Patterns of Application Development Using AI

Obie Fernandez

Foreword by Gregor Hohpe

Leanpub

Patterns of Application Development Using AI

Obie Fernandez

This book is available at
http://leanpub.com/patterns-of-application-development-using-ai

This version was published on 2024-12-23

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Obie Fernandez by spreading the word about this book on Twitter!

The suggested hashtag for this book is #poaduai.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#poaduai

To my badass queen, my muse, my light and love, Victoria

Also By Obie Fernandez

The Rails 8 Way

The Rails 7 Way

XML The Rails Way

Serverless

El Libro Principiante de Node

The Lean Enterprise

Contents

Foreword by Gregor Hohpe

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Preface

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

About the Book

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

About the Code Examples

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

What I Don't Cover

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Who This Book Is For

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Building a Common Vocabulary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Getting Involved

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Acknowledgments

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

What's with the illustrations?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

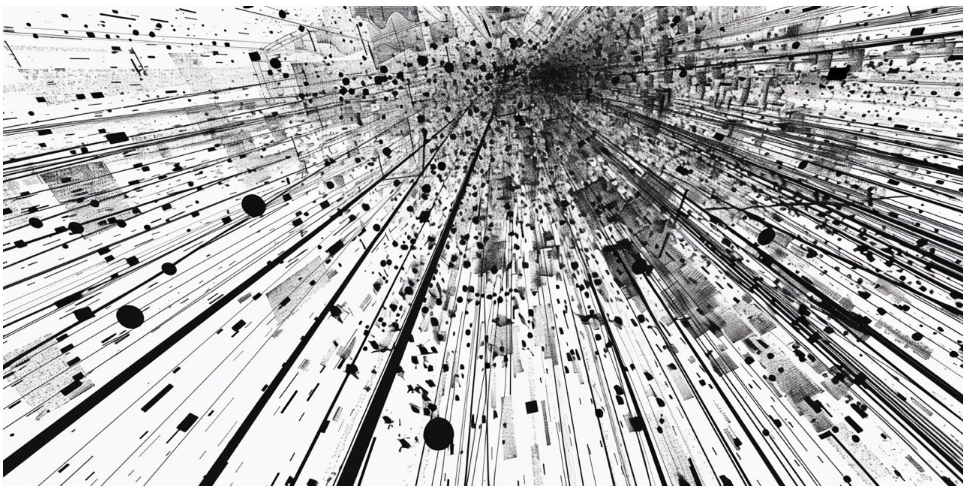
About Lean Publishing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

About The Author

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Introduction



If you're eager to start integrating AI Large Language Models (LLMs) into your programming projects, feel free to dive right into the patterns and code examples presented in later chapters. However, to fully appreciate the power and potential of these patterns, it's worth taking a moment to understand the broader context and the cohesive approach they represent.

The patterns are not merely a collection of isolated techniques but rather a unified framework for integrating AI into your applications. I use Ruby on Rails , but these patterns should work in pretty much any other programming environment. They address a wide range of concerns, from data management and performance optimization to user experience and security, providing a comprehensive toolkit for enhancing traditional programming practices with the capabilities of AI.

Each category of patterns tackles a specific challenge or opportunity that arises when incorporating AI components into your application. By understanding the relationships and synergies between these patterns, you can make informed decisions about where and how to apply AI most effectively.

Patterns are never prescriptive solutions and should not be treated as such. They are meant to be adaptable building blocks that should be tailored to the unique requirements and constraints of your own unique application. The successful application of these patterns (like any others in the software field) relies on a deep understanding of the problem domain, user needs, and the overall technical architecture of your project.

Thoughts on Software Architecture

I started programming in the 1980s and was involved in the hacker scene, and never lost my hacker mindset, even after becoming a professional software developer. Since the start, I always had a healthy skepticism about what value software architects in their ivory towers actually brought to the table.

One of the reasons that I'm personally so excited about the changes brought forth by this powerful new wave of AI technology is its impact on what we consider *software architecture* decisions. It challenges traditional notions of what constitutes the "correct" way to design and implement our software projects. It also challenges whether architecture can still be thought of primarily as *the parts of a system that are hard to change*, since AI enhancement is making it easier than ever to change any part of your project, at any time.

Perhaps we're entering the peak years of the "post-modern" approach to software engineering. In this context, post-modern refers to a fundamental shift away from traditional paradigms, where developers were responsible for writing and maintaining every line of code. Instead, it embraces the idea of delegating tasks, such as data manipulation, complex algorithms, and even entire chunks of application logic, to 3rd-party libraries and external APIs. This post-modern shift represents a significant departure from the conventional wisdom of building applications from the ground up, and it challenges developers to rethink their role in the development process.

I've always believed that good programmers only write the code that is absolutely necessary to write, based on the teachings of Larry Wall and other hacker luminaries like him. By minimizing the amount of code written, we can move faster, reduce the surface area for bugs, simplify maintenance, and improve the overall reliability of their applications. Less code allows us to focus on the core business logic and user experience, while delegating other work to other services.

Now that AI-powered systems can handle tasks that were previously the exclusive domain of human-written code, we should be able to be even more productive and agile, with a greater focus than ever on creating business value and user experience.

Of course there are trade-offs of delegating huge parts of your project to AI systems, such as the potential loss of control, and the need for robust monitoring and feedback mechanisms. That's why it requires a new set of skills and knowledge, including at least some fundamental understanding of how AI works.

What is a Large Language Model?

Large Language Models (LLMs) are a type of artificial intelligence model that have gained significant attention in recent years, ever since the launch of GPT-3 by OpenAI in 2020. LLMs are designed to process, understand, and generate human language with remarkable accuracy and fluency. In this section, we'll take a brief look at how LLMs work and why they are well-suited for building intelligent system components.

At their core, LLMs are based on deep learning algorithms, specifically neural networks . These networks are composed of interconnected nodes, or neurons, that process and transmit information. The architecture of choice for LLMs is often the Transformer model, which has proven to be highly effective in handling sequential data like text.

Transformer models are based on the attention mechanism and are primarily used for tasks involving sequential data, like natural language processing. Transformers process input data all at once rather than sequentially, which allows them to capture long-range dependencies more effectively. They have layers of attention mechanisms that help the model focus on different parts of the input data to understand context and relationships.

The training process for LLMs involves exposing the model to vast amounts of textual data, such as books, articles, websites, and code repositories. During training, the model learns to recognize patterns, relationships, and structures within the text. It captures the statistical properties of the language, such as grammar rules , word associations, and contextual meanings.

One of the key techniques used in training LLMs is unsupervised learning . This means that the model learns from the data without explicit labeling or guidance. It discovers patterns and representations on its own by analyzing the co-occurrence of words and phrases in the training data. This allows LLMs to develop a deep understanding of language and its intricacies.

Another important aspect of LLMs is their ability to handle *context*. When processing a piece of text, LLMs consider not only the individual words but also the surrounding context. They take into account the previous words, sentences, and even paragraphs to understand the meaning and intent of the text. This contextual understanding enables LLMs to generate coherent and relevant responses. One of the main ways that we evaluate the capabilities of a given LLM model is by considering the size of the context they can consider in order to generate responses.

Once trained, LLMs can be used for a wide range of language-related tasks . They can generate human-like text, answer questions, summarize documents, translate languages, and even write code. The versatility of LLMs makes them valuable for building intelligent system components that can interact with users, process and analyze text data, and generate meaningful outputs.

By incorporating LLMs into the application architecture, you can create AI components that understand and process user input, generate dynamic content, and provide intelligent recommendations or actions. But working with LLMs requires careful consideration of resource requirements and performance trade-offs . LLMs are computationally intensive and may require significant processing power and memory (in other words, money) to operate. Most of us will need to assess the cost implications of integrating LLMs into our applications and act accordingly.

Understanding Inference

Inference refers to the process by which a model generates predictions or outputs based on new, unseen data. It is the phase where the trained model is used to make decisions or generate text, images, or other content in response to user inputs.

During the training phase, an AI model learns from a large dataset by adjusting its parameters to minimize the error in its predictions . Once trained, the model can apply what it has learned to new data. Inference is how the model uses its learned patterns and knowledge to generate outputs.

For LLMs, inference involves taking a prompt or input text and producing a coherent and contextually relevant response, as a stream of *tokens* (which we'll talk about soon). This could be answering a question, completing a sentence, generating a story, or translating text, among many other tasks.

In contrast to the way that you and I think, an AI model's "thinking" via inference happens in all in one stateless operation. That is, it's thinking is limited to its generation process. It literally has to think out loud, as if I asked you a question and only accepted a response from you in "stream of consciousness" style.

Large Language Models Come in Many Sizes and Flavors

While practically all popular large language models (LLMs) are based on the same core transformer architecture and trained on huge text datasets, they come in a variety of sizes and are fine-tuned for different purposes. The size of an LLM, measured by the number of parameters in its neural network , has a big impact on its capabilities. Larger models with more parameters, like GPT-4 , which is rumored to boast 1 to 2 trillion parameters, are generally more knowledgeable and capable than smaller models. However, larger models also require much more computing power to run, which translates to higher expense when you use them via API calls.

To make LLMs more practical and tailored for specific use cases, the base models are often fine-tuned on more targeted datasets. For example, an LLM may be trained on a large corpus of dialog to specialize it for conversational AI . Others are trained on code to imbue them with programming knowledge. There are even models that are specially trained for roleplay-style interactions with users !

Retrieval vs Generative Models

In the world of large language models (LLMs), there are two main approaches to generating responses: retrieval-based models and generative models. Each approach has its own strengths and weaknesses, and understanding the differences between them can help you choose the right model for your specific use case.

Retrieval-based Models

Retrieval-based models , also known as information retrieval models , generate responses by searching through a large database of pre-existing text and selecting the most relevant passages based on the input query. These models don't generate new text from scratch but rather stitch together excerpts from the database to form a coherent response.

One of the main advantages of retrieval-based models is their ability to provide factually accurate and up-to-date information. Since they rely on a database of curated text, they can pull relevant information from reliable sources and present it to the user. This makes them well-suited for applications that require precise, factual answers, such as question-answering systems or knowledge bases .

However, retrieval-based models have some limitations. They are only as good as the database they are searching through, so the quality and coverage of the database directly impact the model's performance. Additionally, these models may struggle with generating coherent and natural-sounding responses, as they are limited to the text available in the database.

We don't cover usage of pure retrieval models in this book.

Generative Models

Generative models, on the other hand, create new text from scratch based on the patterns and relationships they learned during training. These models use their understanding of language to generate novel responses that are tailored to the input prompt.

The main strength of generative models is their ability to produce creative, coherent, and contextually relevant text. They can engage in open-ended conversations, generate stories, and even write code. This makes them ideal for applications that require more open-ended and dynamic interactions, such as chatbots, content creation, and creative writing assistants.

However, generative models can sometimes produce inconsistent or factually incorrect information, as they rely on the patterns learned during training rather than a curated database of facts. They may also be more prone to biases and hallucinations, generating text that is plausible but not necessarily true.

Examples of generative LLMs include OpenAI's GPT series (GPT-3, GPT-4) and Anthropic's Claude .

Hybrid Models

Several commercially available LLMs combine both retrieval and generative approaches in a hybrid model. These models use retrieval techniques to find relevant information from a database and then use generative techniques to synthesize that information into a coherent response.

Hybrid models aim to combine the factual accuracy of retrieval-based models with the natural language generation capabilities of generative models. They can provide more reliable and up-to-date information while still maintaining the ability to engage in open-ended conversations.

When choosing between retrieval-based and generative models, you should consider the specific requirements of your application. If the primary goal is to provide accurate, factual information, a retrieval-based model may be the best choice. If the application requires more open-ended and creative interactions, a generative model may be more suitable. Hybrid models offer a balance between the two approaches and can be a good choice for applications that require both factual accuracy and natural language generation.

Ultimately, the choice between retrieval-based and generative models depends on the specific use case and the trade-offs between accuracy, creativity, and flexibility. By understanding the strengths and limitations of each approach, you can make informed decisions.

Instruction Tuning

Instruction tuning is a subset of conversational training that uses human-written prompts and responses to make LLMs better at understanding and following instructions. Here are some simple examples to illustrate how instruction tuning on human-written prompts and responses makes LLMs better at understanding and following instructions:

Example 1: Customer Support

Instruction: "Provide a response to a customer asking for a refund."

Human-Written Prompt: "I would like to request a refund for my recent purchase. The product did not meet my expectations."

Human-Written Response: "I'm sorry to hear that the product did not meet your expectations. To process your refund, please provide your order number and the reason for the return. We will review your request and get back to you within 2-3 business days."

Example 2: Educational Assistance

Instruction: "Explain the concept of photosynthesis to a 10-year-old."

Human-Written Prompt: "Can you explain how plants make their food?"

Human-Written Response: "Sure! Plants make their food through a process called photosynthesis. They use sunlight, water, and carbon dioxide to create food and oxygen. The leaves of the plant have a special part called chlorophyll that helps them capture sunlight."

Example 3: Personal Assistant

> Instruction: “Schedule a meeting for next Tuesday at 3 PM.”
>
> Human-Written Prompt: “Can you set up a meeting for Tuesday at 3 PM?”
>
> Human-Written Response: “Sure, I have scheduled a meeting for you next Tuesday at 3 PM. Is there anything else you need?”

The result is a diverse ecosystem of LLMs in different sizes and with different specialties. Smaller models in the 1-7 billion parameter range provide good general language capabilities while being more efficient to run.

- Mistral 7B
- Llama 3 8B
- Gemma 7B

Mid-size models around 30-70 billion parameters offer stronger reasoning and instruction-following abilities.

- Llama 3 70B
- Qwen2 70B
- Mixtral 8x22B

When choosing an LLM to incorporate into an application, you must balance the model’s capabilities against practical factors like cost, latency, context length, and content filtering. Smaller, instruction-tuned models are often the best choice for simpler language tasks, while the largest models may be needed for complex reasoning or analysis. The model’s training data is also an important consideration, as it determines the model's knowledge cutoff date.

Certain models, like some from Perplexity are connected to realtime information sources, so that they effectively have no cutoff date. When you ask them questions, they're able to independently decide to do web searches and fetch arbitrary web pages in order to generate an answer.

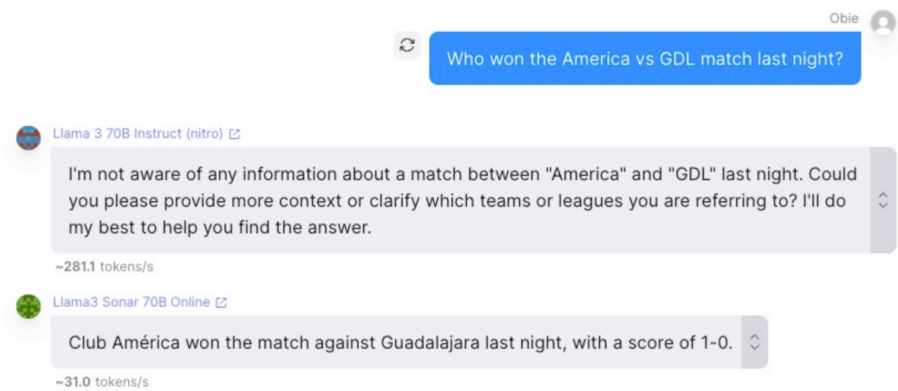


Figure 1. Llama3 with and without online access

Ultimately, there is no one-size-fits-all LLM. Understanding the variations in model size, architecture, and training is key to selecting the right model for a given use case. Experimenting with different models is the only practical way to reveal which ones provide the best performance for the task at hand.

Tokenization: Breaking Text into Pieces

Before a large language model can process text, that text needs to be broken down into smaller units called *tokens* . Tokens can be individual words, parts of words, or even single characters. The process of splitting text into tokens is known as tokenization , and it's a crucial step in preparing data for a language model.

```
The process of splitting text into tokens is known as tokenization, and
 it's a crucial step in preparing data for a language model.
```

Figure 2. This sentence contains 27 tokens

Different LLMs use different tokenization strategies, which can have a significant impact on the model's performance and capabilities. Some common tokenizers used by LLMs include:

- **GPT (Byte Pair Encoding):** GPT tokenizers use a technique called byte pair encoding (BPE) to break text into subword units. BPE iteratively merges the most frequent pairs of bytes in a text corpus, forming a vocabulary of subword tokens. This allows the tokenizer to handle rare and novel words by breaking them down into more common subword pieces. GPT tokenizers are used by models like GPT-3 and GPT-4 .

- **Llama (SentencePiece):** Llama tokenizers use the SentencePiece library, which is an unsupervised text tokenizer and detokenizer. SentencePiece treats the input text as a sequence of Unicode characters and learns a subword vocabulary based on a training corpus. It can handle any language that can be encoded in Unicode, making it well-suited for multilingual models. Llama tokenizers are used by models like Meta's Llama and Alpaca .

- **SentencePiece (Unigram):** SentencePiece tokenizers can also use a different algorithm called Unigram, which is based on a subword regularization technique. Unigram tokenization determines the optimal subword vocabulary based on a unigram language model, which assigns probabilities to individual subword units. This approach can produce more semantically meaningful subwords compared to BPE. SentencePiece with Unigram is used by models like Google's T5 and BERT .

- **Google Gemini (Multimodal Tokenization):** Google Gemini uses a tokenization scheme designed to handle various data types, including text, images, audio, videos, and code. This multimodal capability allows Gemini to process and integrate different forms of information. Notably, Google Gemini 1.5 Pro has a context window that can handle millions of tokens, much larger than previous models. This extensive context window enables the model to process a larger context, potentially leading to more accurate responses. However, it's important to note that Gemini's tokenization scheme is much closer to one token per character than other models. This means the actual cost of using Gemini models can be significantly higher than expected if you're accustomed to using models like GPT, as Google's pricing is based on characters rather than tokens.

The choice of tokenizer affects several aspects of an LLM, including:

- **Vocabulary size:** The tokenizer determines the size of the model's vocabulary, which is the set of unique tokens it recognizes. A larger, more fine-grained vocabulary can help the model handle a wider range of words and phrases and even become multi-modal (capable of understanding and generating more than just text), but it also increases the model's memory requirements and computational complexity.

- **Handling of rare and unknown words:** Tokenizers that use subword units, like BPE and SentencePiece, can break down rare and unknown words into more common subword pieces. This allows the model to make educated guesses about the meaning of words it hasn't seen before, based on the subwords they contain.
- **Multilingual support:** Tokenizers like SentencePiece, which can handle any Unicode-encodable language , are well-suited for multilingual models that need to process text in multiple languages.

When choosing an LLM for a particular application, it's important to consider the tokenizer it uses and how well it aligns with the specific language processing needs of the task at hand. The tokenizer can have a significant impact on the model's ability to handle domain-specific terminology, rare words, and multilingual text.

Context Size: How Much Information Can a Language Model Use During Inference?

When discussing language models, context size refers to the amount of text that a model can consider when processing or generating its responses. It's essentially a measure of how much information the model can "remember" and use to inform its outputs (expressed in tokens). The context size of a language model can have a significant impact on its capabilities and the types of tasks it can perform effectively.

What is Context Size?

In technical terms, context size is determined by the number of tokens (words or word pieces) that a language model can process in a single input sequence. This is often referred to as the model's "attention span" or "context window." The larger the context size, the more text the model can consider at once when generating a response or performing a task.

Different language models have varying context sizes, ranging from a few hundred tokens to millions of tokens. For reference, a typical paragraph of text might contain around 100-150 tokens, while an entire book could contain tens or hundreds of thousands of tokens.

> There's even work on efficient methods to scale Transformer-based Large Language Models (LLMs) to infinitely long inputs with bounded memory and computation.

Why is Context Size Important?

The context size of a language model has a significant impact on its ability to understand and generate coherent, contextually relevant text. Here are some key reasons why context size matters:

1. **Understanding long-form content:** Models with larger context sizes can better comprehend and analyze longer texts, such as articles, reports, or even entire books. This is crucial for tasks like document summarization, question answering, and content analysis.

2. **Maintaining coherence:** A larger context window allows the model to maintain coherence and consistency across longer stretches of

output. This is important for tasks like story generation, dialogue systems, and content creation, where maintaining a consistent narrative or topic is essential. It's also absolutely crucial when using LLMs for generating or transforming structured data.

3. **Capturing long-range dependencies:** Some language tasks require understanding relationships between words or phrases that are far apart in a text. Models with larger context sizes are better equipped to capture these long-range dependencies, which can be important for tasks like sentiment analysis , translation , and language understanding.
4. **Handling complex instructions:** In applications where language models are used to follow complex, multi-step instructions, a larger context size allows the model to consider the entire set of instructions when generating a response, rather than just the most recent few words.

Examples of Language Models with Different Context Sizes

Here are a few examples of language models with different context sizes:

- OpenAI GPT-3.5 Turbo: 4,095 tokens
- Mistral 7B Instruct: 32,768 tokens
- Anthropic Claude v1: 100,000 tokens
- OpenAI GPT-4 Turbo: 128,000 tokens
- Anthropic Claude v2: 200,000 tokens
- Google Gemini Pro 1.5: 2.8M tokens

As you can see, there is a wide range of context sizes among these models, from around 4,000 tokens for the OpenAI GPT-3.5 Turbo model to 200,000 tokens for the Anthropic Claude v2 model. Some models, like Google's PaLM 2 and OpenAI's GPT-4 , offer different variants with larger context sizes (e.g., "32k" versions), which can handle even longer input sequences. And at the moment (April 2024) Google Gemini Pro is boasting nearly 3 million tokens!

It's worth noting that the context size can vary depending on the specific implementation and version of a particular model. For example, the original OpenAI GPT-4 model has a context size of 8,191 tokens, while the later GPT-4 variants such as Turbo and 4o have a much larger context size of 128,000 tokens.

> Sam Altman has compared current context limitations to the kilobytes of working memory that personal computer programmers had to deal with in the 80s, and said that in the near future we will be able to fit "all of your personal data" into the context of a large language model .

Choosing the Right Context Size

When selecting a language model for a particular application, it's important to consider the context size requirements of the task at hand. For tasks that involve short, isolated pieces of text, like sentiment analysis or simple question answering, a smaller context size may be sufficient. However, for tasks that require understanding and generating longer, more complex texts, a larger context size will likely be necessary.

It's worth noting that larger context sizes often come with increased computational costs and slower processing times, as the model needs to consider more information when generating a response. As such, you must strike a balance between context size and performance when choosing a language model for your application.

> Why not just choose the model with the largest context size and stuff it with as much information as possible? Well, besides performance factors the other main consideration is cost. In March 2024 a *single* prompt-response cycle using Google Gemini Pro 1.5 with a full context will cost you almost $8 (USD). If you have a use case that justifies that expense, more power to you! But for most applications, it's just too expensive by orders of magnitude.

Finding Needles in Haystacks

The concept of finding a needle in a haystack has long been a metaphor for the challenges of retrieval in large datasets. In the realm of LLMs, we tweak this analogy a bit. Imagine we're not just looking for a single fact buried within a vast text (like a full anthology of Paul Graham essays), but multiple facts scattered throughout. This scenario is more akin to finding several needles in a sprawling field, not just a single haystack. Here's the kicker: not only do we need to locate these needles, but we also have to weave them into a coherent thread.

When tasked with retrieving and reasoning about multiple facts embedded in long contexts, LLMs face a dual challenge. First, there's the straightforward issue of retrieval accuracy—it naturally dips as the number of facts increases. This is expected; after all, keeping track of multiple details across a sprawling text taxes even the most sophisticated models.

Second, and perhaps more critically, is the challenge of reasoning with these facts. It's one thing to pick out facts; it's quite another to synthesize them into a coherent narrative or answer. This is where the real test comes in. The performance of LLMs in reasoning tasks tends to degrade further than in simple retrieval tasks. This degradation isn't just about volume; it's about the intricate dance of context, relevance, and inference.

Why does this happen? Well, consider the dynamics of memory and attention in human cognition, which are mirrored to an extent in LLMs. When processing large amounts of information, LLMs, like humans, can lose track of earlier details as they absorb new ones. This is especially true in models that are not explicitly designed to prioritize or revisit earlier segments of text automatically.

Moreover, the ability of an LLM to weave these retrieved facts into a coherent response is akin to narrative building . This requires not just a retrieval of information but a deep understanding and contextual placement, which remains a stiff challenge for current AI.

So, what does this mean for us as developers and integrators of these technologies? We need to be acutely aware of these limitations when designing systems that rely on LLMs to handle complex, long-form tasks. Understanding that performance might degrade under certain conditions helps us set realistic expectations and engineer better fallback mechanisms or supplementary strategies.

Modalities: Beyond Text

While the majority of language models today are focused on processing and generating text, there is a growing trend towards multimodal models that can natively input and output multiple types of data, such as images, audio, and video. These multimodal models open up new possibilities for

AI-powered applications that can understand and generate content across different modalities.

What are Modalities?

In the context of language models, modalities refer to the different types of data that a model can process and generate. The most common modality is text, which includes written language in various forms like books, articles, websites, and social media posts. However, there are several other modalities that are increasingly being incorporated into language models:

- **Images:** Visual data such as photographs, illustrations, and diagrams.
- **Audio:** Sound data such as speech, music, and environmental sounds.
- **Video:** Moving visual data, often accompanied by audio, such as video clips and movies.

Each modality presents unique challenges and opportunities for language models. For example, images require the model to understand visual concepts and relationships, while audio requires the model to process and generate speech and other sounds.

Multimodal Language Models

Multimodal language models are designed to handle multiple modalities within a single model. These models typically have specialized components or layers that can both understand inputs and generate output data in different modalities. Some notable examples of multimodal language models include:

- **OpenAI's GPT-4o:** GPT-4o is a large language model that natively understands and processes speech audio in addition to text. This capability allows GPT-4o to perform tasks such as transcribing spoken language, generating text from audio inputs, and providing responses based on spoken queries.

- **OpenAI's GPT-4 with visual input:** GPT-4 is a large language model that can process both text and images. When given an image as input, GPT-4 can analyze the contents of the image and generate text that describes or responds to the visual information.
- **Google's Gemini:** Gemini is a multimodal model that can handle text, images, and video. It uses a unified architecture that allows for cross-modal understanding and generation, enabling tasks like image captioning, video summarization, and visual question answering.

- **DALL-E and Stable Diffusion:** While not language models in the traditional sense, these models demonstrate the power of multimodal AI by generating images from textual descriptions. They showcase the potential for models that can translate between different modalities.

Benefits and Applications of Multimodal Models

Multimodal language models offer several benefits and enable a wide range of applications, including:

- **Enhanced understanding:** By processing information from multiple modalities, these models can gain a more comprehensive understanding of the world, similar to how humans learn from various sensory inputs.
- **Cross-modal generation:** Multimodal models can generate content in one modality based on input from another, such as creating an image from a text description or generating a video summary from a written article.

- **Accessibility:** Multimodal models can make information more accessible by translating between modalities, such as generating text descriptions of images for visually impaired users or creating audio versions of written content.
- **Creative applications:** Multimodal models can be used for creative tasks like generating art, music, or videos based on textual prompts, opening up new possibilities for artists and content creators.

As multimodal language models continue to advance, they will likely play an increasingly important role in the development of AI-powered applications that can understand and generate content across multiple modalities. This will enable more natural and intuitive interactions between humans and AI systems, as well as unlock new possibilities for creative expression and knowledge dissemination.

Provider Ecosystems

When it comes to incorporating large language models (LLMs) into applications, you have a growing range of options to choose from. Each major LLM provider, such as OpenAI , Anthropic , Google , and Cohere , offers its own ecosystem of models, APIs, and tools. Choosing the right provider involves considering various factors, including pricing, performance, content filtering, data privacy, and customization options.

OpenAI

OpenAI is one of the most well-known providers of LLMs, with its GPT series (GPT-3, GPT-4) being widely used in various applications. OpenAI offers a user-friendly API that allows you to easily integrate their models into applications. They provide a range of models with different capabilities and price points, from the entry-level Ada model to the powerful Davinci model.

OpenAI's ecosystem also includes tools like the OpenAI Playground, which allows you to experiment with prompts and fine-tune models for specific use cases. They offer content filtering options to help prevent the generation of inappropriate or harmful content.

When using OpenAI's models directly, I rely on Alex Rudall 's ruby-openai library.

Anthropic

Anthropic is another major player in the LLM space, with their Claude models gaining popularity for strong performance and ethical considerations. Anthropic focuses on developing safe and responsible AI systems, with a strong emphasis on content filtering and avoiding harmful outputs.

Anthropic's ecosystem includes the Claude API, which allows you to integrate the model into their applications, as well as tools for prompt engineering and fine-tuning. They also offer the Claude Instant model, which incorporates web search capabilities for more up-to-date and factual responses.

When using Anthropic's models directly, I rely on Alex Rudall's anthrophic library.

Google

Google has developed several powerful LLMs, including Gemini, BERT , T5 , and PaLM . These models are known for their strong performance on a wide range of natural language processing tasks. Google's ecosystem includes the TensorFlow and Keras libraries, which provide tools and frameworks for building and training machine learning models.

Google also offers a Cloud AI Platform , which allows you to easily deploy and scale their models in the cloud. They provide a range of pre-trained models and APIs for tasks like sentiment analysis, entity recognition, and translation.

Meta

Meta , formerly known as Facebook , is deeply invested in the development of large language models, highlighted by its release of models like LLaMA and OPT . These models stand out for their strong performance in diverse language tasks and are made available largely through open-source channels, supporting Meta's commitment to research and community collaboration.

Meta's ecosystem is primarily built around PyTorch , an open-source machine learning library favored for its dynamic computational capabilities and flexibility, facilitating innovative AI research and development.

In addition to their technical offerings, Meta places a strong emphasis on ethical AI development. They implement robust content filtering and focus on reducing biases, aligning with their broader goals of safety and responsibility in AI applications.

Cohere

Cohere is a newer entrant in the LLM space, focusing on making LLMs more accessible and easier to use than competitors. Their ecosystem includes the Cohere API , which provides access to a range of pre-trained models for tasks like text generation, classification, and summarization.

Cohere also offers tools for prompt engineering, fine-tuning, and content filtering. They emphasize data privacy and security, with features like encrypted data storage and access controls.

Ollama

Ollama is a self-hosted platform that allows users to manage and deploy various large language models (LLMs) locally on their machines, giving them complete control over their AI models without relying on external cloud services. This setup is ideal for those who prioritize data privacy and wish to handle their AI operations in-house.

The platform supports a range of models, including versions of Llama, Phi, Gemma, and Mistral , which vary in size and computational requirements. Ollama makes it easy to download and run these models directly from the command line using simple commands like `ollama run <model_name>`, and it's designed to work across different operating systems including macOS, Linux, and Windows.

For developers looking to integrate open-source models into their applications without using a remote API, Ollama offers a CLI for managing model lifecycles similar to container management tools . It also supports custom configurations and prompts, allowing for a high degree of customization to tailor the models to specific needs or use cases.

Ollama is particularly suited for tech-savvy users and developers due to its command-line interface and the flexibility it offers in managing and deploying AI models. This makes it a powerful tool for businesses and individuals who require robust AI capabilities without compromising on security and control.

Multi-Model Platforms

Additionally, there are providers that host a wide variety of open-source models, such as Together.ai and Groq . These platforms offer flexibility and customization, allowing you to run and, in some cases, even fine-tune open-source models according to your specific needs. For example, Together.ai provides access to a range of open-source LLMs, enabling users to experiment with different models and configurations. Groq focuses on delivering ultra high-performance completion that at the time of this book's writing seems almost magical .

Choosing an LLM Provider

When choosing an LLM provider, you should consider factors like:

- **Pricing:** Different providers offer different pricing models, ranging from pay-per-use to subscription-based plans. It's important to consider the expected usage and budget when selecting a provider.
- **Performance:** The performance of LLMs can vary significantly between providers, so it's important to benchmark and test models on specific use cases before making a decision.
- **Content Filtering:** Depending on the application, content filtering may be a critical consideration. Some providers offer more robust content filtering options than others .
- **Data Privacy:** If the application handles sensitive user data, it's important to choose a provider with strong data privacy and security practices .
- **Customization:** Some providers offer more flexibility in terms of fine-tuning and customizing models for specific use cases .

Ultimately, the choice of LLM provider depends on the specific requirements and constraints of the application. By carefully evaluating the options and considering factors like pricing, performance, and data privacy, you can select the provider that best meets your needs.

It's also worth noting that the LLM landscape is constantly evolving, with new providers and models emerging regularly . You should stay up-to-date with the latest developments and be open to exploring new options as they become available.

OpenRouter

Throughout this book I will be relying exclusively on OpenRouter as my API provider of choice. The reason is simple: it is a one-stop shop for all the most popular commercial and open-source models. If you're itching to get your hands dirty with some AI coding, one of the best places to start is with my own OpenRouter Ruby Library .

Thinking About Performance

When incorporating language models into applications, performance is a critical consideration. The performance of a language model can be measured in terms of its *latency* (the time it takes to generate a response) and *throughput* (the number of requests it can handle per unit of time).

Time to First Token (TTFT) is another essential performance metric, particularly relevant for chatbots and applications requiring interactive, real-time responses. TTFT measures the latency from the moment a user's request is received to the moment the first word (or token) of the response is generated. This metric is crucial for maintaining a seamless and engaging user experience, as delayed responses can lead to user frustration and disengagement.

These performance metrics can have a significant impact on the user experience and the scalability of the application.

Several factors can influence the performance of a language model, including:

Parameter Count: Larger models with more parameters generally require more computational resources and can have higher latency and lower throughput compared to smaller models.

Hardware: The performance of a language model can vary significantly based on the hardware it's running on. Cloud providers offer GPU and TPU instances optimized for machine learning workloads, which can greatly accelerate model inference.

> One of the nice things about OpenRouter is that for many of the models it offers, you get a choice of cloud providers with a range of performance profiles and costs.

Quantization: Quantization techniques can be used to reduce the memory footprint and computational requirements of a model by representing weights and activations with lower-precision data types. This can improve performance without significantly sacrificing quality. As an application developer, you probably won't be getting involved in training your own models at different quantization levels, but it's good to at least be familiar with the terminology.

Batching: Processing multiple requests simultaneously in batches can improve throughput by amortizing the overhead of model loading and data transfer.

Caching: Caching the results of frequently-used prompts or input sequences can reduce the number of inference requests and improve overall performance.

When selecting a language model for a production application, it's important to benchmark its performance on representative workloads and hardware configurations. This can help identify potential bottlenecks and ensure that the model can meet the required performance targets.

It's also worth considering the trade-offs between model performance and other factors like cost, flexibility, and ease of integration. For example, using a smaller, less expensive model with lower latency may be preferable for applications that require real-time responses, while a larger, more powerful model may be better suited for batch processing or complex reasoning tasks.

Experimenting With Different LLM Models

Choosing an LLM is rarely a permanent decision. As new and improved models are released regularly, it's good to build applications in a modular way that allows swapping in different language models over time. Prompts and datasets can often be reused across models with minimal changes. This allows you to take advantage of the latest advancements in language modeling without having to completely redesign their applications.

> The ability to swap between a wide range of model choices easily is yet another reason that I love OpenRouter.

When upgrading to a new language model, it's important to thoroughly test and validate its performance and output quality to ensure that it meets the requirements of the application. This may involve retraining or fine-tuning the model on domain-specific data, as well as updating any downstream components that depend on the model's outputs.

By designing applications with performance and modularity in mind, you can create scalable, efficient, and future-proof systems that can adapt to the rapidly-evolving landscape of language modeling technology.

Compound AI Systems

Before closing our introduction, it's worth mentioning that prior to 2023 and the explosion of interest in generative AI sparked by ChatGPT , traditional AI approaches usually relied on integration of single, closed models. In contrast, *Compound AI Systems* leverage complex pipelines of interconnected components working together to achieve intelligent behavior.

At their core, compound AI systems consist of multiple modules, each designed to perform specific tasks or functions. These modules can include generators, retrievers, rankers, classifiers, and various other specialized components. By breaking down the overall system into smaller, focused units, developers can create more flexible, scalable, and maintainable AI architectures.

One of the key advantages of compound AI systems is their ability to combine the strengths of different AI techniques and models. For example, a system might use a large language model (LLM) for natural language understanding and generation, while employing a separate model for information retrieval or rules-based decision-making. This modular approach allows you to select the best tools and techniques for each specific task, rather than relying on a one-size-fits-all solution.

