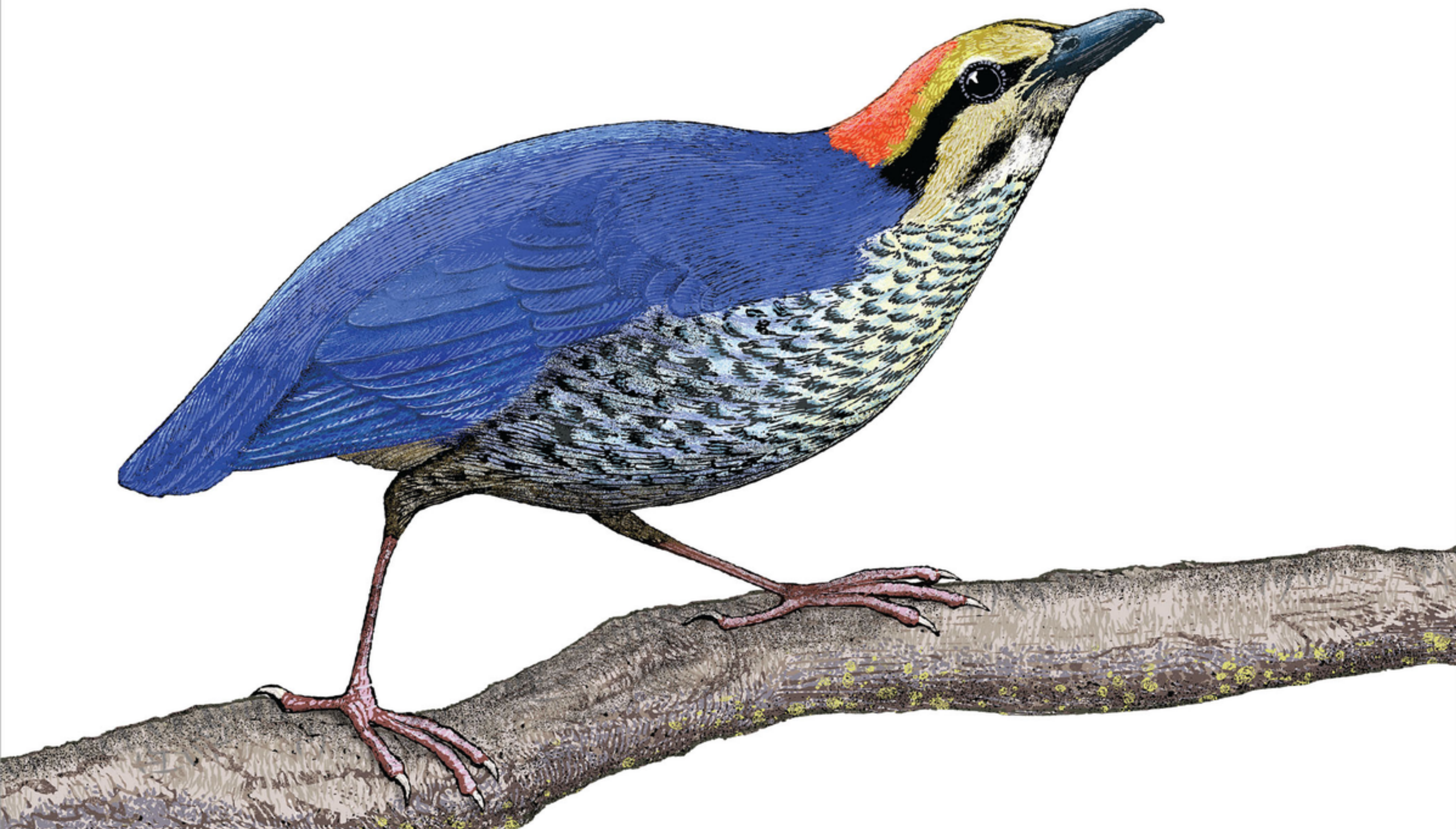
O'REILLY®

Generative AI Design Patterns

Solutions to Common Challenges When Building GenAI Agents and Applications



Valliappa Lakshmanan
& Hannes Hapke

Generative AI Design Patterns

Solutions to Common Challenges When Building GenAI Agents and Applications

Valliappa Lakshmanan and Hannes Hapke

Generative AI Design Patterns

by Valliappa Lakshmanan and Hannes Hapke

Printed in the United States of America.

Published by O’Reilly Media, Inc., 141 Stony Circle, Suite 195, Santa Rosa, CA 95401.

O’Reilly books may be purchased for educational, business, or sales promotional use. Online editions are also available for most titles (*http://oreilly.com*). For more information, contact our corporate/institutional sales department: 800-998-9938 or *corporate@oreilly.com*.

Acquisitions Editor: Nicole Butterfield

Development Editor: Sarah Grey

Production Editor: Christopher Faucher

Copyeditor: Doug McNair

Proofreader: Emily Wydeven

Indexer: Sue Klefstad

Cover Designer: Susan Thompson

Cover Illustrator: Susan Brown

Interior Designer: David Futato

Interior Illustrator: Kate Dullea

October 2025: First Edition

Revision History for the First Edition

- 2025-10-03: First Release

See *http://oreilly.com/catalog/errata.csp?isbn=9798341622661* for release details.

979-8-341-62266-1

[LSI]

Preface

If you're an AI engineer building generative AI (GenAI) applications, you've likely experienced the frustrating gap between the ease of creating impressive prototypes and the complexity of deploying them reliably in production. While foundational models make it easy to build compelling demos, production systems demand solutions to fundamental challenges: hallucinations that compromise accuracy, inconsistent outputs that break downstream processes, knowledge gaps that limit enterprise applicability, and reliability issues that make systems unsuitable for critical applications.

This book bridges that gap by providing 32 battle-tested design patterns that address the recurring problems you'll encounter when building production-grade GenAI applications. These patterns aren't theoretical constructs—they codify proven solutions that are often derived from cutting-edge research and refined by practitioners who have successfully deployed GenAI systems at scale.

Supervised machine learning (ML) involves training a problem-specific model on a large training dataset of example inputs and outputs—but GenAI applications rarely include a training phase. Instead, they commonly use general-purpose foundational models. This book is focused on design patterns for AI applications that are built on top of foundational models, such as Open AI's GPT, Anthropic's Claude, Google's Gemini, or Meta's Llama.

In this book, we cover the entire AI engineering workflow. After an introduction in Chapter 1, Chapter 2 provides practical patterns for controlling content style and format (including Logits Masking [Pattern 1] and Grammar [Pattern 2]). Chapter 3 and Chapter 4 cover integrating external knowledge through sophisticated retrieval-augmented generation (RAG) implementations, from Basic RAG (Pattern 6) to Deep Search (Pattern 12). Chapter 5 is about enhancing your model's reasoning capabilities with patterns like Chain of Thought (Pattern 13), Tree of

Thoughts (Pattern 14), and Adapter Tuning (Pattern 15). Chapter 6 emphasizes building reliable systems with LLM-as-Judge (Pattern 17), Reflection (Pattern 18), and Prompt Optimization (Pattern 20) patterns. Chapter 7 is about creating agentic systems, including Tool Calling (Pattern 21) and Multiagent Collaboration (Pattern 23). Chapter 8 covers optimizing deployment (including Small Language Model [Pattern 24] and Inference Distribution Testing [Pattern 27]), and Chapter 9 discusses implementing safety guardrails, including Self-Check (Pattern 31) and comprehensive Guardrails (Pattern 32).

Who Is This Book For?

This book is for software engineers, data scientists, and enterprise architects who are building applications powered by GenAI foundational models. It captures proven solutions you can employ to solve the common challenges that arise when building GenAI applications and agents. Read it to learn how experts in the field are handling challenges such as hallucinations, nondeterministic answers, knowledge cutoffs, and the need to customize a model for your industry or enterprise. The age-old problems of software engineering have new solutions in this realm. For example, ways to meet latency and constrain costs include distillation, speculative decoding, prompt caching, and template generation.

Understanding the different patterns in this book requires different levels of background knowledge. For example, Chain of Thought (Pattern 13) requires no more than a knowledge of basic programming, Tool Calling (Pattern 21) requires an understanding of API design, and Dependency Injection (Pattern 19) requires some experience developing large-scale software. However, Content Optimization (Pattern 5) requires familiarity with statistics and ML, and Small Language Model (Pattern 24) requires an understanding of hardware optimization. We expect that 75% of the book can be read and understood by a junior software engineer or a third-year computer science student. The remainder will require specialized knowledge or experience.

AI engineering overlaps heavily with software engineering, data engineering, and ML—but in this book, we've limited our focus to core AI engineering. We encourage you to think of this book as a companion to the literature on patterns in related areas. Specifically, the book *Machine Learning Design Patterns* (O'Reilly), also co-authored by Valliappa Lakshmanan, covers proven solutions to recurring issues you'll encounter when training a bespoke machine-learning model for a specific problem.

You'll also likely find yourself working with both bespoke ML models and general-purpose foundational models, depending on the use case. In some situations, you might start with a foundational model but then find that edge cases require you to customize (or *fine-tune*) it for your problem. This book and *Machine Learning Design Patterns* are complementary and will help you work with both models, so we recommend that you read both.

Conventions Used in This Book

The following typographical conventions are used in this book:

Italic

Indicates new terms, URLs, email addresses, filenames, and file extensions.

`Constant width`

Used for program listings, as well as within paragraphs to refer to program elements such as variable or function names, databases, data types, environment variables, statements, and keywords.

`Constant width bold`

Shows commands or other text that should be typed literally by the user.

`Constant width italic`

Shows text that should be replaced with user-supplied values or by values determined by context.

> **TIP**
>
> This element signifies a tip or suggestion.

> **NOTE**
>
> This element signifies a general note.

> **WARNING**
>
> This element indicates a warning or caution.

In the diagrams, the boxes employ a set of color conventions as depicted in Figure P-1.

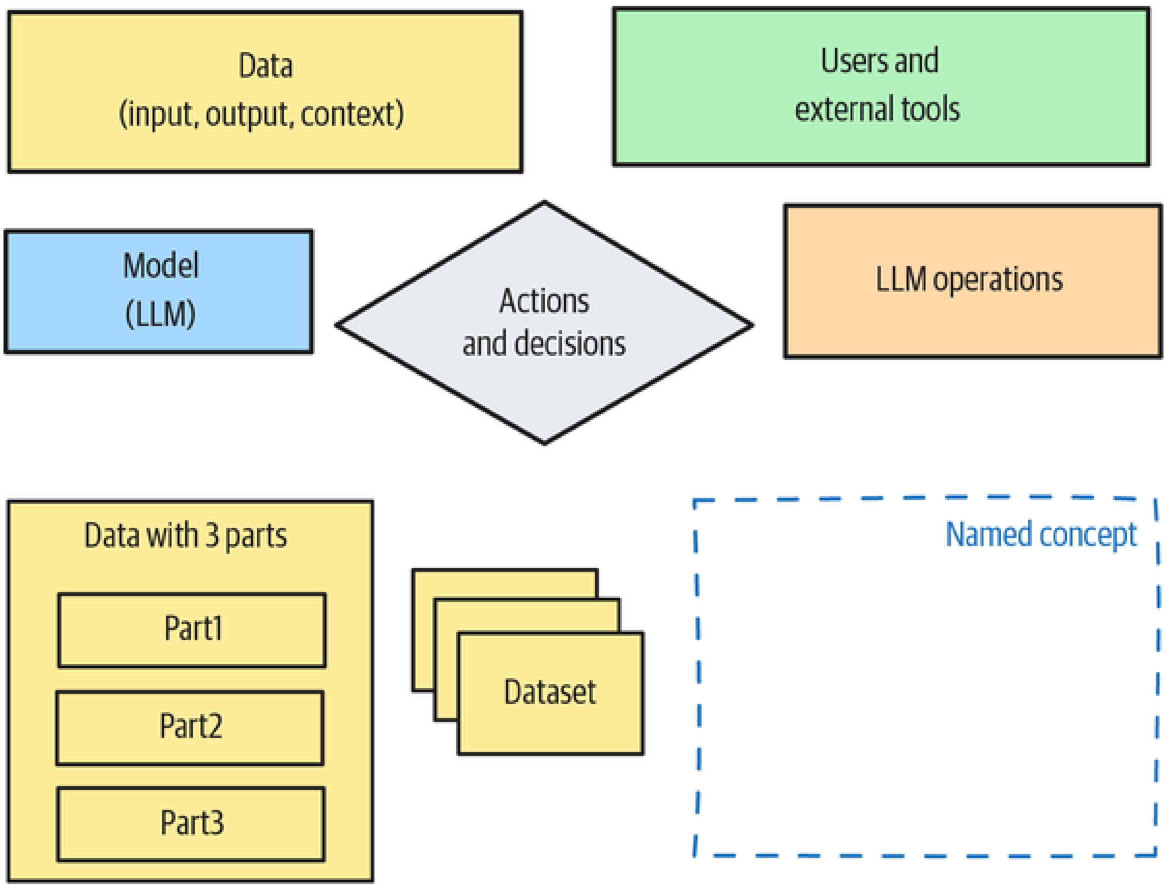


Figure P-1. Representation scheme used in diagrams in this book

Using Code Examples

Supplemental material (code examples, exercises, etc.) is available for download at *https://github.com/lakshmanok/generative-ai-design-patterns*.

If you have a technical question or a problem using the code examples, please send email to *support@oreilly.com*.

This book is here to help you get your job done. In general, if example code is offered with this book, you may use it in your programs and documentation. You do not need to contact us for permission unless you're reproducing a significant portion of the code. For example, writing a program that uses several chunks of code from this book does not require permission. Selling or distributing examples from O'Reilly books does require permission. Answering a question by citing this book and quoting example code does not require permission. Incorporating a significant amount of example code from this book into your product's documentation does require permission.

We appreciate, but generally do not require, attribution. An attribution usually includes the title, author, publisher, and ISBN. For example: "*Generative AI Design Patterns* by Valliappa Lakshmanan and Hannes Hapke (O'Reilly). Copyright 2026 Valliappa Lakshmanan and Hannes Hapke, 979-8-341-62266-1."

If you feel your use of code examples falls outside fair use or the permission given above, feel free to contact us at *permissions@oreilly.com*.

O'Reilly Online Learning

> **NOTE**
>
> For more than 40 years, *O'Reilly Media* has provided technology and business training, knowledge, and insight to help companies succeed.

Our unique network of experts and innovators share their knowledge and expertise through books, articles, and our online learning platform. O'Reilly's online learning platform gives you on-demand access to live training courses, in-depth learning paths, interactive coding environments, and a vast collection of text and video from O'Reilly and 200+ other publishers. For more information, visit *https://oreilly.com*.

How to Contact Us

Please address comments and questions concerning this book to the publisher:

O’Reilly Media, Inc.

141 Stony Circle, Suite 195

Santa Rosa, CA 95401

800-889-8969 (in the United States or Canada)

707-827-7019 (international or local)

707-829-0104 (fax)

support@oreilly.com

https://oreilly.com/about/contact.html

We have a web page for this book, where we list errata and any additional information. You can access this page at *https://oreil.ly/genAI-design-patterns*.

For news and information about our books and courses, visit *https://oreilly.com*.

Find us on LinkedIn: *https://linkedin.com/company/oreilly-media*.

Watch us on YouTube: *https://youtube.com/oreillymedia*.

Acknowledgments

Lak is thankful to his family for their forbearance as he (once again) vanished deep into writing and to collaborators and colleagues who gave him the opportunity to work far and wide with exciting new technology in practical ways. He’s also deeply appreciative of Hannes for the partnership while writing this book.

Hannes would like to thank Lak for his insightful mentorship and guidance throughout the writing process. Lak’s ability to explain complex topics in simple terms is truly exceptional, and Hannes is deeply grateful for being taken on this writing journey, from which he has learned immensely. This book would not have been possible without the unwavering support, endless patience, and love that Whitney, Hannes’s partner, brought to every day of this process. Hannes is profoundly grateful for Whitney’s amazing support, and he also extends his heartfelt appreciation to his family, especially his parents, who encouraged him to pursue his dreams around the world.

We are both thankful to the O’Reilly team (Nicole Butterfield, Corbin Collins, Catherine Dullea, Christopher Faucher, Sarah Grey, and Doug McNair [in alphabetical order]) for their unique blend of professionalism and flexibility. We were fortunate to have technical reviewers (David Cardozo, Mark Edmondson, Jason Fournier, Andrew Stein, and Glen Yu) who provided helpful, actionable, and speedy feedback on almost the entire book. In addition, Madhumita Baskaran, Ying-Jung Chen, Martin Gorner, Skander Hannachi, Ryan Hoium, and Danny Leybzon helped review specific chapters.

Chapter 1. Introduction

GenAI is so powerful and easy to use that even nontechnical users can easily prototype very compelling applications on top of GenAI. However, taking such GenAI prototypes to production is hard because GenAI models are unreliable—they can hallucinate, return different answers to the same input, and can have surprising limitations because of how they are trained. The design patterns in this book capture best practices and solutions to these and other recurring problems you're likely to encounter when building production applications on top of GenAI models.

GenAI Design Patterns

Design patterns, in software engineering, are proven solutions to common problems that occur during software design and development. They represent standardized best practices that have evolved over time through the collective experience of software developers. Design patterns are important because they establish a common vocabulary developers can use to communicate efficiently and because they help improve software quality, maintainability, and scalability.

The concept of design patterns was heavily influenced by the work of architect Christopher Alexander, who introduced patterns in architecture in his book *A Pattern Language* (Oxford University Press, 1977). Design patterns gained significant prominence in software engineering with the publication of the book *Design Patterns: Elements of Reusable Object-Oriented Software* by Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides (Addison-Wesley), which is often called "the Gang of Four book." Since then, design patterns have been cataloged for other software engineering domains, such as for Java Enterprise applications and ML.

When building AI products today, developers increasingly turn to *foundational* GenAI models (such as GPT-4, Gemini, Claude, Llama, DeepSeek, Qwen, and Mistral) that are trained on large, application-agnostic datasets, rather than building custom ML models that need to be trained from scratch on application-specific data. In this book, we'll follow Chip Huyen's *AI Engineering* (O'Reilly) in referring to this approach of building on top of foundational models as *AI engineering* and to practitioners of this approach as *AI engineers*.

AI engineering has a wide range of applications—including natural-language processing (NLP), text generation, code explanation, image understanding, and video synthesis—to power use cases such as content generation, AI assistants, workflow automation, and robotics.

As an AI engineer, you can ask a foundational model to directly generate the content your application needs by sending the model an appropriate text input, which is known as a *prompt*. However, you will face certain common problems—the generated content may not match the style you want, may be missing enterprise knowledge that the model doesn't know about, or may lack certain capabilities. In this book, we catalog a variety of proven solutions to such problems that arise in the context of building applications on top of GenAI foundational models.

