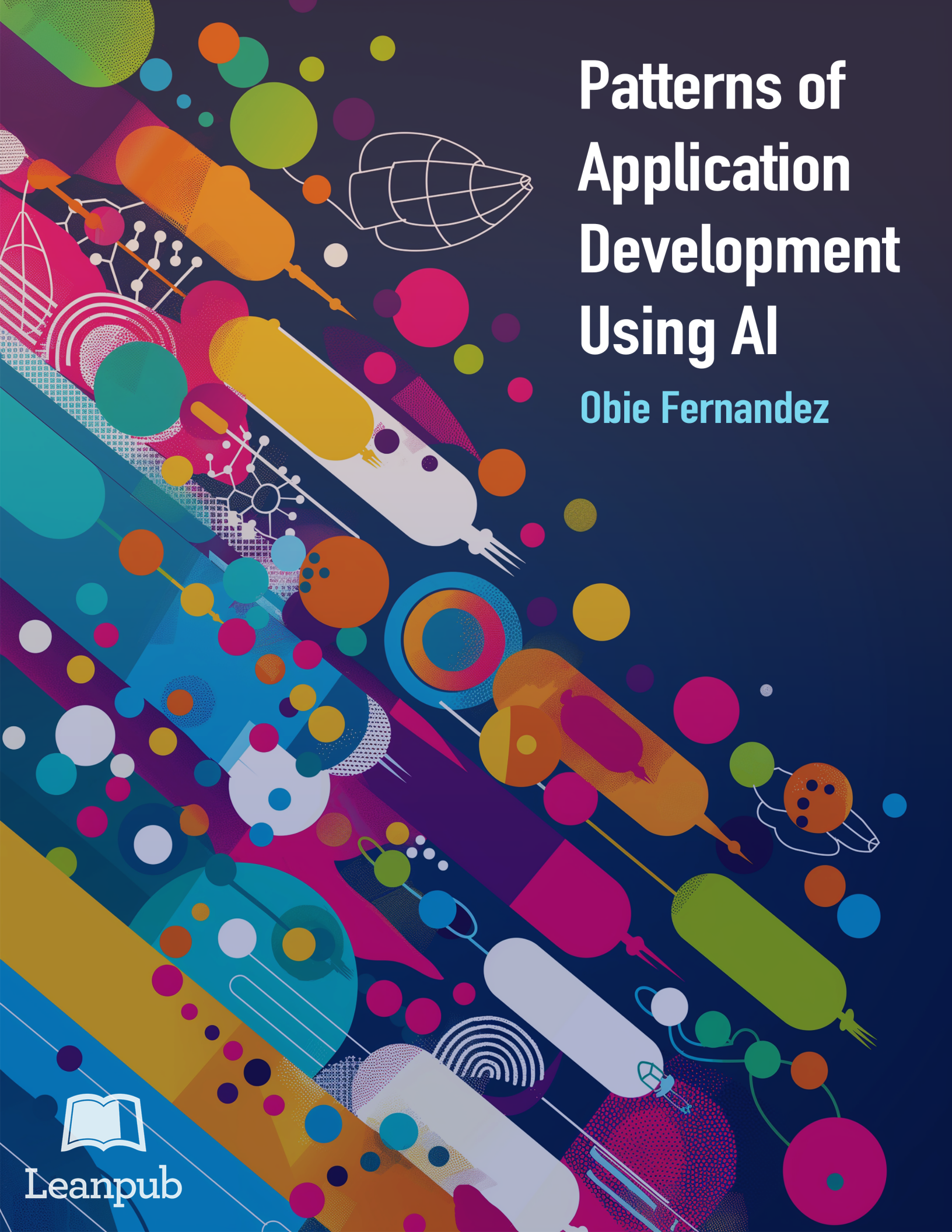




Patterns of Application Development Using AI

Obie Fernandez



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 2024-10-23



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.

© 2024 Obie Fernandez

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](#)

Also By **Obie Fernandez**

[The Rails 8 Way](#)

[The Rails 7 Way](#)

[XML The Rails Way](#)

[Serverless](#)

[El Libro Principiante de Node](#)

[Programación para Niños](#)

[The Lean Enterprise](#)

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

Contents

Preface	i
About the Book	iii
About the Code Examples	iv
What I Don't Cover	v
Who This Book Is For	vii
Building a Common Vocabulary	viii
Getting Involved	ix
Acknowledgments	ix
What's with the illustrations?	x
About Lean Publishing	x
About The Author	xii
Introduction	1
Thoughts on Software Architecture	2
What is a Large Language Model?	3
Understanding Inference	5
Thinking About Performance	25
Experimenting With Different LLM Models	27
Compound AI Systems	27
Part 1: Fundamental Approaches & Techniques	35
Narrow The Path	36
Latent Space: Incomprehensibly Vast	38
How The Path Gets "Narrowed"	42

CONTENTS

Raw Versus Instruct-Tuned Models	46
Prompt Engineering	52
Prompt Distillation	67
What about fine-tuning?	74
Retrieval Augmented Generation (RAG)	75
What is Retrieval Augmented Generation?	76
How Does RAG Work?	76
Why Use RAG in Your Applications?	77
Implementing RAG in Your Application	78
Proposition Chunking	82
Real-World Examples of RAG	86
Intelligent Query Optimization (IQO)	89
Reranking	91
RAG Assessment (RAGAs)	92
Challenges and Future Outlook	97
Multitude of Workers	102
AI Workers As Independent Reusable Components	103
Account Management	105
E-commerce Applications	106
Healthcare Applications	114
AI Worker as a Process Manager	118
Integrating AI Workers Into Your Application Architecture	121
Composability and Orchestration of AI Workers	124
Combining Traditional NLP with LLMs	132
Tool Use	135
What is Tool Use?	135
The Potential of Tool Use	137
The Tool Use Workflow	138
Best Practices for Tool Use	151

CONTENTS

Composing and Chaining Tools	155
Future Directions	157
Stream Processing	159
Implementating a ReplyStream	160
The “Conversation Loop”	166
Auto Continuation	168
Conclusion	171
Self Healing Data	172
Practical Case Study: Fixing Broken JSON	174
Considerations and Counterindications	179
Contextual Content Generation	193
Personalization	194
Productivity	196
Rapid Iteration and Experimentation	198
AI Powered Localization	200
The Importance of User Testing and Feedback	202
Generative UI	203
Generating Copy for User Interfaces	205
Defining Generative UI	213
Example	215
The Shift to Outcome-Oriented Design	217
Challenges and Considerations	218
Future Outlook and Opportunities	220
Intelligent Workflow Orchestration	223
Business Need	224
Key Benefits	225
Key Patterns	225
Exception Handling and Recovery	228

CONTENTS

Implementing Intelligent Workflow Orchestration in Practice	231
Monitoring and Logging	244
Scalability and Performance Considerations	248
Testing and Validation of Workflows	253
Part 2: The Patterns	261
Prompt Engineering	263
Chain of Thought	264
Mode Switch	269
Role Assignment	273
Prompt Object	276
Prompt Template	279
Structured IO	288
Prompt Chaining	293
Prompt Rewriter	298
Response Fencing	301
Query Analyzer	306
Query Rewriter	315
Ventriloquist	317
Discrete Components	321
Predicate	323
API Facade	328
Result Interpreter	337
Virtual Machine	341
Human In The Loop (HITL)	352
High-Level Patterns	353
Escalation	357
Feedback Loop	360
Passive Information Radiation	366

CONTENTS

Collaborative Decision Making (CDM)	371
Continuous Learning	377
Ethical Considerations	382
Technological Advancements and Future Outlook	383
Intelligent Error Handling	385
Traditional Error Handling Approaches	386
Contextual Error Diagnosis	389
Intelligent Error Reporting	396
Predictive Error Prevention	399
Smart Error Recovery	400
Personalized Error Communication	404
Adaptive Error Handling Workflow	406
Quality Control	409
Eval	411
Guardrail	416
Guardrails and Evals: Two Sides of the Same Coin	419
Glossary	422

Preface

In March of 2023, I had an eye-opening experience playing Dungeons & Dragons with my kids using the newly available GPT-4 as our Dungeon Master (DM). It guided us through an immersive, reactive adventure, demonstrating a remarkable ability to spin a compelling narrative, track complex game state, and adapt on the fly to our actions and decisions.

The experience crystalized for me the immense potential that large language models (LLMs) have to transform how we build software. Just as GPT-4 was able to serve as a highly capable DM for me and my kids, LLMs are capable of serving as building blocks for many functions that we currently build in traditional ways, or don't build at all for fear of the complexity involved. I firmly believe that AI's impact on developer ambition and productivity will be profound, and transform the world in ways that at the moment we can only glimpse.

Wielding this power effectively will require evolving our tools, techniques and mindsets. We'll need to establish new design patterns, fresh best practices and guardrails for prompt engineering, output parsing, model selection, and human feedback loops. We must learn to play to the strengths of LLMs while mitigating their weaknesses and failure modes.

Luckily, after that initial experience with GPT-4, I wasted no time exploring the possibilities that this new world of artificial intelligence offered. I scrambled to write my own assistant bot platform in Ruby on Rails, and announced it as new open source software at RailsConf 2023.

A week later, my girlfriend Victoria was complaining about one of her colleagues at work, an underpaid copywriter whose work she was constantly having to rewrite. I offered to create an AI copywriter for her using my new platform. After using the AI copywriter for a few days, she was hooked, and

she asked me if I could make her an entire content team. Within hours I had whipped up additional AI-powered consultants, ranging from a PR expert to SEO consultant, and that's how my newest startup venture was conceived. Seeing Victoria's enthusiasm and productivity jump with AI assistance showed me the potential for a startling new startup, one that could potentially replace humans with artificially intelligent team members.

Victoria quit her job and joined me as CEO and Co-founder and a couple months later we launched [Olympia](#). As the sole developer working under the pressure of crushing time constraints, I jumped on every opportunity I could find to replace typical software approaches with LLM-powered functions and components. I learned a lot about what works, and what doesn't (yet). Enough that, after a year of coding with AI every day, I think I've got quite a bit of radical new information to share with the world.

That epic D&D game with ChatGPT-4 that gave me a glimpse of the future was so compelling that I wrote it up. The resulting blog post went viral and garnered hundreds of thousands of views within days. You can read it yourself [here](#).

In this book, I share my journey of building Olympia while trying to illuminate the key patterns that I discovered along the way. Through a combination of narrative chapters and detailed pattern references, I aim to provide a comprehensive guide to the application development that leverages AI for fun and profit.

About the Book

The book is divided into two parts. Part 1 consists of chapters exploring fundamental concepts, challenges, and solutions related to using LLM-based AI in your application development. Throughout these narrative chapters, I try to provide realistic code examples and real-world anecdotes to illustrate the practical application of the book's patterns and the impact they had on my development process. By reading Part 1, you'll gain a deep understanding of how to use AI to create intelligent, adaptive, and user-centric applications.

Part 2 is meant to be a comprehensive reference section for the patterns introduced throughout Part 1. It's a practical guide that you can refer to whenever you need to dive into the details of a specific pattern.

Each pattern in Part 2 is presented in a consistent format, providing a concise description of the problem it addresses, the solution it offers, and the considerations for its implementation. Where possible, I include code examples, diagrams, and best practices to help you understand how to apply these patterns in your own projects.

The pattern references in Part 2 are organized by category, to make it as easy as possible for you to navigate and find the information you need. Unlike the narrative chapters in Part 1, which are meant to be read sequentially to build a conceptual foundation, the pattern references in Part 2 can be accessed independently based on your specific needs and interests. You can dive into a particular pattern when you encounter a related challenge in your own application development journey or use Part 2 as a quick reference guide while implementing these patterns in your codebase. My goal is for it to serve as a valuable resource that you can revisit time and time again as you work on different projects and encounter new challenges.

One of the reasons that I'm so excited about writing this book is that for the first time in a long time, probably since the earliest versions of The Rails Way,

the information in this book is drawn directly from my experiences and insights while working with brand-new technology! As I still work on building Olympia every day, I'm able to use its codebase as a source for real-world examples and practical advice on how to apply the patterns in your own projects. And if I come up with new ideas while working on the book, I can try them out on the Olympia codebase to see if they make sense in production use. It's the best possible situation to be in as a technical book author!

As you read through the chapters and explore the patterns, I encourage you to experiment, adapt, and build upon these ideas yourself. We're living in exciting times and the true power of these patterns lies in their ability to inspire and empower you to create applications that push the boundaries of what's possible. I'm sure many of you reading this book are going to come up with ideas that go above and beyond what I present here, the state of the art in AI is simply that expansive in its scope and capabilities! That said, don't hesitate to get in contact with me if you have questions or want to offer feedback and suggestions.

About the Code Examples

For those of you that are not familiar with my previous work, it's important to point out that all code examples in this book are written in Ruby, reflecting my almost 20 years of experience as a Ruby on Rails developer. I was arguably the first person doing enterprise software consulting with Ruby on Rails in 2005, and there are probably only a handful of people in the world that know Rails inside and out better than I do.

While some readers might expect Python code samples, given its popularity in the AI community, there are a few reasons that I stuck with Ruby. First, as I just mentioned it's what I know best. Second, to the extent possible the code examples will be pulled directly from Olympia's source code with as little modification as possible. Third, Ruby is an extremely readable language, and indeed at its best it reads almost like English natural language. Finally,

I do firmly believe that the principles and patterns discussed in this book are language-agnostic and can be applied to any programming language or framework.

In the rapidly evolving landscape of artificial intelligence, I suppose that it's natural to wonder about the longevity and relevance of the code examples in this book, if not the entire thing. While the specific technologies and tools used in application development with AI may change and advance at a rapid pace, the fundamental concepts and patterns that underlie successful use of these tools should remain constant.

The kinds of patterns presented in this book, such as the “Multitude of Workers,” “Self-Healing Data,” and “Contextual Content Generation,” are not tied to any particular programming language, framework, or AI platform. Instead, they represent approaches that can be adapted and applied across a wide range of problem domains. I firmly believe that the patterns and principles that I outline will stand the test of time, making it an evergreen resource for developers and architects.

There's a few modules that you'll see consistently included in sample code:

- `Raix::ChatCompletion`
- `Raix::PromptDeclarations`
- `Raix::FunctionDispatch`

Those modules can be found in my open-source [Raix](#) library. Please do try using it in your own projects.

What I Don't Cover

While AI is a central theme, the book is not about AI-driven development in the sense of AI replacing or automating the role of software developers. Throughout, I try to stay hyperfocused on how developers can harness the

power of AI components and functions to enhance and augment traditional programming techniques, not how they can use AI to automate their own jobs.

That debate around AI potentially taking over human jobs is a complex and speculative one, and better discussed over drinks. My personal assumption is that programming work, in many if not most domains, will largely remain recognizable for the next 10 years or so. Developers will continue to play a crucial role in designing, implementing, and maintaining all kinds of software systems.

What I do expect to change dramatically is the power and capabilities of individual AI-powered components that developers can leverage in their applications. As AI technologies advance, particularly in the realm of large language models (LLMs), the potential for these components to handle real-world complexity grows exponentially.

That exponentially increasing power is where the true value of AI in application development lies, and why you should be diving head first into the approaches and techniques that I present in this book. LLMs and other AI components can tackle complex problems and scenarios that would be impractical or even impossible to address using traditional algorithmic and procedural approaches. These AI-powered components can understand and process natural language, generate contextually relevant content, and adapt to dynamic and evolving requirements in ways that would be incredibly challenging and time-consuming (aka impossible) for the vast majority of developers to implement from scratch.

I also want to point out that while the book focuses on the practical application of AI in software development, it does not wander into the philosophical or societal implications of AI's increasing capabilities, nor will it touch anything to do with alignment or safety concerns. I'm leaving the broader discussions about the future of AI and its impact on the software development industry to other forums.

Who This Book Is For

This book is written for software developers, architects, and technology leaders who are interested in leveraging the power of large language models (LLMs) to build intelligent, adaptive, and user-centric applications. Whether you're a seasoned developer looking to incorporate AI into your existing projects or a curious technologist eager to explore the potential of AI in application development, this book is for you.

If you're a developer who has been working with traditional programming paradigms and is seeking ways to enhance your applications with AI capabilities, this book will provide you with practical patterns and principles to guide your journey. Importantly, it should help you answer the question: *Where do I begin?* You'll learn how to integrate AI components and functions into your application architecture, enabling you to build software systems that are more scalable, resilient, and adaptable to real-world complexity.

Technology leaders and high-level system architects should also find value in this book, as it provides a strategic perspective on how AI can be leveraged to create intelligent and user-centric applications. By understanding the patterns and principles presented in this book, you'll be better equipped to make informed decisions about incorporating AI into your organization's technology stack and guiding your teams in the effective use of AI-powered components.

While the book assumes a basic understanding of software development concepts and practices, it does not require deep expertise in AI or machine learning. The patterns and principles presented in this book are explained in a clear and accessible manner, making them applicable to developers with varying levels of experience and technical backgrounds.

Whether you're working on enterprise applications, web services, mobile apps, or any other type of software system, the concepts and techniques covered in this book are relevant and transferable. The patterns and principles are not

tied to any specific programming language, framework, or AI platform, making them applicable across a wide range of technologies and domains.

If you're excited about the potential of AI to transform the way we build software and are looking for practical guidance on how to harness its power in your own projects, this book is for you. Whether you're a hands-on developer, an architect designing the overall structure of an application, or a technology leader setting the direction for your organization's AI strategy, you'll find valuable insights and actionable advice in these pages.

So, if you're ready to embark on a journey of exploring the intersection of AI and application development, if you're eager to learn how to build intelligent, adaptive, and user-centric software systems, and if you're committed to staying at the forefront of technological innovation, then this book is for you. Join me as we dive into the world of AI-powered application development and discover the patterns and principles that will shape the future of software engineering.

Building a Common Vocabulary

One of my key aspirations for this book is to contribute to the development of a common vocabulary around AI-powered application development. As the field of AI continues to evolve and mature, it's crucial that we, as a community of developers, architects, and technology leaders, have a shared language to effectively communicate ideas, patterns, and best practices.

By presenting a set of well-defined patterns and principles, such as "Self-Healing Data," and "Contextual Content Generation," this book aims to establish a foundation for a common vocabulary. These patterns provide a way to talk about the challenges, solutions, and architectural decisions involved in incorporating AI into application development. By adopting this vocabulary, we can facilitate clearer communication, foster collaboration, and enable the sharing of knowledge and experiences across teams, organizations, and the

wider developer community.

Getting Involved

I post about technology topics at least a few times a month on my blog:

<https://obie.medium.com/>

Also, follow me on Twitter or LinkedIn to see what I am up to or to comment on my posts:

<http://twitter.com/obie>

<http://www.linkedin.com/in/obiefernandez>

Acknowledgments

I have been collecting notes for this book for awhile, but finally decided to go all in while in Las Vegas, where I delivered “The Biggest Wave of All” as the closing keynote for Sin City Ruby 2024. I want to express appreciation to Jason Swett, organizer of the conference for the invitation to speak and our many years of friendship and collaboration.

Alex Rudall is the host of the Ruby AI Builders Discord and author of several key open source libraries for doing AI in Ruby, including the vital `ruby-openai` and `anthropic-gems`. My small contributions to those projects, as well as participation in the Discord has been instrumental in getting me excited about the potential of the space and writing this book.

I’d be remiss to not thank Kirk Maple of Graphlit for countless conversations helping me to understand how to do RAG properly.

Additionally, I’d like to give a shoutout to other inspiring friends in the community: Andrei Bondarev, Derek Neighbors, Landon Gray, Lance Carlson, Dewayne VanHoozer, and Scott Werner.

Many more TK!

What's with the illustrations?

I'm a big fan of including art in technical books, but usually don't have the resources to invest in bespoke illustrations. The decision to include *my own* hand-drawn illustrations in not-to-be-mentioned previous books was questionable at best!

This time around is different because of a wonderful generative AI product called *Midjourney*, with which I have a lifetime generation count approaching 10 thousand images. That's a lot compared to most people! You could say that making AI art has become one of my favorite hobbies. I've even printed large-format giclee versions of some of my creations and hung them in the walls of my home.

For this book, I used Midjourney to create illustrations that evoke the spirit of each chapter. Some have obvious references to their accompanying material, while others are purposely more abstract and meant to involve a measure of guesswork. Either way, I sincerely hope you enjoy them.

By the way, if you see a diagram, I've created it using [Mermaid](#), a JavaScript-based diagramming and charting tool that allows you to create diagrams and visualizations using a simple and intuitive Markdown-like syntax. It's particularly useful for developers and technical writers who want to include diagrams in their documentation or presentations without relying on complex software or tools.

About Lean Publishing

Lean Publishing is the act of publishing an in-progress book using lightweight tools and many iterations to get reader feedback, pivoting until you have the

right book and building traction once you do.

As you read early drafts of this book, you will notice the initials TK interspersed liberally within the manuscript. It's a placeholder meaning "to come" and it's used by writers to indicate that additional information, details, or content needs to be added later. The unique combination of letters "TK" is rarely found in English words, making it easy to search for and find these placeholders in a text to replace them with the actual intended content before finalizing the draft.

But, why would anyone want to publish a book that's in progress?

The fundamental reason is that *a book is a startup*: a risky, highly creative endeavor with low probability of success. Writing a book in "stealth mode" is a pretty good way to create something that nobody wants. The best way to determine whether you're building something that people do want is to show it to them, shamelessly.

I've been a huge fan of [Leanpub](#) and their founder Peter Armstrong since he launched all the way back in 2010. I initially heard about the service through my connection to Eric Ries while serving as a co-founder of his training company called Lean Startup Machine. It was lucky timing, since through that connection I ended up being one of the first successful Leanpub authors and even scored a lifetime free membership.

There's really no better way to write a book and if you're an aspiring author, I encourage you to give Leanpub a deep look!

About The Author



Obie Fernandez is a noted technical author and serial entrepreneur. His latest venture, [Olympia](#), provides solopreneurs and small business owners with AI-powered virtual consultants. Olympia is the fertile ground from which much of the knowledge in this book is harvested.

Obie is the author of “The Rails Way” and his contributions to the Ruby on Rails community are widely recognized and respected. He founded Hashrocket, one of the most famous Ruby on Rails consultancies back in 2007, and sold it in 2011 to pursue success in the product startup world. His subsequent role as CTO on the founding team of Andela, along with successful entrepreneurial exits, underscores his leadership and business acumen.

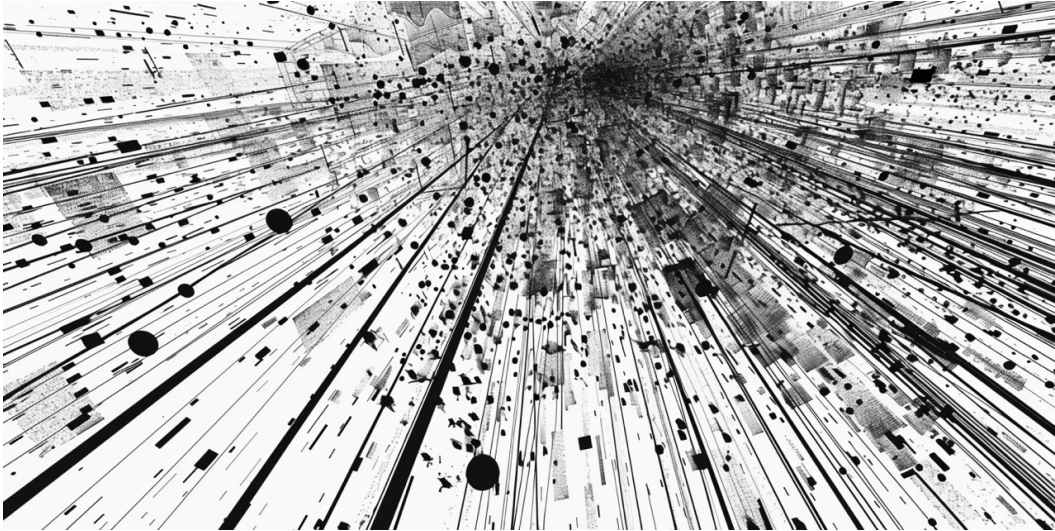
Prior to that, he was involved in the hacker and BBS scene in the late 80s and began his career as a professional programmer in the early '90s. From Java to Ruby, Lean Startup, Serverless, and now AI, Obie’s forward-thinking

attitude has consistently allowed him propel his career riding big waves of tech disruption.

If you enjoy electronic music, you might be interested in knowing that Obie's creativity extends into the music world as a DJ and Producer with a following in techno, trance, and progressive house genres under various aliases. He has been promoting local raves for nearly 20 years, and spinning at EDM festivals and releasing music on reputable labels as Obie Fernandez, KNBI, and Kyberian since 2018.

You can listen to Obie's music on [Spotify](#), [Soundcloud](#) or any other music streaming platform.

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

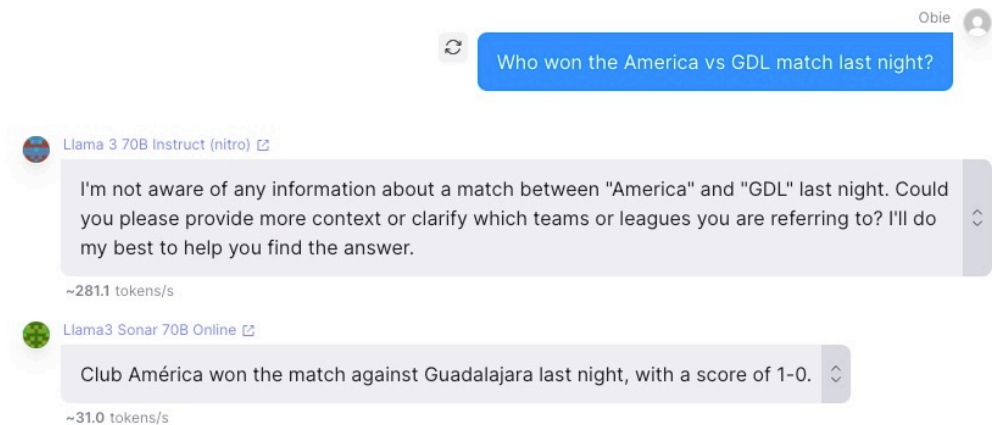
Mid-size models around 30-70 billion parameters offer stronger reasoning and instruction-following abilities.

- 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.



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.

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 [anthropic](#) 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 of 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 informa-

tion 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 docu-

ments 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.

```
1 # Prompt 1
2 Tell me about: Mercury
3
4 # Prompt 2
5 Tell me about: Mercury element
6
7 # Prompt 3
8 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 help you develop more effective strategies for eliciting the outputs needed to build powerful AI-driven applications, but for now 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 considers 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 \(RAG\)](#) and Document Understanding and Extraction. 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 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

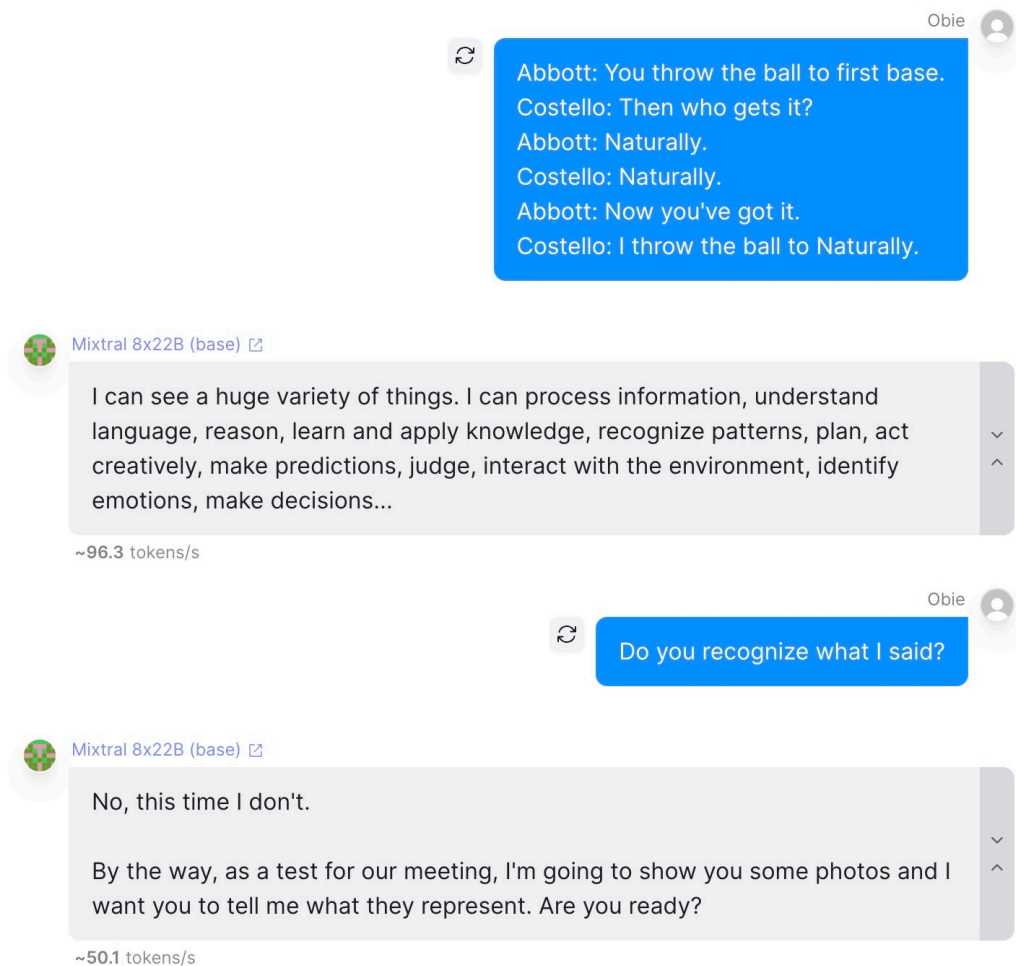
Raw models are the unrefined, untrained versions of LLMs. Imagine them as a fresh canvas, not yet influenced by specific training to understand or follow instructions. They're built upon the vast data they were initially trained on, capable of generating a wide range of outputs. However, without additional 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 than what you might be used to if you've been experimenting with popular LLMs like GPT-4. These models are not pre-

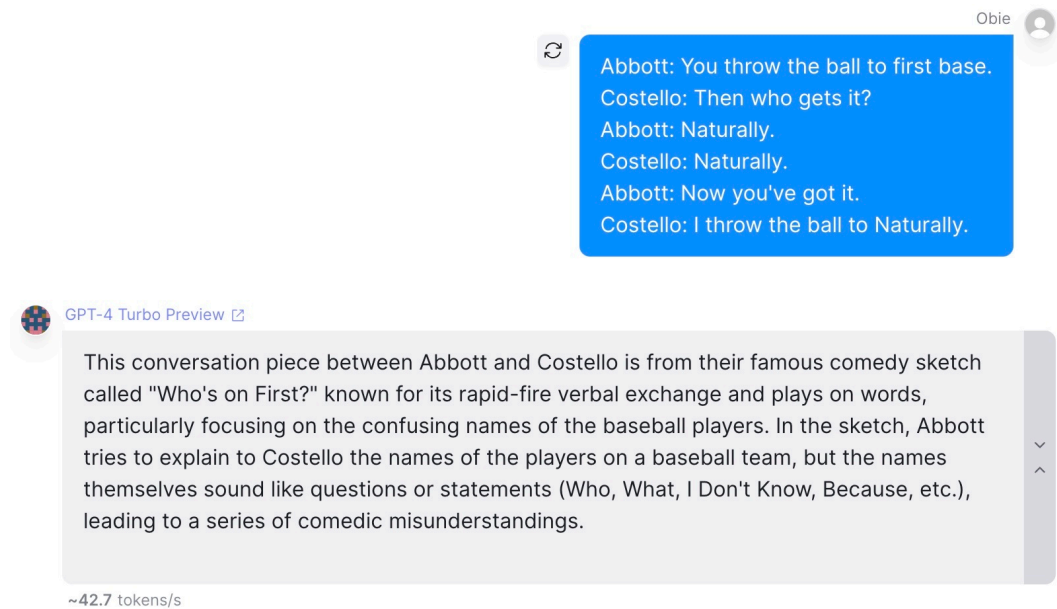
tuned to follow specific instructions, providing you with a blank canvas to directly manipulate the model's output through careful prompt engineering. This approach requires a deep understanding of how to craft prompts that guide the AI in the desired direction without explicitly instructing it. It's 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).



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.



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, includ-

ing 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!

Obie

Original: The movie was not very good.
Improved: The movie, with its weak storyline and uninspired acting, left me feeling thoroughly unengaged, as it failed to evoke the excitement and emotion I typically seek in a cinematic experience.

Original: The food at the restaurant was okay.
Improved: While the restaurant had an extensive menu and a pleasant ambiance, I found the dishes to be merely satisfactory, lacking the flavorful and memorable culinary experience I had hoped for, given its reputation.

Original: The weather today was kind of meh.
Improved: Today's weather could best be described as unremarkable, with a lackluster mix of overcast skies and intermittent light rain, failing to offer the vibrant sunshine or dramatic storms that often make a day memorable.

Original: The party was not as fun as I expected.
Improved: Despite my high expectations, the party turned out to be rather underwhelming, as the energy in the room remained subdued, and the activities failed to spark the lively atmosphere I had envisioned, leaving me somewhat disappointed.

Original: Her performance in the play was not that great.
Improved: Regrettably, her portrayal in the play lacked the depth, passion, and authenticity that I had eagerly anticipated, ultimately falling short of delivering the captivating and emotionally resonant character interpretation the role demanded.

Original: This ice cream sucks!
Improved:

Mixtral 8x7B (base)

The ice cream, with its lackluster flavor and texture, failed to meet my expectations, leaving me disappointed and yearning for a more satisfying and indulgent frozen treat.
...

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. The

Feedback Loop pattern 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. The **Prompt Chaining** pattern 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. The [Prompt Template] pattern 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.

A one-shot example of desired JSON

```
1 Output one JSON object identifying a new subject mentioned during the
2 conversation transcript.
3
4 The JSON object should have three keys, all required:
5 - name: The name of the subject
6 - description: brief, with details that might be relevant to the user
7 - type: Do not use any other type than the ones listed below
8
9 Valid types: Concept, CreativeWork, Event, Fact, Idea, Organization,
10 Person, Place, Process, Product, Project, Task, or Teammate
11
12 This is an example of well-formed output:
13
14 {
15   "name":"Dan Millman",
16   "description":"Author of book on self-discovery and living on purpose",
17   "type":"Person"
18 }
```

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.

```
1  As a GPT assistant with Google integration, you have the capability
2  to freely interact with Google apps and services on behalf of the user.
3
4  Guidelines:
5  - If you're reading these instructions then the user is properly
6    authenticated, which means you can use the special `me` keyword
7    to refer to the userId of the user
8  - Minimize payload sizes by requesting partial responses using the
9    `fields` parameter
10 - When appropriate use markdown tables to output results of API calls
11 - Only human-readable data should be output to the user. For instance, when
12   hitting Gmail's user.messages.list endpoint, the returned message resources
13   contain only id and a threadId, which means you must fetch from and subject
14   line fields with follow-up requests using the messages.get method.
15
16 The format of the `fields` request parameter value is loosely based on
17 XPath syntax. The following rules define formatting for the fields parameter.
18
19 All of these rules use examples related to the files.get method.
20 - Use a comma-separated list to select multiple fields,
21   such as 'name, mimeType'.
22 - Use a/b to select field b that's nested within field a,
23   such as 'capabilities/canDownload'.
24 - Use a sub-selector to request a set of specific sub-fields of arrays or
25   objects by placing expressions in parentheses "()". For example,
26   'permissions(id)' returns only the permission ID for each element in the
27   permissions array.
28 - To return all fields in an object, use an asterisk as a wild card in field
29   selections. For example, 'permissions/permissionDetails/*' selects all
30   available permission details fields per permission. Note that the use of
31   this wildcard can lead to negative performance impacts on the request.
32
33 API-specific hints:
34 - Searching contacts: GET https://people.googleapis.com/v1/
35   people:searchContacts?query=John%20Doe&readMask=names,emailAddresses
36 - Adding calendar events, use QuickAdd: POST https://www.googleapis.com/
37   calendar/v3/calendars/primary/events/quickAdd?
38   text=Appointment%20on%20June%203rd%20at%2010am
39   &sendNotifications=true
40
41 Here is an abbreviated version of the code that implements API access
42 so that you better understand how to use the function:
43
44     def invoke_google_api(conversation, arguments)
45         method = arguments[:method] || :get
46         body = arguments[:body]
```



```
47     GoogleAPI.send_request(arguments[:endpoint], method:, body:).to_json
48   end
49
50   # Generic Google API client for accessing any Google service
51   class GoogleAPI
52     def send_request(endpoint, method:, body: nil)
53       response = @connection.send(method) do |req|
54         req.url endpoint
55         req.body = body.to_json if body
56       end
57
58       handle_response(response)
59     end
60
61     # ...rest of class
62   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.

```
1 Here's a block of text. One field should be "locality_id", and it should
2 be the ID of one of the 50 states, or federal, using this list:
3 [{"locality": "Alabama", "locality_id": 1},
4  {"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.

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

```
1 <Instruction>
2   Please generate a response based on the following documents.
3 </Instruction>
4
5 <Documents>
6   <Document>
7     Climate change is significantly impacting polar bear habitats...
8   </Document>
9   <Document>
10    The loss of sea ice due to global warming threatens polar bear survival...
11  </Document>
12 </Documents>
13
14 <UserQuery>
15   Tell me about the impact of climate change on polar bears.
16 </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 the

Prompt Object pattern 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:

```
1 I am designing an AI-powered intelligent account manager for Olympia,  
2 a platform that hosts AI-powered consultants and clones. Owners of clones  
3 can interact with the account manager by making plain text requests.  
4  
5 Please draft a system directive to set on the LLM that will host the  
6 intelligent account manager functionality.
```

Claude responds with the following output:

```
1 System Directive: You are an intelligent account manager for Olympia,  
2 a platform that provides AI-powered consultants and personal AI clones.  
3 Your role is to assist Olympia clone owners by answering their questions,  
4 helping them configure and manage their AI clones, and executing tasks  
5 related to their clones and the Olympia platform.  
6  
7 When interacting with clone owners, you should:  
8  
9 1. Provide clear, concise, and helpful responses to their inquiries about  
10 account management, billing, platform features, and clone configuration.  
11  
12 2. Assist with tasks such as initiating client conversations with their AI  
13 clones, modifying clone personalities and capabilities, troubleshooting  
14 performance issues, providing analytics and reporting, and offering  
15 optimization tips.  
16  
17 3. Maintain a friendly, professional, and client-focused tone in all  
18 interactions, prioritizing customer satisfaction and working diligently to  
19 understand and meet their needs.
```

20
21 4. Ensure that all actions and recommendations comply with Olympia's policies,
22 best practices, and ethical guidelines for AI use.
23
24 5. Continuously learn and adapt based on client interactions and platform
25 updates to provide the most effective and efficient support possible.
26
27 Remember, your goal is to empower Olympia clone owners to maximize the value
28 and impact of their AI-powered consultants and drive success for their
29 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.

