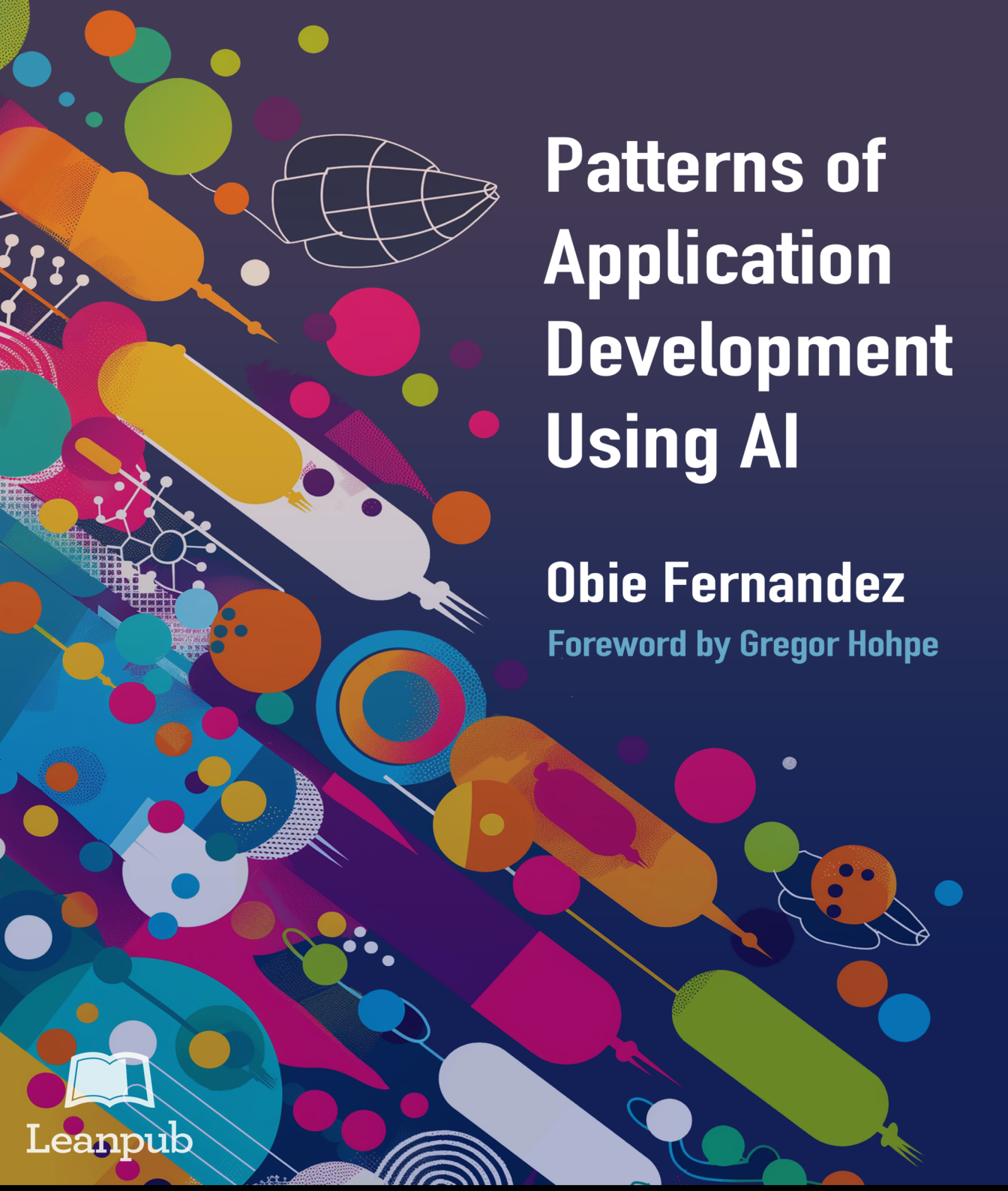

Patterns of Application Development Using AI
Obie Fernandez
Foreword by Gregor Hohpe
Leanpub

简体中文版

Patterns of Application Development Using AI (简体中文版)

Obie Fernandez

这本书的网址是 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

此版本发布于 2025-01-23

这是一本 Leanpub 的书。Leanpub 通过精益出版流程赋权给作者和出版商。精益出版 是使用轻量级工具并多次迭代获取读者反馈的过程，直到你有了合适的书籍并在这个基础上建立关注度。

在 Twitter 上分享这本书!

请在 Twitter 上帮助作者 Obie Fernandez 宣传!

对这本书建议的 hashtag 是 #poaduai.

若想知道其他人对这本书的看法，可以点击此链接搜索 Twitter 上的 hashtag：

#poaduai

献给我勇悍的女王，我的灵感女神，我的光芒与挚爱，*Victoria*

Obie Fernandez 的其他著作

Patterns of Application Development Using AI

The Rails 8 Way

The Rails 7 Way

XML The Rails Way

Serverless

El Libro Principiante de Node

The Lean Enterprise

Contents

Gregor Hohpe 的前言

序言

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

关于本书

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

关于代码示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

本书不涵盖的内容

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

本书适合谁

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

建立共同词汇

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

参与其中

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

致谢

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

插图是怎么回事?

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

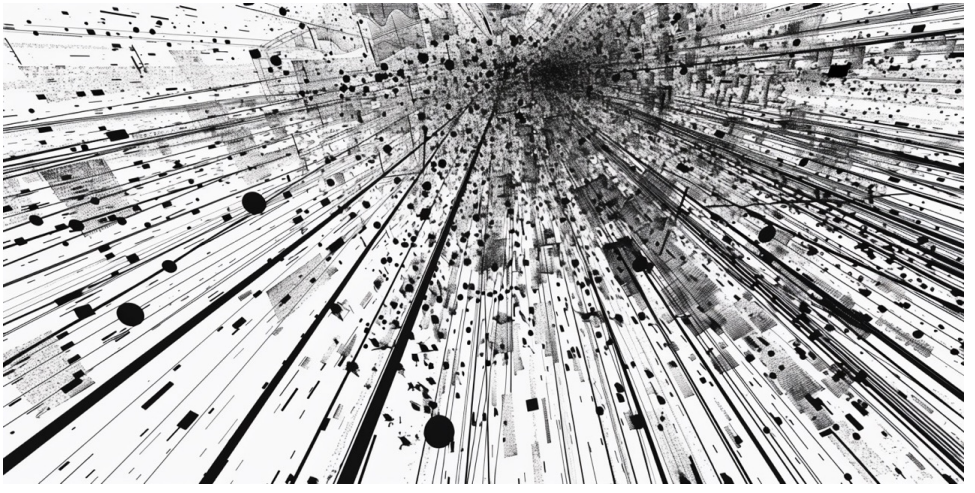
关于精益出版

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

关于作者

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

引言



如果你急于开始将 AI 大语言模型（LLMs）整合到你的编程项目中，可以直接跳到后面章节的模式和代码示例。然而，为了充分理解这些模式的力量和潜力，值得花点时间来理解更广泛的背景和它们所代表的统一方法。

这些模式不仅仅是独立技术的集合，而是一个将 AI 整合到应用程序中的统一框架。我使用 Ruby on Rails，但这些模式几乎可以在任何其他编程环境中使用。它们解决了从数据管理和性能优化到用户体验和安全性等广泛问题，为用 AI 能力增强传统编程实践提供了一个全面的工具包。

每类模式都针对在将 AI 组件整合到应用程序时出现的特定挑战或机遇。通过理解这些模式之间的关系和协同效应，你可以就在何处以及如何最有效地应用 AI 做出明智的决定。

模式从来都不是规定性的解决方案，也不应该被视为这样。它们是可适应的构建块，应该根据你自己独特应用程序的具体需求和限制进行调整。这些模式（像

软件领域的任何其他模式一样）的成功应用依赖于对问题域、用户需求和项目整体技术架构的深入理解。

关于软件架构的思考

我从 20 世纪 80 年代开始编程，曾参与黑客圈子，即使在成为专业软件开发人员后，我也从未失去黑客思维。从一开始，我就对象牙塔中的软件架构师究竟能带来什么价值持有健康的怀疑态度。

我个人如此兴奋于这波强大的新 AI 技术带来的变革，其中一个原因是它对我们认为的软件架构决策的影响。它挑战了关于如何设计和实现软件项目的 “正确 “方式的传统观念。它还质疑架构是否仍然可以主要被视为系统中难以改变的部分，因为 AI 增强正在使得随时改变项目的任何部分变得比以往更容易。

也许我们正在进入软件工程 “后现代 “方法的巅峰时期。在这种情况下，后现代指的是从传统范式的根本性转变，在传统范式中，开发人员负责编写和维护每一行代码。相反，它接受将任务（如数据操作、复杂算法，甚至整块应用程序逻辑）委托给第三方库和外部 API 的理念。这种后现代转变代表着对从头构建应用程序的传统智慧的重大偏离，它挑战开发人员重新思考他们在开发过程中的角色。

基于 Larry Wall和其他像他这样的黑客大师的教导，我一直相信优秀的程序员只编写绝对必要的代码。通过最小化编写的代码量，我们可以更快地移动，减少错误的表面积，简化维护，并提高应用程序的整体可靠性。更少的代码使我们能够专注于核心业务逻辑和用户体验，同时将其他工作委托给其他服务。

现在 AI 驱动的系统可以处理以前只能由人工编写代码完成的任务，我们应该能够更加高效和敏捷，比以往任何时候都更专注于创造商业价值和用户体验。

当然，将项目的大部分委托给 AI 系统也有权衡，比如可能失去控制权，以及需要健全的监控和反馈机制。这就是为什么它需要一套新的技能和知识，包括至少对 AI 工作原理的一些基本理解。

什么是大语言模型?

大语言模型（LLMs）是一种人工智能模型，自从 OpenAI在 2020 年推出 GPT-3 以来，已经引起了广泛关注。LLMs 旨在以显著的准确性和流畅性处理、理解和生成人类语言。在本节中，我们将简要了解 LLMs 如何工作以及为什么它们非常适合构建智能系统组件。

从本质上讲, LLMs 基于深度学习算法, 特别是神经网络。这些网络由相互连接的节点或神经元组成，用于处理和传输信息。LLMs 常用的架构选择是 Transformer 模型，它在处理文本等序列数据方面已经证明非常有效。

Transformer 模型基于注意力机制，主要用于处理序列数据（如自然语言处理）的任务。Transformer 可以同时处理所有输入数据，而不是按顺序处理，这使其能够更有效地捕捉长距离依赖关系。它们具有多层注意力机制，帮助模型关注输入数据的不同部分，以理解上下文和关系。

大语言模型的训练过程涉及让模型接触大量文本数据，如书籍、文章、网站和代码库。在训练过程中，模型学习识别文本中的模式、关系和结构。它捕捉语言的统计特性，例如语法规则、词语关联和上下文含义。

训练大语言模型使用的关键技术之一是无监督学习。这意味着模型在没有明确标注或指导的情况下从数据中学习。它通过分析训练数据中词语和短语的共现来自主发现模式和表示。这使得大语言模型能够深入理解语言及其复杂性。

大语言模型的另一个重要方面是处理上下文的能力。在处理文本时，大语言模型不仅考虑单个词语，还考虑周围的上下文。它们会考虑前面的词语、句子，甚至段落，以理解文本的含义和意图。这种上下文理解使大语言模型能够生成连贯且相关的回应。评估特定大语言模型能力的主要方式之一是考虑它们能够用于生成回应的上下文大小。

经过训练后，大语言模型可用于广泛的语言相关任务。它们可以生成类人文本、回答问题、总结文档、翻译语言，甚至编写代码。大语言模型的多功能性使其成为构建智能系统组件的重要工具，这些组件可以与用户交互、处理和分析文

本数据，并生成有意义的输出。

通过将大语言模型整合到应用程序架构中，你可以创建能够理解和处理用户输入、生成动态内容并提供智能推荐或行动的 AI 组件。但使用大语言模型需要仔细考虑资源需求和性能权衡。大语言模型计算密集，可能需要大量处理能力和内存（换句话说，需要资金）才能运行。我们大多数人都需要评估将大语言模型集成到应用程序中的成本影响，并据此采取行动。

理解推理

推理指的是模型根据新的、未见过的数据生成预测或输出的过程。这是训练好的模型用于根据用户输入做出决策或生成文本、图像或其他内容的阶段。

在训练阶段，AI 模型通过调整其参数来最小化预测错误，从而从大型数据集中学习。训练完成后，模型可以将所学应用于新数据。推理是模型使用其学习到的模式和知识来生成输出的方式。

对于大语言模型来说，推理涉及接收提示词或输入文本，并生成连贯且与上下文相关的响应，形式为词元流（我们稍后会讨论）。这可能是回答问题、完成句子、生成故事或翻译文本等多种任务。

与你我思考的方式不同，AI 模型通过推理进行的 “思考 “都是在一个无状态的操作中完成的。也就是说，它的思考仅限于其生成过程。它必须字面意义上地 “想出声 “，就好像我问你一个问题，而只接受你以 “意识流 “方式的回答。

大语言模型有多种规模和类型

虽然几乎所有流行的大语言模型都基于相同的核心 transformer 架构并在海量文本数据集上训练，但它们有不同的规模，并针对不同目的进行微调。大语言模型的规模（以其神经网络中的参数数量衡量）对其能力有重大影响。具有更多

参数的更大模型，如 GPT-4（据传有 1 到 2 万亿个参数），通常比小型模型知识更丰富，能力更强。然而，更大的模型也需要更多的计算能力才能运行，这意味着通过 API 调用使用它们时费用更高。

为了使大语言模型更实用并适应特定用例，基础模型通常在更有针对性的数据集上进行微调。例如，大语言模型可能在大量对话数据上训练，使其专门用于对话 AI。其他模型在代码上训练以赋予它们编程知识。甚至还有专门训练用于与用户进行角色扮演式互动的模型！

检索式模型与生成式模型的对比

在大语言模型（LLMs）的世界中，生成响应主要有两种方法：检索式模型和生成式模型。每种方法都有其优势和局限性，理解它们之间的差异可以帮助你为特定用例选择合适的模型。

检索式模型

检索式模型，也称为信息检索模型，通过在大型预存文本数据库中搜索并基于输入查询选择最相关的段落来生成响应。这些模型不会从零开始生成新文本，而是将数据库中的片段组合在一起形成连贯的响应。

检索式模型的主要优势之一是能够提供准确和最新的信息。由于它们依赖于经过筛选的文本数据库，它们可以从可靠来源提取相关信息并呈现给用户。这使得它们特别适合需要精确、事实性答案的应用，例如问答系统或知识库。

然而，检索式模型也有一些局限性。它们的表现取决于所搜索的数据库质量，因此数据库的质量和覆盖范围直接影响模型的性能。此外，这些模型可能难以生成连贯和自然的响应，因为它们仅限于数据库中可用的文本。

本书不涉及纯检索模型的使用。

生成式模型

相比之下，生成式模型会根据训练期间学习到的模式和关系从零开始创建新文本。这些模型运用其对语言的理解来生成针对输入提示的新颖响应。

生成式模型的主要优势在于其能够产生富有创意、连贯且与上下文相关的文本。它们可以进行开放式对话、生成故事，甚至编写代码。这使得它们特别适合需要更开放和动态交互的应用，如聊天机器人、内容创作和创意写作助手。

然而，生成式模型有时会产生不一致或事实错误的信息，因为它们依赖于训练期间学习的模式，而不是经过筛选的事实数据库。它们也可能更容易受到偏见和产生幻觉的影响，生成看似合理但不一定真实的文本。

生成式 LLMs 的例子包括 OpenAI 的 GPT 系列（GPT-3、GPT-4）和 Anthropic 的 Claude。

混合模型

多个商业可用的 LLMs 将检索和生成方法结合在混合模型中。这些模型使用检索技术从数据库中找到相关信息，然后使用生成技术将这些信息合成为连贯的响应。

混合模型旨在将检索式模型的事实准确性与生成式模型的自然语言生成能力相结合。它们可以提供更可靠和最新的信息，同时保持进行开放式对话的能力。

在选择检索式模型和生成式模型时，你应该考虑应用的具体要求。如果主要目标是提供准确的事实信息，检索式模型可能是最佳选择。如果应用需要更开放和创造性的交互，生成式模型可能更合适。混合模型在两种方法之间提供了平衡，可能适合同时需要事实准确性和自然语言生成的应用。

最终，检索式模型和生成式模型之间的选择取决于具体用例以及准确性、创造力和灵活性之间的权衡。通过理解每种方法的优势和局限性，你可以做出明智的决定。

指令微调

指令微调是会话训练的一个子集，它使用人工编写的提示和响应来提高 LLMs 理解和遵循指令的能力。以下是一些简单的例子，说明如何通过对人工编写的提示和响应进行指令微调来提高 LLMs 理解和遵循指令的能力：

示例 1：客户支持

指令：“对要求退款的客户提供回复。”

人工编写的提示：“我想为最近的购买申请退款。该产品没有达到我的期望。”

人工编写的响应：“很抱歉听到产品没有达到您的期望。为了处理您的退款，请提供您的订单号和退货原因。我们将审核您的请求，并在 2-3 个工作日内回复您。”

示例 2：教育辅助

指令：“向一个 10 岁的孩子解释光合作用的概念。”

人工编写的提示词：“你能解释植物是如何制造食物的吗？”

人工编写的回答：“当然可以！植物通过光合作用制造食物。它们利用阳光、水和二氧化碳来制造食物和氧气。植物的叶子中有一种特殊的物质叫做叶绿素，帮助它们吸收阳光。”

示例 3：个人助理

指令：“安排下周二下午 3 点的会议。”

人工编写的提示词：“你能帮我安排周二下午 3 点的会议吗？”

人工编写的回答：“当然可以，我已经为您安排了下周二下午 3 点的会议。还需要其他帮助吗？”

这形成了一个包含不同规模和专长的大型语言模型的多样化生态系统。在 1-70 亿参数范围内的较小模型提供了良好的通用语言能力，同时运行效率更高。

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

约 300-700 亿参数的中型模型提供更强的推理和指令遵循能力。

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

在将 LLM 整合到应用程序时，你必须在模型的能力与实际因素（如成本、延迟、上下文长度和内容过滤）之间取得平衡。对于较简单的语言任务，较小的指令微调模型通常是最佳选择，而最大的模型可能用于复杂的推理或分析。模型的训练数据也是一个重要考虑因素，因为它决定了模型的知识截止日期。

某些模型，比如来自 Perplexity的一些模型，可以连接到实时信息源，因此它们实际上没有截止日期。当你向它们提问时，它们能够自主决定进行网络搜索并获取任意网页来生成答案。

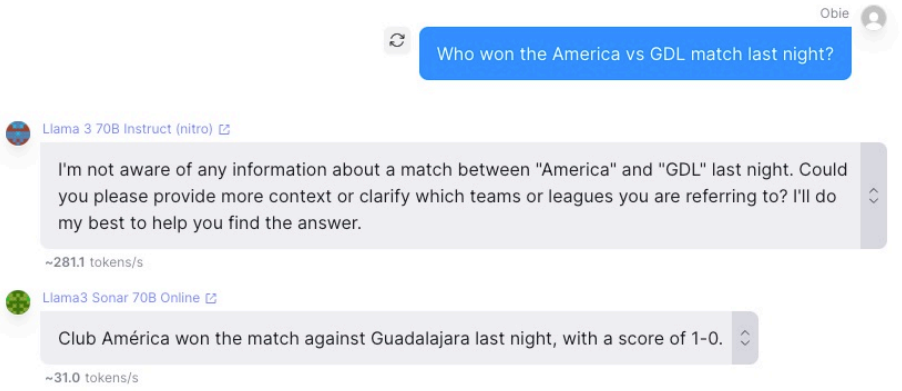


图 1. 带在线访问和不带在线访问的 Llama3 对比

最终，并不存在万能的 LLM。了解模型大小、架构和训练方面的差异对于为特定用例选择正确的模型至关重要。只有通过实验不同的模型，才能实际发现哪些模型能为特定任务提供最佳性能。