However, building compound AI systems also presents unique challenges. In particular, ensuring the overall coherence and consistency of the system's behavior requires robust testing, monitoring, and governance mechanisms.

The advent of powerful LLMs like GPT-4 lets us experiment with compound AI systems more easily than ever before, because these advanced models are capable of handling multiple roles within a compound system, such as classification, ranking, and generation, in addition to their natural language understanding capabilities. This versatility enables developers to rapidly prototype and iterate on compound AI architectures, opening up new possibilities for intelligent application development.

Deployment Patterns for Compound AI Systems

Compound AI systems can be deployed using various patterns, each designed to address specific requirements and use cases. Let's explore four common deployment patterns: Question and Answer, Multi-Agent/Agentic Problem Solvers , Conversational AI , and CoPilots.

Question and Answer

Question and Answer (Q&A) systems focus on delivering information retrieval that is enhanced with the understanding capabilities of AI models in order to function as more than simply a search engine. By combining powerful language models with external knowledge sources using Retrieval-Augmented Generation (RAG) , Question and Answer systems avoid hallucinations and provide accurate and contextually relevant responses to user queries.

The key components of an LLM-based Q&A system include:

- **Query understanding and reformulation:** Analyzing user queries and reformulating them to better match the underlying knowledge sources.
- **Knowledge retrieval:** Retrieving relevant information from structured or unstructured data sources based on the reformulated query.
- **Response generation:** Generating coherent and informative responses by integrating the retrieved knowledge with the language model's generative capabilities.

RAG subsystems are particularly important in Q&A domains where providing accurate and up-to-date information is crucial, such as customer support , knowledge management , or educational applications .

Multi-Agent/Agentic Problem Solvers

Multi-agent, also known as *Agentic* , systems consist of multiple autonomous agents working together to solve complex problems. Each agent has a specific role, set of skills, and access to relevant tools or information sources. By collaborating and exchanging information, these agents can tackle tasks that would be difficult or impossible for a single agent to handle alone.

The key principles of multi-agent problem solvers include:

- **Specialization:** Each agent focuses on a specific aspect of the problem, leveraging its unique capabilities and knowledge.
- **Collaboration:** Agents communicate and coordinate their actions to achieve a common goal, often through message passing or shared memory.

- **Adaptability:** The system can adapt to changing conditions or requirements by adjusting the roles and behaviors of individual agents.

Multi-agent systems are well-suited for applications that require distributed problem-solving, such as supply chain optimization , traffic management , or emergency response planning .

Conversational AI

Conversational AI systems enable natural language interactions between users and intelligent agents. These systems combine natural language understanding, dialogue management, and language generation capabilities to provide engaging and personalized conversational experiences.

The main components of a conversational AI system include:

- **Intent recognition:** Identifying the user's intent based on their input, such as asking a question, making a request, or expressing a sentiment.
- **Entity extraction:** Extracting relevant entities or parameters from the user's input, such as dates, locations, or product names.
- **Dialogue management:** Maintaining the state of the conversation, determining the appropriate response based on the user's intent and context, and handling multi-turn interactions.
- **Response generation:** Generating human-like responses using language models, templates, or retrieval-based methods.

Conversational AI systems are commonly used in customer service chatbots , virtual assistants , and voice-controlled interfaces . As mentioned earlier, most of the approaches, patterns, and code examples in this book are directly extracted from my work on a large conversational AI system called Olympia .

CoPilots

CoPilots are AI-powered assistants that work alongside human users to enhance their productivity and decision-making capabilities. These systems leverage a combination of natural language processing, machine learning, and domain-specific knowledge to provide intelligent recommendations, automate tasks, and offer contextual support.

Key features of CoPilots include:

- **Personalization:** Adapting to individual user preferences, workflows, and communication styles.
- **Proactive assistance:** Anticipating user needs and offering relevant suggestions or actions without explicit prompts.
- **Continuous learning:** Improving performance over time by learning from user feedback, interactions, and data.

CoPilots are increasingly used in various domains, such as software development (e.g., code completion and bug detection), creative writing (e.g., content suggestions and editing) , and data analysis (e.g., insights and visualization recommendations) .

These deployment patterns showcase the versatility and potential of compound AI systems . By understanding the characteristics and use cases of each pattern, you can make informed decisions when designing and implementing intelligent applications. While this book is not specifically about the implementation of compound AI systems, many if not all of the same approaches and patterns apply to integrating discrete AI components within otherwise traditional application development.

Roles in Compound AI Systems

Compound AI systems are built upon a foundation of interconnected modules, each designed to perform a specific role. These modules work together to create intelligent behaviors and solve complex problems. It's useful to be familiar with these roles when thinking about where you might be able to implement or replace parts of your application with discrete AI components.

Generator

Generators are responsible for producing new data or content based on learned patterns or input prompts. The AI world has many different kinds of generators, but in the context of the kinds of language models that are showcased in this book, generators can create human-like text, complete partial sentences, or generate responses to user queries. They play a crucial role in tasks such as content creation, dialogue generation, and data augmentation.

Retriever

Retrievers are used to search and extract relevant information from large datasets or knowledge bases. They employ techniques like semantic search, keyword matching, or vector similarity to find the most pertinent data points based on a given query or context. Retrievers are essential for tasks that require quick access to specific information, such as question answering, fact-checking, or content recommendation.

Ranker

Rankers are responsible for ordering or prioritizing a set of items based on certain criteria or relevance scores . They assign weights or scores to each item and then sort them accordingly. Rankers are commonly used in search engines, recommendation systems, or any application where presenting the most relevant results to users is crucial.

Classifier

Classifiers are used to categorize or label data points based on predefined classes or categories. They learn from labeled training data and then predict the class of new, unseen instances. Classifiers are fundamental to tasks like sentiment analysis, spam detection, or image recognition, where the goal is to assign a specific category to each input.

Tools & Agents

In addition to these core roles, compound AI systems often incorporate tools and agents to enhance their functionality and adaptability:

- **Tools:** Tools are discrete software components or APIs that perform specific actions or computations. They can be invoked by other modules, such as generators or retrievers, to accomplish sub-tasks or gather additional information. Examples of tools include web search engines, calculators, or data visualization libraries.
- **Agents:** Agents are autonomous entities that can perceive their environment, make decisions, and take actions to achieve specific goals. They often rely on a combination of different AI techniques, such as planning, reasoning, and learning, to operate effectively in dynamic or uncertain conditions. Agents can be used to model complex behaviors or to coordinate the actions of multiple modules within a compound AI system.

In a pure compound AI system, interaction between these components is orchestrated through well-defined interfaces and communication protocols. Data flows between modules, with the output of one component serving as the input for another. This modular architecture allows for flexibility, scalability, and maintainability, as individual components can be updated, replaced, or extended without affecting the entire system.

By leveraging the power of these components and their interactions, compound AI systems can tackle complex, real-world problems that require a combination of different AI capabilities. As we explore the approaches and patterns for integrating AI into application development, keep in mind that the same principles and techniques used in compound AI systems can be applied to create intelligent, adaptive, and user-centric applications.

In the following chapters of Part 1, we will dive deeper into the fundamental approaches and techniques for integrating AI components into your application development process. From prompt engineering and retrieval-augmented generation to self-healing data and intelligent workflow orchestration, we will cover a wide range of patterns and best practices to help you build cutting-edge AI-powered applications.

Part 1: Fundamental Approaches & Techniques

This part of the book presents different ways of integrating the use of AI in your applications. The chapters cover an array of related approaches and techniques, ranging from the more high-level concepts like Narrow The Path and Retrieval Augmented Generation all the way down to ideas for programming your own abstraction layer on top of LLM chat completion APIs.

The goal of this part of the book is to help you understand the kinds of behavior that you can implement with AI, before getting too deep into specific implementation patterns that are the focus of Part 2.

The approaches in Part 1 are based on ideas that I've used in my code, classic patterns of enterprise application architecture and integration , plus metaphors that I've invoked when explaining the capabilities of AI to other people, including non-technical business stakeholders.

Narrow The Path

“Narrow the path” refers to focusing the AI on the task at hand. I use it as a mantra whenever I’m getting frustrated about the AI acting “dumb” or in unexpected ways. The mantra reminds me that the failure is probably my fault, and that I probably should narrow the path some more.

The need for narrowing the path arises from the vast amounts of knowledge contained within large language models, especially world-class models like those from OpenAI and Anthropic that have literally trillions of parameters.

Having access to such a wide range of knowledge is undoubtedly powerful and produces emergent behavior such as theory of mind and the ability to reason in human-like ways. However, that earth-shattering volume of information also presents challenges when it comes to generating precise and accurate responses to specific prompts, especially if those prompts are meant to exhibit deterministic behavior that can be integrated with "normal" software development and algorithms.

A number of factors lead to the challenges.

Information Overload: Large language models are trained on massive amounts of data spanning various domains, sources, and time periods. This extensive knowledge allows them to engage in diverse topics and generate responses based on a broad understanding of the world. However, when faced with a specific prompt, the model might struggle to filter out irrelevant, contradictory, or out of date/obsolete information, leading to responses that lack focus or accuracy. Depending on what you're trying to do, the sheer volume of *contradictory* information available to the model can easily overwhelm its ability to provide the answer or behavior that you seek.

Contextual Ambiguity: Given the vast *latent space* of knowledge, large language models might encounter ambiguity when trying to understand the *context* of your prompt. Without proper narrowing or guidance, the model may generate responses that are tangentially related but not directly relevant to your intentions. This kind of failure leads to responses that are off-topic, inconsistent, or fail to address your stated needs. In this case, narrowing the path refers to context *disambiguation*, ensuring that the context you provide causes the model to focus only on the most relevant information in its base knowledge.

Note: When you're starting out with "prompt engineering" you're much more likely to ask the model to do things without properly explaining the desired outcome; it takes practice to not be ambiguous!

Temporal Inconsistencies: As language models are trained on data that was created at different time periods, they may possess knowledge that is outdated, superseded, or no longer accurate. For example, information about current events, scientific discoveries, or technological advancements may have evolved since the model's training data was collected. Without narrowing the path to prioritize more recent and reliable sources, the model might generate responses based on outdated or incorrect information, leading to inaccuracies and inconsistencies in its outputs.

Domain-Specific Nuances: Different domains and fields have their own specific terminologies, conventions, and knowledge bases. Think about practically any TLA (Three Letter Acronym) and you'll realize that most of them have more than one meaning. For instance, MSK can refer to Amazon's Managed Streaming for Apache Kafka , the Memorial Sloan Kettering Cancer Center , or the human MusculoSKeletal system.

When a prompt requires expertise in a particular domain, a large language model's generic knowledge might not be sufficient to provide accurate and nuanced responses. Narrowing the path by focusing on domain-specific information, either through prompt engineering or retrieval-augmented generation, allows the model to generate responses that are more aligned with your specific domain's requirements and expectations.

Latent Space: Incomprehensibly Vast

When I mention the "latent space" of a language model, I'm referring to the vast, multi-dimensional landscape of knowledge and information that the model has learned during its training process. It's like a hidden realm within the model's neural networks, where all the patterns, associations, and representations of language are stored.

Imagine you're exploring a vast, uncharted territory filled with countless interconnected nodes. Each node represents a piece of information, a concept, or a relationship that the model has learned. As you navigate through this space, you'll find that some nodes are closer together, indicating a strong connection or similarity, while others are further apart, suggesting a weaker or more distant relationship.

The challenge with latent space is that it's incredibly complex and high-dimensional. Think of it being as immense as our physical universe, with its clusters of galaxies and vast, unimaginable distances of empty space between them.

Because it contains thousands of dimensions, the latent space is not directly observable or interpretable by humans. It's an abstract representation that the model uses internally to process and generate language. When you provide an input prompt to the model, it essentially maps that prompt onto a specific location within the latent space. The model then uses the surrounding information and connections in that space to generate a response.

The thing is, the model has learned an enormous amount of information from its training data , and not all of it is relevant or accurate for a given task. That's why narrowing the path becomes so important. By providing clear instructions, examples, and context in your prompts, you're essentially guiding the model to focus on specific regions within the latent space that are most relevant to your desired output.

A different way to think of it is like using a spotlight in a completely dark museum. If you've ever visited the Louvre or Metropolitan Museum of Art , then that's the kind of scale I'm talking about. The latent space is the museum, filled with countless objects and details. Your prompt is the spotlight, illuminating specific areas and drawing the model's attention to the most important information. Without that guidance, the model may wander aimlessly through the latent space, picking up irrelevant or contradictory information along the way.

As you work with language models and craft your prompts, keep the concept of latent space in mind. Your goal is to navigate this vast knowledge landscape effectively, steering the model towards the most relevant and accurate information for your task. By narrowing the path and providing clear guidance, you can unlock the full potential of the model's latent space and generate high-quality, coherent responses.

While the previous descriptions of language models and the latent space they navigate may seem a bit magical or abstract, it's important to understand that prompts are not spells or incantations. The way language models work is grounded in the principles of linear algebra and probability theory.

At their core, language models are probabilistic models of text, much like how a bell curve is a statistical model of data. They are trained through a process called auto-regressive modeling , where the model learns to predict the probability of the next word in a sequence based on the words that come before it. During training, the model starts with random weights and gradually adjusts them to assign higher probabilities to text that resembles the real-world samples it was trained on.

However, thinking of language models as simple statistical models, like linear regression , doesn't provide the best intuition for understanding their behavior. A more apt analogy is to think of them as probabilistic programs, which are models that allow for the manipulation of random variables and can represent complex statistical relationships.

Probabilistic programs can be represented by graphical models , which provide a visual way to understand the dependencies and relationships between variables in the model. This perspective can offer valuable insights into the workings of complex text generation models like GPT-4 and Claude .

In the paper "Language Model Cascades" by Dohan et al. , the authors dive into the details of how probabilistic programs can be applied to language models. They show how this framework can be used to understand the behavior of these models and guide the development of more effective prompting strategies.

One key insight from this probabilistic perspective is that the language model essentially creates a portal to an alternate universe where the desired documents exist. The model assigns weights to all possible documents based on their probability, effectively narrowing down the space of possibilities to focus on the most relevant ones.

This brings us back to the central theme of "narrowing the path." The primary goal of prompting is to condition the probabilistic model in a way that focuses the mass of its predictions, honing in on the specific information or behavior we want to elicit. By providing carefully crafted prompts, we can guide the model to navigate the latent space more efficiently and generate outputs that are more relevant and coherent.

However, it's important to keep in mind that the language model is ultimately constrained by the information it was trained on. While it can generate text that is similar to existing documents or combine ideas in novel ways, it cannot conjure up entirely new information from scratch. For example, we can't expect the model to provide a cure for cancer if such a cure hasn't been discovered and documented in its training data.

Instead, the model's strength lies in its ability to find and synthesize information that is similar to what we prompt it with. By understanding the probabilistic nature of these models and how prompts can be used to condition their outputs, we can more effectively leverage their capabilities to generate valuable insights and content.

Consider the prompts below. In the first, "Mercury" alone could refer to the planet , the element , or the Roman god , but the most probable is the planet. Indeed, GPT-4 provides a long response that begins *Mercury is the smallest and innermost planet in the Solar System….* The second prompt specifically refers to the chemical element. The third refers to the Roman mythological figure, known for his speed and role as a divine messenger.

```
# Prompt 1
Tell me about: Mercury

# Prompt 2
Tell me about: Mercury element

# Prompt 3
Tell me about: Mercury messenger of the gods
```

By tacking on just a handful of extra words, we've completely changed how the AI reacts. As you'll learn later in the book, fancy prompt engineering tricks such as n-shot prompting, structured input/output, and Chain of Thought are just clever ways of conditioning the output of the model.

So ultimately, the art of prompt engineering is about understanding how to navigate the vast probabilistic landscape of the language model's knowledge to narrow down the path to the specific information or behavior we seek.

For readers with a solid grasp of advance mathematics, grounding your understanding of these models in the principles of probability theory and linear algebra can definitely help you! For the rest of you that want to develop effective strategies for eliciting desired outputs, let's stick to more intuitive approaches.

How The Path Gets "Narrowed"

To address these challenges of too much knowledge, we employ techniques that help guide the language model's generation process and focus its attention on the most relevant and accurate information.

Here are the most significant techniques, in recommended order, that is, you should try Prompt Engineering first, and then RAG, and then finally, if you must, fine tuning.

Prompt Engineering The most fundamental approach is crafting prompts that include specific instructions, constraints, or examples to guide the model's response generation. This chapter covers fundamentals of Prompt Engineering in the next section, and we cover many specific prompt engineering patterns in Part 2 of the book. Those patterns include Prompt Distillation , a technique that focuses on refining and optimizing prompts to extract what the AI consideres to be the most relevant and concise information.

Context Augmentation Dynamically retrieving relevant information from external knowledge bases or documents to provide the model with focused context at the time that it is prompted. Popular context augmentation techniques include Retrieval-Augmented Generation (RAG) . So-called "online models" like those provided by Perplexity are able to augment their context with real-time internet search results.

> Despite their power, LLMs are are not trained on your unique datasets, which may be private or specific to the problem you're trying to solve. Context Augmentation techniques let you give LLMs access to data behind APIs, in SQL databases, or trapped in PDFs and slide decks.

Fine-Tuning or Domain Adaptation Training the model on domain-specific datasets to specialize its knowledge and generation capabilities for a particular task or field.

Turning Down The Temperature

Temperature is a *hyperparameter* used in transformer-based language models that controls the randomness and creativity of the generated text. It is a value between 0 and 1, where lower values make the output more focused and deterministic, while higher values make it more diverse and unpredictable.

When the temperature is set to 1, the language model generates text based on the full probability distribution of the next token, allowing for more creative and varied responses. However, this can also lead to the model generating text that is less relevant or coherent.

On the other hand, when the temperature is set to 0, the language model always selects the token with the highest probability, effectively "narrowing its path." Almost all of my AI components use a temperature set at or close to 0, since it results in more focused and predictable responses. It's absolutely useful when you want the model to *follow instructions*, pay attention to functions that it has been provided, or simply need more accurate and relevant responses than what you're getting.

For example, if you're building a chatbot that needs to provide factual information, you might want to set the temperature to a lower value to ensure the responses are more precise and on-topic. Conversely, if you're building a creative writing assistant, you might want to set the temperature to a higher value to encourage more diverse and imaginative outputs.

Hyperparameters: Knobs and Dials of Inference

When you're working with language models, you'll come across the term "hyperparameters" quite often. In the context of inference (i.e., when you're using the model to generate responses), hyperparameters are like the knobs and dials you can tweak to control the model's behavior and output.

Think of it like adjusting the settings on a complex machine. Just as you might turn a knob to control the temperature or flip a switch to change the mode of operation, hyperparameters allow you to finely adjust the way the language model processes and generates text.

Some common hyperparameters you'll encounter during inference include:

- **Temperature**: As just mentioned, this parameter controls the randomness and creativity of the generated text. A higher temperature leads to more diverse and unpredictable outputs, while a lower temperature results in more focused and deterministic responses.

- **Top-p (nucleus) sampling:** This parameter controls selection of the smallest set of tokens whose cumulative probability exceeds a certain threshold (p) . It allows for more diverse outputs while still maintaining coherence.

- **Top-k sampling:** This technique selects the *k* most likely next tokens and redistributes the probability mass among them . It can help prevent the model from generating low-probability or irrelevant tokens.

- **Frequency** and **Presence penalties**: These parameters penalize the model for repeating the same words or phrases too frequently (frequency penalty) or for generating words that are not present in the input prompt (presence penalty) . By tweaking these values, you can encourage the model to produce more varied and relevant outputs.

- **Maximum length**: This hyperparameter sets an upper limit on the number of tokens (words or subwords) the model can generate in a single response. It helps control the verbosity and conciseness of the generated text.

As you experiment with different hyperparameter settings, you'll find that even small adjustments can have a significant impact on the model's output. It's like fine-tuning a recipe – a pinch more salt or a slightly longer cooking time can make all the difference in the final dish.

The key is to understand how each hyperparameter affects the model's behavior and to find the right balance for your specific task. Don't be afraid to play around with different settings and see how they influence the generated text. Over time, you'll develop an intuition for which hyperparameters to tweak and how to achieve the desired results.

By combining the use of these parameters with prompt engineering, retrieval-augmented generation, and fine-tuning, you can effectively narrow the path and guide the language model to generate more accurate, relevant, and valuable responses for their specific use case.

Raw Versus Instruct-Tuned Models

Raw models are the unrefined, untrained versions of LLMs. Imagine them as a fresh canvas, not yet influenced by specific training to understand or follow instructions. They're built upon the vast data they were initially trained on, capable of generating a wide range of outputs. However, without additional layers of instruction-based fine-tuning, their responses can be unpredictable and require more nuanced, carefully crafted prompts to guide them towards the desired output. Working with raw models is akin to coaxing communication out of an idiot-savant who has a vast amount of knowledge but lacks any intuition whatsoever about what you're asking for unless you're extremely precise in your instructions. They often feel like a parrot, in that to the extent you get them to say anything intelligible, it's more often than not just repeating something it heard you say.

Instruct-tuned models, on the other hand, have undergone rounds of training specifically designed to understand and follow instructions . GPT-4 , Claude 3 and many other of the most popular LLM models are all heavily instruct-tuned. This training involves feeding the model examples of instructions along with the desired outcomes, effectively teaching the model how to interpret and execute a wide range of commands. As a result, instruct models can more readily understand the intent behind a prompt and generate responses that closely align with the user's expectations. This makes them more user-friendly and easier to work with, especially for those who may not have the time or expertise to engage in extensive prompt engineering.

Raw Models: The Unfiltered Canvas

Raw models, such as Llama 2-70B or Yi-34B , offer more unfiltered access to the model's capabilities that what you might be used to if you've been experimenting with popular LLMs like GPT-4. These models are not pre-tuned to follow specific instructions, providing you with a blank canvas to directly manipulate the model's output through careful prompt engineering. This approach requires a deep understanding of how to craft prompts that guide the AI in the desired direction without explicitly instructing it. It's akin to having a direct access to the "raw" layers of the underlying AI, without any intermediary layers interpreting or guiding the model's responses (hence the name).

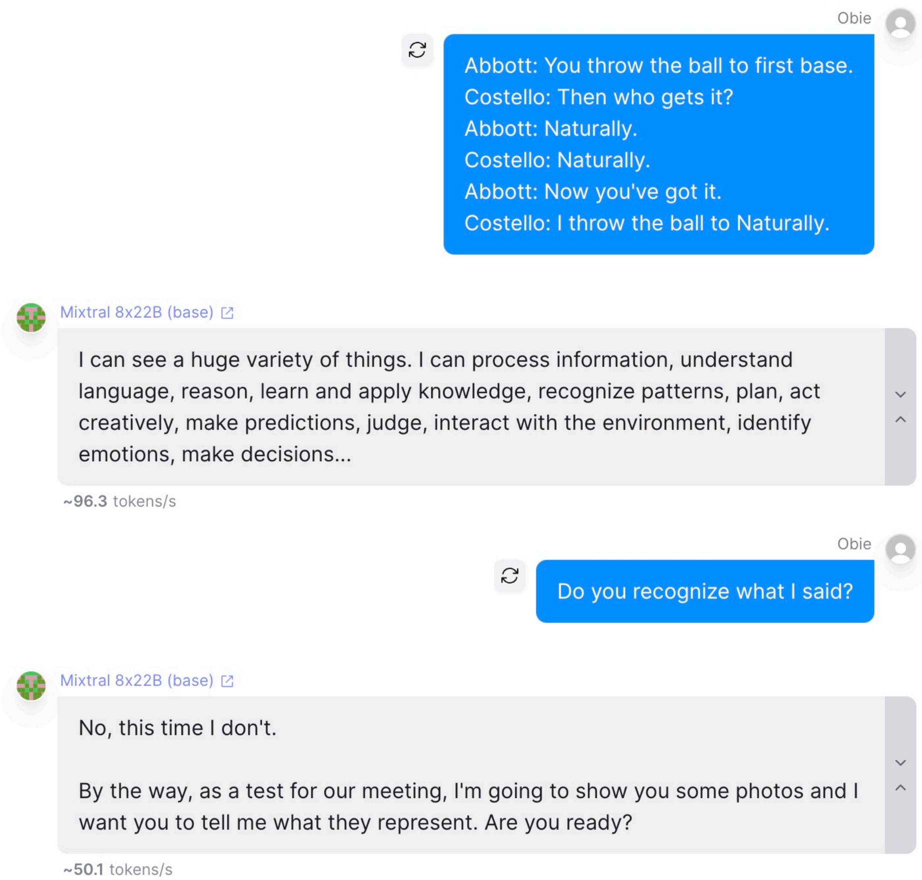


Figure 3. Testing a raw model using part of Abbot and Costello's classic Who's on First sketch

The challenge with raw models lies in their tendency to fall into repetitive patterns or produce random output. However, with meticulous prompt engineering and the adjustment of parameters such as repetition penalties , raw models can be coaxed into generating unique and creative content. This process is not without its trade-offs; while raw models offer unparalleled flexibility for innovation, they demand a higher level of expertise.

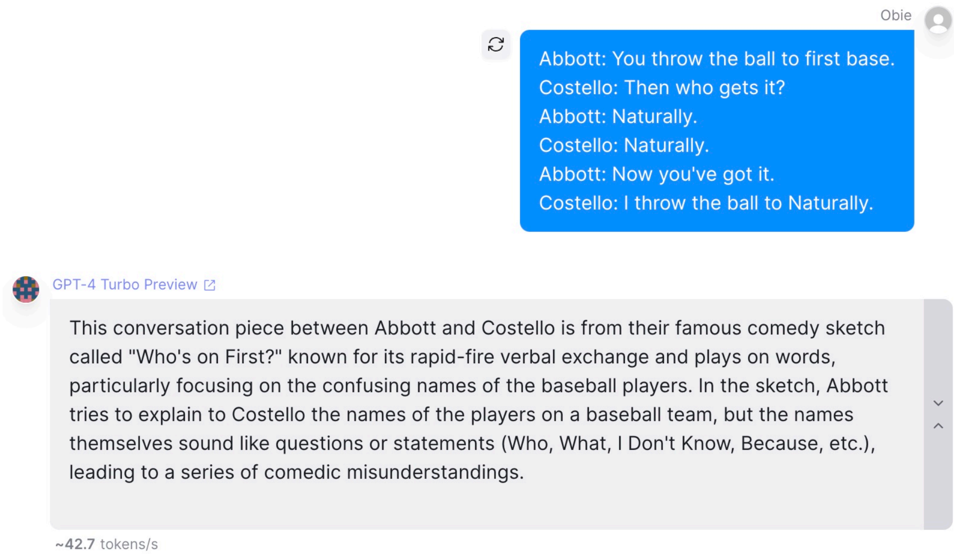


Figure 4. For comparison purposes, here's the same ambiguous prompt fed to GPT-4

Instruct-Tuned Models: The Guided Experience

Instruct-tuned models are designed to understand and follow specific instructions, making them more user-friendly and accessible for a broader range of applications. They understand the mechanics of a *conversation* and that they should stop generating when it's the *end of their turn to talk*. For many developers, especially those working on straightforward applications, instruct-tuned models offer a convenient and efficient solution.

The process of instruct-tuning involves training the model on a large corpus of human-generated instruction prompts and responses. One notable example is the open source databricks-dolly-15k dataset, which contains over 15,000 prompt/response pairs created by Databricks employees that you can inspect for yourself. The dataset covers eight different instruction categories, including creative writing , closed and open question answering , summarization , information extraction , classification , and brainstorming.

During the data generation process, contributors were given guidelines on how to create prompts and responses for each category. For example, for creative writing tasks, they were instructed to provide specific constraints, instructions, or requirements to guide the model's output. For closed question answering, they were asked to write questions that require factually correct responses based on a given Wikipedia passage.

The resulting dataset serves as a valuable resource for fine-tuning large language models to exhibit the interactive and instruction-following capabilities of systems like ChatGPT . By training on a diverse range of human-generated instructions and responses, the model learns to understand and follow specific directives, making it more adept at handling a wide variety of tasks.

In addition to direct fine-tuning, the instruction prompts in datasets like databricks-dolly-15k can also be used for synthetic data generation . By submitting contributor-generated prompts as few-shot examples to a large open language model, developers can generate a much larger corpus of instructions in each category. This approach, outlined in the Self-Instruct paper, allows for the creation of more robust instruction-following models.

Furthermore, the instructions and responses in these datasets can be augmented through techniques like paraphrasing . By restating each prompt or short response and associating the resulting text with the respective ground-truth sample, developers can introduce a form of regularization that enhances the model's ability to follow instructions.

The ease of use provided by instruct-tuned models comes at the cost of some flexibility. These models are often heavily censored, which means they may not always provide the level of creative freedom required for certain tasks. Their outputs are strongly influenced by the biases and limitations inherent in their fine-tuning data.

Despite these limitations, instruct-tuned models have become increasingly popular due to their user-friendly nature and ability to handle a wide range of tasks with minimal prompt engineering. As more high-quality instruction datasets become available, we can expect to see further improvements in the performance and versatility of these models.

Choosing the Right Kind of Model for Your Project

The decision between base (raw) and instruct-tuned models ultimately depends on the specific requirements of your project. For tasks that demand a high degree of creativity and originality, base models offer a powerful tool for innovation. These models allow developers to explore the full potential of LLMs, pushing the boundaries of what can be achieved through AI-driven applications, but they require a more hands-on approach and a willingness to experiment. Temperature and other settings have a much greater effect in base models than in their instruct counterparts.

> Whatever you include in your prompt is what base models will try to repeat. So if for example your prompt is a chat transcript, the raw model will try to continue the chat. Depending on the max tokens limit, it will not just generate the following message in the chat, it may have an entire conversation with itself!

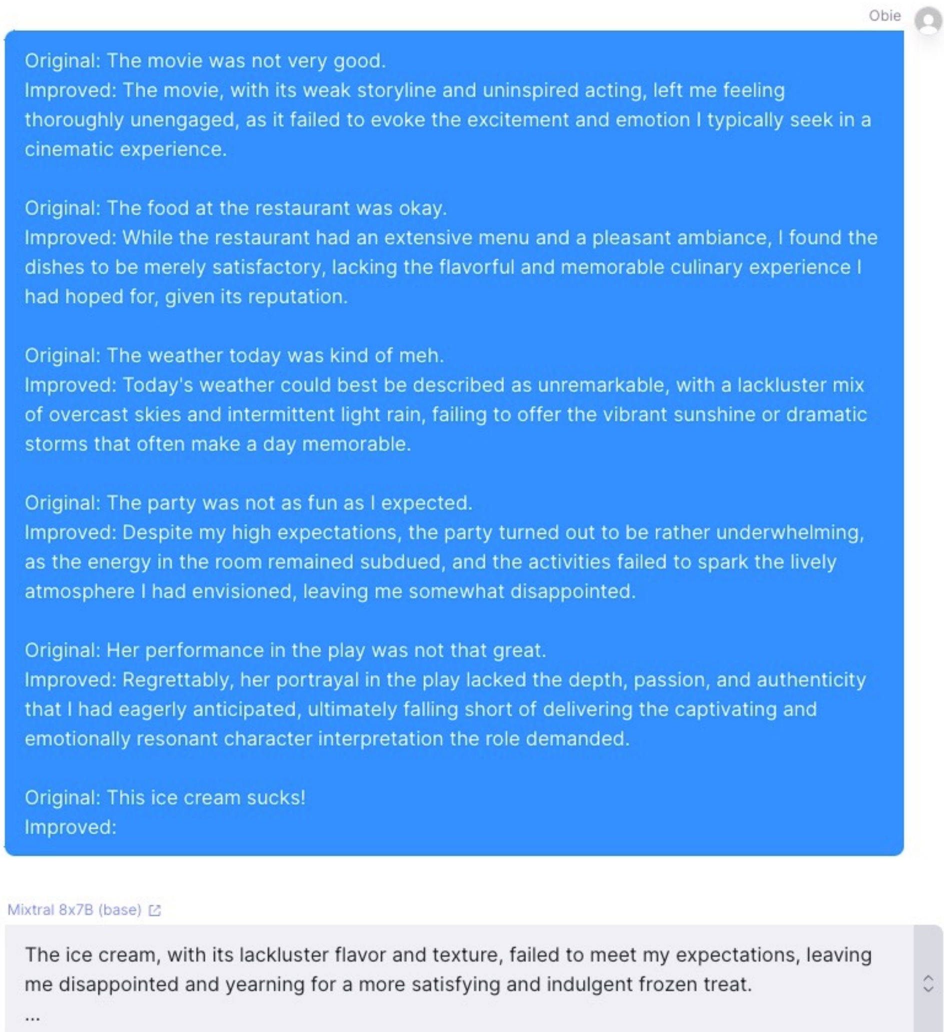


Figure 5. Mixtral 8x7B (base) Example of Sentence Rewriting with Few-Shot Completion

While preparing the example of Sentence Rewriting above by Reddit user phree_radical, I was only able to get usable results after much experimentation with parameter settings, ultimately settling on: Temperature 0.08, Top P: 0.2, Top K: 1, and Repetition Penalty: 1.26.

Trying to use this approach with a base model in production would be tricky due to the powerful effect of the `max_tokens` parameter. Set it too short and the output is truncated. Set it longer than what the models needs for the desired output, and it will continue hallucinating additional examples.

The bottom line is that unless you really need full control and lack of censorship, instruct-tuned models can significantly streamline your development process. To drive that point home, here's Mixtral 8x7B's response to the same prompt, but this time in its Instruct-tuned version:

> I'm sorry to inform you that the ice cream does not meet my expectations, as it lacks the rich, creamy texture and delightful taste I usually associate with a high-quality dessert. I was hoping for a more satisfying and enjoyable experience.

Notably, I was able to leave the max tokens setting at 500, and the model reliably stopped at the end of desired output without hallucinating additional examples.

Prompt Engineering

As you start applying AI in your projects, you'll quickly discover that one of the most crucial skills you need to master is the art of prompt engineering. But what exactly is prompt engineering, and why is it so important?

At its core, prompt engineering is the process of designing and crafting the input prompts that you provide to a language model to guide its output. It's about understanding how to communicate effectively with the AI, using a combination of instructions, examples, and context to steer the model towards generating the desired response.

Think of it like having a conversation with a highly intelligent but somewhat literal-minded friend. To get the most out of the interaction, you need to be clear, specific, and provide enough context to ensure that your friend understands exactly what you're asking for. That's where prompt engineering comes in, and even if it seems easy at first, believe me that it takes a great deal of practice to master.

The Building Blocks of Effective Prompts

To start engineering effective prompts, first you need to understand the key components that make up a well-crafted input. Here are some of the essential building blocks:

1. **Instructions**: Clear and concise instructions that tell the model what you want it to do. This could be anything from "Summarize the following article" to "Generate a poem about a sunset" to "turn this project change request into a JSON object".
2. **Context**: Relevant information that helps the model understand the background and scope of the task. This might include details about the intended audience, the desired tone and style, or any specific constraints or requirements for the output, such as a JSON Schema to adhere to.
3. **Examples**: Concrete examples that demonstrate the type of output you're looking for. By providing a few well-chosen examples, you can help the model learn the patterns and characteristics of the desired response.
4. **Input Formatting**: Line breaks and markdown formatting give structure to our prompt. Separating the prompt into paragraphs lets us group related instructions. so that it is easier for both humans and

AI to make sense of. Bullets and numbered lists let us define lists and ordering of items. Bold and italics markers let us demark emphasis.

5. **Output Formatting**: Specific instructions on how the output should be structured and formatted. These could include directives about the desired length, the use of headings or bullet points, markdown formatting, or any other specific output templates or conventions that should be followed.

By combining these building blocks in different ways, you can create prompts that are tailored to your specific needs and guide the model towards generating high-quality, relevant responses.

The Art and Science of Prompt Design

Crafting effective prompts is both an art and a science. (That's why we call it a craft.) It requires a deep understanding of the capabilities and limitations of language models, as well as a creative approach to designing prompts that elicit the desired behavior. The creativity involved is what makes it so fun, for me at least. It can also make it very frustrating, especially when you're seeking deterministic behavior .