In this book, you will also find detailed explanations of 32 patterns that codify research advances and the experience of experts into advice that you can readily incorporate into your projects. Each chapter offers a set of patterns as potential solutions to a particular problem that commonly arises in AI engineering. For example, Chapter 3 is about solving the problem that foundational models can't generate content that is informed by confidential enterprise data, because they are trained by model providers who don't have access to that data. The patterns presented in that chapter all address this problem. Each section that presents a pattern includes a description of the problem, a proven solution, an end-to-end working example of the pattern, and a discussion of alternatives and other considerations for implementing it.

AI engineers often encounter tasks that are too complex for a foundational model to perform all at once, so a common tactic is to break the complex task into smaller components that can be accomplished by foundational models. Such small software components that provide capabilities with the help of foundational models are called *agents*. Agents become increasingly autonomous as they use GenAI models to plan out a sequence of operations, identify the backend tools that they can invoke for each operation, determine how to recover from errors, and/or evaluate whether the task is complete. Applications that are built by orchestrating agents are called *agentic*. By showing you how to handle the inevitable challenges that arise when building applications on foundational models, the patterns in this book will help you build better agents and agentic applications.

Building on Foundational Models

In this section, we'll quickly cover the basics of AI engineering so that we don't have to repeat this introductory material in the sections on the patterns that follow in later chapters. For deeper coverage of building GenAI applications, we refer you to books such as Omar Sanseviero et al.'s *Hands-On Generative AI with Transformers and Diffusion Models* (O'Reilly), which covers the underlying technology; Chris Fregly et al.'s *Generative AI on AWS* (O'Reilly), which covers hyperscaler offerings; and Leonid Kuligin et al.'s *Generative AI on Google Cloud with LangChain* (Packt), which covers an open source GenAI framework.

A NOTE ON MODELS AND FRAMEWORKS

In *Machine Learning Design Patterns*, we used just two frameworks (scikit-learn and TensorFlow) and a single hyperscaler (Google Cloud Platform [GCP]) for consistency, but many readers felt that the resulting examples were too TensorFlow- and GCP-heavy. Therefore, in this book, we endeavor to be agnostic to model, framework, and hyperscaler.

Our code examples employ a wide range of technologies from a number of different vendors: large language models (LLMs) from OpenAI, Anthropic, Google, Alibaba, and Meta; and GenAI frameworks like LangChain, Pydantic AI, Hugging Face, and DSPy. Our examples are also agnostic to hyperscalers such as Amazon Web Services (AWS), Azure, GCP, and Oracle Cloud Infrastructure. Since you're likely to be using a different model in a different framework to address a different scenario, the code examples are meant only to serve as starting points for your implementation—we fully intend that you will have to adapt the code examples to your preferred LLM, framework, and hyperscaler.

Prompt and Context

When you build AI applications, you typically invoke hosted foundational models through an API. This might be the API provided by the vendor of the foundational model, or it might be a framework that allows you to easily switch between providers.

You invoke a foundational model by sending it a *prompt* and getting back a *response*. You are, doubtless, familiar with doing this by using the web user interface of a foundational model. For example, on ChatGPT, you might type a prompt like this one into the text box:

> Create a pencil sketch in the style of Degas depicting a family of four playing a board game

The simplest prompt typically consists of an *instruction* to the model that asks it to perform some content-generation task. In this case, the model follows the instruction and sends back a response that contains an image of the type requested (see Figure 1-1).[1] Both prompts and responses can be *multimodal*—they could be text, but they could also be images, video, or audio.

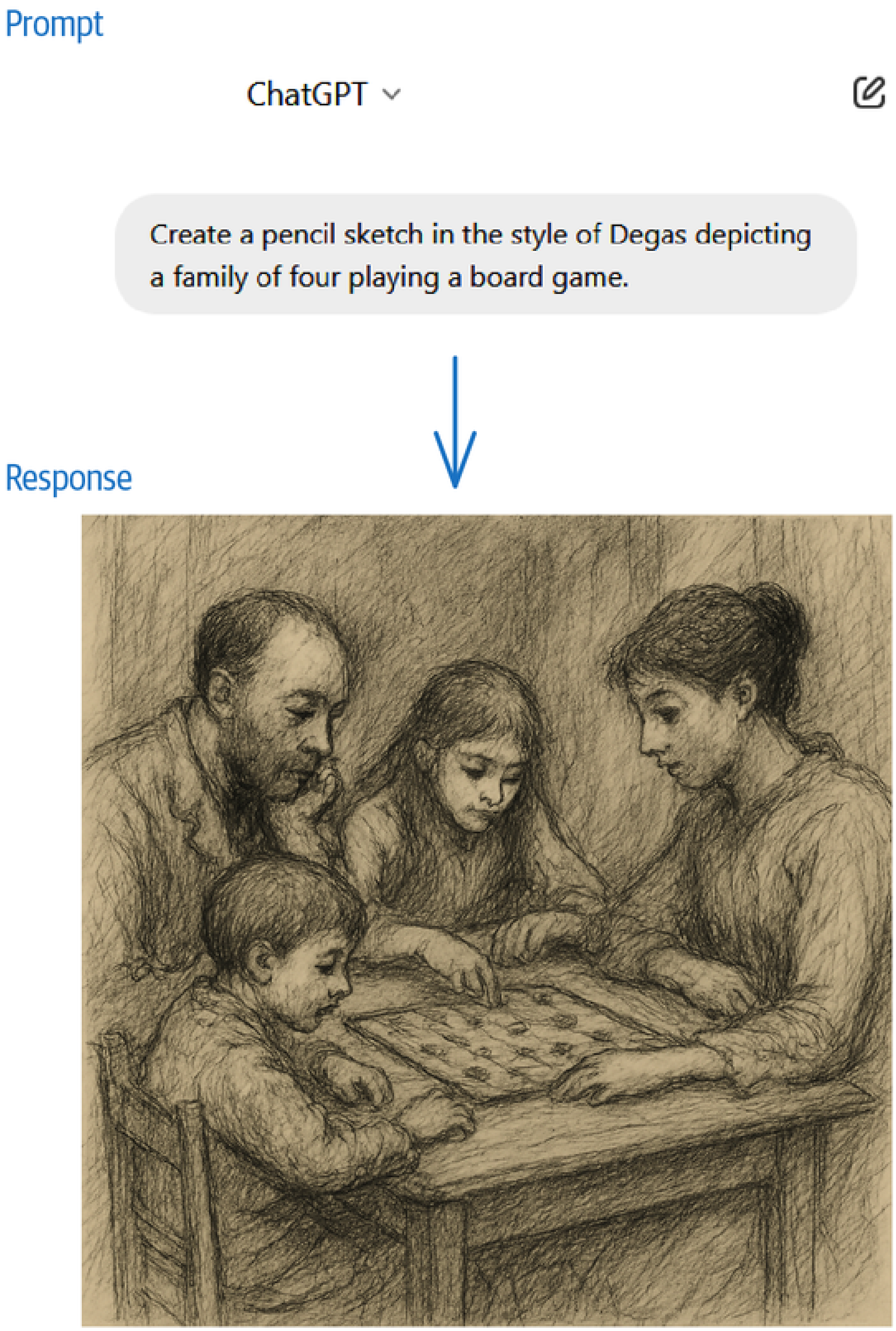


Figure 1-1. Sending a prompt to a foundational model to get back a response

A more complex prompt might consist not just of an instruction but also *context*. This might be information that the model should use in the generation, or it might involve asking the model to take on a specific role. For example, we could tell Claude (on Claude.ai) to take on a personality and use some information when generating content:

> You are an expert marketer who is very familiar with the book market in university towns in Germany.
>
> *Covenant of Water* is a novel that tells the story of three generations of an Orthodox Saint Thomas Christian family in Kerala.
>
> Write a one-paragraph blurb introducing the book to readers at a bookstore in Göttingen, drawing local connections.

The response from Claude Sonnet takes into account the role, context, and instruction:

> **Covenant of Water - A Literary Journey to Kerala**
>
> Nestled between the historic walls of Göttingen, where academic tradition meets literary curiosity, we present Abraham Verghese's masterful epic, "The Covenant of Water." This sweeping saga follows three generations of an Orthodox Saint Thomas Christian family in Kerala, India—a region whose complex religious heritage might resonate with students of our university's renowned theological studies program. As the story unfolds across decades amid Kerala's lush backwaters, readers familiar with Göttingen's own relationship with water—from the sacred Gänseliesel fountain to the meandering Leine River—will find surprising parallels in how geography shapes destiny. Verghese weaves a tale of family secrets, medical mysteries, and generational bonds that transcends cultural boundaries, offering our community of international scholars and lifelong learners a profound exploration of how water—whether in Kerala's monsoons or Lower Saxony's gentle rains—becomes both metaphor and covenant, connecting us across time and space.

Using the Model Provider's API

APIs that are designed to invoke foundational models are available in a wide variety of languages—Python, Go, Java, and TypeScript are all commonly employed. For simplicity, we focus on Python APIs in this book.

To invoke Claude Sonnet 3 through the Anthropic API in Python, you could use the Anthropic Python package as follows (the full code is on GitHub):

```python
import anthropic
client = anthropic.Anthropic(
    # defaults to os.environ.get("ANTHROPIC_API_KEY")
    api_key="YOUR_ANTHROPIC_API_KEY",
)

completion = client.messages.create(
    model="claude-3-7-sonnet-latest",
    system="You are an expert Python programmer.",
    messages=[
        {
            "role": "user",
            "content": [
               {
                  "type": "text",
                  "text": """
Write code to find the median value of a list of integers.
"""
               }
            ]
        }
    ]
)

print(completion.content[0].text)
```

In this API call, the prompt has been broken into two parts—a *system prompt* and a *user prompt*. The system prompt is set by the developer and guides the model's overall behavior, while the user prompt is more dynamic and provides specific instructions for a specific task you want the model to perform. Here, the AI assistant's role has been set in the system prompt while the user prompt and context are sent as messages.

Using an LLM-Agnostic Framework

To perform the same task using the PydanticAI framework, you'd use code such as the following (assuming the needed API key is set in an environment variable):

```
from pydantic_ai import Agent
agent = Agent('anthropic:claude-3-7-sonnet-latest',
              system_prompt="You are an expert Python
programmer.")

result = agent.run_sync(
           "Write code to find the median value of a list of
integers.")
print(result.data)
```

The advantage here is that you can easily switch between foundational model providers by switching the model string to `openai:gpt-4o-mini`, `google-vertex:gemini-2.0-flash`, `groq:llama3-70b-8192`, and so on (see Pydantic's documentation for the full list of models supported).

The class in the Pydantic API that invokes the Claude model is called `Agent`. We'll discuss what agents are in the next section, but before that, let's conclude our discussion of ways to invoke foundational models.

Running Your Model Locally

To run a model such as Llama 3 on your local hardware, you could use the Ollama client to download and run the model that you want to use:

```
ollama run llama3.2
```

Ollama exposes open-weights models with the OpenAI API, so you could use this:

```
from pydantic_ai.models.openai import OpenAIModel
from pydantic_ai.providers.openai import OpenAIProvider
```

```
model = OpenAIModel(
    model_name='llama3.2',
    provider=OpenAIProvider(base_url='http://localhost:11434/v1')
)
```

How Foundational Models Are Created

Unlike traditional machine learning applications, your AI applications will rarely include a training phase. Instead, you'll build them on top of general-purpose foundational models that have been pretrained to perform a wide variety of tasks. You can, for the most part, ignore the internal details of the foundational model—and we're including this section in this book only so you can understand the associated vocabulary.

At the time of writing in spring 2025, DeepSeek is the foundational model with the most available information on its training regimen. We'll use that information to discuss the key steps (see Figure 1-2) involved in creating a foundational model. While OpenAI, Google, Anthropic, and Meta may not have followed this exact process in creating GPT, Gemini, Claude, and Llama, their methods are probably broadly similar.

The DeepSeek base LLM was pretrained (in Step 1 of Figure 1-2) on a diverse, high-quality corpus of 14.8 trillion tokens. (The works of Shakespeare, as a point of comparison, amount to about 1.2 million tokens—so the DeepSeek training dataset is equivalent to 12 million Shakespeares!) Unlike early LLMs, which were trained on words, modern LLMs are trained on *tokens*, which are short sequences of characters. This allows such LLMs to learn things that are not in the vocabulary of the language, like proper names. It isn't just size that helps—the DeepSeek team attributes the high quality of their models to careful data curation, including rigorous deduplication processes.

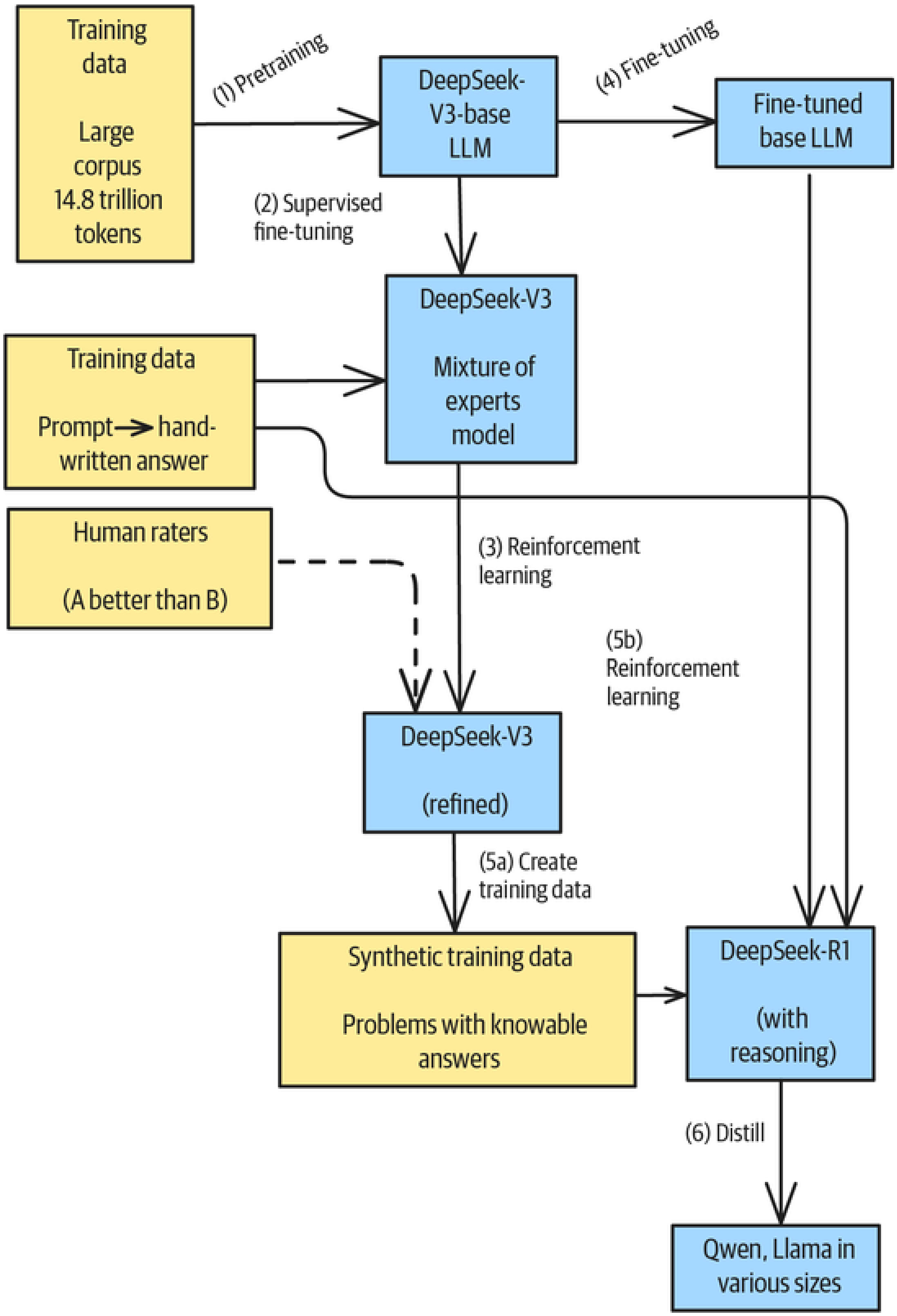


Figure 1-2. Stages involved in training the DeepSeek-R1 model

The *pretraining* stage (Step 1 in Figure 1-2) involves training on this massive dataset of tokens to develop the model's general language-understanding capabilities. The key goal at this stage is to train the model to predict the next token, given a context consisting of the previous tokens in the training input. This is why you'll often hear people call LLMs *next-token predictors*—but next-token prediction is only the first stage of the training regimen.

Following pretraining, the model undergoes *supervised fine-tuning* (SFT, Step 2 in Figure 1-2) to improve its ability to follow instructions and generate high-quality responses. This stage uses carefully curated datasets of human-written examples. Cohere, for example, has said it uses licensed physicians, financial analysts, and accountants to improve its models. Presumably, these practitioners write ideal answers given a prompt. The result, DeepSeek-V3, is a *mixture of experts* (MoE) model: an optimization that allows models to have a large number of parameters while activating only a fraction of them for each token. DeepSeek-V3 has 671 billion parameters in total, but only 37B are activated per token. This allows the model to use different pathways for different types of instructions.

The *reinforcement learning* stage (Step 3 in Figure 1-2) is where the models are further refined based on human preferences. This step involves *reinforcement learning with human feedback* (RLHF), which means showing human raters a pair of generated outputs and asking them which one they prefer. Such *preference tuning* helps align the models' outputs with human expectations and values. This stage is also sometimes called *preference optimization*.

Once DeepSeek-V3 was created with a small number of human-written examples and human preferences, the full DeepSeek-R1 model was developed through the following multistage process:

1. The *cold start* involved fine-tuning a base model (DeepSeek-V3-Base) with thousands of cold-start data points to lay a foundation.

2. *Pure reinforcement learning* (RL) involved applying a pure reinforcement learning approach to enhance reasoning skills.

3. *Rejection sampling* involved choosing the best examples from a distribution of data created during the last successful RL run.

4. SFT involved merging the synthetic data with supervised data from DeepSeek-V3-Base in domains like writing, factual question answering (QA), and self-cognition.

5. *Final RL* was a final reinforcement learning process across diverse prompts and scenarios.

Step 2 (pure RL) was a major breakthrough because the team directly applied reinforcement learning to the base model (DeepSeek-V3-Base) *without* relying on supervised fine-tuning as a preliminary step. This approach allowed the model to explore *chain-of-thought* (CoT) reasoning for solving complex problems, which resulted in the development of DeepSeek-R1-Zero. It turned out that reasoning capabilities in LLMs can be incentivized purely through RL, without the need for SFT, as researchers had long believed. Since human-written examples are expensive, the ability to use pure RL allows for longer training on a much more diverse set of problems than if SFT is required.

To make DeepSeek-R1's capabilities more accessible, the team created distilled versions of the model that can run on more modest hardware while retaining much of the original's reasoning capability. These include models based on Qwen and Llama architectures in various sizes (1.5B, 7B, 8B, 14B, etc.).

The Landscape of Foundational Models

The GenAI foundational model ecosystem has evolved significantly, with distinct categories emerging to serve different needs.