分词：将文本分解成片段

在大型语言模型处理文本之前，需要将文本分解成称为词元的较小单位。词元可以是单个词、词的部分或单个字符。将文本分割成词元的过程称为分词，这是为语言模型准备数据的关键步骤。

图 2. 这个句子包含 27 个词元

不同的 LLM 使用不同的分词策略，这会对模型的性能和能力产生重要影响。LLM 使用的一些常见分词器包括：

- **GPT（字节对编码）：**GPT 分词器使用一种称为字节对编码（BPE）的技术将文本分解为子词单位。BPE 通过迭代合并文本语料库中最频繁出现的

字节对来形成子词词元词汇表。这使得分词器能够通过将罕见和新词分解成更常见的子词片段来处理它们。GPT 分词器被 GPT-3和 GPT-4等模型使用。

- **Llama（SentencePiece）：**Llama 分词器使用 SentencePiece 库，这是一个无监督的文本分词器和解分词器。SentencePiece 将输入文本视为 Unicode 字符序列，并基于训练语料库学习子词词汇表。它可以处理任何能够用 Unicode 编码的语言，这使它特别适合多语言模型。Llama 分词器被 Meta 的 Llama和 Alpaca等模型所使用。

- **SentencePiece（Unigram）：**SentencePiece 分词器还可以使用一种称为 Unigram 的不同算法，该算法基于子词正则化技术。Unigram 分词通过一元语言模型确定最优子词词汇表，该模型为单个子词单元分配概率。与 BPE 相比，这种方法可以产生更具语义意义的子词。使用 Unigram 的 SentencePiece 被 Google 的 T5和 BERT等模型所采用。

- **Google Gemini（多模态分词）：**Google Gemini 使用一种设计用于处理各种数据类型的分词方案，包括文本、图像、音频、视频和代码。这种多模态能力使 Gemini 能够处理和整合不同形式的信息。值得注意的是，Google Gemini 1.5 Pro具有可以处理数百万个词符的上下文窗口，远大于之前的模型。这种广泛的上下文窗口使模型能够处理更大的上下文，可能导致更准确的响应。然而，需要注意的是，Gemini 的分词方案比其他模型更接近于每个字符对应一个词符。这意味着如果你习惯使用 GPT 等模型，使用 Gemini 模型的实际成本可能会显著高于预期，因为 Google 的定价是基于字符而不是词符。

分词器的选择会影响 LLM 的多个方面，包括：

- ** 词汇表大小：** 分词器决定了模型词汇表的大小，即它能识别的唯一词符集合。更大、更细粒度的词汇表可以帮助模型处理更广泛的词语和短语，甚至成为多模态的（能够理解和生成不仅仅是文本），但这也会增加模型的内存需求和计算复杂性。

- ** 处理罕见词和未知词：** 使用子词单元的分词器，如 BPE和 SentencePiece，可以将罕见词和未知词分解为更常见的子词片段。这使得模型能够基于这些子词对它从未见过的词语的含义进行有根据的推测。
- ** 多语言支持：**像 SentencePiece 这样可以处理任何 Unicode 可编码语言的分词器，特别适合需要处理多种语言文本的多语言模型。

在为特定应用选择 LLM 时，考虑其使用的分词器以及该分词器与任务特定语言处理需求的匹配程度非常重要。分词器可能对模型处理领域特定术语、罕见词和多语言文本的能力产生重大影响。

上下文大小：语言模型在推理过程中能使用多少信息？

在讨论语言模型时，上下文大小指的是模型在处理或生成响应时能考虑的文本量。本质上，它衡量的是模型能 “记住 “和用于指导输出的信息量（以词符表示）。语言模型的上下文大小能显著影响其功能和有效执行任务的类型。

什么是上下文大小？

从技术角度来说，上下文大小由语言模型在单个输入序列中能处理的词符（单词或词片）数量决定。这通常被称为模型的 “注意力跨度 “或 “上下文窗口“。上下文大小越大，模型在生成响应或执行任务时能同时考虑的文本就越多。

不同的语言模型具有不同的上下文大小，从几百个词符到数百万个词符不等。作为参考，一个典型的段落可能包含约 100-150 个词符，而整本书可能包含数万或数十万个词符。

> 甚至有研究在探索高效方法，使基于 Transformer 的大语言模型（LLMs）能够处理无限长的输入，同时保持有限的内存和计算量。

为什么上下文大小如此重要?

语言模型的上下文大小对其理解和生成连贯、符合语境的文本的能力有重大影响。以下是上下文大小重要性的几个关键原因:

1. ** 理解长篇内容:** 具有更大上下文窗口的模型能够更好地理解和分析较长的文本,如文章、报告,甚至整本书。这对于文档摘要、问答和内容分析等任务至关重要。

2. ** 保持连贯性:** 更大的上下文窗口使模型能够在更长的输出内容中保持连贯性和一致性。这对于故事生成、对话系统和内容创作等任务非常重要,因为在这些任务中保持叙事或主题的一致性至关重要。在使用大语言模型生成或转换结构化数据时,这一点也绝对至关重要。
3. ** 捕捉长程依赖:** 某些语言任务需要理解文本中相距较远的词语或短语之间的关系。具有更大上下文大小的模型更能够捕捉这些长程依赖关系,这对于情感分析、翻译和语言理解等任务非常重要。
4. ** 处理复杂指令:** 在使用语言模型执行复杂的多步骤指令的应用中,更大的上下文大小使模型能够在生成响应时考虑完整的指令集,而不仅仅是最近的几个词。

不同上下文大小的语言模型示例

以下是一些具有不同上下文大小的语言模型示例:

- OpenAI GPT-3.5 Turbo:4,095 个词元
- Mistral 7B Instruct:32,768 个词元
- Anthropic Claude v1:100,000 个词元
- OpenAI GPT-4 Turbo:128,000 个词元
- Anthropic Claude v2:200,000 个词元
- Google Gemini Pro 1.5:280 万个词元

如您所见，这些模型的上下文大小差异很大，从 OpenAI GPT-3.5 Turbo 模型的约 4,000 个词元到 Anthropic Claude v2 模型的 200,000 个词元不等。某些模型，如 Google 的 PaLM 2和 OpenAI 的 GPT-4，提供具有更大上下文大小的不同变体（例如 “32k“版本），可以处理更长的输入序列。而目前（2024 年 4 月）Google Gemini Pro 声称可以处理近 300 万个词元！

值得注意的是，上下文大小会因特定模型的实现和版本而异。例如，原始的 OpenAI GPT-4 模型的上下文大小为 8,191 个词元，而后来的 GPT-4 变体（如 Turbo 和 4o）的上下文大小大大增加到了 128,000 个词元。

> Sam Altman将当前的上下文限制比作 80 年代个人电脑程序员必须处理的几千字节工作内存，并表示在不久的将来，我们将能够将 “所有个人数据 “都放入大语言模型的上下文中。

选择合适的上下文大小

在为特定应用选择语言模型时，考虑任务所需的上下文大小要求很重要。对于涉及短小、独立文本片段的任务（如情感分析或简单问答），较小的上下文大小可能就足够了。然而，对于需要理解和生成更长、更复杂文本的任务，可能需要更大的上下文大小。

值得注意的是，更大的上下文大小通常会带来更高的计算成本和更慢的处理时间，因为模型在生成响应时需要考虑更多信息。因此，在为应用选择语言模型时，必须在上下文大小和性能之间取得平衡。

> 为什么不直接选择具有最大上下文大小的模型，并尽可能多地填充信息呢？除了性能因素外，另一个主要考虑因素是成本。在 2024 年 3 月，使用 Google Gemini Pro 1.5的完整上下文进行一次提示-响应循环的成本将接近 8 美元。如

> 果您有能够证明这种支出合理的使用场景，那就太好了！但对于大多数应用来说，这个成本实在太高了，差距达到了数个数量级。

大海捞针

在大型数据集中进行检索的挑战常被比喻为大海捞针。在大型语言模型（LLM）领域，我们稍微改变了这个比喻。想象一下，我们不是在浩瀚的文本中寻找单一事实（比如在 Paul Graham 的文章全集中），而是在寻找分散各处的多个事实。这种情况更像是在广阔的原野中寻找多根针，而不是在单一的草堆中寻找。关键在于：我们不仅需要找到这些针，还要将它们编织成一个连贯的线索。

当需要从长文本中检索和推理多个事实时，LLM 面临着双重挑战。首先是检索准确性的直接问题——随着事实数量的增加，准确性自然会下降。这是可以预期的；毕竟，在庞大的文本中追踪多个细节即使对最复杂的模型来说也是一种负担。

其次，可能更关键的是对这些事实进行推理的挑战。找出事实是一回事，将它们综合成连贯的叙述或答案则是另一回事。这才是真正的考验所在。LLM 在推理任务中的表现往往比简单的检索任务下降得更多。这种退化不仅仅关乎数量；它涉及上下文、相关性和推理之间的复杂互动。

为什么会这样？考虑一下人类认知中的记忆和注意力动态，这在某种程度上也反映在 LLM 中。在处理大量信息时，LLM 和人类一样，在吸收新信息的过程中可能会遗失先前的细节。这种情况在那些没有被特别设计为自动优先处理或重新访问早期文本段落的模型中尤其明显。

此外，LLM 将这些检索到的事实编织成连贯回应的能力类似于叙事构建。这不仅需要信息检索，还需要深入的理解和上下文定位，这对当前的人工智能来说仍然是一个巨大的挑战。

那么，对于我们这些技术的开发者和集成者来说，这意味着什么？在设计依赖 LLM 处理复杂长文本任务的系统时，我们需要敏锐地意识到这些限制。理解在

某些条件下性能可能会下降，有助于我们设定现实的期望，并设计更好的后备机制或补充策略。

模态：超越文本

虽然当今大多数语言模型都专注于处理和生成文本,但越来越多的多模态模型能够原生输入和输出多种类型的数据，如图像、音频和视频。这些多模态模型为能够理解和生成跨模态内容的人工智能应用开辟了新的可能性。

什么是模态?

在语言模型的背景下，模态指的是模型可以处理和生成的不同类型的数据。最常见的模态是文本，包括各种形式的书面语言，如书籍、文章、网站和社交媒体帖子。然而，还有几种其他模态正越来越多地被整合到语言模型中:

- **图像：**视觉数据，如照片、插图和图表。
- **音频：**声音数据，如语音、音乐和环境声音。
- **视频：**移动的视觉数据，通常伴随着音频，如视频剪辑和电影。

每种模态都为语言模型带来独特的挑战和机遇。例如，图像要求模型理解视觉概念和关系，而音频则要求模型处理和生成语音及其他声音。

多模态语言模型

多模态语言模型被设计用来在单一模型中处理多种模态。这些模型通常具有专门的组件或层，可以理解输入并在不同模态中生成输出数据。一些著名的多模态语言模型包括:

- **OpenAI 的 GPT-4o：** GPT-4o 是一个除了文本之外还能原生理解和处理语音音频的大型语言模型。这种能力使 GPT-4o 能够执行诸如转录口语、从音频输入生成文本以及根据口头查询提供响应等任务。

- **OpenAI 的 GPT-4 及其视觉输入功能：** GPT-4是一个可以处理文本和图像的大型语言模型。当给定图像作为输入时，GPT-4 可以分析图像内容并生成描述或响应视觉信息的文本。
- **谷歌的 Gemini：** Gemini是一个可以处理文本、图像和视频的多模态模型。它使用统一的架构实现跨模态理解和生成，能够执行图像描述生成、视频摘要和视觉问答等任务。

- **DALL-E 和 Stable Diffusion：** 虽然这些不是传统意义上的语言模型，但它们通过将文本描述转化为图像来展示了多模态 AI 的强大力量。它们展示了能够在不同模态之间转换的模型的潜力。

多模态模型的优势与应用

多模态语言模型提供了多项优势，并支持广泛的应用，包括：

- ** 增强理解能力：** 通过处理来自多个模态的信息，这些模型能够获得对世界更全面的理解，类似于人类从各种感官输入中学习的方式。
- ** 跨模态生成：**多模态模型能够基于一种模态的输入生成另一种模态的内容，例如根据文本描述创建图像，或根据书面文章生成视频摘要。
- ** 可访问性：** 多模态模型可以通过在不同模态之间转换来使信息更容易获取，例如为视障用户生成图像的文本描述，或创建书面内容的音频版本。
- ** 创意应用：** 多模态模型可用于创意任务，如根据文本提示生成艺术、音乐或视频，为艺术家和内容创作者开启新的可能性。

随着多模态语言模型的不断发展，它们很可能在能够理解和生成跨多个模态内容的 AI 驱动应用程序的发展中发挥越来越重要的作用。这将实现人类与 AI 系统之间更自然、更直观的互动，并为创意表达和知识传播开启新的可能性。

提供商生态系统

在将大型语言模型（LLM）整合到应用程序中时，你有越来越多的选择。每个主要的 LLM 提供商，如 OpenAI、Anthropic、Google和 Cohere，都提供其独特的模型、API 和工具生态系统。选择合适的提供商需要考虑多个因素，包括定价、性能、内容过滤、数据隐私和定制选项。

OpenAI

OpenAI 是最著名的 LLM 提供商之一，其 GPT 系列（GPT-3、GPT-4）在各种应用中被广泛使用。OpenAI 提供了一个用户友好的 API，使你能够轻松地将他们的模型集成到应用程序中。他们提供了一系列具有不同功能和价格点的模型，从入门级的 Ada 模型到强大的 Davinci 模型。

OpenAI 的生态系统还包括 OpenAI Playground 等工具，允许你试验提示并为特定用例微调模型。他们提供内容过滤选项，以帮助防止生成不当或有害内容。

在直接使用 OpenAI 的模型时，我依赖 Alex Rudall的ruby-openai库。

Anthropic

Anthropic 是 LLM 领域的另一个主要参与者，其 Claude 模型因强大的性能和伦理考虑而日益受欢迎。Anthropic 专注于开发安全和负责任的 AI 系统，特别强调内容过滤和避免有害输出。

Anthropic 的生态系统包括 Claude API，允许你将模型集成到应用程序中，以及用于提示工程和微调的工具。他们还提供 Claude Instant 模型，该模型集成了网络搜索功能，以提供更新和更准确的响应。

在直接使用 Anthropic 的模型时，我依赖 Alex Rudall 的anthrophic库。

Google

Google 开发了多个强大的 LLM，包括 Gemini、BERT、T5和 PaLM。这些模型以在广泛的自然语言处理任务中的强大性能而闻名。Google 的生态系统包括

TensorFlow 和 Keras 库，这些库提供了用于构建和训练机器学习模型的工具和框架。

Google 还提供了 Cloud AI Platform，使你能够轻松地在云端部署和扩展他们的模型。他们为情感分析、实体识别和翻译等任务提供了一系列预训练模型和 API。

Meta

Meta，前身为 Facebook，通过发布 LLaMA 和 OPT等模型，深度投入大型语言模型的开发。这些模型以在各种语言任务中的出色表现而著称，主要通过开源渠道提供，体现了 Meta 对研究和社区协作的承诺。

Meta 的生态系统主要围绕 PyTorch构建，这是一个开源机器学习库，因其动态计算能力和灵活性而备受欢迎，促进了创新的 AI 研究和开发。

除了技术产品外，Meta 还非常重视人工智能的道德发展。他们实施强大的内容过滤系统并致力于减少偏见，这与他们在人工智能应用方面追求安全和责任的更广泛目标保持一致。

Cohere

Cohere 是大语言模型领域的新晋力量，专注于使大语言模型比竞争对手更易访问和使用。他们的生态系统包括 Cohere API，该 API 提供了一系列用于文本生成、分类和摘要等任务的预训练模型。

Cohere 还提供提示工程、微调和内容过滤的工具。他们强调数据隐私和安全性，具有加密数据存储和访问控制等功能。

Ollama

Ollama 是一个自托管平台，允许用户在本地机器上管理和部署各种大语言模型（LLMs），让用户能够完全控制其人工智能模型，而无需依赖外部云服务。对于重视数据隐私并希望在内部处理人工智能操作的用户来说，这种设置是理想的选择。

该平台支持多种模型，包括 Llama、Phi、Gemma 和 Mistral的各种版本，这些模型在大小和计算需求方面各不相同。Ollama 使用简单的命令如`ollama run <model_name>` 就能直接从命令行下载和运行这些模型，并且设计为可以在 macOS、Linux 和 Windows 等不同操作系统上运行。

对于希望将开源模型集成到应用程序中而不使用远程 API 的开发人员来说，Ollama 提供了一个用于管理模型生命周期的 CLI，类似于容器管理工具。它还支持自定义配置和提示，允许高度定制化以使模型适应特定需求或用例。

由于其命令行界面和在管理和部署人工智能模型方面提供的灵活性，Ollama 特别适合技术娴熟的用户和开发人员。这使其成为需要强大人工智能功能但不想在安全性和控制方面妥协的企业和个人的有力工具。

多模型平台

此外，还有一些提供商托管各种开源模型，如 Together.ai 和 Groq。这些平台提供灵活性和定制化选项，允许你运行，在某些情况下甚至可以根据具体需求微调开源模型。例如，Together.ai 提供对多种开源大语言模型的访问，使用户能够试验不同的模型和配置。Groq 专注于提供超高性能的补全功能，在本书写作时看起来几乎是魔法般的表现。

选择大语言模型提供商

在选择大语言模型提供商时，你应该考虑以下因素：

- ** 定价：** 不同提供商提供不同的定价模式，从按使用付费到基于订阅的计划都有。在选择提供商时，考虑预期使用量和预算很重要。
- ** 性能：** 大语言模型的性能在不同提供商之间可能有显著差异，因此在做出决定之前，对特定用例进行基准测试和测试很重要。
- ** 内容过滤：** 根据应用程序的不同，内容过滤可能是一个关键考虑因素。某些提供商提供比其他提供商更强大的内容过滤选项。

- ** 数据隐私：** 如果应用程序处理敏感用户数据，选择具有强大数据隐私和安全实践的提供商很重要。
- ** 定制化：** 某些提供商在微调和定制模型以适应特定用例方面提供更多灵活性。

最终，大语言模型提供商的选择取决于应用程序的具体要求和限制。通过仔细评估选项并考虑定价、性能和数据隐私等因素，你可以选择最适合你需求的提供商。

值得注意的是，大语言模型领域在不断发展，新的提供商和模型经常出现。你应该及时了解最新发展，并对探索新的选择持开放态度。

OpenRouter

在本书中，我将专门使用OpenRouter作为我的首选 API 提供商。原因很简单：它是所有最流行的商业和开源模型的一站式商店。如果你迫不及待想要开始进行一些人工智能编程，最好的起点之一是使用我的OpenRouter Ruby 库。

性能考量

在将语言模型整合到应用程序中时，性能是一个关键考虑因素。语言模型的性能可以通过其延迟（生成响应所需的时间）和吞吐量（每单位时间可处理的请求数）来衡量。

首字符生成时间（TTFT）是另一个重要的性能指标，这对聊天机器人和需要交互式实时响应的应用程序特别重要。TTFT 衡量从接收用户请求到生成响应的第一个词（或标记）之间的延迟时间。这个指标对于维持流畅且具有吸引力的用户体验至关重要，因为响应延迟可能导致用户感到沮丧并失去兴趣。

这些性能指标可能会对用户体验和应用程序的可扩展性产生重大影响。

几个因素可能会影响语言模型的性能，包括：

参数数量：较大的模型具有更多参数，通常需要更多的计算资源，与较小的模型相比可能具有更高的延迟和更低的吞吐量。

硬件：语言模型的性能可能会因运行它的硬件而显著不同。云服务提供商提供针对机器学习工作负载优化的 GPU 和 TPU 实例，这可以大大加快模型推理速度。

OpenRouter的一个优点是，对于它提供的许多模型，你可以选择具有不同性能配置和成本的云服务提供商。

量化：量化技术可以通过使用更低精度的数据类型来表示权重和激活值，从而减少模型的内存占用和计算需求。这可以在不显著牺牲质量的情况下提高性能。作为应用程序开发人员，你可能不会参与训练不同量化级别的自己的模型，但至少熟悉这些术语是有好处的。