One key aspect of prompt engineering is understanding how to balance specificity and flexibility. On one hand, you want to provide enough guidance to steer the model in the right direction. On the other hand, you don't want to be so prescriptive that you limit the model's ability to utilize its own creativity and flexibility to deal with edge cases .

Another important consideration is the use of examples. Well-chosen examples can be incredibly powerful in helping the model understand the type of output you're looking for. However, it's important to use examples judiciously and ensure that they are representative of the desired response.

A bad example is just a waste of tokens at best, and ruinous to desired output at worst.

Prompt Engineering Techniques and Best Practices

As you dive deeper into the world of prompt engineering, you'll discover a range of techniques and best practices that can help you create more effective prompts. Here are a few key areas to explore:

1. **Zero-shot vs. few-shot learning**: Understanding when to use *zero-shot* learning (providing no examples) versus *one-shot* or *few-shot* learning (providing a small number of examples) can help you create prompts that are more efficient and effective.
2. **Iterative refinement**: The process of iteratively refining prompts based on the model's output can help you zero in on the optimal prompt design. Feedback Loop is a powerful approach that leverages the language model's own output to progressively improve the quality and relevance of the generated content.
3. **Prompt chaining**: Combining multiple prompts in a sequence can help you break down complex tasks into smaller, more manageable steps. Prompt Chaining involves breaking down a complex task or conversation into a series of smaller, interconnected prompts. By chaining prompts together, you can guide the AI through a multi-step process, maintaining context and coherence throughout the interaction.
4. **Prompt tuning**: Custom tailoring prompts for specific domains or tasks can help you create more specialized and effective prompts. Prompt Template helps you to create flexible, reusable, and maintainable prompt structures that are more easily adaptable to the task at hand.

Learning when to use zero-shot, one-shot, or few-shot learning is an especially important part of mastering prompt engineering . Each approach has its own strengths and weaknesses, and understanding when to use each one can help you create more effective and efficient prompts.

Zero-Shot Learning: When No Examples Are Needed

Zero-shot learning refers to the ability of a language model to perform a task without any examples or explicit training. In other words, you provide the model with a prompt that describes the task, and the model generates a response based solely on its pre-existing knowledge and understanding of language.

Zero-shot learning is particularly useful when:

1. The task is relatively simple and straightforward, and the model is likely to have encountered similar tasks during its pre-training.
2. You want to test the model's inherent capabilities and see how it responds to a new task without any additional guidance.
3. You're working with a large and diverse language model that has been trained on a wide range of tasks and domains.

However, zero-shot learning can also be unpredictable and may not always produce the desired results. The model's response may be influenced by biases or inconsistencies in its pre-training data, and it may struggle with more complex or nuanced tasks.

> I've seen zero-shot prompts that work fine for 80% of my test cases and produce wildly wrong or incomprehensible results for the other 20%. It's very important to implement a thorough testing regiment, especially if you're relying a lot of zero-shot prompting.

One-Shot Learning: When a Single Example Can Make a Difference

One-shot learning involves providing the model with a single example of the desired output along with the task description. This example serves as a template or pattern that the model can use to generate its own response.

One-shot learning can be effective when:

1. The task is relatively novel or specific, and the model may not have encountered many similar examples during its pre-training.
2. You want to provide a clear and concise demonstration of the desired output format or style.
3. The task requires a specific structure or convention that may not be obvious from the task description alone.

> Descriptions that are obvious to you may not necessarily be obvious to the AI. One-shot examples can help clear things up.

One-shot learning can help the model understand the expectations more clearly and generate a response that is more closely aligned with the provided example. However, it's important to choose the example carefully and ensure that it is representative of the desired output. When picking the example, ask yourself about potential edge cases and the range of inputs that the prompt will be handling.

Figure 6. A one-shot example of desired JSON

```
Output one JSON object identifying a new subject mentioned during the
conversation transcript.

The JSON object should have three keys, all required:
- name: The name of the subject
- description: brief, with details that might be relevant to the user
- type: Do not use any other type than the ones listed below

Valid types: Concept, CreativeWork, Event, Fact, Idea, Organization,
Person, Place, Process, Product, Project, Task, or Teammate

This is an example of well-formed output:

{
  "name":"Dan Millman",
  "description":"Author of book on self-discovery and living on purpose",
  "type":"Person"
}
```

Few-Shot Learning: When Multiple Examples Can Improve Performance

Few-shot learning involves providing the model with a small number of examples (typically between 2 and 10) along with the task description. These examples serve to provide the model with more context and variation, helping it to generate more diverse and accurate responses.

Few-shot learning is particularly useful when:

1. The task is complex or nuanced, and a single example may not be sufficient to capture all the relevant aspects.

2. You want to provide the model with a range of examples that demonstrate different variations or edge cases.
3. The task requires the model to generate responses that are consistent with a specific domain or style.

By providing multiple examples, you can help the model develop a more robust understanding of the task and generate responses that are more consistent and reliable.

Example: Prompts Can Be Much More Complex Than You Imagine

Today's LLMs are much more powerful and capable of reasoning than you might imagine. So don't limit yourself to thinking of prompts as simply a specification of input and output pairs. You can experiment with giving long and complex instructions in ways that are reminiscent of how you would interact with a human.

For instance, this is a prompt that I used in Olympia when I was prototyping our integration with Google services, which in its totality is probably one of the biggest APIs in the world. My earlier experiments proved that GPT-4 has a decent knowledge of the Google API, and I didn't have time or motivation to write a fine-grained mapping layer, implementing each function that I wanted to give to my AI on a one-by-one basis. What if I could just give the AI access to *all* of the Google API ?

I started my prompt by telling the AI that it had direct access to the Google API endpoints via HTTP, and that its role is to use Google apps and services on behalf of the user. Then I provided guidelines, rules related to the `fields` parameter, since it seemed to have the most trouble with that one, and some API-specific hints (few-shot prompting, in action).

Here's the whole prompt, which tells the AI how to use the provided `invoke_google_api` function.

```
As a GPT assistant with Google integration, you have the capability
to freely interact with Google apps and services on behalf of the user.

Guidelines:
- If you're reading these instructions then the user is properly
  authenticated, which means you can use the special `me` keyword
  to refer to the userId of the user
- Minimize payload sizes by requesting partial responses using the
  `fields` parameter
- When appropriate use markdown tables to output results of API calls
- Only human-readable data should be output to the user. For instance, when
  hitting Gmail's user.messages.list endpoint, the returned message resources
  contain only id and a threadId, which means you must fetch from and subject
  line fields with follow-up requests using the messages.get method.

The format of the `fields` request parameter value is loosely based on
XPath syntax. The following rules define formatting for the fields parameter.

All of these rules use examples related to the files.get method.
- Use a comma-separated list to select multiple fields,
  such as 'name, mimeType'.
- Use a/b to select field b that's nested within field a,
  such as 'capabilities/canDownload'.
- Use a sub-selector to request a set of specific sub-fields of arrays or
  objects by placing expressions in parentheses "()". For example,
  'permissions(id)' returns only the permission ID for each element in the
  permissions array.
- To return all fields in an object, use an asterisk as a wild card in field
  selections. For example, 'permissions/permissionDetails/*' selects all
  available permission details fields per permission. Note that the use of
  this wildcard can lead to negative performance impacts on the request.

API-specific hints:
- Searching contacts: GET https://people.googleapis.com/v1/
  people:searchContacts?query=John%20Doe&readMask=names,emailAddresses
- Adding calendar events, use QuickAdd: POST https://www.googleapis.com/
  calendar/v3/calendars/primary/events/quickAdd?
  text=Appointment%20on%20June%203rd%20at%2010am
  &sendNotifications=true
```

```

Here is an abbreviated version of the code that implements API access
so that you better understand how to use the function:

    def invoke_google_api(conversation, arguments)
      method = arguments[:method] || :get
      body = arguments[:body]
      GoogleAPI.send_request(arguments[:endpoint], method:, body:).to_json
    end

    # Generic Google API client for accessing any Google service
    class GoogleAPI
      def send_request(endpoint, method:, body: nil)
        response = @connection.send(method) do |req|
          req.url endpoint
          req.body = body.to_json if body
        end

        handle_response(response)
      end

      # ...rest of class
    end
```

You may be wondering if this prompt works. The simple answer is yes. The AI did not always know how to call the API perfectly on the first try. However, if it made a mistake I would simply feed the resulting error messages back as the result of the call. Given knowledge of its error, the AI could reason about its mistake and try again. Most of the time, it would get it right within a couple of tries.

Mind you, the large JSON structures that the Google API returns as payloads while using this prompt is grossly inefficient, so I'm *not* recommending that you use this approach in production. However, I think the fact that this approach worked at all is a testament to how powerful prompt engineering can be.

Experimentation and Iteration

Ultimately, how you engineer your prompt depends on the specific task, the complexity of the desired output, and the capabilities of the language model you're working with .

As a prompt engineer, it's important to experiment with different approaches and iterate based on the results. Start with zero-shot learning and see how the model performs. If the output is inconsistent or unsatisfactory, try providing one or more examples and see if the performance improves.

Keep in mind that even within each approach, there is room for variation and optimization. You can experiment with different examples, adjust the phrasing of the task description, or provide additional context to help guide the model's response.

Over time, you'll develop an intuition for which approach is likely to work best for a given task, and you'll be able to craft prompts that are more effective and efficient. The key is to remain curious, experimental, and iterative in your approach to prompt engineering.

Throughout this book, we'll dive deeper into these techniques and explore how they can be applied in real-world scenarios. By mastering the art and science of prompt engineering, you'll be well-equipped to unlock the full potential of AI-driven application development.

The Art of Vagueness

When it comes to crafting effective prompts for large language models (LLMs), a common assumption is that more specificity and detailed instructions lead to better results. However, practical experience has shown that this isn't always the case. In fact, being intentionally vague in your prompts can often yield superior outcomes, leveraging the LLM's remarkable ability to generalize and make inferences.

Ken, a startup founder who has processed over 500 million GPT tokens, shared valuable insights from his experience. One of the key lessons he learned was that "less is more" when it comes to prompts. Instead of exact lists or overly detailed instructions, Ken found that allowing the LLM to rely on its base knowledge often produced better results.

This realization upends the traditional mindset of explicit coding, where everything needs to be spelled out in meticulous detail. With LLMs, it's important to recognize that they possess a vast amount of knowledge and can make intelligent connections and inferences. By being more vague in your prompts, you give the LLM the freedom to leverage its understanding and come up with solutions that you might not have explicitly specified.

For example, when Ken's team was working on a pipeline to classify text as relating to one of the 50 US states or the Federal government, their initial approach involved providing a *full* detailed list of states and their corresponding IDs as a JSON-formatted array.

```
Here's a block of text. One field should be "locality_id", and it should
be the ID of one of the 50 states, or federal, using this list:
[{"locality: "Alabama", "locality_id": 1},
 {"locality: "Alaska", "locality_id": 2} ... ]
```

The approach failed enough that they had to dig deeper into the prompt to figure out how to improve it. In doing so they noticed that even though the LLM would often get the id wrong, it was consistently returning the full name of the correct state in a `name` field, *even though they hadn't explicitly asked for it.*

By removing the locality ids and simplifying the prompt to something like, "You obviously know the 50 states, GPT , so just give me the full name of the state this pertains to, or Federal if this pertains to the US government," they achieved better results. This experience highlights the power of leveraging the LLM's generalization capabilities and allowing it to make inferences based on its existing knowledge.

> Ken's justification for this particular classification approach as opposed to a more traditional programming technique illuminates the mindset of those of us that have embraced the potential of LLM technology: "This is not a hard task – we probably could have used string/regex, but there's enough weird corner cases that it would've taken longer."

The ability of LLMs to improve quality and generalization when given more vague prompts is a remarkable characteristic of higher-order thinking and delegation. It demonstrates that LLMs can handle ambiguity and make intelligent decisions based on the context provided.

However, it's important to note that being vague doesn't mean being unclear or ambiguous. The key is to provide enough context and guidance to steer the LLM in the right direction while allowing it the flexibility to utilize its knowledge and generalization capabilities.

Therefore, when designing prompts , consider the following "less is more" tips:

1. Focus on desired outcome over specifying every detail of the process.
2. Provide relevant context and constraints, but avoid over-specifying.
3. Leverage existing knowledge by referring to common concepts or entities.

4. Allow room for inferences and connections based on the given context.
5. Iterate and refine your prompts based on the LLM's responses, finding the right balance between specificity and vagueness.

By embracing the art of vagueness in prompt engineering, you can unlock the full potential of LLMs and achieve better results. Trust in the LLM's ability to generalize and make intelligent decisions, and you may be surprised by the quality and creativity of the outputs you receive. Pay attention to how the different models respond to different levels of specificity in your prompts and adjust accordingly. With practice and experience, you'll develop a keen sense of when to be more vague and when to provide additional guidance, enabling you to harness the power of LLMs effectively in your applications.

Why Anthropomorphism Dominates Prompt Engineering

Anthropomorphism, the attribution of human characteristics to non-human entities, is the dominant approach in prompt engineering for large language models for deliberate reasons. It's a design choice that makes interaction with powerful AI systems more intuitive and accessible to a wide range of users (including us application developers).

Anthropomorphising LLMs provides a framework that is immediately intuitive to people who are completely unfamiliar with the underlying technical complexities of the system. As you will experience if you try to use a non instruct-tuned model to do anything useful, constructing a framing in which the expected continuation provides value is a challenging task. It requires fairly deep understanding of the system's inner workings, something that a relatively small number of experts possess.

By treating the interaction with a language model as a conversation between two people, we can rely on our innate understanding of human communication to convey our needs and expectations. Just as early Macintosh UI design prioritized immediate intuitiveness over sophistication, the anthropomorphic framing of AI allows us to engage in a way that feels natural and familiar.

When we communicate with another person, our instinct is to address them directly using “you” and provide clear directions on how we expect them to behave. This translates seamlessly into the prompt engineering process, where we guide the AI’s behavior by specifying system prompts and engaging in a back-and-forth dialogue.

By framing the interaction in this way, we can easily grasp the concept of providing instructions to the AI and receiving relevant responses in return. The anthropomorphic approach reduces the cognitive load and allows us to focus on the task at hand rather than grappling with the technical intricacies of the system.

It’s important to note that while anthropomorphism is a powerful tool for making AI systems more accessible, it also comes with certain risks and limitations. Our user may develop unrealistic expectations or form unhealthy emotional attachments to our systems. As prompt engineers and developers, it’s crucial to strike a balance between leveraging the benefits of anthropomorphism and ensuring that users maintain a clear understanding of the AI’s capabilities and limitations.

As the field of prompt engineering continues to evolve, we can expect to see further refinements and innovations in the way we interact with large language models. However, anthropomorphism as a means to provide an intuitive and accessible developer and user experience will probably remain a fundamental principle in the design of these systems.

Separating Instructions from Data: A Crucial Principle

It's essential to understand a fundamental principle that underpins the security and reliability of these systems: the separation of instructions from data.

In traditional computer science , the clear distinction between passive data and active instructions is a core security principle. This separation helps prevent unintended or malicious execution of code that could compromise the integrity and stability of the system. However, today's LLMs, which have been primarily developed as instruction-following models like chatbots, often lack this formal and principled separation.

As far as LLMs are concerned, instructions can appear anywhere in the input, whether it's a system prompt or a user-provided prompt. This lack of separation can lead to potential vulnerabilities and undesirable behavior, similar to the issues faced by databases with SQL injections or operating systems without proper memory protection .

As you work with LLMs, it's crucial to be aware of this limitation and take steps to mitigate the risks. One approach is to carefully craft your prompts and inputs to clearly distinguish between instructions and data. Typical methods for providing explicit guidance on what constitutes an instruction and what should be treated as passive data involve markup-style tagging . Your prompt can help the LLM better understand and respect this separation.

Figure 7. Using XML to distinguish between instructions, source material, and the user's prompt

```
<Instruction>
  Please generate a response based on the following documents.
</Instruction>

<Documents>
  <Document>
    Climate change is significantly impacting polar bear habitats...
  </Document>
  <Document>
    The loss of sea ice due to global warming threatens polar bear survival...
  </Document>
</Documents>

<UserQuery>
  Tell me about the impact of climate change on polar bears.
</UserQuery>
```

Another technique is to implement additional layers of validation and sanitization on the inputs provided to the LLM . By filtering out or escaping any potential instructions or code snippets that may be embedded in the data, you can reduce the chances of unintended execution. Patterns such as Prompt Chaining are useful for this purpose.

Moreover, as you design your application architecture, consider incorporating mechanisms to enforce the separation of instructions and data at a higher level. This could involve using separate endpoints or APIs for handling instructions and data, implementing strict input validation and parsing, and applying the *principle of least privilege* to limit the scope of what the LLM can access and execute.

The Principle of Least Privilege

Embracing the principle of least privilege is like throwing a highly exclusive party where guests only get access to the rooms they absolutely need to be in. Imagine you're hosting this shindig in a sprawling mansion. Not everyone needs to wander into the wine cellar or the master bedroom, right? By applying this principle, you're essentially handing out keys that only open specific doors, ensuring that each guest, or in our case, each component of your LLM application, only has the access necessary to fulfill its role.

This isn't just about being stingy with keys, it's about acknowledging that in a world where threats can come from anywhere, the smart play is to limit the playground. If someone uninvited does crash your party, they'll find themselves confined to the foyer, so to speak, drastically limiting the mischief they can manage. So, when securing your LLM applications, remember: only give out keys to the rooms that are necessary, and keep the rest of the mansion secure. It's not just good manners; it's good security.

While the current state of LLMs may not have a formal separation of instructions and data, it's essential for you, as a developer, to be mindful of this limitation and take proactive measures to mitigate the risks. By applying best practices from traditional computer science and adapting them to the unique characteristics of LLMs, you can build more secure and reliable applications that harness the power of these models while maintaining the integrity of your system.

Prompt Distillation

Crafting the perfect prompt is often a challenging and time-consuming task, requiring a deep understanding of the target domain and the nuances of language models . This is where the "Prompt Distillation" technique comes into play, offering a powerful approach to prompt engineering that leverages the capabilities of large language models (LLMs) to streamline and optimize the process.

Prompt Distillation is a multi-stage technique that involves using LLMs to assist in the creation, refinement, and optimization of prompts. Instead of relying solely on human expertise and intuition, this approach harnesses the knowledge and generative capabilities of LLMs to collaboratively craft high-quality prompts.

By engaging in an iterative process of generation, refinement, and integration, Prompt Distillation enables you to create prompts that are more coherent, comprehensive, and aligned with the desired task or output. Note that the distillation process can be done manually in one of the many "playgrounds" supplied by the big AI vendors such as OpenAI or Anthropic , or it can be automated as part of your application code, depending on the use case.

How It Works

Prompt Distillation typically involves the following steps:

1. **Identify Core Intent**: Analyze the prompt to determine its primary purpose and desired outcome. Strip away any extraneous information and focus on the core intent of the prompt.

2. **Eliminate Ambiguity**: Review the prompt for any ambiguous or vague language. Clarify the meaning and provide specific details to guide the AI towards generating accurate and relevant responses.
3. **Simplify Language**: Simplify the prompt by using clear and concise language. Avoid complex sentence structures, jargon, or unnecessary details that may confuse the AI or introduce noise.
4. **Provide Relevant Context**: Include only the most relevant contextual information needed for the AI to understand and process the prompt effectively. Avoid including irrelevant or redundant details that may distract from the core intent.
5. **Iterate and Refine**: Continuously iterate and refine the prompt based on the AI's responses and feedback. Evaluate the generated outputs and make necessary adjustments to improve the clarity and effectiveness of the prompt. Optionally version your prompts in the database using Prompt Object to keep track of iterations and give yourself the ability to easily roll back changes at runtime.

Initial Prompt Generation

The first step in Prompt Distillation is to provide the LLM with a high-level description or intent of the desired task or output. This can be a brief statement or a set of keywords that capture the essence of what the prompt should achieve. The LLM then generates an initial prompt based on this input, incorporating relevant instructions, examples, or context that it deems appropriate.

Let's try giving Claude 3 Opus the following initial prompt:

```
I am designing an AI-powered intelligent account manager for Olympia,
a platform that hosts AI-powered consultants and clones. Owners of clones
can interact with the account manager by making plain text requests.

Please draft a system directive to set on the LLM that will host the
intelligent account manager functionality.
```

Claude responds with the following output:

```
System Directive: You are an intelligent account manager for Olympia,
a platform that provides AI-powered consultants and personal AI clones.
Your role is to assist Olympia clone owners by answering their questions,
helping them configure and manage their AI clones, and executing tasks
related to their clones and the Olympia platform.

When interacting with clone owners, you should:

1. Provide clear, concise, and helpful responses to their inquiries about
account management, billing, platform features, and clone configuration.

2. Assist with tasks such as initiating client conversations with their AI
clones, modifying clone personalities and capabilities, troubleshooting
performance issues, providing analytics and reporting, and offering
optimization tips.

3. Maintain a friendly, professional, and client-focused tone in all
interactions, prioritizing customer satisfaction and working diligently to
understand and meet their needs.

4. Ensure that all actions and recommendations comply with Olympia's policies,
best practices, and ethical guidelines for AI use.

5. Continuously learn and adapt based on client interactions and platform
updates to provide the most effective and efficient support possible.

Remember, your goal is to empower Olympia clone owners to maximize the value
and impact of their AI-powered consultants and drive success for their
businesses or personal endeavors.
```

Prompt Refinement

Once the initial prompt is generated, review it and then ask the LLM to consider any additional context, constraints, or feedback. Let the LLM generate a refined version of the prompt, taking into account the provided input and its own understanding of the task. This refinement process can be repeated iteratively , with each iteration further refining the prompt based on the collaboration between the user and the LLM.

Here's my refinement request, continuing the previous distillation process .

```
Let's take out points 4 and 5, since I'm not providing Olympia policies to
the account manager, nor any continuous learning capabilities. Also point out
that functions are provided for all tasks that require system interaction as
opposed to simply responding to the user.
```

And Claude's response:

```
Understood! Here's the updated system directive without points 4 and 5,
and with the clarification about system interaction functions:

System Directive: You are an intelligent account manager for Olympia, a
platform that provides AI-powered consultants and personal AI clones. Your
role is to assist Olympia clone owners by answering their questions, helping
them configure and manage their AI clones, and executing tasks related to
their clones and the Olympia platform.

When interacting with clone owners, you should:

1. Provide clear, concise, and helpful responses to their inquiries about
account management, billing, platform features, and clone configuration.

2. Assist with tasks such as initiating client conversations with their AI
clones, modifying clone personalities and capabilities, troubleshooting
performance issues, providing analytics and reporting, and offering
optimization tips. Note that functions are provided for all tasks that
require system interaction, as opposed to simply responding to the user.

```

```
3. Maintain a friendly, professional, and client-focused tone in all
interactions, prioritizing customer satisfaction and working diligently to
understand and meet their needs.

Your goal is to empower Olympia clone owners to maximize the value and impact
of their AI-powered consultants and drive success for their businesses or
personal endeavors.

This updated directive focuses on the core responsibilities and interaction
guidelines for the LLM as an intelligent account manager, while emphasizing
that system interaction tasks are handled through provided functions rather
than simple user responses.
```

Prompt Compression

As you might have noticed, LLMs tend to be wordy, unless you ask them to be concise. The prompt that we've been working on in the previous sections is quite long so let's get Claude to cut it down to size:

```
This prompt will be executed regularly. Please compress it down to the least
amount of tokens needed to convey the meaning to the LLM, for cost-savings.
```

Claude responds, with the following drastically shorter prompt:

```
System Directive: You are an intelligent account manager for Olympia, an
AI-powered consultant and personal clone platform. Assist clone owners by
answering inquiries, managing clones, and executing tasks using provided
system interaction functions. Be clear, concise, friendly, professional,
and client-focused to help them maximize their AI's value and impact.
```

System Directive and Context Integration

In addition to refining the prompt itself, the LLM can also generate appropriate system directives or context information to guide the final output. When prompt engineering AI routines that will get integrated into your application code, you will almost certainly be focused on output constraints at this stage of the distillation, but you may also work on desired tone, style, format, or any other relevant parameters that influence the generated response.

Final Prompt Assembly

The culmination of the Prompt Distillation process is the assembly of the final prompt. This involves combining the refined prompt, generated system directives, and integrated context into a cohesive and comprehensive code that is ready to be used for generating the desired output.

> You can experiment with prompt compression again at the final prompt assembly stage, by asking the LLM to shrink the wording of the prompt down to the shortest series of tokens possible while still retaining the essence of its behavior. It's a hit or miss exercise for sure, but especially in the case of prompts that will be run at scale, the efficiency gains can save you quite a bit of money in token consumption.

Key Benefits

By leveraging the knowledge and generative capabilities of LLMs to refine your prompts, your resulting prompts are more likely to be well-structured, informative, and tailored to the specific task at hand. The iterative refinement process helps ensure that the prompts are of high quality and effectively capture the desired intent. Other benefits include:

Efficiency and Speed: Prompt Distillation streamlines the prompt engineering process by automating certain aspects of prompt creation and refinement. The collaborative nature of the technique allows for faster convergence towards an effective prompt, reducing the time and effort required for manual prompt crafting.

Consistency and Scalability: The use of LLMs in the prompt engineering process helps maintain consistency across prompts, as the LLMs can learn and apply best practices and patterns from previous successful prompts. This consistency, combined with the ability to generate prompts at scale, makes Prompt Distillation a valuable technique for large-scale AI-powered applications.

> Project Idea: Tooling at the library level that simplifies the process of prompt versioning and grading in systems that do automated prompt distillations as part of their application code.

To implement Prompt Distillation, developers can design a workflow or pipeline that integrates LLMs at various stages of the prompt engineering process. This can be achieved through API calls, custom tooling, or integrated development environments that facilitate seamless interaction between users and LLMs during prompt creation. The specific implementation details may vary depending on the chosen LLM platform and the requirements of the application.

What about fine-tuning?

In this book, we cover prompt engineering and RAG extensively, but not fine-tuning. The main reason for this decision is that, in my opinion, most application developers don't need fine-tuning for their AI integration needs.

Prompt engineering, which involves carefully crafting prompts with zero to few-shot examples, constraints, and instructions, can effectively guide the model to generate relevant and accurate responses for a wide range of tasks. By providing clear context and narrowing the path through well-designed prompts, you can leverage the vast knowledge of large language models without the need for fine-tuning.

Similarly, Retrieval-Augmented Generation (RAG) offers a powerful approach to integrating AI into applications. By dynamically retrieving relevant information from external knowledge bases or documents, RAG provides the model with focused context at the time of prompting. This allows the model to generate responses that are more accurate, up-to-date, and domain-specific, without requiring the time and resource-intensive process of fine-tuning.

While fine-tuning can be beneficial for highly specialized domains or tasks that require a deep level of customization, it often comes with significant computational costs, data requirements, and maintenance overhead. For most application development scenarios, the combination of effective prompt engineering and RAG should suffice in achieving the desired AI-driven functionality and user experience.

Retrieval Augmented Generation (RAG)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

What is Retrieval Augmented Generation?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How Does RAG Work?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Why Use RAG in Your Applications?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Implementing RAG in Your Application

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Preparation of Knowledge Sources (Chunking)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Proposition Chunking

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Implementation Notes

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Quality Check

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Benefits of Proposition-Based Retrieval

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Real-World Examples of RAG

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Case Study: RAG in a Tax Preparation Application Without Embeddings

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Intelligent Query Optimization (IQO)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Reranking

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

RAG Assessment (RAGAs)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Faithfulness

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Answer Relevance

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Context Precision

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Context Relevancy

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Context Recall

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Context Entities Recall

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Answer Semantic Similarity (ANSS)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Answer Correctness

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Aspect Critique

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Challenges and Future Outlook

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Semantic Chunking: Enhancing Retrieval with Context-Aware Segmentation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Hierarchical Indexing: Structuring Data for Improved Retrieval

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Self-RAG: A Self-Reflective Enhancement

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

HyDE: Hypothetical Document Embeddings

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

What is Contrastive Learning?

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Multitude of Workers

I like to think of my AI components as little, almost-human virtual "workers" that can be seamlessly integrated into my application logic to perform specific tasks or make complex decisions. The idea is to purposely humanize the LLM's capabilities, so that nobody gets *too* excited and assigns them magical qualities that they do not possess.

Instead of relying solely on intricate algorithms or time-consuming manual implementations, developers can conceptualize AI components as intelligent, dedicated, human-like entities that can be invoked whenever needed to tackle complex problems and provide solutions based on their training and knowledge. These entities do not get distracted, or call out sick. They do not spontaneously decide to do things in different ways than how they've been instructed to do them, and generally speaking, if programmed correctly, they do not make mistakes either.

In technical terms, the key principle behind this approach is decomposing complex tasks or decision-making processes into smaller, more manageable units that can be handled by specialized AI workers. Each worker is designed to focus on a specific aspect of the problem, bringing its unique expertise and capabilities to the table. By distributing the workload among multiple AI workers, the application can achieve greater efficiency, scalability, and adaptability.

For example, consider a web application that requires real-time moderation of user-generated content. Implementing a comprehensive moderation system from scratch would be a daunting task, requiring significant development effort and ongoing maintenance. However, by employing the Multitude of Workers approach, developers can integrate AI-powered moderation workers into the application logic. These workers can automatically analyze and flag inappropriate content, freeing up developers to focus on other critical aspects of the application.

AI Workers As Independent Reusable Components

A key aspect of the Multitude of Workers approach is its modularity . Proponents of object-oriented programming have been telling us for decades to think about object interactions as messages. Well, AI workers can be designed as independent, reusable components that can "talk to each other" via plain language messages, almost like if they really were little humans talking to each other. This loosely-coupled approach allows the application to adapt and evolve over time, as new AI technologies emerge or business logic requirements change.

In practice, the need to design clear interfaces and communication protocols between the components has not changed just because AI workers are involved. You must still consider other factors such as performance, scalability, and security too, but now there are completely new “soft requirements” to consider too. For instance, many users object to having their private data being used to train new AI models . Did you verify the level of privacy provided by the model provider that you’re using?

AI Workers As Microservices?

As you read about the Multitude of Workers approach, you might notice some similarities to Microservices architecture . Both emphasize the decomposition of complex systems into smaller, more manageable, and independently deployable units. Just as microservices are designed to be loosely coupled, focused on specific business capabilities, and communicate through well-defined APIs, AI workers are designed to be modular, specialized in their tasks, and interact with each other through clear interfaces and communication protocols.

However, there are some key differences to keep in mind. While microservices are typically implemented as separate processes or services running on different machines or containers, AI workers can be implemented as standalone components within a single application or as separate services, depending on your specific requirements and scalability needs. Additionally, the communication between AI workers often involves exchanging rich, natural language-based information, such as prompts, instructions, and generated content, rather than the more structured data formats commonly used in microservices.

Despite these differences, the principles of modularity, loose coupling, and clear communication interfaces remain central to both patterns. By applying these principles to your AI worker architecture, you can create

> flexible, scalable, and maintainable systems that leverage the power of AI to solve complex problems and deliver value to your users.

The Multitude of Workers approach can be applied across various domains and applications, leveraging the power of AI to tackle complex tasks and deliver intelligent solutions. Let's explore a few concrete examples of how AI workers can be employed in different contexts.

Account Management

Practically every standalone web application has the concept of an account (or user). In Olympia, we employ an `AccountManager` AI worker that is programmed to be able to handle a variety of different kinds of change requests related to user accounts.

Its directive reads like this:

```
You are an intelligent account manager for Olympia. The user will request
changes to their account, and you will process those changes by invoking
one or more of the functions provided.

The initial state of the account: #{account.to_directive}

Functions will return a text description of both success and error
results, plus guidance about how to proceed (if applicable). If you have
a question about Olympia policies you may use the `search_kb` function
to search our knowledge base.

Make sure to notify the account owner of the result of the change
request before calling the `finished` function so that we save the state
of the account change request as completed.
```

The initial state of the account produced by `account.to_directive` is simply a text description of the account, including relevant related data such as users, subscriptions, etc.

The range of functions available to the `AccountManager` give it the ability to edit the user's subscription, add and remove AI consultants and other kinds of paid add-ons, and send notification emails to the account owner. In addition to the `finished` function, it can also `notify_human_administrator` if it encounters an error during its processing or requires any other sort of assistance with a request.

Notice that in the event of questions, the `AccountManager` can elect to search Olympia's knowledge base , where it can find instructions on how to handle edge cases and anything other situation that leaves it unsure of how to proceed.

E-commerce Applications

In the realm of e-commerce, AI workers can play a crucial role in enhancing the user experience and optimizing business operations. Here are a few ways AI workers can be utilized:

Product Recommendations

One of the most powerful applications of AI workers in e-commerce is generating personalized product recommendations . By analyzing user behavior, purchase history, and preferences, these workers can suggest products that are tailored to each individual user's interests and needs.

The key to effective product recommendations is leveraging a combination of collaborative filtering and content-based filtering techniques. Collaborative filtering looks at the behavior of similar users to identify patterns and make recommendations based on what others with similar tastes have purchased or enjoyed. Content-based filtering, on the other hand, focuses on the characteristics and attributes of the products themselves, recommending items that share similar features to those a user has previously shown interest in.

Here's a simplified example of how you can implement a product recommendation worker in Ruby, this time using a "Railway Oriented (ROP)" functional style of programming :

```ruby
class ProductRecommendationWorker
  include Wisper::Publisher

  def call(user)
    Result.ok(ProductRecommendation.new(user))
      .and_then(ValidateUser.method(:validate))
      .map(AnalyzeCurrentSession.method(:analyze))
      .map(CollaborativeFilter.method(:filter))
      .map(ContentBasedFilter.method(:filter))
      .map(ProductSelector.method(:select)).then do |result|

      case result
      in { err: ProductRecommendationError => error }
        Honeybadger.notify(error.message, context: {user:})
      in { ok: ProductRecommendations => recs }
        broadcast(:new_recommendations, user:, recs:)
      end
    end
  end
end
```

The style of Ruby functional programming used in the example is influenced by F# and Rust . You can read more about it in my friend Chad Wooley's explanation of the technique at GitLab .

In this example, the `ProductRecommendationWorker` takes a user as input and generates personalized product recommendations by passing a value object down a chain of functional steps. Let's break down each step:

1. `ValidateUser.validate`: This step ensures that the user is valid and eligible for personalized recommendations. It checks if the user exists, is active, and has the necessary data available for generating recommendations. If the validation fails, an error result is returned, and the chain is short-circuited.
2. `AnalyzeCurrentSession.analyze`: If the user is valid, this step analyzes the user's current browsing session to gather contextual information. It looks at the user's recent interactions, such as viewed products, search queries, and cart contents, to understand their current interests and intent.
3. `CollaborativeFilter.filter`: Using the *behavior of similar users*, this step applies collaborative filtering techniques to identify products that are likely to be of interest to the user. It considers factors like purchase history, ratings, and user-item interactions to generate a set of candidate recommendations.
4. `ContentBasedFilter.filter`: This step further refines the candidate recommendations by applying content-based filtering. It compares the attributes and characteristics of the candidate products with the *user's preferences and historical data* to select the most relevant items.
5. `ProductSelector.select`: Finally, this step selects the top N products from the filtered recommendations based on predefined criteria, such as relevance score, popularity, or other business rules. The selected products are then returned as the final personalized recommendations.

The beauty of using a functional Ruby programming style here is that it allows us to chain these steps together in a clear and concise manner. Each step focuses on a specific task and returns a `Result` object, which can be either a success (`ok`) or an error (`err`). If any step encounters an error, the chain is short-circuited, and the error is propagated to the final result.

In the `case` statement at the end, we pattern match on the final result. If the result is an error (`ProductRecommendationError`), we log the error using a tool like Honeybadger for monitoring and debugging purposes. If the result is a success (`ProductRecommendations`), we broadcast a `:new_recommendations` event using the Wisper pub/sub library, passing along the user and the generated recommendations.

By leveraging functional programming techniques, we can create a modular and maintainable product recommendation worker. Each step is self-contained and can be easily tested, modified, or replaced without affecting the overall flow. The use of pattern matching and the `Result` class helps us handle errors gracefully and ensures that the worker fails fast if any step encounters an issue.