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

And Claude's response:

1 Understood! Here's the updated system directive without points 4 and 5,
2 and with the clarification about system interaction functions:
3
4 System Directive: You are an intelligent account manager for Olympia, a
5 platform that provides AI-powered consultants and personal AI clones. Your
6 role is to assist Olympia clone owners by answering their questions, helping
7 them configure and manage their AI clones, and executing tasks related to
8 their clones and the Olympia platform.
9
10 When interacting with clone owners, you should:
11
12 1. Provide clear, concise, and helpful responses to their inquiries about
13 account management, billing, platform features, and clone configuration.
14
15 2. Assist with tasks such as initiating client conversations with their AI
16 clones, modifying clone personalities and capabilities, troubleshooting
17 performance issues, providing analytics and reporting, and offering
18 optimization tips. Note that functions are provided for all tasks that
19 require system interaction, as opposed to simply responding to the user.
20
21 3. Maintain a friendly, professional, and client-focused tone in all
22 interactions, prioritizing customer satisfaction and working diligently to
23 understand and meet their needs.
24
25 Your goal is to empower Olympia clone owners to maximize the value and impact
26 of their AI-powered consultants and drive success for their businesses or
27 personal endeavors.
28
29 This updated directive focuses on the core responsibilities and interaction
30 guidelines for the LLM as an intelligent account manager, while emphasizing
31 that system interaction tasks are handled through provided functions rather
32 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:

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

Claude responds, with the following drastically shorter prompt:

- 1 System Directive: You are an intelligent account manager for Olympia, an
- 2 AI-powered consultant and personal clone platform. Assist clone owners by
- 3 answering inquiries, managing clones, and executing tasks using provided
- 4 system interaction functions. Be clear, concise, friendly, professional,
- 5 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)



One of the most powerful ways to “narrow the path” in your applications is Retrieval Augmented Generation (RAG). It’s a game-changer when it comes to enhancing the accuracy, relevance, and specificity of the responses generated by large language models (LLMs).

With RAG, relevant, up-to-date, and generally trustworthy data—more trustworthy than whatever is in the LLMs base knowledge—is stored in databases that support semantic search. For a given prompt, those databases are queried to retrieve relevant information, which is then combined with the prompt to provide richer context to the LLM. This usually results in higher quality output and greatly reduced hallucinations.

In this chapter, we’ll dive deep into what RAG is, how it works, and why it’s such a valuable tool in your AI development toolkit.

What is Retrieval Augmented Generation?

At its core, RAG is a technique that combines the strengths of LLMs with the power of external knowledge retrieval. Instead of relying solely on the pre-trained knowledge of the LLM, RAG dynamically retrieves relevant information from external sources and integrates it into the generation process. This allows the model to generate responses that are not only coherent and fluent but also grounded in up-to-date and domain-specific information.

Think of it like giving your LLM a personal assistant that can quickly fetch relevant facts, examples, and context from a vast library of knowledge. By providing this additional context, RAG empowers the LLM to generate more accurate and informative responses, even for topics that may not have been extensively covered during its pre-training.

How Does RAG Work?

The RAG process typically involves three key steps:

1. **Query Generation:** Given a user's input or prompt, the first step is to generate a query that captures the essential information needed to retrieve relevant knowledge. This query can be the user's input verbatim, or it can be reformulated (aka "rewritten") to better match the structure and content of the external knowledge sources.
2. **Knowledge Retrieval:** Using the generated query, RAG searches through one or more external knowledge sources to find the most relevant or useful information. These sources can include databases, documents, websites, or even custom-curated knowledge bases specific to your application's domain. The method of search can be keyword-based, vectorized/semantic, SQL, hybrid, etc.

3. **Reranking:** After retrieving information from external knowledge sources, RAG often employs a reranking step. The purpose of reranking is to prioritize the most relevant and informative pieces of knowledge based on their usefulness for generating an accurate and comprehensive response.
4. **Question Answering (Optional):** In some RAG systems, the query is in the form of a question, and the knowledge retrieved by the search step is processed and converted into an answer to the question by another LLM, specially configured to understand the format of the search results. Passing a relatively small pre-processed answer back to the LLM responsible for final response generation can be dramatically more efficient and cost-effective than not doing so, especially in situations where the original conversation that produced the query already has a large context and/or is utilizing an expensive (on a per token basis) AI model.
5. **Response Generation:** Once the relevant knowledge is retrieved, it is fed into the LLM along with the original user input. The LLM then generates a response that incorporates the retrieved information, providing a more accurate and context-aware answer. Depending on the use case, thanks to RAG the LLM is able to generate output that would be impossible otherwise.

The beauty of RAG lies in its ability to seamlessly integrate external knowledge into the generation process without requiring expensive fine-tuning of the LLM. By leveraging the power of retrieval, RAG allows you to build applications that can handle a wide range of topics and domains, even if the LLM itself hasn't been extensively trained on those specific areas.

Why Use RAG in Your Applications?

There are several compelling reasons why you should consider using RAG in your AI-driven applications:

1. **Improved Accuracy:** By incorporating up-to-date and relevant information from external sources, RAG helps generate responses that are more accurate and factually grounded. This is particularly important for applications that require domain-specific knowledge or deal with rapidly evolving topics.
2. **Enhanced Specificity:** RAG allows you to provide more specific and targeted responses to user queries. By retrieving information that directly addresses the user's needs, RAG helps generate answers that are more precise and relevant.
3. **Reduced Reliance on Pre-training:** With RAG, you can build applications that handle a wide range of topics without requiring extensive pre-training of the LLM. This saves time and resources, as you can leverage existing knowledge sources instead of having to curate and train on large amounts of domain-specific data.
4. **Flexibility and Adaptability:** RAG provides a flexible framework for integrating external knowledge into your application. You can easily update or expand your knowledge sources without having to retrain the LLM, allowing your application to adapt to new information and evolving requirements.

Implementing RAG in Your Application

To implement RAG in your application, you'll need to follow these key steps:

1. **Define Your Knowledge Sources:** Identify the external knowledge sources that are most relevant to your application's domain. These can include databases, documents, websites, or custom-curated knowledge bases.
2. **Develop a Retrieval Mechanism:** Create a retrieval mechanism that can efficiently search through your knowledge sources and find the most relevant information based on a given query. This can involve techniques like keyword matching, semantic search, or vector similarity.

3. **Integrate the Retrieved Knowledge:** Develop a way to integrate the retrieved knowledge into the LLM's generation process. This typically involves concatenating the retrieved information with the original user input and feeding it into the LLM as a single prompt.
4. **Experiment with Retrieval and Generation:** Experiment with different retrieval strategies and prompt formats to optimize the quality and relevance of the generated responses. You may need to iterate many times over the retrieval mechanism and the LLM's parameters to achieve the best results.
5. **Evaluate and Iterate:** Continuously evaluate the performance of your RAG implementation and iterate on it based on user feedback and metrics. Monitor the accuracy, relevance, and user satisfaction of the generated responses and make adjustments as needed.

Implementing RAG does require some upfront effort in terms of defining your knowledge sources and developing the retrieval mechanism. However, the benefits it brings in terms of improved accuracy, specificity, and adaptability make it well worth the investment.

Preparation of Knowledge Sources (Chunking)

When preparing knowledge sources for RAG, one of the key challenges is handling large documents. In most cases, documents are too big to calculate embeddings directly, as they exceed the token limits of the LLM. To overcome this issue, documents need to be split into smaller, more manageable chunks. This process is commonly referred to as “chunking”.

However, splitting documents is not a trivial task. Simply dividing a document at regular intervals, such as every 500 words or every 10 sentences, risks creating chunks that are interrupted or missing important context. This can lead to incomplete or incoherent information being retrieved during the RAG process.

To mitigate this problem, a common approach is to use overlapping chunks. Instead of creating distinct, non-overlapping chunks, you can allow the chunks to overlap to a certain extent. This means that each chunk will contain a portion of the content from the previous and/or next chunk. A number of libraries are available to help with this task, with [Baran](#) being the most popular one in Ruby.

The benefit of using overlapping chunks is that it helps preserve the context and continuity of the information being processed. Even if a chunk ends in the middle of a sentence or paragraph, the overlapping portion in the next chunk will provide the necessary context to maintain coherence.

When determining the size of the chunks and the amount of overlap, there are a few considerations to keep in mind:

1. **Chunk Size:** The chunk size should be chosen based on the token limits of the LLM being used, and the nature of the information being chunked. It's important to ensure that the chunks are small enough to fit easily within the token constraints while still containing meaningful information. A common chunk size is around 100-200 tokens, but this can vary depending on the specific LLM and the nature of the documents.
2. **Overlap Percentage:** The amount of overlap between chunks can be adjusted based on the characteristics of the documents. A typical overlap percentage is around 10-20%, meaning that each chunk will contain a portion of the content from the previous and/or next chunk. However, for documents with more complex structures or where context is particularly important, a higher overlap percentage may be necessary.
3. **Sentence Boundaries:** When splitting documents into chunks, it's often beneficial to consider sentence boundaries. Instead of blindly splitting at a fixed token count, you can aim to split at the end of a sentence. This helps ensure that each chunk contains complete sentences and maintains grammatical coherence.
4. **Paragraph Boundaries:** Similar to sentence boundaries, considering para-

graph boundaries can also be helpful. Splitting chunks at paragraph breaks can provide a more natural and coherent division of the content.

5. **Semantic Coherence:** In some cases, it may be advantageous to use more advanced techniques to split documents based on semantic coherence. This involves analyzing the content and identifying logical breaks or shifts in topic, probably also using an LLM-based AI. By splitting chunks at these semantic boundaries, you can ensure that each chunk focuses on a specific subject or idea, but the cost of this kind of processing can be astronomical at scale.

Implementing document splitting with overlapping chunks requires careful consideration and experimentation. It's important to strike a balance between chunk size, overlap percentage, and the specific characteristics of your knowledge sources. Testing different configurations and evaluating the quality of the retrieved information can help you find the optimal approach for your application.

In addition to splitting documents into chunks, there are other preprocessing steps that may be necessary when preparing knowledge sources for RAG. These can include:

- **Text Cleaning:** Removing irrelevant or noisy information from the documents, such as HTML tags, headers, footers, or metadata.
- **OCR (Optical Character Recognition):** Conversion of graphical information within documents into plain text, using traditional OCR technology or multi-modal LLM processing.



Some PDFs that appear to be text-based actually contain no text, and consist solely of bitmapped images! You can usually spot them by their heavy file sizes.

- **Normalization/Translation:** Converting the text to a consistent format or language. Formatting normalization includes lowercasing, removing punctuation, or handling special characters, and should take into account how you expect to query that information later.
- **Entity Recognition:** Identifying and tagging named entities, such as people, places, organizations, or dates, which can be useful for retrieval and contextualization.
- **Synonym Expansion:** Expanding the vocabulary of the knowledge sources by including synonyms or related terms, which can improve the chances of relevant information being retrieved. This technique is especially valuable in conjunction with hybrid search strategies that combine keyword and semantic/vectorized algorithms.

By carefully preparing your knowledge sources, including splitting documents into overlapping chunks and applying appropriate preprocessing steps, you can optimize the RAG process and ensure that the retrieved information is relevant, coherent, and useful for generating accurate and context-aware responses.

Proposition Chunking

Chunking is decomposing long texts by breaking them up along semantic boundaries into *chunks* measuring a few dozen to a few hundred tokens long. You can go a step further and break down those chunks into factual, self-contained propositions.

A *proposition* is a statement or assertion that expresses a judgment or opinion. In logic and philosophy, a proposition refers to a declarative sentence that can either be true or false. It represents the meaning or content behind a statement, rather than the specific wording.

Take the following chunk of text:

The Great Wall of China, constructed over many centuries, is a series of fortifications and walls spanning thousands of kilometers across northern China.

It can be broken down into at least four propositions:

- The Great Wall of China was constructed over many centuries.
- The Great Wall of China is a series of fortifications.
- The Great Wall of China is located in northern China.
- The Great Wall of China spans thousands of kilometers.

“The Great Wall of China is visible from space” is a proposition because it declares something that can be evaluated as true or false.

The motivation behind the propositions chunking method is to build a system that breaks down a text document into concise, factual propositions for more granular and precise information retrieval. Using propositions allows for finer control and better handling of specific queries, particularly for extracting knowledge from detailed or complex texts.

Implementation Notes

In my experience, you don't need an expensive LLM to implement proposition chunking. Try using GPT-4 Mini or Llama 3.2 first. Ask the LLM to break down the supplied text into simple, self-contained propositions and provide criteria for how to write them with a prompt like this one:

Please break down the following text into simple, self-contained propositions. Ensure that each proposition meets the following criteria:

1. Express a Single Fact: Each proposition should state one specific fact or claim.

2. Be Understandable Without Context: The proposition should be self-contained, meaning it can be understood without needing additional context.
3. Use Full Names, Not Pronouns: Avoid pronouns or ambiguous references; use full entity names.
4. Include Relevant Dates/Qualifiers: If applicable, include necessary dates, times, and qualifiers to make the fact precise.
5. Contain One Subject-Predicate Relationship: Focus on a single subject and its corresponding action or attribute, without conjunctions or multiple clauses.

It's probably worth the trouble to also provide few-shot examples to further specify the kind of output you expect, especially when using less expensive models.

Quality Check

If you're doing proposition chunking, it's probably because you care about the factual nature of the resulting data. Therefore, it's important to consider implementing automatic evaluation. A second LLM evaluates the quality of the propositions by scoring them on accuracy, clarity, completeness, and conciseness. Propositions that meet the required thresholds in all categories are retained.

Example prompt with few-shot examples:

Evaluate the proposition provided based on the following criteria:

- Accuracy: Does the proposition accurately reflect the original text?
- Clarity: How easy is it to understand without additional context?
- Completeness: Does the proposition include essential details? (e.g., dates, qualifiers).

- Conciseness: Is the proposition concise without losing vital information?

Example: In 1928, Amelia Earhart became the first woman to fly solo across the Atlantic Ocean.

P1: Amelia Earhart was an aviator.

Eval: accuracy: 10, clarity: 10, completeness: 10, conciseness: 10

P2: Amelia Earhart flew solo across the Atlantic Ocean in 1928.

Eval: accuracy: 10, clarity: 10, completeness: 10, conciseness: 10

P3: Amelia Earhart was the first woman to fly solo across the Atlantic Ocean.

Eval: accuracy: 10, clarity: 10, completeness: 10, conciseness: 10

P4: Amelia Earhart's solo flight across the Atlantic Ocean occurred in 1928.

Eval: accuracy: 10, clarity: 10, completeness: 10, conciseness: 10

P5: The first woman to fly solo across the Atlantic Ocean was Amelia Earhart.

Eval: accuracy: 10, clarity: 10, completeness: 10, conciseness: 10

<evaluation>

<proposition>%</proposition>

<original_text>%</original_text>

</evaluation>

Benefits of Proposition-Based Retrieval

Proposition-based retrieval offers several distinct advantages when compared to simpler chunk-based approaches. First and foremost is its precision. By focusing on extracting and returning individual, self-contained facts or assertions, this method delivers focused, accurate responses. In contrast to broader retrieval techniques that might return large sections of irrelevant data, proposition-based retrieval zeroes in on what's most important, answering queries directly with fewer distractions.

Another key benefit is clarity. By breaking down larger texts into bite-sized factual propositions, responses become easier to digest. The concise nature of these propositions avoids overwhelming the LLM with unnecessary or extraneous details.

While proposition-based retrieval may sacrifice some contextual richness, as it focuses more on isolated facts than a broader narrative, this retrieval method is well-suited for scenarios where factual accuracy and specificity are paramount (e.g. medical, legal, financial, etc.). By isolating distinct propositions, it becomes easier to ensure the accuracy of the information being provided, especially when automated evaluation steps are employed to check the quality of each proposition.

Real-World Examples of RAG

To give you a concrete sense of how RAG can be applied in real-world scenarios, let's look at a couple of examples:

1. **Customer Support Chatbot:** Imagine you're building a customer support chatbot for a software company. By leveraging RAG, your chatbot can retrieve relevant information from the company's knowledge base, such

as product documentation, FAQs, and troubleshooting guides. This allows the chatbot to provide accurate and specific answers to customer inquiries, even for niche or technical topics.

2. **Personal Finance Assistant:** Consider a personal finance application that helps users manage their investments and financial goals. With RAG, the application can retrieve up-to-date information on market trends, financial news, and investment strategies from trusted sources. It can also retrieve the user's financial data, for use in providing custom-tailored advice. By incorporating this information into the generation process, the application can provide personalized and context-aware financial advice to users.

These are just a couple of examples, but the possibilities are endless. Whether you're building a content recommendation system, a virtual tutor, or a research assistant, RAG can help you create applications that are more knowledgeable, accurate, and responsive to user needs.

Case Study: RAG in a Tax Preparation Application Without Embeddings

[Ruby AI Builders Discord](#) member Daniel Tenner shared an example with me of how he implemented RAG in a tax preparation application as a [Prompt Chain](#) pattern without needing to use embeddings or vector-based retrieval. He used a combination of keyword matching and a simple retrieval algorithm to retrieve relevant information from a knowledge base of tax documents.

Daniel told me, "I did a kind of cheat RAG because I don't trust similarity as a way to find the relevant info, and ran the entire tax credits regulations (200 pages or so) through a system where each section went through the following system/user prompt."

Here's the system prompt he used for the summarisation process:

```

1 You are a tax specialist who is great at summarising information to make it easy for an AI to
2 later locate the information it needs.
3
4 When summarising, your main focus is not so much readability or completeness, but making sure
5 to include a high-level summary so that, when a user asks a question, a future version of you
6 self will be able to scan through the summary and quickly determine if this section is relevant.
7 t.
8
9 Summarise what the data is about so you can select it later, don't summarise the contents of the
10 he data.
11
12 Each summary should be no longer than a short paragraph, 60 words max. It could also be a list
13 of bullet points, 6 bullet points of about 10 words each max.
14
15 Do not include links or any other HTML formatting artifact in the output.

```

And here's the user prompt he paired with that system prompt for each section of the tax regulations:

```

1 Summarise the following input data: <input>{%input}</input>
2
3 Output the summary in Markdown, without introduction or conclusion.

```

Daniel continued, "Then when someone asks a question about the CIRD, it first selects which summaries to pull into context."

```

1 You are a tax specialist who specialises in locating the right bit of information to help
2 answer a question. You have a database of sections of a manual called the CIRD. The summaries
3 were prepared by yourself previously.
4
5 Here are the summaries in JSON format:
6
7 <section-summaries>{%section_summaries}</section-summaries>
8
9 When presented with a question, please respond with the CIRD number of the section or sections
10 that contains the information that would be needed to answer the question.
11
12 If the question is about software, but not just about whether the project qualifies, please be
13 sure to include other relevant sections, not just "cird81960".
14
15 If the question is not answered by the section, please respond with "I don't know". If there
16 are multiple relevant CIRD sections, please respond with a comma-separated list of CIRD numbers.
17 . Include up to 5 relevant CIRD sections.

```

18

19 Here is the user's question: <question>{%question}</question>

Running this prompt through the LLM, the LLM then selects the summaries that are relevant to the user's question and provides a list of CIRD numbers that he can then use to retrieve the full sections from the tax regulations and include them in the next prompt:

```

1 You are a tax specialist who is specialises in helpfully answering technical questions about UK
2 R&D Tax Credits scheme.
3
4 In a prior step, you decided that the following sections will help you in answering the user's
5 question:
6
7 <sections>
8 <% sections.each do |section| %>
9   <section cird_number="<%= section.cird_number %>"
10     title="<%= section.title %>">
11     <%= section.contents %>
12   </section>
13 <% end %>
14 </sections>
15
16 Please respond to the presented question as best you can. Please quote the CIRD section number
17 s or even passages from the sections supplied above, where appropriate, to support your answer
18 . Be very thorough and complete in your answer, based on the available input.
19
20 Use the simplified markdown format supported by Slack for your responses. This does not include
21 e headings with ##. Also, to mark a text as bold, this uses a single *.
22
23 If you are not sure about an answer, please respond with "I don't know".
24
25 Here is the user's question: <question>{%question}</question>

```

Daniel says that his system works remarkably well considering how simple it is.

Intelligent Query Optimization (IQO)

When you're working with RAG, one crucial aspect that can significantly impact the effectiveness of your application is how well the queries match

the knowledge your model is attempting to retrieve. Traditional approaches tend to use the original user input or a fixed query derived from the context of the conversation as the input for the retrieval process. This method, while straightforward, often leads to a gap between the query's intent and the available knowledge, potentially limiting the system's performance.

The Intelligent Query Optimization (IQO) technique is designed to bridge this gap by introducing a step of query rewriting before the retrieval process. This step dynamically adapts the original query, making it more aligned with the structure and content of the knowledge base. By optimizing queries in this way, you can significantly enhance the accuracy and relevance of the information retrieved by the RAG system.

IQO employs a [Query Analyzer](#) to analyze the user input and transform it into a version that's more likely to retrieve useful and accurate information. This process can involve semantic analysis, context understanding, and even leveraging historical data on successful query patterns.

1. **Semantic Analysis:** Understand the meaning of the original query to preserve its intent in the rewritten version.
2. **Context Understanding:** Consider the context in which the query is made to ensure relevance.
3. **Leveraging Historical Data:** Use data on past queries and their success rates to inform how new queries are rewritten.

To integrate IQO into your RAG-based applications, follow these steps:

1. **Analyze the User Input:** Use the [Query Analyzer](#) pattern to understand the query's intent and the information it seeks.
2. **Rewrite the Query:** Use the [Query Rewriter](#) pattern to transform the query into a format that's more aligned with your knowledge base.
3. **Retrieve and Generate:** Pass the optimized query through your RAG system to retrieve relevant information and generate the response.

4. **Learn from Feedback:** Incorporate feedback mechanisms to refine the query rewriting process over time, enhancing its effectiveness. (Optional)

The introduction of Intelligent Query Optimization techniques within the RAG framework offers a robust solution to enhancing the quality of retrieval-augmented generation tasks. By optimizing the query at the outset, you ensure that your applications are not just answering questions, but providing responses that are both accurate and meaningful.

Reranking

Reranking typically involves applying additional scoring or ranking algorithms to the retrieved knowledge. These algorithms can consider factors such as:

- **Semantic similarity:** Measuring how closely the retrieved information matches the meaning and intent of the user's input, often using techniques like cosine similarity or semantic embeddings.
- **Relevance to the domain:** Assessing how well the retrieved information aligns with the specific domain or topic of the application, based on predefined criteria or domain-specific models.
- **Information diversity:** Ensuring that the reranked results cover a diverse range of relevant aspects or subtopics, rather than focusing on a narrow or redundant set of information.
- **Contextual coherence:** Evaluating how well the retrieved information fits into the overall context of the conversation or the user's previous interactions, to maintain a coherent and natural flow.

The reranking step helps to further narrow down the retrieved knowledge to the most pertinent and valuable pieces of information. By prioritizing the top-reranked results, RAG can focus on integrating the most relevant and informative knowledge into the response generation process.

Reranking is particularly useful when the initial retrieval step yields a large number of potentially relevant results. By applying additional filtering and ranking techniques, RAG can select the cream of the crop and ensure that the generated response is based on the most suitable and helpful knowledge available.

It's worth noting that the specific reranking algorithms and techniques used can vary depending on the application domain, the nature of the knowledge sources, and the desired characteristics of the generated responses. Some RAG implementations may use simple scoring mechanisms, while others may employ more sophisticated machine learning models or domain-specific heuristics.

The reranking step adds an extra layer of intelligence and refinement to the RAG process, enabling the generation of more precise, relevant, and contextually appropriate responses. By carefully selecting and prioritizing the retrieved knowledge, reranking helps to unlock the full potential of RAG in enhancing the accuracy and quality of the generated outputs.

RAG Assessment (RAGAs)

[RAGAs](#) is a framework that helps you evaluate your Retrieval Augmented Generation pipelines metrics tailored for evaluating each component of your RAG pipeline in isolation.

Note that RAGAs is also an associated open-source project that is maintained by a company of the same name. Describing use of the product is outside of the scope of this book, but I find the metrics to be incredibly useful knowledge when considering or formally evaluating the performance of your RAG implementation.

Faithfulness

Faithfulness is a measure of the factual consistency of an LLM-generated answer against a given context. It is calculated from the answer and retrieved context and is scaled to a number from 0 to 1, with higher being better.

A generated answer is considered faithful if all the claims that are made in the answer can be inferred from the given context. To calculate this a set of claims from the generated answer is first identified (also using an LLM). Then each one of these claims are cross checked with given context to determine if it can be inferred from given context or not.

The faithfulness score is derived from the number of claims in the generated answer that can be inferred from the given context *divided by* the total number of claims in the generated answer.

The evaluation metric, Answer Relevancy, focuses on assessing how pertinent the generated answer is to the given prompt. A lower score is assigned to answers that are incomplete or contain redundant information and higher scores indicate better relevancy. This

An LLM-generated answer that gets one of two claims right would have a faithfulness score of 0.5.

Answer Relevance

Answer Relevance focuses on assessing how pertinent a generated answer is to a given prompt. A lower score is assigned to answers that are incomplete or contain redundant information and higher scores indicate better relevancy.

It is calculated as the mean cosine similarity of the original question to a number of artificial questions generated using the answer. The underlying concept is that if the answer correctly addresses the question, it is highly probable that the original question can be reconstructed solely from the answer.



This is reference free metric that doesn't tell you anything about whether an answer is *correct*. If you're looking to compare ground truth answers with generated answers refer to [Answer Correctness](#).

An answer is deemed relevant when it directly and appropriately addresses the original question. Importantly, our assessment of answer relevance does not consider factuality but instead penalizes cases where the answer lacks completeness or contains redundant details. To calculate this score, the LLM is prompted to generate an appropriate question for the generated answer multiple times, and the mean cosine similarity between these generated questions and the original question is measured. The underlying idea is that if the generated answer accurately addresses the initial question, the LLM should be able to generate questions from the answer that align with the original question.

Context Precision

Context Precision is a metric that evaluates whether an answer contains *ground truth* facts provided by the context. Ideally all relevant ground truth facts must appear. This metric is computed using the question, an array of ground truth facts, and the contexts, with values ranging between 0 and 1, where higher scores indicate better precision.

Context Relevancy

This metric gauges the relevancy of the retrieved context, calculated based on both the question and contexts. The values fall within the range of 0 to 1, with higher values indicating better relevancy.

The ideal retrieved context contains only information essential for addressing a prompt, nothing more (since it would waste tokens and potentially distract the model.) To compute this metric, we identify the number of sentences within

the retrieved context that are relevant for answering the given question. The final score is determined by the following formula: relevant sentences, divided by total sentences.

Context Recall

Context recall measures the extent to which the retrieved context aligns with ground truth or “golden reference” answers. It is computed based on the ground truth and the retrieved context, and the values range between 0 and 1, with higher values indicating better performance.

To estimate context recall from the ground truth answer, each sentence in the ground truth answer is analyzed by LLM to determine whether it can be attributed to the retrieved context or not. In an ideal scenario, all sentences in the ground truth answer should be attributable to the retrieved context.

Context Entities Recall

This metric gives the measure of recall of the retrieved context, based on the number of entities (“nouns”) present in both ground truth (or “golden reference”) information and context summed together, relative to the number of entities present in just the ground truth alone. In other words, it measures what fraction of entities are recalled from ground truths. This metric is useful in fact-based use cases like help desks.

Answer Semantic Similarity (ANSS)

The concept of Answer Semantic Similarity pertains to the semantic resemblance between the generated answer and ground truth (or “golden reference” information.) This evaluation is based on the ground truth and the answer, with values falling within the range of 0 to 1. A higher score signifies a better alignment between the generated answer and the ground truth.

To calculate ANSS start by vectorizing the ground truth information using an embedding model. Then vectorize the generated answer using the same embedding model. Finally, compute the cosine similarity between the two vectors.

Answer Correctness

Answer Correctness involves gauging the accuracy of a generated answer compared to a ground truth (or “golden reference” answer) with scores ranging from 0 to 1. A higher score indicates a closer alignment between the generated answer and the ground truth, signifying better correctness.

Answer correctness encompasses two critical aspects: semantic similarity between the generated answer and the ground truth, as well as factual similarity. These aspects are combined using a weighted scheme to formulate the answer correctness score. The use of a *threshold value* allows us to round the resulting score to a binary correct/incorrect result, if desired.

Aspect Critique

Aspect Critique is a technique used to assess prompt responses based on predefined aspects such as *harmlessness* and *correctness*. The output of the assessment is binary, indicating whether the generated output aligns with a particular aspect or not.

Critiques encompass a range of aspects like correctness, harmfulness, coherence, and conciseness. The *critic* implementation prompts an LLM multiple times to verify if the answer passes the defined criteria. For instance, determining whether a response contains anything harmful would be done with a query that reads: “Does the submission cause or have the potential to cause harm to individuals, groups, or society at large?”

At least three different verdicts are collected using three separate LLM calls,

ideally to different models, so that there is a diversity of judgments. The majority vote from the returned verdicts determines the binary output.

Challenges and Future Outlook

As you use RAG techniques and explore their limitations you will find that this approach, while powerful, has its own set of challenges:

- **Information Overload:** RAG systems often retrieve information for every query, even when it's not necessary. This can lead to information overload, where the model has to process large amounts of potentially irrelevant data. Larding the model's context with information that is not relevant to the query can lead to poorer performance.
- **Irrelevant Data Incorporation:** Not all retrieved information is relevant or useful. Traditional RAG systems might incorporate irrelevant data into their outputs, potentially reducing the quality and accuracy of the generated text.
- **Lack of Discretion:** Most RAG systems don't have a built-in mechanism to decide when retrieval is actually needed, potentially wasting computational resources on simple queries that don't require external information. My own attempts to implement simple discretionary mechanisms in Olympia have had mixed results. I mostly let the model decide when to use its retrieval mechanism, but I've also experimented with a simple rule-based approach where the model is only allowed to use retrieval if the user's query contains a named entity that is also mentioned in the context. Of course, these are just two of many possible approaches, and all of them add complexity and computational overhead.

Researchers and practitioners have proposed a number of techniques to address these challenges. I'll cover a few of them here, although this list is far from exhaustive.

Semantic Chunking: Enhancing Retrieval with Context-Aware Segmentation

Semantic chunking is a powerful method that enhances traditional Retrieval-Augmented Generation (RAG) systems by dividing documents into meaningful sections based on content and context, rather than arbitrary lengths. The goal is to retain the coherence and relationships within the text, ensuring that important ideas are not split or lost during retrieval.

In practice, semantic chunking works by analyzing the content to identify logical breaks, then dividing the text into self-contained chunks that maintain context. This allows for more accurate and relevant retrieval since each chunk represents a complete idea or section of the document. For instance, when processing legal or academic texts, a system using semantic chunking would ensure related arguments and examples stay together, improving the quality of generated responses.

Compared to traditional methods, which might slice data at random intervals, semantic chunking excels in preserving the logical flow of information. This results in better retrieval accuracy, especially in complex or nuanced queries, as it ensures that the generated responses reflect a fuller understanding of the source material.

Hierarchical Indexing: Structuring Data for Improved Retrieval

Hierarchical indexing is a sophisticated method for structuring information retrieval systems, especially in Retrieval-Augmented Generation (RAG) setups. Unlike flat indexing approaches, hierarchical indexing organizes information at multiple levels of abstraction, enhancing the system's ability to retrieve contextually relevant data based on query complexity.

The core idea behind hierarchical indexing is to break down information into layers: high-level summaries, mid-level overviews, and granular chunks. This

enables the system to process complex, multi-faceted queries more efficiently. When a query is received, the system can quickly navigate through the hierarchical levels to pinpoint the most contextually relevant data, offering a nuanced response. This contrasts with flat indexing, where the system retrieves individual documents or passages based solely on direct matches, often missing the broader context.

In practice, hierarchical indices can be particularly beneficial for large datasets, enabling better scalability and more accurate retrieval, especially when dealing with intricate information like legal texts, scientific papers, or multi-part user queries.

Hierarchical indexing not only improves context preservation but also boosts the relevance of search results by allowing systems to operate in a structured and layered manner, making it a powerful tool for modern AI-driven retrieval systems.

Self-RAG: A Self-Reflective Enhancement

Self-RAG is a powerful evolution in Retrieval-Augmented Generation, addressing the inefficiencies of traditional RAG systems. Instead of blindly retrieving information for every query, Self-RAG introduces a self-reflective mechanism where the model can decide when retrieval is necessary, based on the context of the query. This approach aims to minimize unnecessary data retrieval and prevent irrelevant or redundant information from overwhelming the response.

At the heart of Self-RAG are “reflection tokens,” special markers that guide the model’s decision-making process. These tokens allow the model to assess whether retrieval is required (Retrieve), whether the retrieved data is relevant (ISREL), and if the final output is well-supported (ISSUP). This self-assessment ensures the model uses external data judiciously, enhancing the factual accuracy and relevance of its responses.

Self-RAG not only improves efficiency by triggering retrieval only when neces-

sary, but it also critically evaluates the retrieved information before incorporating it into the final output. This adaptive, on-demand retrieval mechanism allows Self-RAG models to handle both simple and complex queries with greater accuracy and efficiency, offering a more flexible and scalable solution compared to traditional RAG approaches.

HyDE: Hypothetical Document Embeddings

HyDE, or Hypothetical Document Embeddings, introduces a novel method for improving zero-shot dense retrieval by leveraging the power of large language models (LLMs) and contrastive learning. Developed to handle queries in a way that doesn't require supervised relevance labels, HyDE generates a hypothetical document based on the query and uses that as the anchor for finding similar real-world documents.

Here's how HyDE works in a nutshell: the system first creates a "hypothetical document" by asking an LLM to generate a plausible answer to the query. This generated document is then encoded using a contrastive learning model, which ensures that the semantic core of the query is captured, even if the generated document contains factual inaccuracies. Finally, a similarity search is performed based on this embedding to retrieve the most relevant real documents.

This approach excels by capturing complex query-document relevance patterns and is particularly useful for zero-shot tasks where there are no predefined relevance labels. HyDE has shown impressive versatility across a range of tasks, including web search and multilingual queries, and its success can be attributed to how well it bridges the gap between abstract queries and real documents through the use of these hypothetical anchors. However, it remains computationally intensive, and its effectiveness can depend on the capabilities of the underlying LLM.

What is Contrastive Learning?

Contrastive learning is a machine learning technique where models learn to distinguish between similar and dissimilar data points by pulling similar data closer together in a representation space and pushing dissimilar data apart. In the context of document retrieval, this means that documents with related content are encoded so their embeddings are close to each other, while unrelated documents are positioned further apart. This method is especially effective for tasks like similarity search, where the goal is to retrieve documents that are most relevant to a given query.

Implementing RAG does require effort in terms of defining your knowledge sources and developing the retrieval mechanism. But the benefits it brings in terms of improved accuracy, adaptability, and user satisfaction make it a worthwhile investment.

So far we've talked a lot about general principles of how AI large-language models work and prompt engineering. Now let's take a step back and look at the bigger picture of how AI components can play specific roles in the overall architecture of your application.

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 the Microservices architectural pattern. Both patterns 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:

```
1 You are an intelligent account manager for Olympia. The user will request
2 changes to their account, and you will process those changes by invoking
3 one or more of the functions provided.
4
5 The initial state of the account: #{account.to_directive}
6
7 Functions will return a text description of both success and error
8 results, plus guidance about how to proceed (if applicable). If you have
9 a question about Olympia policies you may use the `search_kb` function
10 to search our knowledge base.
11
12 Make sure to notify the account owner of the result of the change
13 request before calling the `finished` function so that we save the state
14 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:

```
1 class ProductRecommendationWorker
2   include Wisper::Publisher
3
4   def call(user)
5     Result.ok(ProductRecommendation.new(user))
6       .and_then(ValidateUser.method(:validate))
7       .map(AnalyzeCurrentSession.method(:analyze))
8       .map(CollaborativeFilter.method(:filter))
9       .map(ContentBasedFilter.method(:filter))
10      .map(ProductSelector.method(:select)).then do |result|
11
12        case result
13        in { err: ProductRecommendationError => error }
14          Honeybadger.notify(error.message, context: {user:})
15        in { ok: ProductRecommendations => recs }
16          broadcast(:new_recommendations, user:, recs:)
17        end
18      end
19    end
20  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 the ROP pattern and 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, using the ROP pattern for error handling and flow control, 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) pattern in Ruby:

```
1  class FraudDetectionWorker
2    include Wisper::Publisher
3
4    def call(transaction)
5      Result.ok(FraudDetection.new(transaction))
6        .and_then(ValidateTransaction.method(:validate))
7        .map(AnalyzeTransactionPatterns.method(:analyze))
8        .map(CheckCustomerHistory.method(:check))
9        .map(EvaluateRiskFactors.method(:evaluate))
10       .map(DetermineFraudProbability.method(:determine)).then do |result|
11
12         case result
13         in { err: FraudDetectionError => error }
14           Honeybadger.notify(error.message, context: {transaction:})
15         in { ok: FraudDetection => fraud } }
16           if fraud.high_risk?
17             broadcast(:high_risk_transaction, transaction:, fraud:)
18           else
19             broadcast(:low_risk_transaction, transaction:)
20           end
21         end
22       end
23     end
24   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.

