDict):
 interval_id: int
 start_s: float
 end_s: float
 summary: str

class VideoState(TypedDict, total = False):
 # Inputs
```

```

segments: List[Segment]
max_chars_per_window: int
Derived windows
windows: List[List[Segment]]
Map outputs
interval_summaries: Annotated[List[IntervalSummary], operator.add]
Final output
timeline: str

def group_segments(state: VideoState) -> dict:
 segments = state.get("segments", []) or []
 max_chars = int(state.get("max_chars_per_window", 800))

 windows: List[List[Segment]] = []
 current: List[Segment] = []
 current_len = 0

 for seg in segments:
 seg_text = (seg.get("text") or "").strip()
 if not seg_text:
 continue
 seg_len = len(seg_text)

 # Start a new window if adding this segment would exceed the budget.
 if current and (current_len + seg_len) > max_chars:
 windows.append(current)
 current = []
 current_len = 0

 current.append(seg)
 current_len += seg_len

 if current:
 windows.append(current)

 return {"windows": windows}

def continue_to_map(state: VideoState) -> List[Send]:
 sends: List[Send] = []
 for i, window in enumerate(state.get("windows", [])):
 sends.append(Send("summarize_interval", {"interval_id": i, "window": window}))
 return sends

class IntervalTaskState(TypedDict):
 interval_id: int
 window: List[Segment]

def _deterministic_interval_summary(window: List[Segment], max_words: int = 35) -> str:
 joined = " ".join((seg.get("text") or "").strip() for seg in window).strip()
 words = joined.split()
 return ".join(words[:max_words]).strip()

def summarize_interval(state: IntervalTaskState) -> dict:
 interval_id = int(state["interval_id"])
 window = state["window"]

 start_s = float(window[0]["start_s"])
 end_s = float(window[-1]["end_s"])
 summary = _deterministic_interval_summary(window, max_words = 35)

 return {
 "interval_summaries": [
 {
 "interval_id": interval_id,
 "start_s": start_s,
 "end_s": end_s,
 "summary": summary
 }
]
 }

def build_timeline(state: VideoState) -> dict:
 items = state.get("interval_summaries", []) or []
 items_sorted = sorted(items, key = lambda x: int(x["interval_id"]))

 lines = []
 for it in items_sorted:
 lines.append(f"{it['start_s']:.1f}s-{it['end_s']:.1f}s: {it['summary']}")

 return {"timeline": "\n".join(lines).strip()}

def build_graph():
 builder = StateGraph(VideoState)
 builder.add_node("group_segments", group_segments)
 builder.add_node("summarize_interval", summarize_interval)

```

```

builder.add_node("build_timeline", build_timeline)

builder.add_edge(START, "group_segments")
builder.add_conditional_edges("group_segments", continue_to_map,
 ["summarize_interval"])
builder.add_edge("summarize_interval", "build_timeline")
builder.add_edge("build_timeline", END)

return builder.compile()

def main() -> None:
 # Simulated transcript segments (in practice produced by a transcription system).
 segments: List[Segment] = [
 {"start_s": 0.0, "end_s": 8.0, "text": "Welcome back. Today we will reset a
 password and review account security."},
 {"start_s": 8.0, "end_s": 18.0, "text": "First, open settings. Then choose
 security. Enable two-factor authentication."},
 {"start_s": 18.0, "end_s": 30.0, "text": "If you lost access to your phone, use
 recovery codes or contact support."},
 {"start_s": 30.0, "end_s": 45.0, "text": "Finally, confirm the email address and
 sign out of other devices."}
]

 graph = build_graph()
 result = graph.invoke(
 {"segments": segments, "max_chars_per_window": 120},
 {"configurable": {"max_concurrency": 4}}
)

 print(result["timeline"])