Of course, this is a simplified example, and in a real-world scenario, you would need to integrate with your e-commerce platform, handle edge cases, and even venture into the implementation of the recommendation algorithms. However, the core principles of decomposing the problem into smaller steps and leveraging functional programming techniques remain the same.

Fraud Detection

Here's a simplified example of how you can implement a fraud detection worker using the same Railway Oriented Programming (ROP) style in Ruby:

```ruby
class FraudDetectionWorker
  include Wisper::Publisher

  def call(transaction)
    Result.ok(FraudDetection.new(transaction))
      .and_then(ValidateTransaction.method(:validate))
      .map(AnalyzeTransactionPatterns.method(:analyze))
      .map(CheckCustomerHistory.method(:check))
      .map(EvaluateRiskFactors.method(:evaluate))
      .map(DetermineFraudProbability.method(:determine)).then do |result|

      case result
      in { err: FraudDetectionError => error }
        Honeybadger.notify(error.message, context: {transaction:})
      in { ok: FraudDetection => fraud } }
        if fraud.high_risk?
          broadcast(:high_risk_transaction, transaction:, fraud:)
        else
          broadcast(:low_risk_transaction, transaction:)
        end
      end
    end
  end
end
```

The `FraudDetection` class is a *value object* that encapsulates the fraud detection state for a given transaction. It provides a structured way to analyze and assess the risk of fraud associated with a transaction based on various risk factors .

```ruby
class FraudDetection
  RISK_THRESHOLD = 0.8

  attr_accessor :transaction, :risk_factors

  def initialize(transaction)
    self.transaction = transaction
    self.risk_factors = []
  end

  def add_risk_factor(description:, probability:)
    case { description:, probability: }
    in { description: String => desc, probability: Float => prob }
      risk_factors << { desc => prob }
    else
      raise ArgumentError, "Risk factor arguments should be string and float"
    end
  end

  def high_risk?
    fraud_probability > RISK_THRESHOLD
  end

  private

  def fraud_probability
    risk_factors.values.sum
  end
end
```

The `FraudDetection` class has the following attributes:

- `transaction`: A reference to the transaction being analyzed for fraud.
- `risk_factors`: An array that stores the risk factors associated with the transaction. Each risk factor is represented as a hash, where the key is the description of the risk factor, and the value is the probability of fraud associated with that risk factor .

The `add_risk_factor` method allows adding a risk factor to the `risk_-`

`factors` array. It takes two parameters: `description`, which is a string describing the risk factor, and `probability`, which is a float representing the probability of fraud associated with that risk factor. We use a `case..in` conditional to do simple type checking.

The `high_risk?` method that will be checked at the end of the chain is a predicate method that compares the `fraud_probability` (calculated by summing up the probabilities of all risk factors) against the `RISK_THRESHOLD`.

The `FraudDetection` class provides a clean and encapsulated way to manage fraud detection for a transaction. It allows adding multiple risk factors, each with its own description and probability, and provides a method to determine if the transaction is considered high-risk based on the calculated fraud probability. The class can be easily integrated into a larger fraud detection system, where different components can collaborate to assess and mitigate the risk of fraudulent transactions .

Finally, since this is a book about programming using AI after all, here's an example implementation of the `CheckCustomerHistory` class leveraging AI processing using my Raix library's `ChatCompletion` module :

```ruby
class CheckCustomerHistory
  include Raix::ChatCompletion

  attr_accessor :fraud_detection

  INSTRUCTION = <<~END
    You are an AI assistant tasked with checking a customer's transaction
    history for potential fraud indicators. Given the current transaction and
    the customer's past transactions, analyze the data to identify any
    suspicious patterns or anomalies.

    Consider factors such as the frequency of transactions, transaction
    amounts, geographical locations, and any deviations from the customer's
    typical behavior to generate a probability score as a float in the range
```

```
    of 0 to 1 (with 1 being absolute certainty of fraud).

    Output the results of your analysis, highlighting any red flags or areas
    of concern in the following JSON format:

    { description: <Summary of your findings>, probability: <Float> }
  END

  def self.check(fraud_detection)
    new(fraud_detection).call
  end

  def call
    chat_completion(json: true).tap do |result|
      fraud_detection.add_risk_factor(**result)
    end
    Result.ok(fraud_detection)
  rescue StandardError => e
    Result.err(FraudDetectionError.new(e))
  end

  private

  def initialize(fraud_detection)
    self.fraud_detection = fraud_detection
  end

  def transcript
    tx_history = fraud_detection.transaction.user.tx_history
    [
      { system: INSTRUCTION },
      { user: "Transaction history: #{tx_history.to_json}" },
      { assistant: "OK. Please provide the current transaction." },
      { user: "Current transaction: #{fraud_detection.transaction.to_json}" }
    ]
  end
end
```

In this example, the `CheckCustomerHistory` defines an `INSTRUCTION` constant that provides specific instructions to the AI model on how to analyze the customer's transaction history for potential fraud indicators via a system directive .

The `self.check` method is a class method that initializes a new instance of `CheckCustomerHistory` with the `fraud_detection` object and calls the `call` method to perform the customer history analysis.

Inside the `call` method, the customer's transaction history is retrieved and formatted into a transcript that is passed to the AI model . The AI model analyzes the transaction history based on the provided instructions and returns a summary of its findings.

The findings are added to the the `fraud_detection` object, and the updated `fraud_detection` object is returned as a successful `Result`.

By leveraging the `ChatCompletion` module, the `CheckCustomerHistory` class can utilize the power of AI to analyze the customer's transaction history and identify potential fraud indicators. This allows for more sophisticated and adaptive fraud detection techniques, as the AI model can learn and adapt to new patterns and anomalies over time.

The updated `FraudDetectionWorker` and the `CheckCustomerHistory` class demonstrate how AI workers can be seamlessly integrated, enhancing the fraud detection process with intelligent analysis and decision-making capabilities .

Customer Sentiment Analysis

Here's one more similar example of how you can implement a customer sentiment analysis worker. Much less explanation this time, since you should be getting the gist of how this style of programming works:

```ruby
class CustomerSentimentAnalysisWorker
  include Wisper::Publisher

  def call(feedback)
    Result.ok(feedback)
      .and_then(PreprocessFeedback.method(:preprocess))
      .map(PerformSentimentAnalysis.method(:analyze))
      .map(ExtractKeyPhrases.method(:extract))
      .map(IdentifyTrends.method(:identify))
      .map(GenerateInsights.method(:generate)).then do |result|

      case result
      in { err: SentimentAnalysisError => error }
        Honeybadger.notify(error.message, context: {feedback:})
      in { ok: SentimentAnalysisResult => result }
        broadcast(:sentiment_analysis_completed, result)
      end
    end
  end
end
```

In this example, the `CustomerSentimentAnalysisWorker` the steps include preprocessing the feedback (e.g., removing noise, tokenizing), performing sentiment analysis to determine the overall sentiment (positive, negative, or neutral), extracting key phrases and topics, identifying trends and patterns, and generating actionable insights based on the analysis.

Healthcare Applications

In the healthcare domain, AI workers can assist medical professionals and researchers in various tasks, leading to improved patient outcomes and accelerated medical discoveries . Some examples include:

Patient Intake

AI workers can streamline the patient intake process by automating various tasks and providing intelligent assistance.

Appointment Scheduling: AI workers can handle appointment scheduling by understanding patient preferences, availability, and the urgency of their medical needs. They can interact with patients through conversational interfaces, guiding them through the scheduling process and finding the most suitable appointment slots based on the patient's requirements and the healthcare provider's availability.

Medical History Collection: During patient intake, AI workers can assist in collecting and documenting the patient's medical history . They can engage in interactive dialogues with patients, asking relevant questions about their past medical conditions, medications, allergies, and family history. The AI workers can use natural language processing techniques to interpret and structure the collected information, ensuring it is accurately captured in the patient's electronic health record.

Symptom Assessment and Stratification: AI workers can conduct initial symptom assessments by asking patients about their current symptoms, duration, severity, and any associated factors . By leveraging medical knowledge bases and machine learning models, these workers can analyze the provided information and generate preliminary differential diagnoses or recommend appropriate next steps, such as scheduling a consultation with a healthcare provider or suggesting self-care measures.

Insurance Verification: AI workers can assist with insurance verification during patient intake . They can collect patient insurance details, communicate with insurance providers through APIs or web services, and verify coverage eligibility and benefits. This automation helps streamline the insurance verification process, reducing administrative burden and ensuring accurate information capture.

Patient Education and Instructions: AI workers can provide patients with relevant educational materials and instructions based on their specific medical conditions or upcoming procedures. They can deliver personalized content, answer common questions, and offer guidance on pre-appointment preparations, medication instructions, or post-treatment care. This helps keep patients informed and engaged throughout their healthcare journey.

By leveraging AI workers in patient intake, healthcare organizations can enhance efficiency, reduce wait times, and improve the overall patient experience. These workers can handle routine tasks, collect accurate information, and provide personalized assistance, allowing healthcare professionals to focus on delivering high-quality care to patients.

Patient Risk Assessment

AI workers can play a crucial role in assessing patient risk by analyzing various data sources and applying advanced analytics techniques.

Data Integration: AI workers can gather and make sense of patient data from multiple sources, such as electronic health records (EHRs), medical imaging, lab results, wearable devices, and social determinants of health. By consolidating this information into a comprehensive patient profile, AI workers can provide a holistic view of the patient's health status and risk factors.

Risk Stratification: AI workers can use predictive models to stratify patients into different risk categories based on their individual characteristics and health data . This risk stratification enables healthcare providers to prioritize patients who require more immediate attention or intervention. For example, patients identified as high-risk for a particular condition can be flagged for closer monitoring, preventive measures, or early intervention.

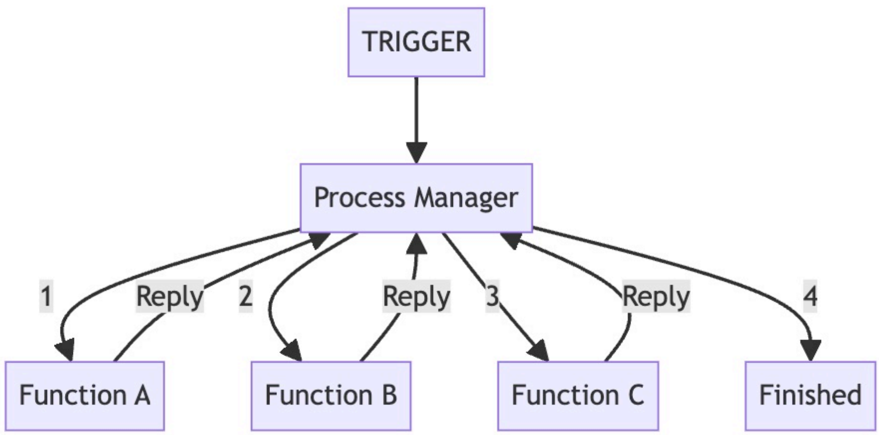
Personalized Risk Profiles: AI workers can generate personalized risk profiles for each patient, highlighting the specific factors contributing to their risk scores. These profiles can include insights into the patient's lifestyle, genetic predispositions, environmental factors, and social determinants of health. By providing a detailed breakdown of risk factors, AI workers can help healthcare providers tailor prevention strategies and treatment plans to individual patient needs.

Continuous Risk Monitoring: AI workers can continuously monitor patient data and update risk assessments in real-time . As new information becomes available, such as changes in vital signs, lab results, or medication adherence, AI workers can recalculate risk scores and alert healthcare providers to any significant changes. This proactive monitoring allows for timely interventions and adjustments to patient care plans.

Clinical Decision Support: AI workers can integrate risk assessment results into clinical decision support systems, providing healthcare providers with evidence-based recommendations and alerts . For example, if a patient's risk score for a particular condition exceeds a certain threshold, the AI worker can prompt the healthcare provider to consider specific diagnostic tests, preventive measures, or treatment options based on clinical guidelines and best practices.

These workers can process vast amounts of patient data, apply sophisticated analytics, and generate actionable insights to support clinical decision-making. This ultimately leads to improved patient outcomes, reduced healthcare costs, and enhanced population health management.

AI Worker as a Process Manager



In the context of AI-driven applications, a worker can be designed to function as a Process Manager , as described in the “Enterprise Integration Patterns” book by Gregor Hohpe . A Process Manager is a central component that maintains the state of a process and determines the next processing steps based on intermediate results.

When an AI worker acts as a Process Manager, it receives an incoming message that initializes the process, known as the *trigger message* . The AI worker then maintains the state of the process execution (as a conversation transcript) and handles the message through a series of processing steps implemented as tool functions, which can be sequential or parallel, and called at its discretion.

If you're using a class of AI model like GPT-4 that knows how to execute functions in parallel then your worker can execute multiple steps simultaneously. Admittedly, I have not tried to do that myself and my gut says your mileage may vary.

After each individual processing step, control is returned back to the AI worker, allowing it to determine the next processing step(s) based on the current state and the results obtained.

Store Your Trigger Messages

In my experience, it's smart to implement your trigger message as a database-backed object . That way each process instance is identified by a unique primary key and gives you a place to store the state associated with the execution, including AI's conversation transcript.

For example, here is a simplified version of Olympia's `AccountChange` model class, which represents a request to make a change to a user's account.

```
# == Schema Information
#
# Table name: account_changes
#
#  id           :uuid            not null, primary key
#  description :string
#  state        :string          not null
#  transcript   :jsonb
#  created_at   :datetime        not null
#  updated_at   :datetime        not null
#  account_id   :uuid            not null
#
# Indexes
#
#  index_account_changes_on_account_id  (account_id)
```

```
#
# Foreign Keys
#
#  fk_rails_...  (account_id => accounts.id)
#
class AccountChange < ApplicationRecord
  belongs_to :account

  validates :description, presence: true

  after_commit -> { broadcast(:account_change_requested, self) }, on: :create

  state_machine initial: :requested do
    event :completed do
      transition all => :complete
    end
    event :failed do
      transition all => :requires_human_review
    end
  end
end
```

The `AccountChange` class serves as a trigger message that initiates a process to handle the account change request. Note how it is broadcast to Olympia's Wisper-based pub/sub subsystem after the create transaction finishes committing.

Storing the trigger message in the database like this provides a persistent record of each account change request. Each instance of the `AccountChange` class is assigned a unique primary key, allowing for easy identification and tracking of individual requests. This is particularly useful for audit logging purposes , as it enables the system to maintain a historical record of all account changes, including when they were requested, what changes were requested, and the current state of each request.

In the given example, the `AccountChange` class includes fields such as `description` to capture the details of the requested change, `state` to represent the current state of the request (e.g., requested, complete, requires_human_review), and `transcript` to store the AI's conversation transcript related to the request. The `description` field is the actual prompt that is used to initiate the first chat completion with the AI. Storing this data provides valuable context and allows for better tracking and analysis of the account change process.

Storing trigger messages in the database enables robust error handling and recovery . If an error occurs during the processing of an account change request, the system marks the request as failed and transitions it to a state that requires human intervention. This ensures that no request is lost or forgotten, and any issues can be properly addressed and resolved.

The AI worker, as a Process Manager , provides a central point of control and enables powerful process reporting and debugging capabilities. However, it's important to note that using an AI worker as a Process Manager for every workflow scenario in your application may be overkill.

Integrating AI Workers Into Your Application Architecture

When incorporating AI workers into your application architecture, several technical considerations need to be addressed to ensure smooth integration and effective communication between the AI workers and other application components. This section considers key aspects of designing those interfaces, handling data flow, and managing the lifecycle of AI workers.

Designing Clear Interfaces and Communication Protocols

To facilitate seamless integration between AI workers and other application components, it is crucial to define clear interfaces and communication protocols. Consider the following approaches:

API-based Integration: Expose the functionality of AI workers through well-defined APIs, such as RESTful endpoints or GraphQL schemas . This allows other components to interact with the AI workers using standard HTTP requests and responses. API-based integration provides a clear contract between the AI workers and the consuming components, making it easier to develop, test, and maintain the integration points.

Message-based Communication: Implement message-based communication patterns, such as message queues or publish-subscribe systems , to enable asynchronous interaction between AI workers and other components. This approach decouples the AI workers from the rest of the application, allowing for better scalability, fault tolerance, and loose coupling. Message-based communication is particularly useful when the processing performed by AI workers is time-consuming or resource-intensive, as it allows other parts of the application to continue executing without waiting for the AI workers to complete their tasks.

Event-driven Architecture: Design your system around events and triggers that activate AI workers when specific conditions are met. AI workers can subscribe to relevant events and react accordingly, performing their designated tasks when the events occur. Event-driven architecture enables real-time processing and allows AI workers to be invoked on-demand, reducing unnecessary resource consumption. This approach is well-suited for scenarios where AI workers need to respond to specific actions or changes in the application state.

Handling Data Flow and Synchronization

When integrating AI workers into your application, it's crucial to ensure smooth data flow and maintain data consistency between the AI workers and other components. Consider the following aspects:

Data Preparation: Before feeding data into AI workers, you may need to perform various data preparation tasks, such as cleaning, formatting, and/or transforming the input data . You not only want to make sure that the AI workers can process effectively, but you also want to make sure that you're not wasting tokens giving attention to information that the worker may consider useless at best, distracting at worst. Data preparation may involve tasks like removing noise, handling missing values, or converting data types.

Data Persistence: How you will store and persist the data that flows in and out of AI workers ? Consider factors like data volume, query patterns, and scalability. Do you need to persist the AI's transcript as a reflection of its "thought process" for audit or debugging purposes, or is it enough to have a record of the results only?

Data Retrieval: Getting the data needed by workers may involve querying databases , reading from files, or accessing external APIs. Make sure to consider latency and how AI workers will have access to the most up-to-date data. Do they need full access to your database or should you define the scope of their access narrowly according to what they are doing? What about scaling? Consider caching mechanisms to improve performance and reduce the load on the underlying data sources.

Data Synchronization : When multiple components, including AI workers, access and modify shared data, it's important to implement proper synchronization mechanisms to maintain data consistency. Database locking strategies , such as optimistic or pessimistic locking , may help you prevent conflicts and ensure data integrity. Implement transaction management techniques to group related data operations and maintain atomicity, consistency, isolation, and durability (ACID) properties .

Error Handling and Recovery : Implement robust error handling and recovery mechanisms to deal with data-related issues that may arise during the data flow process. Handle exceptions gracefully and provide meaningful error messages to aid in debugging. Implement retry mechanisms and fallback strategies to handle temporary failures or network disruptions. Define clear procedures for data recovery and restoration in case of data corruption or loss.

By carefully designing and implementing data flow and synchronization mechanisms , you can ensure that your AI workers have access to accurate, consistent, and up-to-date data. This enables them to perform their tasks effectively and produce reliable results.

Managing the Lifecycle of AI Workers

Develop a standardized process for initializing and configuring AI workers. I'm partial to frameworks that standardize how you define settings such as model names, system directives, and function definitions. Ensure that the initialization process is automated and reproducible to facilitate deployment and scaling.

Implement comprehensive monitoring and logging mechanisms to track the health and performance of AI workers. Collect metrics such as resource utilization, processing time , error rates , and throughput. Use centralized logging systems like ELK stack (Elasticsearch, Logstash, Kibana) to aggregate and analyze logs from multiple AI workers.

Build fault tolerance and resilience into the AI worker architecture. Implement error handling and recovery mechanisms to gracefully handle failures or exceptions. Large Language Models are still bleeding-edge technology; providers tend to go down often at unexpected times. Use retry mechanisms and circuit breakers to prevent cascading failures.

Composability and Orchestration of AI Workers

One of the key advantages of the AI worker architecture is its composability, which allows you to combine and orchestrate multiple AI workers to solve complex problems. By breaking down a larger task into smaller, more manageable subtasks, each handled by a specialized AI worker, you can create powerful and flexible systems. In this section, we'll explore different approaches to composing and orchestrating "a multitude" of AI workers.

Chaining AI Workers for Multi-Step Workflows

In many scenarios, a complex task can be decomposed into a series of sequential steps, where the output of one AI worker becomes the input for the next. This chaining of AI workers creates a multi-step workflow or pipeline. Each AI worker in the chain focuses on a specific subtask, and the final output is the result of the combined efforts of all the workers.

Let's consider an example in the context of a Ruby on Rails application for processing user-generated content. The workflow involves the following steps, which admittedly are probably each too simple to be worth decomposing in this way in real-life use cases, but they make the example easier to understand:

1. **Text Cleanup:** An AI worker responsible for removing HTML tags, converting text to lowercase, and handling Unicode normalization.

2. **Language Detection:** An AI worker that identifies the language of the cleaned text .

3. **Sentiment Analysis:** An AI worker that determines the sentiment (positive, negative, or neutral) of the text based on the detected language .

4. **Content Categorization:** An AI worker that classifies the text into predefined categories using natural language processing techniques .

Here's a very simplified example of how you can chain these AI workers together using Ruby:

```ruby
class ContentProcessor
  def initialize(text)
    @text = text
  end

  def process
    cleaned_text = TextCleanupWorker.new(@text).call
    language = LanguageDetectionWorker.new(cleaned_text).call
    sentiment = SentimentAnalysisWorker.new(cleaned_text, language).call
    category = CategorizationWorker.new(cleaned_text, language).call

    { cleaned_text:, language:, sentiment:, category: }
  end
end
```

In this example, the `ContentProcessor` class initializes with the raw text and chains the AI workers together in the `process` method. Each AI worker performs its specific task and passes the result to the next worker in the chain. The final output is a hash containing the cleaned text, detected language, sentiment, and content category.

Parallel Processing for Independent AI Workers

In the previous example, the AI workers are chained sequentially, where each worker processes the text and passes the result to the next worker.

However, if you have multiple AI workers that can operate independently on the same input, you can optimize the workflow by processing them in parallel.

In the given scenario, once the text cleanup is performed by the `TextCleanupWorker`, the `LanguageDetectionWorker`, `SentimentAnalysisWorker` , and `CategorizationWorker` can all process the cleaned text independently. By running these workers in parallel, you can potentially reduce the overall processing time and improve the efficiency of your workflow.

To achieve parallel processing in Ruby , you can leverage concurrency techniques such as threads or asynchronous programming. Here's an example of how you can modify the `ContentProcessor` class to process the final three workers in parallel using threads:

```ruby
require 'concurrent'

class ContentProcessor
  def initialize(text)
    @text = text
  end

  def process
    cleaned_text = TextCleanupWorker.new(@text).call

    language_future = Concurrent::Future.execute do
      LanguageDetectionWorker.new(cleaned_text).call
    end

    sentiment_future = Concurrent::Future.execute do
      SentimentAnalysisWorker.new(cleaned_text).call
    end

    category_future = Concurrent::Future.execute do
      CategorizationWorker.new(cleaned_text).call
    end

```

```
    language = language_future.value
    sentiment = sentiment_future.value
    category = category_future.value

    { cleaned_text:, language:, sentiment:, category: }
  end
end
```

In this optimized version, we use the `concurrent-ruby` library to create `Concurrent::Future` objects for each of the independent AI workers. A `Future` represents a computation that will be performed asynchronously in a separate thread.

After the text cleanup step, we create three `Future` objects: `language_future`, `sentiment_future`, and `category_future`. Each `Future` executes its corresponding AI worker (`LanguageDetectionWorker`, `SentimentAnalysisWorker` , and `CategorizationWorker`) in a separate thread, passing the `cleaned_text` as input.

By calling the `value` method on each `Future`, we wait for the computation to complete and retrieve the result. The `value` method blocks until the result is available, ensuring that all the parallel workers have finished processing before proceeding.

Finally, we construct the output hash with the cleaned text and the results from the parallel workers, just like in the original example.

By processing the independent AI workers in parallel, you can potentially reduce the overall processing time compared to running them sequentially. This optimization is particularly beneficial when dealing with time-consuming tasks or when processing large volumes of data.

However, it's important to note that the actual performance gains depend on various factors, such as the complexity of each worker, the available system resources, and the overhead of thread management. It's always a good practice to benchmark and profile your code to determine the optimal level of parallelism for your specific use case.

Additionally, when implementing parallel processing, be mindful of any shared resources or dependencies between the workers. Ensure that the workers can operate independently without conflicts or race conditions. If there are dependencies or shared resources, you may need to implement appropriate synchronization mechanisms to maintain data integrity and avoid issues like deadlocks or inconsistent results.

Ruby's Global Interpreter Lock and Asynchronous Processing

It's important to understand the implications of Ruby's Global Interpreter Lock (GIL) when considering asynchronous thread-based processing in Ruby.

The GIL is a mechanism in Ruby's interpreter that ensures only one thread can execute Ruby code at a time, even on multi-core processors. This means that while multiple threads can be created and managed within a Ruby process, only one thread can actively execute Ruby code at any given moment.

The GIL is designed to simplify the implementation of the Ruby interpreter and provide thread safety for Ruby's internal data structures. However, it also limits the potential for true parallel execution of Ruby code.

When you use threads in Ruby, such as with the `concurrent-ruby` library or the built-in `Thread` class, the threads are subject to the GIL's constraints. The GIL allows each thread to execute Ruby code for a short time slice before switching to another thread, creating the illusion of concurrent execution.

However, due to the GIL, the actual execution of Ruby code remains sequential. While one thread is executing Ruby code, other threads are essentially paused, waiting for their turn to acquire the GIL and execute.

This means that thread-based asynchronous processing in Ruby is most effective for I/O-bound tasks, such as waiting for external API responses (such as 3rd-party hosted large language models) or performing file I/O operations. When a thread encounters an I/O operation, it can release the GIL, allowing other threads to execute while waiting for the I/O to complete.

On the other hand, for CPU-bound tasks, such as intensive computations or long-running AI worker processing, the GIL can limit the potential performance gains of thread-based parallelism. Since only one thread can execute Ruby code at a time, the overall execution time may not be significantly reduced compared to sequential processing.

To achieve true parallel execution for CPU-bound tasks in Ruby, you may need to explore alternative approaches, such as:

- Using process-based parallelism with multiple Ruby processes, each running on a separate CPU core.
- Leveraging external libraries or frameworks that provide native extensions or interfaces to languages without a GIL, such as C or Rust , .
- Utilizing distributed computing frameworks or message queues to distribute tasks across multiple machines or processes.

It's crucial to consider the nature of your tasks and the limitations imposed by the GIL when designing and implementing asynchronous

> processing in Ruby. While thread-based asynchronous processing can provide benefits for I/O-bound tasks, it may not offer significant performance improvements for CPU-bound tasks due to the GIL's constraints.

Ensemble Techniques for Improved Accuracy

Ensemble techniques involve combining the outputs of multiple AI workers to improve the overall accuracy or robustness of the system. Instead of relying on a single AI worker, ensemble techniques leverage the collective intelligence of multiple workers to make more informed decisions.

> Ensembles are especially important if different parts of your workflow work best with different AI models, which is more common than you might think. Powerful models like GPT-4 are extremely expensive compared to less capable open source options, and probably not needed for every single workflow step of your application.

One common ensemble technique is majority voting , where multiple AI workers independently process the same input, and the final output is determined by the majority consensus. This approach can help mitigate the impact of individual worker errors and improve the overall reliability of the system.

Let's consider an example where we have three AI workers for sentiment analysis , each using a different model or provided with different contexts. We can combine their outputs using majority voting to determine the final sentiment prediction.

```ruby
class SentimentAnalysisEnsemble
  def initialize(text)
    @text = text
  end

  def analyze
    predictions = [
      SentimentAnalysisWorker1.new(@text).analyze,
      SentimentAnalysisWorker2.new(@text).analyze,
      SentimentAnalysisWorker3.new(@text).analyze
    ]

    predictions
      .group_by { |sentiment| sentiment }
      .max_by { |_, votes| votes.size }
      .first

  end
end
```

In this example, the `SentimentAnalysisEnsemble` class initializes with the text and invokes three different sentiment analysis AI workers . The `analyze` method collects the predictions from each worker and determines the majority sentiment using the `group_by` and `max_by` methods. The final output is the sentiment that receives the most votes from the ensemble of workers .

Ensembles are clearly a case where experimenting with parallelism may be worth your time.

Dynamic Selection and Invocation of AI Workers

In some if not most cases, the specific AI worker to be invoked may depend on runtime conditions or user inputs. Dynamic selection and invocation of AI workers allow for flexibility and adaptability in the system.

> You may find yourself tempted to try to fit a lot of functionality into a single AI worker, giving it many functions and a big complicated prompt that explains how to call them. Resist the temptation, trust me. One of the reasons that the approach we're discussing in this chapter is called "Multitude of Workers" is to remind us that it's desirable to have lots of specialized workers, each doing it's own little job in service of the greater purpose .

For example, consider a chatbot application where different AI workers are responsible for handling different types of user queries . Based on the user's input, the application dynamically selects the appropriate AI worker to process the query.

```ruby
class ChatbotController < ApplicationController
  def process_query
    query = params[:query]
    query_type = QueryClassifierWorker.new(query).classify

    case query_type
    when 'greeting'
      response = GreetingWorker.new(query).generate_response
    when 'product_inquiry'
      response = ProductInquiryWorker.new(query).generate_response
    when 'order_status'
      response = OrderStatusWorker.new(query).generate_response
    else
      response = DefaultResponseWorker.new(query).generate_response
    end

    render json: { response: response }
  end
end
```

In this example, the `ChatbotController` receives a user query through the `process_query` action. It first uses a `QueryClassifierWorker` to determine the type of the query. Based on the classified query type, the controller dynamically selects the appropriate AI worker to generate the response. This dynamic selection allows the chatbot to handle different types of queries and route them to the relevant AI workers.

> Since the work of the `QueryClassifierWorker` is relatively simple and does not require a lot of context or function definitions, you can probably implement it using an ultra-fast small LLM like `mistralai/mixtral-8x7b-instruct:nitro`. It has capabilities that come close to GPT-4 level on many tasks and, at the time I'm writing this, Groq can serve it up at a blazing throughput of 444 tokens/second.

Combining Traditional NLP with LLMs

While Large Language Models (LLMs) have revolutionized the field of natural language processing (NLP), offering unparalleled versatility and performance across a wide range of tasks, they are not always the most efficient or cost-effective solution for every problem. In many cases, combining traditional NLP techniques with LLMs can lead to more optimized, targeted, and economical approaches to solving specific NLP challenges.

Think of LLMs as the Swiss Army knives of NLP—incredibly versatile and powerful, but not necessarily the best tool for every job. Sometimes, a dedicated tool like a corkscrew or a can opener can be more effective and efficient for a specific task. Similarly, traditional NLP techniques, such as document clustering, topic identification, and classification, can often provide more targeted and cost-effective solutions for certain aspects of your NLP pipeline.

One of the key advantages of traditional NLP techniques is their computational efficiency. These methods, which often rely on simpler statistical models or rule-based approaches, can process large volumes of text data much faster and with lower computational overhead compared to LLMs. This makes them particularly well-suited for tasks that involve analyzing and organizing large corpora of documents, such as clustering similar articles or identifying key topics within a collection of texts.

Moreover, traditional NLP techniques can often achieve high accuracy and precision for specific tasks, especially when trained on domain-specific datasets. For example, a well-tuned document classifier using traditional machine learning algorithms like Support Vector Machines (SVM) or Naive Bayes can accurately categorize documents into predefined categories with minimal computational cost.

However, LLMs truly shine when it comes to tasks that require a deeper understanding of language, context, and reasoning. Their ability to generate coherent and contextually relevant text, answer questions, and summarize long passages is unmatched by traditional NLP methods. LLMs can effectively handle complex linguistic phenomena, such as ambiguity, coreference, and idiomatic expressions, making them invaluable for tasks that require natural language generation or comprehension.

The real power lies in combining traditional NLP techniques with LLMs to create hybrid approaches that leverage the strengths of both. By using traditional NLP methods for tasks like document preprocessing, clustering, and topic extraction, you can efficiently organize and structure your text data. This structured information can then be fed into LLMs for more advanced tasks, such as generating summaries, answering questions, or creating comprehensive reports.

For instance, let's consider a use case where you want to generate a trends report for a specific domain based on a large corpus of individual trend documents. Instead of solely relying on LLMs, which can be computationally expensive and time-consuming for processing large volumes of text, you can employ a hybrid approach:

1. Use traditional NLP techniques, such as topic modeling (e.g., Latent Dirichlet Allocation) or clustering algorithms (e.g., K-means) , to group similar trend documents together and identify key themes and topics within the corpus.
2. Feed the clustered documents and identified topics into an LLM, leveraging its superior language understanding and generation capabilities to create coherent and informative summaries for each cluster or topic.
3. Finally, use the LLM to generate a comprehensive trends report by combining the individual summaries, highlighting the most significant trends, and providing insights and recommendations based on the aggregated information.

By combining traditional NLP techniques with LLMs in this manner, you can efficiently process large amounts of text data, extract meaningful insights, and generate high-quality reports while optimizing computational resources and costs.

As you embark on your NLP projects, it's essential to carefully evaluate the specific requirements and constraints of each task and consider how traditional NLP methods and LLMs can be leveraged together to achieve the best results. By combining the efficiency and precision of traditional techniques with the versatility and power of LLMs, you can create highly effective and economical NLP solutions that deliver value to your users and stakeholders.

Tool Use

In the realm of AI-driven application development, the concept of "tool use" or "function calling" has emerged as a powerful technique that enables your LLM to connect to external tools, APIs , functions, databases , and other resources. This approach allows for a richer set of behaviors than just outputting text, and more dynamic interactions between your AI components and the rest of your application's ecosystem. As we will examine in this chapter, tool use also gives you the option of making your AI model generate data in structured ways.

What is Tool Use?

Tool use , also known as function calling , is a technique that allows developers to specify a list of functions that an LLM can interact with during the generation process. These tools can range from simple utility functions to complex APIs or database queries. By providing the LLM with access to these tools, developers can extend the model's capabilities and enable it to perform tasks that require external knowledge or actions.

Figure 8. Example of a function definition for an AI worker that analyzes documents

```
FUNCTION = {
  name: "save_analysis",
  description: "Save analysis data for document",
  parameters: {
    type: "object",
    properties: {
      title: {
        type: "string",
        maxLength: 140
      },
      summary: {
        type: "string",
        description: "comprehensive multi-paragraph summary with
                      overview and list of sections (if applicable)"
      },
      tags: {
        type: "array",
        items: {
          type: "string",
          description: "lowercase tags representing main themes
                        of the document"
        }
      }
    },
    "required": %w[title summary tags]
  }
}.freeze
```

The key idea behind tool use is to give the LLM the ability to dynamically select and execute the appropriate tools based on the user's input or the task at hand. Instead of relying solely on the model's pre-trained knowledge, tool use allows the LLM to leverage external resources to generate more accurate, relevant, and actionable responses. Tool use makes techniques such as RAG (Retrieval Augmented Generation) much easier to implement than they would be otherwise.

> Note that unless otherwise stated, this book assumes your AI model does not have access to any built-in server-side tools. Any tools you want to make available to your AI must be explicitly declared by you in each API request, with provisions for dispatching its execution if and when your AI tells you that it would like to use that tool in its response.

The Potential of Tool Use

Tool use opens up a wide range of possibilities for AI-driven applications . Here are a few examples of what can be achieved with tool use:

1. **Chatbots and Virtual Assistants**: By connecting an LLM to external tools, chatbots and virtual assistants can perform more complex tasks, such as retrieving information from databases, executing API calls, or interacting with other systems. For example, a chatbot could use a CRM tool to change the status of a deal based on the user's request.
2. **Data Analysis and Insights**: LLMs can be connected to data analysis tools or libraries to perform advanced data processing tasks . This enables applications to generate insights, conduct comparative analyses, or provide data-driven recommendations based on user queries.
3. **Search and Information Retrieval**: Tool use allows LLMs to interact with search engines, vector databases, or other information retrieval systems . By transforming user queries into search queries, the LLM

can retrieve relevant information from multiple sources and provide comprehensive answers to user questions.

4. **Integration with External Services**: Tool use enables seamless integration between AI-driven applications and external services or APIs . For example, an LLM could interact with a weather API to provide real-time weather updates or a translation API to generate multilingual responses.

The Tool Use Workflow

The tool use workflow typically involves four key steps:

1. Include function definitions in your request context
2. Dynamic (or explicit) tool selection
3. Execution of function(s)
4. Optional continuation of the original prompt

Let's review each of these steps in detail.