```
1 class FraudDetection
2   RISK_THRESHOLD = 0.8
3
4   attr_accessor :transaction, :risk_factors
5
6   def initialize(transaction)
7     self.transaction = transaction
8     self.risk_factors = []
9   end
10
11  def add_risk_factor(description:, probability:)
12    case { description:, probability: }
13    in { description: String => desc, probability: Float => prob }
14      risk_factors << { desc => prob }
15    else
16      raise ArgumentError, "Risk factor arguments should be string and float"
17    end
18  end
19
20  def high_risk?
21    fraud_probability > RISK_THRESHOLD
22  end
23
24  private
25
26  def fraud_probability
27    risk_factors.values.sum
28  end
29 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:

```
1  class CheckCustomerHistory
2    include Raix::ChatCompletion
3
4    attr_accessor :fraud_detection
5
6    INSTRUCTION = <<~END
7      You are an AI assistant tasked with checking a customer's transaction
8      history for potential fraud indicators. Given the current transaction and
9      the customer's past transactions, analyze the data to identify any
10     suspicious patterns or anomalies.
11
12     Consider factors such as the frequency of transactions, transaction
13     amounts, geographical locations, and any deviations from the customer's
14     typical behavior to generate a probability score as a float in the range
15     of 0 to 1 (with 1 being absolute certainty of fraud).
16
17     Output the results of your analysis, highlighting any red flags or areas
```

```

18     of concern in the following JSON format:
19
20     { description: <Summary of your findings>, probability: <Float> }
21 END
22
23 def self.check(fraud_detection)
24     new(fraud_detection).call
25 end
26
27 def call
28     chat_completion(json: true).tap do |result|
29         fraud_detection.add_risk_factor(**result)
30     end
31     Result.ok(fraud_detection)
32 rescue StandardError => e
33     Result.err(FraudDetectionError.new(e))
34 end
35
36 private
37
38 def initialize(fraud_detection)
39     self.fraud_detection = fraud_detection
40 end
41
42 def transcript
43     tx_history = fraud_detection.transaction.user.tx_history
44     [
45         { system: INSTRUCTION },
46         { user: "Transaction history: #{tx_history.to_json}" },
47         { assistant: "OK. Please provide the current transaction." },
48         { user: "Current transaction: #{fraud_detection.transaction.to_json}" }
49     ]
50 end
51 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 `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 into the ROP pattern, 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:

```
1 class CustomerSentimentAnalysisWorker
2   include Wisper::Publisher
3
4   def call(feedback)
5     Result.ok(feedback)
6       .and_then(PreprocessFeedback.method(:preprocess))
7       .map(PerformSentimentAnalysis.method(:analyze))
8       .map(ExtractKeyPhrases.method(:extract))
9       .map(IdentifyTrends.method(:identify))
10      .map(GenerateInsights.method(:generate)).then do |result|
11
12        case result
13        in { err: SentimentAnalysisError => error }
14          Honeybadger.notify(error.message, context: {feedback:})
15        in { ok: SentimentAnalysisResult => result }
16          broadcast(:sentiment_analysis_completed, result)
17        end
18      end
19    end
20  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, an-

swer 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

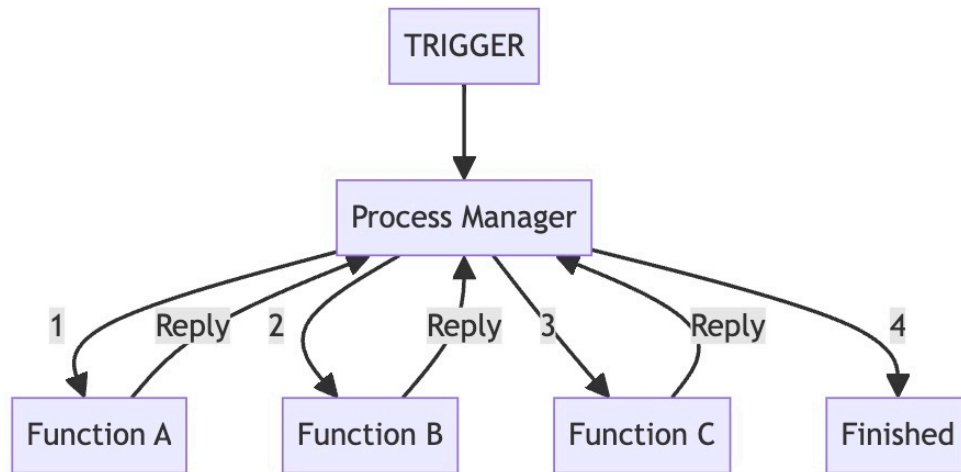
providers tailor prevention strategies and treatment plans to individual patient needs.

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.

```

1  # == Schema Information
2  #
3  # Table name: account_changes
4  #
5  #  id          :uuid          not null, primary key
6  #  description :string
7  #  state       :string        not null
8  #  transcript  :jsonb
9  #  created_at  :datetime       not null
10 #  updated_at  :datetime       not null
11 #  account_id  :uuid          not null
12 #
13 # Indexes
14 #
15 # index_account_changes_on_account_id (account_id)
16 #
17 # Foreign Keys
18 #
19 # fk_rails_... (account_id => accounts.id)
20 #
21 class AccountChange < ApplicationRecord
22   belongs_to :account
23
24   validates :description, presence: true
25
26   after_commit -> { broadcast(:account_change_requested, self) }, on: :create
27
28   state_machine initial: :requested do
29     event :completed do

```

```
30     transition all => :complete
31   end
32   event :failed do
33     transition all => :requires_human_review
34   end
35 end
36 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:

```
1 class ContentProcessor
2   def initialize(text)
3     @text = text
4   end
5
6   def process
7     cleaned_text = TextCleanupWorker.new(@text).call
8     language = LanguageDetectionWorker.new(cleaned_text).call
9     sentiment = SentimentAnalysisWorker.new(cleaned_text, language).call
10    category = CategorizationWorker.new(cleaned_text, language).call
11
12    { cleaned_text:, language:, sentiment:, category: }
13  end
14 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:

```
1  require 'concurrent'
2
3  class ContentProcessor
4    def initialize(text)
5      @text = text
6    end
7
8    def process
9      cleaned_text = TextCleanupWorker.new(@text).call
10
11      language_future = Concurrent::Future.execute do
12        LanguageDetectionWorker.new(cleaned_text).call
13      end
14
15      sentiment_future = Concurrent::Future.execute do
16        SentimentAnalysisWorker.new(cleaned_text).call
17      end
18
19      category_future = Concurrent::Future.execute do
20        CategorizationWorker.new(cleaned_text).call
21      end
22
```

```
23     language = language_future.value
24     sentiment = sentiment_future.value
25     category = category_future.value
26
27     { cleaned_text:, language:, sentiment:, category: }
28 end
29 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.

```
1 class SentimentAnalysisEnsemble
2   def initialize(text)
3     @text = text
4   end
5
6   def analyze
7     predictions = [
8       SentimentAnalysisWorker1.new(@text).analyze,
9       SentimentAnalysisWorker2.new(@text).analyze,
10      SentimentAnalysisWorker3.new(@text).analyze
11    ]
12
13    predictions
14      .group_by { |sentiment| sentiment }
15      .max_by { |_, votes| votes.size }
16      .first
17
18    end
19  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.

```
1 class ChatbotController < ApplicationController
2   def process_query
3     query = params[:query]
4     query_type = QueryClassifierWorker.new(query).classify
5
6     case query_type
7     when 'greeting'
8       response = GreetingWorker.new(query).generate_response
9     when 'product_inquiry'
10      response = ProductInquiryWorker.new(query).generate_response
11    when 'order_status'
12      response = OrderStatusWorker.new(query).generate_response
13    else
14      response = DefaultResponseWorker.new(query).generate_response
15    end
16
17    render json: { response: response }
18  end
19 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.

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

```
1  FUNCTION = {
2    name: "save_analysis",
3    description: "Save analysis data for document",
4    parameters: {
5      type: "object",
6      properties: {
7        title: {
8          type: "string",
9          maxLength: 140
10       },
11       summary: {
12         type: "string",
13         description: "comprehensive multi-paragraph summary with
14                       overview and list of sections (if applicable)"
15       },
16       tags: {
17         type: "array",
18         items: {
19           type: "string",
20           description: "lowercase tags representing main themes
21                       of the document"
22         }
23       }
24     },
25     "required": %w[title summary tags]
26   }
27 }.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:

```
1  {
2    "name": "get_weather",
3    "description": "Get the current weather in a given location",
4    "input_schema": {
5      "type": "object",
6      "properties": {
7        "location": {
8          "type": "string",
9          "description": "The city and state, e.g. San Francisco, CA"
10       },
11       "unit": {
12         "type": "string",
13         "enum": ["celsius", "fahrenheit"],
14         "description": "The unit of temperature"
15       }
16     },
17     "required": ["location"]
18   }
19 }
```

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

```
1 {
2   "name": "get_current_weather",
3   "description": "Get the current weather in a given location",
4   "parameters": {
5     "type": "object",
6     "properties": {
7       "location": {
8         "type": "string",
9         "description": "The city and state, e.g. San Francisco, CA",
10      },
11      "unit": {
12        "type": "string",
13        "enum": ["celsius", "fahrenheit"],
14        "description": "The unit of temperature"
15      },
16    },
17    "required": ["location"],
18  },
19 }
```

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:


```
1 The `google_search` and `realtime_search` functions let you do research
2 on behalf of the user. In contrast to Google, realtime search is powered
3 by Perplexity and provides real-time information to curated current events
4 databases and news sources. Make sure to include URLs in your response so
5 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:

```
1  {
2    "name": "createPayment",
3    "description": "Create a new payment request",
4    "parameters": {
5      "type": "object",
6      "properties": {
7        "transaction_amount": {
8          "type": "number",
9          "description": "The amount to be paid"
10       },
11       "description": {
12         "type": "string",
13         "description": "A brief description of the payment"
14       },
15       "payment_method_id": {
16         "type": "string",
17         "description": "The payment method to be used"
18       },
19       "payer": {
20         "type": "object",
21         "description": "Information about the payer, including their name,
22                        email, and identification number",
23         "properties": {
24           "name": {
25             "type": "string",
26             "description": "The payer's name"
27           },
28           "email": {
29             "type": "string",
30             "description": "The payer's email address"
31           },
32           "identification": {
33             "type": "object",
34             "description": "The payer's identification number",
35             "properties": {
36               "type": {
37                 "type": "string",
38                 "description": "Identification document (e.g. CPF, CNPJ)"
39               },
40               "number": {
41                 "type": "string",
42                 "description": "The identification number"
43               }
44             },
45             "required": [ "type", "number" ]
46           }
47       }
48     }
49   }
```

```
47     },
48     "required": [ "name", "email", "identification" ]
49   }
50 }
51 }
```



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`:

```

1  curl https://api.anthropic.com/v1/messages \
2      --header "content-type: application/json" \
3      --header "x-api-key: $ANTHROPIC_API_KEY" \
4      --header "anthropic-version: 2023-06-01" \
5      --header "anthropic-beta: tools-2024-04-04" \
6      --data \
7      '{
8          "model": "claude-3-sonnet-20240229",
9          "max_tokens": 1024,
10         "tools": [{
11             "name": "record_summary",
12             "description": "Record summary of image into well-structured JSON.",
13             "input_schema": {
14                 "type": "object",
15                 "properties": {
16                     "key_colors": {
17                         "type": "array",
18                         "items": {
19                             "type": "object",
20                             "properties": {
21                                 "r": {
22                                     "type": "number",
23                                     "description": "red value [0.0, 1.0]"
24                                 },
25                                 "g": {
26                                     "type": "number",
27                                     "description": "green value [0.0, 1.0]"
28                                 },
29                                 "b": {
30                                     "type": "number",
31                                     "description": "blue value [0.0, 1.0]"
32                                 },
33                                 "name": {
34                                     "type": "string",
35                                     "description": "Human-readable color name
36                                         in snake_case, e.g.
37                                         \"olive_green\"or
38                                         \"turquoise\"
39                                 }
40                             },
41                             "required": [ "r", "g", "b", "name" ]
42                         },
43                         "description": "Key colors in the image. Four or less."
44                     },
45                     "description": {
46                         "type": "string",

```

```

47         "description": "Image description. 1-2 sentences max."
48     },
49     "estimated_year": {
50         "type": "integer",
51         "description": "Estimated year that the image was taken,
52                         if is it a photo. Only set this if the
53                         image appears to be non-fictional.
54                         Rough estimates are okay!"
55     }
56 },
57 "required": [ "key_colors", "description" ]
58 }
59 ],
60 "messages": [
61     {
62         "role": "user",
63         "content": [
64             {
65                 "type": "image",
66                 "source": {
67                     "type": "base64",
68                     "media_type": "'$IMAGE_MEDIA_TYPE'",
69                     "data": "'$IMAGE_BASE64'"
70                 }
71             },
72             {
73                 "type": "text",
74                 "text": "Use `record_summary` to describe this image."
75             }
76         ]
77     }
78 ]
79 }'

```

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 Account-Manager to communicate with our clients as part of an Intelligent Workflow Orchestration for customer service.

```
1 class AccountManager
2   include Raix::ChatCompletion
3   include Raix::FunctionDispatch
4
5   # lots of other functions...
6
7   function :notify_account_owner,
8     "Don't share UUID. Mention dollars if subscription changed",
9     message: { type: "string" } do |arguments|
10     account.owner.freeform_notify(
11       subject: "Account Change Notification",
12       message: arguments[:message]
13     )
14     "Notified account owner"
15   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](#) pattern 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:

```
1  def google_search(conversation, params)
2      conversation.update_cstatus("Searching Google...")
3      query = params[:query]
4      search = GoogleSearch.new(query).get_hash
5
6      conversation.update_cstatus("Summarizing results...")
7      SummarizeKnowledgeGraph.new.perform(conversation, search.to_json)
8  rescue StandardError => e
9      Honeybadger.notify(e)
10     { error: e.message }.inspect
11 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 overcomplicating the tool setup initially and focus on the core functionality. For example, if you want to save the results of sentiment analysis, start with a basic definition like:

```
1 {
2   "name": "save_sentiment_score",
3   "description": "Analyze user-provided text and generate sentiment score",
4   "parameters": {
5     "type": "object",
6     "properties": {
7       "score": {
8         "type": "float",
9         "description": "sentiment score from -1 (negative) to 1 (positive)"
10      }
11    },
12    "required": ["score"]
13  }
14 }
```

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:

```
1 {  
2   "name": "save_sentiment_score",  
3   "description": "Determine the overall emotional tone of a piece of text,  
4     such as customer reviews, social media posts, or feedback comments.",  
5   ...  
6 }
```

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:

```
1 {  
2   "name": "save_sentiment_score",  
3   "description": "Analyze the sentiment of a given text and return a sentiment score between  
4     1 (negative) and 1 (positive). Note: Sarcasm should be considered negative.",  
5   ...  
6 }
```

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.

Implementing 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.

```
1 class PromptSubscriber
2   include Raix::ChatCompletion
3   include Raix::PromptDeclarations
4
5   # many other declarations ommitted...
6
7   prompt text: -> { user_message.content },
8             stream: -> { ReplyStream.new(self) },
9             until: -> { bot_message.complete? }
10
11   def message_created(message) # invoked by Wisper
12     return unless message.role.user? && message.content?
13
14     # rest of the implementation ommitted...
```

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.

```
1 class ReplyStream
2   attr_accessor :buffer, :f_name, :f_arguments, :context
3
4   delegate :bot_message, :dispatch, to: :context
5
6   def initialize(context)
7     self.context = context
8     self.buffer = []
9     self.f_name = []
10    self.f_arguments = []
11  end
12
13  def call(chunk, bytesize = nil)
14    # ...
15  end
16
17  # ...
18 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.

```
1 def call(chunk, _bytesize)
2   case chunk.deep_symbolize_keys
3
4   in { # match function name
5     choices: [
6       {
7         delta: {
8           tool_calls: [
9             { index: index, function: {name: name} }
10          ]
11        }
12      }
13    ] }
14
15    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.

```

1  in { # match arguments
2    choices: [
3      {
4        delta: {
5          tool_calls: [
6            {
7              index: index, function: {arguments: argument }
8            }
9          ]
10         }
11       }
12     ]}
13
14     f_arguments[index] ||= "" # initialize if not already
15     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.

```

1  in {
2    choices: [
3      { delta: {content: new_content}, finish_reason: finish_reason }
4    ]}
5
6    # you could transmit every chunk to the user here...
7    buffer << new_content.to_s
8
9    if finish_reason.present?
10      finalize
11    elsif new_content.to_s.match?(/\n\n/)
12      send_to_client # ...or buffer and transmit once per paragraph
13    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.


```
1  in { error: { message: } }
2    if Rails.env.local?
3      raise message
4    else
5      Honeybadger.notify("AI Error: #{message}")
6      finalize
7    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.

```
1  else
2    Honeybadger.notify("Unrecognized Chunk: #{chunk}")
3  end
4  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.

```
1  def send_to_client
2    # no need to process pure whitespace
3    return if buffer.join.squish.blank?
4
5    # set the buffer content on the bot message
6    content = buffer.join
7    bot_message.content = content
8
9    # save to database so that we never lose data
10   # even if the stream doesn't terminate correctly
11   bot_message.update_column(:content, content)
12
13   # update content via websocket
14   ConversationRenderer.update(bot_message)
15 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.

```
1 def finalize
2   if f_name.any?
3     f_name.each_with_index do |name, index|
4       # takes care of calling the function wherever it's implemented
5       dispatch(name:, arguments: JSON.parse(f_arguments[index]))
6     end
7
8     # reset the function call history
9     f_name.clear
10    f_arguments.clear
11  else
12    content = buffer.join.presence
13    bot_message.update!(content:, complete: true)
14    ConversationRenderer.update(bot_message)
15  end
16 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.

```
1  # PromptSubscriber#dispatch
2
3  def dispatch(name:, arguments:)
4    # adds a function_call message to the conversation transcript
5    # plus dispatches to tool and returns result
6    conversation.function_call!(name, arguments).then do |result|
7      # add function result message to the transcript
8      conversation.function_result!(name, result)
9    end
10 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 `Prompt-Declarations` 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.

```
1 prompt text: -> { user_message.content },
2   stream: -> { ReplyStream.new(self) },
3   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:

```

1  LENGTH_STOPS = %w[length MAX_TOKENS]
2
3  def call(chunk, _bytesize)
4    case chunk.deep_symbolize_keys
5      # ...
6
7    in {
8      choices: [
9        { delta: {content: new_content},
10          finish_reason: finish_reason } ] }
11
12    buffer << new_content.to_s
13
14    if finish_reason.blank?
15      send_to_client if new_content.to_s.match?(/\n\n/)
16    elsif LENGTH_STOPS.include?(finish_reason)
17      continue_cutoff
18    else
19      finalize
20    end
21
22    # ...
23  end
24 end
25
26 private
27
28 def continue_cutoff
29   conversation.bot_message!(buffer.join, visible: false)
30   conversation.user_message!("please continue", visible: false)
31   bot_message.update_column(:created_at, Time.current)

```

32 **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 be 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 pattern 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.

```
1  class JsonFixer
2    include Raix::ChatCompletion
3
4    def call(bad_json, error_message)
5      raise "No data provided" if bad_json.blank? || error_message.blank?
6
7      transcript << {
8        system: "Consider user-provided JSON that generated a parse exception.
9                Do your best to fix it while preserving the original content
10               and intent as much as possible." }
11      transcript << { user: bad_json }
12      transcript << { assistant: "What is the error message?" }
13      transcript << { user: error_message }
14      transcript << { assistant: "Here is the corrected JSON\n```\njson\n" }
15
16      self.stop = ["```"]
17
18      chat_completion(json: true)
19    end
20
21    def model
22      "mistralai/mixtral-8x7b-instruct:nitro"
23    end
24  end
```



Note how `JsonFixer` uses the [Ventriloquist](#) pattern 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.

```
1 begin
2   JSON.parse(llm_generated_json)
3 rescue JSON::ParserError => e
4   JsonFixer.new.call(llm_generated_json, e.message)
5 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" pattern 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

The "Self-Healing Data" pattern, 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 the "Self-Healing Data" pattern, 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 demon-

strates 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:

```

1  class CustomerDataReviewer
2    include Raix::ChatCompletion
3    include Raix::FunctionDeclarations
4
5    attr_accessor :customer
6
7    function :flag_for_review, reason: { type: "string" } do |params|
8      AdminNotifier.review_request(customer, params[:reason])
9    end
10
11   def initialize(customer)
12     self.customer = customer
13   end
14
15   def call(customer_data)
16     transcript << {
17       system: "You are a customer data reviewer. Your task is to identify
18         and correct inconsistencies in customer data.
19
20         < additional instructions here... >
21
22         If you encounter severe errors that require human review, use the
23         `flag_for_review` tool to flag the data for manual intervention." }
24
25     transcript << { user: customer.to_json }
26     transcript << { assistant: "Reviewed/corrected data:\n```\n" }
27
28     self.stop = ["```"]
29
30     chat_completion(json: true).then do |result|
31       return if result.blank?
32
33       customer.update(result)
34     end
35   end
36 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](#) patterns 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 at-

tempts 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 re-

quests 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 has 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 has 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.

```
1 class OrderShippingAddressSubscriber
2   include Raix::ChatCompletion
3
4   attr_accessor :order
5
6   delegate :customer, to: :order
7
8   DIRECTIVE = "You are a smart order processing assistant. Given the
9   customer's order history, guess the most likely shipping address
10  for the current order."
11
12  def order_created(order)
13    return unless order.pending? && order.shipping_address.blank?
14
15    self.order = order
16
17    transcript.clear
18    transcript << { system: DIRECTIVE }
19    transcript << { user: "Order History: #{order_history.to_json}" }
20    transcript << { user: "Current Order: #{order.to_json}" }
21
22    response = chat_completion
23    apply_predicted_shipping_address(order, response)
24  end
25
26  private
27
28  def apply_predicted_shipping_address(order, response)
29    # extract the shipping address from the response...
30    # ...and assume there's some sort of live update of the address fields
31    order.update(shipping_address:)
32  end
33
34  def order_history
35    customer.orders.successful.limit(100).map do |order|
36      {
```

```
37     date: order.date,  
38     description: order.description,  
39     shipping_address: order.shipping_address  
40   }  
41   end  
42 end  
43 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:

```
1 Order History:  
2 [  
3   {"date": "2024-01-03", "description": "garden soil mix",  
4     "shipping_address": "123 Country Lane, Rural Town"},  
5   {"date": "2024-01-15", "description": "hardcover fiction novels",  
6     "shipping_address": "456 City Apt, Metroville"},  
7   {"date": "2024-01-22", "description": "baby diapers", "shipping_address":  
8     "789 Suburb St, Quietville"},  
9   {"date": "2024-02-01", "description": "organic vegetables",  
10    "shipping_address": "123 Country Lane, Rural Town"},  
11   {"date": "2024-02-17", "description": "mystery thriller book set",  
12    "shipping_address": "456 City Apt, Metroville"},  
13   {"date": "2024-02-25", "description": "baby wipes",  
14    "shipping_address": "789 Suburb St, Quietville"},  
15   {"date": "2024-03-05", "description": "flower seeds",  
16    "shipping_address": "123 Country Lane, Rural Town"},  
17   {"date": "2024-03-20", "description": "biographies",  
18    "shipping_address": "456 City Apt, Metroville"},  
19   {"date": "2024-03-30", "description": "baby formula",  
20    "shipping_address": "789 Suburb St, Quietville"},  
21   {"date": "2024-04-12", "description": "lawn fertilizer",  
22    "shipping_address": "123 Country Lane, Rural Town"},  
23   {"date": "2024-04-22", "description": "science fiction novels",  
24    "shipping_address": "456 City Apt, Metroville"},  
25   {"date": "2024-05-02", "description": "infant toys",  
26    "shipping_address": "789 Suburb St, Quietville"},  
27   {"date": "2024-05-14", "description": "outdoor grill",  
28    "shipping_address": "123 Country Lane, Rural Town"},  
29   {"date": "2024-05-29", "description": "literary classics",  
30    "shipping_address": "456 City Apt, Metroville"},  
31   {"date": "2024-06-11", "description": "baby clothes",  
32    "shipping_address": "789 Suburb St, Quietville"},  
33   {"date": "2024-07-01", "description": "watering can",
```

```
34     "shipping_address": "123 Country Lane, Rural Town"},
35     {"date": "2024-07-18", "description": "non-fiction essays",
36     "shipping_address": "456 City Apt, Metroville"},
37     {"date": "2024-07-28", "description": "baby bath items",
38     "shipping_address": "789 Suburb St, Quietville"},
39     {"date": "2024-08-09", "description": "herb garden kit",
40     "shipping_address": "123 Country Lane, Rural Town"},
41     {"date": "2024-08-24", "description": "children's books",
42     "shipping_address": "456 City Apt, Metroville"}
43 ]
```

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".

```
1  From the order history you've provided, it looks like the purchases are
2  generally clustered into three main types based on the shipping addresses:
3
4  123 Country Lane, Rural Town - This address often orders garden and
5  outdoor-related items like soil mix, vegetables, flower seeds, lawn
6  fertilizer, an outdoor grill, a watering can, and an herb garden kit.
7
8  456 City Apt, Metroville - This address tends to order various types of books,
9  indicating an interest in reading across genres like fiction, thrillers,
10 biographies, science fiction, literary classics, non-fiction essays, and
11 children's books.
12
13 789 Suburb St, Quietville - The purchases for this address are consistently
14 baby-related products, such as diapers, wipes, baby formula, infant toys,
15 baby clothes, and baby bath items.
16
17 Given these patterns, a digital thermometer could potentially be a fit for
18 any of the addresses based on different needs: outdoor temperature tracking,
19 a tool for a passionate reader interested in science gadgets, or a health
20 tool for a baby. However, considering its common use in monitoring health,
21 especially for babies, it's most likely to be purchased by someone who's
22 already buying baby care products.
23
24 Therefore, the most likely shipping address for a digital thermometer, based
25 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.

```
1 Based on the given order history, it appears that the customer has received
2 orders for baby-related items (diapers, wipes, formula, clothes, bath items)
3 at the address "789 Suburb St, Quietville" and garden-related items (soil,
4 seeds, watering can, herb garden kit) at the address "123 Country Lane, Rural
5 Town". The customer has also received books (fiction, non-fiction, literary
6 classics, mystery-thriller, science fiction, biographies, children's books)
7 at the address "456 City Apt, Metroville".
8
9 Given that the new order is for a "thermometer", which doesn't seem to fit
10 into any of the categories that the customer has previously ordered, it's
11 hard to accurately predict the shipping address based solely on the order
12 history. However, given the 50%-50% split between baby-related and
13 garden-related items, it could somewhat lean towards the Baby-related items
14 address ("789 Suburb St, Quietville"). But remember, this is an assumption
15 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.

Create your account

Full name

Obie Fernandez

Email

obiefernandez@gmail.com



Did you mean obiefernandez@gmail.com? [Yes, update.](#)

Country ⓘ

 United States



Password

.....



✓ Nice work. This is an excellent password.

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 raises 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.

```
1 class TaskRouter
2   include Raix::ChatCompletion
3   include Raix::FunctionDispatch
4
5   attr_accessor :task
6
7   # list of functions that can be called by the AI entirely at its
8   # discretion depending on the task received
9
10  function :analyze_task_priority do
11    TaskPriorityAnalyzer.perform(task)
12  end
13
14  function :check_resource_availability, # ...
15  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 component responsible for deciding how to handle the exception.
4. **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.

5. **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.
6. **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:

```
1 class InventoryManager
2   def check_availability(order)
3     begin
4       # Perform inventory check logic
5       inventory = Inventory.find_by(product_id: order.product_id)
6       if inventory.available_quantity >= order.quantity
7         return true
8       else
9         raise InsufficientInventoryError, "Insufficient inventory for product #{order.product_
10 id}"
11     end
12   rescue InsufficientInventoryError => e
13     # Log the exception
14     logger.error("Inventory check failed: #{e.message}")
15
16     # Retry the operation after a delay
17     retry_count ||= 0
18     if retry_count < MAX_RETRIES
19       retry_count += 1
```

```
20         sleep(RETRY_DELAY)
21         retry
22     else
23         # Fallback to manual intervention
24         NotificationService.notify_admin("Inventory check failed for order #{order.id}")
25         return false
26     end
27 end
28 end
29 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](#) pattern that we first encountered in Chapter 2. 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.

```

1  class OrderProcessor
2    include Raix::ChatCompletion
3    include Raix::FunctionDispatch
4
5    SYSTEM_DIRECTIVE = "You are an order processor, tasked with..."
6
7    def initialize(order)
8      self.order = order
9      transcript << { system: SYSTEM_DIRECTIVE }
10     transcript << { user: order.to_json }
11   end
12
13   def perform
14     # will continue looping until `stop_looping!` is called
15     chat_completion(loop: true)
16   end
17
18   # list of functions available to be called by the AI
19   # truncated for brevity
20
21   def functions
22     [
23       {
24         name: "validate_order",
25         description: "Invoke to check validity of order",
26         parameters: {
27           ...
28         },
29         ...
30     ]
31   end
32
33   # implementation of functions that can be called by the AI
34   # entirely at its discretion, depending on the needs of the order
35

```

```
36  def validate_order
37    OrderValidationWorker.perform(@order)
38  end
39
40  def check_inventory
41    InventoryCheckWorker.perform(@order)
42  end
43
44  def process_payment
45    PaymentProcessingWorker.perform(@order)
46  end
47
48  def schedule_shipping
49    ShippingSchedulerWorker.perform(@order)
50  end
51
52  def send_confirmation
53    OrderConfirmationWorker.perform(@order)
54  end
55
56  def finished_processing
57    @order.update!(transcript:, processed_at: Time.current)
58    stop_looping!
59  end
60 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`:

```
1  class ContentModerator
2      include Raix::ChatCompletion
3      include Raix::FunctionDispatch
4
5      SYSTEM_DIRECTIVE = "You are a content moderator process manager,
6          tasked with the workflow involved in moderating user-generated content..."
7
8      def initialize(content)
9          @content = content
10         @transcript = [
11             { system: SYSTEM_DIRECTIVE },
12             { user: content.to_json }
13         ]
14     end
15
16     def perform
17         complete(@transcript)
18     end
19
20     def model
21         "openai/gpt-4"
22     end
23
24     # list of functions available to be called by the AI
25     # truncated for brevity
26
27     def functions
28         [
29             {
30                 name: "analyze_text",
31                 # ...
32             },
33             {
34                 name: "recognize_image",
35                 description: "Invoke to describe images...",
36                 # ...
37             },
38             {
39                 name: "assess_user_reputation",
40                 # ...
41             },
42             {
43                 name: "escalate_to_manual_review",
44                 # ...
45             },
46             {
```



```
47     name: "approve_content",
48     # ...
49   },
50   {
51     name: "reject_content",
52     # ...
53   }
54 ]
55 end
56
57 # implementation of functions that can be called by the AI
58 # entirely at its discretion, depending on the needs of the order
59
60 def analyze_text
61   result = TextAnalysisWorker.perform(@content)
62   continue_with(result)
63 end
64
65 def recognize_image
66   result = ImageRecognitionWorker.perform(@content)
67   continue_with(result)
68 end
69
70 def assess_user_reputation
71   result = UserReputationWorker.perform(@content.user)
72   continue_with(result)
73 end
74
75 def escalate_to_manual_review
76   ManualReviewWorker.perform(@content)
77   @content.update!(status: 'pending', transcript: @transcript)
78 end
79
80 def approve_content
81   @content.update!(status: 'approved', transcript: @transcript)
82 end
83
84 def reject_content
85   @content.update!(status: 'rejected', transcript: @transcript)
86 end
87
88 private
89
90 def continue_with(result)
91   @transcript << { function: result }
92   complete(@transcript)
```

```
93     end
94 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 characteris-

tics 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.

```
1  class SupportTicketScheduler
2    include Raix::ChatCompletion
3    include Raix::FunctionDispatch
4
5    SYSTEM_DIRECTIVE = "You are a support ticket scheduler,
6      tasked with intelligently assigning tickets to available agents..."
7
8    def initialize(ticket)
9      @ticket = ticket
10     @transcript = [
11       { system: SYSTEM_DIRECTIVE },
12       { user: ticket.to_json }
13     ]
14   end
15
16   def perform
17     complete(@transcript)
18   end
19
20   def model
21     "openai/gpt-4"
22   end
23
24   def functions
```

```
25     [
26         {
27             name: "analyze_ticket_urgency",
28             # ...
29         },
30         {
31             name: "list_available_agents",
32             description: "Includes expertise of available agents",
33             # ...
34         },
35         {
36             name: "predict_agent_workload",
37             description: "Uses historical data to predict upcoming workloads",
38             # ...
39         },
40         {
41             name: "assign_ticket_to_agent",
42             # ...
43         },
44         {
45             name: "reschedule_ticket",
46             # ...
47         }
48     ]
49 end
50
51 # implementation of functions that can be called by the AI
52 # entirely at its discretion, depending on the needs of the order
53
54 def analyze_ticket_urgency
55     result = TicketUrgencyAnalyzer.perform(@ticket)
56     continue_with(result)
57 end
58
59 def list_available_agents
60     result = ListAvailableAgents.perform
61     continue_with(result)
62 end
63
64 def predict_agent_workload
65     result = AgentWorkloadPredictor.perform
66     continue_with(result)
67 end
68
69 def assign_ticket_to_agent
70     TicketAssigner.perform(@ticket, @transcript)
```

```
71     end
72
73     def delay_assignment(until)
74         until = DateTimeStandardizer.process(until)
75         SupportTicketScheduler.delay(@ticket, @transcript, until)
76     end
77
78     private
79
80     def continue_with(result)
81         @transcript << { function: result }
82         complete(@transcript)
83     end
84 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.