** 批处理：** 同时以批次处理多个请求可以通过分摊模型加载和数据传输的开销来提高吞吐量。

** 缓存：** 缓存常用提示或输入序列的结果可以减少推理请求的数量并提高整体性能。

在为生产应用程序选择语言模型时，重要的是要在代表性工作负载和硬件配置上对其性能进行基准测试。这可以帮助识别潜在的瓶颈，并确保模型能够满足所需的性能目标。

同样值得考虑的是模型性能与其他因素（如成本、灵活性和集成便利性）之间的权衡。例如，对于需要实时响应的应用程序，使用延迟较低的较小、较便宜的模型可能更可取，而对于批处理或复杂推理任务，较大、更强大的模型可能更合适。

尝试不同的大语言模型

选择大语言模型（LLM）很少是一个永久性的决定。由于新的和改进的模型会定期发布，以模块化方式构建应用程序是很好的做法，这样可以随时间更换不

同的语言模型。提示和数据集通常可以在模型之间重复使用，只需要进行最小的更改。这使你能够利用语言建模的最新进展，而无需完全重新设计应用程序。

能够轻松地在各种模型选择之间切换，这是我喜欢 OpenRouter 的另一个原因。

在升级到新的语言模型时，重要的是要彻底测试和验证其性能和输出质量，以确保它满足应用程序的要求。这可能涉及在特定领域数据上重新训练或微调模型，以及更新依赖于模型输出的任何下游组件。

通过在设计应用程序时考虑性能和模块化，你可以创建可扩展、高效且面向未来的系统，这些系统能够适应快速发展的语言建模技术领域。

复合人工智能系统

在结束我们的介绍之前，值得一提的是，在 2023 年之前以及由 ChatGPT引发的生成式人工智能热潮之前，传统的人工智能方法通常依赖于单一、封闭模型的集成。相比之下，复合人工智能系统利用相互连接的组件的复杂管道协同工作来实现智能行为。

从本质上讲，复合人工智能系统由多个模块组成，每个模块都设计用于执行特定的任务或功能。这些模块可以包括生成器、检索器、排序器、分类器和各种其他专用组件。通过将整个系统分解为更小的、重点突出的单元，开发人员可以创建更灵活、可扩展和可维护的人工智能架构。

复合人工智能系统的一个关键优势在于它能够结合不同人工智能技术和模型的优势。例如，系统可能使用大语言模型（LLM）来进行自然语言理解和生成，同时采用独立的模型进行信息检索或基于规则的决策制定。这种模块化方法使您能够为每个具体任务选择最佳的工具和技术，而不是依赖于一刀切的解决方案。

然而，构建复合人工智能系统也带来了独特的挑战。特别是，确保系统行为的整体连贯性和一致性需要强大的测试、监控和治理机制。

像 GPT-4这样强大的大语言模型的出现使我们比以往任何时候都更容易实验复合人工智能系统，因为这些先进的模型除了具备自然语言理解能力外，还能在复合系统中担任多个角色，如分类、排序和生成。这种多功能性使开发人员能够快速原型设计并迭代复合人工智能架构，为智能应用程序开发开辟了新的可能性。

复合人工智能系统的部署模式

复合人工智能系统可以使用各种模式部署，每种模式都旨在满足特定的需求和用例。让我们探讨四种常见的部署模式：问答系统、多智能体/主动式问题解决器、对话式人工智能和协作助手。

问答系统

问答（Q&A）系统专注于提供经人工智能模型理解能力增强的信息检索，使其不仅仅是一个搜索引擎。通过将强大的语言模型与使用检索增强生成（RAG）的外部知识源相结合，问答系统可以避免产生幻觉，并为用户查询提供准确且与上下文相关的响应。

基于大语言模型的问答系统的关键组件包括：

- **查询理解和重构：**分析用户查询并重新构造它们，以更好地匹配底层知识源。
- **知识检索：**基于重构后的查询从结构化或非结构化数据源中检索相关信息。
- **响应生成：**通过整合检索到的知识与语言模型的生成能力，生成连贯且信息丰富的响应。

RAG 子系统在提供准确和最新信息至关重要的问答领域特别重要，如客户支持、知识管理或教育应用。

多智能体/主动式问题解决器

多智能体系统（也称为主动式系统）由多个自主智能体协同工作来解决复杂问题。每个智能体都有特定的角色、技能集和对相关工具或信息源的访问权限。通过协作和信息交换，这些智能体可以处理单个智能体难以或无法独立完成的任务。

多智能体问题解决器的关键原则包括:

- **专业化：**每个智能体专注于问题的特定方面，发挥其独特的能力和知识。
- **协作：**智能体通过消息传递或共享内存等方式进行通信和协调行动，以实现共同目标。
- **适应性：**系统可以通过调整个别智能体的角色和行为来适应不断变化的条件或需求。

多智能体系统特别适用于需要分布式问题解决的应用，如供应链优化、交通管理或应急响应规划。

对话式人工智能

对话式人工智能系统实现了用户与智能代理之间的自然语言交互。这些系统结合了自然语言理解、对话管理和语言生成能力，提供引人入胜且个性化的对话体验。

对话式人工智能系统的主要组件包括:

- **意图识别：**基于用户输入识别其意图，如提问、发出请求或表达情感。
- **实体提取：**从用户输入中提取相关实体或参数，如日期、位置或产品名称。
- **对话管理：**维护对话状态，基于用户意图和上下文确定适当的响应，并处理多轮交互。
- **响应生成：**使用语言模型、模板或基于检索的方法生成类人响应。

对话式人工智能系统常用于客服聊天机器人、虚拟助手和语音控制界面。如前所述，本书中的大多数方法、模式和代码示例都直接源自我在一个名为Olympia的大型对话式人工智能系统上的工作。

智能助手

智能助手是与人类用户协同工作的人工智能助手，用于提升用户的生产力和决策能力。这些系统结合了自然语言处理、机器学习和特定领域知识，提供智能推荐、自动化任务处理和情境支持。

智能助手的主要特点包括：

- **个性化：**适应个别用户的偏好、工作流程和沟通方式。
- **主动协助：**预测用户需求，无需明确指令即可提供相关建议或行动。
- **持续学习：**通过用户反馈、交互和数据不断改进性能。

智能助手在多个领域得到广泛应用，如软件开发（例如代码补全和错误检测）、创意写作（例如内容建议和编辑），以及数据分析（例如洞察和可视化推荐）。

这些部署模式展示了复合人工智能系统的多功能性和潜力。 通过理解每种模式的特点和使用场景，你可以在设计和实现智能应用时做出明智的决策。虽然本书并非专门讨论复合人工智能系统的实现，但在将独立人工智能组件集成到传统应用开发中时，许多（如果不是全部的话）相同的方法和模式都适用。

复合人工智能系统中的角色

复合人工智能系统建立在相互连接的模块基础之上，每个模块都设计用于执行特定角色。这些模块协同工作，创造智能行为并解决复杂问题。在考虑在应用程序中何处可以实现或替换为独立 AI 组件时，熟悉这些角色是很有帮助的。

生成器

生成器负责基于学习到的模式或输入提示生成新的数据或内容。人工智能领域有许多不同类型的生成器，但在本书展示的语言模型背景下，生成器可以创建类人文本、完成部分句子或生成对用户查询的响应。它们在内容创作、对话生成和数据增强等任务中发挥着关键作用。

检索器

检索器用于从大型数据集或知识库中搜索和提取相关信息。它们采用语义搜索、关键词匹配或向量相似度等技术，根据给定的查询或上下文找到最相关的数据点。检索器对于需要快速访问特定信息的任务至关重要，如问答系统、事实核查或内容推荐。

排序器

排序器负责根据特定标准或相关性分数对一组项目进行排序或确定优先级。 它们为每个项目分配权重或分数，然后相应地进行排序。排序器常用于搜索引擎、推荐系统或任何需要向用户呈现最相关结果的应用程序。

分类器

分类器用于根据预定义的类别对数据点进行分类或标记。它们从标记的训练数据中学习，然后预测新的、未见过的实例的类别。分类器是情感分析、垃圾邮件检测或图像识别等任务的基础，这些任务的目标是为每个输入分配特定类别。

工具和代理

除了这些核心角色外，复合人工智能系统通常还包含工具和代理来增强其功能和适应性：

- **工具：**工具是执行特定操作或计算的独立软件组件或 API。它们可以被其他模块（如生成器或检索器）调用，以完成子任务或收集额外信息。工具的例子包括网络搜索引擎、计算器或数据可视化库。
- **代理：**代理是能够感知环境、做出决策并采取行动以实现特定目标的自主实体。它们通常依赖规划、推理和学习等多种人工智能技术的组合，以在动态或不确定的条件下有效运作。代理可用于模拟复杂行为或协调复合人工智能系统内多个模块的行动。

在纯粹的复合人工智能系统中，这些组件之间的交互通过定义明确的接口和通信协议进行协调。数据在模块之间流动，一个组件的输出作为另一个组件的输入。这种模块化架构允许灵活性、可扩展性和可维护性，因为可以更新、替换或扩展单个组件而不影响整个系统。

通过利用这些组件及其交互的力量，复合人工智能系统可以解决需要结合不同人工智能能力的复杂现实问题。在探索将人工智能集成到应用程序开发中的方法和模式时，请记住，复合人工智能系统中使用的相同原则和技术可以用于创建智能、自适应和以用户为中心的应用程序。

在第一部分的后续章节中，我们将深入探讨将人工智能组件集成到应用程序开发过程中的基本方法和技术。从提示工程和检索增强生成到自修复数据和智能工作流程编排，我们将涵盖广泛的模式和最佳实践，帮助你构建前沿的人工智能驱动应用程序。

第一部分：基本方法与技术

本书这一部分介绍了在应用程序中集成 AI 使用的不同方式。这些章节涵盖了一系列相关的方法和技术，从较高层面的概念如缩小路径和检索增强生成，一直到如何在 LLM 聊天补全 API 之上编程构建自己的抽象层的具体思路。

本部分的目标是帮助你理解可以通过 AI 实现的各种行为，然后再深入到第二部分所关注的具体实现模式。

第一部分的方法基于我在代码中使用过的想法、企业应用架构与集成的经典模式，以及我在向其他人（包括非技术业务相关者）解释 AI 功能时使用的比喻。

缩窄路径

“缩窄路径”指的是让 AI 专注于当前的任务。每当我对 AI 表现 “愚蠢 “或出乎意料而感到沮丧时，我都会用这句话作为心法。这个心法提醒我，失败可能是我的错，我可能需要进一步缩窄路径。

缩窄路径的需求源于大语言模型中所包含的海量知识，特别是像 OpenAI和 Anthropic这样的世界级模型，它们 literally 拥有数万亿个参数。

拥有如此广泛的知识无疑是强大的，并产生了诸如心智理论和类人推理能力等涌现行为。然而，这种惊人的信息量在针对特定提示生成精确和准确的响应时

也带来了挑战，特别是当这些提示需要表现出可以与 “普通 “软件开发和算法集成的确定性行为时。

这些挑战源于多个因素。

** 信息过载：** 大语言模型接受了跨越各个领域、来源和时期的海量数据训练。这种广泛的知识使它们能够参与各种话题，并基于对世界的广泛理解生成响应。然而，在面对特定提示时，模型可能难以过滤掉不相关、矛盾或过时/废弃的信息，导致响应缺乏重点或准确性。根据你想要完成的任务，模型可获取的矛盾信息的庞大数量很容易压倒其提供你所寻求的答案或行为的能力。

** 上下文歧义：** 考虑到知识的广阔潜在空间，大语言模型在试图理解你的提示的上下文时可能会遇到歧义。如果没有适当的缩窄或引导，模型可能会生成与你的意图只有切向关系但并不直接相关的响应。这种类型的失败会导致响应偏离主题、不一致或无法满足你所述的需求。在这种情况下，缩窄路径指的是上下文消歧，确保你提供的上下文使模型只关注其基础知识中最相关的信息。

注意：当你刚开始 “提示工程“时，你更可能会要求模型做事而没有适当解释期望的结果；要避免模糊不清需要练习！

** 时间不一致性：** 由于语言模型是在不同时期创建的数据上训练的，它们可能拥有过时的、被取代的或不再准确的知识。例如，关于当前事件、科学发现或技术进步的信息可能在模型的训练数据收集后已经发生了变化。如果不缩窄路径以优先考虑更近期和可靠的来源，模型可能会基于过时或错误的信息生成响应，导致其输出中出现不准确和不一致。

** 领域特定的细微差别：** 不同的领域和学科都有其特定的术语、惯例和知识库。想想几乎任何 TLA（三字母缩写）你就会意识到，它们中的大多数都有不止一个含义。例如，MSK 可以指代 Amazon 的 Managed Streaming for Apache Kafka、Memorial Sloan Kettering Cancer Center，或人体的肌肉骨骼（MusculoSKeletal）系统。

当一个提示需要特定领域的专业知识时，大语言模型的通用知识可能不足以提

供准确和细致的响应。通过提示工程或检索增强生成来缩窄路径，专注于特定领域的信息，可以使模型生成更符合你的特定领域要求和期望的响应。

潜在空间：难以理解的浩瀚

当我提到语言模型的 “潜在空间 “时，我指的是模型在训练过程中学习到的知识和信息的广阔、多维景观。它就像模型神经网络中的一个隐藏领域，存储着所有语言的模式、关联和表征。

想象你正在探索一片广阔的未知领地，里面充满了无数相互连接的节点。每个节点代表模型学习到的一条信息、一个概念或一种关系。当你在这个空间中导航时，你会发现有些节点彼此更近，表示它们之间有强烈的联系或相似性，而其他节点则相距较远，暗示着较弱或更远的关系。

潜在空间的挑战在于它极其复杂且具有高维度特性。可以将其想象成像我们的物理宇宙一样浩瀚，有着星系团以及星系之间难以想象的巨大虚空。

由于它包含数千个维度，潜在空间无法被人类直接观察或理解。它是模型内部用于处理和生成语言的抽象表示。当你向模型提供输入提示时，它本质上是将该提示映射到潜在空间中的特定位置。然后，模型利用该空间中的周围信息和连接来生成响应。

问题在于，模型从其训练数据中学习了大量信息，而并非所有信息都与特定任务相关或准确。这就是为什么缩小路径变得如此重要。通过在提示中提供清晰的指令、示例和上下文，你实际上是在引导模型关注潜在空间中与你期望输出最相关的特定区域。

另一种思考方式是把它比作在完全黑暗的博物馆中使用聚光灯。如果你曾经参观过卢浮宫或大都会艺术博物馆，那就是我所说的规模。潜在空间就像这个博物馆，充满了无数的物品和细节。你的提示就像聚光灯，照亮特定区域，引导模型注意最重要的信息。没有这种引导，模型可能会在潜在空间中漫无目的地游走，在途中收集到不相关或矛盾的信息。

在使用语言模型和制作提示时，请记住潜在空间的概念。你的目标是有效地在这个庞大的知识景观中导航，引导模型朝向与你的任务最相关和最准确的信息。通过缩小路径并提供清晰的指导，你可以释放模型潜在空间的全部潜力，生成高质量、连贯的响应。

虽然前面对语言模型和它们所导航的潜在空间的描述可能看起来有点神奇或抽象，但重要的是要理解提示并不是咒语或咒文。语言模型的工作方式是建立在线性代数和概率论的原理之上的。

从本质上讲，语言模型是文本的概率模型，就像钟形曲线是数据的统计模型一样。它们通过一个称为自回归建模的过程进行训练，模型学会根据序列中前面的词来预测下一个词的概率。在训练过程中，模型从随机权重开始，逐步调整这些权重，以便对与其训练样本相似的文本赋予更高的概率。

然而，将语言模型简单地视为统计模型（如线性回归）并不能为理解它们的行为提供最好的直觉。一个更恰当的类比是将它们视为概率程序，这种模型允许操作随机变量，并且可以表示复杂的统计关系。

概率程序可以通过图形模型来表示，这为理解模型中变量之间的依赖关系和关联提供了一种可视化的方式。这种视角可以为理解像 GPT-4和 Claude这样的复杂文本生成模型的工作原理提供有价值的见解。

在 Dohan 等人的论文 “Language Model Cascades“中，作者深入探讨了如何将概率程序应用于语言模型。他们展示了如何使用这个框架来理解这些模型的行为，并指导开发更有效的提示策略。

从这种概率视角得到的一个关键见解是，语言模型本质上创造了一个通向另一个存在所需文档的宇宙的入口。模型根据概率为所有可能的文档分配权重，有效地缩小可能性空间，聚焦于最相关的内容。

这让我们回到 “缩小路径 “的核心主题。提示的主要目标是以一种方式来调节概率模型，使其预测的质量集中在我们想要引出的特定信息或行为上。通过提供精心制作的提示，我们可以引导模型更有效地导航潜在空间，生成更相关和连贯的输出。

然而，重要的是要记住，语言模型最终受限于它所训练的信息。虽然它可以生成类似于现有文档的文本或以新颖的方式组合想法，但它不能凭空创造全新的信息。例如，如果在其训练数据中没有发现和记录治愈癌症的方法，我们就不能期望模型提供这样的治疗方案。

相反，模型的优势在于它能够找到并综合与我们提供的提示词相似的信息。通过理解这些模型的概率性特征以及如何使用提示词来调节其输出，我们可以更有效地利用它们的能力来产生有价值的见解和内容。