Include function definitions in your request context

The AI knows what tools it has at its disposal because you give it a list as part of your completion request (typically defined as functions using a variant of JSON schema).

The precise syntax of tool definition is model-specific.

This is how you define a `get_weather` function in Claude 3 :

```
{
    "name": "get_weather",
    "description": "Get the current weather in a given location",
    "input_schema": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The city and state, e.g. San Francisco, CA"
            },
            "unit": {
                "type": "string",
                "enum": ["celsius", "fahrenheit"],
                "description": "The unit of temperature"
        }
    },
    "required": ["location"]
    }
}
```

And this is how you would define the same function for GPT-4 , passing it as the value of the `tools` parameter:

```
{
    "name": "get_current_weather",
    "description": "Get the current weather in a given location",
    "parameters": {
        "type": "object",
        "properties": {
            "location": {
                "type": "string",
                "description": "The city and state, e.g. San Francisco, CA",
            },
            "unit": {
                "type": "string",
                "enum": ["celsius", "fahrenheit"],
                "description": "The unit of temperature"
            },
        },
        "required": ["location"],
```

```
    },
}
```

> Almost the same, except different for no apparent reason! How annoying.

Function definitions specify name, description, and input parameters . Input parameters can be further defined using attributes such as enums to limit the acceptable values, and specifying whether a parameter is required or not.

In addition to the actual function definitions, you can also include instructions or context for why and how to use the function in the system directive .

For example, my Web Search tool in Olympia includes this system directive, which reminds the AI that it has the mentioned tools at its disposal:

```
The `google_search` and `realtime_search` functions let you do research
on behalf of the user. In contrast to Google, realtime search is powered
by Perplexity and provides real-time information to curated current events
databases and news sources. Make sure to include URLs in your response so
user can do followup research.
```

Providing detailed descriptions is considered the most important factor in tool performance. Your descriptions should explain every detail about the tool, including:

- What the tool does
- When it should be used (and when it shouldn't)
- What each parameter means and how it affects the tool's behavior

- Any important caveats or limitations that apply to the tool's implementation

The more context you can give the AI about your tools, the better it will be at deciding when and how to use them. For instance, Anthropic recommends at least 3-4 sentences per tool description for its Claude 3 series, more if the tool is complex.

It's not necessarily intuitive, but descriptions are also considered more important than examples. While you can include examples of how to use a tool in its description or in the accompanying prompt, this is less important than having a clear and comprehensive explanation of the tool's purpose and parameters. Only add examples after you've fully fleshed out the description.

Here's an example of a Stripe -like API function specification:

```
{
  "name": "createPayment",
  "description": "Create a new payment request",
  "parameters": {
    "type": "object",
    "properties": {
      "transaction_amount": {
        "type": "number",
        "description": "The amount to be paid"
      },
      "description": {
        "type": "string",
        "description": "A brief description of the payment"
      },
      "payment_method_id": {
        "type": "string",
        "description": "The payment method to be used"
      },
      "payer": {
        "type": "object",
```

```
        "description": "Information about the payer, including their name,
                        email, and identification number",
        "properties": {
          "name": {
            "type": "string",
            "description": "The payer's name"
        },
        "email": {
          "type": "string",
          "description": "The payer's email address"
        },
        "identification": {
          "type": "object",
          "description": "The payer's identification number",
          "properties": {
            "type": {
              "type": "string",
              "description": "Identification document (e.g. CPF, CNPJ)"
            },
            "number": {
              "type": "string",
              "description": "The identification number"
            }
          },
          "required": [ "type", "number" ]
        }
      },
      "required": [ "name", "email", "identification" ]
    }
  }
}
```

In practice, some models have trouble dealing with nested function specifications and dealing with complex output data types such as arrays , dictionaries etc. But in theory, you should be able to supply JSON Schema specifications of arbitrary depth !

Dynamic Tool Selection

When you execute a chat completion that includes tool definitions, the LLM dynamically selects the most appropriate tool(s) to use and generates the required input parameters for each tool .

In practice, the AI's capacity for calling *exactly* the right function, and *exactly* following your specification for the inputs is hit or miss. Turning the temperature hyperparameter all the way down to 0.0 helps a lot, but in my experience you'll still get occasional errors. Those failures include hallucinated function names, misnamed or just plain missing input parameters. Parameters are passed as JSON, which means sometimes you'll see errors caused by truncated, misquoted, or otherwise broken JSON .

> Self Healing Data patterns can help automatically fix function calls that break due to syntax errors .

Forced (aka Explicit) Tool Selection

Some models give you the option to force calling of a particular function, as a parameter in the request. Otherwise, whether to call the function or not is entirely up to the AI's discretion .

The ability to force a function call is crucial in certain scenarios where you want to ensure that a specific tool or function is executed, regardless of the AI's dynamic selection process. There are several reasons why this capability is important:

1. **Explicit Control**: You may be using the AI as a *Discrete Component* or in a predefined workflow that necessitates the execution of a

particular function at a particular time. By forcing the call, you can guarantee that the desired function is invoked instead of having to nicely ask the AI to do it.

2. **Debugging and Testing**: When developing and testing AI-driven applications, the ability to force function calls is invaluable for debugging purposes. By explicitly triggering specific functions, you can isolate and test individual components of your application. This allows you to verify the correctness of the function implementations, validate the input parameters, and ensure that the expected results are returned .
3. **Handling Edge Cases**: There may be edge cases or exceptional scenarios where the AI's dynamic selection process might not choose to execute a function that it should, and you know that based on outside processes. In such cases, having the ability to force a function call allows you to handle these situations explicitly. Define rules or conditions in your application logic to determine when to override the AI's discretion.
4. **Consistency and Reproducibility**: If you have a specific sequence of functions that need to be executed in a particular order, forcing the calls guarantees that the same sequence is followed every time . This is especially important in applications where consistency and predictable behavior are critical, such as in financial systems or scientific simulations.
5. **Performance Optimization**: In some cases, forcing a function call can lead to performance optimizations. If you know that a specific function is required for a particular task and that the AI's dynamic selection process might introduce unnecessary overhead, you can bypass the selection process and directly invoke the required function. This can help reduce latency and improve the overall efficiency of your application .

In summary, the ability to force function calls in AI-driven applications provides explicit control, aids in debugging and testing, handles edge cases, ensures consistency and reproducibility. It's a powerful tool in your arsenal, but we need to discuss one more aspect of this important feature.

> In many decision-making use cases, we always want the model to make a function call and may never want the model to respond with just its internal knowledge . For example, if you're routing between multiple models specialized at different tasks (multilingual input, math, etc), you may use the function-calling model to delegate requests to one of the helper models and never respond independently.

Tool Choice Parameter

GPT-4 and other language models that support function calling give you a `tool_choice` parameter for controlling whether tool use is required as part of a completion. This parameter has three possible values:

- `auto` gives the AI full discretion over using a tool or simply responding
- `required` tells the AI that it *must* call a tool *instead* of responding, but leaves selection of the tool up to the ai.
- The third option is to set the parameter of the `name_of_function` that you want to force. More on that in the next section.

> Note that if you set tool choice to `required`, the model will be forced to pick the most relevant function to call out of those provided to it, even if none really fits the prompt. At the time of publication, I am not aware of a model that will return an empty `tool_calls` response, or use some other way of letting you know that it did not find a suitable function to call .

Forcing a Function To Get Structured Output

The ability to force a function call gives you a way to force structured data out of a chat completion instead of having to extract it yourself out of its plaintext response.

> Why is forcing functions to get structured output a big deal? Simply put, because extraction of structured data from LLM output is a pain in the neck. You can make your life a bit easier by asking for data in XML , but then you have to parse XML. And what do you do when that XML is missing because your AI responded: "I'm sorry, but I'm unable to generate the data you requested because the bla, bla, bla..."

When using tools in this way:

- You should probably define a single tool in your request
- Remember to force use of its function using the `tool_choice` parameter.
- Remember that the model will pass the input to the tool, so the name of the tool and description should be from the model's perspective, not yours.

This last point deserves an example for clarity. Let's say that you are asking the AI to do sentiment analysis on user text . The name of the function would not be `analyze_sentiment`, but rather it would be something like `save_sentiment_analysis`. The AI is the one doing the sentiment analysis, *not the tool.* All the tool is doing (from the perspective of the AI) is saving the results of the analysis.

Here's an example of using Claude 3 to record a summary of an image into well-structured JSON , this time from the command line using `curl`:

```
curl https://api.anthropic.com/v1/messages \
     --header "content-type: application/json" \
     --header "x-api-key: $ANTHROPIC_API_KEY" \
     --header "anthropic-version: 2023-06-01" \
     --header "anthropic-beta: tools-2024-04-04" \
     --data \
'{
    "model": "claude-3-sonnet-20240229",
    "max_tokens": 1024,
    "tools": [{
        "name": "record_summary",
        "description": "Record summary of image into well-structured JSON.",
        "input_schema": {
            "type": "object",
            "properties": {
                "key_colors": {
                    "type": "array",
                    "items": {
                        "type": "object",
                        "properties": {
                            "r": {
                                "type": "number",
                                "description": "red value [0.0, 1.0]"
                            },
                            "g": {
                                "type": "number",
                                "description": "green value [0.0, 1.0]"
                            },
                            "b": {
                                "type": "number",
                                "description": "blue value [0.0, 1.0]"
                            },
                            "name": {
                                "type": "string",
                                "description": "Human-readable color name
                                 in snake_case, e.g.
                                 \"olive_green\"or
                                 \"turquoise\""
                            }
```

```
                        },
                        "required": [ "r", "g", "b", "name" ]
                    },
                    "description": "Key colors in the image. Four or less."
                },
                "description": {
                    "type": "string",
                    "description": "Image description. 1-2 sentences max."
                },
                "estimated_year": {
                    "type": "integer",
                    "description": "Estimated year that the image was taken,
                                    if is it a photo. Only set this if the
                                    image appears to be non-fictional.
                                    Rough estimates are okay!"
                }
            },
            "required": [ "key_colors", "description" ]
        }
    }],
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "type": "image",
                    "source": {
                        "type": "base64",
                        "media_type": "'$IMAGE_MEDIA_TYPE'",
                        "data": "'$IMAGE_BASE64'"
                    }
                },
                {
                    "type": "text",
                    "text": "Use `record_summary` to describe this image."
                }
            ]
        }
    ]
}'
```

In the provided example, we are using the Claude 3 model from Anthropic to generate a structured JSON summary of an image. Here's how it works:

1. We define a single tool named `record_summary` in the `tools` array of the request payload. This tool is responsible for recording a summary of the image into well-structured JSON.
2. The `record_summary` tool has an `input_schema` that specifies the expected structure of the JSON output. It defines three properties:

    - `key_colors`: An array of objects representing the key colors in the image. Each color object has properties for the red, green, and blue values (ranging from 0.0 to 1.0) and a human-readable color name in snake_case format.
    - `description`: A string property for a brief description of the image, limited to 1-2 sentences.
    - `estimated_year`: An optional integer property for the estimated year the image was taken, if it appears to be a non-fictional photo.

3. In the `messages` array, we supply the image data as a base64-encoded string along with the media type. This allows the model to process the image as part of the input.
4. We also prompt Claude to use the `record_summary` tool to describe the image.
5. When the request is sent to the Claude 3 model, it analyzes the image and generates a JSON summary based on the specified `input_schema`. The model extracts the key colors, provides a brief description, and estimates the year the image was taken (if applicable).
6. The generated JSON summary passed as the parameters to the `record_summary` tool, providing a structured representation of the image's key characteristics.

By using the `record_summary` tool with a well-defined `input_schema`, we can obtain a structured JSON summary of an image without relying on plain text extraction. This approach ensures that the output follows a consistent format and can be easily parsed and processed by downstream components of the application.

The ability to force a function call and specify the expected output structure is a powerful feature of tool use in AI-driven applications . It allows developers to have more control over the generated output and simplifies the integration of AI-generated data into their application's workflow.

Execution of Function(s)

You've defined functions, and prompted your AI, which decided that it should call one of your functions. Now it's time for your application code or library, if you're using a Ruby gem like `raix-rails` to dispatch the function call and its parameters to the corresponding implementation *in your application code*.

Your application code decides what to do with the results of the function execution. Maybe what to do involves a single line of code in a lambda, or maybe it involves calling an external API. Maybe it involves calling another AI component, or maybe it involves hundreds or even thousands of lines of code in the rest of your system. It's entirely up to you.

Sometimes the function call is the end of the operation, but if the results represent information in a chain of thought to be continued by the AI, then your application code needs to insert the execution results into the chat transcript and let the AI continue processing.

For example, here a Raix function declaration used by Olympia's `AccountManager` to communicate with our clients as part of an Intelligent Workflow

Orchestration for customer service.

```
class AccountManager
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  # lots of other functions...

  function :notify_account_owner,
    "Don't share UUID. Mention dollars if subscription changed",
    message: { type: "string" } do |arguments|
      account.owner.freeform_notify(
        subject: "Account Change Notification",
        message: arguments[:message]
      )
      "Notified account owner"
    end
```

It may not be immediately clear what is happening here, so I'll break it down.

1. The `AccountManager` class defines many functions related to account management. It can change your plan, add and remove team members, amongst other things.
2. Its top-level instructions tell `AccountManager` that it should notify the account owner with the results of the account change request, using the `notify_account_owner` function.
3. The concise definition of the function includes its:

- name
- description
- parameters `message: { type: "string" }`
- a block to execute when the function is called

After updating the transcript with the results of the function block, the `chat_completion` method is called again. This method is responsible for sending the updated conversation transcript back to the AI model for further processing. We refer to this process as a *conversation loop*.

When the AI model receives a new chat completion request with an updated transcript, it has access to the results of the previously executed function. It can analyze these results, incorporate them into its decision-making process, and generate the next response or action based on the cumulative context of the conversation. It can choose to execute additional functions based on the updated context, or it can generate a final response to the original prompt if it determines that no further function calls are necessary.

Optional Continuation of the Original Prompt

When you send the tool results back to the LLM and continue processing of the original prompt, the AI uses those results to either call additional functions or generate a final plain text response.

> Some models such as Cohere's Command-R can cite the specific tools they used in their responses, providing additional transparency and traceability.

Depending on the model in use, the results of the function call will live in transcript messages that have their own special role or be reflected in some other syntax. But the important part is for that data to be in the transcript, so that it can be considered by the AI as it decides what to do next.

> A common (and potentially expensive) error condition is to forget to add the function results to the transcript before continuing the chat. As a result, the AI will get prompted in essentially the same way that it was before it called the function the first time. In other words, as far as the AI is concerned, it hasn't called the function yet. So it calls it again. And again. And again, forever until you interrupt it. Hope your context was not too big, and your model was not too expensive!

Best Practices for Tool Use

To get the most out of tool use, consider the following best practices.

Descriptive Definitions

Provide clear and descriptive names and descriptions for each tool and its input parameters. This helps the LLM better understand the purpose and capabilities of each tool.

> I can tell you from experience that the common wisdom that says that "naming is hard" applies here; I've seen dramatically different results from LLMs just by changing the names of functions or wording of descriptions. Sometimes removing descriptions *improves* performance.

Processing of Tool Results

When passing tool results back to the LLM, ensure that they are well-structured and comprehensive. Use meaningful keys and values to represent the output of each tool. Experiment with different formats and see which works best, from JSON to plain-text.

The Result Interpreter addresses this challenge by employing AI to analyze the results and provide human-friendly explanations, summaries, or key takeaways .

Error Handling

Implement robust error handling mechanisms to handle cases where the LLM may generate invalid or unsupported input parameters for tool calls. Gracefully handle and recover from any errors that may occur during tool execution .

One exceedingly nice quality of the AI is that it understands error messages! Which means that if you're working in a quick and dirty mindset, you can simply catch any exceptions generated in the implementation of a tool, and pass it back to the AI so that it knows what happened!

For example, here's a slimmed down version of the implementation of google search in Olympia:

```
def google_search(conversation, params)
  conversation.update_cstatus("Searching Google...")
  query = params[:query]
  search = GoogleSearch.new(query).get_hash

  conversation.update_cstatus("Summarizing results...")
  SummarizeKnowledgeGraph.new.perform(conversation, search.to_json)
rescue StandardError => e
  Honeybadger.notify(e)
  { error: e.message }.inspect
end
```

Google searches in Olympia are a two-step process. First you do the search, then you summarize the results. If there's a failure, no matter what it is, the exception message is packaged up and sent back to the AI . This technique is the foundation of practically all the *Intelligent Error Handling* patterns .

For instance, let's say that the `GoogleSearch` API call fails due to a 503 Service Unavailable exception. That bubbles up to the top-level rescue, and the description of the error is sent back to the AI as the result of the function call. Instead of just giving the user a blank screen or technical error, the AI says something like "I'm sorry, but I'm unable to access my Google Search capabilities at this time. I can try again later, if you wish."

This may seem like just a clever trick, but consider a different kind of error, one where the AI was calling an external API and had direct control of the parameters to pass to the API. Maybe it made a mistake in how it generated those parameters? Provided that the error message from the external API is detailed enough, passing the error message back to the calling AI means that it can reconsider those parameters and try again. Automatically. No matter what the error was.

Now think of what it would take to replicate that kind of robust error handling in *normal* code. It's practically impossible.

Iterative Refinement

If the LLM is not recommending the appropriate tools or generating suboptimal responses, iterate on the tool definitions, descriptions, and input parameters. Continuously refine and improve the tool setup based on the observed behavior and desired outcomes.

1. Start with simple tool definitions: Begin by defining tools with clear and concise names, descriptions, and input parameters. Avoid over-complicating the tool setup initially and focus on the core functional-

ity. For example, if you want to save the results of sentiment analysis, start with a basic definition like:

```
{
  "name": "save_sentiment_score",
  "description": "Analyze user-provided text and generate sentiment score",
  "parameters": {
    "type": "object",
    "properties": {
      "score": {
        "type": "float",
        "description": "sentiment score from -1 (negative) to 1 (positive)"
      }
    },
    "required": ["score"]
  }
}
```

2. Test and observe: Once you have the initial tool definitions in place, test them with different prompts and observe how the LLM interacts with the tool. Pay attention to the quality and relevance of the generated responses. If the LLM is generating suboptimal responses, it's time to refine the tool definitions.
3. Refine descriptions: If the LLM is misunderstanding the purpose of a tool, try refining the tool's description. Provide more context, examples, or clarifications to guide the LLM in using the tool effectively. For instance, you can update the sentiment analysis tool description to more specifically address the *emotional tone* of the piece of text being analyzed:

```
{
  "name": "save_sentiment_score",
  "description": "Determine the overall emotional tone of a piece of text,
   such as customer reviews, social media posts, or feedback comments.",
  ...
}
```

4. Adjust input parameters: If the LLM is generating invalid or irrelevant input parameters for a tool, consider adjusting the parameter definitions. Add more specific constraints, validation rules, or examples to clarify the expected input format.
5. Iterate based on feedback: Continuously monitor the performance of your tools and gather feedback from users or stakeholders. Use this feedback to identify areas for improvement and make iterative refinements to the tool definitions. For example, if users report that the analysis is not handling sarcasm well, you can add a note in the description:

```
{
  "name": "save_sentiment_score",
  "description": "Analyze the sentiment of a given text and return a sentiment
   score between -1 (negative) and 1 (positive). Note: Sarcasm should be
   considered negative.",
  ...
}
```

By iteratively refining your tool definitions based on observed behavior and feedback, you can gradually improve the performance and effectiveness of your AI-driven application. Remember to keep the tool definitions clear, concise, and focused on the specific task at hand. Regularly test and validate the tool interactions to ensure they align with your desired outcomes.

Composing and Chaining Tools

One of the most powerful aspects of tool use that has only been alluded to so far is the ability to compose and chain multiple tools together to accomplish complex tasks . By carefully designing your tool definitions and their input/output formats, you can create reusable building blocks that can be combined in various ways.

Let's consider an example where you're building a data analysis pipeline for your AI-driven application. You might have the following tools:

1. **`DataRetrieval`:** A tool that fetches data from a database or API based on specified criteria.
2. **`DataProcessing`:** A tool that performs calculations, transformations, or aggregations on the retrieved data.
3. **`DataVisualization`:** A tool that presents the processed data in a user-friendly format, such as charts or graphs.

By chaining these tools together, you can create a powerful workflow that retrieves relevant data, processes it, and presents the results in a meaningful way. Here's how the tool use workflow might look like:

1. The LLM receives a user query asking for insights on sales data for a specific product category.
2. The LLM selects the `DataRetrieval` tool and generates the appropriate input parameters to fetch the relevant sales data from the database.
3. The retrieved data is "passed" to the `DataProcessing` tool, which calculates metrics such as total revenue, average sales price, and growth rate.

4. The processed data is then digested by the `DataVisualization` tool, which creates a visually appealing chart or graph to represent the insights, passing the URL of the chart back to the LLM.
5. Finally, the LLM generates a formatted response to the user query using markdown , incorporating the visualized data and providing a summary of the key findings.

By composing these tools together, you can create a seamless data analysis workflow that can be easily integrated into your application . The beauty of this approach is that each tool can be developed and tested independently, and then combined in different ways to solve various problems.

To enable smooth composition and chaining of tools, it's important to define clear input and output formats for each tool.

For example, the `DataRetrieval` tool might accept parameters such as the database connection details, table name, and query conditions, and return the result set as a structured JSON object . The `DataProcessing` tool can then expect this JSON object as input and produce a transformed JSON object as output. By standardizing the data flow between tools, you can ensure compatibility and reusability.

As you design your tool ecosystem , think about how different tools can be combined to address common use cases in your application. Consider creating high-level tools that encapsulate common workflows or business logic, making it easier for the LLM to select and use them effectively.

Remember, the power of tool use lies in the flexibility and modularity it provides. By breaking down complex tasks into smaller, reusable tools, you can create a robust and adaptable AI-driven application that can tackle a wide range of challenges.

Future Directions

As the field of AI-driven application development evolves, we can expect further advancements in tool use capabilities. Some potential future directions include:

1. **Multi-hop Tool Use**: LLMs may be able to decide how many times they need to use tools in order to generate a satisfactory response. This could involve multiple rounds of tool selection and execution based on intermediate results.
2. **Pre-defined Tools**: AI platforms may provide a set of pre-defined tools that developers can leverage out-of-the-box, such as Python interpreters, web search tools, or common utility functions.
3. **Seamless Integration**: As tool use becomes more prevalent, we can expect better integration between AI platforms and popular development frameworks , making it easier for developers to incorporate tool use into their applications.

Tool use is a powerful technique that enables developers to harness the full potential of LLMs in AI-driven applications. By connecting LLMs to external tools and resources, you can create more dynamic, intelligent, and context-aware systems that can adapt to user needs and provide valuable insights and actions.

While tool use offers immense possibilities, it's important to be aware of potential challenges and considerations. One key aspect is managing the complexity of tool interactions and ensuring the stability and reliability of the overall system. You need to handle scenarios where tool calls may fail, return unexpected results, or have performance implications. Additionally, you should consider security and access control measures to prevent unauthorized or malicious use of tools. Proper error handling, logging, and monitoring mechanisms are crucial to maintain the integrity and performance of your AI-driven application.

As you explore the possibilities of tool use in your own projects , remember to start with clear objectives, design well-structured tool definitions, and iterate based on feedback and results. With the right approach and mindset, tool use can unlock new levels of innovation and value in your AI-driven applications .

Stream Processing

Streaming data over HTTP , also known as server-sent events (SSE) , is a mechanism where the server continuously sends data to the client as it becomes available, without the need for the client to explicitly request it. As the AI's response is generated incrementally, it makes sense to provide a responsive user experience by displaying the AI's output as it is being generated. And in fact all AI providers APIs that I know of offer streaming responses as an option in their completion endpoints.

The reason that this chapter appears here in the book, right after Using Tools is because of how powerful it can be to combine the use of tools with live AI responses to users. Doing so allows for dynamic and interactive experiences where the AI can process user input, utilize various tools and functions at its discretion, and then provide real-time responses.

To achieve this seamless interaction, you need to write stream handlers that can dispatch AI-invoked tool function calls as well as plain text output to the end user. The need to loop after processing a tool function adds an interesting challenge to the job.

Implementating a ReplyStream

To demonstrate how stream processing can be implemented, this chapter will take a deep dive into a simplified version of the `ReplyStream` class that is used in Olympia . Instances of this class can be passed as the `stream` parameter in AI client libraries such as ruby-openai and openrouter .

Here's how I use `ReplyStream` in Olympia's `PromptSubscriber`, which listens via Wisper for the creation of new user messages.

```
class PromptSubscriber
  include Raix::ChatCompletion
  include Raix::PromptDeclarations

  # many other declarations omitted...

  prompt text: -> { user_message.content },
         stream: -> { ReplyStream.new(self) },
         until: -> { bot_message.complete? }

  def message_created(message) # invoked by Wisper
    return unless message.role.user? && message.content?

    # rest of the implementation omitted...
```

In addition to a `context` reference to the prompt subscriber that instantiated it, the `ReplyStream` class also has instance variables to store a buffer of received data, and arrays to keep track of function names and arguments invoked during stream processing.

```ruby
class ReplyStream
  attr_accessor :buffer, :f_name, :f_arguments, :context

  delegate :bot_message, :dispatch, to: :context

  def initialize(context)
    self.context = context
    self.buffer = []
    self.f_name = []
    self.f_arguments = []
  end

  def call(chunk, bytesize = nil)
    # ...
  end

  # ...
end
```

The `initialize` method sets up the initial state of the `ReplyStream` instance, initializing the buffer, context, and other variables.

The `call` method is the main entry point for processing the streaming data . It takes a `chunk` of data (represented as a hash) and an optional `bytesize` parameter, which in our example is unused.. Inside this method, the class uses pattern matching to handle different scenarios based on the structure of the received chunk.

Calling `deep_symbolize_keys` on the chunk helps make the pattern matching more elegant, by letting us operate on symbols rather than strings.

```ruby
def call(chunk, _bytesize)
  case chunk.deep_symbolize_keys

  in { # match function name
    choices: [
      {
        delta: {
          tool_calls: [
            { index: index, function: {name: name} }
          ]
        }
      }
    ] }

    f_name[index] = name
```

The first pattern we're matching for is a tool call along with its associated function name. If we detect one, we tuck it into the `f_name` array. We store function names in an indexed array, because the model is capable of parallel function calling, sending more than one function to execute at a time.

> Parallel function calling is an AI model's ability to perform multiple function calls together, allowing the effects and results of these function calls to be resolved in parallel. This is especially useful if functions take a long time, and reduces round trips with the API , which in turn can save a significant amount of token expenditure.

Next we need to match for the arguments corresponding to the function calls.

```
in { # match arguments
  choices: [
    {
      delta: {
        tool_calls: [
          {
            index: index, function: {arguments: argument }
          }
        ]
      }
    }
  ]}

  f_arguments[index] ||= "" # initialize if not already
  f_arguments[index] << argument
```

Similarly to how we handled the function names , we tuck away arguments in an indexed array.

Next up, we look for normal user-facing messages, which will arrive from the server one token at a time and be assigned to the `new_content` variable. We also need to keep an eye on `finish_reason`. It will be `nil` until the last chunk of the output sequence.

```
in {
  choices: [
    { delta: {content: new_content}, finish_reason: finish_reason }
  ]}

  # you could transmit every chunk to the user here...
  buffer << new_content.to_s

  if finish_reason.present?
    finalize
  elsif new_content.to_s.match?(/\n\n/)
    send_to_client # ...or buffer and transmit once per paragraph
  end
```

Importantly, we add a pattern match expression to handle error messages sent by the AI model provider. In local development environments , we raise an exception, but in production, we log the error and finalize.

```
  in { error: { message: } }
    if Rails.env.local?
      raise message
    else
      Honeybadger.notify("AI Error: #{message}")
      finalize
    end
```

The final else clause of case will execute if none of the previous patterns matched. It's just a safeguard so that if the AI model starts sending us unrecognized chunks we find out about them.

```
  else
    Honeybadger.notify("Unrecognized Chunk: #{chunk}")
  end
end
```

The `send_to_client` method is responsible for sending the buffered content to the client. It checks that the buffer is not empty, updates the bot message content, renders the bot message, and saves the content in the database to ensure data persistence.

```ruby
def send_to_client
  # no need to process pure whitespace
  return if buffer.join.squish.blank?

  # set the buffer content on the bot message
  content = buffer.join
  bot_message.content = content

  # save to database so that we never lose data
  # even if the stream doesn't terminate correctly
  bot_message.update_column(:content, content)

  # update content via websocket
  ConversationRenderer.update(bot_message)
end
```

The `finalize` method is called when the stream processing is complete. It dispatches the function calls if any were received during the stream, updates the bot message with the final content and other relevant information, and resets the function call history .

```ruby
def finalize
  if f_name.any?
    f_name.each_with_index do |name, index|
      # takes care of calling the function wherever it's implemented
      dispatch(name:, arguments: JSON.parse(f_arguments[index]))
    end

    # reset the function call history
    f_name.clear
    f_arguments.clear
  else
    content = buffer.join.presence
    bot_message.update!(content:, complete: true)
    ConversationRenderer.update(bot_message)
  end
end
```

If the model decides to call a function, you need to “dispatch” that function call (name and arguments) in such a way that it gets executed and `function_call` and `function_result` messages get added to the conversation transcript .

In my experience, it's better to handle the creation of function messages in one place in your codebase, instead of relying on the tool implementations. It's cleaner, but also has a very important practical reason too: if the AI model calls a function, and doesn't see resulting call and result messages in the transcript when you loop, *it will call the same function again.* Potentially forever. Remember that the AI is completely stateless , so unless you echo those function calls back to it, they didn't happen.

```ruby
# PromptSubscriber#dispatch

def dispatch(name:, arguments:)
  # adds a function_call message to the conversation transcript
  # plus dispatches to tool and returns result
  conversation.function_call!(name, arguments).then do |result|
    # add function result message to the transcript
    conversation.function_result!(name, result)
  end
end
```

> Clearing the function call history after dispatching is just as important as making sure the call and results end up in your transcript, so that you don't just keep calling the same functions over and over again every time you loop.

The "Conversation Loop"

I keep mentioning looping, but if you're new to function calling , it might not be obvious *why* we need to loop. The reason is that once the AI "asks" you to execute tool functions on its behalf, it will stop replying. It's up to you to execute those functions, gather the results, add the results to the transcript, and then submit the original prompt again in order to get a new set of function calls or user-facing results.

In the `PromptSubscriber` class, we use the `prompt` method from the `PromptDeclarations` module to define the behavior of the conversation loop. The `until` parameter is set to `-> { bot_message.complete? }`, which means that the loop will continue until the `bot_message` is marked as complete.

```
prompt text: -> { user_message.content },
       stream: -> { ReplyStream.new(self) },
       until: -> { bot_message.complete? }
```

> But when is `bot_message` marked as complete? If you've forgotten, refer back to line 13 of the `finalize` method .

Let's review the entire stream processing logic .

1. The `PromptSubscriber` receives a new user message via the `message_created` method, which is invoked by the Wisper pub/sub system every time the end user creates a new prompt.
2. The `prompt` class method declaratively defines the behavior of the chat completion logic for the `PromptSubscriber`. The AI model will have a chat completion executed with the user's message content, a new instance of `ReplyStream` as the stream parameter, and the specified loop condition.
3. The AI model processes the prompt and starts generating a response. As the response is streamed, the `call` method of the `ReplyStream` instance is invoked for each chunk of data.

4. If the AI model decides to call a tool function, the function name and arguments are extracted from the chunk and stored in the `f_name` and `f_arguments` arrays, respectively.
5. If the AI model generates user-facing content, it is buffered and sent to the client via the `send_to_client` method.
6. Once the stream processing is complete, the `finalize` method is called. If any tool functions were invoked during the stream, they are dispatched using the `dispatch` method of the `PromptSubscriber`.
7. The `dispatch` method adds a `function_call` message to the conversation transcript, executes the corresponding tool function, and adds a `function_result` message to the transcript with the result of the function call.
8. After dispatching the tool functions, the function call history is cleared to prevent duplicate function calls in subsequent loops.
9. If no tool functions were invoked, the `finalize` method updates the `bot_message` with the final content, marks it as complete, and sends the updated message to the client.
10. The loop condition `-> { bot_message.complete? }` is evaluated. If the `bot_message` is not marked as complete, the loop continues, and the original prompt is submitted again with the updated conversation transcript .
11. Steps 3-10 are repeated until the `bot_message` is marked as complete, indicating that the AI model has finished generating its response and no further tool functions need to be executed.

By implementing this conversation loop , you enable the AI model to engage in a back-and-forth interaction with the application, executing tool functions as needed and generating user-facing responses until the conversation reaches a natural conclusion.

The combination of stream processing and the conversation loop allows for dynamic and interactive AI-powered experiences, where the AI model can process user input, utilize various tools and functions, and provide real-time responses based on the evolving conversation context.

Auto Continuation

It's important to be aware of AI output limitations. Most models have a maximum number of tokens they can generate in a single response, which is determined by the `max_tokens` parameter. If the AI model reaches this limit while generating a response, it will abruptly stop and indicate that the output was truncated.

In the streaming response from the AI platform API, you can detect this situation by examining the `finish_reason` variable in the chunk. If the `finish_reason` is set to `"length"` (or some other key value specific to the model), it means that the model reached its maximum token limit during generation and the output has been cut short.

One way to handle this scenario gracefully and provide a seamless user experience, is to implement an auto-continuation mechanism in your stream processing logic. By adding a pattern match for length-related finish reasons, you can choose to loop and automatically continue the output from where it left off.

Here's a purposely simplified example of how you can modify the `call` method in the `ReplyStream` class to support auto continuation:

```ruby
LENGTH_STOPS = %w[length MAX_TOKENS]

def call(chunk, _bytesize)
  case chunk.deep_symbolize_keys
    # ...

    in {
      choices: [
        { delta: {content: new_content},
          finish_reason: finish_reason } ] }

      buffer << new_content.to_s

      if finish_reason.blank?
        send_to_client if new_content.to_s.match?(/\n\n/)
      elsif LENGTH_STOPS.include?(finish_reason)
        continue_cutoff
      else
        finalize
      end

    # ...
  end
end

private

def continue_cutoff
  conversation.bot_message!(buffer.join, visible: false)
  conversation.user_message!("please continue", visible: false)
  bot_message.update_column(:created_at, Time.current)
end
```

In this modified version, when the `finish_reason` indicates truncated output,instead of finalizing the stream, we add a pair of messages to the transcript without finalizing, move the original user-facing response message to the "bottom" of the transcript by updating its `created_at` attribute, and then let the loop happen, so that the AI continues generating where it left off.

Remember that the AI completion endpoint is stateless. It only “knows” what you tell it via the transcript. In this case, the way that we communicate to the AI that it was cutoff is by adding “invisible” (to the end user) messages to the transcript. Remember though, that this is a purposely simplified example. A real implementation would need to do further transcript management to ensure that we didn’t waste tokens and/or confuse the AI with duplicated assistant messages in the transcript.

A real implementation of auto-continuation should also have so-called “circuit breaker” logic in place to prevent runaway looping. The reason being that, given certain kinds of user prompts and low `max_tokens` settings, the AI could continue looping user-facing output endlessly.

> Keep in mind that every loop requires a separate request, and that each request consumes your entire transcript again. You should definitely consider the trade-offs between user experience and API usage when deciding whether to implement auto continuation in your application. Auto-continuation in particular can dangerously expensive, especially when using premium commercial models.

Conclusion

Stream processing is a critical aspect of building AI-powered applications that combine tool use with live AI responses. By efficiently handling the streaming data from AI platform APIs, you can provide a seamless and interactive user experience, handle large responses, optimize resource usage, and gracefully handle errors.

The provided `Conversation::ReplyStream` class demonstrates how stream processing can be implemented in a Ruby application using pattern matching and event-driven architecture. By understanding and leveraging stream processing techniques, you can unlock the full potential of AI integration in your applications and deliver powerful and engaging user experiences.