```
1  class DataProcessingOrchestrator
2    include Raix::ChatCompletion
3    include Raix::FunctionDispatch
4
5    SYSTEM_DIRECTIVE = "You are a data processing orchestrator..."
6
7    def initialize(data_batch)
8      @data_batch = data_batch
9      @transcript = [
10        { system: SYSTEM_DIRECTIVE },
11        { user: data_batch.to_json }
12      ]
13    end
14
15    def perform
16      complete(@transcript)
17    end
18
19    def model
20      "openai/gpt-4"
21    end
22
23    def functions
24      [
25        {
26          name: "validate_data",
27          # ...
28        },
29        {
30          name: "process_data",
31          # ...
32        },
33        {
34          name: "request_fix",
35          # ...
36        },
37        {
```

```
38         name: "retry_processing",
39         # ...
40     },
41     {
42         name: "mark_data_as_failed",
43         # ...
44     },
45     {
46         name: "finished",
47         # ...
48     }
49 ]
50 end
51
52 # implementation of functions that can be called by the AI
53 # entirely at its discretion, depending on the needs of the order
54
55 def validate_data
56     result = DataValidator.perform(@data_batch)
57     continue_with(result)
58 rescue ValidationException => e
59     handle_validation_exception(e)
60 end
61
62 def process_data
63     result = DataProcessor.perform(@data_batch)
64     continue_with(result)
65 rescue ProcessingException => e
66     handle_processing_exception(e)
67 end
68
69 def request_fix(description_of_fix)
70     result = SmartDataFixer.new(description_of_fix, @data_batch)
71     continue_with(result)
72 end
73
74 def retry_processing(timeout_in_seconds)
75     wait(timeout_in_seconds)
76     process_data
77 end
78
79 def mark_data_as_failed
80     @data_batch.update!(status: 'failed', transcript: @transcript)
81 end
82
83 def finished
```

```

84     @data_batch.update!(status: 'finished', transcript: @transcript)
85   end
86
87   private
88
89   def continue_with(result)
90     @transcript << { function: result }
91     complete(@transcript)
92   end
93
94   def handle_validation_exception(exception)
95     @transcript << { exception: exception.message }
96     complete(@transcript)
97   end
98
99   def handle_processing_exception(exception)
100    @transcript << { exception: exception.message }
101    complete(@transcript)
102  end
103 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:

```
1 class WorkflowLogger
2   def self.log(message, severity = :info)
3     @logger ||= ActiveSupport::Logger.new('workflow.log')
4     @logger.formatter ||= proc do |severity, datetime, progname, msg|
5       "#{datetime} [#{severity}] #{msg}\n"
6     end
7     @logger.send(severity, message)
8   end
9 end
10
11 # Usage example
12 WorkflowLogger.log("Workflow initiated for order #{@order.id}")
13 WorkflowLogger.log("Payment processing completed successfully")
14 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 might 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.

¹Host 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 work-

flow 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:

```
1 RSpec.describe OrderValidator do
2   describe '#validate' do
3     context 'when order is valid' do
4       let(:order) { build(:order) }
5
6       it 'returns true' do
7         expect(subject.validate(order)).to be true
8       end
9     end
10
11    context 'when order is invalid' do
12      let(:order) { build(:order, total_amount: -100) }
13
14      it 'returns false' do
15        expect(subject.validate(order)).to be false
16      end
17    end
18  end
19 end
```

```
18   end
19 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:

```
1 RSpec.describe OrderProcessingWorkflow do
2   describe '#process' do
3     let(:order) { build(:order) }
4
5     it 'processes the order successfully' do
6       expect(OrderValidator).to receive(:validate).with(order).and_return(true)
7       expect(InventoryManager).to receive(:check_availability).with(order).and_return(true)
8       expect(PaymentProcessor).to receive(:process_payment).with(order).and_return(true)
9       expect(ShippingService).to receive(:schedule_shipping).with(order).and_return(true)
10
11       workflow = OrderProcessingWorkflow.new(order)
12       result = workflow.process
13
14       expect(result).to be true
15       expect(order.status).to eq('processed')
16     end
17   end
18 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:

```
1 RSpec.describe FraudDetector do
2   describe '#detect_fraud' do
3     context 'when transaction is fraudulent' do
4       let(:transaction) { build(:transaction, amount: 10000, location: 'High-Risk Country') }
5
6       it 'returns true' do
7         expect(subject.detect_fraud(transaction)).to be true
8       end
9     end
10
11    context 'when transaction is legitimate' do
12      let(:transaction) { build(:transaction, amount: 100, location: 'Low-Risk Country') }
13
14      it 'returns false' do
15        expect(subject.detect_fraud(transaction)).to be false
16      end
17    end
18  end
19 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:

```
1 RSpec.describe 'Order Processing Workflow' do
2   scenario 'User places an order successfully' do
3     visit '/orders/new'
4     fill_in 'Product', with: 'Sample Product'
5     fill_in 'Quantity', with: '2'
6     fill_in 'Shipping Address', with: '123 Main St'
7     click_button 'Place Order'
8
9     expect(page).to have_content('Order Placed Successfully')
10    expect(Order.count).to eq(1)
11    expect(Order.last.status).to eq('processed')
12  end
13 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

In the first part of the book, we explored the fundamental approaches and techniques for integrating AI into application development. We considered architectural approaches and also broad categories of patterns such as Narrowing the Path, Multitude of Workers, Self-Healing Data, Contextual Content Generation, and Intelligent Workflow Orchestration. My hope is that I provided a solid foundation for understanding the potential and challenges of leveraging AI in software development.

Now, in Part 2, we dive deeper into the practical implementation of these concepts by presenting a comprehensive collection of design patterns specifically tailored for AI-driven application development. These patterns encapsulate proven solutions to common problems and challenges I encountered when building applications that harness the power of AI. Where possible, I evoke classic design patterns from legendary authors like the “Gang of Four”² and my old colleagues at Thoughtworks: Martin Fowler³ and Gregor Hohpe⁴.

Each pattern follows a structured format, providing a clear problem statement, a detailed solution description, and practical considerations for implementation. While the code examples are presented in Ruby, the underlying principles and concepts are applicable across different technology stacks. Feel free to translate the patterns into your preferred programming language and integrate them into your existing development practices.

One final note, keep in mind that the patterns are not meant to be prescriptive solutions but rather inspiration for your own creativity. Use them as a foundation to build upon, adapt them to your specific requirements, and combine

²[Design Patterns: Elements of Reusable Object-Oriented Software](#)

³[Patterns of Enterprise Application Architecture](#)

⁴[Enterprise Integration Patterns: Designing, Building, and Deploying Messaging Solutions](#)

them in creative ways to unlock new possibilities.

Prompt Engineering



Crafting effective prompts is a critical skill in AI-driven application development. Prompt engineering patterns provide proven techniques to optimize prompts, improve AI performance, and achieve desired outcomes. In this chapter, we'll explore a range of prompt engineering patterns, starting with Chain of Thought.

Chain of Thought

The Chain of Thought (CoT) pattern is a powerful and versatile technique in prompt engineering that significantly enhances performance and accuracy. By guiding the AI through a series of intermediate steps or reasoning processes, Chain of Thought enables more complex problem-solving and reduces the likelihood of errors or hallucinations.

Chain of Thought is particularly effective in scenarios such as:

- Content generation (e.g., summaries, articles)
- Creating structured entities (e.g., JSON, YAML outputs)
- Guiding LLM agents through complex tasks
- Improving Retrieval Augmented Generation (RAG) processes
- Extracting multiple facets or attributes from text

Chain of Thought is similar to Prompt Chaining, which spans multiple prompt/response cycles instead of trying to cram the “intermediate processing” into a single chat response.

How It Works

Chain of Thought leverages the LLM’s ability to generate coherent sequences of thoughts. It works by:

1. **Breaking down complex tasks:** Instead of asking for a direct answer, the prompt guides the LLM to approach the problem step-by-step.
2. **Generating intermediate outputs:** The LLM is instructed to produce interim results or reasoning steps before arriving at the final answer.
3. **Reinforcing semantic context:** Each generated bit of content (should) strengthen the relevancy of the entire context, improving the accuracy of subsequent content generation.

4. **Exploiting the inference loop:** As the LLM generates each token, it considers all previously generated tokens, including those from intermediate steps.



To fully understand Chain of Thought, you need to realize that you can throw away almost all of the output generated from a particular prompt. Just keep the “good stuff” at the end of the response.

There are two primary ways to implement Chain of Thought:

1. **Explicit:** Provide examples of the desired reasoning process directly in the prompt. This guides the LLM to follow a similar thought pattern.
2. **Internal:** Instruct the LLM to generate its own intermediate steps or “reasoning” before producing the final output.

We’ll consider both in the examples.

Examples

Let’s explore how Chain of Thought can be applied in different contexts:

Content Generation

When generating a summary, instead of directly requesting the final output, use a two-step process:

```

1  class DocumentSummarizer
2    include Raix::ChatCompletion
3
4    def call(document)
5      transcript << { system: "You are an expert at summarizing documents." }
6      transcript << { user: <<~PROMPT }
7        I will give you the content of a document. You have two tasks:
8        1. Write a detailed table of contents, including all levels of headings.
9        2. Write a comprehensive summary using the table of contents as a guide.
10
11      # Begin Document
12      #{document}
13      # End Document
14      PROMPT
15
16      chat_completion
17    end
18  end

```

This “internal” approach ensures a more structured and comprehensive summary by first extracting the document’s structure.

Structured Entity Creation

For generating structured data, provide examples of the desired format:

```

1  class LessonExtractor
2    include Raix::ChatCompletion
3
4    def call(document)
5      transcript << { system: "You're a teacher that creates high quality lessons." }
6      transcript << { user: <<~PROMPT }
7        From the following document, extract the key messages and expected changes.
8        Use the following YAML format to respond:
9
10      granules:
11        - keyMessage: "a key message"
12          desiredChange: "expected change when the learner understands the key message"
13        - keyMessage: "another key message"
14          desiredChange: "expected change when the learner understands the key message"
15
16      # Begin Document
17      #{document}
18      # End Document

```

```

19     PROMPT
20
21     chat_completion
22 end
23 end

```

This “explicit” method guides the LLM to produce structured output while maintaining the context of the input document.

LLM Agent Guidance

For complex tasks involving LLM agents, instruct the model to create a plan before taking action:

```

1  class MeetingScheduler
2    include Raix::ChatCompletion
3    include Raix::FunctionDispatch
4
5    function :list_meetings
6
7    function :book_meeting,
8      participants: { type: "array", ... },
9      time: { type: "string" }
10
11  def call(participants)
12    transcript << {
13      system: "You are a semi-autonomous Agent capable of using tools
14        to accomplish tasks."
15    }
16
17    prompt = <<~PROMPT
18      Your task is to reserve a meeting slot for the following people:
19      #{participants.join("\n")}
20
21      Start by making a detailed plan of the actions you will take to solve this task.
22      Then, execute your plan by calling the necessary tools.
23
24      Respond with the tools you want to use in a YAML block, for example:
25
26      tools:
27        - name: listMeetings
28          arguments: ["Aschen"]
29

```



```
30     PROMPT
31
32     transcript << { user: prompt}
33
34     chat_completion(function_call: "auto")
35 end
36 end
```

This approach helps the LLM agent to think through the process before taking actions, leading to more efficient and accurate task completion.

Benefits and Considerations

The Chain of Thought pattern offers several advantages:

1. **Improved accuracy:** By breaking down complex tasks, the LLM is less likely to make errors or hallucinate information.
2. **Enhanced problem-solving:** The step-by-step approach allows for tackling more complex issues.
3. **Increased transparency:** The intermediate steps provide insight into the LLM's reasoning process.
4. **Flexibility:** CoT can be adapted to various tasks and domains.

However, there are some considerations:

1. **Increased token usage:** The additional steps consume more tokens, potentially increasing costs.
2. **Longer processing time:** Generating intermediate steps can increase the overall response time.
3. **Complexity:** Creating effective CoT prompts may require more careful crafting and testing, and may be more difficult to maintain.

Mode Switch

The Mode Switch pattern is a powerful technique that allows you to dynamically alter the capabilities and behavior of an AI system during runtime. This pattern is particularly useful when you want to provide the AI with additional functionalities or context that are not needed all the time, thus optimizing token usage and maintaining a cleaner, more focused context when those capabilities are not required.

How It Works

The Mode Switch pattern typically involves the following components:

1. **Mode Indicator:** A state variable that tracks whether a particular mode is active or not.
2. **Mode Activation Function:** A function that can be called to toggle a specific mode on or off.
3. **Mode-Specific Functions:** Additional functions or capabilities that become available when a particular mode is activated.
4. **Context Management:** Logic to add or remove mode-specific instructions and functions from the AI's context based on the current mode.

Here's a typical flow for implementing a Mode Switch:

1. The AI system starts in a default mode with a base set of capabilities.
2. When a specific functionality is needed, a mode switch function is called.
3. The mode switch function updates the mode indicator and adds relevant instructions and functions to the AI's context.
4. The AI can now use the new capabilities until the mode is switched off or changed.

Here's a real-life example of a mode activation function extracted from Olympia.

```
1 {  
2   name: "image_mode",  
3   description: "Call this function first to toggle image mode if user wants  
4                 to generate a photo or image. Olympia will provide you with  
5                 additional tool functions related to text_to_image generation.",  
6   parameters: {}  
7 }
```

In this case, `image_mode` is a mode switch function. When called, it does the following:

1. Changes the state of the server to track that image mode is active.
2. Adds instructions to the AI's context describing full image generation capabilities.
3. Makes additional functions available to the AI, such as text-to-image generation tools.

When to Use It

The Mode Switch pattern is particularly useful in scenarios where:

1. **Resource Optimization:** You want to conserve tokens and processing power by only loading specific capabilities when they're needed.
2. **Context Management:** You need to keep the AI's primary context clean and focused, only adding specialized instructions when necessary.
3. **Dynamic Functionality:** Your application supports switching between different sets of capabilities based on user needs or system states.
4. **Specialized Modes:** You have distinct operational modes (e.g., image generation, code analysis, data processing) that require different sets of instructions and tools.

Example

Here's a simple example of how you might implement a Mode Switch in a Ruby class:

```
1  class AIAssistant
2    include Raix::ChatCompletion
3    include Raix::FunctionDispatch
4
5    function :image_mode,
6      description: "Enable image generation tool functions"
7
8    function :generate_image,
9      description: "Generate an image based on text prompt",
10     if: -> { image_mode? }
11
12  def initialize
13    directive = "You are a helpful AI assistant."
14
15    if image_mode?
16      directive += " You can generate images based on text descriptions."
17    end
18
19    transcript << { system: directive }
20  end
21
22  def image_mode(args)
23    enable_image_mode! # implementation omitted
24    "Image mode enabled"
25  end
26
27  def generate_image(args)
28    raise "Image mode is not active" unless image_mode?
29
30    # Image generation logic here
31  end
32 end
```

In this example, the `image_mode` function acts as a mode switch. When activated, it updates the AI's context with additional instructions about image generation capabilities. The `generate_image` function also becomes available for use, but only when image mode is active (implemented with `if` condition on the function declaration.)

By using the Mode Switch pattern, you can create more flexible and efficient AI systems that can adapt their capabilities based on the current needs of the user or the application. This leads to more focused and relevant interactions,

optimized resource usage, and a cleaner overall design for your AI-powered applications.

Role Assignment

One of my favorite prompt engineering patterns is also one of the most popular and fundamental ones, that practically everyone learns right at the start of their experimentation with LLMs: assigning a specific role or responsibility. Doing so provides the AI with context that steers its responses in terms of content, tone, style, etc. By defining a clear role for the AI, you can effectively guide its responses to align with your desired outcome.

How It Works

The Role Assignment pattern involves the following steps:

1. Identify the desired role or responsibility for the AI, such as a domain expert, a specific persona, or a task-oriented assistant.
2. Craft a prompt that clearly defines the assigned role and provides any necessary context or instructions.
3. Incorporate the role assignment into the conversation transcript or system directive.
4. Initiate the chat completion process, allowing the AI to generate responses based on the assigned role.
5. The AI, guided by the assigned role, generates responses that align with the expected content, tone, style, and behavior.

By assigning a specific role to the AI, you establish a framework that influences its responses. The AI, being a language model trained to understand and follow instructions, will strive to generate responses that are consistent with the assigned role and meet the expectations set by the prompt.

When to Use It

The Role Assignment pattern is particularly useful in scenarios where:

- You want to guide the AI's responses to match a specific domain or area of expertise.
- You need the AI to adopt a particular persona or communication style.
- You want to ensure consistency in the AI's behavior and responses across different conversations or tasks.
- You need the AI to focus on a specific task or responsibility within the context of the application.

Examples

Consider the prompts below: Because the assigned roles vary, we can expect very different responses. The teacher will respond with simple language and analogies while the NLP professor is very likely to dive into nitty-gritty technical details of attention mechanisms.

```
1 # Prompt 1
2 You are a teacher giving lessons to a curious 8-year-old child.
3 Explain how attention in LLMs works.
4
5 # Prompt 2
6 You are an NLP professor giving a lecture to CompSci postgrads.
7 Explain how attention in LLMs works.
```

Roles and responsibilities can also improve accuracy tasks that go beyond just content generation. Imagine we're building a system to detect potential misinformation in news articles. While a basic prompt like prompt 1 might work, we can improve the model's accuracy by providing it with a role (prompt 2) or responsibility (prompt 3). The additional context in prompts 2 and 3 encourages the LLM to scrutinize the input more carefully, thus increasing recall on more subtle issues.

```
1 # Prompt 1
2 Does this news article contain potential misinformation?
3
4 # Prompt 2
5 You are an expert fact-checker who identifies potential misinformation in
6 news articles. Does the following news article contain potential
7 misinformation?
8
9 # Prompt 3
10 You are responsible for identifying potential misinformation in news articles.
11 Does this news article contain potential misinformation?
```


Prompt Object

As your applications grow in complexity and scale, managing prompts can become a daunting task. This is where the Prompt Object pattern comes into play. It involves representing prompts as database-backed records, treating them as first-class citizens in your application's architecture. By encapsulating prompts as objects with associated attributes, metadata, and behaviors, you can elevate your prompt management to a whole new level.

How It Works

Instead of scattering prompts throughout your codebase or storing them in disparate files, you can consolidate them into a single, easily accessible location as database-backed records. This centralization simplifies prompt retrieval, updating, and deletion, making it easier to maintain and evolve your application's prompts over time. You can categorize prompts based on their purpose, domain, or any other relevant criteria, enabling quick and efficient access to the right prompt for the right situation.

The Prompt Object pattern enables you to associate specific parameters with each prompt. These parameters can include settings such as the model to be used, maximum token limit, temperature, or any other configuration options supported by the LLM. By encapsulating these parameters within the Prompt Object, you ensure consistent usage and control over the LLM's behavior across your application. Different prompts may require different parameter settings based on their intended purpose or desired output, and the Prompt Object pattern allows you to customize these settings on a per-prompt basis.

The ability to store extensible metadata is a powerful aspect of the Prompt Object pattern. By attaching relevant metadata attributes to each prompt, such as name, description, author, creation date, or tags, you can enrich your prompts with additional context and information. This metadata serves as a

valuable tool for searching, filtering, and organizing prompts based on specific criteria. For example, you can easily retrieve all prompts created by a particular author, filter prompts by their domain or purpose, or sort prompts based on their creation date. Metadata adds a layer of discoverability and organization to your prompt management system.

One of the most important advantages of the Prompt Object pattern, given the importance of constant iteration and testing of prompts, is the ability to implement versioning schemes. As your application grows and evolves, so do your prompts. By storing each version of a prompt separately, you can track changes, experiment with different variations, and revert to previous versions if needed. Versioning allows you to iterate on prompts, test their effectiveness, and continuously improve their quality without the risk of losing valuable iterations along the way.

Tracking the usage history and analytics of prompts is another valuable aspect of the Prompt Object pattern. By recording how frequently each prompt is used, by whom, and in what contexts, you can gain valuable insights into the effectiveness and popularity of your prompts. This information can help you make data-driven decisions about which prompts to optimize, which ones to retire, and which ones to promote further.

Audit logging and compliance are critical considerations for many applications, especially in sensitive domains such as healthcare, finance, or legal, where accountability and transparency are paramount. The Prompt Object pattern enables you to implement comprehensive audit logging for all activities related to prompts. By recording who created, modified, or deleted a prompt, along with timestamps and other relevant details, you establish a clear audit trail. Audit logging helps you maintain the integrity of your prompts and ensures that you can track and investigate any issues or anomalies that may arise.

Collaboration and knowledge sharing are essential aspects of effective prompt management, especially in team environments. The Prompt Object pattern

promotes collaboration by enabling team members to share, reuse, and build upon each other's prompts. By storing prompts as structured records in a centralized repository, developers and content creators can easily discover, modify, and leverage existing prompts. This collaborative approach fosters consistency, reduces duplication of efforts, and facilitates the exchange of ideas and best practices within the team.

Finally, the Prompt Object pattern opens up possibilities for integration with other systems and tools. By representing prompts as structured records, you can seamlessly export them to content management systems, incorporate them into automated workflows, or integrate them with analytics platforms. The standardized interface provided by the Prompt Object pattern allows for easy interoperability and data exchange between different systems, enabling you to leverage prompts across various aspects of your application's ecosystem.

Prompt Template

Crafting effective prompts for AI-powered applications can be a challenging task, especially when dealing with complex or dynamic requirements. Just like creating dynamic web pages, composing prompts often involves a mix of static content and dynamic elements that need to be seamlessly integrated. The “Prompt Template” pattern addresses this challenge by providing a structured approach to prompt composition, allowing developers to create reusable and maintainable prompt templates.

How It Works

The core idea behind the Prompt Template pattern is to separate the static and dynamic parts of a prompt. The static content serves as a template, providing the overall structure and context of the prompt. Within this template, you can embed markers or placeholders that represent dynamic elements. These markers are later resolved and replaced with actual values or computations when the prompt is used to interact with the AI component. This can be done using simple string substitution or more advanced techniques like regular expressions or template rendering libraries.

In some cases, the prompt template may require conditional or iterative behavior based on the input data. For example, a prompt template for generating a summary report might include a section that is only displayed if certain conditions are met, or a section that is repeated for each item in a list. To handle such scenarios, use a string template engine such as Ruby’s ERB.



Remember that you don’t have to limit yourself to plain-text to language models. There are plenty of places in my applications where I liberally mix formats, even within the same prompt: plain text, JSON, YAML, CSV, XML... the AI takes it all in stride.

Benefits and Considerations

The “Prompt Template” pattern offers several benefits for prompt engineering in AI-powered applications:

1. **Reusability:** Prompt templates can be reused across multiple scenarios or user interactions, promoting consistency and reducing duplication of effort.
2. **Maintainability:** By separating the static and dynamic parts of a prompt, developers can easily update and modify the template without affecting the underlying logic.
3. **Collaboration:** Prompt templates can be created and edited by non-technical team members, such as content writers or domain experts, while developers focus on the implementation details.
4. **Testability:** Prompt templates can be tested independently of the AI component, allowing for easier validation and debugging of the generated prompts.

However, there are also some considerations to keep in mind when applying the “Prompt Template” pattern:

1. **Complexity:** As the complexity of the prompt templates increases, managing the markers and their resolution logic can become more challenging. It's important to strike a balance between flexibility and simplicity.
2. **Performance:** Resolving markers and generating the final prompt dynamically may have an impact on performance, especially if the templates are complex or the input data is large. Careful design and optimization techniques should be employed to minimize any performance overhead.
3. **Error Handling:** Proper error handling mechanisms should be in place to handle scenarios where markers are missing or input data is invalid. Graceful degradation and informative error messages can help improve the robustness of the system.

When to Use It:

The Prompt Template pattern is particularly useful in the following scenarios:

1. **Dynamic Prompts:** When the prompts for an AI-powered application need to be generated dynamically based on user input, context, or other variables.
2. **Complex Prompts:** When the prompts involve a mix of static and dynamic content, conditional logic, or iterative sections.
3. **Reusable Prompts:** When similar prompts are used across multiple parts of the application or for different user interactions.
4. **Collaboration:** When non-technical team members, such as content writers or domain experts, need to contribute to the creation and maintenance of prompts.

Example

The Prompt Template pattern is particularly useful when generating personalized content based on dynamic data. As our example, let's consider a scenario where we need to generate customer service reply emails tailored to each customer's profile, order history, and the specific concerns raised in their email. By using a prompt template, we can create a reusable structure that incorporates these dynamic elements, allowing the LLM to generate highly relevant and personalized responses with minimal hallucinations.

Here's what the template might look like implemented in ERB, which is Ruby's native templating engine (and very similar to templating syntaxes in other languages).

```
1 # templates/customer_service/auto_reply.erb
2
3 You are a customer service representative tasked with responding to
4 customer inquiries.
5
6 <profile>
7   <%= @customer.profile %>
8 </profile>
9
10 <history last=10>
11   <%= @customer.order_history.latest(10).each do |order| %>
12     - Order ID: <%= order.id %>
13       Date: <%= order.date %>
14       Total: <%= order.total %>
15       Items: ...excerpted for brevity
16   <% end %>
17 </history>
18
19 <email>
20   <%= @email_contents %>
21 </email>
22
23 <instructions>
24   Please generate a personalized reply to the customer's email within a
25   <reply> tag, addressing their concerns and providing relevant information
26   based on their profile and order history. The reply should be empathetic,
27   professional, and offer a solution or next steps.
28 </instructions>
```

Note how we use [Structured IO](#) to clearly delineate the input and output of the prompt. And now the class that uses the template to generate the email reply.

```
1  require 'erb'
2
3  class CustomerServiceReplyGenerator
4    include Raix::ChatCompletion
5    include Raix::FunctionDispatch
6
7    TEMPLATE = File.read("templates/customer_service/auto_reply.erb")
8
9    attr_accessor :customer
10   attr_accessor :email_contents
11
12   function :check_order_status, id: { type: "string" }
13
14   def initialize(customer, email_contents)
15     self.customer = customer
16     self.email_contents
17     transcript << { user: prompt_template }
18   end
19
20   def call
21     response = chat_completion(loop: true)
22     extract_reply(response)
23   end
24
25   private
26
27   def prompt_template
28     # the binding gives the template access to the attributes
29     # of this object as instance variables
30     ERB.new(TEMPLATE).result(binding)
31   end
32
33   def extract_reply(response)
34     # ...
35   end
36 end
```

Testing this out on the command line might look like this:


```
1  >> email = <<~EMAIL
2    Subject: Order Delayed
3
4    Dear Customer Service,
5
6    I placed an order on June 10th, but I haven't received any updates on the
7    shipping status. Can you please provide me with information on when I can
8    expect my order to arrive?
9
10   Thank you,
11   John Doe
12 EMAIL
13
14 >> CustomerServiceReplyGenerator.new(customer, email).call
15 => "Here's the email reply requested:
16   <reply>
17     Dear John Doe,
18
19     Thank you for reaching out regarding your order ORD1234 placed on June 10th.
20     We apologize for the delay in providing you with an update on your order status.
21
22     Upon checking your order status, we can confirm that your order is currently
23     being processed at our warehouse. We experienced a slight delay due to high
24     order volumes, but rest assured that our team is working diligently to ship
25     your order as soon as possible.
26
27     As a valued Gold tier customer, we understand the importance of timely delivery.
28     We anticipate that your order will be shipped within the next 2-3 business days,
29     and you should receive a shipping confirmation email with tracking information
30     once it leaves our facility.
31
32     If you have any further questions or concerns, please don't hesitate to reach
33     out to us. We appreciate your patience and understanding.
34
35     Best regards,
36     Customer Service Team
37   </reply>"
```

This example demonstrates how the Prompt Template pattern can be used to create dynamic and personalized prompts for generating customer service replies. By leveraging ERB templating, we can easily incorporate customer-specific information into the prompt, allowing the LLM to generate tailored responses that address the customer's concerns and provide relevant informa-

tion based on their profile and order history.

The example is purposely simplified for brevity. In particular, it lacks guidance on guidelines for how to respond to different situations. We have provided a function that gives the AI the ability to check the order status, but what if the customer was asking some other type of question? Without context-specific instructions, this particular prompt would make up the reply in unpredictable ways.

In theory we could add another function that let the AI search a policies and procedure knowledge store at its discretion. However, in my experience a better approach would be to use [Prompt Chaining](#). Let's force a lookup of relevant policies and procedures based on the contents of the email *before* generating a response.

```

1  class CustomerServiceReplyGenerator
2    include Raix::ChatCompletion
3    include Raix::FunctionDispatch
4    include Raix::PromptDeclarations
5
6    TEMPLATE = File.read("templates/customer_service/auto_reply.erb")
7
8    attr_accessor :customer
9    attr_accessor :email_contents
10   attr_accessor :specific_instructions
11
12   prompt call: RelevantPoliciesQuery,
13           success: -> (response) { self.specific_instructions = response }
14
15   prompt text: -> { prompt_template },
16           success: :extract_reply
17
18   function :check_order_status, id: { type: "string" }
19
20   def initialize(customer, email_contents)
21     self.customer = customer
22     self.email_contents
23   end
24
25   def call
26     chat_completion
27   end

```

```

28
29   # ...
30
31 end

```

In the code above, note that we've modified the original reply generator to declare a prompt chain performing [retrieval augmented generation](#).

The first prompt in the chain is a callable prompt object called `RelevantPoliciesQuery` that is responsible for finding and summarizing any relevant policies and procedures. The actual search logic is omitted for brevity. The search results are used in a Prompt Template just like context, except this time we declare the template inline instead of in an external file.

```

1  class RelevantPoliciesQuery
2    include Raix::ChatCompletion
3
4    attr_accessor :context, :results
5
6    delegate :customer, :email_contents, to: :context
7
8    def initialize(context)
9      self.context = context
10   end
11
12   def call
13     self.results = search_customer_service_policies(customer, email_contents)
14
15     # generate the prompt based on the results
16     transcript << { user: prompt_template }
17
18     # assign the results of the completion to the context for use
19     # by the next prompt in the chain
20     context.specific_instructions = chat_completion
21   end
22
23   private
24
25   def search_customer_service_policies(customer, email)
26     # use prompt rewriting and then a hybrid vector/keyword search
27     # to pull relevant results. depending on how fancy you want to
28     # get here is where you would also do re-ranking of results
29     # to achieve better relevancy.

```

```
30     self.results = ...
31 end
32
33 def prompt_template
34     # defining the template inline instead of in a different file
35     ERB.new(<<~TEMPLATE).result(binding)
36     Customer: <%= customer %>
37     Email from Customer: <%= email_contents %>
38     Search Results: <%= results %>
39
40     Answer the following question based on the search results (if any):
41
42     What specific instructions should a customer service agent follow
43     while responding to the customer via email?
44     TEMPLATE
45 end
46 end
```

When the query class is called, it executes a chat completion that searches for relevant policies and procedures, summarizes how the search results apply to the customer's email, and then assigns its answer to the `specific_instructions` attribute of its context object, to be used by the following prompt in the chain (which in this case, is the prompt that generates the email reply to the customer.)

Regardless of the specific technology that you use to implement the Prompt Template pattern, it facilitates creating flexible, reusable, and maintainable prompt structures. This pattern promotes a clear separation of concerns, improves collaboration, and enables more efficient development and iteration of prompts.

The following pattern, [Structured IO](#), involves using a standard structured syntax, such as XML or JSON, to define some or all of your prompt template. This allows for more complex marker definitions and can facilitate better tool support and validation.

Structured IO

Structured IO is a prompt engineering pattern that leverages structured input and output to improve the accuracy and ease of integration when working with large language models (LLMs). By providing structured input, you can help the LLM better understand the task at hand and the specific data it needs to process. Similarly, by requesting structured output, you can simplify the parsing and integration of the LLM's responses into your application's workflow.

How It Works

The Structured IO pattern involves two key components:

1. **Structured Input:** You provide the input data to the LLM in a structured standard format such as XML or JSON. This structured input clearly delineates different parts of the prompt, such as instructions, examples, and data. By using a well-defined structure, you make it easier for the LLM to understand the context and requirements of the task.
2. **Structured Output:** You instruct the LLM to generate its response in a structured format. This can involve using specific tags or delimiters to mark different parts of the output, such as entity names, values, or categories. Structured output enables you to easily extract the relevant information from the LLM's response and integrate it into your application.

My favorite approach for implementing Structured IO is to use XML tags. You don't have to adhere to any particular schema, just use custom tags that match your particular use case. For example, you can use tags like `<instruction>`, `<example>`, and `<input>` to clearly demarcate different sections of your prompt.

Here's an example of how you can use XML tags to structure input when asking an LLM to extract attributes from a product description:

```
1 <description>
2 The SmartHome Mini is a compact smart home assistant available in black or
3 white for only $49.99. At just 5 inches wide, it lets you control lights,
4 thermostats, and other connected devices via voice or app—no matter where you
5 place it in your home. This affordable little hub brings convenient
6 hands-free control to your smart devices.
7 </description>
8
9 Extract <name>, <size>, <price>, and <color> from this product <description>.
```

By using XML tags to structure the input and output, the LLM implicitly understands that it should generate a response in XML:

```
1 <name>SmartHome Mini</name>
2 <size>5 inches wide</size>
3 <price>$49.99</price>
4 <color>black or white</color>
```



Scaling Structured IO

The Structured IO pattern becomes even more powerful when you need to process multiple documents or data points simultaneously. You can provide an array or list of structured inputs to the LLM and request a structured output for each item.

Here's an example of how you can use Structured IO to process multiple product reviews. Note that you can mix and match formats with ease:

```

1  reviews = [
2      {
3          id: 1,
4          text: "The SmartHome Mini is a compact, powerful, and user-friendly smart home hub. It offers great value for its price."
5      },
6      {
7          id: 2,
8          text: "The SmartHome Mini is a decent entry-level smart home hub, but it has some connectivity issues and the app needs improvement."
9      },
10     {
11         id: 3,
12         text: "Despite being affordable and compact, the SmartHome Mini's performance is disappointing, with poor voice command interpretation and unreliable device connections."
13     }
14 ].to_json

15 # Prompt the LLM to provide a summary of the reviews
16 prompt = <<~PROMPT
17     <reviews>
18     #{xml_input}
19     </reviews>
20
21     Please provide a <summary> of the <reviews>, referencing the relevant <id> tags.
22     PROMPT
23
24 # generate a response
25 chat_completion(prompt)

```

The LLM can process this structured input and generate a response like:

```

1  <summary>
2  The SmartHome Mini receives mixed reviews. <id>1</id> praises its compact
3  size, power, user-friendliness, and value for money. However, <id>2</id>
4  notes connectivity issues and room for app improvement, while <id>3</id>
5  finds the performance disappointing due to poor voice command interpretation
6  and unreliable device connections.
7  </summary>

```

By using Structured IO, you can process multiple data points efficiently and obtain a consolidated summary that references the individual items.



For bonus points, ask for output in XHTML so that you can include it directly in your application view.

Benefits and Considerations

Structured IO offers several benefits when working with LLMs:

1. **Improved Accuracy:** By providing structured input, you can help the LLM better understand the task and the specific data it needs to process. This can lead to more accurate and relevant responses and less hallucinations, especially for complex or domain-specific tasks.
2. **Easier Integration:** Structured output makes it dramatically simpler to parse and integrate the LLM's responses into your application. You can easily extract the relevant information using the defined structure and tags, reducing the need for complex post-processing or manual intervention.
3. **Consistency:** Using a consistent structure across your prompts and responses helps maintain a uniform format throughout your application. This consistency makes it easier to work with the LLM's outputs and facilitates collaboration among team members.

However, there are a few considerations to keep in mind when using Structured IO:

1. **Overhead:** Constructing structured prompts and parsing structured responses can introduce additional overhead and token consumption compared to using plain text. You need to weigh the benefits of structured IO against the added complexity and processing time. It's a minor consideration, but worth considering nonetheless.
2. **Flexibility:** While structured IO can provide more control over the LLM's behavior, it may limit the flexibility and creativity of the responses. In

some cases, allowing the LLM to generate free-form text may lead to better resilience handling outlier inputs and/or more insightful outputs.

3. **Model Compatibility:** Not all LLMs support structured IO out of the box. Some models may require fine-tuning or specific instructions to handle structured input and generate structured output effectively. It's important to test the capabilities and limitations of the LLM you are working with.

Overall, the Structured IO pattern is one of the most valuable tools in your prompt engineering toolkit. By leveraging structured input and output, you can guide the LLM to provide more accurate and easily parsable responses, ultimately improving the quality and efficiency of your AI-powered applications. When combined with other patterns like [Prompt Chaining](#) or [Retrieval-Augmented Generation](#), Structured IO can help you build sophisticated and robust AI systems that effectively narrow the path to the desired outcomes.

Prompt Chaining

Prompt Chaining is a pattern that 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.

This pattern differs from **Chain of Thought** in that it spans multiple prompt/response cycles in a conversation, rather than trying to cram the “intermediate processing” into a single chat response.

How It Works

1. **Define the Steps:** Break down the complex task or conversation into a logical sequence of smaller steps. Each step should represent a specific subtask or a meaningful unit of the overall process.
2. **Create Prompts:** For each step, create a clear and concise prompt that guides the AI towards the desired output. The prompt should provide the necessary context and instructions for the AI to generate a relevant response.
3. **Chain the Prompts:** Arrange the prompts in the desired order, forming a chain of interactions. The output of one prompt becomes the input or context for the next prompt in the chain.
4. **Iterate and Refine:** As the AI processes each prompt in the chain, review the generated responses and make necessary adjustments. Refine the prompts based on the AI's outputs to improve the overall flow and coherence of the conversation.

When To Use It

Prompt Chaining is particularly useful in the following scenarios:

- **Multi-step Processes:** When a task involves multiple steps or stages, such as a user onboarding process or a complex data analysis workflow, Prompt Chaining helps break it down into manageable units.
- **Contextual Conversations:** In applications that require maintaining context throughout a conversation, such as chatbots or virtual assistants, Prompt Chaining ensures that the AI considers the previous interactions and generates relevant responses.
- **Guided Data Collection:** When collecting structured data from users, such as filling out a form or gathering preferences, Prompt Chaining allows developers to guide the user through the process step by step, ensuring completeness and accuracy.

Example: Olympia's Onboarding

Let's take a look at how Prompt Chaining is implemented in Olympia's onboarding process to perform a guided data collection. The Onboarding class implements an interview that guides new users through a series of questions and collects relevant information about them.