Because academic benchmarks are saturated and can be gamed, the currently most accepted way to rate GenAI models is to compare them pair-wise in blind tests. LMArena carries out a large-scale comparison, and Figure 1-3 shows the April 2025 leaderboard.

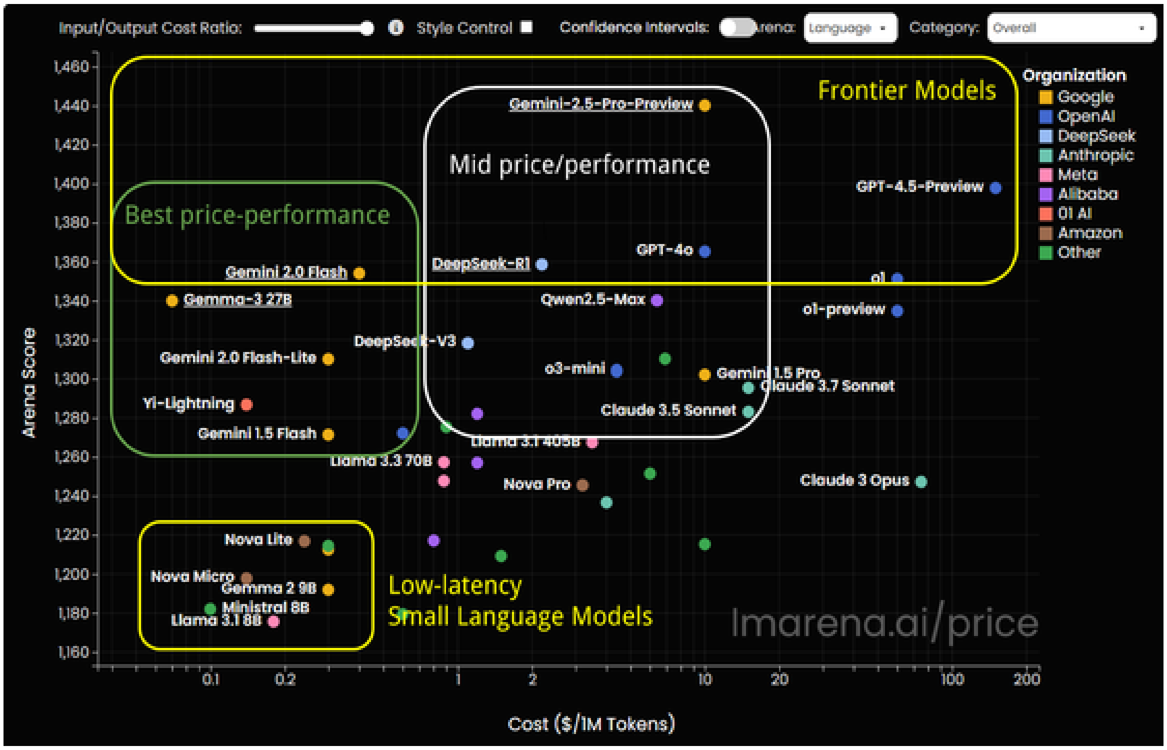


Figure 1-3. The LMArena leaderboard on April 6, 2025 (annotated boxes added by the authors)

The leaderboard shows model rating on the y-axis and cost on the x-axis. You can use this to determine the best model that fits your budget and the lowest price you should expect to pay for a given rating. Note that the Elo rating on the y-axis is a logarithmic relationship,[2] and so is the cost on the x-axis.

The highest-rated models overall (see the y-axis) on the day we screenshotted the leaderboard were Gemini 2.5 Pro Preview, GPT-4.5 (Preview), GPT-4o, and DeepSeek-R1, with Gemini 2.0 Flash and OpenAI o1 also nearly as high. The ranking changes daily as new models are released and more ratings are collected, but the leaderboard has been remarkably consistent over time, with the flagship models from Google, OpenAI, and Anthropic almost always on top. Together, these are referred to as *frontier models*.

Frontier models such as GPT 5 and Gemini 2.5 Pro represent the state of the art in language model capabilities, and they offer the highest performance across reasoning, knowledge, and multimodal tasks. However, they are resource intensive and costly and can't be run locally due to their size and proprietary nature. You'd use them in enterprise-grade applications requiring sophisticated reasoning and where speed is not a concern. Recent developments in frontier models include multimodal capabilities, enhanced reasoning, and extended-content windows (with up to two million tokens in some models).

Distilled versions of frontier models balance performance with efficiency, offering reasonable capabilities at lower costs and with faster response times. Leading examples include Gemini Flash, Claude Sonnet, and GPT-4o-mini. These models tend to offer good performance on common tasks like content generation and summarization. They also offer fast response times, and they're cost-effective for high-volume applications. The cost difference between running Gemini Pro and Gemini Flash becomes quite apparent once you note that the x-axis in Figure 1-3 is logarithmic—Flash is 20 times less expensive.

Open-weight models have their parameters publicly available, allowing for transparency, community improvement, and customization. Examples include Llama, Mistral, DeepSeek, Qwen, and Falcon. These models offer strong performance but generally lag behind frontier models, and they can be fine-tuned on proprietary data but require more expertise to host. However, there are hosted API services such as Together.ai, as well as fully managed API endpoints on the hyperscalers, that address this issue.

Locally hostable models are designed to run on consumer or enterprise hardware without requiring cloud connectivity. Examples include Llama 8B and Gemma 2B, hardware-optimized versions of which are available through NVIDIA NIM. This allows for complete privacy, with no data leaving your local devices and no ongoing API costs. However, these models have significantly reduced capabilities compared to cloud models. The demand for frontier models in "air-gapped" systems that are disconnected from the internet has led some proprietary model vendors to offer this service as well—for example, Gemini can be run on-premises in Google Distributed Cloud and OpenAI can be run on Azure's on-premises offerings.

Agentic AI

The class in the Pydantic API that invoked the Claude model was called `Agent`:

```
from pydantic_ai import Agent
agent = Agent('anthropic:claude-3-7-sonnet-latest',
      ...
```

What is an agent? In computer science, the term *agent* has long been used to describe software entities that act on behalf of users or other programs. When you invoke a foundational model, you specify a role, provide some context, and ask it to carry out some instruction. In the computer science sense, then, the LLM is acting as your agent.

For example, here's an example of creating an agent to manage inventory levels in a store (the full code is on GitHub):

```
agent = Agent(
  f"anthropic:{MODEL_ID}",
  system_prompt="You are an inventory manager who orders just in
time.",
  ...
)
```

Autonomy

In AI, an agent is also expected to be somewhat autonomous. Here, the LLM functions as the agent's brain, so you don't need to tell it how exactly to manage inventory levels beyond the *goal* (which is to order "just in time").

Suppose you have a list of items in your inventory, plus data on how well they've been selling and how long they'll take to deliver:

```
@dataclass
class InventoryItem:
    name: str
    quantity_on_hand: int
    weekly_quantity_sold_past_n_weeks: [int]
```

```
    weeks_to_deliver: int

items = [
    InventoryItem("itemA", 300, [50, 70, 80, 100], 2),
    InventoryItem("itemB", 100, [70, 80, 90, 70], 2),
    InventoryItem("itemC", 200, [80, 70, 90, 80], 1)
]
```

Provide the list of items to the agent and it will figure out which ones to reorder:

```
result = agent.run_sync(f"""
Identify which of these items need to be reordered this week.

**Items**
{items}
""")
```

The result will include this, in part:

> itemB
>
> **quantity_to_order=300** reason_to_reorder='Current stock (100) is insufficient to cover projected demand over delivery time. Based on recent weekly sales (70-90 units), we need to **order enough to cover the 2-week delivery period** plus maintain safety stock.'

Compare this to traditional programming, where you'd have to write code to explicitly manage inventory. Such *autonomy*—which means the ability to operate independently without constant human guidance or being explicitly programmed to do so—is the key differentiator between traditional software and AI agents.

Characteristics of Agents

Besides autonomy, agents are usually expected to have the following characteristics:

Goal orientation

Agents work toward specific objectives, rather than simply responding to input prompts. The goal of the inventory manager agent, which is to manage inventory just in time, was set in its system prompt.

Planning and reasoning

Notice that the inventory manager agent was able to plan the steps to determine how many items to reorder. It identified the range of recent weekly sales, projected the maximum sale forward by the delivery window, determined how many items would be required, and then calculated the number of items to reorder. None of this needed to be explicitly programmed.

Perception and action

Agents can gather the data they need ("perceive") and act on their environment. You can usually give them this ability by enabling them to call external functions (such as searching the web, invoking calculators, and writing to databases) through Tool Calling (Pattern 21), which we'll discuss in Chapter 7. With Tool Calling, you can go beyond explicitly providing the list of items to the inventory manager. Instead, the agent can retrieve the quantity on hand and weekly sales from backend databases. Also, instead of just telling you that item B needs to be reordered, it can invoke an API, perhaps even an API on the vendor's site, to place the order.

Adaptability and learning

How do you know that ordering 300 items is correct? A human inventory manager would plug in assumptions of weekly sales for the next two weeks and validate that the store will not run out—and an agent can do the same thing. In Chapters 6 and 9 (respectively), you'll see patterns such as Reflection (Pattern 18) and Self-Check (Pattern 31) that allow an agent to evaluate its output and self-correct.

At the time of writing, agentic behavior remains an aspirational goal for applications built on foundational models—nondeterminism, hallucinations, and various other failure modes pose challenges to building fully autonomous AI applications. Take nondeterminism, for example—each time, you might get a different list and quantity of items to reorder. (Try it!) Planning works in simple cases, but not in hard ones. In Chapter 5, you'll see patterns such as Chain of Thought (Pattern 13) that improve the ability of an agent to do planning and reasoning. Many of the design patterns in this book are ways to make your AI applications more agentic, or at least to push the boundaries of what you can build.

Fine-Grained Control

Foundational models give you fine-grained control over the generation process by allowing you to control sampling and beam search (both of which are discussed later in this chapter). The generation settings of LLMs provide powerful tools for controlling the balance between deterministic, high-quality outputs and creative, diverse responses. Understanding settings' mathematical underpinnings, from logits to sampling strategies, can help you control model behavior in simpler ways than with the design patterns covered in later chapters of this book.

Logits

Language models have hundreds of layers, but the very last layer predicts the next word to continue the text generation. They don't predict just one token, but instead, they provide a set of candidate tokens and the probability that each of those tokens will be the next one.

Logits are the raw, unnormalized outputs from a language model's final layer before they're converted into probabilities. Logits represent the model's assessment of how likely each token in its vocabulary is to be the next token in a sequence. Suppose, for illustration purposes, that there are five possible continuations to a sequence being generated and that the logits of each of the possible continuations are as shown on the left-hand side of Figure 1-4.

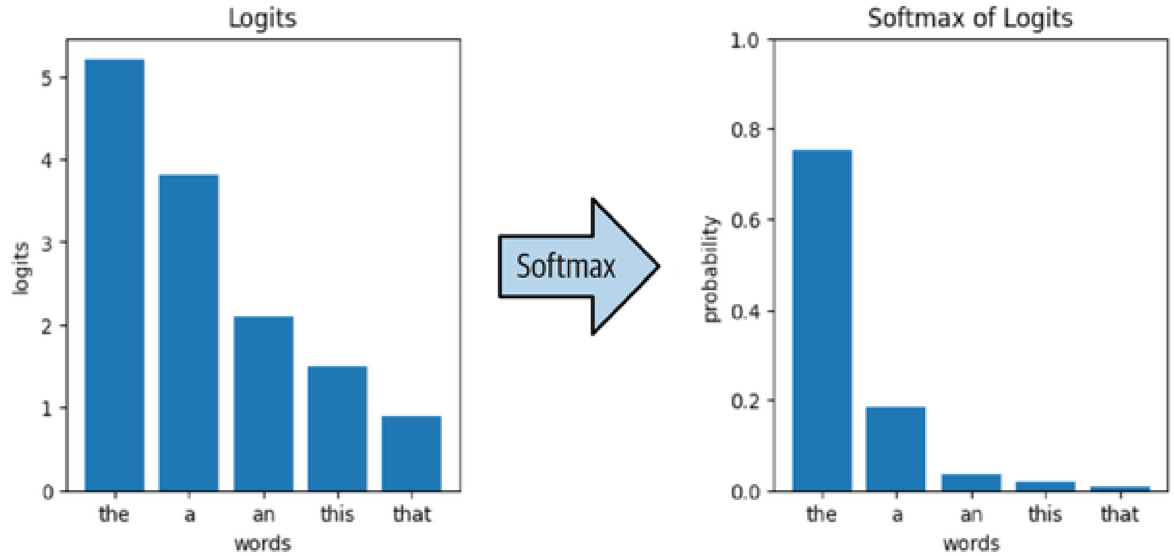


Figure 1-4. The softmax of the logits gives you the probability of various continuations

If the model were to use *greedy sampling*, where only the most likely word is chosen, it would simply select *the* since that word has the highest logits. However, such a strategy leads to highly repetitive and uninteresting text, so models use a sampling strategy in which all the possible continuation words have some nonzero probability of being selected. The transformation from logits to probabilities occurs through the `softmax` function:

$$P\left(token_i\right) = e^{logit_i} / \Sigma_j e^{logit_j}$$

Here, `P(token_i)` is the probability of selecting `token_i`. The `softmax` function accentuates the peaks and dampens the tails—for example, compare the length of the bars for *the* and *a* before and after the `softmax` function is applied in Figure 1-4.

If the distribution of potential continuations is less peaked, as shown in Figure 1-5, then the impact of the `softmax` is less pronounced.

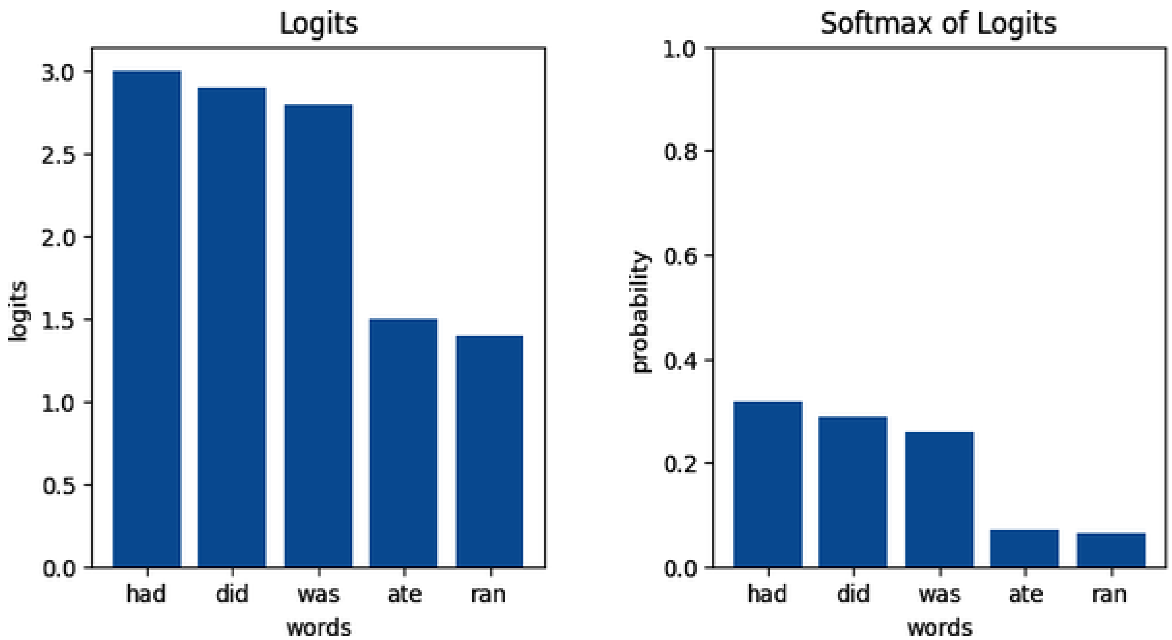


Figure 1-5. The impact of the `softmax` of the logits is less pronounced if the distribution is not peaked

We'll explore the use of logits to control style in Chapter 2.

Temperature

Temperature (*T*) is a hyperparameter that controls the randomness of token selection by scaling the logits before they're passed through the `softmax` function. The modified `softmax` equation with temperature is as follows:

$$P\left(token_i\right) = e^{logit_i/T} / \Sigma_j e^{logit_j/T}$$

The effect of scaling the same graphs with different values of *T* is shown in Figure 1-6.

As you can see, setting the temperature to zero turns on greedy sampling. As the temperature increases, the likelihood that less likely "tail" words will be chosen also increases. The effect of temperature is less pronounced when the distribution was less peaked to begin with.

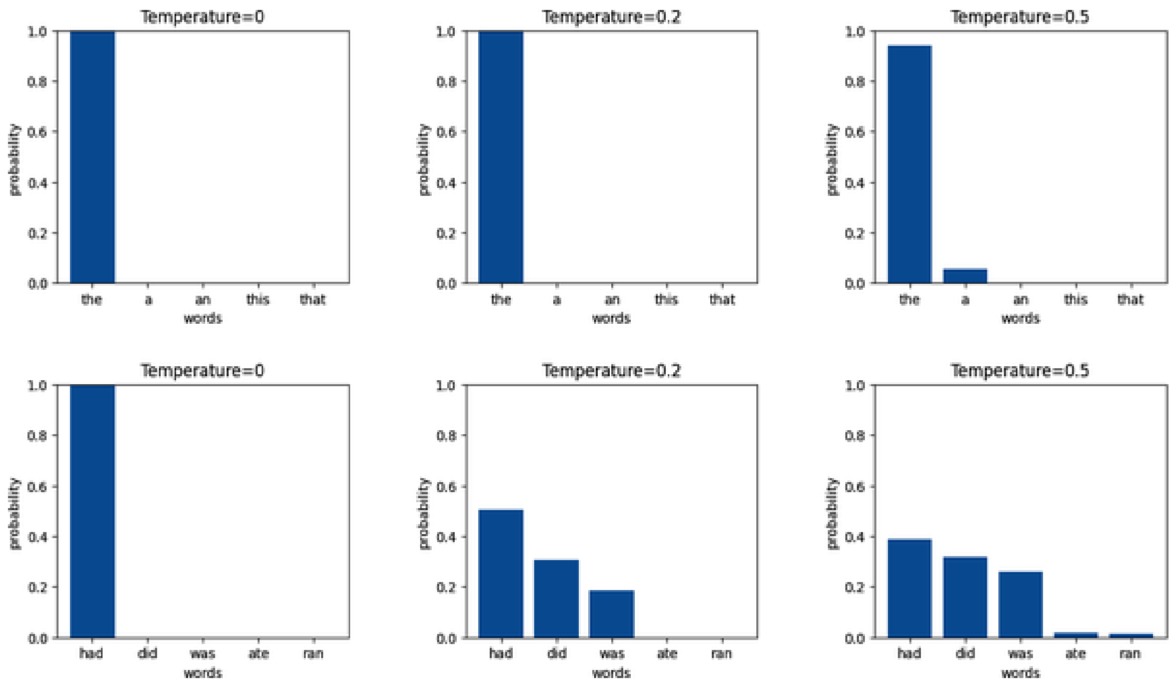


Figure 1-6. Effects of scaling the logits by temperature

Here's how to vary the temperature when using PydanticAI:

```
agent = Agent('anthropic:claude-3-7-sonnet-latest',
              model_settings={
                 "temperature": 0.5
              },
              system_prompt="Complete the sentence.")
```

Here's how to do it when using the Anthropic API directly:

```
completion = client.messages.create(
    model="claude-3-7-sonnet-latest",
    system="Complete the sentence.",
    temperature=0.5,
    messages=[
        ...
    ]
)
```

Here's an example, showing various continuations of the phrase *The trade war caused* produced at three different temperature settings:

0.0 : The trade war caused significant disruptions to global supply chains, leading to increased prices for consumers and economic uncertainty for businesses across multiple industries.