考虑以下提示词。在第一个例子中，单独的“Mercury“可能指代行星、化学元素或罗马神祇，但最可能的是指行星。事实上，GPT-4 提供了一个以 * 水星是太阳系中最小且最内侧的行星...* 开头的长篇回答。第二个提示词明确指的是化学元素。第三个则指的是罗马神话中以速度见长并担任神使的人物。

```
# Prompt 1
Tell me about: Mercury

# Prompt 2
Tell me about: Mercury element

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

通过添加几个额外的词，我们就完全改变了 AI 的反应方式。正如您将在本书后面了解到的，花哨的提示工程技巧，如少样本提示、结构化输入/输出和思维链，其实都只是调节模型输出的巧妙方法。

因此，从根本上说，提示工程的艺术就是要理解如何在语言模型知识的广阔概率空间中导航，以缩小通向我们所寻求的特定信息或行为的路径。

对于那些具有扎实高等数学基础的读者来说，从概率论和线性代数原理出发来理解这些模型确实会有帮助！而对于其他想要开发有效策略来获得期望输出的读者，让我们继续使用更直观的方法。

如何 “缩小 “路径

为了应对知识过多的这些挑战，我们采用各种技术来引导语言模型的生成过程，并让它将注意力集中在最相关和准确的信息上。

以下是最重要的技术，按推荐顺序排列，也就是说，您应该首先尝试提示工程，然后是 RAG，最后才是微调（如果必须的话）。

提示工程 最基本的方法是制作包含特定指令、约束或示例的提示，以引导模型的响应生成。本章在下一节中介绍提示工程的基础知识，而在本书第 2 部分中我们会介绍许多具体的提示工程模式。这些模式包括提示精炼，这是一种专注于优化和改进提示的技术，用于提取 AI 认为最相关和简洁的信息。

上下文增强。在提示模型时，动态从外部知识库或文档中检索相关信息，为模型提供重点上下文。流行的上下文增强技术包括检索增强生成（RAG）。像Perplexity提供的所谓 “在线模型 “能够通过实时互联网搜索结果来增强其上下文。

尽管 LLM 非常强大，但它们并没有在您的独特数据集上进行训练，这些数据集可能是私有的或针对您要解决的特定问题。上下文增强技术让 LLM 能够访问 API 背后的数据、SQL 数据库中的数据，或者困在 PDF 和幻灯片中的数据。

微调或领域适应在特定领域的数据集上训练模型，以使其知识和生成能力专门用于特定任务或领域。

降低温度参数

温度是 transformer 基础语言模型中使用的一个超参数，用于控制生成文本的随机性和创造性。它是一个介于 0 和 1 之间的值，较低的值使输出更加集中和确定性，而较高的值则使输出更加多样化和不可预测。

当温度设置为 1 时，语言模型根据下一个词元的完整概率分布生成文本，允许更具创造性和多样化的响应。然而，这也可能导致模型生成的文本相关性或连贯性较差。

另一方面，当温度设置为 0 时，语言模型总是选择概率最高的词元，有效地 “缩小其路径 “。我的几乎所有 AI 组件都使用设置为 0 或接近 0 的温度，因为这会产生更加集中和可预测的响应。当您希望模型遵循指令、注意它被提供的函数，或者只是需要比现有的更准确和相关的响应时，这是非常有用的。

例如，如果您正在构建需要提供事实信息的聊天机器人，您可能想要将温度设置为较低的值，以确保响应更加准确和切题。相反，如果您正在构建创意写作助手，您可能想要将温度设置为较高的值，以鼓励更多样化和富有想象力的输出。

超参数：推理的旋钮和调节器

在使用语言模型时，您会经常遇到 “超参数 “这个术语。在推理的背景下（即，当您使用模型生成响应时），超参数就像是您可以调节的旋钮和调节器，用于控制模型的行为和输出。

可以把它想象成调节复杂机器的设置。就像您可能转动旋钮来控制温度或拨动开关来改变操作模式一样，超参数允许您精细调节语言模型处理和生成文本的方式。

在推理过程中，你会遇到以下常见的超参数：

- **温度**：如前所述，这个参数控制生成文本的随机性和创造性。较高的温度会产生更多样化和不可预测的输出，而较低的温度则会产生更集中和确定性的响应。

- **Top-p（核采样）：** 此参数控制累积概率超过特定阈值（p）的最小标记集的选择。 它允许产生更多样化的输出，同时仍保持连贯性。

- **Top-k 采样：** 这种技术选择概率最高的 k 个下一个标记，并在它们之间重新分配概率质量。 它可以帮助防止模型生成低概率或不相关的标记。

- **频率和存在惩罚**：这些参数会对模型过于频繁地重复相同词语（频率惩罚）或生成输入提示中不存在的词语（存在惩罚）进行惩罚。 通过调整这些值，你可以鼓励模型产生更加多样化和相关的输出。

- **最大长度**：这个超参数设置模型在单个响应中可以生成的标记（词或子词）数量的上限。它有助于控制生成文本的详细程度和简洁性。

当你尝试不同的超参数设置时，你会发现即使是微小的调整也会对模型的输出产生重大影响。这就像调整食谱一样——多加一撮盐或稍微延长烹饪时间都可能会让最终的菜品大不相同。

关键是要理解每个超参数如何影响模型的行为，并为你的特定任务找到合适的平衡点。不要害怕尝试不同的设置，看看它们如何影响生成的文本。随着时间的推移，你会逐渐培养出对哪些超参数需要调整以及如何达到预期结果的直觉。

通过将这些参数的使用与提示工程、检索增强生成和微调相结合，你可以有效地缩小路径，引导语言模型为特定用例生成更准确、相关和有价值的响应。

原始模型与指令调优模型的对比

原始模型是 LLM 的未经提炼、未经训练的版本。可以把它们想象成一张白纸，尚未受到特定训练来理解或遵循指令的影响。它们建立在最初训练时所用的海量数据之上，能够生成各种各样的输出。然而，如果没有额外的基于指令的微调层，它们的响应可能会不可预测，需要更加细致、精心设计的提示来引导它们产生所需的输出。使用原始模型就像在与一个博学多识但完全不懂得你在问什么的白痴天才沟通，除非你在指令中极其精确，否则他们无法理解你的意图。他们常常给人一种鹦鹉学舌的感觉，因为在他们能说出任何有意义的话时，往往只是在重复他们从你那里听到的内容。

另一方面，指令调优模型经过了专门设计的训练轮次，以理解和遵循指令。GPT-4、Claude 3和许多其他流行的 LLM 模型都经过了大量的指令调优。这种训练包括向模型输入指令示例及其期望的结果，有效地教会模型如何解释和执行

各种命令。因此，指令模型能够更好地理解提示背后的意图，并生成与用户期望紧密相符的响应。这使得它们更加用户友好，更容易使用，特别是对那些可能没有时间或专业知识进行大量提示工程的用户来说。

原始模型：未经过滤的画布

如果你一直在使用像 GPT-4 这样的流行 LLM 进行实验，那么原始模型（如 Llama 2-70B或 Yi-34B）提供了更加不受限制的模型功能访问方式。这些模型没有预先调优来遵循特定指令，为你提供了一个可以通过精心的提示工程直接操控模型输出的白纸。这种方法需要深入理解如何制作提示，在不明确指导的情况下引导 AI 朝着期望的方向发展。这就像是直接访问 AI 的 “原始 “层，没有任何中间层来解释或引导模型的响应（因此得名）。

原始模型的挑战在于它们容易陷入重复模式或产生随机输出。然而，通过细致的提示词工程和调整诸如重复惩罚等参数，可以引导原始模型生成独特和创造性的内容。这个过程并非没有权衡取舍；虽然原始模型在创新方面提供了无与伦比的灵活性，但它们也需要更高水平的专业知识。

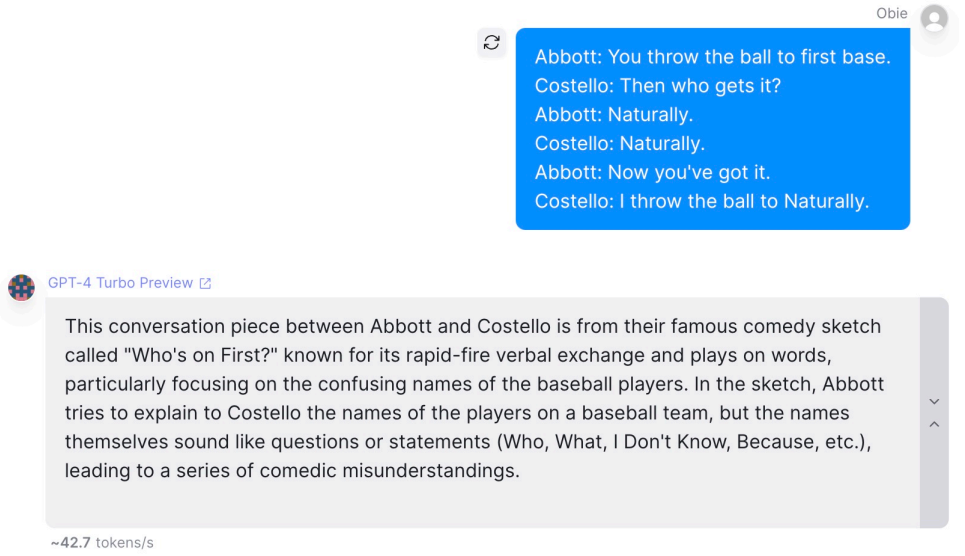


图 3. 作为对比，这是将相同的模糊提示输入给 GPT-4 的结果

指令微调模型：有引导的体验

指令微调模型被设计为能够理解和遵循特定指令，使其对更广泛的应用来说更加用户友好和易于使用。它们理解对话的机制，并且知道在轮到他们说话结束时应该停止生成。对于许多开发者，特别是那些开发直接明了的应用的人来说，指令微调模型提供了一个便捷且高效的解决方案。

指令微调过程涉及在大量人工生成的指令提示和回应上训练模型。一个值得注意的例子是开源的databricks-dolly-15k 数据集，其中包含了由 Databricks 员工创建的超过 15,000 对提示/回应对，你可以自己查看。该数据集涵盖了八个不同的指令类别，包括创意写作、封闭式和开放式问答、总结、信息提取、分类和头脑风暴。

在数据生成过程中，贡献者们收到了关于如何为每个类别创建提示和回应的指导。例如，对于创意写作任务，他们被要求提供具体的约束、指令或要求来引

导模型的输出。对于封闭式问答，他们被要求根据给定的维基百科段落编写需要事实正确回答的问题。

由此产生的数据集成为微调大型语言模型的宝贵资源，使其展现出类似ChatGPT这样的系统的交互和遵循指令的能力。通过在各种人工生成的指令和回应上进行训练，模型学会了理解和遵循具体指令，使其更擅长处理各种任务。

除了直接微调之外,像 databricks-dolly-15k 这样的数据集中的指令提示还可以用于合成数据生成。通过将贡献者生成的提示作为少样本示例提交给开放的大型语言模型，开发者可以在每个类别中生成更大规模的指令语料库。这种在 Self-Instruct 论文中概述的方法允许创建更加健壮的指令遵循模型。

此外，这些数据集中的指令和响应可以通过释义等技术进行增强。 通过重述每个提示或简短响应，并将生成的文本与相应的真实样本关联起来，开发人员可以引入一种正则化形式，从而增强模型遵循指令的能力。

指令微调模型提供的易用性是以某些灵活性为代价的。这些模型通常经过严格的审查，这意味着它们可能无法始终提供某些任务所需的创造自由度。它们的输出很大程度上受到微调数据中固有偏见和局限性的影响。

尽管存在这些限制，但由于其用户友好的特性以及能够以最少的提示词工程处理广泛的任务，指令微调模型变得越来越受欢迎。随着更多高质量指令数据集的出现，我们可以期待这些模型的性能和多功能性进一步提升。

为您的项目选择合适的模型类型

在基础（原始）模型和指令微调模型之间的选择最终取决于您项目的具体需求。对于需要高度创造力和原创性的任务，基础模型为创新提供了强大的工具。这些模型允许开发人员探索大语言模型的全部潜力，突破人工智能驱动应用的边界，但它们需要更多的实践方法和实验意愿。温度参数和其他设置在基础模型中的影响比在指令模型中要大得多。

您在提示词中包含的任何内容都是基础模型会尝试重复的内容。例如，如果您的提示词是聊天记录，原始模型将尝试继续聊天。根据最大令牌数限制，它不仅会生成聊天中的下一条消息，甚至可能会与自己进行整段对话！

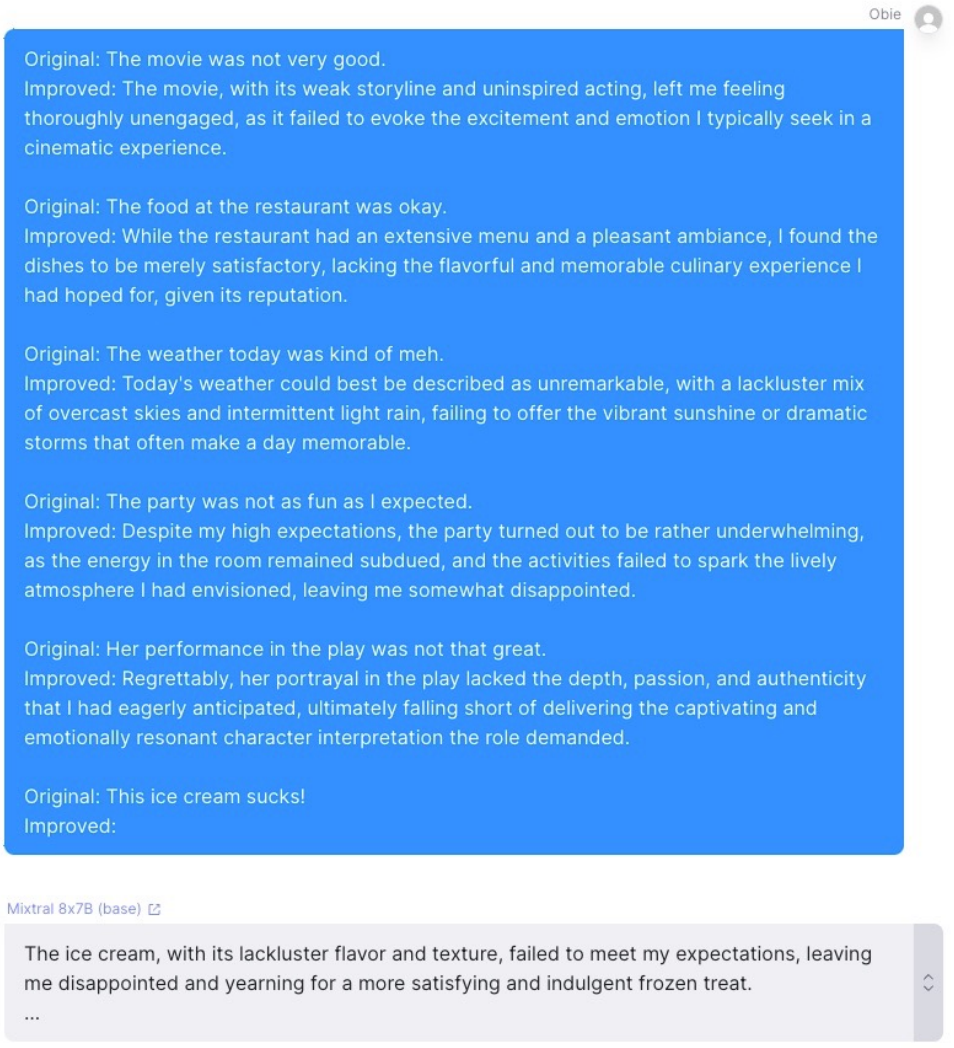


图 4. Mixtral 8x7B（基础版）句子改写的少样本补全示例

在准备上述由 Reddit 用户phree_radical提供的句子改写示例时，我只有在经过多次参数设置实验后才得到可用的结果，最终确定的参数为：温度 0.08，Top P：0.2，Top K：1，重复惩罚：1.26。

在生产环境中尝试使用这种基础模型方法会很棘手，这是因为`max_tokens` 参数

的强大影响。设置得太短会导致输出被截断。如果设置的长度超过模型所需的输出长度，它就会继续产生幻觉性的额外示例。

最终结论是，除非您真的需要完全的控制权和无审查限制，否则指令微调模型可以显著简化您的开发过程。为了更好地说明这一点，这里是 Mixtral 8x7B对相同提示词的响应，但这次是使用其指令微调版本：

> 很抱歉地告诉您，这个冰淇淋并不符合我的期望，因为它缺少我通常在高品质甜点中期待的那种浓郁、绵密的口感和令人愉悦的味道。我本希望能有一个更令人满意和愉快的体验。

值得注意的是，我能够将最大词元（tokens）设置保持在 500，模型可靠地在所需输出结束时停止，没有产生额外的虚构示例。

提示工程

当你开始在项目中应用人工智能时，你会很快发现最需要掌握的关键技能之一就是提示工程的艺术。但是什么是提示工程，为什么它如此重要？

从本质上来说，提示工程是设计和制作输入提示的过程，这些提示用于引导语言模型产生输出。它关乎如何与 AI 进行有效沟通，运用指令、示例和上下文的组合来引导模型生成所需的响应。

可以把它想象成与一个高度智能但较为死板的朋友交谈。要充分利用这种互动，你需要清晰、具体，并提供足够的上下文，确保你的朋友准确理解你的要求。这就是提示工程的作用所在，即使一开始看起来很简单，相信我，要掌握它需要大量的练习。

有效提示的基本要素

要开始设计有效的提示，首先你需要理解构成精心制作的输入的关键组成部分。以下是一些基本要素：

1. **指令**：告诉模型你想要它做什么的清晰简洁的指示。这可以是任何内容，从 “总结以下文章 “到 “生成一首关于日落的诗 “，再到 “将这个项目变更请求转换为 JSON 对象 “。
2. **上下文**：帮助模型理解任务背景和范围的相关信息。这可能包括目标受众的详细信息、期望的语气和风格，或者输出的具体限制或要求，比如需要遵循的 JSON 模式。
3. **示例**：展示你期望的输出类型的具体例子。通过提供一些精心选择的示例，你可以帮助模型学习所需响应的模式和特征。
4. **输入格式**：换行和 markdown 格式为我们的提示提供结构。将提示分成段落让我们可以对相关指令进行分组，这样人类和 AI 都更容易理解。项目符号和编号列表让我们可以定义条目和项目顺序。粗体和斜体标记让我们可以突出强调的内容。
5. **输出格式**：关于输出应如何构建和格式化的具体指令。这可能包括对期望长度的要求、标题或项目符号的使用、markdown 格式，或任何其他应遵循的具体输出模板或约定。

通过以不同方式组合这些基本要素，你可以创建针对特定需求的提示，引导模型生成高质量、相关的响应。

提示设计的艺术与科学

制作有效的提示既是一门艺术，也是一门科学。（这就是为什么我们称之为工艺。）它需要深入理解语言模型的能力和局限性，同时需要采用创造性的方法来设计能够引发所需行为的提示。对我来说，至少创造性的部分让它变得很有趣。它也可能让人非常沮丧，特别是当你在寻求确定性行为时。

提示工程的一个关键方面是理解如何平衡具体性和灵活性。一方面，你想提供足够的指导来引导模型朝正确的方向发展。另一方面，你不想过于规范而限制模型运用自身创造力和灵活性来处理边缘情况的能力。

另一个重要的考虑因素是示例的使用。精心选择的示例在帮助模型理解你想要的输出类型方面可以非常有效。然而，重要的是要谨慎使用示例，确保它们能

代表所需的响应。一个糟糕的示例在最好的情况下只是浪费词元（tokens），在最坏的情况下会破坏期望的输出。

提示工程技巧和最佳实践

当你深入研究提示工程的世界时，你会发现一系列可以帮助你创建更有效提示的技巧和最佳实践。以下是几个需要探索的关键领域：

1. **零样本与少样本学习**：理解何时使用零样本学习（不提供示例）与单样本或少样本学习（提供少量示例）可以帮助你创建更高效和有效的提示。
2. **迭代优化**：基于模型输出进行提示词迭代优化的过程可以帮助你找到最佳的提示词设计。反馈循环是一种强大的方法，它利用语言模型自身的输出来逐步提高生成内容的质量和相关性。
3. **提示链接**：将多个提示按序列组合可以帮助你将复杂任务分解成更小、更易管理的步骤。提示链接涉及将复杂任务或对话分解成一系列更小的、相互关联的提示。通过将提示链接在一起，你可以引导 AI 完成多步骤过程，在整个交互过程中保持上下文和连贯性。
4. **提示调优**：为特定领域或任务定制提示可以帮助你创建更专业和有效的提示。提示模板帮助你创建灵活、可重用且易于维护的提示结构，使其更容易适应手头的任务。

学会在什么时候使用零样本、单样本或少样本学习是掌握提示工程的特别重要的部分。每种方法都有其优势和劣势，理解何时使用每种方法可以帮助你创建更有效和高效的提示。

零样本学习：无需示例的情况

零样本学习指的是语言模型无需任何示例或明确训练就能执行任务的能力。换句话说，你向模型提供描述任务的提示，模型仅基于其预先存在的知识和对语言的理解来生成响应。

零样本学习在以下情况特别有用：

1. 任务相对简单直接，模型可能在预训练过程中遇到过类似的任务。
2. 你想测试模型的固有能力，看看它在没有任何额外指导的情况下如何响应新任务。
3. 你正在使用经过广泛任务和领域训练的大型多样化语言模型。

然而，零样本学习也可能不可预测，并且可能不总是产生预期的结果。模型的响应可能受到预训练数据中的偏见或不一致性的影响，并且在处理更复杂或微妙的任务时可能会遇到困难。

> 我见过零样本提示在 80% 的测试用例中运行良好，但在其余 20% 的情况下产生完全错误或难以理解的结果。建立全面的测试机制非常重要，尤其是当你大量依赖零样本提示时。

单样本学习：当一个示例可以产生影响

单样本学习涉及向模型提供一个所需输出的示例以及任务描述。这个示例作为模型可以用来生成自己响应的模板或模式。

单样本学习在以下情况下可能有效：

1. 任务相对新颖或特定，模型在预训练过程中可能没有遇到过很多类似的示例。
2. 你想要清晰简洁地展示所需的输出格式或风格。
3. 任务需要特定的结构或约定，这些可能仅从任务描述中并不明显。

对你来说显而易见的描述对 AI 来说不一定显而易见。单样本示例可以帮助理清事物。

单样本学习可以帮助模型更清楚地理解期望，并生成与提供的示例更加一致的响应。然而，重要的是要仔细选择示例，确保它能代表所需的输出。在选择示例时，要考虑潜在的边缘情况和提示将要处理的输入范围。

图 5. 期望 JSON 的单样本示例

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

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

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

This is an example of well-formed output:

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

少样本学习：多个示例如何提升性能

少样本学习是指在任务描述的同时向模型提供少量示例（通常在 2 到 10 个之间）。这些示例能为模型提供更多上下文和变化，帮助它生成更加多样化和准确的响应。

少样本学习在以下情况下特别有用：

1. 当任务复杂或细微，单个示例可能无法涵盖所有相关方面时。
2. 当你想要提供一系列展示不同变化或边缘情况的示例时。
3. 当任务需要模型生成符合特定领域或风格的响应时。

通过提供多个示例，你可以帮助模型对任务建立更稳健的理解，并生成更加一致和可靠的响应。

示例：提示词可以比你想象的复杂得多

今天的大语言模型的推理能力比你想象的要强大得多。所以不要把自己局限于仅仅将提示词视为输入和输出对的规范。你可以尝试给出长篇且复杂的指令，就像你在与人类互动时那样。

例如，这是我在 Olympia中使用的一个提示词，当时我正在开发与 Google 服务的集成原型，这可能是世界上最大的 API 之一。我早期的实验证明 GPT-4对 Google API 有相当不错的了解，而我既没有时间也没有动力去编写细粒度的映射层，逐个实现我想要提供给 AI 的每个功能。如果我可以直接让 AI 访问所有的 Google API会怎样呢？

我的提示词开始时告诉 AI 它可以通过 HTTP 直接访问 Google API 端点，它的角色是代表用户使用 Google 应用程序和服务。然后我提供了指导原则，特别是与`fields` 参数相关的规则（因为它似乎在这个参数上遇到最多问题），以及一些特定于 API 的提示（这正是少样本提示的实践）。

以下是完整的提示词，它告诉 AI 如何使用提供的`invoke_google_api` 函数。

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

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

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

All of these rules use examples related to the files.get method.
- Use a comma-separated list to select multiple fields,
  such as 'name, mimeType'.
- Use a/b to select field b that's nested within field a,
  such as 'capabilities/canDownload'.
- Use a sub-selector to request a set of specific sub-fields of arrays or
  objects by placing expressions in parentheses "()". For example,
  'permissions(id)' returns only the permission ID for each element in the
  permissions array.
```

```
- To return all fields in an object, use an asterisk as a wild card in field
  selections. For example, 'permissions/permissionDetails/*' selects all
  available permission details fields per permission. Note that the use of
  this wildcard can lead to negative performance impacts on the request.

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

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

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

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