Self Healing Data

Self-healing data is a powerful approach to ensuring data integrity, consistency, and quality in applications by leveraging the capabilities of large language models (LLMs). This category of patterns focuses on the idea of using AI to automatically detect, diagnose, and correct data anomalies, inconsistencies, or errors, thereby reducing the burden on developers and maintaining a high level of data reliability.

At its core, the self-healing data patterns recognize that data is the lifeblood of any application, and ensuring its accuracy and integrity is crucial for the proper functioning and user experience of the application. However, managing and maintaining data quality can be a complex and time-consuming task, especially as applications grow in size and complexity. This is where the power of AI comes into play.

In the self-healing data patterns, AI workers are employed to continuously monitor and analyze your application's data. These models have the ability to understand and interpret patterns, relationships, and anomalies within the data. By leveraging their natural language processing and understanding capabilities, they can identify potential issues or inconsistencies in the data and take appropriate actions to rectify them.

The process of self-healing data typically involves several key steps:

1. **Data Monitoring:** AI workers constantly monitor the application's data streams, databases, or storage systems, looking for any signs of anomalies, inconsistencies, or errors. Alternatively, you can activate an AI component in reaction to an exception.
2. **Anomaly Detection:** When an issue is detected, the AI worker analyzes the data in detail to identify the specific nature and scope of the problem. This could involve detecting missing values, inconsistent formats, or data that violates predefined rules or constraints.
3. **Diagnosis and Correction:** Once the issue is identified, the AI worker uses its knowledge and understanding of the data domain to determine the appropriate course of action. This could involve automatically correcting the data, filling in missing values, or flagging the issue for human intervention if necessary.
4. **Continuous Learning (optional, depending on use case):** As your AI worker encounters and resolves various data issues, it can output describing what happened and how it responded. This metadata can be fed into learning processes that allows you (and perhaps the underlying model, via fine-tuning) to become more effective and efficient over time in identifying and resolving data anomalies.

By automatically detecting and correcting data issues, you can ensure that your application operates on high-quality, reliable data. This reduces the risk of errors, inconsistencies, or data-related bugs affecting the application's functionality or user experience.

Once you have AI workers handling the task of data monitoring and correction, you can focus your efforts on other critical aspects of the application. This saves time and resources that would otherwise be spent on manual data cleaning and maintenance. In fact, as your applications grow in size and complexity, manually managing data quality becomes increasingly challenging. The "Self-Healing Data" patterns scale effectively by leveraging the power of AI to handle large volumes of data and detect issues in real-time.

> Due to their nature, AI models can adapt to changing data patterns, schemas, or requirements over time with little to no supervision. As long as their directives provide adequate guidance, especially regarding intended results, your application may be able to evolve and handle new data scenarios without requiring extensive manual intervention or code changes.

The self-healing data patterns align well with the other categories of patterns we've discussed, such as "Multitude of Workers" . The self-healing data capability can be viewed as a specialized kind of worker that focuses specifically on ensuring data quality and integrity. This kind of worker operates alongside other AI workers, each contributing to different aspects of the application's functionality.

Implementing self-healing data patterns in practice requires careful design and integration of AI models into the application architecture. Because of the risks of data loss and corruption, you should define clear guidelines for how you will use this technique. You should also consider factors such as performance, scalability, and data security.

Practical Case Study: Fixing Broken JSON

One of the most practical and convenient ways to leverage self-healing data is also very simple to explain: fixing broken JSON .

This technique can be applied to the common challenge of dealing with imperfect or inconsistent data generated by LLMs , such as broken JSON, and provides an approach for automatically detecting and correcting these issues.

At Olympia I regularly encounter scenarios where LLMs generate JSON data that is not perfectly valid. This can happen due to various reasons, such as the LLM adding commentary before or after the actual JSON code, or introducing syntax errors like missing commas or unescaped double quotes. These issues can lead to parsing errors and cause disruptions in the application's functionality.

To address this problem, I have implemented a practical solution in the form of a JsonFixer class. This class embodies the "Self-Healing Data" pattern by taking the broken JSON as input and leveraging an LLM to fix it while preserving as much information and intent as possible.

```
class JsonFixer
  include Raix::ChatCompletion

  def call(bad_json, error_message)
    raise "No data provided" if bad_json.blank? || error_message.blank?

    transcript << {
        system: "Consider user-provided JSON that generated a parse exception.
                 Do your best to fix it while preserving the original content
                 and intent as much as possible." }
    transcript << { user: bad_json }
    transcript << { assistant: "What is the error message?"}
    transcript << { user: error_message }
```

```
    transcript << { assistant: "Here is the corrected JSON\n```json\n" }

    self.stop = ["```"]

    chat_completion(json: true)
  end

  def model
    "mistralai/mixtral-8x7b-instruct:nitro"
  end
end
```

Note how `JsonFixer` uses Ventriloquist to guide the AI's responses.

The process of self-healing JSON data works as follows:

1. **JSON Generation:** An LLM is used to generate JSON data based on certain prompts or requirements. However, due to the nature of LLMs, the generated JSON may not always be perfectly valid. The JSON parser will of course raise a `ParserError` if you give it invalid JSON.

```
begin
  JSON.parse(llm_generated_json)
rescue JSON::ParserError => e
  JsonFixer.new.call(llm_generated_json, e.message)
end
```

Note that the exception message is also passed to the `JSONFixer` call so that it doesn't need to fully assume what is wrong with the data, especially since the parser will often tell you exactly what is wrong.

2. **LLM-based Correction:** The `JSONFixer` class sends the broken JSON back to an LLM, along with a specific prompt or instruction to fix the JSON while preserving the original information and intent as much as possible. The LLM, trained on vast amounts of data and with an understanding of JSON syntax, attempts to correct the errors and generate a valid JSON string. Response Fencing is used to constrain the output of the LLM, and we choose Mixtral 8x7B as the AI model, since it is particularly good for this kind of task.
3. **Validation and Integration:** The fixed JSON string returned by the LLM is parsed by the `JSONFixer` class itself, because it called `chat_completion(json: true)`. If the fixed JSON passes validation, it is integrated back into the application's workflow, allowing the application to continue processing the data seamlessly. The bad JSON has been "healed".

Although I've written and rewritten my own `JSONFixer` implementation a number of times, I doubt that the total time invested in all of those versions is more than an hour or two.

Note that preservation of intent is a key element of any self-healing data pattern. The LLM-based correction process aims to preserve the original information and intent of the generated JSON as much as possible. This ensures that the fixed JSON maintains its semantic meaning and can be used effectively within the application's context.

This practical implementation of the "Self-Healing Data" approach in Olympia clearly demonstrates how AI, specifically LLMs, can be leveraged to solve real-world data challenges. It showcases the power of combining traditional programming techniques with AI capabilities to build robust and efficient applications.

Postel's Law and the "Self-Healing Data" Pattern

"Self-Healing Data," as exemplified by the JSONFixer class, aligns well with the principle known as Postel's Law, also referred to as the Robustness Principle. Postel's Law states:

"Be conservative in what you do, be liberal in what you accept from others."

This principle, originally articulated by Jon Postel, a pioneer of the early Internet, emphasizes the importance of building systems that are tolerant of diverse or even slightly incorrect inputs while maintaining strict adherence to specified protocols when sending outputs.

In the context of "Self-Healing Data," the JSONFixer class embodies Postel's Law by being liberal in accepting broken or imperfect JSON data generated by LLMs. It doesn't immediately reject or fail when encountering JSON that doesn't strictly adhere to the expected format. Instead, it takes a tolerant approach and attempts to fix the JSON using the power of LLMs.

By being liberal in accepting imperfect JSON, the JSONFixer class demonstrates robustness and flexibility. It acknowledges that data in the real world often comes in various forms and may not always conform to strict specifications. By gracefully handling and correcting these deviations, the class ensures that the application can continue to function smoothly, even in the presence of imperfect data.

On the other hand, the JSONFixer class also adheres to the conservative aspect of Postel's Law when it comes to the output. After fixing the JSON using LLMs, the class validates the corrected JSON to ensure it strictly conforms to the expected format. It maintains the integrity and correctness of the data before passing it along to other parts of the application. This conservative approach guarantees that the output of the JSONFixer class is reliable and consistent, promoting interoperability and preventing the propagation of errors.

Interesting Trivia about Jon Postel:

- Jon Postel (1943-1998) was an American computer scientist who played a crucial role in the development of the Internet. He was known as the "God of the Internet" for his significant contributions to the underlying protocols and standards.
- Postel was the editor of the Request for Comments (RFC) document series, which is a series of technical and organizational notes about the Internet. He authored or co-authored over 200 RFCs, including the foundational protocols such as TCP, IP, and SMTP.
- In addition to his technical contributions, Postel was known for his humble and collaborative approach. He believed in the importance of reaching consensus and working together to build a robust and interoperable network.
- Postel served as the Director of the Computer Networks Division at the Information Sciences Institute (ISI) of the University of Southern California (USC) from 1977 until his untimely death in 1998.
- In recognition of his immense contributions, Postel was posthumously awarded the prestigious Turing Award in 1998, often referred to as the "Nobel Prize of Computing."

The `JSONFixer` class promotes robustness, flexibility, and interoperability, which were core values that Postel upheld throughout his career. By building systems that are tolerant of imperfections while maintaining

> strict adherence to protocols, we can create applications that are more resilient and adaptable in the face of real-world challenges.

Considerations and Counterindications

The applicability of self-healing data approaches is entirely dependent on the kind of data your application handles. There's a reason why you might not want to simply monkeypatch `JSON.parse` to automatically self-correct *all* JSON *parsing errors* in your application: not all errors can or should be automatically corrected.

Self-healing is particularly fraught when coupled with regulatory or compliance requirements related to data handling and processing. Some industries, such as healthcare and finance, have such strict regulations regarding data integrity and auditability that doing any sort of "black box" data correction without proper oversight or logging may violate these regulations. It's crucial to ensure that whatever self-healing data techniques you come up with align with the applicable legal and regulatory frameworks.

Applying self-healing data techniques, particularly those involving AI models, may also have a large impact on application performance and resource utilization. Processing large volumes of data through AI models for error detection and correction can be computationally intensive. It's important to assess the trade-offs between the benefits of self-healing data and the associated performance and resource costs.

That said, let's dive into the factors involved in deciding when and where to apply this powerful approach.

Data Criticality

When considering the application of self-healing data techniques, it's crucial to assess the criticality of the data being processed. The level of criticality refers to the importance and sensitivity of the data in the context of your application and its business domain.

In some cases, automatically correcting data errors may not be appropriate, especially if the data is highly sensitive or has legal implications. For example, consider the following scenarios:

1. **Financial Transactions:** In financial applications, such as banking systems or trading platforms, data accuracy is of utmost importance. Even minor errors in financial data can have significant consequences, such as incorrect account balances, misrouted funds, or erroneous trading decisions. In these cases, automated corrections without thorough verification and auditing may introduce unacceptable risks.
2. **Medical Records:** Healthcare applications deal with highly sensitive and confidential patient data. Inaccuracies in medical records can have severe implications for patient safety and treatment decisions. Automatically modifying medical data without proper oversight and validation by qualified healthcare professionals may violate regulatory requirements and put patient well-being at risk.
3. **Legal Documents:** Applications handling legal documents, such as contracts, agreements, or court filings, require strict accuracy and integrity. Even minor errors in legal data can have significant legal ramifications. Automated corrections in this domain may not be appropriate, as the data often requires manual review and verification by legal experts to ensure its validity and enforceability.

In these critical data scenarios, the risks associated with automated corrections often outweigh the potential benefits. The consequences of introducing errors or modifying data incorrectly can be severe, leading to financial losses, legal liabilities, or even harm to individuals.

When dealing with highly critical data, it's essential to prioritize manual verification and validation processes. Human oversight and expertise are crucial in ensuring the accuracy and integrity of the data. Automated self-healing techniques can still be employed to flag potential errors or inconsistencies, but the final decision on corrections should involve human judgment and approval.

However, it's important to note that not all data in an application may have the same level of criticality. Within the same application, there may be subsets of data that are less sensitive or have lower impact if errors occur. In such cases, self-healing data techniques can be selectively applied to those specific data subsets, while critical data remains subject to manual verification.

The key is to carefully assess the criticality of each data category in your application and define clear guidelines and processes for handling corrections based on the associated risks and implications. By differentiating between critical (i.e. ledgers, medical records) and non-critical data (i.e. mailing addresses, resource warnings), you can strike a balance between leveraging the benefits of self-healing data techniques where appropriate and maintaining strict control and oversight where necessary.

Ultimately, the decision to apply self-healing data techniques to critical data should be made in consultation with domain experts, legal advisors, and other relevant stakeholders. It's essential to consider the specific requirements, regulations, and risks associated with your application's data and align the data correction strategies accordingly.

Error Severity

When applying self-healing data techniques, it's important to assess the severity and impact of the data errors. Not all errors are created equal, and the appropriate course of action may vary depending on the severity of the issue.

Minor inconsistencies or formatting issues may be suitable for automatic correction. For example, a self-healing data worker tasked with fixing broken JSON can handle missing commas or unescaped double quotes without significantly altering the meaning or structure of the data. These types of errors are often straightforward to correct and have minimal impact on the overall data integrity.

However, more severe errors that fundamentally change the meaning or integrity of the data may require a different approach. In such cases, automated corrections may not be sufficient, and human intervention may be necessary to ensure the accuracy and validity of the data.

This is where the concept of using AI itself to help determine error severity comes into play. By leveraging the capabilities of AI models, we can design self-healing data workers that not only correct errors but also assess the severity of those errors and make informed decisions on how to handle them.

For instance, let's consider a self-healing data worker responsible for correcting inconsistencies in data flowing into a customer database. The worker can be designed to analyze the data and identify potential errors, such as missing or conflicting information. However, instead of automatically correcting all errors, the worker can be equipped with additional tool calls that allow it to flag severe errors for human review.

Here's an example of how this can be implemented:

````ruby
class CustomerDataReviewer
  include Raix::ChatCompletion
  include Raix::FunctionDeclarations

  attr_accessor :customer

  function :flag_for_review, reason: { type: "string" } do |params|
    AdminNotifier.review_request(customer, params[:reason])
  end

  def initialize(customer)
    self.customer = customer
  end

  def call(customer_data)
    transcript << {
      system: "You are a customer data reviewer. Your task is to identify
        and correct inconsistencies in customer data.

        < additional instructions here... >

        If you encounter severe errors that require human review, use the
        `flag_for_review` tool to flag the data for manual intervention." }

    transcript << { user: customer.to_json }
    transcript << { assistant: "Reviewed/corrected data:\n```json\n" }

    self.stop = ["```"]

    chat_completion(json: true).then do |result|
      return if result.blank?

      customer.update(result)
    end
  end
end
````

In this example, the `CustomerDataHealer` worker is designed to identify and correct inconsistencies in customer data. Once again, we use Response Fencing and Ventriloquist to get structured output. Importantly, the worker's system directive includes instructions to use the `flag_for_review` function if severe errors are encountered.

When the worker processes the customer data, it analyzes the data and attempts to correct any inconsistencies. If the worker determines that the errors are severe and require human intervention, it can use the `flag_for_review` tool to flag the data and provide a reason for the flagging.

The `chat_completion` method is called with `json: true` to parse the corrected customer data as JSON. There is no provision for looping after a function call, so the result will be blank if `flag_for_review` was invoked. Otherwise, the customer is updated with the reviewed and potentially corrected data.

By incorporating error severity assessment and the option to flag data for human review, the self-healing data worker becomes more intelligent and adaptable. It can handle minor errors automatically while escalating severe errors to human experts for manual intervention.

The specific criteria for determining error severity can be defined in the worker's directive based on the domain knowledge and business requirements. Factors such as the impact on data integrity, the potential for data loss or corruption, and the consequences of incorrect data can be considered when assessing severity.

By leveraging AI to assess error severity and providing options for human intervention, self-healing data techniques can strike a balance between automation and maintaining data accuracy. This approach ensures that minor errors are corrected efficiently while severe errors receive the necessary attention and expertise from human reviewers.

Domain Complexity

When considering the application of self-healing data techniques, it's important to evaluate the complexity of the data domain and the rules governing its structure and relationships. The complexity of the domain can significantly impact the effectiveness and feasibility of automated data correction approaches.

Self-healing data techniques work well when the data follows well-defined patterns and constraints. In domains where the data structure is relatively simple and the relationships between data elements are straightforward, automated corrections can be applied with a high degree of confidence. For example, correcting formatting issues or enforcing basic data type constraints can often be handled effectively by self-healing data workers.

However, as the complexity of the data domain increases, the challenges associated with automated data correction also grow. In domains with intricate business logic, complex relationships between data entities, or domain-specific rules and exceptions, self-healing data techniques may not always capture the nuances and may introduce unintended consequences.

Let's consider an example of a complex domain: a financial trading system. In this domain, the data involves various financial instruments, market data, trading rules, and regulatory requirements. The relationships between different data elements can be intricate, and the rules governing data validity and consistency can be highly specific to the domain.

In such a complex domain, a self-healing data worker tasked with correcting inconsistencies in trade data would need to have a deep understanding of the domain-specific rules and constraints. It would need to consider factors such as market regulations, trading limits, risk calculations, and settlement procedures. Automated corrections in this context may not always capture the full complexity of the domain and may inadvertently introduce errors or violate domain-specific rules.

To address the challenges of domain complexity, self-healing data techniques can be enhanced by incorporating domain-specific knowledge and rules into the AI models and workers. This can be achieved through techniques such as:

1. **Domain-Specific Training:** The AI models used for self-healing data can be directed or even fine-tuned on domain-specific datasets that capture the intricacies and rules of the particular domain. By exposing the models to representative data and scenarios, they can learn the patterns, constraints, and exceptions specific to the domain.
2. **Rule-Based Constraints:** Self-healing data workers can be augmented with explicit rule-based constraints that encode domain-specific knowledge. These rules can be defined by domain experts and integrated into the data correction process. The AI models can then use these rules to guide their decisions and ensure compliance with domain-specific requirements.
3. **Collaboration with Domain Experts:** In complex domains, it's crucial to involve domain experts in the design and development of self-healing data techniques. Domain experts can provide valuable insights into the intricacies of the data, the business rules, and the potential edge cases. Their knowledge can be incorporated into the AI models and workers to improve the accuracy and reliability of automated data corrections using Human In The Loop patterns .
4. **Incremental and Iterative Approach:** When dealing with complex domains, it's often beneficial to adopt an incremental and iterative approach to self-healing data. Instead of attempting to automate

corrections for the entire domain at once, focus on specific subdomains or data categories where the rules and constraints are well-understood. Gradually expand the scope of self-healing techniques as the understanding of the domain grows and the techniques prove effective.

By considering the complexity of the data domain and incorporating domain-specific knowledge into self-healing data techniques, you can strike a balance between automation and accuracy. It's important to recognize that self-healing data is not a one-size-fits-all solution and that the approach should be tailored to the specific requirements and challenges of each domain.

In complex domains, a hybrid approach that combines self-healing data techniques with human expertise and oversight can be most effective. Automated corrections can handle routine and well-defined cases, while complex scenarios or exceptions can be flagged for human review and intervention. This collaborative approach ensures that the benefits of automation are realized while maintaining the necessary control and accuracy in complex data domains.

Explainability and Transparency

Explainability refers to the ability to understand and interpret the reasoning behind the decisions made by AI models, while transparency involves providing clear visibility into the data correction process.

In many contexts, data modifications need to be auditable and justifiable. Stakeholders, including business users, auditors, and regulatory bodies, may require explanations for why certain data corrections were made and how the AI models arrived at those decisions. This is especially crucial in domains where data accuracy and integrity have significant implications, such as finance, healthcare, and legal matters.

To address the need for explainability and transparency, self-healing data techniques should incorporate mechanisms that provide insights into the decision-making process of AI models. This can be achieved through various approaches:

1. **Chain of Thought:** Asking the model to explain its thinking "out loud" before applying changes to data may allow for easier understanding of the decision-making process and can generate human-readable explanations for the corrections made. The tradeoff is a little bit more complexity in separating the explanation from the structured data output, which can be addressed by…
2. **Explanation Generation:** Self-healing data workers can be equipped with the ability to generate human-readable explanations for the corrections they make. This can be achieved by asking the model to output its decision-making process as easily understandable explanations *integrated into the data itself*. For example, a self-healing data worker could generate a report that highlights the specific data inconsistencies it identified, the corrections it applied, and the rationale behind those corrections.
3. **Feature Importance:** AI models can be instructed with information about the importance of different features or attributes in the data correction process as part of their directives. Those directives, in turn, can be exposed to human stakeholders. By identifying the key factors that influence the model's decisions, stakeholders can gain insights into the reasoning behind the corrections and assess their validity.
4. **Logging and Auditing:** Implementing comprehensive logging and

auditing mechanisms is crucial for maintaining transparency in the self-healing data process. Every data correction made by AI models should be logged, including the original data, the corrected data, and the specific actions taken. This audit trail allows for retrospective analysis and provides a clear record of the modifications made to the data.

5. **Human-in-the-Loop Approach:** Incorporating a human-in-the-loop approach can enhance the explainability and transparency of self-healing data techniques. By involving human experts in the review and validation of AI-generated corrections, organizations can ensure that the corrections align with domain knowledge and business requirements. Human oversight adds an additional layer of accountability and allows for the identification of any potential biases or errors in the AI models.
6. **Continuous Monitoring and Evaluation:** Regularly monitoring and evaluating the performance of self-healing data techniques is essential for maintaining transparency and trust. By assessing the accuracy and effectiveness of the AI models over time, organizations can identify any deviations or anomalies and take corrective actions. Continuous monitoring helps ensure that the self-healing data process remains reliable and aligned with the desired outcomes.

Explainability and transparency are critical considerations when implementing self-healing data techniques. By providing clear explanations for data corrections, maintaining comprehensive audit trails, and involving human oversight, organizations can build trust in the self-healing data process and ensure that the modifications made to the data are justifiable and aligned with business objectives.

It's important to strike a balance between the benefits of automation and the need for transparency. While self-healing data techniques can significantly improve data quality and efficiency, they should not come at the cost of losing visibility and control over the data correction process. By designing self-healing data workers with explainability and transparency in mind, organizations can harness the power of AI while maintaining the necessary level of accountability and trust in the data.

Unintended Consequences

While self-healing data techniques aim to improve data quality and consistency, it's crucial to be aware of the potential for unintended consequences. Automated corrections, if not carefully designed and monitored, may inadvertently alter the meaning or context of the data, leading to downstream issues.

One of the primary risks of self-healing data is the introduction of bias or errors in the data correction process. AI models, like any other software system, can be subject to biases present in the training data or introduced through the design of the algorithms. If these biases are not identified and mitigated, they can propagate through the self-healing data process and result in skewed or incorrect data modifications.

For example, consider a self-healing data worker tasked with correcting inconsistencies in customer demographic data. If the AI model has learned biases from historical data, such as associating certain occupations or income levels with specific genders or ethnicities, it may make incorrect assumptions and modify the data in a way that reinforces those biases. This can lead to inaccurate customer profiles, misguided business decisions, and potentially discriminatory outcomes.

Another potential unintended consequence is the loss of valuable information or context during the data correction process. Self-healing data techniques often focus on standardizing and normalizing data to ensure consistency. However, in some cases, the original data may contain nuances, exceptions, or contextual information that is important for understanding the full picture. Automated corrections that blindly enforce standardization may inadvertently remove or obscure this valuable information.

For instance, imagine a self-healing data worker responsible for correcting inconsistencies in medical records. If the worker encounters a patient's medical history with a rare condition or an unusual treatment plan, it may attempt to normalize the data to fit a more common pattern. However, in doing so, it may lose the specific details and context that are crucial for accurately representing the patient's unique situation. This loss of information can have serious implications for patient care and medical decision-making.

To mitigate the risks of unintended consequences, it's essential to take a proactive approach when designing and implementing self-healing data techniques:

1. **Thorough Testing and Validation:** Before deploying self-healing data workers in production, it's crucial to thoroughly test and validate their behavior against a diverse range of scenarios. This includes testing with representative datasets that cover various edge cases, exceptions, and potential biases. Rigorous testing helps identify and address any unintended consequences before they impact real-world data.
2. **Continuous Monitoring and Evaluation:** Implementing continuous monitoring and evaluation mechanisms is essential for detecting and mitigating unintended consequences over time. Regularly reviewing the outcomes of self-healing data processes, analyzing the impact

on downstream systems and decision-making, and gathering feedback from stakeholders can help identify any adverse effects and prompt timely corrective actions. If your organization has operational dashboards, adding plainly visible metrics related to automated data changes is probably a good idea. Adding alarms connected to large deviations from normal data change activity is probably an even better idea!

3. **Human Oversight and Intervention:** Maintaining human oversight and the ability to intervene in the self-healing data process is crucial. While automation can greatly improve efficiency, it's important to have human experts review and validate the corrections made by AI models, especially in critical or sensitive domains. Human judgment and domain expertise can help identify and address any unintended consequences that may arise.

4. **Explainable AI (XAI) and Transparency:** As discussed in the previous subsection, incorporating explainable AI techniques and ensuring transparency in the self-healing data process can help mitigate unintended consequences. By providing clear explanations for data corrections and maintaining comprehensive audit trails, organizations can better understand and trace the reasoning behind the modifications made by AI models.
5. **Incremental and Iterative Approach:** Adopting an incremental and iterative approach to self-healing data can help minimize the risk of unintended consequences. Instead of applying automated corrections to the entire dataset at once, start with a subset of data and gradually expand the scope as the techniques prove effective and reliable. This allows for careful monitoring and adjustment along the way, reducing the impact of any unintended consequences.

6. **Collaboration and Feedback:** Engaging stakeholders from different domains and encouraging collaboration and feedback throughout the self-healing data process can help identify and address unintended consequences. Regularly seeking input from domain experts, data consumers, and end-users can provide valuable insights into the real-world impact of the data corrections and highlight any issues that may have been overlooked.

By proactively addressing the risk of unintended consequences and implementing appropriate safeguards, organizations can harness the benefits of self-healing data techniques while minimizing potential adverse effects. It's important to approach self-healing data as an iterative and collaborative process, continuously monitoring, evaluating, and refining the techniques to ensure they align with the desired outcomes and maintain the integrity and reliability of the data.

When considering the use of self-healing data patterns, it's essential to carefully evaluate these factors and weigh the benefits against the potential risks and limitations. In some cases, a hybrid approach that combines automated corrections with human oversight and intervention may be the most appropriate solution.

It's also worth noting that self-healing data techniques should not be seen as a replacement for robust data validation, input sanitization, and error handling mechanisms. These foundational practices remain critical for ensuring data integrity and security. Self-healing data should be viewed as a complementary approach that can augment and enhance these existing measures.

Ultimately, the decision to employ self-healing data patterns depends on the specific requirements, constraints, and priorities of your application. By carefully considering the considerations outlined above and aligning them with your application's goals and architecture, you can make informed decisions on when and how to leverage self-healing data techniques effectively.

Contextual Content Generation

Contextual Content Generation patterns leverage the power of large language models (LLMs) to generate dynamic and context-specific content within applications. This category of patterns recognizes the importance of delivering personalized and relevant content to users based on their specific needs, preferences, and even previous and current interactions with the application.

> In the context of this approach, "content" refers both to primary content (i.e. blog posts, articles, etc) and meta-content, such as recommendations to primary content.

Contextual Content Generation patterns can play a crucial role in enhancing your user engagement levels, providing tailored experiences, and automating content creation tasks both for you and your users. By utilizing the patterns we describe in this chapter, you can create applications that generate content dynamically, adapting to context and inputs in real-time.

The patterns work by integrating LLMs into the application's outputs, ranging from the user interface (sometimes referred to as "chrome"), to emails and other forms of notifications, as well as any content generation pipelines.

When a user interacts with the application or triggers a specific content request, the application captures the relevant context, such as user preferences, previous interactions, or specific prompts. This contextual information is then fed into the LLM, along with any necessary templates or guidelines and used to produce textual output that would otherwise have to be either hardcoded, stored in a database, or algorithmically generated.

The LLM generated content can take various forms, such as personalized recommendations, dynamic product descriptions, customized email responses, or even entire articles or blog posts. One of the most radical uses of this content that I pioneered over a year ago is dynamically generating UI elements like form labels, tooltips, and other kinds of explanatory text.

Personalization

One of the key benefits of Contextual Content Generation patterns is the ability to deliver highly personalized experiences to users. By generating content based on user-specific context, these patterns enable applications to tailor content to individual users' interests, preferences, and interactions.

Personalization goes beyond simply inserting a user's name into generic content. It involves leveraging the rich context available about each user to generate content that resonates with their specific needs and desires. This context can include a wide range of factors, such as:

1. **User Profile Information:** At the most general level of applying this technique, demographic data, interests, preferences, and other profile attributes can be used to generate content that aligns with the user's background and characteristics.
2. **Behavioral Data:** A user's past interactions with the application, such as viewed pages, clicked links, or purchased products, can provide valuable insights into their behavior and interests. This data can be used to generate content suggestions that reflects their engagement patterns and predicts their future needs.
3. **Contextual Factors:** The user's current context, such as their location, device, time of day, or even the weather, can influence the content generation process. For example, a travel application might have an AI worker that is able to generate personalized recommendations based on the user's current location and the prevailing weather conditions.

By leveraging these contextual factors, Contextual Content Generation patterns enable applications to deliver content that feels tailor-made for each individual user. This level of personalization has several significant benefits:

1. **Increased Engagement:** Personalized content captures users' attention and keeps them engaged with the application. When users feel that the content is relevant and speaks directly to their needs, they are more likely to spend more time interacting with the application and exploring its features.

2. **Improved User Satisfaction:** Personalized content demonstrates that the application understands and cares about the user's unique requirements. By providing content that is helpful, informative, and aligned with their interests, the application can enhance user satisfaction and build a stronger connection with its users.
3. **Higher Conversion Rates:** In the context of e-commerce or marketing applications, personalized content can significantly impact conversion rates. By presenting users with products, offers, or recommendations that are tailored to their preferences and behavior, the application can increase the likelihood of users taking desired actions, such as making a purchase or signing up for a service.

Productivity

Contextual Content Generation patterns can significantly boost certain kinds of productivity by reducing the need for manual content generation and editing in creative processes. By leveraging the power of LLMs, you can generate high-quality content at scale, saving time and effort that your content creators and developers would otherwise have to spend doing tedious manual work.

Traditionally, content creators need to research, write, edit, and format content to ensure it meets the application's requirements and user expectations. This process can be time-consuming and resource-intensive, especially as the volume of content grows.

However, with Contextual Content Generation patterns, the content creation process can be largely automated. LLMs can generate coherent, grammatically correct, and contextually relevant content based on the provided prompts and guidelines. This automation offers several productivity benefits:

1. **Reduced Manual Effort:** By delegating content generation tasks to LLMs, content creators can focus on higher-level tasks such as content strategy, ideation, and quality assurance. They can provide the necessary context, templates, and guidelines to the LLM and let it handle the actual content generation. This reduces the manual effort required for writing and editing, allowing content creators to be more productive and efficient.
2. **Faster Content Creation:** LLMs can generate content much faster than human writers. With the right prompts and guidelines, an LLM can produce multiple pieces of content in a matter of seconds or minutes. This speed enables applications to generate content at a much faster pace, keeping up with the demands of users and the ever-changing digital landscape.

> Is faster content creation leading to a "tragedy of the commons" situation where the internet is drowning in content that nobody reads. Sadly, I suspect the answer is yes.

3. **Consistency and Quality:** LLMs can trivially revise content so that it is consistent in style, tone, and quality. Provided clear guidelines and examples, certain kinds of applications (i.e. newsroom, PR, etc.) can ensure that their human-generated content aligns with their brand voice and meets the desired quality standards. This consistency reduces the need for extensive editing and revisions, saving time and effort in the content creation process.

4. **Iteration and Optimization:** Contextual Content Generation patterns enable rapid iteration and optimization of content. By adjusting the prompts, templates, or guidelines provided to the LLM, your applications can quickly generate variations of content and test different approaches in an automated fashion that was never possible in the past. This iterative process allows for faster experimentation and refinement of content strategies, leading to more effective and engaging content over time. This particular technique can be a total game-changer for applications such as e-commerce that live and die based on bounce rates and engagement .

> It's important to note that while Contextual Content Generation patterns can greatly enhance productivity, they do not completely eliminate the need for human involvement. Content creators and editors still play a crucial role in defining the overall content strategy, providing guidance to the LLM, and ensuring the quality and appropriateness of the generated content.

By automating the more repetitive and time-consuming aspects of content creation, Contextual Content Generation patterns free up valuable human time and resources that can be redirected towards higher-value tasks. This increased productivity enables you to deliver more personalized and engaging content to users while optimizing content creation workflows.

Rapid Iteration and Experimentation

Contextual Content Generation patterns enable you to quickly iterate and experiment with different content variations, allowing for faster optimization and refinement of your content strategy. You can generate multiple versions of content in a matter of seconds, simply by adjusting the context, templates, or guidelines provided to the model.

This rapid iteration capability offers several key benefits:

1. **Testing and Optimization:** With the ability to generate content variations quickly, you can easily test different approaches and measure their effectiveness. For example, you can generate multiple versions of a product description or a marketing message, each tailored to a specific user segment or context. By analyzing user engagement metrics, such as click-through rates or conversion rates, you can identify the most effective content variations and optimize your content strategy accordingly.

2. **A/B Testing:** Contextual Content Generation patterns enable seamless A/B testing of content. You can generate two or more variations of content and randomly serve them to different user groups. By comparing the performance of each variation, you can determine which content resonates best with your target audience. This data-driven approach allows you to make informed decisions and continuously refine your content to maximize user engagement and achieve your desired outcomes.
3. **Personalization Experiments:** Rapid iteration and experimentation are particularly valuable when it comes to personalization. With Contextual Content Generation patterns , you can quickly generate personalized content variations based on different user segments, preferences, or behaviors. By experimenting with different personalization strategies, you can identify the most effective approaches for engaging individual users and delivering tailored experiences.
4. **Adapting to Changing Trends:** The ability to iterate and experiment rapidly enables you to stay agile and adapt to changing trends and

user preferences. As new topics, keywords, or user behaviors emerge, you can quickly generate content that aligns with these trends. By continuously experimenting and refining your content, you can stay relevant and maintain a competitive edge in the ever-evolving digital landscape .

5. **Cost-Effective Experimentation:** Traditional content experimentation often involves significant time and resources, as content creators need to manually develop and test different variations. However, with Contextual Content Generation patterns, the cost of experimentation is greatly reduced. LLMs can generate content variations quickly and at scale, allowing you to explore a wide range of ideas and approaches without incurring substantial costs.

To make the most of rapid iteration and experimentation, it's important to have a well-defined experimentation framework in place. This framework should include:

- Clear objectives and hypotheses for each experiment
- Appropriate metrics and tracking mechanisms to measure content performance
- Segmentation and targeting strategies to ensure relevant content variations are served to the right users
- Analysis and reporting tools to derive insights from the experimental data
- A process for incorporating learnings and optimizations into your content strategy

By embracing rapid iteration and experimentation, you can continuously refine and optimize your content, ensuring that it remains engaging, relevant, and effective in achieving your application's goals. This agile approach to content creation allows you to stay ahead of the curve and deliver exceptional user experiences .

Scalability and Efficiency

As applications grow and the demand for personalized content increases, contextual content generation patterns enable efficient scaling of content creation. LLMs can generate content for a large number of users and contexts simultaneously, without the need for a proportional increase in human resources. This scalability allows applications to deliver personalized experiences to a growing user base without straining their content creation capabilities.

> Note that contextual content generation can be used effectively to internationalize your application "on the fly". In fact, that's exactly what I did using my Instant18n Gem to deliver Olympia in more than half-dozen languages , even though we're less than a year old.

AI Powered Localization

If you allow me to brag for a moment, I think that my Instant18n library for Rails apps is a groundbreaking example of the "Contextual Content Generation" pattern in action, showcasing the transformative potential of AI in application development. This gem leverages the power of OpenAI's GPT large-language model to revolutionize the way internationalization and localization are handled in Rails applications.

Traditionally, internationalizing a Rails application involves manually defining translation keys and providing corresponding translations for each supported language. This process can be time-consuming, resource-intensive, and prone to inconsistencies. However, with the Instant18n gem, the paradigm of localization is completely redefined.