0.5 : The trade war caused significant disruptions to global supply chains, leading to increased prices for consumers and economic uncertainty for businesses across multiple industries. Many manufacturers were forced to reconsider their production strategies, while farmers faced reduced export opportunities as retaliatory tariffs limited access to international markets. The long-term effects included accelerated efforts to diversify supply chains away from affected regions and renewed debates about the effectiveness of protectionist trade policies.

0.8 : The trade war caused significant disruptions to global supply chains, forcing many companies to reconsider their manufacturing strategies and sourcing policies. It led to increased tariffs on imported goods, higher prices for consumers, and economic uncertainty in affected industries. Several businesses reported decreased profits as they absorbed additional costs or lost market share in foreign markets. The prolonged tension also contributed to volatility in financial markets and complicated diplomatic relations between the involved nations.

As you can see, increasing the temperature tends to lead to more creative output.

In Chapter 3 and Chapter 6, you will see situations, like RAG and LLM-as-Judge, where it may be necessary to use low or even zero temperatures.

Top-K Sampling

Top-K sampling restricts token selection to only the *k* most likely tokens from the vocabulary, effectively truncating the long tail of the probability distribution. This can help you avoid off-the-wall continuations at high temperatures.

Here’s the impact of setting different top-K values to continue the phrase *The spaceship*:

1 : The spaceship zoomed through the vast expanse of space, its powerful engines glowing blue against the darkness as it carried its crew toward distant stars and unknown adventures.

10 : The spaceship glided silently through the vast emptiness of space, its powerful engines propelling it toward the distant galaxy where no human had ventured before.

100 : The spaceship soared through the starry expanse, its gleaming hull reflecting the distant light of alien suns as it carried its crew toward unexplored worlds beyond the edge of known space.

As you can see, when the top-K value is low, the generated text closely follows phrases that you can find in existing science fiction.

Nucleus Sampling

Nucleus sampling dynamically selects the smallest set of tokens whose cumulative probability exceeds a threshold *p*. Hence, it's also called *top-P sampling*. This creates a "nucleus" of tokens that represent the bulk of the probability mass. Figure 1-7 shows the impact of applying varying top-P values to our illustrative distributions.

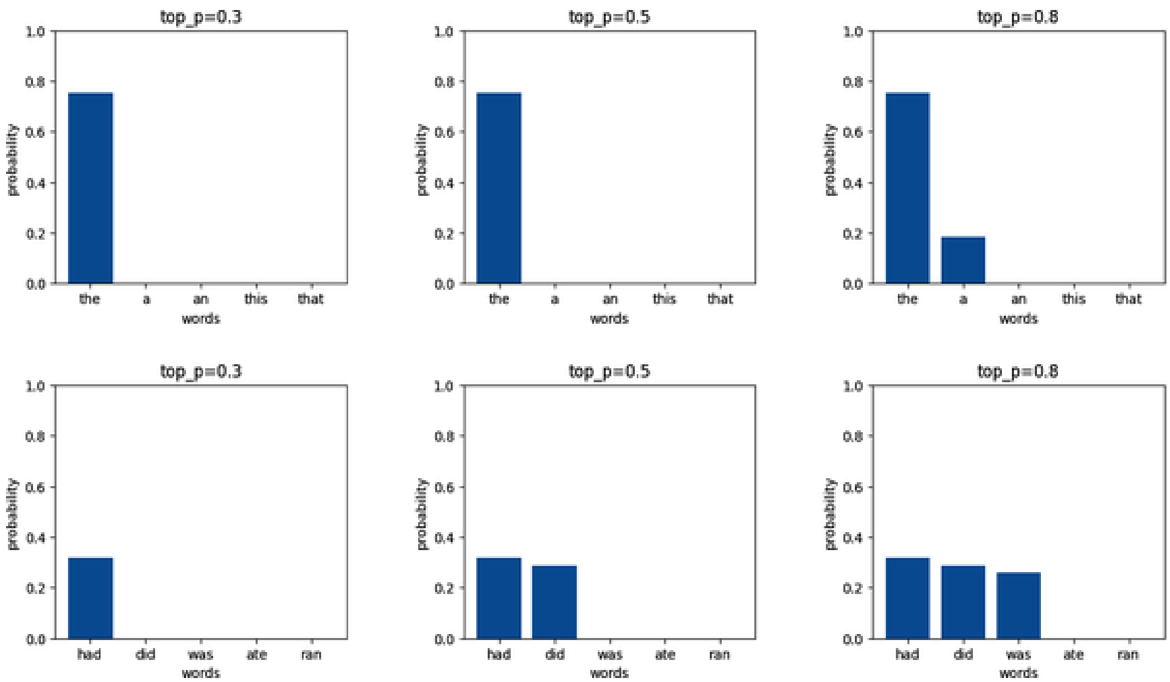


Figure 1-7. Effects of top-P sampling

Nucleus sampling adapts to the model's confidence at each step. When the model is very confident, few tokens are considered (as with low top-K), and when the model is uncertain, more tokens are considered (as with high top-K). Therefore, it generally produces more natural text than fixed top-K sampling does.

Beam Search

Models don't choose continuation tokens one at a time. Instead, they employ *beam search*, which is a deterministic search algorithm that aims to find the most likely sequence of tokens by exploring multiple possible continuations in parallel. In this way, the model can consider the probability of the entire sequence, not just the next token.

Some models give you knobs to control the beam search.

Repetition penalties reduce the probability of tokens that have already appeared in the generated text will appear again, thus helping to prevent repetitive loops. These penalties include the following:

Frequency penalty

This increases the more a word or phrase appears in the generated text.

Presence penalty

This applies whenever a word or phrase is repeated, and it thus encourages the model to employ a diverse range of tokens in the generated text.

Length penalties adjust the scoring of sequences based on their length. These penalties include the following:

Minimum length penalty

This forces the model to generate at least a certain number of tokens.

Maximum length penalty

This caps generation at a specified length.

Length normalization penalty

This divides sequence scores by a function of their length to prevent bias toward shorter sequences.

Beam search width is the number of parallel sequences considered during generation. You will see beam search being used in patterns in Chapter 2 and Chapter 6.

At the time of writing, OpenAI and Gemini support repetition penalties but Anthropic does not. The Transformers library in Hugging Face supports length penalties and beam search width, but none of the hosted models do.

In-Context Learning

In "traditional" ML, where models are predominantly trained for single problems like classifications or regressions, models require extensive training to adapt their weights or parameters to a new task. But you can use

LLMs for a wide variety of tasks by changing just the prompt without changing the weights.

Foundational models' capability to adapt to new tasks based solely on examples or instructions they're given in a prompt is called *in-context learning*. It emerges from training on diverse datasets that include various task formats and instructions. When given examples of a task within a prompt, the model identifies the underlying pattern and applies it to new instances. This process doesn't modify the model's weights, but rather, it leverages the model's existing knowledge to perform new tasks.

Zero-Shot Learning

Zero-shot learning involves instructing a model to perform a task without providing any examples. This technique relies on the model's pretrained knowledge and its ability to understand natural language instructions.

For example, you could ask Gemini-Flash to do the following:

> Analyze the use of light in Claude Monet's *Impression, Sunrise* and explain how it exemplifies impressionist techniques.

It would respond with something like this (though your results may vary, since model outputs are nondeterministic):

> In *Impression, Sunrise,* Monet employs light to capture a fleeting moment, prioritizing its transient effects over precise details. The hazy atmosphere and vibrant reflections on the water showcase the impressionist focus on perception and the subjective experience of light, rather than a realistic depiction of the scene. This emphasis on capturing the immediate "impression" of light became a hallmark of the Impressionist movement.

Few-Shot Learning

Few-shot learning extends this approach by including a small number of examples in the prompt. The examples serve as demonstrations that help the model understand the task's structure and expected output format, and the model then applies that structure to new instances. Because the examples are placed in the context of the prompt, this is also called *in-context learning*.

Few-shot learning is a simple and effective form of *context engineering*, which is exactly what it sounds like—adding finely crafted data and instructions to the context of the prompt to achieve the desired performance. You'll encounter many patterns in this book, especially in Chapter 5, that build on few-shot learning.

For example, suppose you want to build an application that allows the user to describe a painting they see in words and have the application identify it. You can teach it what goes into an identification by providing an example question and answer for *Impression, Sunrise* by Claude Monet:

```
agent = Agent(MODEL_ID,
              system_prompt="""You are an expert on art history.
I will describe
a painting. You should identify it.
""")

result = agent.run_sync("""
Example:
```
Description: shows two small rowboats in the foreground and a red
Sun.
Answer:
   Painting: Impression, Sunrise
   Artist: Claude Monet
   Year: 1872
   Significance: Gave the Impressionist movement its name;
captured the fleeting
effects of light and atmosphere, with loose brushstrokes.
```

Description: The painting shows a group of people eating at a
table under an
outside tent. The men are wearing boating hats.

""")
```

The result correctly identifies a Renoir painting that matches the description:

Answer:

Painting: Luncheon of the Boating Party

Artist: Pierre-Auguste Renoir

Year: 1881

Significance: Captures a joyful social gathering of Renoir's friends at the Maison Fournaise restaurant in Chatou, France, depicting the carefree atmosphere of Parisian life during the Belle Époque; known for its vibrant colors, lively composition, and portrayal of light and shadow.

Instead of requiring you to build a domain-specific model, collect lots of training data, and perform the training tasks, in-context learning allows LLMs to perform new tasks "on the fly." The examples "guide" the LLM to mimic the input-output patterns they demonstrate.

When zero-shot learning (in which no examples are provided) does not solve your problem, you can try adding examples to your prompt and giving the model practical instructions to complete the task. Using in-context learning with a few examples is easier than using traditional ML because you don't need to curate an extensive dataset to fine-tune a model. You can often achieve good results with few-shot learning.

With in-context learning, it's possible to quickly prototype solutions. It also boosts production use cases: it's much faster to update examples in a prompt than to curate, retrain, and redeploy an ML model.

However, in-context learning has a few limitations:

- It only works when the foundational model already has the necessary knowledge and capability.
- Adding many examples consumes valuable tokens of your model's context window and will slow your inference time.

- LLMs sometimes struggle to generalize more complex problems based on a few examples.

In these scenarios, post-training might offer a better approach.

Post-Training

Post-training methods involve modifying the model weights of pretrained models to customize them to new tasks or domains. A post-trained model will have to be deployed and utilized from a different end point than the foundational model that was its starting point.

Post-Training Methods

Recall from Figure 1-2 that training a foundational model involves multiple stages. The first stage involves training a base LLM on predicting the next word, the second stage involves training the base LLM to perform tasks using SFT, and the third stage involves RLHF. You can post-train the model in any or all of these ways.

Here is some terminology you might encounter that is associated with post-training. These are not wholly different methods—there's some overlap between them:

Continued pretraining (CPT)

> You can continue training the base model if you have a dataset that contains vocabulary (such as industry jargon) and associations that the foundational model wasn't trained on. This requires access to the full weights and architecture of the base model. It's important to note that training base models is extremely expensive and time-consuming, and moreover, you'll have to perform the remaining stages of training on this new model. In March 2023, Bloomberg used this approach on financial documents, and just months later, it found that foundational models handily outperformed the domain-specific model. Since then, few organizations have chosen this approach.

Supervised fine-tuning (SFT)

You can use SFT to further train language models on datasets consisting of (prompt, response) pairs in a supervised manner. If you use prompts that consist of specific instructions ("improve this management plan," etc.) and a wide variety of data, you can improve the model's ability to follow those specific natural language instructions. If you train the model on a wide variety of instructions, you can enhance its zero-shot capabilities across diverse tasks. You can also use such *instruction tuning* to enhance generalization or to elicit more helpful or honest behavior from the model. Whether the model will display this behavior will depend on the size and diversity of your dataset—if you do SFT on just a single task (instruction), then the model will probably forget the tasks it learned earlier and not generalize to new tasks. On the other hand, if your instruction training dataset consists of multiple tasks, then the model may generalize to new tasks even after just one round of SFT.

Parameter-efficient fine-tuning (PeFT)

Because foundational models are so large, it's unwieldy to train them further. So, *parameter-efficient fine-tuning* (PeFT) approaches have emerged to make the training process more practical for large models. *Low-rank adaptation* (LoRA) represents weight updates using smaller matrices through low-rank decomposition. Instead of fine-tuning all model parameters, LoRA freezes the original pretrained weights and adds small, trainable "adapter" matrices that are decomposed into low-rank representations. LoRA drastically reduces the number of trainable parameters (making them up to 10,000 times fewer) and reduces GPU memory requirements (making them up to 3 times fewer). It also does not add inference latency, and it often performs on par with full fine-tuning. *Quantization-aware low-rank adaptation* (QLoRA) is an extension of LoRA that quantizes all the weights of the model. Its training process is more memory efficient, albeit slower. On the other hand, the quantized fine-tuned model takes up less space and is therefore faster than LoRA.

Preference tuning

> You can post-train a model by having it generate two outputs to the same prompt and giving it feedback on which output is better. When such preferences are provided by humans and the post-training is carried out through reinforcement learning, it's RLHF. Often, the post-training is carried out using direct preference optimization (DPO) because it is more efficient. DeepSeek introduced group relative policy optimization (GRPO), wherein multiple responses are generated and a response is assigned a score that is normalized by the average reward of the group.

The type of post-training you can do is intimately connected to the structure of the dataset. If the dataset consists purely of text completions, then you can use it to do unsupervised training, such as teaching a model new vocabulary or completely new associations via CPT. If the dataset consists of ideal responses to inputs (input-output pairs), then you can use it for SFT or instruction tuning. If the dataset consists of two responses to each input and notes which of the two is preferable, then you can use it for preference tuning. You can do any of these forms of post-training in a parameter efficient way or on quantized models.

You will encounter post-training in several patterns in this book, including Content Optimization (Pattern 5 in Chapter 2), Adapter Tuning (Pattern 15 in Chapter 5), and Prompt Optimization (Pattern 20 in Chapter 6). At the time of writing (June 2025), open weights models are the only ones that support all the forms of post-training above. If you are using a hosted model, please consult up-to-date documentation from your model provider to determine whether a pattern that requires post-training is possible (or change to a model that provides the needed capability).

Fine-Tuning a Frontier Model

Companies like OpenAI and Anthropic, as well as hyperscalers such as AWS and Google Cloud, have streamlined the process of post-training frontier models using SFT. It’s possible to upload a training dataset of input-output pairs and launch the fine-tuning process, and the result will be an endpoint of an adapter-tuned model that can be used just like the foundational model.

We’ll illustrate this by using OpenAI’s GPT series of models, but fine-tuning Anthropic Claude on Amazon Bedrock or Google’s Gemini on Vertex AI is quite similar. Once you generate your training pairs (you’ll need at least a hundred pairs, but a couple of thousand pairs are even better), you’ll need to store them in a JSON line-formatted file. You can then load the training dataset as follows:

```
training_file = client.files.create(
    file=open("training_data.jsonl", "rb"),
    purpose="fine-tune"
)
```

Once the data is loaded, you can kick off a fine-tuning job like this:

```
job = client.fine_tuning.jobs.create(
    training_file=training_file.id,
    model="gpt-3.5-turbo"  # Base model to fine-tune
)
```

The fine-tuning will take a few minutes to a few hours, depending on the size of the training dataset and the model. Once the training is completed, you can query the model identifier for your fine-tuned LLM like this:

```
job_status = client.fine_tuning.jobs.retrieve(job.id)
if job_status.status == 'succeeded':
    print(f"Model ID: {job_status.fine_tuned_model}")
```

The result will be a model ID of the following form:

```
ft:<BASE MODEL>-0125:<ORG_NAME>::<JOB ID>
```

Use this model ID in the inference API to invoke the fine-tuned model. It has the same API as the base model:

```
completion = client.chat.completions.create(
    model=job_status.fine_tuned_model,  # Use the fine-tuned
model
    messages=messages
)
print(completion.choices[0].message.content)
```

Fine-Tuning an Open-Weight Model

Unsloth.ai provides you with the capability to fine-tune and train open-weight LLMs like Gemma and Llama. You can run Unsloth on your local hardware or use its managed fine-tuning services.

To fine-tune the 4-bit quantized version of Llama 3, start by loading in the model and its *tokenizer* (the class that will break an input text sequence into tokens of the sort the model expects):

```
from unsloth import FastLanguageModel
max_seq_length = 2048
model, tokenizer = FastLanguageModel.from_pretrained(
    model_name="unsloth/Meta-Llama-3.1-8B-bnb-4bit",
    max_seq_length=max_seq_length,
    load_in_4bit=True,
    dtype=None,
)
```

Then, attach a set of adapter weights to the base Llama model:

```
model = FastLanguageModel.get_peft_model(
    model,
    r=16,
    target_modules=["q_proj", "k_proj", "v_proj", "up_proj",
"down_proj",
                    "o_proj", "gate_proj"],
    use_gradient_checkpointing="unsloth"
)
```

In the previous code, you specified the *rank* (or *matrix size*) of the LoRA layer and the specific layers of the model to which they are to be applied. The previous code also applies LoRA to attention mechanisms (Q, K, and V matrices) and various projection layers.

Assuming that your input and output pairs are in one of the formats Unsloth supports, you can load the dataset using the following code:

```
dataset = load_dataset("...", split="train")
dataset = dataset.map(apply_template, batched=True)
```

Then, on a machine with sufficiently powerful GPUs, you can launch the training process by using the following code:

```
trainer=SFTTrainer(
    model=model,
    tokenizer=tokenizer,
    train_dataset=dataset,
    dataset_text_field="text",
)
trainer.train()
```

Once the model is trained, you can save it and push it to Hugging Face. There, you can merge the base model and its adapter layer into a single model before pushing it:

```
model.save_pretrained_merged("model", tokenizer,
save_method="merged_16bit")
model.push_to_hub_merged("...", tokenizer,
save_method="merged_16bit")
```

Then, you can use this model just like the base model.