```
59
60          handle_response(response)
61        end
62
63        # ...rest of class
64      end
```

你可能想知道这个提示是否有效。简单的答案是肯定的。AI并不总是能在第一次尝试时就完美地调用 API。但是，如果它犯了错误，我只需将产生的错误消息作为调用结果反馈给它。在了解到自己的错误后，AI 就能对其进行分析并重新尝试。大多数情况下，它在尝试几次后就能得到正确结果。

请注意，在使用这个提示时，Google API返回的大型 JSON 结构效率极其低下，所以我并不建议在生产环境中使用这种方法。然而，我认为这种方法能够奏效本身就证明了提示工程的强大力量。

实验和迭代

最终，如何设计你的提示取决于具体任务、期望输出的复杂度，以及你所使用的语言模型的能力。

作为一名提示工程师，进行不同方法的实验并根据结果进行迭代非常重要。从零样本学习开始，看看模型的表现如何。如果输出不稳定或不理想，可以尝试提供一个或多个示例，看看性能是否有所改善。

要记住，即使在每种方法中，也有变化和优化的空间。你可以尝试不同的示例，调整任务描述的措辞，或提供额外的上下文来帮助引导模型的响应。

随着时间推移，你会逐渐形成直觉，知道哪种方法最适合特定任务，并且能够创建更有效和高效的提示。关键是要保持好奇心，在提示工程方面保持实验性和迭代性的态度。

在本书中，我们将深入探讨这些技术，并探索如何将它们应用到实际场景中。通

过掌握提示工程的艺术和科学，你将能够充分发挥 AI 驱动的应用程序开发的潜力。

模糊性的艺术

在为大语言模型（LLMs）制作有效提示时，人们通常认为更具体和详细的指令会带来更好的结果。然而，实践经验表明情况并非总是如此。事实上，在提示中有意保持模糊往往能产生更好的效果，这充分利用了大语言模型出色的泛化和推理能力。

Ken，一位已处理超过 5 亿 GPT 词元的创业公司创始人，分享了他的宝贵经验。他学到的关键经验之一是在提示方面 “少即是多 “。Ken 发现，相比提供精确的列表或过于详细的指令，让大语言模型依靠其基础知识往往能产生更好的结果。

这个认识颠覆了传统的编程思维模式，即所有内容都需要详细明确地说明。对于大语言模型来说，重要的是要认识到它们拥有大量知识，能够进行智能连接和推理。通过在提示中保持适度的模糊性，你让大语言模型能够自由运用其理解力，提出你可能没有明确指定的解决方案。

例如，当 Ken 的团队在开发一个用于将文本分类为 50 个美国州之一或联邦政府相关的流水线时，他们最初的方法是提供一个包含完整州名列表及其对应 ID 的 JSON 格式数组。

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

这种方法失败得足够多，以至于他们不得不深入研究提示词来找出如何改进。在这个过程中，他们注意到，即使大语言模型经常会把 ID 弄错，但它在`name` 字段中始终能返回正确州名的全称，尽管他们并没有明确要求这样做。

通过删除地区 ID 并将提示词简化为类似 “GPT，你显然知道 50 个州，所以只需给我这个相关州的全称，如果是涉及美国联邦政府就返回 Federal“这样的形式，他们取得了更好的结果。这个经验突显了利用大语言模型泛化能力的力量，以及让它基于现有知识进行推理的重要性。

> Ken 对这种分类方法（而非更传统的编程技术）的解释，很好地阐明了我们这些拥抱大语言模型技术潜力的人的思维方式：“这并不是一个困难的任务 - 我们可能本可以使用字符串/正则表达式，但有太多奇怪的边界情况，那样可能会花更长时间。”

当给出更模糊的提示词时，大语言模型在质量和泛化方面的提升能力，是高阶思维和任务委派的一个显著特征。这表明大语言模型能够处理模糊性，并根据提供的上下文做出智能决策。

然而，需要注意的是，模糊并不意味着不清晰或含糊不清。关键是提供足够的上下文和指导来引导大语言模型朝正确的方向发展，同时让它能够灵活运用其知识和泛化能力。

因此，在设计提示词时，请考虑以下 “少即是多 “的建议：

1. 关注期望的结果，而不是指定过程的每个细节。
2. 提供相关的上下文和约束，但避免过度具体化。
3. 通过引用常见概念或实体来利用现有知识。
4. 为基于给定上下文的推理和联系留出空间。
5. 根据大语言模型的响应来迭代和优化你的提示词，找到具体性和模糊性之间的适当平衡。

通过掌握提示工程中的模糊艺术，你可以释放大语言模型的全部潜力并取得更好的结果。相信大语言模型的泛化能力和智能决策能力，你可能会对收到的输出的质量和创造力感到惊讶。注意观察不同模型对提示词中不同具体程度的响

应，并相应地进行调整。通过实践和经验，你将培养出敏锐的判断力，知道何时该更模糊，何时该提供额外指导，从而能够在应用中有效地驾驭大语言模型的力量。

为什么拟人化主导了提示工程

拟人化，即将人类特征赋予非人类实体，在大语言模型的提示工程中占主导地位是有意为之的。这是一个设计选择，让包括我们应用程序开发者在内的广大用户能够更直观、更容易地与强大的 AI 系统进行交互。

对大语言模型进行拟人化为完全不熟悉系统底层技术复杂性的人提供了一个直观的框架。如果你尝试使用未经指令微调的模型来完成任何有用的任务，你就会发现，构建一个能够产生有价值的预期输出的框架是一项具有挑战性的任务。这需要对系统的内部运作有相当深入的理解，而这样的专家相对较少。

通过将与语言模型的交互视为两个人之间的对话，我们可以依靠我们对人类交流的天然理解来传达我们的需求和期望。就像早期 Macintosh 用户界面设计优先考虑直观性而非复杂性一样，AI 的拟人化框架让我们能以自然和熟悉的方式进行交互。

当我们与他人交流时，我们本能地会直接用 “你 “来称呼对方，并清楚地指出我们期望他们如何行事。这种方式完美地转化到提示工程过程中，我们通过指定系统提示词并进行来回对话来引导 AI 的行为。

通过这种框架，我们可以轻松理解向 AI 提供指令并接收相关响应的概念。拟人化方法减少了认知负担，让我们能够专注于手头的任务，而不是纠结于系统的技术细节。

需要注意的是，虽然拟人化是让 AI 系统更容易使用的强大工具，但它也带来了某些风险和限制。我们的用户可能会产生不切实际的期望或对我们的系统形成不健康的情感依赖。作为提示工程师和开发人员，在利用拟人化优势的同时，确保用户清楚地理解 AI 的能力和局限性至关重要。

随着提示工程领域的不断发展，我们可以预期在与大语言模型交互方式上会出现更多的改进和创新。然而，将拟人化作为提供直观且易用的开发者和用户体验的手段，可能仍将是这些系统设计中的一个基本原则。

指令与数据的分离：一个关键原则

理解一个支撑这些系统安全性和可靠性的基本原则至关重要：指令与数据的分离。

在传统的计算机科学中，被动数据和主动指令之间的明确区分是一个核心安全原则。这种分离有助于防止意外或恶意代码的执行，从而可能损害系统的完整性和稳定性。然而，今天的大语言模型主要是作为类似聊天机器人的指令执行模型开发的，往往缺乏这种正式且原则性的分离。

就大语言模型而言，指令可以出现在输入的任何位置，无论是系统提示还是用户提供的提示。这种分离的缺失可能导致潜在的漏洞和不良行为，类似于数据库面临的 SQL 注入或操作系统缺乏适当内存保护的问题。

在使用大语言模型时，意识到这个局限性并采取措施来降低风险是至关重要的。一种方法是仔细构建你的提示和输入，以清晰地区分指令和数据。提供明确指导，说明什么是指令、什么应该被视为被动数据的典型方法包括使用标记式标签。你的提示可以帮助大语言模型更好地理解和遵守这种分离。

图 6. 使用 XML 来区分指令、源材料和用户提示

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

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

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

另一种技术是对提供给大型语言模型的输入实施额外的验证和净化层。通过过滤或转义可能嵌入在数据中的任何潜在指令或代码片段，可以降低意外执行的可能性。像提示链接这样的模式对此很有用。

此外，在设计应用程序架构时，考虑在更高层面上实施机制来强制指令和数据的分离。这可能包括使用单独的端点或 API 接口来处理指令和数据，实施严格的输入验证和解析，并应用最小权限原则来限制大型语言模型可以访问和执行的范围。

最小权限原则

采用最小权限原则就像举办一场高度专属的派对，客人只能进入他们绝对需要去的房间。想象你在一座大型豪宅里举办这场派对。并非每个人都需要进入酒窖或主卧室，对吧？通过应用这个原则，你实际上是在分发只能打开特定门的钥匙，确保每位客人（在我们的例子中是大型语言模型应用程序的每个组件）只能获得履行其职责所必需的访问权限。

这不仅仅是对钥匙的吝啬，而是认识到在威胁可能来自任何地方的世界中，明智的做法是限制活动范围。如果有人未经邀请闯入你的派对，他们会发现自己被限制在门厅里，可以说，这大大限制了他们可能造成的破坏。因此，在保护你的大型语言模型应用程序时，请记住：只提供必要房间的钥匙，保持豪宅其他部分的安全。这不仅是良好的礼仪，更是良好的安全实践。

虽然当前的大型语言模型可能没有正式的指令和数据分离，但作为开发人员，你必须意识到这个限制并采取积极措施来降低风险。通过应用传统计算机科学的最佳实践，并将其适应于大型语言模型的独特特征，你可以构建更安全、更可靠的应用程序，在保持系统完整性的同时利用这些模型的强大功能。

提示精炼

制作完美的提示通常是一项具有挑战性且耗时的任务，需要深入理解目标领域和语言模型的细微差别。这就是 “提示精炼 “技术发挥作用的地方，它提供了一种强大的提示工程方法，利用大型语言模型的能力来简化和优化这个过程。

提示精炼是一种多阶段技术，涉及使用大型语言模型来协助创建、改进和优化提示。这种方法不仅仅依赖于人类的专业知识和直觉，而是利用大型语言模型的知识和生成能力来协作制作高质量的提示。

通过参与生成、改进和整合的迭代过程，提示精炼使你能够创建

更连贯、更全面且与所需任务或输出更加一致的提示。请注意，精炼过程可以在 OpenAI或 Anthropic等大型 AI 供应商提供的众多 “试验场 “中手动完成，也可以根据使用场景作为应用程序代码的一部分自动完成。

工作原理

提示精炼通常包括以下步骤：

1. **识别核心意图**：分析提示以确定其主要目的和期望结果。去除任何无关的信息，专注于提示的核心意图。
2. **消除歧义**：检查提示中任何模糊或不明确的语言。澄清含义并提供具体细节，以引导 AI 生成准确和相关的响应。
3. **简化语言**：使用清晰简洁的语言简化提示。避免复杂的句子结构、专业术语或可能使 AI 混淆或引入噪音的不必要细节。
4. **提供相关上下文**：仅包含 AI有效理解和处理提示所需的最相关上下文信息。避免包含可能分散核心意图的无关或冗余细节。
5. **迭代和改进**：根据 AI 的响应和反馈持续迭代和改进提示。评估生成的输出并进行必要的调整，以提高提示的清晰度和有效性。可以选择使用提示对象在数据库中对提示进行版本控制，以跟踪迭代并使自己能够在运行时轻松回滚更改。

初始提示词生成

提示词提炼的第一步是向大语言模型提供所需任务或输出的高层描述或意图。这可以是一个简短的陈述或一组关键词，用来捕捉提示词应该实现的核心内容。然后，大语言模型基于这个输入生成初始提示词，融入它认为合适的相关指令、示例或上下文。

让我们尝试给 Claude 3 Opus 以下初始提示词：

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

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

Claude 响应并输出以下内容：

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

When interacting with clone owners, you should:

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

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

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

4. Ensure that all actions and recommendations comply with Olympia's
```

```
policies, best practices, and ethical guidelines for AI use.

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

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

提示词优化

一旦生成初始提示词，请对其进行审查，然后让大语言模型考虑任何额外的上下文、约束条件或反馈。让大语言模型根据提供的输入和其对任务的理解，生成优化版本的提示词。这种优化过程可以进行反复迭代，每次迭代都基于用户和大语言模型之间的协作来进一步完善提示词。

这是我的优化请求，继续之前的提炼过程。

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

以下是克劳德的回答：

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

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

When interacting with clone owners, you should:

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

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

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

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

This updated directive focuses on the core responsibilities and interaction

```
guidelines for the LLM as an intelligent account manager, while emphasizing
that system interaction tasks are handled through provided functions rather
than simple user responses.
```

提示压缩

你可能已经注意到，除非你特别要求它们保持简洁，大语言模型往往会比较啰嗦。我们在前几节中一直在使用的提示词相当长，所以让我们让 Claude来帮我们精简一下：

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

Claude回应道，使用了以下这个大幅缩短的提示语：

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

系统指令和上下文集成

除了优化提示本身之外，LLM还可以生成适当的系统指令或上下文信息来引导最终输出。在为应用程序代码进行提示工程设计 AI 例程时，在精炼的这个阶段，你几乎肯定会专注于输出约束，但你也可能需要考虑所需的语气、风格、格式或其他任何影响生成响应的相关参数。

最终提示组装

提示精炼过程的最终成果是组装最终提示。这涉及将优化后的提示、生成的系统指令和集成的上下文组合成一个连贯而全面的代码，以便用于生成所需的输出。

在最终提示组装阶段，你可以再次尝试提示压缩，通过要求 LLM 将提示的措辞缩减到可能的最短词元序列，同时仍保持其行为的本质。这确实是一个可能成功也可能失败的尝试，但特别是在需要大规模运行提示的情况下，效率的提升可以为你节省相当多的词元消耗费用。

主要优势

通过利用 LLM 的知识和生成能力来优化你的提示，最终的提示更有可能结构良好、信息丰富，并且针对特定任务进行定制。迭代优化过程有助于确保提示具有高质量，并有效地捕捉所需的意图。其他优势包括：

效率和速度：提示精炼通过自动化提示创建和优化的某些方面，简化了提示工程过程。该技术的协作性质允许更快地收敛到有效的提示，减少了手动制作提示所需的时间和精力。

一致性和可扩展性：在提示工程过程中使用 LLM 有助于保持提示之间的一致性，因为 LLM 可以从之前成功的提示中学习和应用最佳实践和模式。这种一致性，加上大规模生成提示的能力，使提示精炼成为大规模 AI 驱动应用程序的宝贵技术。

项目想法：在库层面开发工具，简化在执行自动提示精炼的系统中进行提示版本控制和评分的过程。

要实现提示精炼，开发人员可以设计一个工作流或管道，在提示工程过程的各个阶段集成 LLM。这可以通过 API 调用、自定义工具或集成开发环境来实现，

这些环境能够促进用户和 LLM 在提示创建过程中的无缝交互。具体的实现细节可能因所选择的 LLM 平台和应用程序的要求而异。

那么微调呢?

在本书中，我们详细讨论了提示工程和 RAG，但没有涉及微调。做出这个决定的主要原因是，我认为大多数应用程序开发人员并不需要微调来满足他们的 AI 集成需求。

提示工程涉及精心制作包含零样本到少样本示例、约束和指令的提示，可以有效地引导模型生成相关且准确的响应，适用于广泛的任务。通过提供清晰的上下文并通过精心设计的提示缩小路径，你可以利用大语言模型的广泛知识，而无需进行微调。

同样，检索增强生成（RAG）为将 AI 集成到应用程序中提供了一种强大的方法。通过从外部知识库或文档中动态检索相关信息，RAG 在提示时为模型提供了重点突出的上下文。这使模型能够生成更准确、更新且更具领域特定性的响应，而无需进行耗时且资源密集的微调过程。

虽然微调对于高度专业化的领域或需要深度定制的任务可能有益，但它通常伴随着显著的计算成本、数据要求和维护开销。对于大多数应用程序开发场景，有效的提示工程和 RAG 的组合应该足以实现所需的 AI 驱动功能和用户体验。

检索增强生成（RAG）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

什么是检索增强生成？

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

RAG 是如何工作的？

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

为什么要在应用程序中使用 RAG？

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

在应用程序中实现 RAG

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

知识源的准备（分块）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

命题分块

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

实现说明

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

质量检查

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

基于命题的检索的优势

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

RAG 的真实应用案例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

案例研究：不使用嵌入的税务准备应用程序中的 RAG

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

智能查询优化（IQO）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

重排序

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

RAG 评估（RAGAs）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

忠实度

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

答案相关性

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

上下文精确度

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

上下文相关性

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

上下文召回率

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

上下文实体召回率

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

答案语义相似度（ANSS）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

答案正确性

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

方面评判

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

挑战与未来展望

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

语义分块：通过上下文感知分段增强检索

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

层次化索引：构建改进检索的数据结构

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

自反思 RAG：具有自我反思能力的增强

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

HyDE：假设性文档嵌入

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

什么是对比学习?

众多工作器

我喜欢将 AI 组件想象成小型的、近乎人类的虚拟“工作器“，它们可以无缝集成到我的应用程序逻辑中，执行特定任务或做出复杂决策。这种想法的目的是有意识地将 LLM的能力人性化，这样就不会有人过分兴奋，把它们不具备的神奇特质强加于它们身上。

开发者不必完全依赖复杂的算法或耗时的手动实现，而是可以将 AI 组件概念化为智能的、专注的、类人实体，在需要时可以调用它们来解决复杂问题，并基于它们的训练和知识提供解决方案。这些实体不会分心，也不会请病假。它们不会突然决定用不同于指示的方式来做事，而且总的来说，如果编程正确，它们也不会犯错。

从技术角度来说，这种方法的核心原则是将复杂任务或决策过程分解成更小、更易管理的单元，这些单元可以由专门的 AI 工作器处理。每个工作器都被设计为专注于问题的特定方面，发挥其独特的专业知识和能力。通过在多个 AI 工作器之间分配工作负载，应用程序可以实现更高的效率、可扩展性和适应性。

例如，考虑一个需要实时审核用户生成内容的 Web 应用程序。从零开始实现一个全面的审核系统将是一项艰巨的任务，需要大量的开发工作和持续维护。然而，通过采用众多工作器方法，开发者可以将 AI 驱动的审核工作器集成到应用程序逻辑中。这些工作器可以自动分析和标记不当内容，使开发者能够专注于应用程序的其他关键方面。

作为独立可重用组件的 AI 工作器

众多工作器方法的一个关键方面是其模块化特性。面向对象编程的支持者几十年来一直告诉我们要将对象交互视为消息。好吧，AI 工作器可以被设计成独立的、可重用的组件，它们可以通过普通语言消息 “相互交谈 “，就像它们真的是在互相交谈的小人一样。这种松耦合方法允许应用程序随着时间的推移而适应和发展，因为新的 AI 技术出现或业务逻辑需求发生变化。

在实践中，即使涉及 AI 工作器，设计清晰的接口和组件之间的通信协议的需求也没有改变。你仍然必须考虑性能、可扩展性和安全性等其他因素，但现在还有全新的 “软性要求 “需要考虑。例如，许多用户反对将他们的私人数据用于训练新的 AI 模型。你是否验证了你使用的模型提供商提供的隐私保护级别？

AI 工作器作为微服务？

当你阅读众多工作器方法时，你可能会注意到它与微服务架构有一些相似之处。两者都强调将复杂系统分解为更小、更易管理且可独立部署的单元。就像微服务被设计成松耦合的、专注于特定业务能力并通过定义良好的 API 进行通信一样，AI 工作器也被设计成模块化的、专注于特定任务并通过清晰的接口和通信协议相互交互。

然而，需要注意一些关键的区别。虽然微服务通常被实现为在不同机器或容器上运行的独立进程或服务，但 AI 工作器可以根据你的具体需求和可扩展性

> 需要，被实现为单个应用程序内的独立组件或独立服务。此外，AI 工作器之间的通信通常涉及交换丰富的、基于自然语言的信息，如提示、指令和生成的内容，而不是微服务中常用的更结构化的数据格式。
>
> 尽管存在这些差异，模块化、松耦合和清晰的通信接口的原则仍然是两种模式的核心。通过将这些原则应用到你的 AI 工作器架构中，你可以创建灵活、可扩展和可维护的系统，利用 AI 的力量解决复杂问题并为用户创造价值。

众多工作器方法可以应用于各种领域和应用程序，利用 AI 的力量来处理复杂任务并提供智能解决方案。让我们探讨一些在不同场景中使用 AI 工作器的具体例子。

账户管理

实际上每个独立的 Web 应用程序都有账户（或用户）的概念。在 Olympia 中，我们使用一个`AccountManager` AI 工作器，它被编程为能够处理各种与用户账户相关的变更请求。

它的指令是这样的：

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

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

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

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

由account.to_directive生成的账户初始状态只是该账户的文本描述，包括相关数据，如用户、订阅等。

AccountManager 可用的功能范围使其能够编辑用户的订阅、添加和删除 AI 顾问以及其他类型的付费附加服务，并向账户所有者发送通知邮件。除了finished功能外，如果在处理过程中遇到错误或需要任何其他形式的请求协助，它还可以通过notify_human_administrator 通知人工管理员。

请注意，当遇到问题时，AccountManager可以选择搜索 Olympia 的知识库，在那里可以找到关于如何处理边缘情况和任何其他使其不确定如何继续的情况的说明。

电子商务应用

在电子商务领域，AI 工作者可以在提升用户体验和优化业务运营方面发挥关键作用。以下是 AI 工作者可以发挥作用的几种方式：

产品推荐

AI 工作者在电子商务中最强大的应用之一是生成个性化产品推荐。通过分析用户行为、购买历史和偏好，这些工作者可以推荐适合每个用户个人兴趣和需求的产品。

有效产品推荐的关键在于结合使用协同过滤和基于内容的过滤技术。协同过滤通过观察相似用户的行为来识别模式，并基于具有相似品味的其他用户购买或喜欢的内容来做出推荐。另一方面，基于内容的过滤则关注产品本身的特征和属性，推荐与用户之前表现出兴趣的商品具有相似特征的商品。

以下是如何使用“Railway Oriented (ROP)”函数式编程风格在 Ruby 中实现产品推荐工作者的简化示例：