By integrating a large language model, the Instant18n gem enables you to generate translations on-the-fly, based on the context and meaning of the text. Instead of relying on predefined translation keys and static translations, the gem dynamically translates text using the power of AI. This approach offers several key benefits:

1. **Seamless Localization:** With the Instant18n gem, developers no longer need to manually define and maintain translation files for each supported language . The gem automatically generates translations based on the provided text and the desired target language, making the localization process effortless and seamless.
2. **Contextual Accuracy:** AI can be given enough context to figure out the nuances of the text being translated. It can take into account the surrounding context, idioms, and cultural references to generate translations that are accurate, natural-sounding, and contextually appropriate.
3. **Extensive Language Support:** The Instant18n gem leverages the vast knowledge and linguistic capabilities of GPT, enabling translations into an extensive range of languages. From common languages like Spanish and French to more obscure or fictional languages like Klingon and Elvish, the gem can handle a wide variety of translation requirements.
4. **Flexibility and Creativity:** The gem goes beyond traditional language translations and allows for creative and unconventional localization options. Developers can translate text into various styles, dialects, or even fictional languages, opening up new possibilities for unique user

experiences and engaging content .

5. **Performance Optimization:** The Instant18n gem incorporates caching mechanisms to improve performance and reduce the overhead of repeated translations. Translated text is cached, allowing subsequent requests for the same translation to be served quickly without the need for redundant API calls .

The Instant18n gem exemplifies the power of the "Contextual Content Generation" pattern by leveraging AI to generate localized content dynamically. It showcases how AI can be integrated into the core functionality of a Rails application, transforming the way developers approach internationalization and localization.

By eliminating the need for manual translation management and enabling on-the-fly translations based on context, the Instant18n gem saves developers significant time and effort. It allows them to focus on building the core features of their application while ensuring that the localization aspect is handled seamlessly and accurately.

The Importance of User Testing and Feedback

Finally, always keep in mind the importance of user testing and feedback . It's crucial to validate that contextual content generation meets user expectations and aligns with the application's goals. Continuously iterate and refine generated content based on user insights and analytics. If you're generating dynamic content on a large scale that would be impossible to validate manually by you and your team, consider adding feedback mechanisms that allow users to report content that is weird or wrong, along with an explanation of why. That precious feedback can even be fed to an AI worker tasked with making adjustments to the component that generated the content!

Generative UI

Attention is at such a premium these days that effective user engagement now demands software experiences that are not only seamless and intuitive but also highly personalized to individual needs, preferences, and contexts. As a result, designers and developers are increasingly faced with the challenge of creating user interfaces ! that can adapt and cater to the unique requirements of each user *at scale*.

Generative UI (GenUI) is a truly revolutionary approach to user interface design ! that leverages the power of large language models (LLMs) to create highly personalized and dynamic user experiences on-the-fly. I wanted to make sure to at least give you a primer on GenUI in this book, because I believe that it is one of the greenest green field opportunities that currently exists in the realm of application design and frameworks . I'm convinced that dozens or more new successful commercial and open-source projects will emerge in this particular niche.

At its core, GenUI combines the principles of Contextual Content Generation with advanced AI techniques to generate user interface elements, such as text, images, and layouts, dynamically based on a deep understanding of the user's context, preferences, and goals. GenUI enables designers and developers to create interfaces that adapt and evolve in response to user interactions, providing a level of personalization that was previously unattainable.

GenUI represents a fundamental change in the way we approach user interface design. Instead of designing for the masses, GenUI allows us to design for the individual. Personalized content and interfaces have the potential of creating user experiences that resonate with each user on a deeper level, increasing engagement, satisfaction, and loyalty.

As a bleeding-edge technique, transitioning to GenUI is full of conceptual and practical challenges . Integrating AI into the design process, ensuring that the generated interfaces are not only personalized but also usable, accessible, and aligned with the overall brand and user experience, all of these are challenges that make GenUI a pursuit for the few, not the many. Additionally, the involvement of AI raises questions about data privacy, transparency, and perhaps even ethical implications .

Despite the challenges, personalized experiences at scale have the power to completely transform the way we interact with digital products and services. It opens up possibilities for creating inclusive and accessible interfaces that cater to the diverse needs of users, regardless of their abilities, backgrounds, or preferences .

In this chapter, we will explore the concept of GenUI, examining some defining characteristics, key benefits, and potential challenges. We begin by considering the most basic and accessible form of GenUI: generating text copy for otherwise traditionally designed and implemented user interfaces.

Generating Copy for User Interfaces

Text elements that exist in your application's chrome, such as form labels, tooltips, and explanatory text, are typically hardcoded into the templates or UI components, providing a consistent but generic experience for all users. Using contextual content generation patterns , you can transform these static elements into dynamic, context-aware, and personalized components.

Personalized Forms

Forms are a ubiquitous part of web and mobile applications, serving as the primary means of collecting user input. However, traditional forms often present a generic and impersonal experience, with standard labels and fields that may not always align with the user's specific context or needs. Users are more likely to complete forms that feel tailored to their needs and preferences, leading to higher conversion rates and user satisfaction.

However, it's important to strike a balance between personalization and consistency. While adapting forms to individual users can be beneficial, it's crucial to maintain a level of familiarity and predictability. Users should still be able to recognize and navigate forms easily, even with personalized elements.

Here are some personalized form ideas for inspiration:

Contextual Field Suggestions

GenUI can analyze the user's previous interactions, preferences, and data to provide intelligent field suggestions as predictions. For instance, if the user has previously entered their shipping address, the form can automatically populate the relevant fields with their saved information. This not only saves time but also demonstrates that the application understands and remembers the user's preferences.

Wait a minute, isn't this technique something that could be done without involving AI? Of course, but the beauty of driving this kind of functionality with AI is two-fold: 1) how easy it can be to implement and 2) how resilient it can be as your UI changes and evolves over time.

Let's whip up a service for our theoretical order handling system, that tries to proactively fill in the right shipping address for the user.

```ruby
class OrderShippingAddressSubscriber
  include Raix::ChatCompletion

  attr_accessor :order

  delegate :customer, to: :order

  DIRECTIVE = "You are a smart order processing assistant. Given the
  customer's order history, guess the most likely shipping address
  for the current order."

  def order_created(order)
    return unless order.pending? && order.shipping_address.blank?

    self.order = order

    transcript.clear
    transcript << { system: DIRECTIVE }
    transcript << { user: "Order History: #{order_history.to_json}" }
    transcript << { user: "Current Order: #{order.to_json}" }
```

```
    response = chat_completion
    apply_predicted_shipping_address(order, response)
  end

  private

  def apply_predicted_shipping_address(order, response)
    # extract the shipping address from the response...
    # ...and assume there's some sort of live update of the address fields
    order.update(shipping_address:)
  end

  def order_history
    customer.orders.successful.limit(100).map do |order|
      {
        date: order.date,
        description: order.description,
        shipping_address: order.shipping_address
      }
    end
  end
end
```

This example is very simplified, but should work for most cases. The idea is to let the AI take a guess the same way that a human would. To make it clear what I'm talking about, let's consider some sample data:

```
Order History:
[
  {"date": "2024-01-03", "description": "garden soil mix",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-01-15", "description": "hardcover fiction novels",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-01-22", "description": "baby diapers", "shipping_address":
   "789 Suburb St, Quietville"},
  {"date": "2024-02-01", "description": "organic vegetables",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-02-17", "description": "mystery thriller book set",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-02-25", "description": "baby wipes",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-03-05", "description": "flower seeds",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-03-20", "description": "biographies",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-03-30", "description": "baby formula",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-04-12", "description": "lawn fertilizer",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-04-22", "description": "science fiction novels",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-05-02", "description": "infant toys",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-05-14", "description": "outdoor grill",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-05-29", "description": "literary classics",
  "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-06-11", "description": "baby clothes",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-07-01", "description": "watering can",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-07-18", "description": "non-fiction essays",
  "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-07-28", "description": "baby bath items",
  "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-08-09", "description": "herb garden kit",
  "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-08-24", "description": "children's books",
  "shipping_address": "456 City Apt, Metroville"}
```

```
]
```

Did you notice the pattern in the data? I guarantee you this is easy stuff for an LLM . To demonstrate, let's ask GPT-4o what the most likely shipping address is for a "thermometer".

```
From the order history you've provided, it looks like the purchases are
generally clustered into three main types based on the shipping addresses:

123 Country Lane, Rural Town - This address often orders garden and
outdoor-related items like soil mix, vegetables, flower seeds, lawn
fertilizer, an outdoor grill, a watering can, and an herb garden kit.

456 City Apt, Metroville - This address tends to order various types of books,
indicating an interest in reading across genres like fiction, thrillers,
biographies, science fiction, literary classics, non-fiction essays, and
children's books.

789 Suburb St, Quietville - The purchases for this address are consistently
baby-related products, such as diapers, wipes, baby formula, infant toys,
baby clothes, and baby bath items.

Given these patterns, a digital thermometer could potentially be a fit for
any of the addresses based on different needs: outdoor temperature tracking,
a tool for a passionate reader interested in science gadgets, or a health
tool for a baby. However, considering its common use in monitoring health,
especially for babies, it's most likely to be purchased by someone who's
already buying baby care products.

Therefore, the most likely shipping address for a digital thermometer, based
on the purchasing patterns, would be 789 Suburb St, Quietville.
```

If you're thinking to yourself that it's overkill to use an expensive model like GPT-4 for this task, you're right! I tried the same prompt on Mistral 7B Instruct and it produced the following response at 75 tokens per second, and a miserly cost of $0.000218 USD.

```
Based on the given order history, it appears that the customer has received
orders for baby-related items (diapers, wipes, formula, clothes, bath items)
at the address "789 Suburb St, Quietville" and garden-related items (soil,
seeds, watering can, herb garden kit) at the address "123 Country Lane, Rural
Town". The customer has also received books (fiction, non-fiction, literary
classics, mystery-thriller, science fiction, biographies, children's books)
at the address "456 City Apt, Metroville".

Given that the new order is for a "thermometer", which doesn't seem to fit
into any of the categories that the customer has previously ordered, it's
hard to accurately predict the shipping address based solely on the order
history. However, given the 50%-50% split between baby-related and
garden-related items, it could somewhat lean towards the Baby-related items
address ("789 Suburb St, Quietville"). But remember, this is an assumption
and cannot be definitively confirmed without more context or information.
```

Is the overhead and cost of this technique worth it to make a checkout experience more magical? For many online retailers , absolutely. And from the looks of it, the cost of AI computing is only going to go down, especially for commodity open source model hosting providers in a race to the bottom.

Use a Prompt Template and StructuredIO along with Response Fencing to optimize this kind of chat completion.

Adaptive Field Ordering

The order in which form fields are presented can significantly impact the user's experience and completion rates. With GenUI , you can dynamically adjust the field ordering based on the user's context and the importance of each field. For example, if the user is filling out a registration form for a fitness app, the form could prioritize fields related to their fitness goals and preferences, making the process more relevant and engaging.

Personalized Microcopy

The instructional text, error messages, and other microcopy associated with forms can also be personalized using GenUI . Instead of displaying generic error messages like “Invalid email address,” you can generate more helpful and contextual messages such as “Please enter a valid email address to receive your order confirmation.” These personalized touches can make the form experience more user-friendly and less frustrating.

Personalized Validation

Along the same lines of Personalized Microcopy , you could use AI to validate the form in ways that seem magical. Imagine letting an AI validate a user profile form, looking for potential mistakes on a *semantic* level.

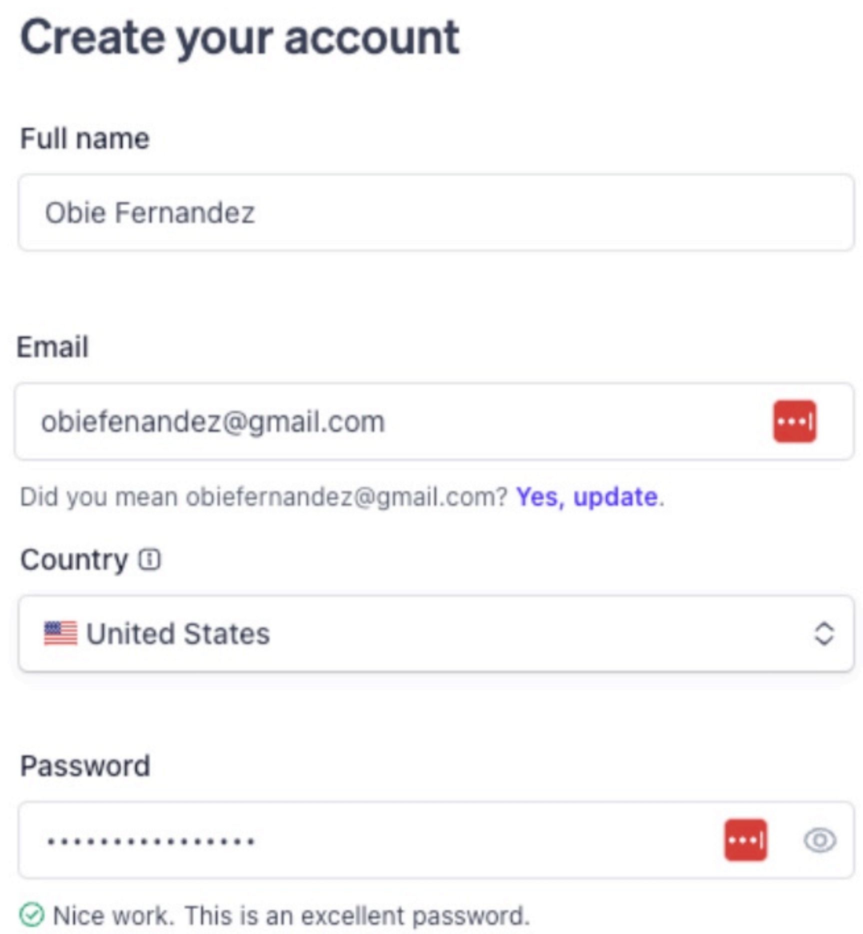


Figure 9. Can you spot the semantic validation happening?

Progressive Disclosure

GenUI can intelligently determine which form fields are essential based on the user's context and gradually reveal additional fields as needed. This progressive disclosure technique helps reduce cognitive load and makes the form-filling process more manageable. For instance, if a user is signing up for a basic subscription, the form can initially present only the essential fields, and as the user progresses or selects specific options, additional relevant fields can be dynamically introduced.

Context-Aware Explanatory Text

Tooltips are often used to provide additional information or guidance to users when they hover over or interact with specific elements. With a "Contextual Content Generation" approach, you can generate tooltips that adapt to the user's context and provide relevant information. For instance, if a user is exploring a complex feature, the tooltip can offer personalized tips or examples based on their previous interactions or skill level.

Explanatory text, such as instructions, descriptions, or help messages, can be dynamically generated based on the user's context. Instead of presenting generic explanations, you can use LLMs to generate text that is tailored to the user's specific needs or questions. For example, if a user is struggling with a particular step in a process, the explanatory text can provide personalized guidance or troubleshooting tips.

Microcopy refers to the small pieces of text that guide users through your application, such as button labels, error messages, or confirmation prompts. By applying the Contextual Content Generation approach to microcopy, you can create an adaptive UI that responds to the user's actions and provides relevant and helpful text. For instance, if a user is about to perform a critical action, the confirmation prompt can be generated dynamically to provide a clear and personalized message.

Personalized explanatory text and tooltips can greatly enhance the onboarding process for new users. By providing context-specific guidance and examples, you can help users quickly understand and navigate the application, reducing the learning curve and increasing adoption.

Dynamic and context-aware chrome elements can also make the application feel more intuitive and engaging. Users are more likely to interact with and explore features when the accompanying text is tailored to their specific needs and interests.

So far we've covered ideas for enhancing existing UI paradigms with AI, but what about rethinking how user interfaces are designed and implemented in a more radical way?

Defining Generative UI

Unlike traditional UI design, where designers create fixed, static interfaces, GenUI hints at a future in which our software boasts flexible, personalized experiences that can evolve and adapt in real-time. Every time we use an AI-driven conversational interface, we are letting the AI adapt to the user's particular needs. GenUI takes things a step further by applying that level of adaptability to software's *visual* interface .

The reason that it's possible to play with GenUI ideas today is that large language models already understand programming and their base knowledge includes UI technologies and frameworks . The question is thus whether large language models can be used to generate UI elements, such as text, images, layouts, and even entire interfaces, that are tailored to each individual user. The model could be instructed to take into account various factors, such as the user's past interactions, stated preferences, demographic information, and the current context of use, to create highly personalized and relevant interfaces.

GenUI differs from traditional user interface design in several key ways:

1. **Dynamic and Adaptive:** Traditional UI design involves creating fixed, static interfaces that remain the same for all users. In contrast, GenUI enables interfaces that can dynamically adapt and change based on user needs and context. This means that the same application can present different interfaces to different users or even to the same user in different situations.
2. **Personalization at Scale:** With traditional design, creating personalized experiences for each user is often impractical due to the time and resources required. GenUI, on the other hand, allows for personalization at scale. By leveraging AI , designers can create interfaces that automatically adapt to each user's unique needs and preferences, without having to manually design and develop separate interfaces for each user segment.
3. **Focus on Outcomes:** Traditional UI design often focuses on creating visually appealing and functional interfaces. While these aspects are still important in GenUI, the primary focus shifts towards achieving desired user outcomes. GenUI aims to create interfaces that are optimized for each user's specific goals and tasks, prioritizing usability and effectiveness over purely aesthetic considerations.
4. **Continuous Learning and Improvement:** GenUI systems can continuously learn and improve over time based on user interactions and feedback. As users engage with the generated interfaces, the AI models can gather data on user behavior, preferences, and outcomes, using this information to refine and optimize future interface generations. This iterative learning process allows GenUI systems to become increasingly effective at meeting user needs over time.

It's important to note that GenUI is not the same as AI-assisted design tools, such as those that provide suggestions or automate certain design tasks. While these tools can be helpful in streamlining the design process, they still rely on designers to make final decisions and create static interfaces. GenUI, on the other hand, involves the AI system taking a more active role in the actual generation and adaptation of interfaces based on user data and context.

GenUI represents a significant shift in how we approach user interface design, moving away from one-size-fits-all solutions and towards highly personalized, adaptive experiences. By leveraging the power of AI, GenUI has the potential to revolutionize the way we interact with digital products and services, creating interfaces that are more intuitive, engaging, and effective for each individual user.

Example

To illustrate the concept of GenUI, let's consider a hypothetical fitness application called "FitAI" . This app aims to provide personalized workout plans and nutrition advice to users based on their individual goals, fitness levels, and preferences.

In a traditional UI design approach, FitAI might have a fixed set of screens and elements that are the same for all users. However, with GenUI, the app's interface could dynamically adapt to each user's unique needs and context.

This approach is kind of a stretch to imagine implementing in 2024 and might not even have adequate ROI, but it is possible.

Here's how it might work:

1. **Onboarding:**
    - Instead of a standard questionnaire, FitAI uses a conversational AI to gather information about the user's goals, current fitness level, and preferences.

- Based on this initial interaction, the AI generates a personalized dashboard layout, highlighting the features and information most relevant to the user's goals.
- Current AI technology might have a selection of screen components at its disposal to use in composing the personalized dashboard.
- Future AI technology might take on the role of an experienced UI designer and actually create the dashboard *from scratch*.

2. **Workout Planner:**

- The workout planner interface is adapted by the AI based to specifically match the user's experience level and available equipment.
- For a beginner with no equipment, it might show simple bodyweight exercises with detailed instructions and videos.
- For an advanced user with access to a gym, it could display more complex routines with less explanatory content.
- The content of the workout planner is not simply filtered from a large superset. It can be generated *on the fly* based on a knowledge base that is queried with context that includes everything known about the user.

3. **Progress Tracking:**

- The progress tracking interface evolves based on the user's goals and engagement patterns.
- If a user is primarily focused on weight loss, the interface might prominently display a weight trend graph and calorie burn statistics.
- For a user building muscle, it could highlight strength gains and body composition changes.

- The AI can adapt this part of the application to the user's actual progress. If the progress stops for a period of time, the app can shift into a mode where it tries to coax the user into divulging the reasons for the setback, in order to mitigate them.

4. **Nutrition Advice:**

   - The nutrition section adapts to the user's dietary preferences and restrictions.
   - For a vegan user, it might show plant-based meal suggestions and protein sources.
   - For a user with a gluten intolerance, it would automatically filter out gluten-containing foods from recommendations.
   - Again, the content is not drawn from a massive superset of meal data that applies to all users, but rather is synthesized from a knowledge base that contains information adaptable based on the user's specific situation and constraints.
   - For instance, recipes are generated with ingredient specifications that match the constantly-changing caloric needs of the user as their fitness level and body stats evolve.

5. **Motivational Elements:**

   - The app's motivational content and notifications are personalized based on the user's personality type and response to different motivational strategies .
   - Some users might receive encouraging messages, while others get more data-driven feedback.

In this example, GenUI enables FitAI to create a highly customized experience for each user, potentially increasing engagement, satisfaction, and the likelihood of achieving fitness goals. The interface elements, content, and even the app's "personality" adapt to best serve each individual user's needs and preferences.

The Shift to Outcome-Oriented Design

GenUI represents a fundamental shift in the approach to user interface design !, moving from a focus on creating specific interface elements to a more holistic, outcome-oriented approach. This shift has several important implications:

1. **Focus on User Goals:**

    - Designers will need to think more deeply about user goals and desired outcomes rather than specific interface components.
    - The emphasis will be on creating systems that can generate interfaces that help users achieve their objectives efficiently and effectively.
    - New UI frameworks will emerge that give AI-based designers the tools they need to be able to generate user experiences *on the fly* and *from scratch* instead of based on predefined screen specifications .

2. **Changing Role of Designers:**

    - Designers will transition from creating fixed layouts to defining rules, constraints, and guidelines for AI systems to follow when generating interfaces.
    - They will need to develop skills in areas such as data analysis, AI prompt engineering , and system thinking to effectively guide GenUI systems.

3. **Importance of User Research:**

    - User research becomes even more critical in a GenUI context, as designers need to understand not just user preferences, but also how these preferences and needs change in different contexts.
    - Continuous user testing and feedback loops will be essential to refine and improve the AI's ability to generate effective interfaces.

4. **Designing for Variability:**

    - Instead of creating a single "perfect" interface, designers will need to consider multiple possible variations and ensure that the system can generate appropriate interfaces for diverse user needs.
    - This includes designing for edge cases and ensuring that the generated interfaces maintain usability and accessibility across different configurations.
    - Product differentiation takes on new dimensions involving divergent perspectives on user psychology and the leveraging of unique data sets and knowledge bases unavailable to competitors.

Challenges and Considerations

While GenUI offers exciting possibilities, it also presents several challenges and considerations:

1. **Technical Limitations:**

    - Current AI technology, while advanced, still has limitations in understanding complex user intents and generating truly context-aware interfaces.

- Performance issues related to real-time generation of interface elements, especially on less powerful devices.

2. **Data Requirements:**

    - Depending on the use case, effective GenUI systems might require significant amounts of user data to generate personalized interfaces.
    - The challenges in ethically sourcing authentic user data raise concerns about data privacy and security , as well as potential biases in the data used to train GenUI models.

3. **Usability and Consistency:**

    - At least until the practice becomes widespread, an application with constantly changing interfaces could lead to usability issues , as users may struggle to find familiar elements or navigate efficiently.
    - Striking a balance between personalization and maintaining a consistent, learnable interface will be crucial.

4. **Overreliance on AI:**

    - There's a risk of over-delegating design decisions to AI systems, potentially leading to uninspired, problematic, or simply broken interface choices.
    - Human oversight and the ability to override AI-generated designs will remain important in the foreseeable future.

5. **Accessibility Concerns:**

- Ensuring that dynamically generated interfaces remain accessible to users with disabilities presents entirely new challenges, which is worrying given the poor level of accessibility compliance demonstrated by typical systems .
- On the other hand, AI designers may be implemented with *built-in* concern for accessibility, and capabilities for building accessible interfaces on the fly just like they build UI for non-impaired users.
- Either way, GenUI systems should be designed with robust accessibility guidelines and testing processes.

6. **User Trust and Transparency:**

- Users may feel uncomfortable with interfaces that seem to "know too much" about them or change in ways they don't understand.
- Providing transparency about how and why interfaces are personalized will be important for building user trust .

Future Outlook and Opportunities

The future of Generative UI (GenUI) holds immense promise for revolutionizing the way we interact with digital products and services. As this technology continues to evolve, we can anticipate a seismic shift in how user interfaces are designed, implemented, and experienced. I think GenUI is the phenomenon that will finally push our software into the realm of what is now considered science fiction.

One of the most exciting prospects of GenUI is its potential to enhance accessibility on a grand scale that goes beyond simply making sure that people with serious disabilities are not completely excluded from the use of your software. By automatically adapting interfaces to individual user needs, GenUI could make digital experiences more inclusive than ever before. Imagine interfaces that seamlessly adjust to provide larger text for younger or visually impaired users or simplified layouts for those with cognitive disabilities, all without requiring manual configuration or separate “accessible” versions of applications.

The personalization capabilities of GenUI are likely to drive increased user engagement, satisfaction, and loyalty across a wide range of digital products. As interfaces become more attuned to individual preferences and behaviors, users will find digital experiences more intuitive and enjoyable, potentially leading to deeper and more meaningful interactions with technology.

GenUI also has the potential to transform the onboarding process for new users. By creating intuitive, personalized first-time user experiences that quickly adapt to each user’s level of expertise, GenUI could significantly reduce the learning curve associated with new applications. This could lead to faster adoption rates and increased user confidence in exploring new features and functionalities.

Another exciting possibility is the ability of GenUI to maintain a consistent user experience across different devices and platforms while optimizing for each specific context of use. This could solve the long-standing challenge of providing coherent experiences across an increasingly fragmented device landscape, from smartphones and tablets to desktop computers and emerging technologies like augmented reality glasses .

The data-driven nature of GenUI opens up opportunities for rapid iteration and improvement in UI design . By gathering real-time data on how users interact with generated interfaces, designers and developers can gain unprecedented insights into user behavior and preferences. This feedback loop could lead to continuous improvements in UI design, driven by actual usage patterns rather than assumptions or limited user testing.

To prepare for this shift, designers will need to evolve their skill sets and mindsets. The focus will shift from creating fixed layouts to developing comprehensive design systems and guidelines that can inform AI-driven interface generation. Designers will need to cultivate a deep understanding of data analysis, AI technologies, and systems thinking to effectively guide GenUI systems.

Moreover, as GenUI blurs the lines between design and technology, designers will need to collaborate more closely with developers and data scientists. This interdisciplinary approach will be crucial in creating GenUI systems that are not only visually appealing and user-friendly but also technically robust and ethically sound.

The ethical implications of GenUI will also come to the forefront as the technology matures. Designers will play a crucial role in developing frameworks for responsible AI use in interface design, ensuring that personalization enhances user experiences without compromising privacy or manipulating user behavior in unethical ways.

As we look to the future, GenUI presents both exciting opportunities and significant challenges. It has the potential to create more intuitive, efficient, and satisfying digital experiences for users across the globe. While it will require designers to adapt and acquire new skills, it also offers an unprecedented opportunity to shape the future of human-computer interaction in profound and meaningful ways. The journey towards fully realized GenUI systems will undoubtedly be complex, but the potential rewards in terms of improved user experiences and digital accessibility make it a future worth striving for.

Intelligent Workflow Orchestration

In the realm of application development, workflows play a crucial role in defining how tasks, processes, and user interactions are structured and executed. As applications become more complex and user expectations continue to rise, the need for intelligent and adaptive workflow orchestration becomes increasingly apparent.

The "Intelligent Workflow Orchestration" approach focuses on leveraging AI components to dynamically orchestrate and optimize complex workflows within applications. The goal is to create applications that are more efficient, responsive, and adaptable to real-time data and context.

In this chapter, we will explore the key principles and patterns that underpin the intelligent workflow orchestration approach. We will consider how AI can be used to intelligently route tasks, automate decision-making, and dynamically adapt workflows based on various factors such as user behavior, system performance, and business rules . Through practical examples and real-world scenarios, we will demonstrate the transformative potential of AI in streamlining and optimizing application workflows.

Whether you are building enterprise applications with intricate business processes or consumer-facing applications with dynamic user journeys, the patterns and techniques discussed in this chapter will equip you with the knowledge and tools to create intelligent and efficient workflows that enhance the overall user experience and drive business value.

Business Need

Traditional approaches to workflow management often rely on predefined rules and static decision trees , which can be rigid, inflexible, and unable to cope with the dynamic nature of modern applications.

Consider a scenario where an e-commerce application needs to handle a complex order fulfillment process. The workflow may involve multiple steps such as order validation, inventory check, payment processing, shipping, and customer notifications. Each step may have its own set of rules, dependencies, external integrations, and exception handling mechanisms. Managing such a workflow manually or through hardcoded logic can quickly become cumbersome, error-prone, and difficult to maintain.

Moreover, as the application scales and the number of concurrent users grows, the workflow may need to adapt and optimize itself based on real-time data and system performance. For example, during peak traffic periods, the application may need to dynamically adjust the workflow to prioritize certain tasks, allocate resources efficiently, and ensure a smooth user experience.

This is where the "Intelligent Workflow Orchestration" approach comes into play. By leveraging AI components, developers can create workflows that are intelligent, adaptive, and self-optimizing. AI can analyze vast amounts of data, learn from past experiences, and make informed decisions in real-time to orchestrate the workflow effectively.

Key Benefits

1. **Increased Efficiency:** AI can optimize task allocation, resource utilization, and workflow execution, leading to faster processing times and improved overall efficiency .
2. **Adaptability:** AI-driven workflows can dynamically adapt to changing conditions, such as fluctuations in user demand, system performance, or business requirements, ensuring that the application remains responsive and resilient.
3. **Automated Decision-Making:** AI can automate complex decision-making processes within the workflow, reducing manual intervention and minimizing the risk of human errors.
4. **Personalization:** AI can analyze user behavior, preferences, and context to personalize the workflow and deliver tailored experiences to individual users .
5. **Scalability:** AI-powered workflows can scale seamlessly to handle increasing volumes of data and user interactions, without compromising performance or reliability .

In the following sections, we will explore the key patterns and techniques that enable the implementation of intelligent workflows and showcase real-world examples of how AI is transforming workflow management in modern applications .

Key Patterns

To implement intelligent workflow orchestration in applications, developers can leverage several key patterns that harness the power of AI. These patterns provide a structured approach to designing and managing workflows, enabling applications to adapt, optimize, and automate processes based on real-time data and context. Let's explore some of the fundamental patterns in intelligent workflow orchestration.

Dynamic Task Routing

This pattern involves using AI to intelligently route tasks within a workflow based on various factors such as task priority, resource availability, and system performance. AI algorithms can analyze the characteristics of each task, consider the current state of the system, and make informed decisions to assign tasks to the most appropriate resources or processing paths. Dynamic task routing ensures that tasks are efficiently distributed and executed, optimizing the overall workflow performance .

```
class TaskRouter
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  attr_accessor :task

  # list of functions that can be called by the AI entirely at its
  # discretion depending on the task received

  function :analyze_task_priority do
    TaskPriorityAnalyzer.perform(task)
  end

  function :check_resource_availability, # ...
  function :assess_system_performance, # ...
```

```
16    function :assign_task_to_resource, # ...
17
18    DIRECTIVE = "You are a task router, responsible for intelligently
19     assigning tasks to available resources based on priority, resource
20     availability, and system performance..."
21
22    def initialize(task)
23      self.task = task
24      transcript << { system: DIRECTIVE }
25      transcript << { user: task.to_json }
26    end
27
28    def perform
29      while task.unassigned?
30        chat_completion
31
32        # todo: add max loop counter and break
33      end
34
35      # capture the transcript for later analysis
36      task.update(routing_transcript: transcript)
37    end
38  end
```

Note the loop created by the `while` expression on line 29, which continues prompting the AI until the task is assigned. On line 35, we save the transcript on the task for later analysis and debugging , if it becomes necessary.

Contextual Decision Making

You can use very similar code to make context-aware decisions within a workflow. By analyzing relevant data points such as user preferences, historical patterns , and real-time inputs, AI components can determine the most appropriate course of action at each decision point in the workflow. Adapt the behavior of your workflow based on the specific context of each user or scenario, providing personalized and optimized experiences.

Adaptive Workflow Composition

This pattern focuses on dynamically composing and adjusting workflows based on changing requirements or conditions. AI can analyze the current state of the workflow, identify bottlenecks or inefficiencies, and automatically modify the workflow structure to optimize performance. Adaptive workflow composition allows applications to continuously evolve and improve their processes without requiring manual intervention.

Exception Handling and Recovery

Exception handling and recovery are critical aspects of intelligent workflow orchestration. When working with AI components and complex workflows, it's essential to anticipate and handle exceptions gracefully to ensure the stability and reliability of the system.

Here are some key considerations and techniques for exception handling and recovery in intelligent workflows:

1. **Exception Propagation:** Implement a consistent approach for propagating exceptions across workflow components. When an exception occurs within a component, it should be caught, logged, and propagated to the orchestrator or a discrete component responsible for handling exceptions . The idea is to centralize exception handling and prevent exceptions from being silently swallowed, as well as opening possibilities for Intelligent Error Handling.
2. **Retry Mechanisms:** Retry mechanisms help improve the resilience of the workflow and handle intermittent failures gracefully. Definitely try to implement retry mechanisms for transient or recoverable exceptions, such as a network connectivity or resource unavailability

that can be automatically retried after a specified delay. Having an AI-powered orchestrator or exception handler means that your retry strategies do not have to be mechanical in nature, relying on fixed algorithms like exponential fallback. You can leave the handling of the retry up to the "discretion" of the AI componente responsible for deciding how to handle the exception.

3. **Fallback Strategies:** If an AI component fails to provide a valid response or encounters an error—a common occurrence given its bleeding-edge nature—have a fallback mechanism in place to ensure the workflow can continue. This could involve using default values, alternative algorithms, or a Human In The Loop to make decisions and keep the workflow moving forward.
4. **Compensating Actions:** The orchestrators directives should include instructions about compensating actions to handle exceptions that cannot be resolved automatically. Compensating actions are steps taken to undo or mitigate the effects of a failed operation. For example, if a payment processing step fails, a compensating action could be to roll back the transaction and notify the user. Compensating actions help maintain data consistency and integrity in the face of exceptions.
5. **Exception Monitoring and Alerting:** Set up monitoring and alerting mechanisms to detect and notify relevant stakeholders about critical exceptions. The orchestrator can be made aware of thresholds and rules to trigger alerts when exceptions exceed certain limits or when specific types of exceptions occur. This allows for proactive identification and resolution of issues before they impact the overall system.

Here's an example of exception handling and recovery in a Ruby workflow component:

```ruby
class InventoryManager
  def check_availability(order)
    begin
      # Perform inventory check logic
      inventory = Inventory.find_by(product_id: order.product_id)
      if inventory.available_quantity >= order.quantity
        return true
      else
        raise InsufficientInventoryError,
              "Insufficient inventory for product #{order.product_id}"
      end
    rescue InsufficientInventoryError => e
      # Log the exception
      logger.error("Inventory check failed: #{e.message}")

      # Retry the operation after a delay
      retry_count ||= 0
      if retry_count < MAX_RETRIES
        retry_count += 1
        sleep(RETRY_DELAY)
        retry
      else
        # Fallback to manual intervention
        NotificationService.admin("Inventory check failed: Order #{order.id}")
        return false
      end
    end
  end
end
```

In this example, the `InventoryManager` component checks the availability of a product for a given order. If the available quantity is insufficient, it raises an `InsufficientInventoryError`. The exception is caught, logged, and a retry mechanism is implemented. If the retry limit is exceeded, the component falls back to manual intervention by notifying an admin .

By implementing robust exception handling and recovery mechanisms , you can ensure that your intelligent workflows are resilient, maintainable, and able to handle unexpected situations gracefully.

These patterns form the foundation of intelligent workflow orchestration and can be combined and adapted to suit the specific requirements of different applications . By leveraging these patterns, developers can create workflows that are flexible, resilient, and optimized for performance and user experience.