Considerations

Fine-tuning models gives you the unique ability to customize LLMs to your domain-specific use cases, but fine-tuned models come with additional complexities—so make sure that the benefits of fine-tuning are worth these additional headaches:

Data requirements

Instead of providing a few in-context examples, you must use a more significant number of samples (more than a hundred) to customize the range of tasks the LLM can handle. This will require you to collect the samples ahead of time before attempting to fine-tune your LLMs. If you don't have the time or resources right now to do that, consider starting with in-context learning, collecting the data, and then fine-tuning your model later to boost its performance and provide more consistent, reliable outputs.

Catastrophic forgetting

Fine-tuning LLMs can lead to *catastrophic forgetting*, in which the model overemphasizes the examples provided during fine-tuning and loses its previously acquired knowledge. This wipes out LLMs' primary advantage: their comprehensive world knowledge. You can mitigate this issue by fine-tuning your model on a small dataset, for only a few epochs, and by selecting an appropriate learning rate during the fine-tuning process. Generally, you should begin the fine-tuning learning rate at the value where the pretraining phase concluded (typically around 1e-5).

Additional complexity

Before releasing the fine-tuned model to a production environment, you need to evaluate its performance and make sure it hasn't picked up any counterproductive constructs, like biased language. Also, whenever a new version of your foundational model is released, you must fine-tune the model again and redo the training. Finally, you need to track the lineage of the training and validation data used to train the fine-tuned model. These are additional tasks that require careful execution, and using in-context learning is far simpler since it only requires you to add a handful of examples.

Additional costs

Providers like OpenAI charge a higher per-token rate for inference on fine-tuned models than for requests to their standard models. They charge a higher price because prompts to fine-tune models can be much shorter and can elicit the same or higher-quality output. Since providers calculate prices on input and output tokens, they increase token pricing to recover the overhead costs of hosting your fine-tuned model.

On the other hand, when you fine-tune open models, the model inference cost reduces—but you must pay for the GPU costs while performing the fine-tuning procedures. Depending on the base LLM, this can start at a few dollars but quickly spiral to hundreds of dollars per model version.

The Organization of the Rest of the Book

The rest of this book covers 32 design patterns, organized into eight chapters. You'll learn how to control model outputs, enhance knowledge retrieval, improve reasoning capabilities, increase reliability, enable action, optimize performance, and implement safeguards. The section on each pattern includes a clear problem statement, a solution approach, practical usage scenarios, and code examples. We hope that this book will help you learn how to build robust and effective GenAI applications.

In Chapter 2, we show you how to control the style and format of AI-generated content—which is a critical skill for ensuring brand consistency, accuracy, and compliance. You'll learn how to implement Logits Masking (Pattern 1) to ensure that text conforms to specific style rules by intercepting the generation process at the sampling stage. The section on Grammar (Pattern 2) will teach you to constrain outputs to specific formats or data schemas using formal grammar specifications. With Style Transfer (Pattern 3), you'll discover how to convert content to mimic specific tones through few-shot learning or fine-tuning. The section on Reverse Neutralization (Pattern 4) will show you how to generate content in specialized styles by first creating neutral content and then transforming it. Finally, Content Optimization (Pattern 5) will equip you with methods to determine optimal content styles through systematic comparison and preference tuning—which are particularly valuable for marketing, advertising, and educational materials where effective style factors aren't immediately obvious.

In Chapters 3 and 4, you’ll learn patterns that can help you build AI systems that leverage external knowledge sources to address fundamental limitations like knowledge cutoffs, confidential data access, and hallucinations. You’ll begin with Basic RAG (Pattern 6) and learn to ground AI responses in relevant information from knowledge bases. The section on Semantic Indexing (Pattern 7) will teach you to capture meaning across different media types by using embeddings, thus moving beyond simple keyword matching. With Indexing at Scale (Pattern 8), you’ll master techniques for managing outdated or contradictory information through metadata, filtering, and reranking. Index-Aware Retrieval (Pattern 9) will equip you with advanced methods like hypothetical answers, query expansion, and GraphRAG to improve retrieval quality. Node Postprocessing (Pattern 10) will show you how to handle irrelevant content and ambiguous entities through reranking and contextual compression. You’ll learn to build Trustworthy Generation (Pattern 11) systems that maintain user trust despite inevitable errors, and finally, the section on Deep Search (Pattern 12) will teach you iterative processes for complex information retrieval that overcome context window constraints and enable multihop reasoning.

In Chapter 5, we discuss powerful techniques to enhance the reasoning and specialized capabilities of language models. You’ll learn Chain of Thought (CoT) (Pattern 13), which enables models to break down complex problems into intermediate reasoning steps and dramatically improve their performance on mathematical problems and logical deductions. The section on Tree of Thoughts (ToT) (Pattern 14) will teach you to implement tree search approaches for problems requiring exploration of multiple solution paths—which are ideal for strategic thinking and planning tasks. With Adapter Tuning (Pattern 15), you’ll discover how to efficiently specialize large models by training small add-on neural network layers while keeping original model weights frozen, thus making specialized adaptation practical with limited data (from 100 to 10,000 examples). Finally, the section on Evol-Instruct (Pattern 16) will show you how to efficiently generate high-quality instruction-tuning datasets by evolving instructions through multiple iterations, thus enabling you to teach models new domain-specific tasks without extensive manual data creation.

In Chapter 6, you'll encounter patterns for building more reliable AI systems that can be trusted in production environments. You'll learn LLM-as-Judge (Pattern 17) to evaluate generative AI capabilities through detailed, multidimensional feedback—which is a foundational skill for comparing models and tracking improvements. The section on Reflection (Pattern 18) will teach you how to enable models to correct earlier responses based on feedback, which significantly improves reliability in complex tasks. With Dependency Injection (Pattern 19), you'll master techniques for independently developing and testing each component of an LLM chain, thus making your systems more maintainable and robust. Finally, the section on Prompt Optimization (Pattern 20) will show you how to systematically set and update prompts by optimizing them on example datasets, which reduces maintenance overhead when dependencies change and ensures consistent performance over time.

In Chapter 7, we discuss ways to transform your AI systems from passive information providers into active agents that can take meaningful actions in the world. You'll master Tool Calling (Pattern 21) to learn how to bridge LLMs with software APIs so they can invoke functions with appropriate parameters and incorporate the results into their responses. This enables real-time data access, connections to enterprise systems, and complex calculations. The section on Code Execution (Pattern 22) will teach you to leverage LLMs to generate code that can be executed by external systems—which is perfect for creating visualizations, annotating images, or updating databases. With Multiagent Collaboration (Pattern 23), you'll learn to design systems of specialized single-purpose agents that are organized in ways that mimic human organizational structures, thus enabling complex reasoning, multistep problem solving, collaborative content creation, and self-improving systems that can handle extended interactions without human intervention.

In Chapter 8, you'll learn essential patterns for deploying generative AI within real-world constraints of cost, latency, and computational resources.

The section on Small Language Model (SLM) (Pattern 24) will demonstrate how to leverage smaller, more efficient models that can run on edge devices or with limited resources while still delivering acceptable performance for specific tasks. With Prompt Caching (Pattern 25), you'll discover techniques to reduce redundant computations and API calls and thus significantly lower costs for frequently requested content. Inference Optimization (Pattern 26) will equip you with methods to maximize throughput and minimize latency through techniques like speculative decoding, continuous batching, and prompt compression. The section on Degradation Testing (Pattern 27) will show you how to systematically evaluate model performance across different deployment scenarios and thus ensure consistent quality over time. Finally, the Long-Term Memory (Pattern 28) section will demonstrate how to maintain user history and dynamically apply personalization.

Chapter 9 will equip you with critical patterns for ensuring your generative AI applications operate safely, ethically, and within appropriate boundaries. The section on Template Generation (Pattern 29) will teach you how to pregenerate and review templates that require only deterministic string replacement at inference time—which is ideal for high-volume personalized communications where human review isn't scalable. With Assembled Reformat (Pattern 30), you'll learn to separate content creation into low-risk steps: first assembling data safely and then formatting it attractively to reduce the risk of inaccurate or hallucinated content. The section on Self-Check (Pattern 31) will show you how to use token probabilities to detect potential hallucinations cost-effectively in factual responses. Finally, Guardrails (Pattern 32) will equip you with comprehensive approaches to wrapping LLM calls with preprocessing and postprocessing layers that enforce security, privacy, content moderation, and alignment constraints—which is essential whenever your application could face malicious adversaries.

Finally, Chapter 10 demonstrates how the patterns from the first nine chapters can be composed into a production-ready agentic application.

1 Look carefully at the sketch. Do you see any issues? Do all the characters have the appropriate number of fingers? Is the girl's face symmetric? Does the board game correspond to anything recognizable? At the time of writing, image models struggle with symmetry, count, and exact recall.

2 A difference of 400 in the Elo rating corresponds to 10:1 odds that the higher-ranked player will beat the lower-ranked one.

Chapter 2. Controlling Content Style

The patterns in this chapter all have to do with controlling the style of the content (such as text, images, and video) generated by *foundational models*, which are models that have been trained on large datasets and are capable of generating a wide variety of content. However, the style of that content will, by default, be based on the training process that the model provider has used. Even if you stick to a single model version, model responses are *stochastic*—which means you may get different responses even if you repeat a question exactly (assuming that there is no caching going on). This means that downstream applications and end users that use responses from a GenAI model will have to deal with quite a wide variety of possible styles.

For example, we asked a number of foundational models the same question:

> What’s a good side dish for pierogi? Answer in a single sentence.

We then recorded the answers (see Table 2-1). As you can see, even when we restrict the answer to a single sentence, different models can answer the same question very differently. Imagine how much more diverse the results would be had we not restricted the style of the response in any way!

Table 2-1. Answers to the same question in different styles (answers retrieved from the models in February 2025)

Model	Model provider	Answer
GPT-4	OpenAI	A great side dish for pierogi is sautéed onions with butter and a sprinkle of crispy bacon bits.
Claude Sonnet 3.5	Anthropic	A tangy sauerkraut or caramelized onions complement pierogies perfectly by adding contrasting acidity or sweetness to the dumplings.
Gemini 2.0 Flash	Google	Sautéed onions and mushrooms are a classic and delicious side dish for pierogi.
Llama 3.2 70B	Meta	A traditional and delicious side dish for pierogi is fried onions and sour cream, but other popular options include sautéed spinach, braised red cabbage, or a simple green salad with a light vinaigrette.
DeepSeek-R1	DeepSeek	A tangy sauerkraut salad or caramelized roasted carrots with dill make excellent, flavorful sides for pierogi.
Mistral Small 24B	Mistral AI	A classic side dish for pierogi is coleslaw, especially when garlic and herbs are added.

How can you control (or restrict) the style of the response? Naturally, the answer depends on your goals—do you want to control the tone, the vocabulary, the reading level, or the formatting? You can try to control any of these aspects of style by using prompt engineering, but such an approach is extremely brittle—the results will vary from model to model and from one attempt to another. The patterns in this chapter provide a variety of more sophisticated and robust solutions to the problem of controlling style, so either choose the one that best meets your needs or combine the patterns.

Logits Masking (Pattern 1) ensures that generated text conforms to a set of rules. Grammar (Pattern 2) ensures that generated text conforms to a user-specified schema or standard data format—which is like Logits Masking, but it's carried out server-side by the model provider. Style Transfer (Pattern 3) uses example translations to ensure that text or generated images have the desired characteristics of some reference content. Reverse Neutralization (Pattern 4) provides a way to perform style transfer when only the reference content is available. Finally, Content Optimization (Pattern 5) is a way to choose whichever style performs best, without having to identify what the factors of that style are.

Pattern 1: Logits Masking

Logits Masking provides a way for application clients of a foundational model to ensure that the text the model generates conforms to a set of rules. These rules are often static, but in some cases, they can change based on the content already generated.

Problem

Sometimes, when you're generating text using LLMs, you want it to conform to specific style rules. These rules might be in place for branding, accuracy, compliance, or other reasons.

Here are some illustrative rules to help you understand the kinds of problems that this pattern tries to solve:

Branding

> If the text is talking about Item A, it should use brand words associated with that model (*sporty*, *performant*, and so on) and not words associated with Item B (*spacious*, *luxurious*, and the like).

Accuracy

> When you're generating a letter for bill payment, make sure the invoice ID and amount due are not repeated in the text of the letter. Such repetition increases the chances of error, perhaps because only the values at the canonical location are validated.

Compliance

> If the answer to a question refers to a case study involving Customer A, make sure that the text does not include any mention of its competitors (B, C, and D, who are also our customers). Customers may have agreed to let us refer to their case studies, but only in content that does not refer to their direct competitors.

Stylebook

> You may be in an industry with a stylebook, such as *The Chicago Manual of Style*, or publishing in a venue that requires adherence to the APA citation style.

The naive approach, or *antipattern*, is to generate content, evaluate it against the relevant rules, and regenerate it if the content doesn't conform to the rules (see Figure 2-1). However, such a "try-and-try-again" approach only works for edge cases when very few responses (less than 10% or so) need to be regenerated. Otherwise, multiple retries will dramatically increase latency and sometimes won't even converge toward an acceptable answer.

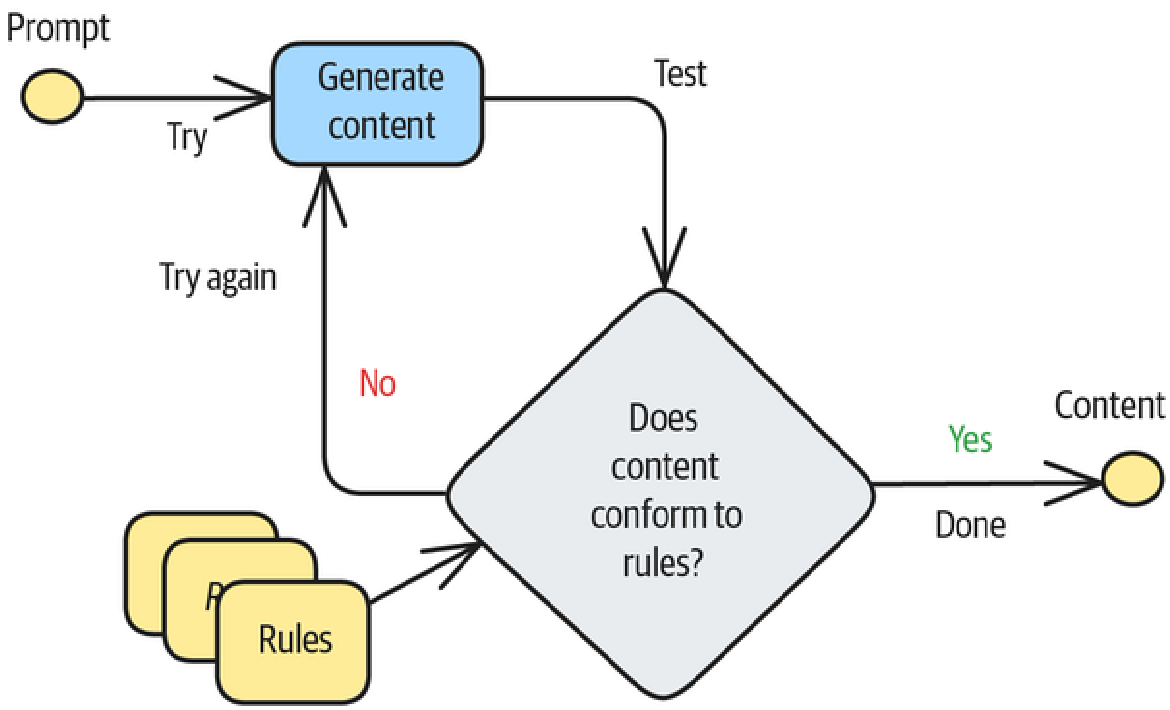


Figure 2-1. An antipattern: the try-and-try-again approach of applying style rules to content

WHEN IS THE TRY-AND-TRY-AGAIN APPROACH ACCEPTABLE?

If content has to be generated 3.2 times on average, then the typical latency will be 3.2 times that of a single call. The *tail latency*, which is the latency of the slowest requests, will be much greater because there is no guarantee that the approach will ever converge. Also, reducing the number of attempts allowed will increase the *refusal rate* of your application.

Whether or not an attempt is successful is a binary outcome, and you need to try only until the first success. Because the number of attempts needed in such a process follows a geometric distribution, you can calculate the average and tail latency. If only *p*% of your generations succeed, then you can estimate the average and tail latency of the try-and-try-again approach as follows:

- The average number of generations required before you get a successful one is given by this formula: 100/*p*. For example, if 10% of your generations fail, then 90% succeed—so you will need 100/90, or 1.1, attempts on average, and the average latency goes up by 10%. On the other hand, if 70% of your generations fail, only 30% succeed—so you will need 100/30, or 3.3, attempts on average, and the average latency goes up by 230%. If you use an *exponential backoff* method that increases the time between attempts to avoid overloading the server, then the latency increase will be even higher.

- There's no closed formula for the 99th percentile of the latency. Instead, you have to estimate it numerically from the geometrical distribution. You can use the online calculator at eStat to look up the correct value. For a 90% success rate, plug in 0.9 for the value of *p* and look for 0.99 in *P*(*X* <= *x*). The 99th percentile of the number of attempts required is 2 if *p* = 90 and 13 if *p* = 30!

> The bottom line is that the try-and-try-again approach is acceptable only if the LLM call has a very high success rate.

A better approach to making generated text conform to a set of rules is to use Logits Masking.

Solution

As discussed in "Beam Search", foundational models generate text by sampling from a sequence of possible continuations. The idea behind Logits Masking is to intercept the generation at this sampling stage.

Logits Masking works as follows:

- Rather than wait until the full content is generated, you obtain the set of possible continuations at each intermediate point.
- You zero out the probability associated with continuations that do not meet the rules.
- As long as there is at least one continuation that meets the rules, generation can proceed.
- If there is no continuation that meets the rules or if the generation is at a point that you have previously encountered as a dead end, you need to back up one step and retry generation.
- After some maximum number of generation attempts, you send a refusal back to the user saying that you are unable to generate content that meets the rules.

The impact of Logits Masking is to prune nonconforming beams in beam search; this ensures that generated text conforms to specific rules.

Figure 2-2 depicts how to implement Logits Masking. The solid gray boxes show the sequence selection approach that suffices for simple use cases, and the hatched gray boxes show the sequence regeneration steps that are needed in more complex situations.