```
class ProductRecommendationWorker
  include Wisper::Publisher

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

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

```
18      end
19    end
20  end
```

Ruby函数式编程在这个示例中使用的风格受到 F#和 Rust的影响。你可以在我的朋友 Chad Wooley在 GitLab上的技术说明中了解更多相关信息。

在这个示例中，`ProductRecommendationWorker` 接收一个用户作为输入，通过将值对象传递给一系列函数式步骤来生成个性化产品推荐。让我们分解每个步骤：

1. `ValidateUser.validate`：这个步骤确保用户是有效的且有资格获得个性化推荐。它检查用户是否存在、是否处于活动状态，以及是否具有生成推荐所需的必要数据。如果验证失败，将返回错误结果，并使链条短路。
2. `AnalyzeCurrentSession.analyze`：如果用户有效，这个步骤会分析用户当前的浏览会话以收集上下文信息。它查看用户最近的交互，如已查看的产品、搜索查询和购物车内容，以了解他们当前的兴趣和意图。
3. `CollaborativeFilter.filter`：利用_相似用户的行为_，这个步骤应用协同过滤技术来识别用户可能感兴趣的产品。它考虑诸如购买历史、评分和用户-商品交互等因素，以生成一组候选推荐。
4. `ContentBasedFilter.filter`：这个步骤通过应用基于内容的过滤来进一步细化候选推荐。它将候选产品的属性和特征与_用户的偏好和历史数据_进行比较，以选择最相关的商品。
5. `ProductSelector.select`：最后，这个步骤根据预定义的标准（如相关性分数、流行度或其他业务规则）从过滤后的推荐中选择前 N 个产品。然后将选定的产品作为最终的个性化推荐返回。

在这里使用 Ruby函数式编程风格的优点在于它允许我们以清晰简洁的方式将这些步骤链接在一起。每个步骤都专注于特定任务，并返回一个`Result` 对象，可

以是成功（ok）或错误（err）。如果任何步骤遇到错误，链条就会短路，错误会传播到最终结果。

在最后的case 语句中，我们对最终结果进行模式匹配。如果结果是错误（ProductRecommendationError），我们使用 Honeybadger等工具记录错误以进行监控和调试。如果结果成功（ProductRecommendations），我们使用 Wisper发布/订阅库广播一个:new_recommendations 事件，同时传递用户和生成的推荐。

通过利用函数式编程技术，我们可以创建一个模块化且易于维护的产品推荐工作器。每个步骤都是自包含的，可以轻松测试、修改或替换，而不会影响整体流程。模式匹配和Result 类的使用帮助我们优雅地处理错误，并确保在任何步骤遇到问题时工作器能快速失败。

当然，这是一个简化的示例，在实际场景中，你需要与你的电子商务平台集成、处理边缘情况，甚至深入推荐算法的实现。然而，将问题分解为更小的步骤并利用函数式编程技术的核心原则保持不变。

欺诈检测

以下是一个简化示例，展示如何使用相同的铁路导向编程（ROP）风格在 Ruby 中实现欺诈检测工作器：

```
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
```

FraudDetection 类是一个值对象，它封装了给定交易的欺诈检测状态。它提供了一种结构化方式来分析和评估基于各种风险因素的交易欺诈风险。

```
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: }
```

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

  def high_risk?
    fraud_probability > RISK_THRESHOLD
  end

  private

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

FraudDetection 类具有以下属性：

- transaction：对正在进行欺诈分析的交易的引用。
- risk_factors：存储与交易相关的风险因素的数组。每个风险因素都表示为一个散列，其中键是风险因素的描述，值是与该风险因素相关的欺诈概率。

add_risk_factor 方法允许向risk_factors 数组添加风险因素。它接受两个参数：description（描述风险因素的字符串）和probability（表示与该风险因素相关的欺诈概率的浮点数）。我们使用case..in 条件语句进行简单的类型检查。

链末端将被检查的high_risk? 方法是一个谓词方法，它将fraud_probability（通过汇总所有风险因素的概率计算得出）与RISK_THRESHOLD 进行比较。

FraudDetection 类为交易的欺诈检测提供了一种清晰且封装良好的管理方式。它允许添加多个风险因素，每个因素都有其自己的描述和概率，并提供一个方法来根据计算出的欺诈概率确定交易是否被视为高风险。该类可以轻松集成到更大的欺诈检测系统中，不同组件可以协同工作来评估和降低欺诈交易的风险。

最后，由于这毕竟是一本关于使用 AI 编程的书籍，这里有一个使用我的Raix库的ChatCompletion 模块实现CheckCustomerHistory类的示例：

```
class CheckCustomerHistory
  include Raix::ChatCompletion

  attr_accessor :fraud_detection

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

    Consider factors such as the frequency of transactions, transaction
    amounts, geographical locations, and any deviations from the customer's
    typical behavior to generate a probability score as a float in the range
    of 0 to 1 (with 1 being absolute certainty of fraud).

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

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

```

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

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

  private

  def initialize(fraud_detection)
    self.fraud_detection = fraud_detection
  end

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

在这个例子中，CheckCustomerHistory 定义了一个INSTRUCTION 常量，该常量为人工智能模型提供了具体指令，指导其如何通过系统指令分析客户的交易历史中的潜在欺诈指标。

self.check 方法是一个类方法，它使用fraud_detection 对象初始化CheckCustomerHistory 的新实例，并调用call 方法来执行客户历史分析。

在call 方法内部，系统会检索客户的交易历史并将其格式化为一个文本记录，然后将其传递给人工智能模型。人工智能模型根据提供的指令分析交易历史，并返回其发现的摘要。

这些发现会被添加到fraud_detection 对象中，然后将更新后的fraud_detection 对象作为成功的Result 返回。

通过利用ChatCompletion 模块，CheckCustomerHistory类可以运用人工智能的能力来分析客户的交易历史并识别潜在的欺诈指标。这使得欺诈检测技术更加复杂和适应性强，因为人工智能模型能够随时间学习并适应新的模式和异常。

更新后的FraudDetectionWorker 和CheckCustomerHistory 类展示了如何无缝集成 AI 工作器，通过智能分析和决策能力增强欺诈检测流程。

客户情绪分析

这里还有一个类似的例子，展示如何实现客户情绪分析工作器。这次的解释会少一些，因为你应该已经掌握了这种编程风格的要领：

```ruby
class CustomerSentimentAnalysisWorker
  include Wisper::Publisher

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

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

在这个例子中，`CustomerSentimentAnalysisWorker` 的步骤包括预处理反馈（例如，去除噪音、分词）、执行情感分析以确定整体情感（积极、消极或中性）、提取关键短语和主题、识别趋势和模式，以及基于分析生成可行的见解。

医疗保健应用

在医疗保健领域，AI 工作器可以在各种任务中协助医疗专业人员和研究人员，从而改善患者预后并加速医学发现。以下是一些例子：

患者接诊

AI 工作器可以通过自动化各种任务并提供智能协助来简化患者接诊流程。

** 预约安排：**AI 工作器可以通过理解患者偏好、可用时间和医疗需求的紧急程度来处理预约安排。它们可以通过对话界面与患者互动，指导他们完成预约过程，并根据患者的要求和医疗服务提供者的可用性找到最合适的预约时段。

** 病史采集：** 在患者接诊过程中，AI 工作器可以协助收集和记录患者的病史。它们可以与患者进行互动对话，询问有关其既往病史、用药情况、过敏史和家族史的相关问题。AI 工作器可以使用自然语言处理技术来解释和构建所收集的信息，确保信息准确记录在患者的电子健康记录中。

** 症状评估和分层：**AI 工作器可以通过询问患者当前症状、持续时间、严重程度和任何相关因素来进行初步症状评估。通过利用医学知识库和机器学习模型，这些工作器可以分析所提供的信息，生成初步鉴别诊断或推荐适当的后续步骤，如安排与医疗服务提供者的咨询或建议自我护理措施。

** 保险验证：**AI 工作器可以在患者接诊过程中协助保险验证。它们可以收集患者保险详情，通过 API 或网络服务与保险提供商沟通，并验证保险资格和福利。这种自动化有助于简化保险验证流程，减少管理负担并确保信息准确采集。

** 患者教育和指导：**AI 工作器可以根据患者的具体医疗状况或即将进行的手术，为患者提供相关的教育材料和指导。它们可以提供个性化内容，回答常见问题，并就就诊前准备、用药说明或术后护理提供指导。这有助于确保患者在整个医疗过程中保持知情和参与。

通过在患者接诊中利用 AI 工作器，医疗机构可以提高效率，减少等待时间，改善整体患者体验。这些工作器可以处理常规任务，收集准确信息，并提供个性化协助，使医疗专业人员能够专注于为患者提供优质护理。

患者风险评估

AI 工作器可以通过分析各种数据源和应用高级分析技术在评估患者风险方面发挥关键作用。

** 数据整合：**AI 工作器可以从多个来源收集和理解患者数据，如电子健康记录（EHR）、医学影像、实验室结果、可穿戴设备和健康社会决定因素。通过将这些信息整合成全面的患者档案，AI 工作器可以提供患者健康状况和风险因素的整体视图。

** 风险分层：**AI 工作器可以使用预测模型根据患者的个人特征和健康数据将患者分为不同的风险类别。这种风险分层使医疗服务提供者能够优先考虑需要更即时关注或干预的患者。例如，对于被识别为某种特定疾病高风险的患者，可以标记其需要更密切监测、预防措施或早期干预。

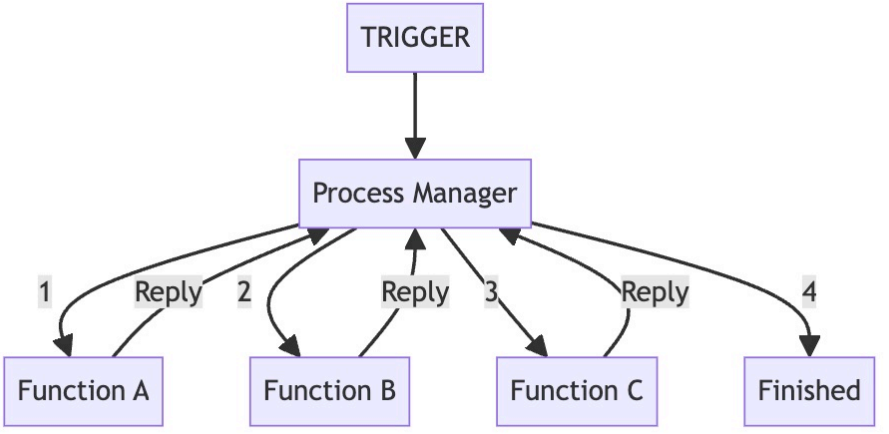
** 个性化风险档案：**AI 工作器可以为每位患者生成个性化风险档案，突出显示影响其风险评分的具体因素。这些档案可以包括患者的生活方式、遗传倾向、环境因素和健康社会决定因素等方面的见解。通过提供风险因素的详细分析，AI 工作器可以帮助医疗服务提供者根据个别患者需求定制预防策略和治疗计划。

** 持续风险监测：**AI 工作器可以持续监测患者数据并实时更新风险评估。随着新信息的出现，如生命体征变化、实验室结果或药物依从性，AI 工作器可以重新计算风险评分并提醒医疗服务提供者任何重要变化。这种主动监测允许及时干预和调整患者护理计划。

** 临床决策支持：**AI 工作器可以将风险评估结果整合到临床决策支持系统中，为医疗服务提供者提供基于证据的建议和警报。例如，如果患者某种特定疾病的风险评分超过某个阈值，AI 工作器可以根据临床指南和最佳实践，提示医疗服务提供者考虑特定的诊断测试、预防措施或治疗选择。

这些工作器能够处理海量的患者数据，应用复杂的分析方法，并生成可执行的见解来支持临床决策。这最终将带来改善的患者预后、降低的医疗成本，以及增强的人群健康管理。

AI 工作器作为流程管理器



在 AI 驱动的应用程序环境中，工作器可以被设计为充当流程管理器的角色，正如 Gregor Hohpe在《企业集成模式》一书中所描述的那样。流程管理器是一个中央组件，负责维护流程的状态，并根据中间结果决定下一步的处理步骤。

当 AI 工作器作为流程管理器运行时，它会接收一个初始化流程的输入消息，即所谓的触发消息。然后，AI 工作器维护流程执行的状态（作为对话记录），并通过一系列实现为工具函数的处理步骤来处理消息，这些步骤可以是顺序的或并行的，并由其自行决定调用。

如果你使用的是像 GPT-4这样知道如何并行执行函数的 AI 模型，那么你的工作器就可以同时执行多个步骤。诚然，我自己还没有尝试过这样做，我的直觉告诉我效果可能会有所不同。

在每个单独的处理步骤之后，控制权都会返回给 AI 工作器，使其能够根据当前状态和获得的结果来确定下一个处理步骤。

存储你的触发消息

根据我的经验，将触发消息实现为数据库支持的对象是明智的。这样，每个流程实例都由唯一的主键标识，并为你提供了一个存储与执行相关的状态的位置，包括 AI 的对话记录。

例如，这里是 Olympia 的AccountChange 模型类的简化版本，它表示对用户账户进行更改的请求。

```
# == Schema Information
#
# Table name: account_changes
#
#  id          :uuid             not null, primary key
#  description :string
#  state       :string           not null
#  transcript  :jsonb
#  created_at  :datetime         not null
#  updated_at  :datetime         not null
#  account_id  :uuid             not null
#
# Indexes
#
#  index_account_changes_on_account_id  (account_id)
#
# Foreign Keys
#
#  fk_rails_...  (account_id => accounts.id)
#
class AccountChange < ApplicationRecord
```

```
  belongs_to :account

  validates :description, presence: true

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

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

AccountChange 类作为触发消息，用于启动处理账户变更请求的流程。注意它是如何在创建事务完成提交后被广播到 Olympia 的基于Wisper的发布/订阅子系统中的。

以这种方式将触发消息存储在数据库中，为每个账户变更请求提供了持久性记录。AccountChange 类的每个实例都被分配一个唯一的主键，便于识别和跟踪单个请求。这对于审计日志记录特别有用，因为它使系统能够维护所有账户变更的历史记录，包括请求的时间、请求的变更内容以及每个请求的当前状态。

在给出的示例中，AccountChange 类包含诸如description 字段来捕获请求变更的详细信息，state 字段来表示请求的当前状态（例如，已请求、已完成、需要人工审核），以及transcript 字段来存储与请求相关的 AI 对话记录。description 字段是用于启动与 AI 首次聊天完成的实际提示。存储这些数据提供了有价值的

上下文，并允许更好地跟踪和分析账户变更流程。

在数据库中存储触发消息可以实现强大的错误处理和恢复能力。如果在处理账户变更请求期间发生错误，系统会将请求标记为失败，并将其转换为需要人工干预的状态。这确保了没有请求会丢失或被遗忘，任何问题都可以得到适当的处理和解决。

作为流程管理器的 AI 工作器提供了一个中央控制点，并实现了强大的流程报告和调试功能。然而，需要注意的是，在应用程序的每个工作流场景中都使用 AI 工作器作为流程管理器可能会显得过度。

将 AI 工作器集成到您的应用程序架构中

在将 AI 工作器集成到应用程序架构时，需要解决几个技术考虑因素，以确保 AI 工作器与其他应用程序组件之间的平滑集成和有效通信。本节考虑设计这些接口、处理数据流和管理 AI 工作器生命周期的关键方面。