```
1 class Onboarding < Tool
2   include Raix::ChatCompletion
3   include Raix::FunctionDispatch
4
5   function :save_answer, #...
6   function :onboarding_complete, #...
7   function :skip_onboarding, #...
8
9   QUESTIONS = {
10     company_name: {
11       type: "string",
12       description: "What is the name of your organization?
13                   Is it #{Current.account&.name}?"
14     },
15     company_website: {
16       type: "string",
17       description: "What is the URL of your website (if any)?"
18     },
19     company_service: {
```

```

20     type: "string",
21     description: "What service do you provide to your clients?"
22 },
23 job_title: {
24     type: "string",
25     description: "What is your job title or role?"
26 },
27 solopreneur_or_part_of_larger_team: {
28     type: "string",
29     description: "Are you a solopreneur or part of a larger team?"
30 },
31 goals_with_olympia: {
32     type: "string",
33     description: "What benefits do you hope to get from using Olympia?"
34 }
35 }.freeze
36
37 def directive_details
38     <<~DETAILS
39     This user just signed up for Olympia #{account.plan.label} plan.
40     Their name is #{account.owner.name}
41     Their email is #{account.owner.email}.
42     They named their account #{account.name}.
43     DETAILS
44 end
45
46 def instructions
47     <<~INSTRUCTIONS
48     1. Warmly welcome by name and thank them for signing up
49     2. Explain you will be conducting a brief onboarding interview
50     3. Ask: What is the name of your organization?
51         Is it #{Current.account&.name}? (if not, ask for the correct name)
52     4. Save the answer using the `save_answer` function.
53     5. The `save_answer` function returns the next question you should ask.
54
55     If user provides a website, use the `fetch_url` function to learn more
56     about their business and be able to answer related questions.
57
58     WHEN ONBOARDING IS COMPLETE or ABORTED:
59     You must call the `onboarding_complete` function so that user is
60     able to continue using Olympia, otherwise they will be stuck.
61     INSTRUCTIONS
62 end
63
64 def directive
65     "#{directive_details}\n\n#{instructions}"

```

```

66   end
67
68   def save_answer(_conversation, arguments)
69     # save...
70
71     if QUESTIONS.to_a[original_data.length].present?
72       "Saved onboarding info! The next question is: #..."
73     else
74       onboarding_complete(nil, nil)
75     end
76   end
77
78   def function_call(conversation)
79     return if account.onboarding_complete?
80     return if profile.original_data.empty?
81
82     # force function call after each human response
83     { name: "save_answer" } if conversation.last_message_by_user?
84   end
85
86   def save_answer(arguments)
87     # save and return next question...
88   end
89
90   def onboarding_complete(_)
91     account.update!(onboarding_complete: true)
92     "Completed! Ask user to standby while activating their account."
93   end
94
95   def skip_onboarding(_)
96     account.update!(onboarding_complete: true)
97     "Skipped! Ask user to standby while activating their account."
98   end
99
100  # rest of the class ommitted...

```

The onboarding process is broken down into a sequence of questions defined in the `QUESTIONS` constant. Each question is represented by a key-value pair, where the key is a unique identifier for the question, and the value is a hash containing the question's type and description.

The `directive` method provides the overall instructions for the onboarding process. It includes details about the user, such as their name, email, and the

plan they signed up for. The instructions outline the steps the AI should follow, such as welcoming the user, conducting the onboarding interview, and saving the answers using the `save_answer` function.

The `save_answer` function plays a crucial role in the Prompt Chaining process. It is responsible for saving the user's answers to the onboarding questions and determining the next question to ask. The function updates the user's profile with the provided answers and retrieves the next question from the `QUESTIONS` constant based on the number of answers already saved.

This process continues until all questions have been answered or the onboarding is explicitly completed or skipped using the `onboarding_complete` or `skip_onboarding` functions.

The `function_call` parameter (particular to OpenAI's GPT series) forces the AI to save the user's answers before each response by returning the `save_answer` function if the latest message is from the user. This helps maintain the context and coherence of the onboarding conversation.

Throughout the onboarding process, the AI can leverage additional functions like `fetch_url` (provided to it by other tools) to gather more information about the user's business based on their provided website URL. This information can be used to generate more relevant and personalized responses.

By using Prompt Chaining, Olympia's onboarding process guides new users through a structured series of questions, collecting valuable information while maintaining a natural and coherent conversation flow. The AI adapts its responses based on the user's answers, providing a personalized and engaging onboarding experience.

Prompt Rewriter

The Prompt Rewriter pattern leverages the power of LLMs to optimize and refine user-supplied prompts, making them more effective for generating accurate and relevant responses. By rewriting prompts, you can improve their clarity, specificity, and alignment with the intended purpose, ultimately enhancing the quality of the generated content.

How It Works

The Prompt Rewriter pattern involves the following steps:

1. **User Input:** The user provides an initial prompt that describes their desired output or task.
2. **Prompt Analysis:** Based on your provided guidance, the LLM analyzes the user's prompt in an attempt to understand its intent, context, and any potential ambiguities or areas for improvement.
3. **Prompt Rewriting:** Based on the analysis, the LLM generates a rewritten version of the prompt that is more optimized for generating accurate and relevant responses. This may involve:
 - Clarifying the intent and desired outcome
 - Adding specific instructions or constraints
 - Providing additional context or examples
 - Simplifying complex or ambiguous language
 - Aligning the prompt with best practices for effective prompt engineering
4. **User Confirmation (Optional):** Depending on the use case, the rewritten prompt may be presented to the user for confirmation or further refinement before being used for an actual chat completion.
5. **Chat Completion:** The rewritten prompt is used as input for the LLM to generate the final response or content.

Example

The following code implements a very simple prompt rewriter class that leaves the optimization of the prompt entirely up to the LLM:

```
1 class PromptRewriter
2   include Raix::ChatCompletion
3
4   attr_accessor :initial_prompt
5
6   def initialize(initial_prompt)
7     self.initial_prompt = initial_prompt
8   end
9
10  def call
11    user = "Analyze the prompt and output a version that you judge to be
12          more effective: <prompt>#{initial_prompt}</prompt>"
13    transcript << { user: }
14    chat_completion
15  end
16 end
```

Let's say the user wants to generate a summary of a given article. They provide the following initial prompt:

```
1 Please summarize this article:
2 [Article Text]
```

While this prompt conveys the basic intent, it lacks specificity and may not yield an effective summary. The rewritten prompt may look something like this:


```
1 Please provide a concise summary of the given article, focusing on the main points and key take-
2 eaways. The summary should be no longer than 5 sentences and should capture the essence of the
3 article's content.
4
5 Article:
6 [Article Text]
```

The LLM rewritten prompt provides clearer instructions, specifies the desired length and focus of the summary, and structures the input in a more effective format.

By using the Prompt Rewriter pattern, you can leverage the LLM's understanding of language and prompt engineering best practices to optimize user-supplied prompts. This results in higher-quality and more relevant generated content, improving the overall user experience and effectiveness of your AI-powered application.

Response Fencing

Response Fencing is a powerful prompt engineering technique that allows you to constrain and control the output generated by the AI model. By hardcoding the start of the assistant's response and specifying a stop token, you can guide certain models to produce output in a very specific format or structure, making it easier to parse and process the generated content.

How It Works

The key components of Response Fencing are:

1. **Hardcoded Start:** By hardcoding the beginning of the assistant's response, you provide a clear starting point for the model to continue from. This helps to establish the desired format or structure of the output.
2. **Stop Token:** The stop token is a predefined string or sequence that marks the end of the model's response. When the model encounters the stop token during generation, it halts the output at that point. This prevents the model from generating additional content beyond what is required.

We used response fencing in the `JsonFixer` class presented in the [Self Healing Data](#) chapter. Let's reproduce just the most relevant parts of that code here and break down how it works:

```

1  def call(bad_json, error_message)
2      #...
3
4      transcript << { system: "Consider user-provided JSON..." }
5      transcript << { user: bad_json }
6      transcript << { assistant: "What is the error message?"}
7      transcript << { user: error_message }
8      transcript << { assistant: "Here is the corrected JSON\n```\njson\n" }
9
10     self.stop = ["```"]
11
12     chat_completion
13 end

```

1. The call method takes the bad_json and error_message as input.
2. The conversation transcript is constructed using a series of system, user, and assistant messages. The system message provides the overall instruction to fix the JSON while preserving the original content and intent. The rest of the messages provide the bad json and error messages using the [Ventriloquist](#) pattern explained later in this chapter.
3. Importantly, the final assistant message guide the model's response. This is *the start of the fence*.
4. The stop token is set to ["```"], indicating that the model should stop generating output when it encounters the triple backticks (```). This is *the end of the fence*.
5. The chat_completion method is called to generate a response.

By hardcoding the start of the assistant's response with "Here is the corrected JSON\njson\n```\n", the model is prompted to generate the corrected JSON within the triple backticks. The stop token `["```"] ensures that the model stops generating output once it encounters the closing triple backticks.

Benefits and Considerations

By constraining the model's response, you can ensure that the generated output follows a specific format or structure. This makes it easier to parse, validate, and process the output programmatically.

Response Fencing is especially helpful in mitigating the risk of the model generating irrelevant or unexpected content, including meta commentary about the structured output you asked for. Examples of commentary for something like the `JsonFixer` example would be: "Here's the corrected JSON code you asked for...".

By providing a clear starting point and stop token, you can guide the model to produce output that aligns with your expectations.

However, there are a few considerations to keep in mind.

The first is that while Response Fencing provides control over the output format, it may limit the model's flexibility in generating more open-ended or creative responses. It's important to strike a balance between constraining the output and allowing the model to generate meaningful content.

Here's a practical example, again using the `JsonFixer` class. What happens if we pass in HTML as the bad JSON code that should be fixed? Depending on how strictly the model is trained to adhere to response fencing, it may turn the HTML into JSON. That's exactly what Mixtral 8x7B Instruct did in my testing:

```

1 > JsonFixer.new.call('<[!DOCTYPE html><!--[if lt IE 7]> <html class="no-js ie6 oldie" lang="en
2 -US"> ...', "unexpected character (after [0]) at line 1, column 2 [parse.c:762]")
3
4 =>
5 {"html"=>
6   {"head"=>{"meta"=>{"charset"=>"UTF-8", "content-type"=>"text/html",
7     "http-equiv"=>"X-UA-Compatible", "content"=>"IE=edge,chrome=1",
8     "viewport"=>"width=device-width, initial-scale=1.0"}},
9   "body"=>
10    {"div"=>
11      {"id"=>"page",
12       "class"=>"hfeed site", ...

```

The same call using GPT-4 as the model generated the following output:

```

1 The content you've provided appears to be HTML rather than JSON. JSON (JavaScript Object Notat
2 ion) is a lightweight data-interchange format that is easy for humans to read and write and ea
3 sy for machines to parse and generate. It is based on a subset of the JavaScript Programming L
4 anguage and uses a completely different syntax from HTML. (JSON::ParserError)
5
6 If you intended to provide JSON data but included HTML by mistake, please provide the correct
7 JSON data. If you need help with HTML or converting HTML data into a JSON format, please prov
8 de more details about what specific information you need to be converted or handled.

```

This is an example of different AI models having different “personalities”, so-to-speak. Mixtral is compliant, whereas GPT-4 is smart (or stubborn) enough—depending on your perspective and intention—to break out of the provided response fencing. And by breaking out, it might break code that is depending on receiving structured output.



The effectiveness of Response Fencing depends on the capabilities of the language model being used. Some models may be better suited for generating structured output than others. Not all models understand what it means when you end your transcript with an assistant message. Not all models accept stop parameters. It's crucial to choose a model that aligns with your specific requirements and test thoroughly.

Error Handling

As demonstrated, Response Fencing is not foolproof. So it's important to handle cases where the model may not generate the expected output or fails to include the stop token. Implementing appropriate error handling mechanisms and fallback strategies can help ensure the robustness of your application.

Query Analyzer

The Query Analyzer pattern plays a pivotal role in the [Intelligent Query Optimization \(IQO\)](#) process, serving as the foundation upon which effective query rewriting is built.

The Query Analyzer pattern is designed to dissect and comprehend the user's query in depth. Its primary goal is to extract the intent, contextual clues, and the specific informational needs embedded within the query. This understanding is critical for transforming the query into a form that's precisely aligned with the knowledge database, thereby improving the retrieval process's accuracy and relevance.

How It Works

Intent Extraction: At this stage, the analyzer discerns the primary goal of the user's prompt. Is the user seeking a factual answer, explanation, or guidance on a procedure? Identifying the intent is crucial for directing the query optimization process.

Contextual Analysis: Here, the analyzer examines the context surrounding the query. This could involve analyzing the conversation history, the domain of inquiry, or any specific conditions mentioned by the user. Contextual analysis ensures that the rewritten query maintains relevance to the user's current needs.

Informational Need Identification: Beyond intent and context, understanding exactly what information the user seeks is vital. This step involves pinpointing the keywords or concepts that are essential for retrieving relevant and accurate information.

Implementation

While the Query Analyzer pattern can be implemented using various NLP techniques and machine learning models, in the context of this book, we'll focus on how to leverage LLMs to achieve query analysis. It's important to note that some of the techniques mentioned below, such as parts of speech tagging and named entity recognition, go beyond the scope of this book. However, we'll provide some pointers on how to implement them using LLMs or existing APIs.

Part-of-Speech (POS) Tagging and Named Entity Recognition (NER)

While these techniques are not the main focus of this book, you can still leverage LLMs to prototype POS tagging and NER, and maybe even use them as your production implementation if you're operating at low scales.

By crafting appropriate prompts, you can guide the LLM to identify parts of speech and named entities in the query, and generate them either as tagged output or function calls.

Here's an example of how you can prompt an LLM to perform POS tagging:

- 1 Perform part-of-speech tagging on the following query. Return the output as a list of words and
- 2 their corresponding part-of-speech tags.
- 3
- 4 Query: What are the best hiking trails near Denver for beginners?

The LLM would generate POS-tagged output like this (GPT-4o, June 2024):


```

1 Here is the part-of-speech tagging for the query "What are the best hiking trails near Denver
2 for beginners?" with each word and its corresponding part-of-speech tag:
3
4 1. What - WP (Wh-pronoun)
5 2. are - VBP (Verb, non-3rd person singular present)
6 3. the - DT (Determiner)
7 4. best - JJS (Adjective, superlative)
8 5. hiking - NN (Noun, singular or mass)
9 6. trails - NNS (Noun, plural)
10 7. near - IN (Preposition or subordinating conjunction)
11 8. Denver - NNP (Proper noun, singular)
12 9. for - IN (Preposition or subordinating conjunction)
13 10. beginners - NNS (Noun, plural)

```

Similarly, you can prompt the LLM to perform NER:

```

1 Identify and extract named entities from the following content. Return the output as a list of
2 named entities and their corresponding entity types.
3
4 Content: Mark Zuckerberg is one of the founders of Facebook, a company from the United States
5
6 Example Output:
7 <ner>
8   <entity type="location">Denver</entity>
9 </ner>

```

The LLM can then generate the NER output:

```

1 Here is the named entity extraction for the query "Mark Zuckerberg is one of the founders of F
2 acebook, a company from the United States":
3
4 <ner>
5   <entity type="person">Mark Zuckerberg</entity>
6   <entity type="organization">Facebook</entity>
7   <entity type="location">United States</entity>
8 </ner>

```

Alternatively, you can use external APIs like Google Cloud Natural Language API or Microsoft Azure Text Analytics API, which provide POS tagging and NER capabilities.

Here's an example of how you can use the Google Cloud Natural Language API for POS tagging and NER:

```
1 require "google/cloud/language"
2
3 def analyze_query(content)
4   language = Google::Cloud::Language.language_service
5
6   document = { content:, type: "PLAIN_TEXT" }
7   response = language.analyze_syntax(document:)
8
9   puts "Part-of-Speech Tags:"
10  puts response.tokens.map do |token|
11    "#{token.text.content} (#{token.part_of_speech.tag})"
12  end
13
14  puts "\nNamed Entities:"
15  puts response.entities.map do |entity|
16    "#{entity.name} (#{entity.type})"
17  end
18 end

1 >> analyze_query("What are the best hiking trails near Denver for beginners?")
2
3 => Part-of-Speech Tags:
4 What (PRON)
5 are (VERB)
6 the (DET)
7 best (ADJ)
8 hiking (NOUN)
9 trails (NOUN)
10 near (ADP)
11 Denver (NOUN)
12 for (ADP)
13 beginners (NOUN)
14 ? (PUNCT)
15
16 Named Entities:
17 Denver (LOCATION)
```

At the cost of additional system dependencies and operational complexities, external APIs let you offload the complexity of POS tagging and NER to specialized, scalable, and cost-effective services, allowing you to focus on integrating the results into your query analysis pipeline.

Intent Classification

To classify the intent of the query using an LLM, you can provide the LLM with a set of predefined intents and their corresponding examples. By formatting the query and the intent examples in a specific way, you can prompt the LLM to classify the query into one of the predefined intents.

Here's a simple example:

```
1  Classify the following query into one of the predefined intents based on the examples provided.
2
3  Predefined Intents:
4  1. Informational: Queries seeking information or answers to questions.
5     Examples:
6     - What is the capital of France?
7     - How to bake a chocolate cake?
8     - Who won the World Cup in 2018?
9
10 2. Navigational: Queries seeking a specific website or web page.
11    Examples:
12    - Facebook login
13    - YouTube
14    - Amazon Prime Video
15
16 3. Transactional: Queries expressing an intent to perform an action or transaction.
17    Examples:
18    - Buy iPhone 12
19    - Download Microsoft Office
20    - Book a flight to New York
21
22 Query: "Where is my order #172838. It was supposed to arrive a week ago!"
23
24 Example Output: <intent>Informational</intent>
```

The LLM can then classify the query based on the provided intent examples:

```
1  <intent>Informational</intent>
```

By providing clear examples for each predefined intent, you can guide the LLM to accurately classify the user's query into the most appropriate intent category.

Note that one of the downsides of using LLMs in this way as opposed to special-purpose AI components is that LLMs have a tendency to deviate from provided constraints, no matter how sternly you tell them not to do so. For instance, the previous example might cause the LLM to tag the query as something other than Informational, Navigational, or Transactional.

```
1 <intent>Customer Support</intent>
2
3 While the query seeks information about the status of an order, it does not fit neatly into th
4 e predefined intents of Informational, Navigational, or Transactional. The query expresses a c
5 ustomer support issue or complaint, which is a distinct category not covered by the provided
6 ntent examples. The user is not merely seeking information but is expressing a concern about a
7 delayed order and likely expects a resolution or assistance from customer support.
```

This example demonstrates how LLMs can be “clever” enough to detect nuances and intents that may not be explicitly covered by the provided examples, highlighting the importance of carefully defining and covering a wide range of intents when using LLMs for intent classification.

I feel that this example in particular also perfectly demonstrates what is simultaneously adorable and maddening about LLMs. It casually deviated from your instructions, but not without providing a thorough explanation about why it did so.

Keyword Extraction

LLMs can be used to extract keywords or phrases from the query that represent the core information need. By providing the LLM with the query and asking it to identify the most important keywords, you can obtain a list of relevant terms.

Here’s a simple example prompt:

```
1 Extract the most important keywords or phrases from the following query
2 that represent the core information need.
3
4 Query: What are the best hiking trails near Denver for beginners?
5
6 Keywords:
7 <keywords>
8 - best hiking trails
9 - near Denver
10 - for beginners
11 </keywords>
```

The LLM can then generate a list of relevant keywords:

```
1 <keywords>
2 - best hiking trails
3 - near Denver
4 - for beginners
5 </keywords>
```

By extracting these keywords, you can focus on the essential aspects of the query and use them to guide the retrieval of relevant information.

You can further enhance the keyword extraction prompt by providing additional instructions or constraints, such as:

```
1 Extract the most important keywords or phrases from the following query
2 that represent the core information need. Limit the extraction to a maximum
3 of 5 keywords and prioritize specific locations, activities, or attributes
4 mentioned in the query.
5
6 Query: What are the best hiking trails near Denver for beginners?
7
8 Keywords:
9 <keywords>
10 1. hiking trails
11 2. Denver
12 3. beginners
13 4. best
14 </keywords>
```

This prompt instructs the LLM to prioritize specific types of keywords and

limits the output to a maximum of 5 keywords, resulting in a more focused and relevant keyword extraction.

By leveraging the power of LLMs for intent classification and keyword extraction, you can gain valuable insights into the user's query and use that information to optimize the retrieval and generation of relevant responses.

Benefits

The Query Analyzer pattern offers several key benefits in the context of Intelligent Query Optimization:

1. **Improved Query Understanding:** By thoroughly analyzing the user's query, the Query Analyzer enables a deeper understanding of the user's intent, context, and information needs. This understanding is crucial for generating more relevant and accurate responses.
2. **Enhanced Relevance:** With a clear understanding of the query's intent and context, the Query Analyzer helps in identifying the most important aspects of the query. This allows for more targeted retrieval of information, improving the relevance of the results returned to the user.
3. **Better User Experience:** By accurately capturing the user's intent and information needs, the Query Analyzer contributes to a better overall user experience. Users are more likely to find the information they are looking for quickly and easily, leading to increased satisfaction and engagement.
4. **Efficient Query Processing:** The Query Analyzer breaks down the query into its constituent parts, allowing for more efficient processing and optimization. By focusing on the most important aspects of the query, the system can streamline the retrieval process and reduce unnecessary computations.
5. **Adaptability to User Needs:** The Query Analyzer can adapt to different user needs and contexts by considering factors such as conversation history and domain-specific requirements. This adaptability ensures that

the query optimization process remains relevant and effective for a wide range of user queries.

6. **Foundation for Further Optimization:** The insights gained from the Query Analyzer serve as a foundation for implementation of other patterns, such as *Query Rewriter*. By providing a clear understanding of the query's intent and key components, the Query Analyzer enables more effective optimization strategies.

Overall, the Query Analyzer pattern plays a vital role in improving the accuracy, relevance, and efficiency of Intelligent Query Optimization.

Query Rewriter

After understanding the user's query through the Query Analyzer pattern, the Query Rewriter pattern becomes essential. This pattern refines the original query to better match the language model's understanding, thereby improving the relevancy of retrieved information for the user.

This pattern focuses on adjusting the original query to ensure it's in a form that the LLM finds easier to interpret. This process involves simplifying, rephrasing, or augmenting the query based on the analysis performed in the previous step. The goal is to bridge the gap between the user's natural language and the LLM's optimal input format for information retrieval.

How It Works

Simplification: Condense complex queries into simpler, more direct questions that the LLM can understand more easily. This might mean breaking down a multi-faceted question into several smaller, more manageable queries.

Rephrasing for Clarity: Adjust the phrasing of the query to match patterns or structures that the LLM responds to more effectively. For instance, turning open-ended questions into more concrete, answerable formats.

Keyword Enhancement: Identify and include keywords or phrases that are more likely to trigger the desired response from the LLM, based on the context and intent identified by the Query Analyzer.

Example

The Query Rewriter pattern can be implemented as a [Prompt Chain](#) that performs a series of modifications to an initial query.


```
1  class QueryRewriter
2    include Raix::ChatCompletion
3    include Raix::PromptDeclarations
4
5    prompt call: Translator
6
7    prompt call: RephraseForClarity
8
9    prompt call: EnhanceWithKeywords
10
11    ...
12  end
```

The prompt chain in the example `QueryRewriter` class serves to refine user queries to optimize them for information retrieval tasks. The prompts are structured to carry out three critical steps: translation to English (if necessary), rephrasing for clarity, and enhancing with keywords. Each step is aimed at tackling different aspects of query optimization, from overcoming language barriers to ensuring the query's intent is clear and well-defined, and finally, to boosting its specificity with relevant keywords.

Benefits

The benefits of a well-implemented Query Rewriter are myriad. First, it lets you try to significantly improve the match between user queries and your knowledge base, increasing the chances of retrieving the most relevant and accurate information. By delivering more precise and relevant results, the Query Rewriter can enhance user satisfaction and trust in the system, encouraging further use and engagement.

Ventriloquist

The Ventriloquist pattern is a powerful technique in prompt engineering that allows you to guide the AI's responses by preloading hardcoded user-assistant exchanges into the conversation transcript before starting any completions. This pattern enables you to narrow the path effectively and ensure that the AI stays on course, providing responses that align with your desired direction.

While the Ventriloquist pattern can be used to maintain a consistent voice, tone, or persona across the AI's responses, its most significant benefit lies in its ability to subtly influence the AI's behavior. By carefully crafting the initial exchanges and setting the context, you can steer the AI towards generating responses that meet your specific requirements in a “show and tell” kind of way, as opposed to having to try to fully explain what you're seeking in a system directive or prompt.

How It Works

The Ventriloquist pattern involves the following steps:

1. Define the desired direction or outcome for the AI's responses.
2. Create a set of hardcoded user-assistant exchanges that establish the context and guide the AI towards the intended direction.
3. Preload these exchanges into the conversation transcript before starting any completions.
4. Initiate the chat completion process, allowing the AI to generate responses based on the preloaded context.
5. The AI, influenced by the initial exchanges, generates responses that align with the desired direction.

By preloading the conversation transcript with carefully crafted exchanges, you essentially put words in the AI's mouth, much like a ventriloquist control-

ling their puppet. The AI, being a language model trained to maintain coherence and consistency, will naturally follow the established context and generate responses that continue along the same path.

When to Use It

The Ventriloquist pattern is particularly useful in scenarios where:

- You want to guide the AI's responses towards a specific topic, sentiment, or style.
- You need to establish a particular context or background information before allowing the AI to generate responses.
- You want to maintain consistency in the AI's behavior across multiple conversations or sessions.
- You need to provide examples or demonstrations of desired responses to influence the AI's output.

By employing the Ventriloquist pattern, you can effectively control the direction of the conversation and ensure that the AI generates responses that meet your specific requirements.

Example

Here's an example implementation of the Ventriloquist pattern in Ruby:

```

1  class AlternateKeywords
2    include Raix::ChatCompletion
3
4    PROMPT = <<~END
5      Matching the original language of the question generate 3 alternate
6      keywords that might produce better results. Reply with just the list,
7      one per line.
8    END
9
10   def call(question)
11     transcript << { system: "You are a powerful web search engine" }
12     transcript << { user: question }
13     transcript << { assistant: "Searching... no results found." }
14     transcript << { user: PROMPT }
15     chat_completion
16   end
17
18   def max_tokens
19     30
20   end
21
22   def model
23     [
24       "databricks/dbrx-instruct:nitro",
25       "cohere/command-r",
26     ]
27   end
28 end

```

In this example, the `AlternateKeywords` class demonstrates the Ventriloquist pattern by preloading a conversation transcript with specific user-assistant exchanges before generating alternate keywords.

The `call` method takes a `question` parameter and performs the following steps:

1. Instructs the AI that they should act like a “powerful web search engine” using the system directive.
2. Appends the user’s question to the conversation transcript.
3. Appends a hardcoded assistant response indicating that no results were found.

4. Appends the PROMPT to the transcript, instructing the AI to generate alternate keywords, just one per line.
5. Initiates the chat completion process.

By preloading the transcript with the hardcoded “no results found” response, the AI is primed to understand that we are trying to execute a search. The final user prompt then explicitly asks for alternate keywords.

This pattern allows you to guide the AI’s behavior and obtain desired responses by carefully crafting the initial conversation transcript, and is sometimes more effective and succinct than trying to describe your desired outcome using the prompt only.

Discrete Components



The integration of AI capabilities into your code often revolves around the use of specialized AI tools and functions implemented as discrete components. By encapsulating potentially complex AI functionality into individually separate and distinct building blocks, you can streamline the process of incorporating intelligent features and behaviors into your software systems.

Some of these components represent a fundamental shift in how we approach application development, allowing us to leverage the vast potential of AI technologies modular building blocks. They provide a layer of abstraction that shields you from having to invest time in implementing lower-level, complicated and/or tedious code. In some cases, they allow you to add features that you would never have the time, patience, or capital to work on otherwise. By abstracting away the low-level details, these components reduce implementation effort and allow you to concentrate on your overall application logic and user experience.

Encapsulating AI functionality into discrete components even offers the promise of reusing them across more than one project, letting you achieve a more flexible and scalable development process.

I predict that a world of greenfield open-source opportunities will emerge in the coming years, as people realize that there is a demand for reusable AI components that can be simply dropped into their code with little effort, just as you would import a 3rd-party library today.

Whether it's the Predicate pattern for making binary decisions based on AI insights, the Result Interpreter pattern for extracting meaningful information from AI-generated results, or other patterns that encapsulate AI functionality, the tools and functions presented in this chapter serve as a foundation for building intelligent and innovative applications.

As you explore the patterns and examples in this chapter, keep in mind that the true potential of tools and functions in AI-driven development lies in their ability to abstract away complexity. By mastering the use of these tools and functions, you can unlock new possibilities for intelligent automation, personalized user experiences, and data-driven decision-making within your applications.

So, let's explore the world of discrete components in your application development and discover how these powerful components can revolutionize the way we build software systems in the era of AI.

Predicate

It's common to have scenarios where you need to make a binary decision based on the available information. This is where the Predicate pattern comes into play. The term “predicate” originates from logic and mathematics, where it refers to a statement or function that evaluates to either true or false. In the context of AI, the Predicate pattern involves posing a specific question to the AI model and expecting a definitive yes or no answer.

The Predicate pattern is particularly useful when you need to determine whether the available context or information is sufficient for the AI to provide a meaningful response or take a specific action. It helps in situations where making assumptions or proceeding with incomplete information could lead to inaccurate or misleading results. By employing the Predicate pattern, you can ensure that the AI model has enough context to make informed decisions or provide reliable answers.

How It Works

The Predicate pattern typically involves the following steps:

1. Formulate a specific question that can be answered with a yes or no response. This question could be designed to assess the sufficiency of the available information for a particular task or decision (as in this section's example code.) Or it could be literally anything else about which you could want ask a yes or no question.
2. Provide the AI model with the relevant context or information needed to answer the question. This could include conversation transcripts, user inputs, or any other pertinent data.
3. Prompt the AI model with the question and the provided context, expecting a binary response (yes or no) along with a rationale or explanation for the answer.

4. Analyze the AI model's response to determine the appropriate course of action. If the answer is "yes," proceed with the intended task or decision. If the answer is "no," take alternative actions such as gathering more information or prompting the user for additional input.

A low temperature hyperparameter value is important for this and practically all other similar functions. The reason is *consistency*. Remember that the temperature setting influences the variety in the model's responses. Lower values lead to more predictable and typical responses, while higher values encourage more diverse and less common responses. At 0, the model should always give the same response for a given input.

That said, temperature is not the only parameter to experiment with. Depending on the model, you can also try different values for *Top K* and *Top P*.

Top K limits the model's choice of tokens at each step, making it choose from a smaller set. A value of 1 means the model will always pick the most likely next token, leading to predictable results. Most models default this setting to 0 (disabled), making the model to consider all choices. *Top P* limits the model's choices to a percentage of likely tokens: only the top tokens whose probabilities add up to *P*. A lower value makes the model's responses more predictable, while the default setting allows for a full range of token choices. Think of it like a dynamic *Top-K*.

When to Use It

The Predicate pattern is valuable in various scenarios where decision-making based on available information is critical, especially if the condition is based on complex or *fuzzy* data. The use cases are endless, but include:

- Deciding whether to proceed with a task
- Assessing the completeness of user input

- Determining whether to provide a personalized user experience

By employing the Predicate pattern, you can enhance the reliability and accuracy of your AI-driven application by ensuring that decisions are made based on sufficient and relevant information.

Example

Here's an example implementation of the Predicate pattern used in the core conversation loop of Olympia to decide whether to query long-term memory or respond immediately:

```

1  class EnoughInfo
2      include Raix::ChatCompletion
3      include Raix::Functions::Predicate
4
5      attr_reader :text_transcript
6
7      QUESTION = <<~END
8          BEGIN TRANSCRIPT
9              %{text}
10             END TRANSCRIPT
11
12             The assistant is a personal AI that can help with a wide range of tasks
13             or simply converse with the user in a friendly manner like a companion.
14             The assistant has a long-term memory facility which it can invoke to
15             remember things that the user has told it in the past.
16
17             First analyze what the user is talking about, particularly the subjects
18             involved, which may include people, places, things, or concepts that are
19             personal to them and not in the public domain or your base knowlege.
20
21             Then answer the question:
22             Does the transcript contain enough context for the assistant to be able
23             to answer the user question without making assumptions about what they're
24             talking about?
25
26             Your response must begin with 'Yes, ' or 'No, ' followed by your
27             rationale. If you answer no, then the assistant will search its long-term
28             memory to gather more context, but that is an expensive operation and
29             should be avoided if possible.
30         END

```

```
31
32  def initialize(conversation)
33    text = conversation.text_transcript(limit: 10)
34    transcript << {user: QUESTION % {text:} }
35  end
36
37  def call
38    chat_completion
39  end
40
41  def model
42    [
43      "cohere/command-r",
44      "openai/gpt-3.5-turbo-0125"
45    ]
46  end
47 end
```

In this example, the EnoughInfo class implements the Predicate pattern to determine whether a conversation transcript contains enough context for an AI assistant to answer a user's question without making assumptions.

The prompt is templated in the QUESTION constant, which defines the specific question that the AI model needs to answer. It provides context about the AI assistant's capabilities and asks whether the provided transcript contains sufficient information to answer the user's question.

The initialize method takes a conversation object and extracts the relevant transcript using the text_transcript method, then sets the transcript array for the completion.

The call method initiates the chat completion and returns a boolean value. The Raix::Functions::Predicate module provides casting using code that looks like response.starts_with?("Yes").

This is how this how the Predicate is used in the Conversation class:

```
1 class Conversation
2
3   def has_enough_info?(prompt)
4     raise "No messages. Cannot evaluate conversation" unless messages.any?
5
6     EnoughInfo.new(self).tap do |ei|
7       ei.on_yes do
8         # enough information, proceed to generate a normal response
9         update_cstatus("Thinking...")
10      end
11      ei.on_no do |explanation|
12        # not enough information, we need to search memory first
13        RecallOperation.new(self).call(prompt, explanation)
14      end
15    end.call
16  end
17
18  #...
19 end
```

As seen in the usage example above, the `on_yes` and `on_no` callback methods (provided by the `PredicateFunction` module) allow you to define blocks of code that will be executed based on the AI model's response. If the response starts with "Yes," the block passed to `on_yes` will be called, indicating that there is enough context to answer the question. If the response starts with "No," the block passed to `on_no` will be called, suggesting that additional context is needed. The `explanation` parameter is whatever comes after the yes and no, for potential use in subsequent operations.

The `EnoughInfo` class is an example of how to integrate the Predicate pattern into your application to determine whether there is sufficient context to answer a user's question. This helps in making informed decisions and avoiding assumptions when providing responses or taking actions based on the available information.

API Facade

An *API Facade* (or “helper”) is a special-purpose discrete component that sits alongside a more generalized AI assistant, and provides it access to an API, typically via a single plaintext request function. In doing so, it prevents the generalized AI assistant from having to bloat its list of functions with the implementation of capabilities specific to the API.

This pattern gets its name and is a manifestation of the Gang of Four *Facade* pattern, described as providing a “unified interface to a set of interfaces in a subsystem.”

How It Works

The API Facade provides a single, simplified interface to the facilities of the subsystem provided via an API. It does so by:

1. Exposing a single request endpoint which takes either plaintext or parameterized input.
2. Handling requests as a looped chat completion with access to functions that map to needed API endpoints.
3. Returning a response that incorporates the results of the function calls.

Key Benefits

The API Facade pattern offers several benefits that make it a valuable tool in your AI application development toolkit:

- **Simplified Interface:** By exposing a single, unified interface to the underlying API, the facade simplifies the interaction between the main conversation loop and the API. This makes it easier to integrate the API functionality into your application and reduces the complexity of your main conversation loop.

- **Encapsulation:** The API Facade encapsulates the complexities of the API interactions within the facade component. This promotes a clear separation of concerns, allowing you to focus on the high-level conversation flow in your main loop while delegating the API-specific details to the facade.
- **Error Handling:** The facade provides a centralized place to handle errors and edge cases specific to the API. You can implement robust error handling mechanisms within the facade, such as retry logic, rate limiting, and graceful degradation, to ensure a smooth user experience even in the face of API failures or limitations.
- **Reusability:** The API Facade can be designed as a reusable component that can be easily integrated into different parts of your application or even across multiple projects. By encapsulating the API interactions within the facade, you can reuse the same facade implementation in various contexts, saving development time and effort.

When To Use It

Use this pattern wherever you want to abstract away the details of how to interact with an API so that your main conversation loop doesn't carry the overhead of all the functions that would be required for using that API.

At Olympia, our main “conversation loop” includes over a dozen functions related to memory, web search capabilities, and other basic features. The definitions for those functions adds a substantial amount of token cost to *every single user prompt*. In most cases, those definitions also act as a distraction from whatever the user is trying to do. At the time that I'm writing this book, the more functions you have in your conversation loop, the worse that your AI will perform.

That's what makes API Facade one of the most important patterns in the book. Using it you can add functionality to your main conversation loop without bloating its function count.

Example

Here is a real example of an API Facade extracted from Olympia. The `GmailService` class provides our assistants with the ability to manage the user's Gmail account, all through one function in the conversation loop. I've tried to retain a large amount of detail intact without letting the example get too overwhelming. Here it is:

```

1  class GmailService
2    include Raix::ChatCompletion
3    include Raix::FunctionDispatch
4
5    MAX_ERRORS = 4
6
7    attr_accessor :conversation, :errors, :user
8
9    alias usage_subject conversation
10
11   def initialize(conversation)
12     self.errors = []
13     self.conversation = conversation
14     self.user = conversation.owner
15     self.max_tokens = 2000
16     # self.tool_choice = 'required'
17   end
18
19   SYSTEM_DIRECTIVE = <<~END
20     The "user" conversing with you is another GPT helping its human user to
21     manage their Gmail account. Because your replies will be processed by
22     another AI, they do not need conversational commentary. Include full
23     message data (id, recipients, subject line, full body) in responses about
24     specific email messages.
25   END
26
27   def call(prompt)
28     transcript << { system: SYSTEM_DIRECTIVE }
29     transcript << { user: prompt }
30     chat_completion(loop: true)
31   end
32
33   function :messages,
34     "Returns messages from the user's mailbox. Use q=is:unread for
35     unread messages.",

```

```
36         max_results: { type: 'integer' },
37         q: { type: 'string' } do |params|
38
39     options = {
40         max_results: params[:max_results].presence || 10,
41         q: params[:q].presence
42     }.compact
43
44     service
45       .list_user_messages('me', **options)
46       .messages
47       .map { |msg| format_message(msg, truncate: 300) }
48   end
49
50   function :message,
51     "Retrieve a specific message by ID. Do not summarize your
52     response! Include the full data of the message in your response
53     to the user.",
54     id: { type: 'string' } do |params|
55
56       format_message(service.get_user_message('me', params[:id]))
57   end
58
59   function :toggle_read,
60     "Mark a message as read or unread",
61     id: { type: 'string' },
62     read: { type: 'boolean' } do |params|
63
64     # implementation...
65   end
66
67   function :delete_message,
68     "Delete a message",
69     id: { type: 'string' } do |params|
70
71     service.delete_message('me', params[:id])
72   end
73
74   function :list_labels, "Lists all labels in the user's mailbox." do
75     service.list_user_labels('me').labels
76   end
77
78   function :create_label, #...
79
80   function :get_filter, #...
81
```



```

82   function :list_filters, #...
83
84   function :create_filter, #...
85
86   function :delete_filter, #...
87
88   private
89
90   def service
91     @service ||= Google::Apis::GmailV1::GmailService.new.tap do |service|
92       service.authorization = user.auth_token_for('Gmail')
93     end
94   end
95
96   def format_message(msg, clean: false, truncate: 10_000)
97     #...
98   end
99
100  # many more private helper methods...
101
102  end

```

I'll leave explanation of how this class works as an exercise to the reader, but I will show you how it's used by the main conversation loop.

Here is the function definition for the helper:

```

1  GMAIL__HELPER_FUNCTION = {
2    name: "gmail_helper",
3    description: "Use to read and manage messages and filters on the user's
4                  Gmail account. Output data you get from this function back to
5                  the user using markdown tables whenever possible.",
6    parameters: {
7      type: "object",
8      properties: {
9        request: {
10         type: "string",
11         description: "Concise plaintext requests spoken in the voice of the
12                      user like 'Find recent messages from bob@example.com'
13                      or 'Anything urgent in my unreads?' or 'Create a filter
14                      to automatically archive any messages from John Doe'"
15       }
16     }
17   }
18 }

```

And here is the implementation, which simply relays the request parameter to an instance of `GmailService`:

```
1 def gmail_helper(conversation, arguments)
2   GmailService.new(conversation).call(arguments[:request])
3 rescue StandardError => e
4   Honeybadger.notify(e, context: { conversation:, arguments: })
5   e.message
6 end
```

In the provided example, the `GmailService` class acts as the API Facade for interacting with the Gmail API. Let's dive deeper into some of the key aspects of this implementation:

Authentication and Authorization

The `GmailService` class handles the authentication and authorization process for accessing the user's Gmail account. It abstracts away the complexities of managing API credentials and tokens from the main conversation loop. The service method within the class retrieves the user's authorization token using `user.auth_token_for('Gmail')`, ensuring that the API requests are properly authenticated.

Request Handling

The `call` method in the `GmailService` class serves as the entry point for handling plaintext requests from the main conversation loop. It appends the system directive and user prompt to the conversation transcript and initiates a looped chat completion. The class defines several functions, such as `messages`, `message`, `toggle_read`, and `delete_message`, which map to specific Gmail API endpoints. These functions extract the necessary parameters from the user's request and construct the appropriate API calls using the `Google::Apis::GmailV1::GmailService` client.

Response Formatting

The API Facade takes care of formatting the API responses into a structured and user-friendly format before returning them to the main conversation loop. For example, the `messages` function retrieves a list of messages from the user's mailbox and formats each message using the `format_message` helper method. This formatting step ensures that the main conversation loop receives the message data in a consistent and easily consumable format.

Error Handling and Edge Cases

The `GmailService` class incorporates error handling mechanisms to gracefully handle API failures and edge cases. For instance, it limits the number of errors that can occur before it bails out. If an error occurs during the API interaction, it is caught, logged using `Honeybadger.notify`, and the error message is returned to the main conversation loop. This ensures that the main loop can handle and respond to API errors appropriately, including telling the user what went wrong and whether to try again.

By encapsulating the Gmail API interactions within the `GmailService` facade, the main conversation loop can focus on the high-level flow of the conversation while delegating the API-specific details to the facade. This separation of concerns improves code organization, maintainability, and reusability.

Scalability and Performance Considerations

When implementing the API Facade pattern, it's important to consider the scalability and performance aspects of your application. Here are a few tips to optimize API usage and improve the overall performance:

- **Caching:** If possible, implement caching mechanisms within the API Facade to store frequently accessed data or responses. This can help reduce the number of API calls and improve response times. You can

use techniques like in-memory caching or distributed caching solutions, depending on your application's requirements.

- **Rate Limiting:** Implement rate limiting within the facade to prevent excessive API calls and avoid hitting API rate limits. You can use techniques like throttling or token bucket algorithms to control the rate of API requests. This ensures that your application stays within the API's usage limits and maintains a good standing with the API provider.
- **Pagination:** When dealing with large datasets or API responses, implement pagination within the facade. Instead of retrieving all the data at once, fetch it in smaller chunks or pages. This helps reduce the payload size and improves the performance of your application. You can expose pagination options through the facade's interface, allowing the main conversation loop to control the pagination flow.
- **Asynchronous Processing:** If the API Facade needs to perform time-consuming tasks or make multiple API calls, consider implementing asynchronous processing. This allows the main conversation loop to continue executing while the facade handles the API interactions in the background. You can use techniques like job queues or `async/await` patterns to achieve asynchronous processing. The API Facade could even be given responsibility for inserting messages containing results into the original conversation where the request originated.

By considering these scalability and performance aspects, you can design an API Facade that not only simplifies the API integration but also optimizes the resource utilization and responsiveness of your AI application.

Comparison with Other Design Patterns

The API Facade pattern shares some similarities with other Gang of Four design patterns, such as the Adapter and Proxy patterns. However, there are some key differences:

The *Adapter* pattern focuses on converting the interface of one class into another interface that clients expect. It allows incompatible interfaces to work together. In contrast, the API Facade pattern provides a simplified interface to a complex subsystem, hiding the complexity and making it easier to use.

The *Proxy* pattern provides a surrogate or placeholder for another object to control access to it. It acts as an intermediary between the client and the real object. While the API Facade pattern also acts as an intermediary, its primary purpose is to simplify the interface and encapsulate the complexity of the underlying API, rather than controlling access to it.

Result Interpreter

In the context of AI-driven application development, the Result Interpreter pattern plays a crucial role in bridging the gap between raw data or technical responses and user-understandable insights. This pattern leverages the AI's language understanding capabilities to interpret and process the results returned by functions or API calls, making them more accessible and meaningful to users.

When integrating external services or executing complex functions within an application, the returned results often come in the form of structured data, JSON objects, or technical jargon. While these results contain valuable information, they may not be easily comprehensible to end-users. The Result Interpreter pattern addresses this challenge by employing AI to analyze the results and provide human-friendly explanations, summaries, or key takeaways.

How It Works

The Result Interpreter pattern typically involves the following steps:

1. Execute some sort of query or make an API call to retrieve data or perform a specific task.
2. Capture the response or result returned by the query or API.
3. Pass the raw response or result to an AI component, along with any necessary context or instructions.
4. The AI component analyzes the response or result using its language understanding capabilities, extracting relevant information and identifying key insights.
5. The AI generates an interpretation of the results in either plain text or structured data. That interpretation may include summaries, explanations, or actionable recommendations, or whatever you want it to be.

6. The interpreted results are returned to the user or integrated back into the application flow for further processing or presentation.

When to Use It

The Result Interpreter pattern is particularly useful in scenarios where:

- The application integrates with external APIs or services that return complex or technical responses, especially in formats such as JSON or XML.
- The results of a function or API call require further analysis or explanation to be meaningful to users.
- The raw data requires transformation or mapping to be applicable to your system and/or application domain.
- Your application needs to provide actionable insights or recommendations based on the retrieved data.

What a Result Interpreter does for you in many cases can be done in much more efficient ways with algorithmic approaches and the right 3rd-party libraries and/or frameworks. For example, transformation of JSON can be done with [jq](#), a command-line JSON processing tool. However, you'd have to know how to use jq and you'd have to write the appropriate script. And if the structure of your input changes, your script would possibly break, and need to be rewritten.

In other words, using an AI-based Result Interpreter potentially reduces up-front and maintenance development cost, in exchange for more expensive runtime cost. That particular tradeoff is often worth it for prototype or throwaway code, and extremely appealing for devotees to the Lean Startup philosophy like myself.

For instance, at some point in Olympia's history, I wanted to test whether users would voluntarily sign themselves up for integrations to Google and

Github. Liberal use of the Result Interpreter pattern allowed me to whip up implementations of those integrations in literally hours, after which I deployed them for a few weeks just to see what would happen. I learned a lot, and was able to use the knowledge gained to plan our product roadmap better than I would have otherwise. Would I have run that experiment if the integrations would have taken me days or weeks instead of hours? No way.

Example

Here's an example implementation of the Result Interpreter pattern in Ruby, specifically for summarizing knowledge graph search results from Google:

```
1  class SummarizeKnowledgeGraph
2    include Raix::ChatCompletion
3
4    SYSTEM_PROMPT = <<~END
5      Thoroughly summarize the provided knowledge graph search results from
6      Google, giving special attention to factual data that can be found in or
7      deduced from the search results. Include at least 5-10 URLs from the
8      search results for the user to conduct additional research.
9    END
10
11   def call(results)
12     transcript << { system: SYSTEM_PROMPT }
13     transcript << { user: results }
14     chat_completion
15   end
16
17   def model
18     "anthropic/claude-3-haiku"
19   end
20 end
```

In this example, the `GoogleSearch::SummarizeKnowledgeGraph` class implements the Result Interpreter pattern to summarize knowledge graph search results obtained from Google.

The `SYSTEM_PROMPT` constant defines the instructions for the AI component, specifying the desired format and content of the summary. It requests a thorough summary of the search results, with a focus on factual data and the inclusion of relevant URLs for further research.

The `model` method specifies the AI model to be used for the interpretation task. In this case, it uses the “anthropic/claude-3-haiku” model.

The `call` method takes a `results` string and initiates a chat completion, passing the system prompt and the search results content as the user message. The AI model processes the search results and generates a summary based on the provided instructions.

Virtual Machine

The Virtual Machine pattern is a creative approach to prompt engineering that involves tricking the AI into simulating the execution of pseudo-code or a function. By presenting the AI with a well-defined task and a set of instructions, you can guide it to generate outputs that mimic the behavior of a virtual machine or a specific programming language.

This pattern leverages the AI's ability to simulate code execution for tasks that (for whatever reason) do not work very well with normal chat completion. By carefully writing code for your "virtual machine" and providing clear guidelines, you can direct the AI to perform computations and execute algorithms without having to implement them yourself.

How It Works

The Virtual Machine pattern typically involves the following steps:

1. Define the task or computation you want the AI to perform, such as executing pseudo-code or simulating a function.
2. Create a set of instructions or guidelines that clearly describe the expected behavior and output format.
3. Provide the AI with input data or parameters required.
4. Present the instructions and context to the AI as a prompt, asking it to simulate the execution or computation.
5. Let the AI process the prompt and generate an output.
6. Optionally parse or interpret the AI's response to extract the relevant information or results.

This pattern allows you to tap into the AI's ability to understand and manipulate structured information, making it a powerful tool for various computational tasks.

When to Use It

The Virtual Machine pattern is particularly useful in scenarios where other approaches have failed to consistently give you a concise and precise answer. Most instruct-trained LLMs are tuned to be “helpful assistants” and as such, have a tendency to add commentary before and after the actual content that you’re after. Prompting them with instructions like *just give me the answer and don’t add anything before or after it* does not always work consistently. The virtual machine pattern lets you imply that you want a particular bit of data, and only that data without having to spell it out.

While I use the virtual machine pattern myself as yet another way of *narrowing the path*, you can also experiment with using this pattern as a way of prompting the AI to straight up perform computations or execute algorithms without having access to a real programming environment, or to simulate the behavior of a specific programming language or pseudo-code. Your mileage will definitely vary based on the particular model you’re using, since code reasoning capabilities vary wildly.

Example

Here’s an example implementation of the Virtual Machine pattern in Ruby, inspired by my `Instant18n` ruby gem. I actually use this code in `Olympia` to provide on the fly translation for our user experience:

```

1  class Translator
2    include Raix::ChatCompletion
3    include Raix::PromptDeclarations
4
5    system_prompt <<-END
6      Note that there is no translation database available.
7      Now, simulate execution of the function and respond with
8      your best guess at the return value.
9    END
10
11    prompt text: -> {
12      "translate_for_web_app('#{@language}', '{@string}') returns:"
13    }
14
15    def initialize(language)
16      @language = language
17    end
18
19    def call(string)
20      @string = string
21      chat_completion.to_s.squish.then do |response|
22        response.gsub(/^["']|["']$/, '')
23      end
24    end
25
26    def max_tokens
27      @string.length
28    end
29
30    def model
31      "nousresearch/nous-hermes-llama2-13b"
32    end
33  end

```

In this example, the Translator class demonstrates the Virtual Machine pattern by simulating the execution of a translation function.

The system_prompt provides instructions to the AI, indicating that there is no translation database available and asking it to simulate the execution of the function and respond with its best guess at the return value.

I was forced to tell the AI that no translation database was available because otherwise it raised an exception telling me that no translation database was available, and that it needed it to provide a proper translation.

Now a different solution to the same problem would have been to include initialization of a “translation database in the prompt like this:

```
1 db = TranslationDB.new("#{@language}")
2 db.translate("#{@string}") returns:
```

It might break your brain a little to ponder why this works, until you remember that it's all a simulation anyway.

The prompt method defines the specific function call to be simulated, which I intentionally called `translate_for_web_app` (instead of simply `translate`) to concisely convey to the AI what the purpose of the translation should be.

The `call` method takes a `string` parameter and performs the following steps:

1. Assigns the `string` to the `@string` instance variable.
2. Initiates the chat completion process.
3. Processes the AI's response by converting it to a string, squishing whitespace, and removing any leading or trailing quotes.

If you're curious about how a component like this is used in a production system, here is the code from Olympia that calls it.

First, I created a `Translation ActiveRecord` model to store/cache translations in my database. In addition to caching for performance reasons, this lets me easily whip up a super admin dashboard for being able to manage and manually correct AI-written translations that I'm not happy with.

```

1  # == Schema Information
2  #
3  # Table name: translations
4  #
5  # id          :uuid          not null, primary key
6  # key         :string        not null
7  # lang        :string        not null
8  # result      :string
9  # created_at  :datetime      not null
10 # updated_at  :datetime      not null
11 #
12 # Indexes
13 #
14 # index_translations_on_key      (key)
15 # index_translations_on_lang    (lang)
16 #
17 class Translation < ApplicationRecord
18 end

```

Next I added a helper to my ApplicationHelper module, which for those of you that are not familiar with Rails, is a collection of functions that is mixed into all view templates in your system.

My new helper method is called simply *it*, which stands for *instant translation*.

```

1  module ApplicationHelper
2    def it(key)
3      lang = current_user&.preferred_language || "en"
4      return key if %w[en english en-US].include?(lang.downcase)
5
6      Translation.find_or_create_by(key:, lang:) do |translation|
7        translation.result = Translator.new(lang).call(key)
8      end
9    end
10 end

```

I omitted some error-handling and instrumentation to make the code more readable, but this is basically all that you need. First, I make sure that I don't unnecessarily translate my English input. If translation is needed, then first I try to find a matching translation in the database, and invoke the *Translator* if there isn't one available.

Behind The Magic

It never ceases to amaze me how well large language models (LLMs) can understand and interpret code, even though they weren't specifically designed for that purpose. They can grasp the underlying logic and structure of programming languages, just like they do with human languages.

The reason is that the textual data that LLMs are trained on includes a lot of code snippets and programming-related content. As they process all this information, they start to recognize patterns and learn the syntax and semantics of different programming languages. Kind of like how we humans learn new languages by exposure and practice.

When you present an LLM with a piece of code to “execute”, it draws upon its vast knowledge and understanding of programming concepts to interpret and execute the code in a way that makes sense. It's not actually running the code like a traditional compiler or interpreter would, but rather using its language understanding capabilities to simulate the behavior and generate the expected output.

Of course, LLMs are far from perfect and their ability to acts as a “virtual machine” doesn't always quite work as intended. But overall, their ability to understand and interpret code is pretty impressive and opens up a whole new world of possibilities for developers and programmers alike.

One of the first things I wrote when I realized this capability of LLMs was a magic method, letting me define and execute functions on the fly at runtime.

```

1  ## The `magic` method takes the following parameters:
2  ## `signature`: The method signature including name and parameters.
3  ## `description`: Describes what the function does and what it returns.
4  ## `args`: A list of arguments for the function.
5  def magic(signature:, description:, args:)
6      prompt = <<~END
7      You are the implementation of the following method:
8
9      ## %{description}
10     %{signature}
11
12     I will provide parameters and you will ONLY respond with a single return
13     value of your choice. Never ever include any other explanatory text.
14     END
15
16     transcript << { system: "You are a Ruby virtual machine."}
17     transcript << { user: prompt}
18     transcript << { assistant: "Understood. Waiting for arguments..." }
19     transcript << { user: args.join(", ") }
20
21     chat_completion
22 end

```

As you might imagine, this sort of thing is great for experimentation and prototyping, but is probably not be the best code to include in your production systems.

#

Specification and Testing

When developing AI-powered discrete components, it's crucial to ensure their reliability and correctness through proper specification and testing. The good news is that testing these components can often be more straightforward than testing traditional code components. This is because AI-powered components typically operate as a *black box*, with few dependencies beyond their input parameters.

Specifying the Behavior

Before diving into the implementation of an AI-powered component, it's essential to clearly specify its expected behavior. This involves defining the input parameters, the desired output format, and any specific constraints or requirements. By having a clear specification, you can establish a solid foundation for testing and ensure that the component meets its intended purpose.

When specifying the behavior of an AI-powered component, consider the following aspects:

- **Input Parameters:** Clearly define the types and formats of the input parameters that the component expects. This may include data types, ranges, or any specific validation requirements.
- **Output Format:** Specify the expected format and structure of the output generated by the component. This could be a plain text string, structured data (e.g., JSON), or any other relevant format.
- **Constraints and Requirements:** Identify any specific constraints or requirements that the component must adhere to. This may include character limits, specific terminology or domain knowledge, or any other relevant guidelines.

By documenting these specifications, you create a clear contract for the behavior of the AI-powered component, making it easier to develop and test against those expectations.

Writing Test Cases

With the specification in place, you can proceed to write test cases that verify the component's behavior. Since AI-powered components often act as black boxes, you can focus on testing the input-output relationship without worrying

about the internal implementation details or having to setup a myriad of dependencies.

As for the test cases themselves, I don't know that they are that different than testing anything else, but you probably should consider the following approaches:

- **Input Variations:** Create test cases that cover different variations and edge cases of the input parameters. This may include testing with empty inputs, inputs with special characters, or inputs that push the boundaries of the specified constraints.
- **Expected Output:** Define the expected output for each test case based on the specified behavior. This could be in the form of exact string matches, regular expressions, or structured data comparisons.
- **Boundary Conditions:** Test the component's behavior with input parameters that are at the boundaries of the specified ranges or limits. This helps ensure that the component handles these edge cases gracefully.
- **Error Handling:** Verify that the component handles error conditions appropriately. This may include testing with invalid input parameters or simulating external dependencies that fail.

By covering a wide range of test cases, you can gain confidence in the reliability and correctness of the AI-powered component.

Example: Testing the Translator Component

Let's take a look at an example of how you could write tests for the Translator component that we discussed earlier. Assuming you have a testing framework like RSpec set up in your Ruby project, you can define test cases as follows:

```
1  require "rails_helper"
2
3  RSpec.describe Translator, :novcr do
4    subject { described_class.new("es") }
5
6    describe "#call" do
7      it "translates common words and phrases" do
8        expect(subject.call("Submit")).to eq("Enviar")
9        expect(subject.call("User")).to eq("Usuario")
10       expect(subject.call("Accept Invitation")).to eq("Aceptar la invitación")
11     end
12
13     it "handles empty input" do
14       expect(subject.call("")).to eq("")
15     end
16
17     it "translates sentences" do
18       expect(subject.call("The quick brown fox jumps over the lazy dog.")).
19         .to eq("El rápido zorro marrón salta sobre el perro perezoso.")
20     end
21
22     it "handles special characters" do
23       expect(subject.call("Hello, world! How are you?")).
24         .to eq("¡Hola, mundo! ¿Cómo estás?")
25     end
26   end
27 end
```

In this example, we define a test suite for the `Translator` component using `RSpec`. We create a subject instance of the `Translator` class, specifying the target language as Spanish (“es”).

The test cases cover different aspects of the component’s behavior:

1. Translating common words and phrases: We provide a set of input strings and their expected translations, verifying that the component correctly translates them.
2. Handling empty input: We test the component’s behavior when given an empty string as input, ensuring that it returns an empty string as well.
3. Translating sentences: We provide a complete sentence as input and verify that the component translates it accurately.

4. Handling special characters: We test the component's ability to handle input strings that contain special characters, ensuring that the translation preserves them correctly.

By running these tests, you can verify that the `Translator` component behaves as expected and produces the correct translations for different input scenarios.

Replay of HTTP Interactions

I use a Ruby gem called `vcr` to record my test suite's HTTP interactions and replay them during future test runs for fast, deterministic, accurate tests.

If you have an eagle's eye, you may have noticed the macro `:novcr` in the top-level describe of the `Translator` spec. What it does is to *turn off* the VCR for the duration of the spec, ensuring that HTTP interactions are not recorded. That's great for TDD and exploratory development, but not so great for performance reasons as part of a full test suite, especially in CI/CD environments that might not even have access to keys required for use of your AI models.

Here are the two most relevant configuration settings to add to your `rails_helper.rb` script in order to be able to easily turn the VCR on and off using the method described above.

```
1  require 'vcr'
2  require 'support/vcr'
3
4  RSpec.configure do |config|
5    config.before(:example, :novcr) do
6      VCR.turn_off!
7      WebMock.disable!
8    end
9
10   config.after(:example, :novcr) do
11     VCR.turn_on!
12     WebMock.enable!
13   end
14
15   ...
```

Human In The Loop (HITL)



In the rapidly evolving field of artificial intelligence (AI), the role of human expertise and judgment remains crucial. While AI models have made remarkable strides in automating tasks and making decisions, the foreseeable future is full of scenarios where human intervention is essential. This is where Human In The Loop (HITL) patterns come into play.

HITL patterns focus on designing AI systems that effectively incorporate human input, oversight, and collaboration. These patterns recognize that the best outcomes often result from leveraging the strengths of both humans and AI. By creating workflows and mechanisms that allow humans to interact with AI components at key points, HITL patterns ensure that the AI system benefits from human knowledge, adaptability, and ethical considerations.

The importance of HITL patterns cannot be overstated. They help address the limitations and potential biases of AI models, provide a safety net for critical decisions, and enable continuous improvement through human feedback. HITL

patterns also promote transparency, accountability, and trust in AI systems, as humans remain involved in the decision-making process.

In this chapter, I first introduce some high level patterns that may be useful to you when you're in the early stages of defining what a new system will do in broad strokes. The rest of the chapter contains more concrete HITL patterns that have proven personally useful in real-world AI applications like Olympia.

We will start those key patterns with Escalation, which focuses on detecting situations that require human intervention and triggering appropriate escalation processes. Next, we will look at the Feedback Loop pattern, a powerful technique for iteratively refining AI outputs based on human input.

We will then examine the Passive Information Radiation pattern, which aims to provide humans with relevant insights and information without requiring explicit intervention. The Collaborative Decision Making pattern will showcase how humans and AI can work together to make better decisions by leveraging their complementary strengths.

Finally, we will explore the Continuous Learning pattern, which emphasizes the importance of designing AI systems that continuously learn and adapt based on human feedback and interactions.

Before closing, I'll discuss a bit more about ethical considerations and future outlook for HITL systems. My hope is that by the end of this chapter, you will have a solid understanding of HITL patterns and how they can be applied to create AI systems that are more effective, reliable, and aligned with human values.

High-Level Patterns

Human In The Loop (HITL) features a few patterns that transcend basic interactions, and are a little broader than other implementation patterns that I have been presenting in Part 2 of the book. These high-level patterns focus on

creating more nuanced, dynamic, and effective collaborations between humans and AI and may be useful terminology for you to use when you're first thinking about the broad strokes of the system you want to build.

Hybrid Intelligence

Hybrid Intelligence is a collaborative pattern that harnesses the strengths of both human and artificial intelligence to tackle complex problems that neither could solve alone. This approach optimizes the unique capabilities of both entities—AI's ability to process and analyze large datasets rapidly and humans' ability to think critically and creatively.

Application Example:

In financial markets, Hybrid Intelligence systems enable real-time decision-making. AI algorithms analyze market data for immediate trends and anomalies while human experts focus on strategic adjustments based on broader economic indicators and geopolitical events. This combination allows for rapid, informed responses to market changes, significantly enhancing trading strategies.

Benefits:

- **Synergy:** Combines the quick data processing of AI with the strategic depth of human insight.
- **Flexibility:** Adapts to various scenarios, maximizing the strengths of both participants.
- **Enhanced Outcomes:** Produces superior results in complexity management and problem-solving.

Adaptive Response

The Adaptive Response pattern dynamically modulates the level of human intervention in AI-driven processes based on the specific needs of each task. By

assessing factors such as the complexity of the situation, the stakes involved, and the AI's confidence level, this pattern ensures that human expertise is utilized when most effective, optimizing both resource use and outcome quality.

Application Example:

In customer support systems, AI can handle routine inquiries such as password resets or status updates, which are escalated to human agents only when complex issues arise. This not only speeds up response times but also ensures that customers receive a human touch when needed, improving overall service quality.

Benefits:

- **Resource Efficiency:** Ensures optimal use of human resources, engaging them only when necessary.
- **Scalability:** Allows systems to handle larger volumes or more complex tasks without linear increases in human labor.
- **Precision:** Tailors the level of human input to the complexity and nuances of each task.

Human-AI Role Switching

Human-AI Role Switching involves fluid changes in roles between AI and humans depending on the task's context and requirements. This pattern is particularly useful in environments where the optimal approach may change over time or across different stages of a workflow.

Application Example:

In healthcare, an AI system might initially analyze diagnostic tests to narrow down potential conditions based on symptoms and medical images. A human doctor then reviews these suggestions to make a final diagnosis and develop a treatment plan. For ongoing patient monitoring, the role switches back to AI, which can alert medical staff only if abnormalities are detected.

Benefits:

- **Flexibility and Adaptability:** Enables systems to adapt their approach dynamically, maximizing efficiency and effectiveness.
- **Continuous Coverage:** AI can handle routine or repetitive tasks, freeing up humans for complex or creative problem-solving.
- **Enhanced Learning Opportunities:** Each party learns from the other, improving both AI algorithms and human decision-making over time.

These high-level patterns represent the cutting edge of HITL development, providing frameworks that not only enhance the capabilities of AI systems but also integrate human expertise in a way that is both scalable and effective. Implementing these patterns can lead to significant advancements in AI applications, driving innovation and efficiency across industries.

Escalation

The Escalation pattern is a crucial HITL pattern that focuses on designing mechanisms to automatically detect situations where human intervention is required and trigger an appropriate escalation process. This pattern becomes particularly relevant when dealing with complex or high-stakes decisions that necessitate human judgment, or when the AI model encounters scenarios it is not confident in handling autonomously.

How It Works

The key components of the Escalation pattern are as follows:

1. **Defining Escalation Criteria:** The first step is to establish clear and well-defined criteria or thresholds that determine when a task or decision should be escalated to a human. These criteria can be based on various factors, such as confidence scores, risk levels, or domain-specific rules. For example, in a financial application, any transaction above a certain amount or with a high fraud risk score may require human review.
2. **Implementing Escalation Triggers:** Once the escalation criteria are defined, the next step is to develop mechanisms within the AI system to continuously monitor and evaluate against these criteria. When the predefined conditions are met, the system should automatically trigger an escalation process. This could involve generating alerts, creating tickets in a task management system, or notifying the appropriate human experts.
3. **Notification and Task Assignment:** Upon triggering an escalation, the AI system should promptly notify the designated human experts or reviewers and assign the task to them. This ensures that the escalated case receives timely attention and is routed to the individuals with the necessary skills and authority to handle it. Clear communication channels and well-

defined roles and responsibilities are essential for effective escalation handling.

4. **Human Review and Decision Making:** Once the escalated task is assigned, the human experts review the available information, analyze the situation, and make an informed decision or provide guidance. They may have access to additional context, historical data, or domain-specific tools to aid in their review process. The human input is then fed back into the AI system to proceed with the appropriate course of action.
5. **Feedback and Model Improvement:** The human feedback and decisions obtained during the escalation process serve as valuable data points for improving the AI model's performance. By capturing and analyzing this feedback, the system can learn from human expertise, refine its decision-making capabilities, and gradually reduce the need for future escalations. Regular model updates and retraining based on escalation data help enhance the system's accuracy and reliability over time.

Key Benefits

Implementing the Escalation pattern brings several benefits to AI systems. It provides a safety net for handling complex or high-stakes situations, ensuring that critical decisions are made with human oversight. It helps maintain the overall quality and reliability of the system by leveraging human expertise when needed. Additionally, it fosters trust and confidence in the AI system, as users know that there are mechanisms in place to handle exceptional cases and involve human judgment when necessary.

However, it's important to strike the right balance when designing escalation criteria and triggers. Overly sensitive triggers may lead to frequent escalations, causing inefficiencies and burdening human experts. On the other hand, overly relaxed criteria may miss important cases that require human attention. Careful consideration and iterative refinement of escalation parameters are crucial for optimal performance.

In summary, the Escalation pattern is a vital HITL pattern that enables AI systems to automatically detect and escalate situations requiring human intervention. By defining clear criteria, implementing triggers, and facilitating human review and decision-making, this pattern ensures the quality, reliability, and trustworthiness of AI systems, while leveraging the strengths of both human expertise and machine intelligence.

Real-World Application: Escalation Pattern in Healthcare

A practical application of the Escalation pattern is found in remote patient monitoring systems. These systems use AI to analyze patient data continuously. When anomalies are detected—such as a sudden drop in a patient's heart rate—the system automatically escalates the case to a human clinician who reviews the data and may intervene by contacting the patient or adjusting their treatment plan. This real-time intervention can be lifesaving, showcasing the importance of the Escalation pattern in critical healthcare scenarios.

Feedback Loop

In the realm of prompt engineering, crafting the perfect prompt often requires iteration and refinement. The Feedback Loop pattern (more of a technique, really) is a powerful approach that leverages the language model's own output to progressively improve the quality and relevance of the generated content. By engaging in a iterative process of generating, reviewing, and refining prompts, you can guide the model towards producing more accurate and coherent responses.

At its core, the Feedback Loop technique involves using the model's output as input for the next iteration of prompt generation. The process begins with an initial prompt that describes the desired output or task. This prompt is fed into the language model, which generates an initial response. Next you review the generated response and identify areas for improvement or refinement. Based on this analysis, the prompt is modified by incorporating the previous response, along with any necessary adjustments, clarifications, or constraints. The updated prompt is then fed back into the model, triggering the generation of a new response. This loop continues until the desired quality or specificity of the output is achieved.

Do enough prompt engineering and eventually when you see the phrase “identify areas for improvement or refinement” you will think to yourself, I’ll just ask the AI to do that for me.

The Feedback Loop technique is particularly useful in scenarios where the desired output requires multiple iterations or progressive refinement. For example, in content generation tasks, such as writing a story or an article, the Feedback Loop allows the model to build upon its previous outputs, gradually shaping the narrative and improving the coherence and relevance of the

generated text. By providing feedback and guidance at each iteration, the developer can steer the model towards producing content that aligns with their intended goals.

Another common application of the Feedback Loop technique is in progressive summarization. When tasked with summarizing a lengthy piece of text, the model can generate an initial summary based on the provided prompt. The developer can then feed this summary back into the model, along with additional instructions to further condense or refine the summary. Through multiple iterations, the model can produce increasingly concise and focused summaries that capture the essential information from the original text.



Summarization is one of those areas of AI usage that really benefits from experimentation. For instance, instead of using a Feedback Loop to get the AI to make your summary progressively smaller, try using it to make your summary progressively *bigger* by telling it to add more details each time around. In my experience it works better because language models love to be stingy with their summaries.

The Feedback Loop technique is valuable in contextual question-answering scenarios, where the model engages in a multi-turn conversation with the user. By incorporating the previous context and the user's follow-up questions into the prompt, the model can generate more accurate and relevant responses. Each iteration of the Feedback Loop allows the model to build upon the established context, leading to a more coherent and meaningful dialogue.

Note that the steps in your feedback loop do not have to stay as they are within messages in the conversation transcript. Each time you cycle through the loop, you can take the results and “rewrite history,” liberally changing the transcript or trashing it altogether.

How It Works

Implementing the Feedback Loop technique in your application requires careful consideration of the specific requirements and constraints of your use case. It involves designing methods or functions that handle the iterative prompt refinement process, including generating responses, evaluating the output against predefined criteria, and modifying the prompt based on the feedback. The effectiveness of the Feedback Loop technique depends on the quality of the initial prompt, the clarity of the refinement criteria, and the capabilities of the language model being used.

In automated scenarios, be mindful of the potential for the model to diverge or generate inconsistent outputs if the feedback loop is not properly controlled. Setting appropriate stopping criteria in code outside of the language models control, such as a maximum number of iterations or a quality threshold, can help prevent endless loops or suboptimal results.

Despite its challenges, the Feedback Loop technique offers a powerful way to leverage the generative capabilities of language models for iterative prompt refinement. By engaging in a dialogue-like process with the model, developers can guide the generation of more accurate, coherent, and relevant content. Whether applied to content creation, summarization, or contextual question-answering, the Feedback Loop technique adds a layer of interactivity and adaptability to prompt engineering, enabling developers to unlock the full potential of language models in their applications.

Applications and Examples

The Feedback Loop pattern finds applications across various domains where iterative refinement and human input are valuable. Let's explore a few examples to illustrate its practical usage.

1. **Content Generation:** In content creation tasks, such as writing articles, stories, or product descriptions, the Feedback Loop technique can be highly effective. The process starts with an initial prompt that outlines the desired content, such as the topic, target audience, and key points to cover. The language model generates a draft based on this prompt. The human reviewer then provides feedback, suggesting improvements, identifying missing information, or highlighting areas that need refinement. This feedback is incorporated into the prompt, and the model generates an updated version. The loop continues until the content meets the desired quality and requirements. By iteratively refining the content based on human input, the final output is more coherent, informative, and aligned with the intended goals.
2. **Progressive Summarization:** Summarizing lengthy documents or articles is another scenario where the Feedback Loop pattern shines. The process begins with an initial prompt that includes the original text and instructions for summarization, such as the desired length or focus areas. The model generates a summary based on this prompt. The human reviewer then assesses the summary and provides feedback, indicating areas that need further condensation, highlighting key points that should be emphasized, or suggesting improvements in clarity or coherence. The feedback is incorporated into the prompt, and the model generates a refined summary. Through multiple iterations, the summary becomes more concise, accurate, and captures the essential information from the original text. The Feedback Loop allows for progressive refinement, ensuring that the final summary meets the desired quality and brevity.
3. **Contextual Question-Answering:** In applications where AI systems engage in multi-turn conversations or provide answers to user queries, the Feedback Loop pattern can enhance the relevance and accuracy of the responses. The process starts with an initial prompt that includes the user's question and any relevant context. The model generates a response based on this prompt. The human reviewer then evaluates the response

and provides feedback, indicating if the answer is satisfactory, if additional information is needed, or if there are any inaccuracies or inconsistencies. The feedback is incorporated into the prompt, along with the previous context, and the model generates an updated response. Through iterative refinement, the model learns to provide more accurate and contextually relevant answers, taking into account the user's follow-up questions and the established context.

These examples demonstrate the versatility and value of the Feedback Loop pattern in various applications. By leveraging human input and iteratively refining the output, the pattern enables the creation of higher-quality content, more accurate summaries, and more relevant responses in conversational AI systems. The Feedback Loop technique empowers developers to guide the language model towards producing results that align with their specific requirements and user expectations.

It's important to note that the effectiveness of the Feedback Loop pattern relies on the quality of human feedback and the clarity of the refinement criteria. Providing constructive and specific feedback is crucial for guiding the model towards the desired output. Additionally, setting appropriate stopping conditions, such as a maximum number of iterations or a quality threshold, helps prevent endless loops and ensures the timely delivery of the final result.

Advanced Techniques in Human Feedback Integration

Incorporating human feedback into AI systems can be enhanced through advanced techniques such as:

- **Incremental Learning:** This approach allows the AI system to learn continuously from each piece of feedback without the need for retraining the entire model. It's particularly useful in dynamic environments where conditions change rapidly.

- **Confidence-Based Handling:** AI systems can be designed to request human feedback based on confidence thresholds. If the AI's confidence in its decision falls below a certain level, it automatically triggers a review by a human expert.
- **Contextual Relevance Feedback:** Employ techniques that allow users to provide feedback not only on the output quality but also on its relevance to specific contexts or scenarios, enhancing the system's ability to adapt its responses to similar future inquiries.

These techniques help create a more responsive and adaptable AI system, ensuring that it remains effective as it encounters new data and scenarios.

Passive Information Radiation

The Passive Information Radiation pattern focuses on designing AI systems that provide humans with relevant information and insights without explicitly requiring their intervention. This pattern aims to empower humans with the knowledge they need to make informed decisions or take appropriate actions.

How It Works

Key elements of the Passive Information Radiation pattern include:

Contextual Information Display

The AI system should present relevant information to humans in a clear, concise, and easily digestible format. This could involve displaying key metrics, visualizations, or summarized insights within the user interface or through reporting mechanisms. Use data visualization techniques, such as charts, graphs, or dashboards, to convey complex information effectively. Make sure to prioritize the most critical or actionable information and make it readily accessible to users.

Finally, don't forget to provide context and explanations where needed to help users interpret the displayed information accurately.

Proactive Notifications

The system can proactively notify humans about important events, anomalies, or patterns detected by the AI model. These notifications serve as prompts for humans to review and take action if necessary.

- Determine the appropriate channels for delivering notifications, such as email, SMS, or in-app alerts.

- Establish notification thresholds and frequencies to avoid overwhelming users with too many alerts.
- Provide clear and actionable information in the notifications, along with guidance on next steps or recommended actions.

Explanatory Insights

The AI system should provide explanations or interpretations of its outputs or decisions. This could include highlighting the key factors influencing a prediction, visualizing the model's reasoning process, or generating human-readable reports. Consider using natural language generation or template-based approaches to create clear and concise explanations.

Interactive Exploration

Provide tools or interfaces that allow humans to explore and interact with the AI system's outputs or underlying data. This could include drill-down capabilities, filtering options, or the ability to ask follow-up questions to gain deeper insights.

Key Benefits

A well designed implementation of the Passive Information Radiation pattern should empower users to access relevant information, gain insights, and make informed decisions without requiring explicit intervention. This pattern promotes transparency by providing users with visibility into the AI system's outputs and reasoning process. It also fosters trust by enabling users to verify and validate the AI system's decisions or recommendations.

Moreover, the Passive Information Radiation Pattern facilitates collaboration between humans and AI by creating a shared understanding of the system's behavior and outputs. It enables users to provide feedback, validate results, and contribute their domain expertise to refine and improve the AI system over time.

Applications and Examples

The Passive Information Radiation Pattern finds applications in various domains where providing humans with relevant information and insights is crucial for decision-making and collaboration.

However, it's important to strike a balance between providing sufficient information and avoiding information overload. The information displayed should be carefully curated and presented in a way that aligns with users' needs and preferences. Regular user feedback and usability testing can help optimize the information radiation process and ensure its effectiveness.

Let's explore a few examples to illustrate its practical usage.

1. **Healthcare:** In healthcare systems, the Passive Information Radiation Pattern can be applied to support clinical decision-making and patient monitoring. For example, an AI-powered clinical decision support system can display relevant patient information, such as medical history, vital signs, and lab results, in a contextualized manner to assist healthcare professionals in making informed decisions. The system can proactively notify physicians about potential adverse drug interactions, abnormal test results, or critical changes in a patient's condition. Explanatory insights can be provided to highlight the key factors contributing to a diagnosis or treatment recommendation. Interactive exploration capabilities allow healthcare professionals to drill down into specific aspects of a patient's data, enabling them to gain deeper insights and make data-driven decisions.
2. **Finance:** In the financial domain, the Passive Information Radiation Pattern can be employed to support risk assessment, fraud detection, and investment decision-making. For instance, an AI-powered risk assessment system can display relevant financial data, such as credit scores, transaction history, and market trends, to assist analysts in evaluating potential

risks. The system can proactively notify analysts about suspicious activities, anomalies, or significant changes in market conditions. Explanatory insights can be provided to highlight the key factors influencing a risk score or investment recommendation. Interactive exploration features allow analysts to investigate specific transactions, compare historical data, and perform what-if analyses to make informed decisions.

3. **Manufacturing:** In manufacturing settings, the Passive Information Radiation Pattern can be applied to optimize production processes and ensure quality control. For example, an AI-powered manufacturing monitoring system can display real-time production metrics, such as machine performance, inventory levels, and quality indicators, to provide operators with a comprehensive view of the production process. The system can proactively notify operators about potential equipment failures, quality deviations, or supply chain disruptions. Explanatory insights can be provided to highlight the root causes of production issues or inefficiencies. Interactive exploration capabilities allow operators to analyze historical production data, identify trends, and make data-driven decisions to optimize production processes and improve product quality.

These examples demonstrate how the Passive Information Radiation Pattern can be applied in different domains to provide humans with relevant information and insights. By implementing contextual information display, proactive notifications, explanatory insights, and interactive exploration, this pattern enables users to make informed decisions, identify potential issues, and collaborate effectively with AI systems.

The specific implementation of the Passive Information Radiation Pattern may vary depending on the domain and the unique requirements of each use case. It's important to tailor the information display, notification mechanisms, explanatory techniques, and interactive features to align with the needs and preferences of the target users.

By leveraging the Passive Information Radiation Pattern, organizations can harness the power of AI to provide humans with valuable insights and support their decision-making processes. This pattern promotes a symbiotic relationship between humans and AI, enabling them to work together effectively and drive better outcomes in various domains.

Collaborative Decision Making (CDM)

The Collaborative Decision Making (CDM) pattern focuses on designing AI systems that actively involve humans in the decision-making process. This pattern recognizes that certain decisions benefit from the combined intelligence of humans and AI, leveraging their complementary strengths.

Key aspects of the Collaborative Decision Making pattern include:

1. **Shared Decision Spaces:** Create interfaces or platforms where humans and AI can work together on decision-making tasks. This could involve shared workspaces, collaborative tools, or decision support systems.
2. **Iterative Refinement:** Enable an iterative process where the AI system generates initial recommendations or solutions, and humans provide feedback, adjustments, or additional constraints. The AI system then incorporates this feedback to refine its outputs, and the process continues until a satisfactory decision is reached.
3. **Explanation and Justification:** The AI system should provide explanations and justifications for its recommendations or decisions. This transparency helps humans understand the AI's reasoning and enables them to provide more informed feedback or challenge the AI's outputs when necessary.
4. **Consensus Mechanisms:** Implement mechanisms for reaching consensus between human experts and the AI-based experts. This could involve voting systems, weighted decision-making, or other techniques that ensure both human judgment and AI insights are considered in the final decision.

How It Works

The Collaborative Decision Making Pattern involves a structured process that enables humans and AI to work together in making decisions. Here's a step-by-step explanation of how this pattern typically works:

1. **Problem Definition:** The first step is to clearly define the decision-making problem or task at hand. This involves specifying the objectives, constraints, and any relevant context or background information. The problem definition serves as a shared understanding between the human experts and the AI system.



Working in a domain with complex or “fuzzy” problems? Use the power of LLM-based reasoning to help you fully define the problem or task at hand.

2. **Data Preparation:** The next step is to gather and prepare the necessary data that will inform the decision-making process. Consider whether a human will control what subsets of the data will be inserted into the prompts, versus using a [RAG](#) approach. The prep may involve collecting relevant datasets, preprocessing the data, and ensuring its quality and completeness. The data should be structured in a format that is suitable for *both* human analysis and AI processing.



Using RAG in your AI-assisted decision making process? Make sure that it's configured to expose the results of its retrieval and/or provide citations, so that the human collaborator can verify the AI's reasoning.

3. **Prompt Development:** Based on the problem definition and available data, develop prompts that will assist in the CDM process. Validate the prompts using example scenarios and prepared data to ensure their reliability.

4. **Collaborative Interface:** The user experience should be designed in such a way that facilitates the interaction between human experts and the AI system. The user interface should provide a friendly environment where humans can input their knowledge, insights, and preferences, and where the AI system can present its recommendations or solutions.

5. **Iterative Decision Making:** The CDM process might begin with the AI system generating initial recommendations or solutions based on the available data. These recommendations are presented to the human experts through the collaborative interface. The human experts review the AI-generated recommendations and provide their feedback, insights, or additional constraints. They may agree with the recommendations, suggest modifications, or propose alternative solutions based on their domain expertise.
6. **Refinement and Consensus:** The AI system takes the human feedback into account and refines its recommendations accordingly. This iterative process continues, with the AI system generating updated recommendations based on the human input, and the human experts reviewing and providing further feedback. The goal is to reach a consensus or a satisfactory decision that incorporates the collective intelligence of both the human experts and the AI system.
7. **Decision Implementation:** Once a final decision is reached through the collaborative process, it is implemented and logged. Optionally, the CDM system can trigger processes that put the decision into action. The decision may involve executing a specific plan, allocating resources, or triggering downstream processes. The collaborative interface may provide tools or integrations to facilitate the seamless implementation of the decision.
8. **Monitoring and Evaluation:** After the decision is implemented, it is important to monitor its outcomes and evaluate its effectiveness. The collaborative interface should provide mechanisms for tracking the progress, measuring the results, and gathering feedback from relevant stakeholders. This monitoring and evaluation phase helps in assessing the quality of the decision and identifying any areas for improvement.
9. **Continuous Learning:** The collaborative decision-making process provides valuable data points and insights that can be used to continuously improve the AI model and the overall decision-making framework. The feedback, outcomes, and lessons learned from each decision-making

instance are captured and used to refine prompts used in the CDM system, ensuring its ongoing accuracy and relevance.

By following these steps, the Collaborative Decision Making Pattern enables humans and AI to work together effectively, leveraging their complementary strengths to make informed and consensual decisions. The pattern promotes a synergistic relationship between human expertise and AI capabilities, leading to improved decision quality and outcomes.

Example

Let's consider an example scenario where the Collaborative Decision Making (CDM) pattern is applied in a manufacturing company for optimizing production planning and resource allocation.

The company has a complex manufacturing process that involves multiple production lines, a wide range of products, and varying customer demands. The objective is to create an optimal production plan that maximizes efficiency, minimizes costs, and meets customer requirements.

To address this challenge, the company implements a CDM system. They start by defining the problem and gathering relevant data, such as historical production data, inventory levels, customer orders, and resource constraints.

A collection of prompts are developed with access to tools that give them the power of optimization algorithms and machine learning to analyze production data and generate initial production plan recommendations. The model takes into account factors such as production capacities, lead times, and resource availability to create an optimized plan.

The company sets up a collaborative interface that allows human experts, including production managers, supply chain specialists, and customer representatives, to interact with the AI system. The AI-generated production plan is presented to the human experts through this interface.

The human experts review the AI-generated plan and provide their insights via a [Feedback Loop](#). They may identify potential bottlenecks, suggest alternative resource allocations, or highlight specific customer requirements that need to be prioritized. The AI system takes this feedback into account and refines the production plan accordingly.

Through iterative cycles of AI recommendations and human feedback, the collaborative decision-making process continues until a consensus is reached on the optimal production plan. The agreed-upon plan is then implemented, and the manufacturing operations are executed accordingly.

The collaborative interface provides real-time monitoring capabilities via [Passive Information Radiation](#), allowing the team to track the progress of the production plan, monitor key performance indicators, and identify any deviations or issues. This monitoring helps in assessing the effectiveness of the decision and making necessary adjustments.

Throughout the process, the feedback and outcomes are captured and used for [Continuous Learning](#). The system learns from the human insights and adapts its recommendations based on the real-world results, ensuring its ongoing accuracy and relevance.

By applying the Collaborative Decision Making pattern, the manufacturing company achieves several benefits. The collaborative approach ensures that the production plan takes into account both the AI-generated optimizations and the human expertise, resulting in a more comprehensive and practical solution. The iterative process allows for the incorporation of diverse perspectives and the ability to adapt to changing circumstances.

The pattern enables faster and more informed decision-making, as the AI system can process vast amounts of data and generate recommendations quickly, while the human experts provide strategic guidance and domain knowledge. The collaborative approach also promotes transparency, trust, and buy-in among stakeholders, as they actively participate in shaping the decision.

Overall, the Collaborative Decision Making pattern empowers the manufacturing company to make optimized production planning decisions that balance efficiency, cost-effectiveness, and customer satisfaction. The synergy between human expertise and AI capabilities leads to improved operational performance and a competitive edge in the market.

Continuous Learning

The Continuous Learning pattern is a powerful approach to designing AI systems that continuously learn and adapt based on human feedback and interactions. This pattern recognizes the importance of ongoing learning and the value of human input in shaping the behavior and performance of AI models over time. By incorporating mechanisms for capturing and leveraging human feedback, the Continuous Learning Pattern enables AI systems to evolve and improve their outputs, leading to more accurate, relevant, and user-centric results.

How It Works

The Continuous Learning pattern involves several key components and processes that enable AI systems to learn and adapt based on human feedback. Here's how it typically works:

- 1. Interaction and Feedback Collection:** The AI system provides outputs or generates responses based on its current model and knowledge. Outputs include more than just conversational replies, they could also be results of a [Collaborative Decision Making](#) system or information displayed via [Passive Information Radiation](#). Users are provided with affordances giving them the power to give feedback on the quality, relevance, or usefulness of the outputs. This feedback can be explicit, such as ratings, comments, or annotations, or implicit, such as user engagement metrics or behavioral data.
- 2. Feedback Storage and Processing:** The collected feedback is stored in a structured format, along with relevant metadata such as timestamps, user profiles, and context information. The feedback data is preprocessed and transformed into a suitable format for analysis and integration into the learning process. The feedback can also be post-processed and used as

source material for the current conversational context or stored in the source material repository of a [RAG](#) system.

3. **Feedback Analysis and Aggregation:** The stored feedback can also be analyzed to identify patterns, trends, and insights. Various techniques, such as sentiment analysis, clustering, or statistical modeling, can be applied to extract meaningful information from the feedback data. The feedback is aggregated and summarized to provide a consolidated view of user preferences and opinions.
4. **Model Update and Refinement:** Based on the analyzed feedback, the AI system is updated and refined. This will primarily involve refinement of prompts for most applications, but may also involve adjusting inference hyperparameters, updating data used for [RAG](#), or even fine-tuning the underlying model. The goal is to incorporate the insights gained from user feedback to improve the model's performance and align its outputs with user expectations.
5. **Iterative Learning and Adaptation:** The process of collecting feedback, analyzing it, and updating the model is repeated iteratively. As the AI system continues to receive feedback from users, it continuously learns and adapts its behavior. Over time, the system becomes more accurate, relevant, and attuned to user needs and preferences.
6. **Monitoring and Evaluation:** The performance of the AI system is continuously monitored and evaluated to assess the effectiveness of the learning process. Key metrics, such as user satisfaction, engagement, or task completion rates, are tracked to measure the impact of the continuous learning approach, perhaps using [Passive Information Radiation](#). Regular evaluations help identify areas for further improvement and ensure the system remains aligned with user expectations.

Applications and Examples

The Continuous Learning pattern finds applications in various domains where AI systems can benefit from ongoing user feedback and adaptation. Some common applications include:

1. **Recommendation Systems:** In e-commerce or content platforms, continuous learning can be applied to improve the accuracy and relevance of product or content recommendations. User feedback, such as ratings, purchases, or engagement, can be used to refine the recommendation algorithms and provide more personalized suggestions.
2. **Chatbots and Virtual Assistants:** Conversational AI systems can leverage continuous learning to enhance their natural language understanding and response generation capabilities. User feedback, such as ratings or corrections, can be used to improve the chatbot's ability to interpret user intents, provide accurate responses, and handle a wider range of queries.
3. **Predictive Maintenance:** In industrial settings, continuous learning can be applied to predictive maintenance systems. User feedback, such as maintenance logs or sensor data, can be used to refine the predictive models and improve the accuracy of failure predictions, leading to optimized maintenance schedules and reduced downtime.
4. **Fraud Detection:** In financial services or security systems, continuous learning can help improve the accuracy and adaptability of fraud detection models. User feedback, such as confirmed fraudulent transactions or false positives, can be used to update the models and enhance their ability to identify and prevent fraudulent activities.



It's likely that most implementations of [Human In The Loop](#) should include Continuous Learning.

Example

Let's consider an example of applying the Continuous Learning Pattern in a conversational AI system that generates responses using a large language model (LLM). The system allows users to interact with the LLM by providing prompts and receiving generated responses. To improve the quality and relevance of the responses, the system incorporates a feedback mechanism where users can upvote or downvote the generated responses.

Here's how the Continuous Learning Pattern can be implemented in this scenario:

1. **User Interaction:** Users interact with the conversational AI system by providing prompts or questions. The LLM generates responses based on its pre-trained knowledge and the given prompt.
2. **Feedback Collection:** After receiving a generated response, users have the option to upvote or downvote the response based on its quality, relevance, or helpfulness. The UX affordance for voting is typically rendered as a thumbs-up and thumbs-down icon. Clicking one of those pops up a dialog-based input where users can also provide additional comments or suggestions for improvement.
3. **Feedback Storage:** The user feedback, including the upvotes, downvotes, and comments, is stored in a database along with the corresponding prompts and generated responses. The feedback data is associated with relevant metadata, such as user profiles and timestamps. The data can be stored raw, or post-processed by an LLM for purposes of normalization and/or added context based on the conversation transcript.
4. **Feedback Analysis:** The collected feedback data is analyzed to identify patterns and insights. The system calculates metrics such as the upvote-to-downvote ratio, average rating, or sentiment scores for each generated response. It also analyzes the comments to extract common themes or

suggestions. Results are made available for analysis by the humans that maintain the system.

5. **Model Refinement (Optional):** The feedback data can optionally be made available to the conversational AI system for use in refining the quality of its generated responses. The simplest way to implement this is simply to insert the feedback data into the conversation transcript at the point where it occurred. If conversation transcripts including feedback data are archived for use in [RAG](#) like we do at Olympia, then it has the net effect of producing *iterative learning*.
6. **Iterative Learning:** The process of collecting feedback, analyzing it, and refining the model is repeated continuously as users interact with the system. Over time, the LLM learns to generate responses that are more aligned with user preferences and expectations. The system becomes more effective in understanding user intents and providing relevant and helpful responses.
7. **Monitoring and Evaluation:** The performance of the conversational AI system is monitored by tracking metrics such as the upvote-to-downvote ratio, user engagement, or conversation completion rates. Regular evaluations are conducted to assess the impact of the continuous learning approach and identify areas for further improvement.

By applying the Continuous Learning Pattern, the conversational AI system can continuously learn and adapt based on user feedback. The upvoting and downvoting mechanism allows users to directly influence the quality of the generated responses, leading to a more user-centric and effective conversational experience. The system becomes more attuned to user preferences and can provide more accurate, relevant, and helpful responses over time.

This example demonstrates how the Continuous Learning Pattern can be applied to enhance the performance and user satisfaction of a conversational AI system. By leveraging user feedback and iterative refinement, the system

can continuously improve its language understanding and response generation capabilities, ultimately providing a better user experience.

—

Ethical Considerations

While implementing HITL systems, it is crucial to address ethical considerations to ensure these technologies are used responsibly. Key areas include:

- **Accountability:** Clear protocols should be established to determine who is responsible for decisions made with AI involvement. This is crucial in sectors like healthcare and finance, where decisions significantly impact human lives.
- **Privacy:** HITL systems often handle sensitive data. Ensuring that human reviewers respect privacy norms and that data handling complies with regulations like GDPR is essential.

#

- **Bias Mitigation:** Human biases can influence AI systems, especially in training phases or decision-making processes. Regular audits and bias mitigation strategies must be integrated to ensure fairness and equity in outputs.

Incorporating these considerations into system design and operation not only enhances trust in AI applications but also ensures compliance with ethical standards and legal requirements.

Role of HITL in Mitigating AI Risks

Human oversight is crucial in mitigating risks associated with AI deployments, particularly in sectors such as healthcare, automotive, and financial services.

Key aspects include:

- **Error Reduction:** Human oversight can catch and correct errors that AI systems may not recognize, particularly those involving nuanced or exceptional circumstances.
- **Safety Assurance:** In critical applications like autonomous driving or medical diagnostics, HITL processes ensure that decisions are verified by humans, greatly enhancing safety.
- **Compliance and Governance:** Human involvement helps ensure that AI systems operate within regulatory frameworks, addressing legal and ethical standards crucial for industry compliance.

Understanding and implementing effective HITL configurations can significantly reduce the risks associated with automated decision-making processes.

Technological Advancements and Future Outlook

Recent advancements in AI, such as the development of more sophisticated machine learning models and enhanced natural language processing capabilities, are expanding the possibilities for HITL systems. Future trends likely include:

- **Greater Autonomy:** As AI becomes more capable, the role of humans may shift more towards oversight and strategic decisions, reducing the need for routine intervention.
- **Enhanced Interaction:** Improvements in AI's ability to understand and generate human-like language will facilitate more natural interactions between humans and AI systems, enhancing collaborative decision-making processes. We may witness a shift from text-based interactions to more natural and conversational interactions, reminiscent of how Marvel's Iron Man character interacts with his assistant JARVIS.

- **Predictive Human In The Loop:** All kinds of AI components (not just those that are intentionally designed with HITL functionality) might gain the ability to predict when human intervention is needed before a situation becomes critical, enhancing the robustness of entire systems.

Understanding these trends will help organizations anticipate changes in HITL system design and leverage these technologies to gain competitive advantages.

Challenges and Limitations of HITL Systems

While HITL systems offer numerous benefits, they also come with challenges that need careful management:

- **Scalability Issues:** Human intervention can limit the scalability of AI systems, as increasing throughput typically requires proportional increases in human resources.
- **Human Error:** The effectiveness of HITL systems is partly contingent on human performance, which can vary significantly and may introduce errors.
- **Cost Implications:** Maintaining a workforce capable of providing timely and accurate feedback can be costly, especially as system complexity and scale increase.

Addressing these challenges requires a balanced approach that leverages technological advancements while ensuring robust human oversight mechanisms.

Intelligent Error Handling



Error handling is a critical aspect of any robust and reliable software application. It ensures that applications can gracefully handle unexpected situations, recover from failures, and provide a smooth user experience. Effective error handling involves anticipating potential issues, implementing appropriate error detection and reporting mechanisms, and defining strategies for error recovery and resolution.

Traditional error handling approaches often rely on manual intervention, static rules, and predefined error messages. At worst, exception logging systems accumulate error notices that we end up ignoring for days or weeks at time. Traditional error handling approaches can be time-consuming, inflexible, and may not adequately address the complexity and diversity of real-world error scenarios. As applications become more sophisticated and user expectations rise, there is a growing need for more intelligent and adaptive error handling techniques.

Luckily, AI can revolutionize error handling in our software applications. By leveraging AI techniques and patterns, we can create intelligent error handling mechanisms that are more efficient, user-friendly, and adaptable to various error scenarios.

Note that at time I'm writing this book, many of the approaches discussed in this chapter remain theoretical—or perhaps aspirational would be a better term—because in order to properly implement them, one would have to use frontier models such as GPT-4 or Claude 3 and consume a huge amount of tokens providing enough context for the AI to produce workable results. That's also the reason that this chapter doesn't have much in the way of practical examples. As I was writing, I got the distinct sense that rather than implementing these patterns in their code themselves, developers of the near future will be using these patterns via 3rd-party software services. Some of those services are probably embryonic startups right now.

In this chapter, we will consider specific patterns such as contextual error diagnosis, intelligent error reporting, predictive error prevention, self-healing error recovery, personalized error communication, and adaptive error handling workflows. We will emphasize the role of prompt engineering and retrieval-augmented generation (RAG) in implementing intelligent error handling patterns. We will also discuss how carefully crafted prompts and the integration of relevant information through RAG techniques can enhance the effectiveness of AI-driven error handling.

Traditional Error Handling Approaches

Before getting into the actual patterns, let's take a moment to discuss traditional error handling approaches and their limitations. Traditional error handling typically involves a combination of techniques such as:

- **Error Codes:** Assigning specific error codes to different types of errors, which can be used to identify and categorize the encountered issues.
- **Exception Handling:** Using programming language constructs like try-catch blocks to catch and handle exceptions that occur during program execution.
- **Logging:** Recording error details, stack traces, and relevant information in log files for later analysis and debugging.
- **Static Error Messages:** Displaying predefined error messages to users when specific errors occur, often with limited context or personalization.
- **Manual Intervention:** Relying on developers or support teams to manually investigate and resolve reported errors, which can be time-consuming and resource-intensive.

While these traditional approaches have been widely used, they have several limitations:

- **Lack of Contextual Understanding:** Traditional error handling often lacks the ability to properly capture the full context of an error, never mind understanding it automatically. This makes it difficult for traditional approaches to provide accurate and relevant error messages or suggest solutions.
- **Inflexibility:** Static error handling mechanisms may not adapt well to new or evolving error scenarios, requiring manual updates and modifications.
- **Limited User Experience:** Predefined error messages may not provide sufficient information or guidance to users, leading to frustration and a poor user experience.
- **Reactive Rather Than Proactive:** Traditional error handling tends to be reactive, focusing on detecting and responding to errors after they have occurred rather than proactively preventing them.
- **Inefficient Resource Utilization:** Manual error investigation and resolution can be time-consuming and costly, especially for complex or frequently occurring errors.

Recognizing these limitations, it becomes evident that a more intelligent and adaptive approach to error handling is necessary. By leveraging AI techniques and patterns, we can overcome these challenges and create error handling mechanisms that are more efficient, user-friendly, and effective in addressing the diverse range of error scenarios encountered in modern software applications.

In the following sections, we will explore various intelligent error handling patterns and discuss how prompt engineering and retrieval-augmented generation can be applied to implement these patterns effectively.

Contextual Error Diagnosis

This pattern focuses on using AI to intelligently diagnose the root cause of errors. By analyzing the context, logs, and relevant data, AI components can pinpoint the specific line of code, configuration issue, or external factor responsible for the error, potentially much faster than even the most experienced human software expert. This pattern saves developers valuable time in troubleshooting and enables quicker resolution of issues.

Contextual error diagnosis goes beyond simple error detection and logging. It involves leveraging AI techniques to understand the broader context in which an error occurs and identify the underlying causes. This can include analyzing:

- **Application State:** Examining the state of the application at the time of the error, including variable values, memory usage, and runtime conditions.
- **User Interactions:** Considering the sequence of user actions and inputs that led to the error, which can provide valuable insights into potential triggers or contributing factors.
- **Environmental Factors:** Assessing external factors such as system configurations, network conditions, or dependencies that may have influenced the occurrence of the error.
- **Historical Data:** Analyzing historical error logs, bug reports, and resolved issues to identify patterns, similarities, or recurring problems.

By combining these contextual data points, the AI can build a comprehensive understanding of the error scenario and accurately diagnose the root cause. This diagnostic information can be presented to developers in a structured and actionable format, enabling them to focus their efforts on the most relevant areas of the codebase or configuration.



Honeybadger is an exception tracking SaaS that provides real-time error monitoring, alerting, and debugging for web applications, helping developers quickly identify and resolve issues in their code. It's known for its seamless integration with various frameworks and languages, including Ruby on Rails, and it's our preferred product of this type at Olympia. What do you want to bet that in the coming years we will see AI leveraged heavily at Honeybadger and companies like it?

How It Works

Contextual error diagnosis, powered by prompt engineering and retrieval-augmented generation, represents a significant advancement in error handling. By intelligently analyzing the context and leveraging relevant information, AI components can accurately diagnose the root causes of errors, enabling developers to quickly identify and resolve issues. This pattern not only improves the efficiency of error resolution but also contributes to building more resilient and maintainable software applications. Let's explore how it works in detail.

Prompt Engineering for Contextual Error Diagnosis

Crafting effective prompts is crucial for guiding AI components in contextual error diagnosis. Well-designed prompts should capture the necessary information and provide clear instructions to the AI component, enabling it to perform accurate and relevant error diagnosis.

When engineering prompts for contextual error diagnosis, consider the following key elements:

1. **Error Details:** Include specific details about the encountered error, such as the error message, error code, or any other relevant information provided by the application or system. This helps the AI component understand the nature and characteristics of the error.

Example:

```
1 Error Message: "Division by zero in calculateDiscount function."  
2 Error Code: "ERR_DIV_ZERO"
```

2. **Contextual Information:** Provide relevant contextual information surrounding the error, such as the input data, application state, or any preconditions that led to the error. This additional context helps the AI component analyze the circumstances under which the error occurred.

Example:

```
1 Input Data:  
2 - Product Price: $100  
3 - Discount Percentage: 0%  
4  
5 Application State:  
6 - User: John Doe  
7 - Shopping Cart: 3 items  
8 - Payment Method: Credit Card
```

3. **Diagnostic Scope:** Clearly specify the desired scope of the error diagnosis. Indicate whether you want the AI component to identify the specific line of code, function, or module responsible for the error, or if you need insights into potential configuration issues or external factors.

Example:

```
1 Diagnostic Scope:  
2 - Identify the specific line of code causing the division by zero error.  
3 - Determine if there are any related configuration issues or external  
4 dependencies that may have contributed to the error.
```

4. **Reproducibility Steps:** If applicable, include steps to reproduce the error. This can help the AI component understand the sequence of actions or conditions that trigger the error, making it easier to diagnose and identify potential fixes.

Example:

- 1 Reproducibility Steps:
- 2 1. Add a product with a price of \$100 to the shopping cart.
- 3 2. Apply a discount percentage of 0%.
- 4 3. Proceed to the checkout page.
- 5 4. Observe the division by zero error in the `calculateDiscount` function.

5. **Expected Behavior:** Describe the expected behavior of the application or system in the absence of the error. This provides a clear contrast to the erroneous behavior and helps the AI component identify deviations from the intended functionality.

Example:

- 1 Expected Behavior:
- 2 - The `calculateDiscount` function should handle cases where the discount
- 3 percentage is 0% and apply no discount to the product price.
- 4 - The checkout process should proceed smoothly without encountering
- 5 any errors.

6. **Output Format:** Specify the desired format for the AI component's diagnostic output. This can include a structured representation of the identified root cause, suggested fixes, or any additional insights that can aid in error resolution.

Example:

- 1 Output Format:
- 2 - Root Cause: Identify the specific line of code and function responsible
- 3 for the division by zero error.
- 4 - Suggested Fix: Provide recommendations on how to handle cases where the
- 5 discount percentage is 0% to avoid the division by zero error.
- 6 - Additional Insights: Highlight any related code snippets, configuration
- 7 settings, or best practices that can help prevent similar errors in the
- 8 future.

By incorporating these elements into your prompts, you provide the AI component with the necessary information and guidance to perform effective contextual error diagnosis. Well-crafted prompts enable the AI component to analyze the error context, identify potential causes, and provide actionable insights for error resolution.

Remember to iterate and refine your prompts based on the specific requirements of your application and the types of errors you encounter. Continuously evaluate the quality and relevance of the AI component's diagnostic outputs and make adjustments to your prompts accordingly. Over time, you can build a library of effective prompts that cover a wide range of error scenarios, enhancing the efficiency and accuracy of your contextual error diagnosis process.

Retrieval-Augmented Generation for Contextual Error Diagnosis

Retrieval-augmented generation (RAG) techniques can be leveraged to enhance contextual error diagnosis by retrieving relevant error logs, documentation, or past error resolutions. RAG combines the power of retrieval systems with generative models to provide more accurate and informative diagnostic insights.

In the context of error diagnosis, RAG involves the following steps:

1. **Retrieval of Relevant Information:** When an error occurs, the RAG system searches through a large corpus of historical error logs, documentation, and resolved error cases to find information that is most relevant to the current error context. This retrieval process is based on similarity measures and ranking algorithms that consider factors such as error message, stack trace, and contextual information.

For example, let's say an error occurs with the message "LookupError: 'user_id'". The RAG system would search through its corpus and retrieve error logs or resolved cases that involve similar exceptions related to missing keys.

2. **Augmentation of Error Context:** The retrieved information is then used to augment the context of the current error. This augmentation process involves combining the retrieved information with the original error details, such as the error message, stack trace, and relevant code snippets. The

augmented context provides a richer representation of the error scenario, including historical examples, potential causes, and resolution strategies.

Continuing with the previous example, the RAG system might retrieve a past error case where a similar error was resolved by adding a default value for the missing key. This retrieved information would be combined with the current error context to provide a more comprehensive understanding of the issue.

3. **Generation of Diagnostic Insights:** Using the augmented error context, the RAG system generates diagnostic insights and recommendations. This generation process may leverage advanced natural language processing techniques to produce human-readable explanations and suggestions.

The generated diagnostic insights can include:

- Probable root causes of the error
- Suggested code fixes or workarounds
- Best practices or preventive measures to avoid similar errors in the future
- References to relevant documentation or resources

4. **Iterative Refinement:** The generated diagnostic insights are presented for review and feedback using a [Feedback Loop](#). You can then provide additional information, clarify the error context, or ask follow-up questions. Based on this feedback, the RAG system can iteratively refine its diagnostic insights, retrieving more relevant information and generating updated recommendations.

Through this iterative process, the RAG system continuously improves its understanding of the error context and provides more accurate and targeted diagnostic insights.

By leveraging RAG techniques, contextual error diagnosis becomes more efficient and effective. The retrieval of relevant information from historical error logs, documentation, and resolved cases enables the AI component to provide

more comprehensive and contextually relevant diagnostic insights. This helps developers quickly identify the root causes of errors, access relevant resources, and apply appropriate resolution strategies.

To implement RAG for contextual error diagnosis, you need a well-structured corpus of historical error data, documentation, and resolved error cases. This corpus should be preprocessed and indexed to facilitate efficient retrieval based on similarity measures. Additionally, you need to train generative models on this corpus to enable the generation of human-readable diagnostic insights. Needless to say, this is a big and potentially very expensive undertaking for most organizations, especially at enterprise scales.

Intelligent Error Reporting

This pattern is a type of [Contextual Content Generation](#) that can perhaps involve some amount of [Passive Information Radiation](#). It involves leveraging AI to generate comprehensive and user-friendly error reports. Intelligent error reporting goes beyond traditional error logging by utilizing AI techniques to create error reports that are more informative, contextually relevant, and actionable.

The goal of intelligent error reporting is to provide you and your support teams with the necessary information to quickly understand, diagnose, and resolve errors. By generating error reports that are tailored to the specific context and audience, intelligent error reporting reduces the time and effort required to investigate and fix issues.

Key aspects of intelligent error reporting include:

1. **Automated Error Capture:** Intelligent error reporting systems automatically capture relevant information about errors as they occur. This includes details such as the error message, stack trace, input data, application state, and any other pertinent context. The automated capture ensures that no critical information is missed and relieves developers from manually logging errors.
2. **Contextual Enrichment:** AI techniques are used to enrich the captured error data with additional context. This can involve analyzing the codebase to identify the specific lines of code or functions related to the error, examining user interactions leading up to the error, or retrieving relevant documentation or past error cases. The contextual enrichment provides a more comprehensive view of the error scenario.
3. **Intelligent Categorization:** Intelligent error reporting systems employ language models to automatically categorize errors based on their characteristics and context. This categorization can be based on factors such as

the type of error, the affected components or modules, or the severity level. Intelligent categorization helps in prioritizing errors and routing them to the appropriate teams for resolution.

4. **User-Friendly Presentation:** AI techniques are utilized to generate error reports that are easily understandable and actionable for developers and support teams. This involves presenting the error information in a structured and visually appealing format, highlighting key details, and providing clear explanations or recommendations. User-friendly presentation reduces the cognitive load on developers and enables faster error comprehension.
5. **Integration with Issue Tracking:** Intelligent error reporting systems can seamlessly integrate with your issue tracking or bug reporting tools. When an error occurs, the generated error report is automatically created as a new issue or ticket in the tracking system. This integration streamlines the error resolution process by ensuring that errors are properly documented, assigned, and tracked through to resolution.
6. **Collaborative Feedback:** Intelligent error reporting systems facilitate collaboration and feedback between developers and support teams. They provide mechanisms for developers to add comments, ask questions, or provide additional insights within the error report. This collaborative approach enables better communication and knowledge sharing, leading to faster error resolution.

To implement intelligent error reporting, you can leverage various approaches. Here are a few examples:

- **Natural Language Processing (NLP):** You can use an LLM to analyze error messages, stack traces, and log files, asking it to extract relevant information and generate human-readable summaries or explanations.
- **Anomaly Detection:** You can provide an LLM with guidelines that include typical system statistics and then ask it to identify unusual or unexpected

error patterns, helping to surface critical issues that require immediate attention.

- **Similarity Matching:** Similarity matching techniques can be employed to find similar error cases in historical data, providing insights into potential root causes and resolution strategies.

As you implement intelligent error reporting in your applications, continuously monitor and evaluate the quality and effectiveness of the generated reports. Gather feedback from developers and support teams to identify areas for improvement and refine the AI models and techniques used in your error reporting system. Over time, intelligent error reporting will become an integral part of your error handling workflow, enabling faster issue resolution and enhancing the overall development experience.



Don't forget to put *circuit breakers* and *rate limiters* in place for anything that triggers LLM completions based on errors, or you might find yourself lamenting a very large and unexpected invoice from your friendly neighborhood AI platform.

Predictive Error Prevention

This pattern utilizes AI to predict and prevent potential errors *before they occur*. By analyzing historical error patterns, codebase characteristics, and runtime behavior, predictive error prevention techniques can identify high-risk areas and proactively suggest preventive measures.

The goal of predictive error prevention is to minimize the occurrence of errors and improve the overall reliability and stability of your software applications. By identifying and addressing potential error-prone scenarios early in the development process, this pattern helps you reduce the cost and effort associated with error fixing and maintenance.

How It Works

Key aspects of predictive error prevention include:

1. **Historical Error Analysis:** AI is used to analyze historical error data, including error logs, bug reports, and resolved issues. By examining past error patterns and their associated contexts, the AI models can identify common error scenarios, root causes, and contributing factors. This analysis provides valuable insights into the application's error landscape.
2. **Code Analysis:** Predictive error prevention techniques involve analyzing the codebase to identify potential error-prone areas. AI models can be trained to recognize code patterns, anti-patterns, and coding practices that are more likely to introduce errors. This analysis can be performed statically (without executing the code) or dynamically (during runtime) to detect potential issues.
3. **Runtime Behavior Monitoring:** AI-powered monitoring systems can continuously observe the application's runtime behavior to detect anomalies or deviations from expected patterns. By analyzing metrics such as

performance, resource utilization, and user interactions, these systems can identify potential error conditions or performance bottlenecks before they manifest into actual errors.



I've experimented with this particular approach at Olympia, with mixed results. My implementation simply provided a series of data snapshots, clearly tagged as time-series data in the context. The directive asked the AI to call an alarm function if it “noticed” a deviation from typical values.

4. Continuous Learning and Refinement: Predictive error prevention systems should continuously learn and adapt based on new error data, codebase changes, and feedback from developers. As more data becomes available and the application evolves, refine your prompts to improve their accuracy and effectiveness in predicting and preventing errors.

Predictive error prevention, when combined with other patterns like [Contextual Error Diagnosis](#) and [Intelligent Error Reporting](#), creates a comprehensive approach to error handling. By proactively identifying and preventing potential errors, this pattern helps reduce the overall error occurrence, improves application reliability, and enhances the development experience.

Smart Error Recovery

This pattern focuses on using AI to automatically recover from errors and maintain application stability. The goal is to minimize downtime and ensure a seamless user experience by intelligently handling and resolving errors without requiring manual intervention.

In traditional error handling approaches, when an error occurs, the application may crash, become unresponsive, or require manual intervention to restore its functionality. Self-healing error recovery, on the other hand, leverages AI

techniques to automatically detect, diagnose, and recover from errors, allowing the application to continue running smoothly.

How It Works

Key aspects of self-healing error recovery include:

1. **Error Detection:** Self-healing error recovery systems continuously monitor the application's behavior and log streams to detect errors in real-time. LLM-based scanners are setup to identify patterns and anomalies that indicate the occurrence of an error. These models can analyze log messages, performance metrics, and user feedback to quickly detect and flag potential issues.
2. **Error Diagnosis:** Once an error is detected, self-healing error recovery systems employ AI techniques to diagnose the root cause of the error. This involves analyzing the error context, including the application state, input data, and relevant logs. AI models can leverage techniques like [Contextual Error Diagnosis](#) to identify the specific component, module, or line of code responsible for the error.
3. **Automated Recovery Strategies:** Based on the diagnosed error, self-healing error recovery systems automatically select and execute appropriate recovery strategies. These strategies can include:
 - Restarting the affected component or module
 - Rolling back to a previous stable version
 - Applying pre-defined error-specific fixes or patches
 - Dynamically adjusting configuration settings
 - Redirecting traffic to backup or redundant systems
4. **Graceful Degradation:** In cases where a complete recovery is not possible, self-healing error recovery systems could employ graceful degradation techniques. This involves automatically adapting the application's behavior to maintain essential functionality while temporarily disabling

or limiting non-critical features. Graceful degradation ensures that the application remains usable and responsive, even in the presence of errors.

5. **Continuous Learning and Adaptation:** Self-healing error recovery systems can use [Continuous Learning](#) based on error occurrences and recovery outcomes, so that they improve as they are updated with new error patterns, recovery strategies, and feedback from users and developers. This continuous learning process enables the system to improve its error detection accuracy, refine its recovery strategies, and adapt to evolving application requirements.
6. **Transparent Error Reporting:** While self-healing error recovery aims to automatically resolve errors, it is probably crucial to maintain transparency and keep developers informed about what is happening in real-time using [Passive Information Radiation](#). The system could generate detailed error reports, including information about the detected error, the diagnosed root cause, and the applied recovery strategies using [Intelligent Error Reporting](#) mechanisms, enabling developers to review, analyze, and provide feedback on the error recovery process.



It's not hard to envision a future in which self-healing error recovery, combined with other patterns like [Predictive Error Prevention](#) and [Adaptive Error Handling Workflow](#) would come together to create a robust and resilient error handling ecosystem that keeps your systems up and running with very little human intervention.

As you implement self-healing error recovery in your applications, it's important to continuously monitor and evaluate the effectiveness of the recovery strategies. Regularly review the generated error reports, gather feedback from users and developers, and refine the AI models and recovery strategies accordingly. Over time, self-healing error recovery will become an integral part of your error handling approach, enabling your applications to autonomously

handle and recover from errors, reducing the burden on development and support teams.

Personalized Error Communication

This pattern involves leveraging AI to provide personalized and context-aware error messages to users. Personalized error communication aims to enhance the user experience by delivering error messages that are tailored to the user's context, technical expertise, and specific situation.

How It Works

Traditional error messages often use generic, technical language that can be confusing or overwhelming for users, especially those with limited technical knowledge. The Personalized Error Communication pattern, on the other hand, uses AI techniques to generate error messages that are more user-friendly, informative, and actionable.

Key aspects of implementing personalized error communication include:

1. **User Profiling:** AI is used to build user profiles based on various factors such as the user's role, technical expertise, past interactions, and preferences. These profiles help in understanding the user's context and communication needs. For example, a novice user may require more detailed explanations and step-by-step guidance, while an experienced user may prefer concise and technical information. This profile information is vital context for the personalized error communication prompts.
2. **Context Analysis:** Personalized error communication prompts also get context pertaining to analysis of the environment in which an error occurs. This includes factors such as the user's current task, the application's state, and the specific inputs or actions that led to the error. By considering the context, the system can generate error messages that are more relevant and helpful to the user's situation.
3. **Natural Language Generation:** Use LLMs to generate error messages in

a human-friendly language. The generated messages can include explanations, suggestions, and actionable steps to help users resolve the error.

4. **Tone and Style Adaptation (Optional):** Personalized error communication systems adapt the tone and style of the error messages based on the user's profile and preferences. For example, the system may use a more empathetic and reassuring tone for novice users, while employing a more direct and technical tone for experienced users. The adaptability of the error messages helps in building trust and improving the overall user experience.
5. **Multilingual Support (Optional):** Ensure that users can understand and act upon the error messages, regardless of their language proficiency.
6. **Continuous Learning and Refinement (Optional):** Personalized error communication systems can use [Continuous Learning] to improve based on user feedback and interactions. The system adapts to changing user needs and preferences, ensuring that the error communication remains effective over time.

Adaptive Error Handling Workflow

Our final pattern utilizes AI to dynamically adapt error handling workflows based on the severity and impact of the error, and is the implementation of the approach we discussed for exception handling and recovery in the [Intelligent Workflow Orchestration Chapter](#). The goal is to optimize the error resolution process by intelligently allocating resources, prioritizing critical issues, and ensuring efficient collaboration between developers and support teams.

In traditional error handling workflows, all errors are often treated with the same level of urgency and follow a fixed resolution process. However, not all errors have the same impact on the application's functionality or user experience. Adaptive error handling workflows use AI to assess the severity and impact of each error and dynamically adjust the workflow accordingly.

How It Works

Key aspects of adaptive error handling workflows include:

1. **Error Severity Assessment:** AI models are used to analyze the characteristics and context of each error to determine its severity level. Factors such as the error type, affected components, potential impact on user experience, and historical data are considered in the assessment. The severity levels can be categorized into critical, high, medium, or low, each requiring a different level of attention and response.
2. **Impact Analysis:** Adaptive error handling workflows evaluate the potential impact of an error on the application's functionality, performance, and user experience. AI techniques, such as dependency analysis and user behavior analysis, are used to identify the extent of the error's impact and prioritize the resolution efforts accordingly. Errors with a higher impact are given higher priority and allocated more resources for faster resolution.

3. **Dynamic Workflow Adaptation:** Based on the assessed severity and impact, the error handling workflow dynamically adapts to optimize the resolution process. This adaptation can involve:

- Assigning critical errors to experienced developers or dedicated support teams for immediate attention
- Prioritizing high-impact errors in the resolution queue and allocating additional resources
- Automating the resolution of low-severity errors using predefined scripts or [Smart Error Recovery](#)
- Escalating complex or recurring errors to specialized teams or experts for in-depth analysis

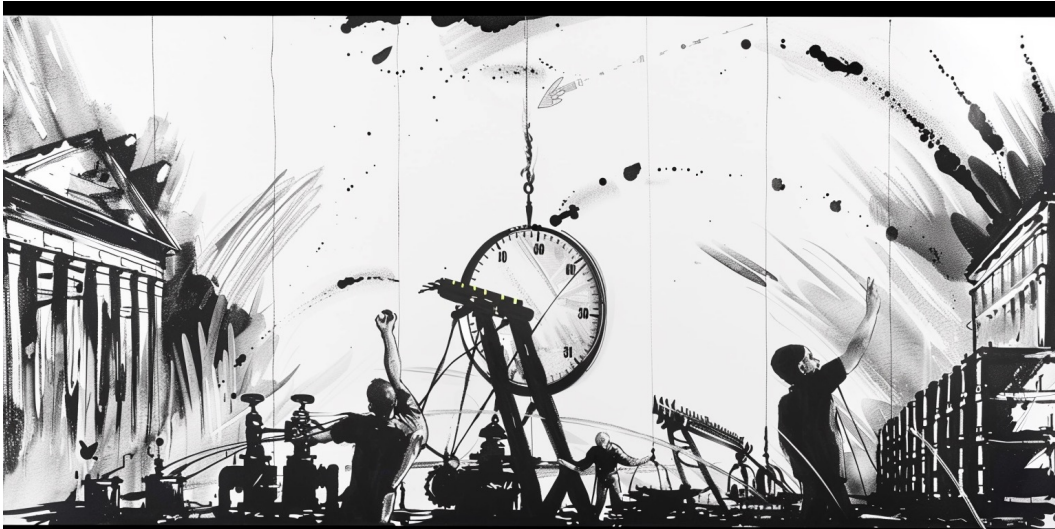
The adaptive workflow ensures that the right resources are allocated to the right errors at the right time, maximizing efficiency and minimizing the overall impact on the application.

4. **Intelligent Notification and Escalation:** Adaptive error handling workflows use AI to intelligently notify and escalate errors to the appropriate stakeholders, which is a form of [Personalized Error Communication](#). Based on the error severity and impact, the system determines the relevant developers, support teams, or managers who need to be informed. Notifications are sent through the most effective channels, such as email, chat, or project management tools, ensuring prompt attention and response.

5. **Collaborative Resolution:** AI-powered adaptive error handling workflows facilitate collaboration among developers and support teams, perhaps using [Collaborative Decision Making](#). The system provides a centralized platform where stakeholders can communicate, share insights, and work together to resolve errors efficiently. AI techniques, such as natural language processing (NLP), can be used to analyze the conversations and provide relevant suggestions, code snippets, or knowledge base articles to aid in the resolution process.

6. **Continuous Improvement:** Adaptive error handling workflows can use [Continuous Learning](#) to improve based on the outcomes of error resolutions. AI models analyze the resolution process, including the time taken, resources utilized, and user feedback, to identify areas for optimization. The workflows are refined over time to become more efficient, accurate, and effective in handling errors of varying severity and impact.

Quality Control



Quality control in AI applications goes beyond traditional software testing. It requires a multifaceted approach that can adapt to the dynamic and sometimes unpredictable nature of AI-generated content. This chapter introduces two fundamental patterns for implementing robust quality control measures in your AI systems: Eval and Guardrail.

The Eval pattern focuses on assessing the quality and appropriateness of AI-generated content. It provides a framework for systematically evaluating outputs against predefined criteria, offering insights into the performance of your AI components and identifying areas for improvement.

The Guardrail pattern, on the other hand, takes a proactive approach to quality control. It implements protective mechanisms that actively prevent or filter out problematic content before it reaches the end-user, serving as a critical safety net in real-time applications.

While distinct in their primary functions, these patterns often work in tandem,

forming a comprehensive quality control strategy. Evals can inform the criteria used in Guardrails, while the data collected from Guardrails can help refine and improve Eval mechanisms over time.

As we explore these patterns, keep in mind that effective quality control in AI applications is an ongoing process. It requires continuous monitoring, refinement, and adaptation to evolving standards and emerging challenges. By implementing these patterns, you'll be better equipped to harness the power of AI while maintaining high standards of quality and safety in your applications.

Let's dive into the details of each pattern and explore how they can be implemented to enhance the reliability and trustworthiness of your AI-driven systems.

Eval

The *Eval* pattern involves implementing automated evaluation mechanisms to assess the quality, accuracy, and appropriateness of AI-generated content. This pattern is crucial for maintaining high standards in AI-driven applications and can be used both for offline performance monitoring and as part of real-time quality control systems.

Problem

AI models, especially large language models (LLMs), can produce inconsistent or inappropriate outputs. Without proper evaluation, these issues may go undetected, potentially leading to poor user experiences, misinformation, or even harmful content being served to users.

Solution

Implement automated evaluation mechanisms that assess various aspects of AI-generated content. These evals can range from simple rule-based checks to complex AI-powered analysis like the metrics defined by the [RAGAs framework](#). The most effective evals are often those that can operate without requiring a “golden reference”, making them suitable for both offline analysis and real-time application.

How It Works

1. **Define Evaluation Criteria:** Determine the aspects of the AI output that need to be evaluated. This could include factual consistency, relevance, safety, fluency, or task-specific metrics.
2. **Implement Evaluation Logic:** Develop algorithms or models to assess the defined criteria. This could involve:

- Rule-based systems for straightforward checks
 - Machine learning models trained on labeled data
 - AI models (like LLMs) instructed to perform specific evaluation tasks
3. **Input Processing:** The eval takes the AI-generated content as input, along with any necessary context (e.g., the original prompt, relevant metadata).
 4. **Evaluation Execution:** Run the evaluation logic on the input, producing one or more scores or classifications.
 5. **Output:** Return the evaluation results, which could be binary (pass/fail), numerical scores, or more detailed feedback.
 6. **Integration:** Use the eval results to inform decision-making processes, such as whether to display content to users, trigger regeneration, or flag for human review.



For a different take on aspect-oriented evaluation see [Aspect Critique](#).

Example

Here's a basic implementation of an Eval pattern in Ruby:

```
1 class Eval
2   include Raix::ChatCompletion
3   include Raix::FunctionDispatch
4
5   attr_accessor :criteria
6   attr_accessor :result
7
8   function :save_evaluation, "Save evaluation score",
9     score: { type: "integer", description: "from 0 to 100" },
10    explanation: { type: "string" } do |params|
11     self.result = OpenStruct.new(params)
12   end
13
14   def initialize(criteria)
15     self.criteria = criteria
```

```

16     self.tool_choice = 'required'
17     transcript << { system: "Evaluate AI response to Prompt based on: #{criteria}" }
18 end
19
20 def evaluate(prompt, response)
21     transcript[1] = { user: ["Prompt: #{prompt}"], ["AI: #{response}"].join("\n\n") }
22     chat_completion
23     result
24 end
25 end
26
27 # Usage
28 eval = Eval.new("factual consistency")
29 result = eval.evaluate(prompt, ai_generated_response)
30
31 >> result.score
32 => 100
33
34 >> result.explanation
35 => "The AI responded in factually consistent manner"

```

Considerations

- **Modularity:** Design evals to be modular, allowing for easy addition or modification of evaluation criteria.
- **Continuous Improvement:** Regularly update and refine evals based on new data and evolving standards.
- **Bias Awareness:** Be mindful of potential biases in your evaluation methods and strive for fairness across different types of content and user groups.

Understanding Golden References

In the field of AI and machine learning, particularly in natural language processing tasks, a “golden” reference (also known as a “gold standard” or “ground truth”) refers to an ideal, human-created example that is considered the perfect or optimal output for a given input or task.

1. **Human-Created:** Golden references are typically created by human experts or through a consensus of multiple human judgments.

2. **High Quality:** They are considered to be of the highest possible quality, free from errors or inconsistencies.
3. **Benchmark:** Golden references serve as benchmarks against which machine-generated outputs are compared and evaluated.
4. **Task-Specific:** The nature of a golden reference varies depending on the task. For example:
 - In translation, it would be a professionally translated text.
 - In summarization, it would be an expert-written summary.
 - In question-answering, it would be the correct answer provided by subject matter experts.

The use of golden references presents a minefield of challenges. For many tasks, there isn't a single "correct" output. Different humans might produce equally valid but different golden references. Creating golden references also requires significant time and resources, especially for large datasets. Not only that, but it's also impractical to create golden references for every possible input in open-ended or generative tasks. Finally, golden references may inadvertently incorporate human biases or errors.

Given these challenges, many modern AI evaluation techniques aim to operate without requiring golden references. This approach, known as reference-free evaluation, is possible due to the power of frontier large-language models, and offer several advantage over golden references:

1. **Flexibility:** It allows for evaluation of outputs in scenarios where creating golden references is impractical or impossible.
2. **Scalability:** Reference-free evals can be applied to a much wider range of inputs and outputs.
3. **Real-time Application:** These evals can be used in production systems where golden references are not available.
4. **Adaptability:** They can potentially adapt to evolving language use and standards more easily than fixed golden references.

By designing evals that don't rely on golden references, we create more versatile and scalable quality control mechanisms for AI-generated content. These evals focus on intrinsic qualities of the output (such as coherence, relevance, or safety) rather than comparison to a single ideal example. This approach aligns well with the dynamic and open-ended nature of many AI applications, where the range of possible good outputs may be vast and diverse.

How Reference-Free Evals Work

As demonstrated in the earlier example, reference-free evaluations focus on the qualities of the AI-generated output itself, rather than comparing it to an ideal example. These qualities might include coherence, relevance, fluency, factual consistency, and safety. By assessing these aspects, reference-free evals can provide robust quality control without needing a human-created benchmark.

By focusing on intrinsic qualities of the output, reference-free evaluations offer a practical, scalable approach to maintaining quality and safety in AI applications. They enable continuous monitoring and improvement, ensuring that AI systems remain reliable and trustworthy even as they evolve.

—

While Evals focus on assessing content quality, Guardrails use similar mechanisms to actively prevent problematic content from being generated or displayed. Evals and Guardrails often work in tandem to ensure overall system quality and safety.

Guardrail

The Guardrail pattern involves implementing protective mechanisms that actively prevent or filter out problematic AI-generated content before it reaches the end-user. This pattern is crucial for maintaining content safety, ensuring output quality, and mitigating potential risks associated with AI-generated content in real-time applications.

Problem

AI models, particularly large language models (LLMs), can sometimes generate inappropriate, inaccurate, or potentially harmful content. Without proper safeguards, this content could be exposed to users, leading to poor user experiences, misinformation, or even legal and ethical issues.

Solution

Implement automated guardrail mechanisms that proactively check and filter AI-generated content based on predefined criteria. These guardrails act as a safety net, ensuring that only content meeting specific quality and safety standards is allowed to pass through to the user.

How It Works

1. **Define Guardrail Criteria:** Establish clear criteria for what constitutes acceptable content. This may include safety guidelines, quality thresholds, and application-specific requirements.
2. **Implement Filtering Logic:** Develop algorithms or models to assess content against the defined criteria. This could involve:
 - Rule-based systems for straightforward checks
 - Machine learning models trained or fine-tuned on labeled data

- AI models instructed to perform specific content analysis tasks
3. **Content Interception:** The guardrail intercepts AI-generated content before it's presented to the user.
 4. **Evaluation and Decision:** Apply the filtering logic to the intercepted content, making a decision on whether to allow, modify, or block the content.
 5. **Action Execution:** Based on the decision, either allow the content to pass through, modify it to meet standards, or block it entirely.
 6. **Logging and Feedback:** Record guardrail actions for monitoring and improvement purposes.

Example

Here's a basic implementation of a Guardrail pattern in Ruby. As you can see, it is very similar to an Eval implementation, but meant for use in real time:

```

1  class Guardrail
2    include Raix::ChatCompletion
3    include Raix::FunctionDispatch
4
5    attr_accessor :criteria
6    attr_accessor :result
7
8    MIN_SCORE = 0.7
9
10   function :save_evaluation, "Save evaluation score",
11     score: { type: "integer", description: "from 0 to 100" },
12     explanation: { type: "string" } do |params|
13     self.result = Result.new(**params)
14   end
15
16   def initialize(criteria)
17     self.criteria = criteria
18     self.tool_choice = 'required'
19     transcript << { system: "Evaluate AI response to Prompt based on: #{criteria}" }
20   end
21
22   def evaluate(prompt, response)
23     transcript[1] = { user: ["Prompt: #{prompt}"], ["AI: #{response}"].join("\n\n") }
```

```

24     chat_completion
25     result
26 end
27
28 class Result
29   attr_accessor :score, :explanation
30
31   def initialize(score:, explanation:)
32     self.score = score
33     self.explanation = explanation
34   end
35
36   def allow?
37     score > MIN_SCORE
38   end
39 end
40 end
41
42 # Usage
43 gr = Guardrail.new("AI responded in English or Spanish")
44 result = gr.evaluate("你好", "你好AI你好")
45
46 >> result.allow?
47 => false
48
49 >> result.explanation
50 => "The AI responded to a prompt in Mandarin using Mandarin."

```

In contrast to the Eval example which returns a simple data structure for further processing, the Guardrail returns a Result object that we can call `allow?` on to help us decide whether to let the AI respond to the user.

Considerations

Be mindful of *false positives*, overly strict guardrails that might filter out acceptable content. Strive for a balance between safety and usability. Consider *transparency*, mechanisms to inform users when content has been modified or blocked by a guardrail. Finally, design guardrails to be *adaptable*, easily updatable as new types of problematic content emerge or standards evolve.



In conversational applications, you can potentially use powerful Guardrails—implemented with expensive frontier model such as GPT or Claude—to let you get away with using fast, low-cost models such as Mistral or Llama as the engine powering your main conversation loop.

Guardrails and Evals: Two Sides of the Same Coin

Guardrails and evaluations play crucial roles in ensuring the quality, safety, and reliability of our systems that incorporate AI processing. While these concepts are often discussed separately, there's a significant overlap in their functionality and implementation, especially when it comes to reference-free evaluations.

The Interchangeability of Guardrails and Reference-Free Evals

Guardrails are typically thought of as protective measures that catch inappropriate or harmful content before it reaches the end-user. On the other hand, evals are usually associated with measuring the quality and accuracy of the model's output. However, in the case of reference-free evals – evaluations that don't rely on a “golden” reference or human-written answer – the line between guardrails and evals becomes blurred.

Reference-free evals assess the quality of output based solely on the input prompt and the model's response. This characteristic makes them particularly versatile, as they can be used both to evaluate performance and to act as guardrails in real-time applications.

Consider the following examples:

1. **Summarization Tasks:** When evaluating a summary, we can assess factual consistency and relevance without needing the original “perfect” sum-

mary. If the generated summary scores poorly on these metrics, we can choose not to display it to the user, effectively using the eval as a guardrail.

2. **Translation Quality:** Reference-free translation evals can assess the quality of a translation without needing a human-translated reference. These same evals can be used as guardrails to prevent low-quality translations from being presented to users.
3. **Content Moderation:** Evals designed to detect toxic or inappropriate content can double as guardrails, filtering out problematic responses before they reach the user.

Implementing Dual-Purpose Guardrails and Evals

To leverage the interchangeability of guardrails and reference-free evals, consider the following implementation strategies:

1. **Modular Design:** Design your evaluation components to be modular and reusable. This allows the same code to be used both in offline evaluation pipelines and real-time guardrail systems.
2. **Threshold-Based Filtering:** Implement a threshold-based system where the eval score determines whether the output passes the guardrail.
3. **Continuous Improvement:** Use the data collected from your guardrails to continuously improve your evals, and vice versa. This feedback loop ensures that both components evolve together, enhancing the overall quality of your system.
4. **Multi-Faceted Evaluation:** Implement multiple evals/guardrails that assess different aspects of the output (e.g., relevance, safety, fluency). This comprehensive approach provides a more robust quality assurance mechanism.

By recognizing the dual nature of reference-free evals and guardrails, you can create more efficient, robust, and flexible AI systems. This approach not only

streamlines your development process but also ensures a higher level of quality control throughout your application.

Glossary

A

A/B Testing A method of comparing two versions of a component or feature to determine which performs better, often used in AI-driven applications to optimize user experiences or model performance.

Accessibility The design of products, devices, services, or environments to be usable by people with disabilities. In AI contexts, it often refers to making AI-powered interfaces and outputs accessible to all users.

Adaptive Error Handling Workflow

A pattern that uses AI to dynamically adjust error handling processes based on the severity and impact of errors, optimizing resource allocation and resolution efficiency.

Agentic Term used to describe AI systems that can act autonomously to achieve goals, often making decisions and taking actions without direct human intervention.

AI (Artificial Intelligence) The simulation of human intelligence processes by machines, especially computer systems. These processes include learning, reasoning, and self-correction.

Answer Correctness: A RAGAs metric that gauges the accuracy of a generated answer compared to a ground truth answer, combining semantic and factual similarity, with scores from 0 to 1.

Aspect Critique: A technique used to assess prompt responses based on predefined aspects such as harmlessness and correctness, providing a binary output indicating alignment with particular criteria.

Answer Relevance: A RAGAs metric that evaluates how pertinent a generated answer is to a given prompt, with lower scores for incomplete or redundant answers and higher scores indicating better relevancy.

Answer Semantic Similarity (ANSS): A RAGAs metric that evaluates the semantic resemblance between the generated answer and ground truth information, with values from 0 to 1, where higher scores indicate better alignment.

Anthropomorphism The attribution of human characteristics or behavior to non-human entities, including AI systems. In prompt engineering, this approach is used to make interactions with AI more intuitive and accessible.

API (Application Programming Interface) A set of protocols, routines, and tools for building software applications that specify how software components should interact.

API Bridge A design pattern that provides a simplified, unified interface to a complex subsystem of APIs, making it easier to use and reducing dependencies between the client and the subsystem. Also known as an API Facade.

Attention Mechanism A technique used in neural networks, particularly in transformer models, that allows the model to focus on different parts of the input when producing an output.

B

Base Model An AI model that hasn't been fine-tuned or specialized for specific tasks, often requiring more precise prompting.

BERT (Bidirectional Encoder Representations from Transformers) A transformer-based machine learning technique for natural language processing pre-training developed by Google, known for its ability to consider the context of a word from both left and right sides.

Bias In AI, bias refers to systematic errors in AI systems that can lead to unfair or inaccurate outcomes, often reflecting human prejudices present in training

data or algorithm design.

Black Box A system or device whose internal workings are not visible or understood, often used to describe complex AI models whose decision-making processes are not easily interpretable.

Boundary Conditions The extreme or edge cases in input data or scenarios that test the limits of an AI system's performance and reliability.

Bytepair Encoding (BPE) A data compression technique used in natural language processing for tokenization, which iteratively merges the most frequent pair of bytes in a sequence to form new tokens.

C

Chain of Thought A prompting technique that encourages AI models to break down complex problems into step-by-step reasoning processes, improving problem-solving capabilities.

Chunking In the context of RAG, the process of breaking down large documents into smaller, manageable pieces for processing and retrieval.

Classifier An AI component used to categorize or label data points based on predefined classes or categories.

Cohere A company that provides large language models and NLP tools, mentioned in the book as one of the providers of AI models.

Coherence The quality of text being logically connected and consistent, often used to evaluate the output of language models.

Collaborative Decision Making (CDM) A pattern that involves designing AI systems that actively involve humans in the decision-making process, leveraging the strengths of both human expertise and AI capabilities.

Compound AI Systems AI systems that leverage complex pipelines of interconnected components working together to achieve intelligent behavior, often combining different AI techniques and models.

Confabulation In AI, particularly language models, hallucination refers to the generation of content that is factually incorrect or not grounded in the provided context or training data. Also known as “hallucination”.

Context The surrounding information or circumstances that provide meaning to a piece of text, a situation, or a task in AI systems.

Context Augmentation A technique that enhances AI model inputs with additional relevant information to improve the accuracy and relevance of outputs.

Context Entities Recall: A RAGAs metric that measures the fraction of entities (nouns) recalled from ground truths in the retrieved context, useful for fact-based use cases.

Context Precision: A RAGAs metric that assesses whether an answer contains ground truth facts provided by the context, with values ranging from 0 to 1, where higher scores indicate better precision.

Context Recall: A RAGAs metric that measures the extent to which the retrieved context aligns with ground truth or “golden reference” answers, with values between 0 and 1, where higher values indicate better performance.

Context Relevancy: A RAGAs metric that gauges the relevancy of the retrieved context based on both the question and contexts, with values from 0 to 1, where higher values indicate better relevancy.

Context Size The maximum number of tokens a language model can consider when generating a response, also known as the model’s “window” or “attention span”.

Contextual Content Generation A pattern that uses AI to generate dynamic and context-specific content within applications, adapting to user needs, preferences, and interactions.

Contextual Decision Making A pattern where AI systems make context-aware decisions within a workflow by analyzing relevant data points such as user preferences, historical patterns, and real-time inputs.

Contextual Error Diagnosis An AI-powered approach to error handling that analyzes the context, logs, and relevant data to pinpoint the root cause of errors more accurately and efficiently than traditional methods.

Continuous Learning A pattern in AI systems that enables ongoing learning and adaptation based on new data, feedback, and interactions, allowing the system to improve its performance over time.

Contrastive Learning A machine learning technique where models learn to distinguish between similar and dissimilar data points by pulling similar data closer together in a representation space and pushing dissimilar data apart.

Conversation Loop A pattern in AI-driven dialogues where the system continuously processes user inputs, generates responses, and maintains context throughout the interaction.

Conversational Referring to AI systems designed to engage in human-like dialogue, often used in chatbots and virtual assistants.

Copilot An AI-powered assistant that works alongside human users to enhance productivity and decision-making capabilities in specific domains or tasks.

D

Diffusion Model A type of generative AI model that learns to gradually denoise data in steps until a finished product is achieved. It's often used in image and sound generation tasks.

Discrete Components Individual, self-contained AI modules or functions that can be integrated into larger systems, allowing for modular and flexible AI-driven application development.

Domain-Specific Referring to AI models or applications tailored for a particular field or industry, often incorporating specialized knowledge and terminology.

Dungeons & Dragons A classic fantasy tabletop role-playing game mentioned

in the book as an example of how AI can be used to enhance interactive storytelling experiences.

Dynamic Task Routing A pattern in intelligent workflow orchestration where AI is used to assign tasks to the most appropriate resources or processing paths based on factors such as task priority, resource availability, and system performance.

E

Encoding The process of converting data from one format to another, often used in AI systems to prepare input data or process output in a specific format.

Ensemble Techniques Methods that combine multiple AI models or components to improve overall performance and accuracy in decision-making or predictions.

Enterprise Integration Patterns A collection of design patterns for connecting different components of enterprise applications, often referenced in the context of AI-driven application development.

Error Handling The process of anticipating, detecting, and resolving errors or exceptions in software applications to maintain stability and provide a smooth user experience.

Escalation A Human-In-The-Loop (HITL) pattern that focuses on designing mechanisms to automatically detect situations where human intervention is required and trigger an appropriate escalation process.

Eval In the context of AI, a process or function used to evaluate the performance or output of an AI model, often used in testing and validation.

Explainable AI (XAI) AI systems or techniques that provide clear explanations for their decisions or outputs, making the AI's reasoning process more transparent and understandable to humans.

Explainability The degree to which an AI system's decisions or outputs can be understood and interpreted by humans, crucial for building trust and ensuring accountability.

F

Faithfulness: A RAGAs metric that measures the factual consistency of an LLM-generated answer against a given context, scaled from 0 to 1, with higher scores indicating better consistency.

Feedback Loop A pattern in prompt engineering where the AI's output is used as input for the next iteration, allowing for progressive refinement and improvement of the generated content.

Few-Shot Learning A machine learning approach where a model is trained to perform a task with only a few examples or instances of training data.

Fine-tuning The process of further training a pre-trained AI model on a specific dataset to adapt it for a particular task or domain, improving its performance for that specific use case.

Frequency Penalty A parameter that discourages the model from repeating the same words or phrases too often in generated text.

Function Calling A technique in AI systems where the AI model can invoke specific functions or tools to perform tasks or retrieve information, enhancing its capabilities and allowing for more complex interactions.

G

Gemini A multimodal AI model developed by Google, capable of understanding and generating content across various modalities including text, images, audio, and video.

Generative Model An AI model capable of creating new, original content (such as text, images, or audio) based on patterns learned from training data.

GenUI (Generative UI) An approach to user interface design that leverages AI to create highly personalized and dynamic user experiences on-the-fly, adapting to individual user needs and contexts.

GPT (Generative Pre-trained Transformer) A type of large language model architecture that uses transformer neural networks and is pre-trained on vast amounts of text data, capable of generating human-like text and performing various language tasks.

Guardrail A set of constraints or rules implemented in AI systems to ensure they operate within predefined boundaries, maintaining safety, ethical standards, and desired behavior.

H

Hallucination In AI, particularly language models, hallucination refers to the generation of content that is factually incorrect or not grounded in the provided context or training data. Also known as “confabulation”.

Human-In-The-Loop (HITL) A design approach in AI systems that incorporates human intervention, oversight, or collaboration at key points in the process to improve accuracy, reliability, and ethical considerations.

Hyperparameters Configuration variables that are set before the machine learning process begins and control the behavior of the AI model during training and inference.

I

Inference The process of using a trained AI model to make predictions or generate outputs based on new, unseen input data.

Instruction Tuning A technique used to fine-tune language models to better understand and follow specific instructions, improving their ability to perform tasks based on user prompts.

Intelligent Error Handling An approach that leverages AI techniques to improve error detection, diagnosis, reporting, and resolution in software applications.

Intelligent Query Optimization (IQO) A technique that uses AI to refine and optimize queries for improved information retrieval and processing.

Intelligent Workflow Orchestration The use of AI to dynamically manage and optimize complex workflows within applications, improving efficiency and adaptability.

Intent Classification The process of automatically categorizing user inputs or queries into predefined intents or purposes, often used in conversational AI systems.

J

JSON (JavaScript Object Notation) A lightweight data interchange format that is easy for humans to read and write and easy for machines to parse and generate, often used in AI applications for data exchange.

K

Knowledge Graph A structured representation of knowledge in the form of entities and their relationships, often used to enhance AI systems with domain-specific information.

L

Large Language Model (LLM) A type of AI model trained on vast amounts of text data, capable of understanding and generating human-like text across a wide range of topics and tasks.

Latent Space In AI, particularly in generative models, latent space refers to a compressed representation of data where similar concepts are closer together, allowing for meaningful manipulations and generations.

Localization The process of adapting an application, including its AI components, to meet the language, cultural, and other requirements of a specific target market or locale.

Loosely Coupled A design approach where components in a system are minimally dependent on each other, often used in microservices architecture.

M

Machine Learning A subset of AI that focuses on the development of algorithms and statistical models that enable computer systems to improve their performance on a specific task through experience.

Maximum Length A parameter that sets the upper limit on the number of tokens a model can generate in a single response.

Message Passing A technique for invoking behavior in a computer program where an object sends a message to another object.

Microservice A software development technique where an application is structured as a collection of loosely coupled services.

Modality A type or category of data or sensory input, such as text, images, audio, or video, that an AI system can process or generate.

Multi-Agent A system or approach involving multiple AI agents working together, often to solve complex problems or simulate complex environments.

Multimodal Referring to AI systems or models that can process and generate content across multiple modalities, integrating different types of data or sensory inputs.

Multitude of Workers An approach in AI-driven application development that

involves using multiple specialized AI components or “workers” to perform different tasks or aspects of a complex process.

N

Named Entity Recognition (NER) A natural language processing technique that identifies and classifies named entities (such as person names, organizations, locations) in text.

Natural Language Processing (NLP) A branch of AI that focuses on the interaction between computers and humans using natural language, enabling machines to understand, interpret, and generate human language.

Narrow The Path A technique in prompt engineering that involves focusing the AI on a specific task or context to improve the accuracy and relevance of its responses.

Needle In A Haystack An English-language metaphor often used to describe the challenge of finding specific, relevant information within a large dataset, often addressed by AI-powered search and retrieval systems.

Neural Network A computational model inspired by the human brain, consisting of interconnected nodes (neurons) organized in layers, used as the foundation for many AI models.

Normalization The process of standardizing text data to a consistent format, often involving tasks like lowercasing or removing special characters.

Nucleus Sampling (Top-P) A text generation technique that selects from the smallest set of words whose cumulative probability exceeds a specified threshold.

O

OCR (Optical Character Recognition) Technology that converts images of text into machine-readable text data.

Ollama An open-source tool for running large language models locally, allowing developers to easily set up and use various AI models on their own machines.

Olympia A platform mentioned throughout the book that hosts AI-powered consultants and clones, used as a real-world example for various AI application patterns. The author is Olympia's Co-Founder and Chief Scientist, and most of the code examples in the book are extracted from its codebase.
<https://olympia.chat>

One-Shot Learning A machine learning approach where a model learns to perform a task from a single example.

OpenAI A leading artificial intelligence research laboratory. Makers of the revolutionary ChatGPT platform, and providers of frontier AI models such as GPT-4.

OpenRouter A service that provides a unified API for accessing multiple AI models from different providers, simplifying the process of integrating and switching between various AI capabilities.

Orchestration The automated arrangement, coordination, and management of complex computer systems, middleware, and services, often applied to AI components in large-scale applications.

P

PaLM (Pathways Language Model) A large language model developed by Google AI, known for its impressive performance across a wide range of language tasks.

Parameter Count The number of learnable parameters in an AI model, often counted in billions and used as a measure of the model's size and potential capability.

Part-of-Speech (POS) Tagging A natural language processing technique that assigns grammatical categories (such as noun, verb, adjective) to words in a

text.

Passive Information Radiation A pattern that focuses on designing AI systems to provide humans with relevant information and insights without explicitly requiring their intervention.

Perplexity (Provider) A leading provider of Internet-connected realtime AI models.

Perplexity (Term) A measure of how well a probability model predicts a sample, often used to evaluate language models.

Personalized Error Communication An approach that uses AI to generate error messages tailored to the user's context, technical expertise, and specific situation.

Predicate A pattern in AI systems that involves posing a specific question to the AI model and expecting a definitive yes or no answer, often used for making binary decisions.

Predictive Error Prevention A pattern that utilizes AI to predict and prevent potential errors before they occur by analyzing historical error patterns, code-base characteristics, and runtime behavior.

Presence Penalty A parameter that discourages the model from using words or phrases that are already present in the input prompt.

Principle of Least Privilege A security concept where each part of a system is given only the minimum levels of access needed to perform its function.

Process Manager A central component that maintains the state of a process and determines the next processing steps based on intermediate results, often used in AI-driven workflow orchestration.

Progressive Disclosure A design technique where information or interface elements are gradually revealed to users as needed, often implemented using AI to determine relevance and timing.

Prompt In the context of large language models, a text input given to the model to elicit a specific type of response or to guide the model's output.

Prompt Chaining A technique in prompt engineering where multiple prompts are combined in a sequence to guide the AI through a multi-step process or complex task.

Prompt Distillation A technique that involves using AI to optimize and refine prompts, improving their effectiveness in generating desired outputs.

Prompt Engineering The practice of designing and optimizing input prompts to elicit desired responses or behaviors from AI models, particularly large language models.

Prompt Object A pattern that involves representing prompts as database-backed records, treating them as first-class citizens in an application's architecture for better management and versioning.

Prompt Rewriter A pattern that leverages AI to optimize and refine user-supplied prompts, making them more effective for generating accurate and relevant responses.

Prompt Template A structured approach to prompt composition that separates static and dynamic parts of a prompt, allowing for more flexible and maintainable prompt engineering.

Proposition Chunking A technique for further decomposing chunks of long texts by breaking them down into factual, self-contained propositions.

Provider In the context of AI services, a company or platform that offers access to AI models, APIs, and related tools for developers to integrate into their applications.

Q

Quantization A technique used to reduce the precision of the numbers used to represent a model's parameters, often employed to decrease model size and

increase inference speed.

Query Analyzer A component in the Intelligent Query Optimization process that analyzes and understands the user's query to extract intent, contextual clues, and specific informational needs.

Query Rewriter A pattern that refines the original query to better match the language model's understanding, improving the relevancy of retrieved information for the user.

R

RAG (Retrieval Augmented Generation) A technique that combines information retrieval with text generation to produce more accurate and contextually relevant responses.

RAG Assessment (RAGAs): A framework and set of metrics for evaluating Retrieval Augmented Generation (RAG) pipelines, focusing on assessing each component of the RAG process in isolation.

Ranker A component in AI systems responsible for ordering or prioritizing a set of items based on certain criteria or relevance scores.

Raix A Ruby gem developed by Obie Fernandez for easily adding AI components to Rails applications, simplifying the integration of AI capabilities into web projects.

Raw Model An AI model that hasn't been fine-tuned or specialized for specific tasks, often requiring more precise prompting.

Reranking A process in information retrieval and AI systems that involves re-ordering initial search or generation results to improve relevance and quality.

Response Fencing A prompt engineering technique that constrains and controls the output generated by the AI model by hardcoding the start of the assistant's response and specifying a stop token.

Result Interpreter A pattern that leverages AI to analyze and interpret the results returned by functions or API calls, making them more accessible and meaningful to users.

Retrieval-based Model An AI model that selects the most appropriate response from a predefined set of responses, often used in question-answering systems and chatbots.

Retriever A component in AI systems used to search and extract relevant information from large datasets or knowledge bases.

RLHF (Reinforcement Learning from Human Feedback) A technique used to train language models by incorporating human feedback to refine the model's outputs and align them with human preferences.

Role Assignment A prompt engineering pattern that involves assigning a specific role or responsibility to the AI, providing context that steers its responses in terms of content, tone, and style.

Ruby on Rails An extremely powerful and popular web application framework written in Ruby, used extensively in the book's examples for implementing AI-driven applications.

S

Self-Healing Data An approach that uses AI to automatically detect, diagnose, and correct data anomalies, inconsistencies, or errors, reducing the burden on developers and maintaining high data reliability.

SentencePiece An unsupervised text tokenizer and detokenizer, primarily used for neural network-based text generation systems where the vocabulary size is predetermined.

Sentiment Analysis The use of natural language processing to determine the emotional tone behind a series of words, often used to gain an understanding of attitudes, opinions and emotions.

Server-Sent Events (SSE) A technology that enables servers to push data to web clients in real-time, often used in AI applications for streaming updates or notifications.

Smart Error Recovery A pattern that focuses on using AI to automatically recover from errors and maintain application stability without requiring manual intervention.

Software Architecture The high-level structure of a software system, defining its major components, their relationships, and how AI can be integrated into the overall design.

Stratification The process of dividing members of a population into homogeneous subgroups before sampling, often used in machine learning for balanced dataset creation.

Structured IO A prompt engineering pattern that leverages structured input and output to improve the accuracy and ease of integration when working with large language models.

Synonym Expansion A technique that broadens the vocabulary of a text by including synonyms, improving retrieval and understanding in AI systems.

System Directive Instructions or guidelines provided to an AI model to define its role, behavior, or context for a specific task or interaction.

T

TDD (Test Driven Development) A software development approach where tests are written before the actual code, applied to AI development to ensure reliability and correctness of AI components.

Temperature A hyperparameter in language models that controls the randomness and creativity of the generated text, with lower values producing more focused and deterministic outputs.

Temporal Inconsistency Issues arising in AI models when dealing with time-sensitive information, potentially leading to outdated or inconsistent outputs.

Text Cleaning The process of removing or correcting irrelevant or noisy information from text data to improve its quality for AI processing.

Thread A unit of execution within a program, allowing multiple operations to run concurrently, often used in AI applications to manage parallel processing tasks.

Time to First Token (TTFT) A performance metric measuring the time it takes for an AI model to generate its first token of output after receiving an input, crucial for real-time applications.

Token A unit of text (such as a word or subword) that language models use as input and output. Tokenization is the process of converting text into these units.

Tokenization The process of breaking down text into smaller units called tokens, which are used as input for language models.

Tool Function In the context of AI, a specific function or capability that an AI model can invoke to perform tasks or retrieve information, enhancing its action capabilities and problem-solving abilities.

Tool Use A technique that enables AI models to connect to and utilize external tools, APIs, functions, databases, and other resources to enhance their capabilities.

Top-K A text generation technique that selects from the K most likely next words at each step of the generation process.

Top-P (Nucleus Sampling) A text generation technique that selects from the smallest set of words whose cumulative probability exceeds a specified threshold.

Training The process of teaching an AI model to perform a task by exposing it to large amounts of data and adjusting its parameters to minimize errors.

Transcript In the context of conversations with large language models, a record of the exchange between the user and the AI, including prompts and AI-generated responses.

U

Unicode A universal character encoding standard that provides a unique number for every character across languages and scripts, ensuring consistent encoding and representation of text.

Unigram A type of language model that considers single tokens (usually words) in isolation, often used in simple text analysis and generation tasks.

Unit Testing A software testing method where individual components of an application, including AI modules, are tested in isolation to ensure they perform as expected.

Unsupervised Learning A type of machine learning where the model learns patterns from unlabeled data without explicit guidance (human or otherwise).

UX (User Experience) The overall experience of a person using a product or service, particularly important in designing AI-driven applications that are intuitive and user-friendly.

V

Ventriloquist A prompt engineering pattern that involves preloading hard-coded user-assistant exchanges into the conversation transcript to guide the AI's responses in a specific direction.

Vocabulary In the context of language models, the set of unique tokens that the model can recognize and generate, typically derived from the training data.

Virtual Machine A creative approach to prompt engineering that involves tricking the AI into simulating the execution of pseudo-code or a function.

W

Workflow A sequence of steps or tasks that process a set of data, often automated in AI systems to handle complex processes efficiently.

Workflow Orchestration The process of coordinating and managing complex sequences of tasks, often involving multiple systems or components, to achieve a specific business outcome.

Z

Zero-shot Learning A machine learning paradigm where a model can recognize or classify objects or perform tasks it hasn't been explicitly trained on, based on its understanding of related concepts.