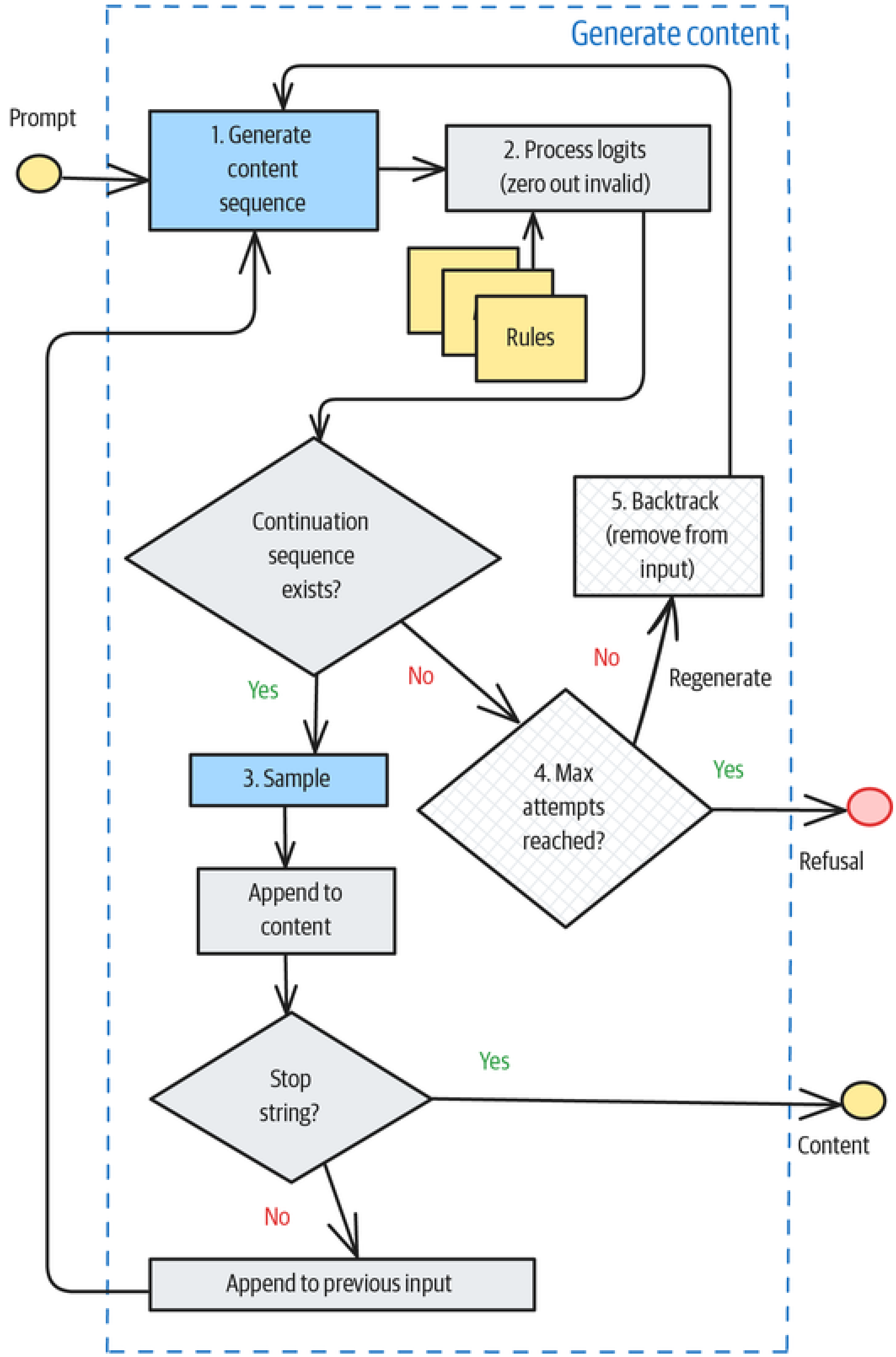


Figure 2-2. How to implement Logits Masking

To demonstrate these steps, we'll use the Transformers library. We'll show only the relevant code snippets; the full code for this pattern is in the *01_logits_masking folder* of the GitHub repository for this book.

THE TRANSFORMERS LIBRARY

The Transformers library from Hugging Face simplifies working with pretrained Transformer-based foundational models, regardless of the ML framework (Tensorflow, PyTorch, or JAX) that was used to train the foundational model. It offers support for a wide range of tasks such as text classification, question answering, summarization, translation, and text generation. For example, to do sentiment analysis using Llama, you could use the following:

```
from transformers import pipeline
MODEL_ID = "meta-llama/Llama-3.2-3B-Instruct"
classifier = pipeline('sentiment-analysis', MODEL_ID)
result = classifier("""I am learning a lot of neat concepts 🧑🏻‍💻
💡✍🏻 from the
GenAI design patterns book. 📖🐦
""")
```

Through Hugging Face, the library also provides access to thousands of pretrained models and simplifies the process of fine-tuning these models for specific downstream tasks and deploying them.

We are using Transformers to illustrate Logits Masking only so that we have a concrete implementation with which to illustrate the steps needed and discuss the controlling parameters. You can carry out these steps by using other LLM frameworks or even the model providers' client libraries, although the details will vary.

Step 1: Intercepting sampling

To intercept sampling and get access to the probability of each continuation sequence, you need to create a `LogitsProcessor` subclass, which is part of Transformers' generation library. In the `LogitsProcessor` subclass, you initialize the tokenizer and any parameters you need to apply your rules as follows:

```
class MyRulesLogitsProcessor(LogitsProcessor):
    def __init__(self, tokenizer, rules):
        self.tokenizer = tokenizer
        self.rules = rules
```

Once you've created the logits processor, you have to pass it in when you invoke the Transformer pipeline. To do so, you create a text generation pipeline as normal:

```
from transformers import pipeline
pipe = pipeline(
    task="text-generation",
    model=MODEL_ID,
)
```

Then, when invoking the pipeline, you specify the size of a continuation sequence (`max_new_tokens`), the number of possible continuations you want to generate (`num_beams`), and an instance of your `LogitsProcessor` subclass that will intercept the sampling:

```
rules_processor = MyRulesLogitsProcessor(pipe.tokenizer, rules)
results = pipe(input_message,
               max_new_tokens=256,
               do_sample=True,
               temperature=0.8,
               num_beams=10,
               logits_processor=[rules_processor])
```

Step 2: Zeroing out invalid sequences

You implement the core of the interception (see step 2 in Figure 2-2) by overriding the `__call__` method of the logits processor, which receives a set of sequences and their corresponding probabilities in the form of logits:

```
def __call__(
    self, input_ids: torch.LongTensor, input_logits:
torch.FloatTensor
```

```
) -> torch.FloatTensor:
    output_logits = input_logits.clone()
    # make changes to the output_logits based on your rules
    return output_logits
```

Note two things about the signature of the `__call__` method in the previous code. First, the inputs are token IDs, not natural-language characters. Typically, you'll have to decode the tokens before applying the rules, so you'll do this by using the following:

```
seq = self.tokenizer.decode(input_id)
```

Second, because logits are the log of the probability, *zeroing out* the probability of a sequence means setting the logits to negative infinity. Do this by using the following:

```
output_logits[idx] = -np.inf
```

Putting this together, the implementation of the `__call__` method will be as follows:

```
def __call__(
       self, input_ids: torch.LongTensor, input_logits:
torch.FloatTensor
    ) -> torch.FloatTensor:
        output_logits = input_logits.clone()
        for idx, input_id in enumerate(input_ids):
            seq = self.tokenizer.decode(input_id)
            if not self.apply_rules(seq, self.rules):
                output_logits[idx] = -np.inf # zero out
      return output_logits
```

This code snippet assumes that you have an appropriate `apply_rules()` available.

Steps 3 (sampling, per Figure 2-2) and 4 (determining whether the maximum number of attempts have been exceeded) are straightforward, so let's finish this section by looking at how to do step 5.

Step 5: Backtracking and regenerating sequences

In the Transformers library, the pipeline call doesn't allow you to backtrack and regenerate (see step 5 in Figure 2-2). To do that, you have to take full control of the generation loop, and you do it by calling the `pipe.model.generate()` method for set numbers of tokens (16 in this example) at a time:

```
results = pipe.model.generate(
            **input_ids,
            max_new_tokens=16,
            num_beams=10,
            output_scores=True,
        )
```

The `input_ids` for the first call to `generate()` consists of the prompt from the user. Then, you need to append the generated sequences from previous steps. Therefore, to create the `input_ids` parameter, you'd do something like this:

```
input_ids = pipe.tokenizer(
        input_prompt + '\n'.join(text_so_far)),
        return_tensors="pt").to("cuda")
```

The preceding code appends previously generated text to the original prompt, tokenizes it into IDs, and sends the sequence of IDs as a PyTorch tensor (`return_tensors="pt"`) that's ready for GPU computations (`to("cuda")`) by the model.

A typical loop around `generate()` will involve logic to (re)initialize the generation, apply the rules, backtrack and remove previously generated sequences, or stop the generation. You'll want to maintain the necessary state variables—and the reasons for this will become clearer when we examine concrete examples in the next section.

You'll also need some logic to stop the generation. A common approach is to stop the generation when the model outputs a *stop string*. The input prompt could include an instruction to output the stop string, or the stop string could be part of examples provided in the context. This stop string can also be passed into the `generate()` call to do an early stop while generating a sequence (for example, before 16 new tokens are generated).

Examples

To illustrate Logits Masking, we'll use a couple of examples. The first is a simple problem in which you can simply select a continuation sequence (Steps 1–3 in the previous section, as depicted in Figure 2-2). The next is a more complex problem to illustrate sequence regeneration after backing up one or more generation sequences (in Steps 1–5).

Sequence selection

Imagine that you're a product marketer for nutritional supplements and you need to write product descriptions that will go on the pages of ecommerce sites. These sites tend to reject certain phrases like *award winning*, but based on search engine optimization and analysis of your website traffic, you might know good phrases to include in your product instead.

For the purposes of this book, let's use the set of top nutrition keywords that was created by MarketKeep using Google Ads Keyword Planner. The code for this example is in the logits_masking notebook on GitHub, so please follow this discussion by looking at that code.

Zero-shot learning doesn't work

We could try to do this with a zero-shot prompt:

```
system_prompt = f"""You are a product marketer for a company that
makes nutrition
supplements. Balance your product descriptions to attract
customers, optimize
SEO, and stay within accurate advertising guidelines.
"""
user_prompt = f"""Write a product description for a protein
drink."""
```

The result, though, does not meet our requirements:

> Introducing PowerBoost, a delicious and convenient protein drink that helps you fuel your active lifestyle. With 20 grams of protein and 0 g of sugar, this refreshing beverage supports muscle growth and recovery after your toughest workouts. Made with high-quality whey protein and essential vitamins, PowerBoost is the perfect way to recharge and refuel on the go.

It does include two good SEO terms—*whey* and *whey protein*—but unfortunately, it also includes three words that increase the odds of being banned: *quality*, *growth*, and *perfect*. Sure, *high-quality* and *muscle growth* seem innocuous, but why take the chance?

Using Logits Masking

Now, try using Logits Masking to choose continuations that have the maximum number of positive words and the least number of negative words. First, write an evaluation function that counts the occurrences:

```
def evaluate(descr: str, positives, negatives) -> int:
    descr = descr.lower()
    num_positive = np.sum([1 for phrase in positives if phrase in
descr])
    num_negative = np.sum([1 for phrase in negatives if phrase in
descr])
    return int(num_positive - num_negative)
```

To do this, write a subclass of `LogitsProcessor` and select the continuation sequence that has the characteristics you desire:

```
class BrandLogitsProcessor(LogitsProcessor):
    ...

    def __call__(
        self, input_ids: torch.LongTensor, input_logits:
torch.FloatTensor
    ) -> torch.FloatTensor:
        output_logits = input_logits.clone()

        num_matches = [0] * len(input_ids)
        for idx, seq in enumerate(input_ids):
            # decode the sequence
            decoded = self.tokenizer.decode(seq)
```

```
            # count the number of words that start with desired
letter
            num_matches[idx] = evaluate(decoded, self.positives,
self.negatives)
        max_matches = np.max(num_matches)

        # logits goes from -inf to zero.  Mask out the non-max
sequences;
        # torch doesn't like it to be -np.inf
        for idx in range(len(input_ids)):
            if num_matches[idx] != max_matches:
                output_logits[idx] = -10000

        return output_logits
```

When we (the authors) did this, we got a product description that included several “good” terms (*whey*, *whey protein*, and *nutrients*) and avoided any of the words that could get us banned:

> Fuel your active lifestyle with our premium protein drink, packed with 20 grams of whey protein, 10 grams of branched-chain amino acids (BCAAs), and essential vitamins and minerals to support muscle recovery and overall well-being. Our unique blend of whey protein isolate and micellar casein provides a sustained release of nutrients, helping to build and repair muscle tissue. With no artificial flavors or sweeteners, our protein drink is a guilt-free way to support your fitness goals. Enjoy the taste of a refreshing beverage while nourishing your body with the nutrients it needs to thrive.

Note the use of *premium* to get around the problem with *quality*. We’ll always get a product description, and the description that we get is the best possible one we could have gotten, given the set of potential continuation sequences. Step 3 is automatically valid the way we are doing this because we will always have some sequence—even if it has no positive words.

Sequence regeneration

Now, let’s look at a more complex problem where zero/few-shot learning simply doesn’t work and is unlikely to work in the absence of a powerful reasoning model that’s capable of backtracking and correcting its work. The full code for this section is in the sequence_regeneration notebook on GitHub.

This example involves poetry and might appear niche, but remember that the purpose of Logits Masking is to ensure that generated content conforms to a set of rules that you enforce programmatically. The rules can be quite complex and dynamic.

One-shot examples don't work

Imagine that you're generating poems for a children's book and you want to generate an *acrostic* poem about some animal. This means that the first letters of the poem need to spell out an adjective that's suitable for the animal. For example, the first letters of an acrostic poem about rabbits might spell out *quick* or *cute*, and the entire poem needs to be a single phrase that describes the animal or something the animal might do. Thus, an acrostic poem about rabbits might be one such as this:

> Quietly, the rabbit bides its time
>
> Under the garden deck
>
> In wait for
>
> Carrot greens,
>
> Kale, and parsley.

Putting the entire description in the opening paragraph of this subsection into the system prompt and using the acrostic poem about rabbits as a single-shot example, we asked Llama 3.2 to generate an acrostic about a tiger, and we got the following:

> Powerful eyes gleam in the night
>
> Occupying shadows, a fierce sight
>
> Ruling the forest with gentle might
>
> Elegant, a creature of beauty bright

The model probably tried to generate *POWER* as the starting letters for the lines of the poem but failed and instead came up with *PORE*. There are several ways to fix this. Reflection (Pattern 18 in Chapter 6) might work, for example. So would using a more powerful reasoning model and giving it "thinking tokens." Here, though, let's see how to use Logits Masking to solve this with a smaller model.

Initializing the poem

To initialize the acrostic, we'll constrain the starts of lines explicitly. To do so, we'll generate a list of adjectives for the animal by using the following prompt:

```
system_prompt=f""" You are an expert on words who has access to a
thesaurus.
Respond with a list of adjectives that could complete the phrase
"As ___ as a
{animal}" For example, for a rabbit, you could respond with
these: quick, fast,
gentle, playful
Respond with just a list of words without any introduction or
preamble.
"""
user_prompt=f"""Give me the best {num_words*3} adjectives that
would complete
the phrase 'As ___ as a {animal}'
"""
```

The resulting list of adjectives for *tiger* is this:

['wild', 'agile', 'regal', 'swift', 'fierce']

We then use the following prompt to generate an appropriate phrase to start the poem:

```
def get_phrase_that_starts_with(animal: str, letter: str):
    system_prompt=f"""
    You are writing a children's book. Write a phrase about a
{animal}
    that starts with the letter {letter}
    Respond with just the phrase without an introduction or
preamble.
```

```
    """
    user_prompt=f"""Write a phrase about a {animal} that starts
with the letter
    {letter}
    """
    input_message = [
        {"role": "system", "content": system_prompt},
        {"role": "user", "content": user_prompt}
    ]

    result = pipe(input_message, max_new_tokens=256)
    phrase = result[0]['generated_text'][-1]['content']
    return ' '.join(phrase.split()[:3]) # max 3 words
```

A phrase for the animal *tiger* that starts with the letter *N* would be as follows:

Nimbly navigating the

Then, we put these together to initialize the poem:

```
def initialize_poem(animal: str, allowed_start_words: [str]):
    # the weight of a word is inversely proportional to its
length
    lengths = [1.0*len(w) for w in allowed_start_words]
    max_len = np.max(lengths)
    weights = (max_len - lengths)
    weights = weights / np.sum(weights)

    start_word = random.choices(population=allowed_start_words,
weights=weights,
                                   k=1)[0].lower()
    start_letter = start_word[0].upper()
    return start_word, [get_phrase_that_starts_with(animal,
start_letter)]
```

This method returns the acrostic word and the starting point for a poem that starts with that word. So, for *tiger*, we might get the following:

```
('wild', ['Wrapped in warm,'])
```

Generating a poem

We start by initializing the poem and creating state variables to store the poem we've generated so far in this iteration:

```
def write_acrostic(animal: str, max_iter=10,
num_sequences_per_iter=10):
    allowed_start_words = get_potential_starts(animal, 10)
    start_word, poem_so_far, prev_start_poem = None, None, None
    for iter in range(max_iter):
        # reinitialize if we are stuck at a starting point
        if poem_so_far is None or poem_so_far == prev_start_poem:
            start_word, poem_so_far = initialize_poem(animal,

allowed_start_words)
        prev_start_poem = poem_so_far # for next iter
```

To generate the poem, we join the input prompt to the poem generated so far and ask the model to generate completion sequences:

```
# generate poem
inputs = pipe.tokenizer('\n'.join(
     get_input_prompts(animal, '\n'.join(poem_so_far),
start_word)),
     return_tensors="pt").to("cuda")

results = pipe.model.generate(
     **inputs,
     max_new_tokens=16,
     num_beams=num_sequences_per_iter,
     num_return_sequences=num_sequences_per_iter,
     output_scores=True,
     renormalize_logits=True,
     return_dict_in_generate=True,
)
```

Applying Logits Masking

Assuming that the acrostic start word is *BOLD*, we'll get back sequences of the following form:

Brightening up the forest floor

Often, the tiger stalks its prey

Lur

And here's another example:

Brightening up the forest floor

Lively eyes watch all around

Owning the night

We'll keep the first sequence and zero out the second, which doesn't fit the acrostic. Typically, you'll get multiple sequences that match. We could do random sampling based on the probability, but we'll take the simpler approach of doing *greedy decoding*, in which we choose the sequence with the highest probability:

```
candidate_starts = ''.join([line[0] for line in
candidate_poem]).lower()
continue_seq = False
found_poem = False
if len(start_word) >= len(candidate_starts) and
   start_word[:len(candidate_starts)] == candidate_starts:
     continue_seq = True
     if len(start_word) == len(candidate_starts):
          found_poems.append({
                        "poem": candidate_poem,
                        "prob": float(np.exp(seq_prob.cpu())),
                        "word": start_word
          }) # YEAH!

     if continue_seq:
          if seq_prob > best_prob_in_iter:
              best_prob_in_iter = seq_prob
              # even if a poem is found, the last line might be
incomplete,
              # so continue the sequence
              best_poem_in_iter = candidate_poem
```

In this code, we take advantage of knowing that if the number of complete lines generated equals the length of the acrostic word, then the poem is complete. We don't need an explicit stop string. We can add this poem to the list of found poems, along with the probability of the sequences that form the poem.