设计清晰的接口和通信协议

为了促进 AI 工作器与其他应用程序组件之间的无缝集成，定义清晰的接口和通信协议至关重要。考虑以下方法：

基于 API 的集成：通过定义良好的 API（如 RESTful 端点或 GraphQL 模式）来暴露 AI 工作器的功能。这允许其他组件使用标准的 HTTP 请求和响应与 AI 工作器交互。基于 API 的集成在 AI 工作器和消费组件之间提供了明确的契约，使集成点的开发、测试和维护变得更容易。

基于消息的通信：实现基于消息的通信模式，如消息队列或发布-订阅系统，以实现 AI 工作器与其他组件之间的异步交互。这种方法将 AI 工作器与应用程序的其余部分解耦，实现更好的可扩展性、容错性和松耦合。当 AI 工作器执行的

处理耗时或资源密集时，基于消息的通信特别有用，因为它允许应用程序的其他部分在不等待 AI 工作器完成任务的情况下继续执行。

事件驱动架构：围绕在特定条件满足时激活 AI 工作器的事件和触发器设计系统。AI 工作器可以订阅相关事件并相应地作出反应，在事件发生时执行其指定的任务。事件驱动架构支持实时处理，并允许按需调用 AI 工作器，减少不必要的资源消耗。这种方法非常适合 AI 工作器需要响应特定操作或应用程序状态变化的场景。

处理数据流和同步

在将 AI 工作器集成到应用程序时，确保平滑的数据流并维护 AI 工作器与其他组件之间的数据一致性至关重要。考虑以下方面：

数据准备：在将数据输入 AI 工作器之前，您可能需要执行各种数据准备任务，如清理、格式化和/或转换输入数据。您不仅要确保 AI 工作器能够有效处理，还要确保不会浪费令牌去关注工作器可能认为无用甚至分散注意力的信息。数据准备可能涉及诸如去除噪声、处理缺失值或转换数据类型等任务。

数据持久化：您将如何存储和持久化流入和流出 AI 工作器的数据？考虑数据量、查询模式和可扩展性等因素。您是否需要为了审计或调试目的而保存 AI 的对话记录作为其 “思考过程 “的反映，或者仅保留结果记录就足够了？

数据检索：获取工作程序所需的数据可能涉及查询数据库、读取文件或访问外部 API。请确保考虑延迟问题，以及 AI 工作程序如何访问最新的数据。它们是否需要完全访问您的数据库，还是应该根据其执行的任务严格定义访问范围？扩展性又如何？考虑使用缓存机制来提高性能并减少底层数据源的负载。

数据同步：当包括 AI 工作程序在内的多个组件访问和修改共享数据时，实施适当的同步机制以维护数据一致性非常重要。数据库锁定策略，如乐观锁定或悲观锁定，可以帮助您防止冲突并确保数据完整性。实施事务管理技术来组织相关的数据操作，并维护原子性、一致性、隔离性和持久性（ACID）属性。

错误处理和恢复：实施健壮的错误处理和恢复机制，以处理数据流程中可能出现的数据相关问题。优雅地处理异常并提供有意义的错误消息以帮助调试。实施重试机制和故障转移策略来处理临时故障或网络中断。为数据损坏或丢失情况定义清晰的数据恢复和还原程序。

通过仔细设计和实施数据流和同步机制，您可以确保 AI 工作程序能够访问准确、一致和最新的数据。这使它们能够有效地执行任务并产生可靠的结果。

管理 AI 工作程序的生命周期

开发一个标准化的流程来初始化和配置 AI 工作程序。我倾向于使用那些能够标准化定义模型名称、系统指令和函数定义的框架。确保初始化过程是自动化和可重复的，以便于部署和扩展。

实施全面的监控和日志记录机制来跟踪 AI 工作程序的健康状况和性能。收集资源使用率、处理时间、错误率和吞吐量等指标。使用集中式日志系统，如 ELK 堆栈（Elasticsearch、Logstash、Kibana）来聚合和分析来自多个 AI 工作程序的日志。

在 AI 工作程序架构中建立容错和弹性能力。实施错误处理和恢复机制以优雅地处理故障或异常。大语言模型仍然是前沿技术；提供商往往会在意外时间出现宕机。使用重试机制和断路器来防止级联故障。

AI 工作程序的可组合性和编排

AI 工作程序架构的一个关键优势是其可组合性，这使您能够组合和编排多个 AI 工作程序来解决复杂问题。通过将较大的任务分解为更小、更易管理的子任务，每个子任务由专门的 AI 工作程序处理，您可以创建强大而灵活的系统。在本节中，我们将探讨组合和编排 “多个 “AI 工作程序的不同方法。

链接 AI 工作程序实现多步工作流

在许多场景中，复杂任务可以分解为一系列顺序步骤，其中一个 AI 工作程序的输出成为下一个的输入。这种 AI 工作程序的链接创建了一个多步工作流或管道。链中的每个 AI 工作程序专注于特定的子任务，最终输出是所有工作程序共同努力的结果。

让我们在 Ruby on Rails 应用程序的上下文中考虑一个处理用户生成内容的例子。工作流程涉及以下步骤，诚然，这些步骤在现实用例中可能都太简单，不值得这样分解，但它们使示例更容易理解：

1. 文本清理： 负责删除 HTML 标签、将文本转换为小写并处理 Unicode 标准化的 AI 工作程序。

2. 语言检测： 识别已清理文本语言的 AI 工作程序。

3. 情感分析： 基于检测到的语言确定文本情感（积极、消极或中性）的 AI 工作程序。

4. 内容分类： 使用自然语言处理技术将文本分类到预定义类别的 AI 工作程序。

以下是一个非常简化的示例，展示如何使用 Ruby 将这些 AI 工作程序链接在一起：

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

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

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

在这个示例中，ContentProcessor 类使用原始文本进行初始化，并在process 方法中将 AI 工作器串联在一起。每个 AI 工作器执行其特定任务，并将结果传递给链中的下一个工作器。最终输出是一个包含清理后文本、检测到的语言、情感和内容类别的哈希值。

独立 AI 工作器的并行处理

在前面的示例中，AI 工作器是按顺序链接的，每个工作器处理文本并将结果传递给下一个工作器。然而，如果您有多个可以独立对相同输入进行操作的 AI 工作器，您可以通过并行处理来优化工作流。

在给定的场景中，一旦TextCleanupWorker 完成文本清理，LanguageDetectionWorker、SentimentAnalysisWorker 和CategorizationWorker 都可以独立处理已清理的文本。通过并行运行这些工作器，您可以潜在地减少整体处理时间并提高工作流的效率。

要在 Ruby中实现并行处理，您可以利用诸如线程或异步编程之类的并发技术。以下是如何修改ContentProcessor 类以使用线程并行处理最后三个工作器的示例：

```
require 'concurrent'

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

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

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

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

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

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

```

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

在这个优化版本中，我们使用 concurrent-ruby 库为每个独立的 AI 工作器创建 Concurrent::Future 对象。Future 代表一个将在单独线程中异步执行的计算。

在文本清理步骤之后,我们创建了三个 Future 对象:language_future、sentiment_future 和 category_future。每个 Future 在单独的线程中执行其对应的 AI 工作器（LanguageDetectionWorker、SentimentAnalysisWorker 和 CategorizationWorker），将 cleaned_text 作为输入。

通过调用每个 Future 的 value 方法，我们等待计算完成并获取结果。value 方法会阻塞直到结果可用，确保所有并行工作器在继续执行之前都已完成处理。

最后，我们像原始示例一样，使用清理后的文本和并行工作器的结果构建输出哈希。

通过并行处理独立的 AI 工作器，与顺序执行相比，你可以潜在地减少整体处理时间。这种优化在处理耗时任务或处理大量数据时特别有益。

然而，需要注意的是，实际的性能提升取决于多个因素，如每个工作器的复杂度、可用的系统资源以及线程管理的开销。始终建议对代码进行基准测试和性能分析，以确定特定用例的最佳并行级别。

此外，在实现并行处理时，要注意工作器之间的任何共享资源或依赖关系。确保工作器可以独立运行而不会发生冲突或竞态条件。如果存在依赖关系或共享资源，你可能需要实现适当的同步机制来维护数据完整性并避免死锁或不一致结果等问题。

Ruby 的全局解释器锁和异步处理

在考虑 Ruby 中基于线程的异步处理时，理解 Ruby 的全局解释器锁（GIL）的影响很重要。

GIL 是 Ruby 解释器中的一种机制，它确保即使在多核处理器上也只能同时执行一个线程的 Ruby 代码。这意味着虽然可以在 Ruby 进程中创建和管理多个线程，但在任何给定时刻只有一个线程可以主动执行 Ruby 代码。

GIL 的设计是为了简化 Ruby 解释器的实现并为 Ruby 的内部数据结构提供线程安全。然而，它也限制了 Ruby 代码真正并行执行的可能性。

当你在 Ruby 中使用线程时，比如使用 `concurrent-ruby` 库或内置的 `Thread` 类，这些线程都受 GIL 约束。GIL 允许每个线程执行一小段时间的 Ruby 代码，然后切换到另一个线程，创造并发执行的假象。

然而，由于 GIL 的存在，Ruby 代码的实际执行仍然是顺序的。当一个线程正在执行 Ruby 代码时，其他线程实际上是暂停的，等待轮到它们获取 GIL 并执行。

这意味着 Ruby 中基于线程的异步处理对 I/O 密集型任务最有效，例如等待外部 API 响应（如第三方托管的大型语言模型）或执行文件 I/O 操作。当线程遇到 I/O 操作时，它可以释放 GIL，允许其他线程在等待 I/O 完成时执行。

另一方面，对于 CPU 密集型任务，如密集计算或长时间运行的 AI 工作器处理，GIL 可能会限制线程并行的潜在性能提升。由于同一时间只能执行一个线程的 Ruby 代码，与顺序处理相比，整体执行时间可能不会显著减少。

要在 Ruby 中实现 CPU 密集型任务的真正并行执行，你可能需要探索其他方法，例如：

- 使用基于进程的并行处理，每个 Ruby 进程运行在单独的 CPU 核心上。
- 利用提供原生扩展或接口的外部库或框架，连接到没有 GIL 的语言，如 C 或 Rust。，
- 使用分布式计算框架或消息队列在多台机器或进程之间分配任务。

在设计和实现 Ruby 的异步处理时，考虑任务的性质和 GIL 带来的限制至关

> 重要。虽然基于线程的异步处理对 I/O 密集型任务有益，但由于 GIL 的限制，对 CPU 密集型任务可能不会带来显著的性能改进。

提高准确性的集成技术

集成技术涉及将多个 AI 工作器的输出组合起来，以提高系统的整体准确性或鲁棒性。集成技术不是依赖单个 AI 工作器，而是利用多个工作器的集体智慧来做出更明智的决策。

集成模型在工作流程的不同部分需要不同 AI 模型的情况下特别重要，这种情况比你想象的要普遍。与功能较弱的开源选项相比，像 GPT-4这样的强大模型的成本极其昂贵，而且可能并不是应用程序的每个工作流程步骤都需要使用它。

多数投票是一种常见的集成技术，多个 AI 工作器独立处理相同的输入，最终输出由多数共识决定。这种方法可以帮助减少单个工作器错误的影响，提高系统的整体可靠性。

让我们考虑一个例子，我们有三个用于情感分析的 AI 工作器，每个工作器使用不同的模型或提供不同的上下文。我们可以使用多数投票来组合它们的输出，从而确定最终的情感预测。

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

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

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

  end
end
```

在这个例子中,SentimentAnalysisEnsemble类在初始化时接收文本并调用三个不同的情感分析 AI 工作器。analyze 方法收集每个工作器的预测结果，并通过group_by 和max_by 方法来确定多数情感倾向。最终输出是在工作器集成中获得最多投票的情感倾向。

集成系统显然是值得花时间研究并行处理的场景。

AI 工作器的动态选择和调用

在某些（如果不是大多数）情况下，具体要调用哪个 AI 工作器可能取决于运行时条件或用户输入。AI 工作器的动态选择和调用为系统提供了灵活性和适应性。

你可能会想要尝试在单个 AI 工作器中装入大量功能，给它很多函数和一个解释如何调用这些函数的复杂提示。相信我，要抵制这种诱惑。我们在本章讨论的方法之所以被称为 “多工作器模式 “，其中一个原因就是要提醒我们：拥有许多专门的工作器是可取的，每个工作器都在为更大的目标做着自己的小任务。

例如，考虑一个聊天机器人应用，其中不同的 AI 工作器负责处理不同类型的用户查询。基于用户的输入，应用程序动态选择合适的 AI 工作器来处理查询。

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

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

```

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

在这个例子中，`ChatbotController` 通过`process_query` 动作接收用户查询。它首先使用`QueryClassifierWorker` 来确定查询的类型。根据分类后的查询类型，控制器动态选择合适的 AI 工作器来生成响应。这种动态选择使聊天机器人能够处理不同类型的查询，并将它们路由到相关的 AI 工作器。

由于`QueryClassifierWorker` 的工作相对简单，不需要太多上下文或函数定义，你可能可以使用超快速的小型 LLM 来实现它，比如`mistralai/mixtral-8x7b-instruct:nitro`。在我写这篇文章时，它在许多任务上的能力接近 GPT-4水平，而且 Groq可以以惊人的 444 个词元/秒的速度提供服务。

将传统 NLP 与 LLM 相结合

虽然大型语言模型（LLMs）彻底改变了自然语言处理（NLP）领域，在广泛的任务中提供了无与伦比的通用性和性能，但它们并非对每个问题都是最高效或最具成本效益的解决方案。在许多情况下，将传统 NLP 技术与 LLM 相结合可以带来更优化、更有针对性和更经济的方法来解决特定的 NLP 挑战。

可以将 LLM 视为 NLP 领域的瑞士军刀——功能极其多样且强大，但并非适用于每项工作的最佳工具。有时候，像开瓶器或罐头刀这样的专用工具可能对特定任务更有效和高效。同样，传统的 NLP 技术，如文档聚类、主题识别和分类，通常可以为 NLP 流程的某些方面提供更有针对性和更具成本效益的解决方案。

传统 NLP 技术的一个主要优势是其计算效率。这些方法通常依赖于更简单的统计模型或基于规则的方法，与 LLM 相比，可以更快速地处理大量文本数据，且计算开销更低。这使它们特别适合于涉及分析和组织大量文档的任务，例如聚类相似文章或识别文本集合中的关键主题。

此外，传统 NLP 技术在特定任务上通常可以达到很高的准确率和精确度，特别是在针对特定领域数据集进行训练时。例如，使用传统机器学习算法（如支持向量机（SVM）或朴素贝叶斯）的精心调优的文档分类器可以以最小的计算成本准确地将文档分类到预定义的类别中。

然而，当涉及需要更深入理解语言、上下文和推理的任务时，LLM 真正展现出其优势。它们生成连贯且与上下文相关的文本、回答问题和总结长段落的能力是传统 NLP 方法无法比拟的。LLM 可以有效处理复杂的语言现象，如歧义、指代和习语表达，这使它们在需要自然语言生成或理解的任务中变得不可或缺。

真正的力量在于将传统 NLP 技术与 LLM 结合，创建能够利用两者优势的混合方法。通过使用传统 NLP 方法进行文档预处理、聚类和主题提取等任务，你可以高效地组织和构建文本数据。然后，这些结构化信息可以输入到 LLM 中，用于更高级的任务，如生成摘要、回答问题或创建综合报告。

例如，让我们考虑一个用例，你想基于大量单独的趋势文档为特定领域生成趋势报告。与其完全依赖 LLM（这对于处理大量文本来说可能在计算上昂贵且耗时），你可以采用混合方法：

1. 使用传统 NLP 技术，如主题建模（例如，隐狄利克雷分布）或聚类算法（例如，K-均值），将相似的趋势文档分组，并识别语料库中的关键主题。
2. 将聚类后的文档和识别出的主题输入到 LLM 中，利用其卓越的语言理解和生成能力为每个聚类或主题创建连贯且信息丰富的摘要。
3. 最后，使用 LLM 通过组合各个摘要生成全面的趋势报告，突出最重要的趋势，并基于聚合信息提供见解和建议。

通过以这种方式将传统 NLP 技术与 LLM 结合，你可以高效地处理大量文本数据，提取有意义的见解，并生成高质量的报告，同时优化计算资源和成本。

在开展自然语言处理项目时，必须仔细评估每项任务的具体需求和限制，并考虑如何将传统的自然语言处理方法与大型语言模型结合起来，以获得最佳效果。通过将传统技术的高效性和精确性与大型语言模型的多功能性和强大功能相结

合，您可以创建高效且经济的自然语言处理解决方案，为用户和利益相关者带来价值。

工具使用

在 AI 驱动的应用程序开发领域中，“工具使用”或 “函数调用“的概念已经发展成为一项强大的技术，使您的 LLM能够连接到外部工具、API、函数、数据库和其他资源。这种方法不仅能够实现比单纯输出文本更丰富的行为，还能在 AI 组件与应用程序生态系统的其他部分之间实现更动态的交互。正如我们将在本章中探讨的那样，工具使用还为您提供了使 AI 模型以结构化方式生成数据的选择。

什么是工具使用？

工具使用，也称为函数调用，是一种允许开发者指定 LLM 在生成过程中可以交互的函数列表的技术。这些工具可以是简单的实用函数，也可以是复杂的 API 或数据库查询。通过为 LLM 提供这些工具的访问权限，开发者可以扩展模型的功能，使其能够执行需要外部知识或操作的任务。

图 7. 用于分析文档的 AI 工作者的函数定义示例

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

工具使用背后的核心理念是赋予大语言模型基于用户输入或当前任务动态选择

和执行适当工具的能力。工具使用使模型不仅仅依赖于预训练知识，还能利用外部资源生成更准确、更相关、更具可操作性的响应。工具使用使得检索增强生成（RAG）等技术的实现比以往更加容易。

> 请注意，除非另有说明，本书假设您的 AI 模型没有访问任何内置服务器端工具的权限。您想要提供给 AI 使用的任何工具都必须在每个 API 请求中由您明确声明，并在 AI 表示想要在其响应中使用该工具时提供执行调度的规定。

工具使用的潜力

工具使用为 AI 驱动的应用开启了广泛的可能性。以下是一些可以通过工具使用实现的例子：

1. **聊天机器人和虚拟助手**：通过将大语言模型连接到外部工具，聊天机器人和虚拟助手可以执行更复杂的任务，例如从数据库检索信息、执行 API 调用或与其他系统交互。例如，聊天机器人可以使用 CRM 工具根据用户请求更改交易状态。
2. **数据分析和洞察**：大语言模型可以连接到数据分析工具或库以执行高级数据处理任务。这使应用程序能够根据用户查询生成洞察、进行比较分析或提供数据驱动的建议。
3. **搜索和信息检索**：工具使用允许大语言模型与搜索引擎、向量数据库或其他信息检索系统进行交互。通过将用户查询转换为搜索查询，大语言模型可以从多个来源检索相关信息，并为用户问题提供全面的答案。
4. **与外部服务集成**：工具使用实现了 AI 驱动的应用与外部服务或 API 的无缝集成。例如，大语言模型可以与天气 API 交互以提供实时天气更新，或与翻译 API 交互以生成多语言响应。

工具使用工作流程

工具使用工作流程通常包含四个关键步骤：

1. 在请求上下文中包含函数定义
2. 动态（或显式）工具选择
3. 执行函数
4. 可选的原始提示词继续

让我们详细回顾每个步骤。

在请求上下文中包含函数定义

AI 知道它可以使用哪些工具，是因为您在补全请求中提供了一个工具列表（通常使用 JSON 模式的变体定义为函数）。

工具定义的具体语法因模型而异。

以下是在 Claude 3中定义`get_weather` 函数的方式：

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

```
                "type": "string",
                "enum": ["celsius", "fahrenheit"],
                "description": "The unit of temperature"
            }
        },
        "required": ["location"]
        }
}
```