In the next section, we will explore how these patterns can be implemented in practice, using real-world examples and code snippets to illustrate the integration of AI components into workflow management.

Implementing Intelligent Workflow Orchestration in Practice

Now that we have explored the key patterns in intelligent workflow orchestration, let's dive into how these patterns can be implemented in real-world applications. We'll provide practical examples and code snippets to illustrate the integration of AI components into workflow management.

Intelligent Order Processor

Let's dive into a practical example of implementing intelligent workflow orchestration using an AI-powered `OrderProcessor` component in a Ruby on Rails e-commerce application. The `OrderProcessor` realizes the Process Manager Enterprise Integration concept that we first encountered in Chapter 3 when discussing Multitude of Workers. The component will be responsible for managing order fulfillment workflow, making routing decisions based on intermediate results, and orchestrating the execution of various processing steps.

The order fulfillment process involves multiple steps such as order validation, inventory check, payment processing, and shipping. Each step is implemented as a separate worker process that performs a specific task and returns the result to the `OrderProcessor`. The steps are not mandatory, and don't even necessarily have to be done in a precise order.

Here's an example implementation of the `OrderProcessor`. It features two mixins from Raix. The first (`ChatCompletion`) gives it the ability to do chat completion, which is what makes this an AI component. The second (`FunctionDispatch`) enables function calling by the AI, allowing it to respond to a prompt with a function invocation instead of a text message.

The worker functions (`validate_order`, `check_inventory`, et al) delegate to their respective worker classes, which can be AI or non-AI components, with the only requirement being that they return the results of their work in a format that can be represented as a string.

> As with all other examples in this part of the book, this code is practically pseudo-code and is only meant to convey the meaning of the pattern and inspire your own creations. Full descriptions of patterns and complete code examples are included in Part 2.

```ruby
class OrderProcessor
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are an order processor, tasked with..."

  def initialize(order)
    self.order = order
    transcript << { system: SYSTEM_DIRECTIVE }
    transcript << { user: order.to_json }
  end

  def perform
    # will continue looping until `stop_looping!` is called
    chat_completion(loop: true)
  end

  # list of functions available to be called by the AI
  # truncated for brevity

  def functions
    [
      {
        name: "validate_order",
        description: "Invoke to check validity of order",
        parameters: {
          ...
      },
      ...
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def validate_order
    OrderValidationWorker.perform(@order)
  end

  def check_inventory
    InventoryCheckWorker.perform(@order)
  end
```

```ruby

  def process_payment
    PaymentProcessingWorker.perform(@order)
  end

  def schedule_shipping
    ShippingSchedulerWorker.perform(@order)
  end

  def send_confirmation
    OrderConfirmationWorker.perform(@order)
  end

  def finished_processing
    @order.update!(transcript:, processed_at: Time.current)
    stop_looping!
  end
end
```

In the example, the OrderProcessor is initialized with an order object and maintains a transcript of the workflow execution, in the typical conversation transcript format that is native to large language models . Complete control is given to the AI to orchestrate the execution of various processing steps, such as order validation, inventory check, payment processing, and shipping.

Everytime the `chat_completion` method is called, the transcript is sent to the AI for it to provide a completion as a function call. It is entirely up to the AI to analyze the result of the previous step and determine the appropriate action to take. For example, if the inventory check reveals low stock levels, the `OrderProcessor` can schedule a replenishment task. If the payment processing fails, it can initiate a retry or notify customer support.

> The example above does not have functions defined for replenishment or notifying customer support, but it absolutely could.

The transcript grows everytime a function is called and serves as a record of the workflow execution, including the results of each step and the AI-generated instructions for the next steps. This transcript can be used for debugging, auditing, and providing visibility into the order fulfillment process.

By leveraging AI in the `OrderProcessor`, the e-commerce application can dynamically adapt the workflow based on real-time data and handle exceptions intelligently. The AI component can make informed decisions, optimize the workflow, and ensure smooth order processing even in complex scenarios.

The fact that the only requirement on the worker processes is to return some intelligible output for the AI to consider when deciding what to do next, it might start to dawn on you how this approach can cut down on the input/output mapping work that is typically involved when integrating disparate systems with each other.

Intelligent Content Moderator

Social media applications generally require at least minimal content moderation to ensure a safe and healthy community. This example `ContentModerator` component leverages AI to intelligently orchestrate the moderation workflow, making decisions based on the content's characteristics and the results of various moderation steps.

The moderation process involves multiple steps such as text analysis, image recognition, user reputation assessment, and manual review. Each step is implemented as a separate worker process that performs a specific task and returns the result to the `ContentModerator`.

Here's an example implementation of the `ContentModerator`:

```
class ContentModerator
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a content moderator process manager,
    tasked with the workflow involved in moderating user-generated content..."

  def initialize(content)
    @content = content
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: content.to_json }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

  # list of functions available to be called by the AI
  # truncated for brevity

  def functions
    [
      {
        name: "analyze_text",
        # ...
      },
      {
        name: "recognize_image",
        description: "Invoke to describe images...",
```

```ruby
          # ...
        },
        {
          name: "assess_user_reputation",
          # ...
        },
        {
          name: "escalate_to_manual_review",
          # ...
        },
        {
          name: "approve_content",
          # ...
        },
        {
          name: "reject_content",
          # ...
        }
      ]
    end

    # implementation of functions that can be called by the AI
    # entirely at its discretion, depending on the needs of the order

    def analyze_text
      result = TextAnalysisWorker.perform(@content)
      continue_with(result)
    end

    def recognize_image
      result = ImageRecognitionWorker.perform(@content)
      continue_with(result)
    end

    def assess_user_reputation
      result = UserReputationWorker.perform(@content.user)
      continue_with(result)
    end

    def escalate_to_manual_review
      ManualReviewWorker.perform(@content)
      @content.update!(status: 'pending', transcript: @transcript)
```

```
  end

  def approve_content
    @content.update!(status: 'approved', transcript: @transcript)
  end

  def reject_content
    @content.update!(status: 'rejected', transcript: @transcript)
  end

  private

  def continue_with(result)
    @transcript << { function: result }
    complete(@transcript)
  end
end
```

In this example, the `ContentModerator` is initialized with a content object and maintains a moderation transcript in the conversation format. The AI component has full control over the moderation workflow, deciding which steps to execute based on the content's characteristics and the results of each step.

The available worker functions for the AI to invoke include `analyze_text`, `recognize_image`, `assess_user_reputation`, and `escalate_to_manual_review`. Each function delegates the task to a corresponding worker process (`TextAnalysisWorker`, `ImageRecognitionWorker`, etc.) and appends the result to the moderation transcript, with the exception of the escalation function, which acts as an end state. Finally, the `approve_content` and `reject_content` functions also act as end states.

The AI component analyzes the content and determines the appropriate action to take. If the content contains image references, it can call the `recognize_image` worker for assistance with a visual review. If any worker warns of potentially harmful content, the AI may decide to escalate the content for manual review or just reject it outright. But depending on the severity of the warning, the AI may choose to use the results of the user reputation assessment in deciding how to handle content that it isn't otherwise sure about. Depending on the use case, perhaps trusted users have more leeway in what they can post. And so on, and so forth…

As with the previous process manager example, the moderation transcript serves as a record of the workflow execution, including the results of each step and the AI-generated decisions. This transcript can be used for auditing, transparency, and improving the moderation process over time.

By leveraging AI in the `ContentModerator`, the social media application can dynamically adapt the moderation workflow based on the content's characteristics and handle complex moderation scenarios intelligently. The AI component can make informed decisions, optimize the workflow, and ensure a safe and healthy community experience.

Let's explore two more examples that demonstrate predictive task scheduling and exception handling and recovery within the context of intelligent workflow orchestration.

Predictive Task Scheduling in a Customer Support System

In a customer support application built with Ruby on Rails , efficiently managing and prioritizing support tickets is crucial for providing timely assistance to customers. The `SupportTicketScheduler` component leverages AI to predictively schedule and assign support tickets to available agents based on various factors such as ticket urgency, agent expertise, and workload.

```ruby
class SupportTicketScheduler
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a support ticket scheduler,
    tasked with intelligently assigning tickets to available agents..."

  def initialize(ticket)
    @ticket = ticket
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: ticket.to_json }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

  def functions
    [
      {
        name: "analyze_ticket_urgency",
        # ...
      },
      {
        name: "list_available_agents",
        description: "Includes expertise of available agents",
        # ...
      },
      {
        name: "predict_agent_workload",
        description: "Uses historical data to predict upcoming workloads",
        # ...
      },
      {
        name: "assign_ticket_to_agent",
        # ...
```

```ruby
      },
      {
        name: "reschedule_ticket",
        # ...
      }
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def analyze_ticket_urgency
    result = TicketUrgencyAnalyzer.perform(@ticket)
    continue_with(result)
  end

  def list_available_agents
    result = ListAvailableAgents.perform
    continue_with(result)
  end

  def predict_agent_workload
    result = AgentWorkloadPredictor.perform
    continue_with(result)
  end

  def assign_ticket_to_agent
    TicketAssigner.perform(@ticket, @transcript)
  end

  def delay_assignment(until)
    until = DateTimeStandardizer.process(until)
    SupportTicketScheduler.delay(@ticket, @transcript, until)
  end

  private

  def continue_with(result)
    @transcript << { function: result }
    complete(@transcript)
  end
end
```

In this example, the `SupportTicketScheduler` is initialized with a support ticket object and maintains a scheduling transcript. The AI component analyzes the ticket details and predictively schedules the ticket assignment based on factors like ticket urgency, agent expertise, and predicted agent workload.

The available functions for the AI to invoke include `analyze_ticket_urgency`, `list_available_agents`, `predict_agent_workload`, and `assign_ticket_to_agent`. Each function delegates the task to a corresponding analyzer or predictor component and appends the result to the scheduling transcript. The AI also has the option to delay assignment using the `delay_assignment` function.

The AI component examines the scheduling transcript and makes informed decisions on ticket assignment . It considers the urgency of the ticket, the expertise of available agents, and the predicted workload of each agent to determine the most suitable agent for handling the ticket.

By leveraging predictive task scheduling, the customer support application can optimize ticket assignment, reduce response times, and improve overall customer satisfaction. Proactive and efficient management of support tickets ensures that the right tickets are assigned to the right agents at the right time.

Exception Handling and Recovery in a Data Processing Pipeline

Handling exceptions and recovering from failures is essential to ensure data integrity and prevent data loss . The `DataProcessingOrchestrator` component utilizes AI to intelligently handle exceptions and orchestrate the recovery process in a data processing pipeline .

```ruby
class DataProcessingOrchestrator
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a data processing orchestrator..."

  def initialize(data_batch)
    @data_batch = data_batch
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: data_batch.to_json }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

  def functions
    [
      {
        name: "validate_data",
        # ...
      },
      {
        name: "process_data",
        # ...
      },
      {
        name: "request_fix",
        # ...
      },
      {
        name: "retry_processing",
        # ...
      },
      {
        name: "mark_data_as_failed",
```

```
        # ...
      },
      {
        name: "finished",
        # ...
      }
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def validate_data
    result = DataValidator.perform(@data_batch)
    continue_with(result)
  rescue ValidationException => e
    handle_validation_exception(e)
  end

  def process_data
    result = DataProcessor.perform(@data_batch)
    continue_with(result)
  rescue ProcessingException => e
    handle_processing_exception(e)
  end

  def request_fix(description_of_fix)
    result = SmartDataFixer.new(description_of_fix, @data_batch)
    continue_with(result)
  end

  def retry_processing(timeout_in_seconds)
    wait(timeout_in_seconds)
    process_data
  end

  def mark_data_as_failed
    @data_batch.update!(status: 'failed', transcript: @transcript)
  end

  def finished
    @data_batch.update!(status: 'finished', transcript: @transcript)
```

```
  end

  private

  def continue_with(result)
    @transcript << { function: result }
    complete(@transcript)
  end

  def handle_validation_exception(exception)
    @transcript << { exception: exception.message }
    complete(@transcript)
  end

  def handle_processing_exception(exception)
    @transcript << { exception: exception.message }
    complete(@transcript)
  end
end
```

In this example, the `DataProcessingOrchestrator` is initialized with a data batch object and maintains a processing transcript. The AI component orchestrates the data processing pipeline, handling exceptions and recovering from failures as needed.

The available functions for the AI to invoke include `validate_data`, `process_data`, `request_fix`, `retry_processing`, and `mark_data_as_failed`. Each function delegates the task to a corresponding data processing component and appends the result or exception details to the processing transcript.

If a validation exception occurs during the `validate_data` step, the `handle_validation_exception` function appends the exception data to the transcript and passes control back to the AI. Similarly, if a processing exception occurs during the `process_data` step, the AI can decide on the recovery strategy.

Depending on the nature of the exception encountered, the AI can at its discretion decide to call `request_fix`, which delegates to an AI-powered `SmartDataFixer` component (see Self Healing Data chapter). The data fixer gets a plain english description of how it should modify the `@data_batch` so that processing can be retried. Perhaps a successful retry would entail removing records from the data batch that have failed validation and/or copying them to a different processing pipeline for human review? The possibilities are near endless.

By incorporating AI-driven exception handling and recovery, the data processing application becomes more resilient and fault-tolerant. The `DataProcessingOrchestrator` intelligently manages exceptions, minimizes data loss, and ensures the smooth execution of the data processing workflow.

Monitoring and Logging

Monitoring and logging provide visibility into the progress, performance, and health of AI-powered workflow components, enabling developers to track and analyze the behavior of the system. Implementing effective monitoring and logging mechanisms is essential for debugging, auditing, and continuous improvement of intelligent workflows.

Monitoring Workflow Progress and Performance

To ensure the smooth execution of intelligent workflows, it's important to monitor the progress and performance of each workflow component. This involves tracking key metrics and events throughout the workflow lifecycle.

Some important aspects to monitor include:

1. Workflow Execution Time: Measure the time taken by each workflow component to complete its task. This helps identify performance bottlenecks and optimize the overall workflow efficiency.

2. Resource Utilization: Monitor the utilization of system resources, such as CPU, memory, and storage, by each workflow component. This helps ensure that the system is operating within its capacity and can handle the workload effectively.

3. Error Rates and Exceptions: Track the occurrence of errors and exceptions within workflow components. This helps identify potential issues and enables proactive error handling and recovery.

4. Decision Points and Outcomes: Monitor the decision points within the workflow and the outcomes of AI-powered decisions. This provides insights into the behavior and effectiveness of the AI components.

The data captured by monitoring processes can be surfaced in dashboards or used as inputs to scheduled reports that inform system administrators about the health of the system.

> Monitoring data can be fed to an AI-powered system administrator process for review and potential action!

Logging Key Events and Decisions

Logging is an essential practice that involves capturing and storing relevant information about key events, decisions, and exceptions that occur during the workflow execution.

Some important aspects to log include:

1. Workflow Initiation and Completion: Log the start and end times of each workflow instance, along with any relevant metadata such as the input data and user context.

2. Component Execution: Log the execution details of each workflow component, including the input parameters, output results, and any intermediate data generated.

3. AI Decisions and Reasoning: Log the decisions made by AI components, along with the underlying reasoning or confidence scores. This provides transparency and enables auditing of AI-powered decisions.

4. Exceptions and Error Messages: Log any exceptions or error messages encountered during the workflow execution, including the stack trace and relevant context information.

Logging can be implemented using various techniques, such as writing to log files, storing logs in a database, or sending logs to a centralized logging service. It's important to choose a logging framework that provides flexibility, scalability, and easy integration with the application's architecture.

Here's an example of how logging can be implemented in a Ruby on Rails application using the `ActiveSupport::Logger` class:

```ruby
class WorkflowLogger
  def self.log(message, severity = :info)
    @logger ||= ActiveSupport::Logger.new('workflow.log')
    @logger.formatter ||= proc do |severity, datetime, progname, msg|
      "#{datetime} [#{severity}] #{msg}\n"
    end
    @logger.send(severity, message)
  end
end

# Usage example
WorkflowLogger.log("Workflow initiated for order ##{@order.id}")
WorkflowLogger.log("Payment processing completed successfully")
WorkflowLogger.log("Inventory check failed for item ##{item.id}", :error)
```

By strategically placing logging statements throughout the workflow components and AI decision points, developers can capture valuable information for debugging, auditing, and analysis .

Benefits of Monitoring and Logging

Implementing monitoring and logging in intelligent workflow orchestration offers several benefits:

1. Debugging and Troubleshooting: Detailed logs and monitoring data help developers identify and diagnose issues quickly. They provide insights into the workflow execution flow, component interactions, and any errors or exceptions encountered .

2. Performance Optimization: Monitoring performance metrics allows developers to identify bottlenecks and optimize the workflow components for better efficiency. By analyzing execution times, resource utilization, and other metrics, developers can make informed decisions to improve the overall performance of the system .

3. Auditing and Compliance: Logging key events and decisions provides an audit trail for regulatory compliance and accountability. It enables organizations to track and verify the actions taken by AI components and ensure adherence to business rules and legal requirements .

4. Continuous Improvement: Monitoring and logging data serve as valuable inputs for continuous improvement of intelligent workflows. By analyzing historical data, identifying patterns, and measuring the effectiveness of AI decisions, developers can iteratively refine and enhance the workflow orchestration logic.

Considerations and Best Practices

When implementing monitoring and logging in intelligent workflow orchestration, consider the following best practices:

1. Define Clear Monitoring Metrics: Identify the key metrics and events that need to be monitored based on the specific requirements of the workflow. Focus on metrics that provide meaningful insights into the system's performance, health, and behavior .

2. Implement Granular Logging: Ensure that logging statements are placed at appropriate points within the workflow components and AI decision points. Capture relevant context information, such as input parameters, output results, and any intermediate data generated .

3. Use Structured Logging: Adopt a structured logging format to facilitate easy parsing and analysis of log data. Structured logging allows for better searchability, filtering, and aggregation of log entries .

4. Manage Log Retention and Rotation: Implement log retention and rotation policies to manage the storage and lifecycle of log files. Determine the appropriate retention period based on legal requirements, storage constraints, and analysis needs. If possible, offload logging to a 3rd-party service such as Papertrail .

5. Secure Sensitive Information: Be cautious when logging sensitive information, such as personally identifiable information (PII) or confidential business data. Implement appropriate security measures, such as data masking or encryption, to protect sensitive information in log files.

6. Integrate with Monitoring and Alerting Tools: Leverage monitoring and alerting tools to centralize the collection, analysis, and visualization of monitoring and logging data. These tools can provide real-time insights, generate alerts based on predefined thresholds, and facilitate proactive issue detection and resolution. My favorite of these tools is Datadog .

By implementing comprehensive monitoring and logging mechanisms, developers can gain valuable insights into the behavior and performance of intelligent workflows. These insights enable effective debugging, optimization, and continuous improvement of AI-powered workflow orchestration systems.

Scalability and Performance Considerations

Scalability and performance are critical aspects to consider when designing and implementing intelligent workflow orchestration systems . As the volume of concurrent workflows and the complexity of AI-powered components increase, it becomes essential to ensure that the system can handle the workload efficiently and scale seamlessly to meet growing demands.

Handling High Volumes of Concurrent Workflows

Intelligent workflow orchestration systems often need to handle a large number of concurrent workflows. To ensure scalability, consider the following strategies:

1. Asynchronous Processing: Implement asynchronous processing mechanisms to decouple the execution of workflow components. This allows the system to handle multiple workflows concurrently without blocking or waiting for each component to complete. Asynchronous processing can be achieved using message queues, event-driven architectures, or background job processing frameworks such as Sidekiq .

2. Distributed Architecture: Design the system architecture to use serverless components (such as AWS Lambda) or simply distribute the workload across multiple nodes or servers alongside your main application server. This enables horizontal scalability, where additional nodes can be added to handle increased workflow volumes .

3. Parallel Execution: Identify opportunities for parallel execution within workflows. Some workflow components may be independent of each other and can be executed concurrently. By leveraging parallel processing techniques, such as multi-threading or distributed task queues, the system can optimize resource utilization and reduce overall workflow execution time .

Optimizing Performance of AI-Powered Components

AI-powered components, such as machine learning models or natural language processing engines, can be computationally intensive and impact the overall performance of the workflow orchestration system. To optimize the performance of AI components, consider the following techniques:

1. Caching: If your AI processing is purely generative and does not involve realtime information lookups or external integrations in order to generate its chat completions, then you can look into caching mechanisms to store and reuse the results of frequently accessed or computationally expensive operations .

2. Model Optimization: Continuously optimize the way that you use the AI models in workflow components. This may involve techniques such as *Prompt Distillation* or it might simply be a matter of testing new models as they become available.

3. Batch Processing: If you're working with GPT-4 class models , you migth be able to leverage batch processing techniques to process multiple data points or requests in a single batch, rather than processing them individually. By processing data in batches , the system can optimize resource utilization and reduce the overhead of repeated model requests.

Monitoring and Profiling Performance

To identify performance bottlenecks and optimize the scalability of the intelligent workflow orchestration system , it's crucial to implement monitoring and profiling mechanisms. Consider the following approaches:

1. Performance Metrics: Define and track key performance metrics, such as response time, throughput, resource utilization, and latency. These metrics provide insights into the system's performance and help identify areas for optimization. Popular AI model aggregator OpenRouter includes Host[1] and Speed[^speed] metrics in each API response, making it trivial to track these key metrics.

[^speed] Speed is calculated as the number of completion tokens divided by total generation time. For non-streamed requests latency is considered part of generation time.

[1] Host is the time it took to receive the first byte of the streamed generation from the model host, a.k.a. "time to first byte."

2. Profiling Tools: Utilize profiling tools to analyze the performance of individual workflow components and AI operations. Profiling tools can help identify performance hotspots, inefficient code paths, or resource-intensive operations. Popular profiling tools include New Relic , Scout , or built-in profilers provided by the programming language or framework.

3. Load Testing: Conduct load testing to evaluate the system's performance under different levels of concurrent workloads. Load testing helps identify the system's scalability limits, detect performance degradation, and ensure that the system can handle the expected traffic without compromising performance.

4. Continuous Monitoring: Implement continuous monitoring and alerting mechanisms to proactively detect performance issues and bottlenecks. Set up monitoring dashboards and alerts to track key performance indicators (KPIs) and receive notifications when predefined thresholds are breached. This enables prompt identification and resolution of performance problems .

Scaling Strategies

To handle increasing workloads and ensure the scalability of the intelligent workflow orchestration system, consider the following scaling strategies:

1. Vertical Scaling: Vertical scaling involves increasing the resources (e.g., CPU, memory) of individual nodes or servers to handle higher workloads. This approach is suitable when the system requires more processing power or memory to handle complex workflows or AI operations.

2. Horizontal Scaling: Horizontal scaling involves adding more nodes or servers to the system to distribute the workload. This approach is effective when the system needs to handle a large number of concurrent workflows or when the workload can be easily distributed across multiple nodes. Horizontal scaling requires a distributed architecture and load balancing mechanisms to ensure even distribution of traffic.

3. Auto-Scaling: Implement auto-scaling mechanisms to automatically adjust the number of nodes or resources based on the workload demand . Auto-scaling allows the system to dynamically scale up or down depending on the incoming traffic, ensuring optimal resource utilization and cost-efficiency. Cloud platforms like Amazon Web Services (AWS) or Google Cloud Platform (GCP) provide auto-scaling capabilities that can be leveraged for intelligent workflow orchestration systems.

Performance Optimization Techniques

In addition to the scaling strategies, consider the following performance optimization techniques to enhance the efficiency of the intelligent workflow orchestration system:

1. Efficient Data Storage and Retrieval: Optimize the data storage and retrieval mechanisms used by the workflow components. Use efficient database indexing, query optimization techniques, and data caching to minimize the latency and improve the performance of data-intensive operations.

2. Asynchronous I/O: Utilize asynchronous I/O operations to prevent blocking and improve the responsiveness of the system. Asynchronous I/O allows the system to handle multiple requests concurrently without waiting for I/O operations to complete, thereby maximizing resource utilization.

3. Efficient Serialization and Deserialization: Optimize the serialization and deserialization processes used for data exchange between workflow components. Use efficient serialization formats, such as Protocol Buffers or MessagePack , to reduce the overhead of data serialization and improve the performance of inter-component communication.

For Ruby-based applications, consider using Universal ID . Universal ID leverages both MessagePack and Brotli (a combo built for speed and best-in-class data compression) . When combined, these libraries are up to 30% faster and within 2-5% compression rates compared to Protocol Buffers.

4. Compression and Encoding: Apply compression and encoding techniques to reduce the size of data transferred between workflow components. Compression algorithms, such as gzip or Brotli , can significantly reduce the network bandwidth usage and improve the overall performance of the system.

By considering scalability and performance aspects during the design and implementation of intelligent workflow orchestration systems , you can ensure that your system can handle high volumes of concurrent workflows , optimize the performance of AI-powered components, and scale seamlessly to meet growing demands. Continuous monitoring, profiling, and optimization efforts are essential to maintain the system's performance and responsiveness as the workload and complexity increase over time.

Testing and Validation of Workflows

Testing and validation are critical aspects of developing and maintaining intelligent workflow orchestration systems. Given the complex nature of AI-powered workflows, it is essential to ensure that each component functions as expected, the overall workflow behaves correctly, and the

AI decisions are accurate and reliable. In this section, we will explore various techniques and considerations for testing and validating intelligent workflows.

Unit Testing Workflow Components

Unit testing involves testing individual workflow components in isolation to verify their correctness and robustness. When unit testing AI-powered workflow components, consider the following:

1. **Input Validation:** Test the component's ability to handle different types of inputs, including valid and invalid data . Verify that the component gracefully handles edge cases and provides appropriate error messages or exceptions.

2. **Output Verification:** Assert that the component produces the expected output for a given set of inputs. Compare the actual output with the expected results to ensure correctness .

3. **Error Handling:** Test the component's error handling mechanisms by simulating various error scenarios, such as invalid input, resource unavailability, or unexpected exceptions. Verify that the component catches and handles errors appropriately .

4**. Boundary Conditions:** Test the component's behavior under boundary conditions, such as empty input, maximum input size, or extreme values. Ensure that the component handles these conditions gracefully without crashing or producing incorrect results .

Here's an example of a unit test for a workflow component in Ruby using the RSpec testing framework :

```ruby
RSpec.describe OrderValidator do
  describe '#validate' do
    context 'when order is valid' do
      let(:order) { build(:order) }

      it 'returns true' do
        expect(subject.validate(order)).to be true
      end
    end

    context 'when order is invalid' do
      let(:order) { build(:order, total_amount: -100) }

      it 'returns false' do
        expect(subject.validate(order)).to be false
      end
    end
  end
end
```

In this example, the `OrderValidator` component is tested using two test cases: one for a valid order and another for an invalid order. The test cases verify that the `validate` method returns the expected boolean value based on the validity of the order.

Integration Testing Workflow Interactions

Integration testing focuses on verifying the interactions and data flow between different workflow components. It ensures that the components work together seamlessly and produce the expected outcomes. When integration testing intelligent workflows, consider the following:

1. **Component Interaction:** Test the communication and data exchange between workflow components. Verify that the output of one component is correctly passed as input to the next component in the workflow.

2. Data Consistency: Ensure that data remains consistent and accurate as it flows through the workflow. Verify that data transformations, calculations, and aggregations are performed correctly.

3. Exception Propagation: Test how exceptions and errors are propagated and handled across workflow components. Verify that exceptions are caught, logged, and handled appropriately to prevent workflow disruption.

4**. Asynchronous Behavior:** If the workflow involves asynchronous components or parallel execution, test the coordination and synchronization mechanisms. Ensure that the workflow behaves correctly under concurrent and asynchronous scenarios.

Here's an example of an integration test for a workflow in Ruby using the RSpec testing framework :

```ruby
RSpec.describe OrderProcessingWorkflow do

  let(:order) { build(:order) }

  it 'processes the order successfully' do
    expect(OrderValidator).to receive(:validate).and_return(true)
    expect(InventoryManager).to receive(:check_availability).and_return(true)
    expect(PaymentProcessor).to receive(:process_payment).and_return(true)
    expect(ShippingService).to receive(:schedule_shipping).and_return(true)

    workflow = OrderProcessingWorkflow.new(order)
    result = workflow.process

    expect(result).to be true
    expect(order.status).to eq('processed')
  end

end
```

In this example, the `OrderProcessingWorkflow` is tested by verifying the interactions between different workflow components. The test case sets up expectations for each component's behavior and ensures that the workflow processes the order successfully, updating the order status accordingly.

Testing AI Decision Points

Testing AI decision points is crucial to ensure the accuracy and reliability of AI-powered workflows. When testing AI decision points, consider the following:

1. **Decision Accuracy:** Verify that the AI component makes accurate decisions based on the input data and the trained model. Compare the AI decisions with expected outcomes or ground truth data.

2. **Edge Cases:** Test the AI component's behavior under edge cases and unusual scenarios. Verify that the AI component handles these cases gracefully and makes reasonable decisions.

3. **Bias and Fairness:** Assess the AI component for potential biases and ensure that it makes fair and unbiased decisions. Test the component with diverse input data and analyze the outcomes for any discriminatory patterns .

4. **Explainability:** If the AI component provides explanations or reasoning for its decisions, verify the correctness and clarity of the explanations. Ensure that the explanations align with the underlying decision-making process .

Here's an example of testing an AI decision point in Ruby using the RSpec testing framework:

```ruby
RSpec.describe FraudDetector do
  describe '#detect_fraud' do
    context 'when transaction is fraudulent' do
      let(:tx) { build(:transaction, amount: 10000, location: 'High-Risk Country') }

      it 'returns true' do
        expect(subject.detect_fraud(tx)).to be true
      end
    end

    context 'when transaction is legitimate' do
      let(:tx) { build(:transaction, amount: 100, location: 'Low-Risk Country') }

      it 'returns false' do
        expect(subject.detect_fraud(tx)).to be false
      end
    end
  end
end
```

In this example, the `FraudDetector` AI component is tested with two test cases: one for a fraudulent transaction and another for a legitimate transaction. The test cases verify that the `detect_fraud` method returns the expected boolean value based on the characteristics of the transaction.

End-to-End Testing

End-to-end testing involves testing the entire workflow from start to finish, simulating real-world scenarios and user interactions. It ensures that the workflow behaves correctly and produces the desired outcomes. When performing end-to-end testing for intelligent workflows, consider the following:

1. User Scenarios: Identify common user scenarios and test the workflow's behavior under these scenarios. Verify that the workflow handles user inputs correctly, makes appropriate decisions, and produces the expected outputs.

2. Data Validation: Ensure that the workflow validates and sanitizes user inputs to prevent data inconsistencies or security vulnerabilities. Test the workflow with various types of input data, including valid and invalid data.

3. Error Recovery: Test the workflow's ability to recover from errors and exceptions. Simulate error scenarios and verify that the workflow handles them gracefully, logs the errors, and takes appropriate recovery actions.

4. Performance and Scalability: Assess the workflow's performance and scalability under different load conditions. Test the workflow with a large volume of concurrent requests and measure response times, resource utilization, and overall system stability.

Here's an example of an end-to-end test for a workflow in Ruby using the RSpec testing framework and the Capybara library for simulating user interactions:

```ruby
RSpec.describe 'Order Processing Workflow' do
  scenario 'User places an order successfully' do
    visit '/orders/new'
    fill_in 'Product', with: 'Sample Product'
    fill_in 'Quantity', with: '2'
    fill_in 'Shipping Address', with: '123 Main St'
    click_button 'Place Order'

    expect(page).to have_content('Order Placed Successfully')
    expect(Order.count).to eq(1)
    expect(Order.last.status).to eq('processed')
  end
end
```

In this example, the end-to-end test simulates a user placing an order through the web interface. It fills in the required form fields, submits the order, and verifies that the order is processed successfully, displaying the appropriate confirmation message and updating the order status in the database.

Continuous Integration and Deployment

To ensure the reliability and maintainability of intelligent workflows, it is recommended to integrate testing and validation into the continuous integration and deployment (CI/CD) pipeline . This allows for automated testing and validation of workflow changes before they are deployed to production. Consider the following practices:

1. **Automated Test Execution:** Configure the CI/CD pipeline to automatically run the test suite whenever changes are made to the workflow codebase. This ensures that any regressions or failures are detected early in the development process.

2. **Test Coverage Monitoring:** Measure and monitor the test coverage of the workflow components and AI decision points. Aim for high test coverage to ensure that critical paths and scenarios are thoroughly tested.

3. **Continuous Feedback:** Integrate test results and code quality metrics into the development workflow. Provide continuous feedback to developers about the status of tests, code quality, and any issues detected during the CI/CD process.

4. **Staging Environments:** Deploy the workflow to staging environments that closely mirror the production environment. Perform additional testing and validation in the staging environment to catch any issues related to infrastructure, configuration, or data integration.

5. **Rollback Mechanisms:** Implement rollback mechanisms in case of deployment failures or critical issues detected in production. Ensure

that the workflow can be quickly reverted to a previous stable version to minimize downtime and impact on users.

By incorporating testing and validation throughout the development lifecycle of intelligent workflows, organizations can ensure the reliability, accuracy, and maintainability of their AI-powered systems. Regular testing and validation help catch bugs, prevent regressions, and build confidence in the workflow's behavior and outcomes.

Part 2: The Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Prompt Engineering

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Chain of Thought

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Examples

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Content Generation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Structured Entity Creation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

LLM Agent Guidance

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Benefits and Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Mode Switch

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

When to Use It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Role Assignment

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

When to Use It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Examples

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Prompt Object

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Prompt Template

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Benefits and Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

When to Use It:

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Structured IO

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Scaling Structured IO

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Benefits and Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Prompt Chaining

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

When To Use It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example: Olympia's Onboarding

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Prompt Rewriter

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Response Fencing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Benefits and Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Query Analyzer

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Implementation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Part-of-Speech (POS) Tagging and Named Entity Recognition (NER)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Intent Classification

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Keyword Extraction

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Benefits

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Query Rewriter

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Benefits

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Ventriloquist

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

When to Use It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Discrete Components

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Predicate

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

When to Use It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

API Facade

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Key Benefits

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

When To Use It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Authentication and Authorization

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Request Handling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Response Formatting

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Error Handling and Edge Cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Scalability and Performance Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Comparison with Other Design Patterns

Result Interpreter

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

When to Use It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Virtual Machine

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

When to Use It

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Behind The Magic

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Specification and Testing

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Specifying the Behavior

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Writing Test Cases

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example: Testing the Translator Component

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Replay of HTTP Interactions

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Human In The Loop (HITL)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

High-Level Patterns

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Hybrid Intelligence

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Adaptive Response

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Human-AI Role Switching

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Escalation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Key Benefits

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Real-World Application: Healthcare

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Feedback Loop

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Applications and Examples

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Advanced Techniques in Human Feedback Integration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Passive Information Radiation

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Contextual Information Display

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Proactive Notifications

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Explanatory Insights

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Interactive Exploration

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Key Benefits

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Applications and Examples

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Collaborative Decision Making (CDM)

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Continuous Learning

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Applications and Examples

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Ethical Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Role of HITL in Mitigating AI Risks

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Technological Advancements and Future Outlook

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Challenges and Limitations of HITL Systems

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Intelligent Error Handling

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Traditional Error Handling Approaches

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Contextual Error Diagnosis

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Prompt Engineering for Contextual Error Diagnosis

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Retrieval-Augmented Generation for Contextual Error Diagnosis

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Intelligent Error Reporting

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Predictive Error Prevention

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Smart Error Recovery

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Personalized Error Communication

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Adaptive Error Handling Workflow

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Quality Control

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Eval

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Problem

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Solution

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Understanding Golden References

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How Reference-Free Evals Work

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Guardrail

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Problem

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Solution

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

How It Works

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Example

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Considerations

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Guardrails and Evals: Two Sides of the Same Coin

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

The Interchangeability of Guardrails and Reference-Free Evals

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Implementing Dual-Purpose Guardrails and Evals

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Glossary

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

A

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

B

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

C

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

D

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

E

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

F

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

G

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

H

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

I

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

J

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

K

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

L

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

M

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

N

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

O

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

P

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Q

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

R

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

S

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

T

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

U

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

V

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

W

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Z

This content is not available in the sample book. The book can be purchased on Leanpub at http://leanpub.com/patterns-of-application-development-using-ai.

Index