Regeneration logic

If, on the other hand, all the continuation sequences have been zeroed out, we backtrack by removing generated lines one at a time (starting at the end) until there is a sequence that meets the acrostic requirements. We can use the poem fragment that starts with "Brightening up the forest floor" as the starting point for the next call to `generate()`:

```
# remove the lines that don't fit and try again
while True:
    # remove a line, and see if it matches the start word
    best_bad_poem_in_iter = best_bad_poem_in_iter[:-1]
    if len(best_bad_poem_in_iter) == 0:
        # reinitialize, potentially to different start word
        start_word, poem_so_far = initialize_poem(animal,
allowed_start_words)
        break
     candidate_starts = ''.join([line[0] for line in
                                 best_bad_poem_in_iter]).lower()
     if len(start_word) >= len(candidate_starts) and
        start_word[:len(candidate_starts)] == candidate_starts:
        poem_so_far = best_bad_poem_in_iter # start from here for
same start_word
        break
```

If you end up stuck on a starting point, then reinitialize the poem (potentially with a new adjective for the animal) and continue the process.

Example output

For *tiger*, the process generates acrostic poems whose lines start with letters spelling out the words *BOLD* and *SWIFT*. Here's the first poem we get:

Boldly, the brave tiger stalks its prey

Owning the forest with its might,

Lurking in the shadows, waiting to pounce,

Daring to be the king of the night

And here's the second:

Smiling softly, the

Wild eyes gleam

In the

Forest depths,

Tigers stalk

For an owl, we get a poem such as this:

Silently swooping through the night

Hooting softly, a gentle sound

Alertly watching, with eyes so bright

Ruling the darkness, all around

Prowling quietly, with stealthy pace

As you can see, this process of taking over the generation loop and applying Logits Masking to select the continuation sequence gives us enough control to generate poetry that meets our strict style criteria, even when we have to backtrack and regenerate sequences.

Considerations

Logits Masking is a way of using much of the machinery of the LLM to generate text while imposing your preferences on the sampling. It's useful when continuation sequences can easily be censored to remove disallowed options.

The simple sequence selection approach works when censoring tends to leave behind a few valid options. In more complex scenarios, where it is highly likely that censoring will remove *all* of the generated options, you might need to backtrack and regenerate sequences.

Alternatives

If you think you have a problem that you can address by using Logits Masking, consider the following alternatives:

- Simpler ways of specifying a desired style include providing few-shot examples in the context (see Pattern 3, Style Transfer) and providing detailed instructions in the prompt through prompt engineering. However, these do not provide a strict enforcement mechanism—you can't be sure that your generated text will conform to the rules.
- Using a more powerful model might be an option because such models are typically better at following instructions. Reasoning models, where you provide thinking tokens to allow the model itself to retract and regenerate, may also work. However, more powerful models tend to cost more and be slower.
- Try-and-try-again (where you generate fully, test, and retry generation if the generated text doesn't conform to the rules) might be a reasonable option if the chances of conformance are high enough. The 99th percentile of the number of generations required is only 2 if $p = 0.9$.
- Logits Masking doesn't provide any hints to the model when you ask it to regenerate a sequence. If your rules engine provides helpful error messages, consider Reflection (Pattern 18), in which you update the prompt with an error message. This can reduce the number of attempts required to create conforming content.

- If the rules you want to apply can be represented in certain types of standard forms, you can offload Logits Masking to the model provider by providing it with the rules you want to impose. This is Pattern 2, Grammar, which we'll consider next.

Extension to autocomplete

An interesting application of Logits Masking is to implement autocomplete functionality. Autocomplete is a common feature of many web applications. Google, for example, suggests ways to complete a query when a user starts to type in its text input box. According to Google, the suggestions are based on "real searches and on word patterns found across the web."

Implementing autocomplete based on real searches in your own application will require you to log users' search terms and suggest query completions based on what previous users have typed in most often. However, because this process leaks data among users by providing insight into what other users are searching for, you may not be able to apply this approach in some application areas. One way around this security problem is to restrict autocomplete to previous queries within a specific deployment or made by an individual user. This, however, leads to a large number of cold-start situations, as many queries will be new.

In many situations, therefore, you might want to implement autocomplete solely based on word patterns. How can you do this? While you could build an index out of your documents and keep it up to date as your documents change, a simpler way might be to use Logits Masking, with the document held in the context. The approach goes like this:

1. Ask an LLM to complete the query that the user is typing in by adding a single phrase or sentence.
2. In the LogitsProcessor, obtain the list of top completions, show the user the top completions, and have them either select one or type a different phrase altogether.

3. If they select one of the completions, apply sequence selection and zero out the logits of the other possibilities. If they type a different phrase, use sequence regeneration, using that as the new starting point.

For a simple implementation that grounds the autocomplete in a document (rather than on the web), see the autocomplete notebook in the GitHub repository. In a real implementation, you would cache the entire document in the context to give yourself access to the full knowledge base without paying for input tokens on each query.

Caveats

If you want to use Logits Masking, the model you're using needs to provide access to logits. As we write this in June 2025, Anthropic's Claude family of models don't provide such access, but OpenAI, Google, and Meta do. Even among these three, the level of access varies. OpenAI provides read access to logprobs across almost all of its models. Google's Gemini Flash model, but not the Gemini Pro model, allows the use of `responseLogprobs` (recall that *logit* is the log of the probability). Meta's Llama is the most permissive, but it requires that you self-host the models. Model providers fear that providing logits makes it easier for others to train smaller models on the output of the foundational models, so in practice, using Logits Masking restricts your choice of foundational models. We hope that, with more widespread knowledge of the benefits of this pattern, other model providers will start to support the feature. In the example code for this pattern in the GitHub repository, we used Meta's Llama 3.2 model because it is open-weights and provides read-and-write access to the logprobs.

A second key consideration is that intercepting the sampling means that each sequence being generated requires communication between the model and the client code. Unless the model is locally hosted or deployed in such a way that you can run client code on a colocated processor (talk to your model provider about that), such communication requirements might add unacceptable latency. This means Logits Masking is often applicable only to locally hosted models. However, it provides a way for these smaller models to match the performance of larger, more costly models on certain types of complex problems.

A third consideration is that Logits Masking works by censoring certain generations. If there's no candidate token sequence that meets the rules, you won't be able to generate valid content. In the pattern discussion, we suggest regenerating from a different starting point in this situation, but a simpler way is to raise an error or refuse the request. It's important for AI engineers using Logits Masking to provide enough information in the prompt to make this a rare occurrence.

Self-Check (Pattern 31 in Chapter 9) is another use of logits.

References

In reinforcement learning, the idea of zeroing out logits is called *invalid action masking*. Its first recorded use was by Vinyals et al. (2019) in the game *StarCraft II*. Theoretical justification for this practice was provided by Huang and Ontañón (2020).

Romain Florenz (2025) creates social graphs with LLMs by using a finite-state machine as the Logits Masking mechanism that iteratively guides token generation.

Pattern 2: Grammar

The Grammar pattern provides a way to ensure that generated text conforms to style rules that can be represented as a *context-free metasyntax*—which is a formal way to describe the allowable structures and compositions of phrases and sentences while imposing few to no restrictions on the actual content. Common situations in which this is the case include ensuring that the content fits a specific data schema or is in a standard data format.

Problem

In many cases, you'd like the text generated by the LLM to follow a specific format—which could be as simple as a comma-separated list or as complex as a syntactically valid structured query language (SQL) statement. This is often because you are going to hand the generated text to a downstream application, which expects to operate on the LLM response without having to do all sorts of parsing and validation.

Some naive approaches would be to state the format you want in the prompt ("provide the output in JSON") or to provide a few examples of the format you want ("structure the output like this") and hope that the LLM always generates text that follows the syntax of the examples. Some model providers, such as Anthropic, encourage this approach, but the problem is that relying on the LLM's ability to follow instructions is brittle (since it has a chance of breaking each time the LLM version changes), unreliable (since LLM generation is stochastic), and costly (since it's typically the larger models that are better at instruction following). What makes this an antipattern is that every consumer of the LLM call has to guard against the LLM potentially failing to follow instructions.

A better approach is to represent the rules you want in a generalizable way, which is called a *grammar*. Then, the model framework will apply your grammar specification to constrain the set of tokens it generates, so that the generated text will conform exactly to the grammar.

Solution

When a foundational model generates text, it does so token by token. At each point, it generates a set of candidate tokens that could follow and then chooses among them. Some model providers and frameworks allow you to specify a grammar to apply to these candidate tokens. The model framework can restrict the next token to the ones legally allowed by the grammar, and it does so by zeroing out the probability of disallowed tokens.

> **TIP**
>
> You can think of the Grammar pattern as the model framework that does Logits Masking (Pattern 1) on your behalf. Therefore, we encourage you to also read through the section on Logits Masking to better understand the Grammar pattern. In particular, we cover logits processing in much more detail there.

You can specify the grammar directly by using a grammar-constrained logits processor, or you can use more user-friendly options, such as specifying a data format or passing in a schema description. Let's look at all three options. The full code for this pattern is in the examples/02_grammar folder in the GitHub repository—we show only relevant code snippets in our discussion, so refer to that code for complete details.

Option 1: Using the grammar-constrained logits processor

To use the Grammar pattern, you provide a formal grammar along with the model to the generation pipeline. The pipeline will ensure that the grammar rules are used to constrain the tokens that the model outputs (see Figure 2-3).

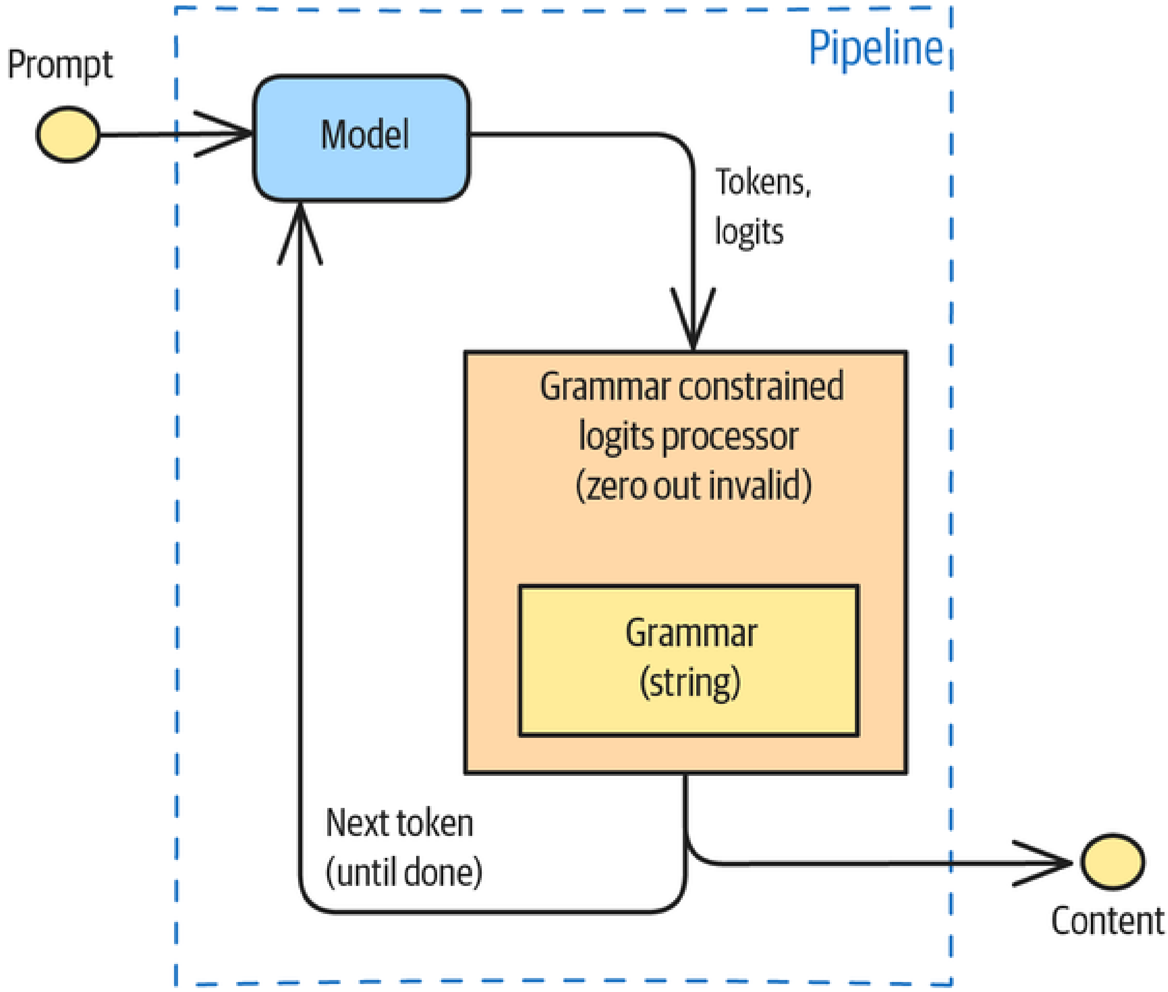


Figure 2-3. Using Grammar to constrain the tokens that are output by the model

When using the Transformers framework, you can accomplish this with three steps:

1. Represent the syntax you want in the form of a formal grammar.
2. Create a `LogitsProcessor` that will apply this grammar.
3. Pass in the logits processor to the pipeline.

Step 1: Create a formal grammar

The Transformer framework supports grammar specifications in Backus-Naur form (BNF). This is great because almost all formal formats and programming languages have readily available BNF descriptions. For example, you can find the BNFs for regular expressions, for a SQL

TIMESTAMP or CREATE INDEX statement, or for a line of a CSV file by using a quick internet search for “BNF for [*insert topic*].”

Thus, if you want the LLM to generate valid SQL timestamps, you’d specify the grammar as a string:

```
grammar_str = """
timestamp_literal ::=
{ t 'yyyy-mm-dd hh:mi:ss' } |'date_literal time_literal'

date_literal ::=
{ d'yyyy-mm-dd'}
  |mm-dd-yyyy| mm/dd/yyyy| mm-dd-yy| mm/dd/yy| yyyy-mm-dd
  | yyyy/mm/dd| dd-mon-yyyy| dd/mon/yyyy| dd-mon-yy| dd/mon/yy

time_literal ::=
{ t 'hh:mi:ss'}|hh:mi:ss[:mls]
"""
```

Because allowed timestamps could contain just the date or just the time, this grammar includes those two types as well.

Step 2: Create a logits processor that applies grammar

Second, use the grammar string to create a grammar-constrained logits processor, as follows:

```
grammar = IncrementalGrammarConstraint(grammar_str,
                                       "timestamp_literal",
                                       pipe.tokenizer)
grammar_processor = GrammarConstrainedLogitsProcessor(grammar)
```

You have to provide the root element of the grammar, which is `timestamp_literal`, when creating the constraint. In this way, you can pass in the grammar of the entire SQL spec and still select the specific data type or instruction that you want.

Step 3: Apply logits processing

Finally, pass the logits processor when invoking the pipeline:

```
results = pipe(input_message,
                max_new_tokens=256,
                do_sample=False,
                logits_processor=[grammar_processor])
```

Now, whenever the LLM generates text, the grammar-constrained logits processor will process the tokens, which will zero out the probability associated with any tokens that are not permitted by the grammar. Hence, the output will always be text that conforms to the desired grammar.

Option 2: Using standard data format

If you want the response to be in a standard data format that is directly supported by the model provider's API, then the usage is a lot simpler. For example, if you want JSON responses from OpenAI, simply specify JSON when making the call to the LLM:

```
response = client.chat.completions.create(
        model=MODEL_ID,
        messages=input_message,
        response_format={"type": "json_object"}
    )
```

Then, the content of the response message will be in JSON. It's essential that the prompt (either the system prompt or the user instruction) *explicitly* specifies that you want a JSON output, so that it will generate the necessary tokens.

> **WARNING**
>
> The XML parser in LangChain is *not* an example of the Grammar pattern, and that's because it relies on the model's instruction-following capability to create the XML tags needed. You should use it with care since, unlike with the Grammar pattern, there's no guarantee that you'll get back a compliant XML result.

Option 3: Using user-specified schema

In the previous JSON response, the JSON attributes and elements were not specified. But what if we want to specify the exact JSON attributes we need? Many models refer to this functionality as *structured output*, and they support it either through JSON itself or through Python `dataclass` (or Pydantic) objects.

For example, if you want OpenAI to generate a receipt consisting of line items, you can specify the schema of the output JSON as follows:

```
"schema": {
        "type": "object",
        "properties": {
            "quantity": {
                "type": "int",
                "description": "How many items were purchased"
            },
            "name": {
                "type": "string",
                "description": "Name of item purchased",
            }
        },
        "additionalProperties": false,
        "required": [
            "quantity", "name"
        ]
    }
```

OpenAI also supports the Python `dataclass`, as in the following Gemini example.

If you want Gemini to generate a receipt consisting of line items, you can create a Python `dataclass`:

```
@dataclass
class LineItem:
    description: str
    quantity: int
    amount: float

@dataclass
class Receipt:
```

```
    items: LineItem[]
    total_amount: float
```

Then, when invoking the model to generate content, you specify the schema:

```
response = client.models.generate_content(
    model='gemini-2.0-flash',
    contents=[f"Parse the receipt contained in the image",
image],
    config={
        'response_mime_type': 'application/json',
        'response_schema': Receipt,
    },
)
```

The response itself is still a string and will be in JSON format. The Pydantic object is used solely to specify the structure of the JSON, but you can use the JSON parsing library in Python to parse the JSON text into an object belonging to the `dataclass`:

```
import json
data_obj = json.loads(
    response.text,
    object_hook=lambda args: Receipt(**args)
)
```

Python data classes provide a flexible and powerful way to constrain the style of a GenAI model's output. To implement the structured outputs feature, the model provider takes on the responsibility of performing Logits Masking (Pattern 1). It does this server-side by converting the schema or data class into rules or logic (see Figure 2-4).