if __name__ == "__main__":
 main()

```

ch\_08\scr\video\_summary.py

First, `group_segments` turns the raw transcript segments into windows. It walks the ordered segments and packs adjacent ones together until the total text would exceed `max_chars_per_window`, then starts a new window. This is a stand-in for “fit the chunk into model context”.

Next, `continue_to_map` fans out one summarisation task per window. It creates a `Send` call to `summarize_interval` for each window, passing `interval_id` plus the window content. Because each task is independent, the runtime can run them concurrently.

`summarize_interval` is the map step. For each window it computes the time range from the first and last segment (`start_s`, `end_s`) and produces a short summary for that interval. In the sample this is deterministic word-truncation, but the shape matches a real LLM call: “summarise what happens in this time slice”.

`interval_summaries` uses `Annotated[..., operator.add]` so results from multiple map tasks are appended, not overwritten. That reducer is required because many parallel tasks write to the same state key.

Finally, `build_timeline` is the reduce step. It sorts `interval_summaries` by `interval_id` (to undo any non-deterministic completion order) and formats one line per interval with timestamps. The output is a chronological, timestamped summary rather than a free-form paragraph.

#### 8.6.4 State, memory, and trade-offs

Map-reduce style workflows keep a lot of state on purpose. The state is not only “the latest answer”. It is a running record of what happened during the workflow: which chunks (or time intervals) were created, which ones were processed, what intermediate summaries were produced, and what final artefact was generated. When you persist or trace that record, it

makes the pipeline transparent. If the final output is missing an important point, you can trace where it disappeared: it was never captured in the map step, or it was dropped during the reduce/synthesis step.

The downside is cost. You often pay in tokens (and sometimes in wall-clock latency) because you replace one large model call with many smaller calls plus a final synthesis. However, if the map step runs concurrently, wall-clock latency can stay similar—or even improve—while total compute cost still increases. This approach becomes worthwhile when the input is too large to handle comfortably in one prompt, when you need a persisted/traceable trail of intermediate artefacts, or when quality depends on broad, consistent coverage rather than a few isolated excerpts. For extremely large inputs, you may also need hierarchical reduction (reduce in batches, then reduce the batch summaries) to keep the final synthesis bounded.

## 8.7 SUMMARY

This chapter explains how LangGraph helps you build agent workflows that behave like software systems instead of improvisational prompt blobs. The main idea is simple: you model the application as a graph of small steps (nodes) connected by edges, all operating on an explicit state object. The graph is the control flow, and state is the shared working memory. Once you make those two things concrete, you can reason about the system, test it, and debug it when it fails.

A recurring theme is that “reasoning” is not a special capability you either have or don’t have. In production, reasoning is mostly the result of structure. The chapter shows how you can shape reasoning by shaping workflow. You can route requests to different paths, loop over a plan/act/check cycle, retry with repaired inputs, and stop when you meet a clear success condition. Instead of asking the model to hold everything in its head, you decide what each step is responsible for and what the next step should do with the result. That makes the system easier to extend: adding a verification pass or a fallback path is a graph change, not a rewrite of a giant prompt.

State is the glue that makes this reliable. It is not only “the final answer.” State is the record of what happened: the user’s request, messages exchanged, retrieved evidence, intermediate summaries, control flags, and diagnostics. This is where you keep the working notes that you do not want to show to the user but that you do want for transparency and debugging. When the final output is wrong or missing something important, you can trace the failure: did the relevant point never get captured, did retrieval miss it, or did synthesis drop it? That kind of forensic analysis is only possible when you keep intermediate artifacts on purpose.

The chapter also emphasizes that state is not a free-form bag of values. You typically define a schema (for example, a `TypedDict` or `dataclass`) so everyone agrees what keys exist and what they mean. Nodes treat incoming state as read-only and return partial updates as dictionaries. This “read state, write updates” style keeps node logic small and makes it easier to test each step in isolation.

Once you allow parallelism and fan-out, you need deterministic merge rules. LangGraph executes in steps and merges all node updates for that step. If two nodes update the same key, “last write wins” is usually not what you want. The chapter explains reducers: per-key merge functions that define how to combine concurrent updates safely (for example, appending messages or accumulating partial results). Reducers are what make map-reduce patterns, multi-retriever fan-out, and other concurrent workflows predictable rather than flaky.

Durability is the next jump from “cool demo” to “real system.” With checkpointing, you can persist state so a workflow can resume across turns, survive restarts, and support pause-and-resume flows. Interrupts let the graph stop at a controlled point to request an external decision (often a human approval) and then resume using the saved state instead of reconstructing context from scratch. This is powerful, but it comes with an operational lesson: any node that performs side effects (sending an email, charging a card, writing to a database) must be designed with idempotency in mind, because a resumed or retried step may run again.

The chapter also separates two meanings of “memory.” Checkpoints preserve the evolving context of a single thread: one conversation or one job. Long-term memory is different: it is a store of durable facts that should survive beyond any single run and be available across threads. The chapter introduces a pattern where you write facts to a store using namespaces and identifiers and then retrieve them either by direct key lookup or by search (including semantic similarity when you want retrieval “by meaning”). Keeping checkpoints and stores separate prevents accidental bloat and keeps behavior explainable.

Finally, the chapter ties the reasoning and memory model to the practical problem every LLM system hits: context limits. It introduces short-context strategies for handling large inputs, and frames them as design trade-offs. “Stuff” tries to fit everything into one prompt; it is simple but fails when inputs are large. “Refine” processes input iteratively, updating a running summary; it can work well when order matters and you want gradual accumulation. “Map\_reduce” splits input into chunks, summarizes each chunk (map), and then synthesizes those summaries (reduce). The key benefit is broad coverage and an inspectable trail of intermediate artifacts. The cost is usually higher total tokens and more model calls, even if wall-clock time stays reasonable when the map phase runs concurrently. For extremely large inputs, the chapter motivates hierarchical reduction (reduce in batches, then reduce again) so the final synthesis stays bounded.

The takeaway is that LangGraph makes the hard parts explicit: reasoning becomes workflow structure, memory becomes managed state plus persistence, and context limits become engineering trade-offs you can measure and tune.