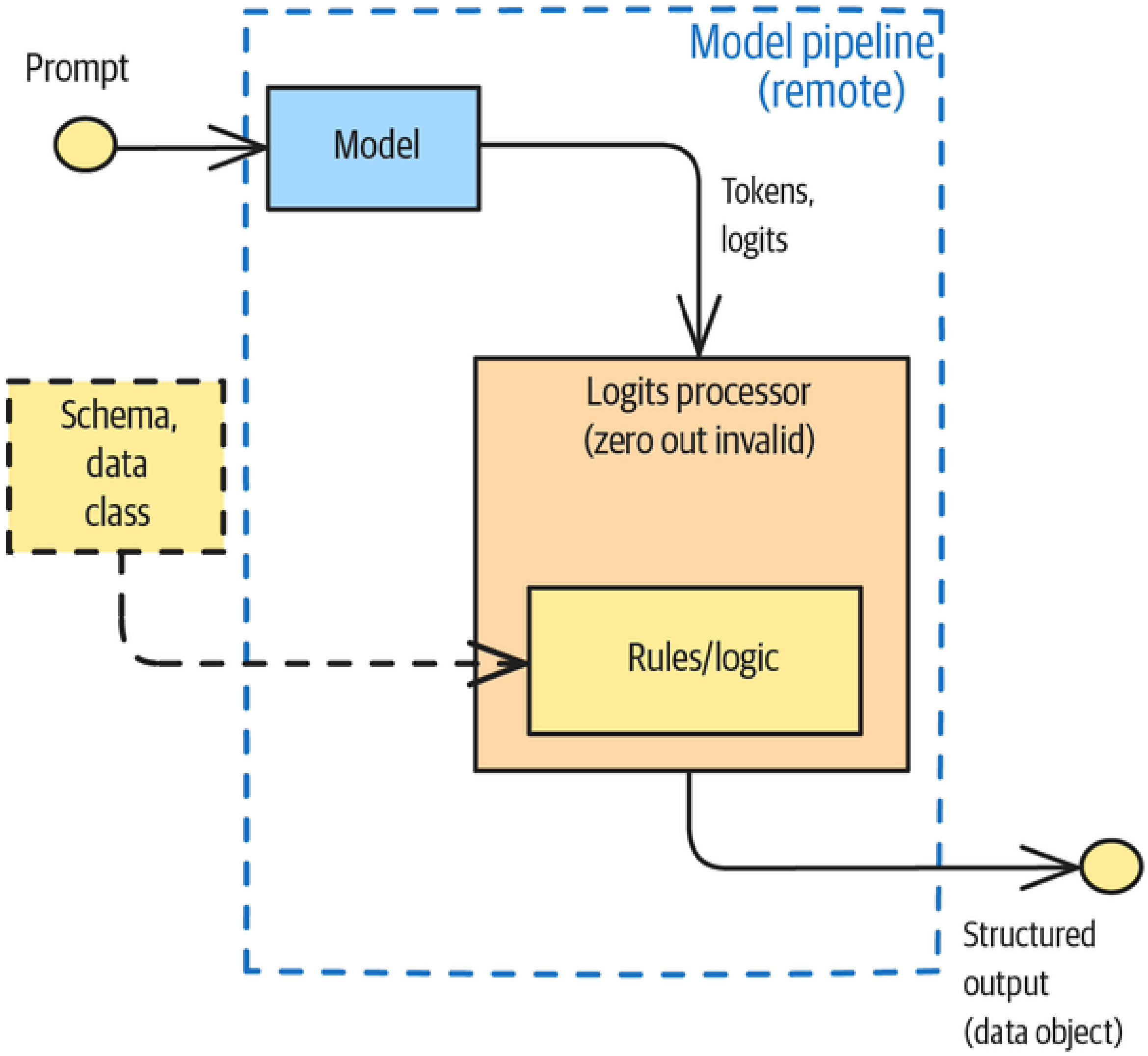


Figure 2-4. Implementing the structured outputs feature

Examples

Let's look at a few illustrative examples for each of the options discussed in the previous section.

Arithmetic expressions

Suppose you're writing educational software for elementary school students. You want the software to generate arithmetic expressions, not actual answers. For example, given a question such as "How many eggs are there in a carton containing three dozen eggs?" you'd want the model to respond with “number_of_dozens × number_per_dozen = number_of_eggs,” instead of supplying just the answer (“36”).

You can use Grammar to enforce this constraint. To do so, write a prompt that explains that you want the model to generate an expression:

```
system_prompt = """
You are a math instructor. I will ask you a math question.
Respond with the mathematical expression that can be used to
solve the problem.
"""
```

This, however, is not enough. The model’s instruction-following capability is not perfect, and there is no guarantee that it will output mathematical expressions. It might simply provide the answer, or it might provide the reasoning required to generate the answer.

To constrain the model, write a grammar for simple arithmetic expressions:

```
grammar_str = """
root  ::= (expr "=" ws term "\n")+
expr  ::= term ([-+*/] term)*
term  ::= ident | num | "(" ws expr ")" ws
ident ::= [a-z] [a-z0-9_]* ws
num   ::= [0-9]+ ws
ws    ::= [ \t\n]*
    """
```

This grammar allows expressions that contain an equal sign that connects terms and expressions. The expression itself (`expr`) is one or more terms connected by -, +, *, or /. A term could be an identity (`num_dozen`), a number (12), or an expression surrounded by parentheses. The previous grammar also defines an identity (`ident`), a number (`num`), and a whitespace (`ws`).

Next, use this grammar to constrain the tokens allowed in the response:

```
 grammar = IncrementalGrammarConstraint(grammar_str, "root",
pipe.tokenizer)
 grammar_processor = GrammarConstrainedLogitsProcessor(grammar)
```

Then, pass the grammar processor to `pipe`:

```
results = pipe(input_message,
               max_new_tokens=256,
               do_sample=False,
               logits_processor=[grammar_processor])
```

Now, ask the model this question:

Bill has 3 apples and 2 oranges.

Mae has 2 apples and 4 oranges.

How many apples do Bill and Mae have in total?

This obtains the following response:

bill_apples + mae_apples = total_apples

3 + 2 = 5

You can check that the grammar truly does constrain the output by asking a question for which the right answer is not allowed by the grammar:

Bill has 3 apples and 2 oranges.

Mae has 2 apples and 4 oranges.

Do Bill and Mae have more apples than oranges?

Now, the model outputs this:

3 + 2 = 5

2 + 4 = 6

The correct answer is (3 + 2) > (2 + 4), but our grammar string doesn't allow >, so the model cannot output it. The constraint is therefore working properly.

The pipe separator

Suppose that you want the LLM to extract three pieces of information and output them, separated from one another by the pipe character (|). You can do that by using this prompt:

> You will be given a short paragraph about a book.
>
> Extract the author, title, and publication year of the book.
>
> Return the result as author | title | year
>
> If any piece of information is not found, fill the spot with NULL.

To constrain the model to produce precisely this format, use the following grammar string:

```
record ::= author separator title separator year
author ::= [a-zA-Z ]* | unk
title ::= [a-zA-Z ]* | unk
year ::= [1-2][0-9][0-9][0-9] | unk
unk ::= "NULL"
separator ::= "|"
```

This grammar allows names and titles to consist of lowercase and uppercase letters and spaces, but not non-English characters such as those with accent or diacritical marks. This can be important if your downstream processing code is a legacy system that expects 7-bit ASCII characters. Thus, say you pass in this paragraph:

> Love in the Time of Cholera (Spanish: El amor en los tiempos del cólera) is a novel written in Spanish by Colombian Nobel Prize–winning author Gabriel García Márquez and published in 1985.

Then, the model will output the author’s name with the accents removed:

> Gabriel Garcia Marquez | Love in the Time of Cholera |1985

Next, say you pass in this paragraph:

> The Tirukkural (Tamil: , lit. “sacred verses”) is a classic Tamil language text whose authorship is traditionally attributed to Valluvar, also known in full as Thiruvalluvar. The text has been dated variously from 300 BCE to the 5th century CE. The traditional accounts describe it as the last work of the third Sangam, but linguistic analysis suggests a later date of 450 to 500 CE and that it was composed after the Sangam period.

The model will then output this:

> Valluvar | The Tirukkural |NULL

The grammar allows author, title, and/or year to also take on the literal value `NULL`. With conflicting information about the year in the text, the LLM chooses `NULL` as the extracted value for the year. Also, although the paragraph includes the title in two forms (*Tirukkural* and), only one of them matches the grammar specification—and that’s the one that shows up in the result.

You can also use the Grammar pattern to ensure that generated content is valid, as long as the validation rules can be expressed in BNF.

JSON output format

Normally, this pattern requires you to define a grammar. However, because nearly all hosted foundational models support the ability to generate JSON, you have a more convenient approach available to output JSON. Here’s how you could extract the author, title, and year by using OpenAI and JSON mode:

```
def parse_book_info(paragraph: str) -> str:
    system_prompt = """
    You will be given a short paragraph about a book.
    Extract the author, title, and publication year of the book.
    Return the result as JSON with the keys author, title, and
year.
    If any piece of information is not found, fill the spot with
NULL
    """

    input_message = [
        {"role": "developer", "content": system_prompt},
```

```
        {"role": "user", "content": paragraph}
    ]

    response = client.chat.completions.create(
        model=MODEL_ID,
        messages=input_message,
        response_format={"type": "json_object"}
    )
    return response.choices[0].message.content
```

When you pass in the paragraph about *Love in the Time of Cholera*, the output JSON is this:

```
{
    "author": "Gabriel García Márquez",
    "title": "Love in the Time of Cholera",
    "year": 1985
}
```

When you pass in the paragraph about the Tamil classic text, the output JSON is this:

```
{
    "author": "Valluvar",
    "title": "The Tirukkural",
    "year": "NULL"
}
```

Even though these JSON fields are author, title, and year, this is not constrained and relies on having a high-capacity model like GPT-4o-mini. The JSON mode doesn't constrain whether the author's name contains accent marks—sometimes it will, sometimes it won't. Similarly, the JSON mode doesn't constrain the book's title to be only English characters, so you should guard against the extraction returning the Tamil name at least sometimes.

Extracting invoice information

Besides JSON, another common format that is supported out of the box is Pythonic data classes. So, you don't need to define a grammar string for these either.

If you want to consistently extract three pieces of information (purpose, amount, and currency) from emails requesting payment, you can define an `Invoice` class using Python:

```
from dataclasses import dataclass
from enum import Enum

class CurrencyEnum(str, Enum):
    USD = 'USD'
    UKP = 'UKP'
    INR = 'INR'
    EUR = 'EUR'

@dataclass
class Invoice:
    purpose: str
    amount: float
    currency: CurrencyEnum = CurrencyEnum.USD
```

Then, to parse invoice information from a paragraph of text, you can use the PydanticAI framework to get structured outputs in an LLM-agnostic way:

```
from pydantic_ai import Agent

def parse_invoice_info(paragraph: str) -> str:
    system_prompt = """
    You will be given a short snippet from an email that
represents an invoice.
    Extract the purpose and amount of the invoice.
    """

    agent = Agent(model,
                  result_type=Invoice,
                  system_prompt=system_prompt)

    response = agent.run_sync(paragraph)
    return response.output
```

In this framework, `response.output` returns a Python data object.

Next, pass in the input text:

> Requesting reimbursement for taxi ride to airport. I paid $32.30.

And in response, we get this:

```
Invoice(purpose='taxi ride to airport', amount=32.3,
        currency=<CurrencyEnum.USD: 'USD'>)
```

Because of the Grammar pattern, we are guaranteed to get an `Invoice` object.

> **DON'T BEG FOR COMPLIANCE**
>
> Begging an LLM to produce format in a specific form is an antipattern. Don't just hope that the LLM will comply with a prompt like this:
>
> > Please do not add any extra formatting or lengthy explanations. Just answer "YES" or "NO." Make sure to use all caps.
>
> Instead, use Grammar to ensure compliance:
>
> ```
> from typing import Literal
> agent = Agent(model,
>               result_type=Literal["YES", "NO"])
> ```

Considerations

Grammar is a way of specifying a set of constraints in the form of a metasyntax to ensure that a model response conforms to that metasyntax.

Variations

The canonical way to provide a metasyntax is to specify a BNF grammar for your constraints and then use a framework class that applies the grammar to perform Logits Masking. An easier way to specify your constraints is as a Python data (Pydantic) model using the corresponding capability in the foundational model's API.

The Pydantic approach has these benefits over BNF:

Ease of use

It’s easier to write a `dataclass` consisting of a few classes and attributes than it is to specify rules in BNF.

Latency

BNF constraints are applied via Logits Masking by the model framework, and they’re therefore applied client-side. On the other hand, Pydantic constraints are applied by the model provider and are therefore applied server-side in the case of APIs such as GPT-4, Gemini, and Claude. Therefore, the Pydantic approach reduces the number of network calls.

Model support

Using BNF requires access to `logprobs`, and as mentioned in the “Caveats” section of “Logits Masking (Pattern 1),” support for this is not universal. On the other hand, every modern model supports Grammar constraints via Python `dataclass`.

The BNF approach is nevertheless more flexible than the Pydantic approach, and you need to use it if your style rules involve more than just data formats. For example, it’s possible to express validation rules as BNF, whereas any validation beyond `Enum` is hard to do in Pydantic. To illustrate, here’s a potential BNF grammar that a company might employ if it accepts three types of United States (US) credit cards:

```
<credit_card_number> ::= <visa_number> | <mc_number> |
<amex_number>
<visa_number> ::= "4" <digit>{12,15}
<mc_number> ::= ("51".."55" <digit>{14}) | ("2221".."2720"
<digit>{12})
<amex_number> ::= "34" <digit>{13} | "37" <digit>{13}
<digit> ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" |
"9"
```

It’s considerably harder to do this in Pydantic since you would have to write validator logic—no longer would this be just a simple `dataclass`:

```
class CreditCard(BaseModel):
    number: str

    @field_validator('number')
    @classmethod
    def validate_number(cls, value: str) -> str:
        value = value.replace(" ", "").replace("-", "")
        if not value.isdigit():
            raise ValueError("Credit card number must contain
only digits.")

        length = len(value)
        first_digit = value[0]
        first_two_digits = value[:2]
        first_four_digits = value[:4]
        first_six_digits = value[:6]

        # Visa
        if first_digit == '4' and (length == 13 or length == 16):
            return value

        # Mastercard
        if (first_two_digits in ('51', '52', '53', '54', '55') or
            2221 <= int(first_four_digits) <= 2720) and length ==
16:
            return value

        # American Express
        if first_two_digits in ('34', '37') and length == 15:
            return value

        raise ValueError("Invalid credit card number format.")
```

Although we’ve described the schema approach in the form of a Python `dataclass`, the capability to represent a schema using data classes is not restricted to Python. For example, JavaScript developers can take advantage of Ollama’s and OpenAI’s support for schema specification by using Zod.

Alternatives

Many of the alternatives to Logits Masking—such as Style Transfer (Pattern 3), using a more powerful model, try and try again, and Reflection (Pattern 18)—are also alternatives to Grammar for the same reasons. Rather than repeating the discussion here, we refer you to the "Alternatives" section of "Logits Masking (Pattern 1)."

Logits Masking is itself an alternative to Grammar. Consider Logits Masking rather than Grammar if any of the following apply:

- The rules you want to apply can't be represented as a `dataclass`. Any rule that consists of logic rather than mere representation fits this profile. For example, you may want to mask out names of competitor products, but only in specific situations.
- The BNF grammar is very complex and hard to debug.
- The rules are dynamic and depend on the content. For example, you may want to mask out the name of a competitor product when the content is talking about a new launch, but not if the content relates to an in-market product.
- The rules need to be fetched from a database or rules engine. For example, rules on what to mask may vary by client.
- Masking depends on user input, such as in the autocomplete use case that was discussed in "Logits Masking (Pattern 1)."
- The rules involve invoking an external tool or API.

In these situations, consider implementing a logits processor and writing code to determine whether to mask an input token sequence.

Caveats

If the model does not output any tokens that meet the grammar constraints, generation will fail. This failure can manifest itself in the following ways:

Endless whitespace

Sometimes, the failure will take the form of an endless loop of whitespace because whitespace is often allowed by the grammar and is typically one of the tokens that's always available in the candidate space.

Increased refusals

At other times, the failure will take the form of increased refusals, especially if you ask the LLM to produce nested fields or overly long structures. This is because the likelihood of arriving at a point where none of the candidate output tokens fits the grammar increases with increased length and complexity.

Inaccurate results

Only tokens that are allowed by the grammar will be included in the output. If your grammar is too restrictive, you might get inaccurate results. Therefore, it can be helpful to give the model an option that allows it to escape the restrictive grammar. For example, to specify that a field should be a float but also allow the model to emit the `"Unknown"` string, you can define the field as follows:

```
currency_rate: float | Literal["Unknown"]
```

The Grammar pattern has a couple of alternate names. Because data structures are a common format for expressing the grammar specification, this pattern is also sometimes called *structured outputs*. Also, because it works by constraining the logits that are possible, it's sometimes called *constrained decoding*. However, be careful—support for "structured outputs" by a model or framework does not necessarily mean that the Grammar pattern is being employed. For example, at the time we are writing this (June 2025), LangGraph implements support for structured outputs by using an additional LLM call to postprocess the original response into the desired format. Such a postprocessing approach is more wasteful, more expensive, and less reliable than the Grammar pattern that ensures compliance through the manipulation of logits.

References

Early approaches that incorporated the grammar into the prompt, like the one in Wang et al.'s 2024 paper "Grammar Prompting for Domain-Specific Language Generation with Large Language Models", have turned out to be error-prone. Using grammar masking and constrained decoding was first detailed in a 2024 paper by Netz, Reimar, and Rumpe, but the idea was implemented earlier, in 2023, by Rickard for regular expressions and Jones for BNF grammars. Grammar need not be strict—in 2025, a group of MIT researchers assigned weights to structured data continuations using Monte Carlo simulations and demonstrated that this makes AI-generated code more accurate.

The sieves library relies on Grammar to implement NLP tasks reliably. Grammar is now a cornerstone of agent frameworks and capabilities such as Fireworks AI, Databricks, MCP, and Dify. Grammar is supported by GPT-4 and Gemini.[1]

Pattern 3: Style Transfer

The Style Transfer pattern allows you to teach a GenAI model to convert content in a readily available form into content in some desired style. You do this by showing the model example input-and-output pairs that illustrate the conversion. There are two variants: *few-shot learning*, in which you have just a few examples and you put them into the prompt context, and *model fine-tuning*, in which you adapt a pretrained model to do the conversion by using a large dataset of example input-and-output pairs.

In the first two pattern sections of this chapter, we discussed how to control the style of the model's generation through either dynamic logic (via Pattern 1, Logits Masking) or structured rules (via Pattern 2, Grammar). In many situations, it's difficult to express nuances through rules, so you can use Style Transfer to show the model some examples and let it extrapolate from those examples to unseen situations.

Problem

Suppose you want a GenAI model to generate content that mimics a specific tone and style of texts. Let's assume that your situation satisfies these three criteria (also see Figure 2-5):

Available content

> The content that you want is readily available, but it's just not in the tone or style you want to use. Perhaps the content is available in academic research papers, but you want to use parts of the content (perhaps the methods and results) in marketing brochures targeted at nontechnical executives.

Nuanced style

> It's difficult to express the nuances of what you want in a few rules. The characteristics of the desired style might be very subtle, and humans may find it hard to express what these characteristics are. However, humans will often recognize the right style. ("I know it when I see it.") For example, it's hard to express what vocabulary is allowed in marketing brochures. Can we use the term *fine-tuning*? How about *reinforcement learning*?

Example conversions

> You *do* have examples in which experts took readily available content and converted it into content in the style you want to use. For example, you may have a few handcrafted marketing brochures that were written based on research articles.

The Style Transfer pattern applies when your situation satisfies these three criteria.