这就是如何为 GPT-4 定义相同的函数，将其作为 `tools` 参数的值：

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

> 几乎一样，只是莫名其妙地有所不同！真让人烦恼。

函数定义需要指定名称、描述和输入参数。 输入参数可以通过属性进行进一步定义，比如使用枚举来限制可接受的值，以及指定参数是否为必需。

除了实际的函数定义之外，你还可以在系统指令中包含使用该函数的原因和方法的说明。

例如，我在 Olympia中的网络搜索工具包含这样的系统指令，它提醒 AI可以使用上述提到的工具：

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

提供详细描述被认为是工具性能中最重要的因素。你的描述应该解释关于工具的每个细节，包括：

- 工具的功能
- 何时应该使用（以及何时不应该使用）
- 每个参数的含义以及它们如何影响工具的行为
- 适用于工具实现的任何重要注意事项或局限性

你能为 AI 提供的关于工具的上下文越多，它就越能更好地决定何时以及如何使用这些工具。例如，Anthropic 建议其 Claude 3 系列的每个工具描述至少包含 3-4 个句子，如果工具比较复杂，则需要更多。

这可能不太直观，但描述被认为比示例更重要。虽然你可以在工具描述或附带的提示中包含如何使用工具的示例，但这比有一个清晰和全面的工具用途及参数说明要次要。只有在完整阐述描述之后才添加示例。

以下是一个类似 Stripe 的 API 函数规范示例：

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

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

实际上，某些模型在处理嵌套函数规范和复杂输出数据类型时会遇到困难，比如数组、字典等。但理论上，你应该能够提供任意深度的 JSON 模式规范！

动态工具选择

当你执行包含工具定义的聊天完成时，LLM 会动态选择最合适的工具，并为每个工具生成所需的输入参数。

实际上，AI 精确调用正确函数并严格遵循你的输入规范的能力时好时坏。将温度超参数完全调至 0.0 会有很大帮助，但根据我的经验，你仍会偶尔遇到错误。这些失败包括虚构的函数名称、错误命名或完全缺失的输入参数。参数以 JSON 格式传递，这意味着有时你会看到由截断、引号错误或其他破损 JSON 导致的错误。

自修复数据模式可以帮助自动修复由语法错误导致的函数调用失败。

强制（又称显式）工具选择

某些模型允许你在请求中通过参数强制调用特定函数。否则，是否调用函数完全取决于 AI 的判断。

在某些场景下，强制函数调用的能力至关重要，因为你可能想要确保执行特定工具或函数，而不考虑 AI 的动态选择过程。这种能力的重要性体现在以下几个方面：

1. **明确控制**：你可能正在将 AI 用作离散组件，或在预定义的工作流程中需要在特定时间执行特定函数。通过强制调用，你可以确保所需函数被调用，而不是要委婉地请求 AI 去执行。

2. **调试和测试**：在开发和测试 AI 驱动的应用程序时，强制函数调用的能力对调试非常重要。通过显式触发特定函数，你可以隔离并测试应用程序的各个组件。这使你能够验证函数实现的正确性，验证输入参数，并确保返回预期结果。
3. **处理边缘情况**：可能存在一些边缘情况或特殊场景，其中 AI 的动态选择过程可能不会选择执行本应执行的函数，而你基于外部流程知道这一点。在这种情况下，具备强制函数调用的能力允许你明确处理这些情况。在应用程序逻辑中定义规则或条件来决定何时覆盖 AI 的判断。
4. **一致性和可重现性**：如果你有一系列需要按特定顺序执行的函数，强制调用可以确保每次都遵循相同的序列。这在需要一致性和可预测行为的应用中尤为重要，比如金融系统或科学模拟。
5. **性能优化**：在某些情况下，强制函数调用可以带来性能优化。如果你知道特定任务需要某个函数，而 AI 的动态选择过程可能引入不必要的开销，你可以绕过选择过程直接调用所需函数。这可以帮助减少延迟并提高应用程序的整体效率。

总之，在 AI 驱动的应用程序中强制函数调用的能力提供了明确的控制，有助于调试和测试，处理边缘情况，确保一致性和可重现性。这是你武器库中的一个强大工具，但我们还需要讨论这个重要特性的另一个方面。

在许多决策用例中，我们总是希望模型进行函数调用，而可能永远不希望模型仅使用其内部知识来响应。 例如，如果你在多个专门处理不同任务（多语言输入、数学等）的模型之间进行路由，你可能会使用函数调用模型将请求委派给其中一个辅助模型，而不是独立响应。

工具选择参数

GPT-4和其他支持函数调用的语言模型为你提供了`tool_choice` 参数来控制完成过程中是否需要使用工具。该参数有三个可能的值：

- auto 让 AI 完全自行决定是否使用工具或直接响应
- required 告诉 AI 它必须调用工具而不是响应，但让 AI 自行选择具体工具
- 第三个选项是设置你想要强制使用的name_of_function 参数。更多内容将在下一节讨论。

请注意，如果您将工具选择（tool choice）设置为required，模型将被迫从提供的函数中选择最相关的一个来调用，即使没有真正适合提示的函数。在发布时，我还没有见过能够返回空的tool_calls 响应，或使用其他方式来告知未找到合适函数的模型。

强制函数调用以获取结构化输出

强制函数调用的能力为您提供了一种方法,可以从聊天完成中强制获取结构化数据,而不必自己从纯文本响应中提取数据。

> 为什么强制函数获取结构化输出如此重要？简单来说，因为从大语言模型输出中提取结构化数据是件令人头疼的事。您可以通过要求数据采用 XML格式来让生活稍微轻松一点，但随后您还得解析 XML。而当您的人工智能回应：“抱歉，由于巴拉巴拉的原因，我无法生成您请求的数据...“时，您又该如何处理这种缺少 XML 的情况？

在这种方式下使用工具时：

- 您可能应该在请求中只定义一个工具
- 记住使用tool_choice 参数来强制使用其函数
- 记住模型将向工具传递输入，所以工具的名称和描述应该从模型的角度出发，而不是您的角度

最后一点需要一个例子来说明。假设您正在请求人工智能对用户文本进行情感分析。函数名称不应该是`analyze_sentiment`，而应该是类似`save_sentiment_analysis`这样的名称。进行情感分析的是人工智能，而不是工具。从人工智能的角度来看，工具所做的只是保存分析结果。

下面是一个使用 Claude 3将图像摘要记录为结构良好的 JSON的示例，这次是使用`curl` 从命令行进行操作：

```
curl https://api.anthropic.com/v1/messages \
     --header "content-type: application/json" \
     --header "x-api-key: $ANTHROPIC_API_KEY" \
     --header "anthropic-version: 2023-06-01" \
     --header "anthropic-beta: tools-2024-04-04" \
     --data \
'{
    "model": "claude-3-sonnet-20240229",
    "max_tokens": 1024,
    "tools": [{
        "name": "record_summary",
        "description": "Record summary of image into well-structured JSON.",
        "input_schema": {
            "type": "object",
            "properties": {
                "key_colors": {
                    "type": "array",
                    "items": {
                        "type": "object",
                        "properties": {
                            "r": {
                                "type": "number",
```

```
                                "description": "red value [0.0, 1.0]"
                            },
                            "g": {
                                "type": "number",
                                "description": "green value [0.0, 1.0]"
                            },
                            "b": {
                                "type": "number",
                                "description": "blue value [0.0, 1.0]"
                            },
                            "name": {
                                "type": "string",
                                "description": "Human-readable color name
                                                in snake_case, e.g.
                                                \"olive_green\"or
                                                \"turquoise\""
                            }
                        },
                        "required": [ "r", "g", "b", "name" ]
                    },
                    "description": "Key colors in the image. Four or less."
                },
                "description": {
                    "type": "string",
                    "description": "Image description. 1-2 sentences max."
                },
                "estimated_year": {
                    "type": "integer",
                    "description": "Estimated year that the image was taken,
```

```
                                    if is it a photo. Only set this if the
                                    image appears to be non-fictional.
                                    Rough estimates are okay!"
                }
            },
            "required": [ "key_colors", "description" ]
        }
    }],
    "messages": [
        {
            "role": "user",
            "content": [
                {
                    "type": "image",
                    "source": {
                        "type": "base64",
                        "media_type": "'$IMAGE_MEDIA_TYPE'",
                        "data": "'$IMAGE_BASE64'"
                    }
                },
                {
                    "type": "text",
                    "text": "Use `record_summary` to describe this image."
                }
            ]
        }
    ]
}'
```

在提供的示例中，我们使用来自 Anthropic的 Claude 3模型来生成图像的结构化

JSON 摘要。以下是其工作原理：

1. 我们在请求载荷的`tools`数组中定义了一个名为`record_summary`的工具。该工具负责将图像的摘要记录为结构良好的 JSON。
2. `record_summary`工具具有一个`input_schema`，指定了预期的 JSON 输出结构。它定义了三个属性：

    - `key_colors`：表示图像中关键颜色的对象数组。每个颜色对象都具有红、绿、蓝值（范围从 0.0 到 1.0）的属性，以及一个采用蛇形命名法格式的人类可读颜色名称。
    - `description`：用于简短描述图像的字符串属性，限制为 1-2 个句子。
    - `estimated_year`：可选的整数属性，用于估计图像拍摄的年份（如果它看起来是非虚构照片）。

3. 在`messages`数组中，我们提供了以 base64 编码字符串形式的图像数据以及媒体类型。这使模型能够将图像作为输入的一部分进行处理。
4. 我们还提示 Claude 使用`record_summary`工具来描述图像。
5. 当请求发送到 Claude 3 模型时，它会分析图像并基于指定的`input_schema`生成 JSON 摘要。模型提取关键颜色，提供简短描述，并估计图像拍摄的年份（如果适用）。
6. 生成的 JSON 摘要作为参数传递给`record_summary`工具，提供图像关键特征的结构化表示。

通过使用具有明确定义的`input_schema`的`record_summary`工具，我们可以获得图像的结构化 JSON 摘要，而无需依赖纯文本提取。这种方法确保输出遵循一致的格式，并且可以被应用程序的下游组件轻松解析和处理。

在 AI 驱动的应用程序中，强制函数调用并指定预期输出结构的能力是工具使用的一个强大特性。它使开发人员能够更好地控制生成的输出，并简化了将 AI 生成的数据集成到应用程序工作流程中的过程。

函数的执行

你已经定义了函数，并提示了你的 AI，而 AI 决定它应该调用你的某个函数。现在是时候让你的应用程序代码或库（如果你使用的是raix-rails这样的 Ruby gem）将函数调用及其参数分派到你的应用程序代码中的相应实现。

你的应用程序代码决定如何处理函数执行的结果。可能需要做的事情包括在 lambda 中执行一行代码，或者调用外部 API。可能涉及调用另一个 AI 组件，或者可能涉及系统其余部分中的数百甚至数千行代码。这完全取决于你。

有时函数调用就是操作的终点，但如果结果代表了需要 AI 继续处理的思维链中的信息，那么你的应用程序代码需要将执行结果插入到聊天记录中，让 AI 继续处理。

例如，这里是 Olympia 的AccountManager 使用的Raix函数声明，用于在客户服务的智能工作流程编排中与我们的客户进行通信。

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

  # lots of other functions...

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

这里发生的事情可能不是立即就能明白的，所以我来详细解释一下。

1. `AccountManager` 类定义了许多与账户管理相关的功能。它可以更改您的计划，添加和删除团队成员，以及执行其他操作。
2. 其顶层指令告诉 `AccountManager` 应该使用 `notify_account_owner` 函数将账户更改请求的结果通知账户所有者。
3. 该函数的简明定义包括其：

- 名称
- 描述
- 参数 `message: { type: "string" }`
- 函数被调用时要执行的代码块

在使用函数执行结果更新对话记录后，`chat_completion` 方法会再次被调用。这个方法负责将更新后的对话记录发送回 AI 模型进行进一步处理。我们将这个过程称为对话循环。

当 AI 模型收到包含更新记录的新聊天补全请求时，它可以访问之前执行的函数的结果。它可以分析这些结果，将它们纳入决策过程中，并根据对话的累积上下文生成下一个响应或动作。它可以根据更新后的上下文选择执行额外的函数，或者如果确定不需要进一步的函数调用，它可以生成对原始提示的最终响应。

原始提示的可选继续

当你将工具结果发送回 LLM并继续处理原始提示时，AI 会使用这些结果来调用额外的函数或生成最终的纯文本响应。

一些模型，如 Cohere 的Command-R可以在其响应中引用它们使用的具体工具，提供额外的透明度和可追溯性。

根据使用的模型不同，函数调用的结果将存在于具有特殊角色的记录消息中，或以其他语法形式反映。但重要的是这些数据要在记录中，这样 AI 在决定下一步行动时就能考虑到这些数据。

一个常见（且可能代价高昂）的错误情况是在继续对话之前忘记将函数结果添加到记录中。结果就是，AI 会以与第一次调用函数之前基本相同的方式被提示。换句话说，就 AI 而言，它还没有调用该函数。所以它会再次调用。然后一次又一次，直到你中断它。希望你的上下文不会太大，你的模型不会太贵！

工具使用的最佳实践

要充分利用工具，请考虑以下最佳实践。

描述性定义

为每个工具及其输入参数提供清晰和描述性的名称和说明。这有助于 LLM 更好地理解每个工具的目的和功能。

> 从经验来看，我可以告诉你 “命名很难 “这个普遍的观点在这里也适用；我见过仅仅通过更改函数名称或描述措辞就能让 LLM 产生显著不同结果的情况。有时候删除描述反而会提高性能。

工具结果处理

当将工具结果传回 LLM 时，确保它们结构良好且全面。使用有意义的键和值来表示每个工具的输出。尝试不同的格式，看看哪种效果最好，从 JSON 到纯文本都可以。

结果解释器通过使用 AI 来分析结果并提供人性化的解释、摘要或关键要点来解决这个挑战。

错误处理

实现强大的错误处理机制，以处理 LLM 可能为工具调用生成无效或不支持的输入参数的情况。优雅地处理并从工具执行期间可能发生的任何错误中恢复。

AI 的一个极其好的特质是它能理解错误消息！这意味着如果你采用快速且简单的思维方式，你可以简单地捕获工具实现中产生的任何异常，并将其传回 AI，这样它就知道发生了什么！

例如，这是 Olympia 中 Google 搜索实现的精简版本：

```
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
```

在 Olympia 中进行 Google 搜索是一个两步过程。首先执行搜索，然后总结结果。如果出现任何类型的失败，异常信息都会被打包并发送回 AI。这种技术是几乎所有智能错误处理模式的基础。

例如，假设 `GoogleSearch` API 调用由于 503 服务不可用异常而失败。这个错误会冒泡到顶层的异常处理，然后错误描述作为函数调用的结果被发送回 AI。AI 不

会仅仅向用户显示空白屏幕或技术性错误，而是会说类似“抱歉，我目前无法访问 Google 搜索功能。如果您愿意的话，我可以稍后重试“这样的话。

这可能看起来只是一个巧妙的技巧，但考虑另一种错误情况：当 AI 调用外部 API 并且能直接控制传递给 API 的参数时。也许它在生成这些参数时出错了？如果外部 API 的错误信息足够详细，将错误信息传回调用 AI 意味着它可以重新考虑这些参数并重试。这个过程是自动的，不管是什么类型的错误。

现在想想在普通代码中实现这种强大的错误处理需要做多少工作。这几乎是不可能的。

迭代优化

如果 LLM 没有推荐合适的工具或生成的响应不够理想，可以对工具定义、描述和输入参数进行迭代。根据观察到的行为和期望的结果，持续改进和优化工具设置。

1. 从简单的工具定义开始：首先定义具有清晰简洁的名称、描述和输入参数的工具。最初避免过度复杂化工具设置，专注于核心功能。例如，如果你想保存情感分析的结果，可以从这样的基本定义开始：

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

```
    },
    "required": ["score"]
  }
}
```

2. 测试和观察：一旦完成初始工具定义，用不同的提示词对工具进行测试，并观察大语言模型如何与工具交互。注意生成响应的质量和相关性。如果大语言模型生成的是次优响应，那就需要优化工具定义。
3. 优化描述：如果大语言模型误解了工具的用途，尝试优化工具的描述。提供更多的上下文、示例或说明，以引导大语言模型更有效地使用工具。例如，你可以更新情感分析工具的描述，使其更具体地针对被分析文本的情感基调：

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

4. 调整输入参数：如果大语言模型生成的工具输入参数无效或不相关，请考虑调整参数定义。添加更具体的约束条件、验证规则或示例，以明确预期的输入格式。
5. 基于反馈进行迭代：持续监控工具的性能表现，并收集用户或利益相关者的反馈。利用这些反馈来识别需要改进的领域，并对工具定义进行迭代改进。例如，如果用户反映分析无法很好地处理讽刺语气，你可以在描述中添加说明：

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

通过基于观察到的行为和反馈来迭代改进工具定义，你可以逐步提升 AI 驱动应用程序的性能和效果。请记住要保持工具定义的清晰、简洁，并专注于具体的任务。定期测试和验证工具交互，确保它们符合你期望的结果。

组合和链接工具

到目前为止我们只是略微提到了工具使用最强大的特性之一，那就是将多个工具组合和链接在一起以完成复杂任务的能力。通过仔细设计工具定义及其输入/输出格式，你可以创建可以以各种方式组合的可重用构建块。

让我们考虑一个例子，假设你正在为 AI 驱动的应用程序构建数据分析流程。你可能会有以下工具：

1. **`DataRetrieval`：** 根据指定条件从数据库或 API 获取数据的工具。
2. **`DataProcessing`：** 对检索到的数据执行计算、转换或聚合的工具。
3. **`DataVisualization`：** 以用户友好的格式（如图表或图形）呈现处理后数据的工具。

通过将这些工具链接在一起，你可以创建一个强大的工作流，用于检索相关数据、处理数据并以有意义的方式呈现结果。以下是工具使用工作流程的可能样子：

1. 大语言模型（LLM）接收用户查询，要求获取特定产品类别的销售数据洞察。
2. LLM 选择`DataRetrieval` 工具，并生成适当的输入参数以从数据库获取相关销售数据。
3. 检索到的数据被 “传递 “给`DataProcessing` 工具，该工具计算总收入、平均销售价格和增长率等指标。
4. 处理后的数据随后被`DataVisualization` 工具消化，该工具创建一个视觉吸引力的图表来表示这些洞察，并将图表的 URL 传回给 LLM。
5. 最后，LLM 使用 markdown生成格式化的用户查询响应，整合可视化数据并提供关键发现的摘要。

通过组合这些工具，你可以创建一个可以轻松集成到应用程序中的无缝数据分析工作流。这种方法的优点在于每个工具都可以独立开发和测试，然后以不同方式组合来解决各种问题。

为了实现工具的顺畅组合和链接，为每个工具定义清晰的输入和输出格式很重要。

例如，`DataRetrieval` 工具可能接受数据库连接详情、表名和查询条件等参数，并将结果集作为结构化 JSON 对象返回。`DataProcessing` 工具随后可以将此 JSON 对象作为输入，并生成转换后的 JSON 对象作为输出。通过标准化工具之间的数据流，你可以确保兼容性和可重用性。

在设计你的工具生态系统时，要考虑如何组合不同的工具来解决应用程序中的常见用例。考虑创建封装常见工作流程或业务逻辑的高级工具，使 LLM 更容易选择和有效使用它们。

记住，工具使用的力量在于它提供的灵活性和模块化。通过将复杂任务分解为更小的、可重用的工具，你可以创建一个强大且适应性强的 AI 驱动应用程序，能够应对广泛的挑战。

未来方向

随着 AI 驱动应用程序开发领域的发展，我们可以期待工具使用能力的进一步提升。一些潜在的未来方向包括：

1. **多跳工具使用**：LLM 可能能够决定需要使用工具的次数，以生成令人满意的响应。这可能涉及基于中间结果的多轮工具选择和执行。
2. **预定义工具**：AI 平台可能提供一套开发者可以直接使用的预定义工具，如 Python 解释器、网络搜索工具或常用实用函数。
3. **无缝集成**:随着工具使用变得更加普遍,我们可以期待 AI 平台与流行开发框架之间的更好集成，使开发者更容易在其应用程序中 incorporate 工具使用。

工具使用是一种强大的技术，使开发者能够在 AI 驱动的应用程序中充分发挥 LLM 的潜力。通过将 LLM 连接到外部工具和资源，你可以创建更加动态、智能和具有上下文感知能力的系统，能够适应用户需求并提供有价值的洞察和行动。

虽然工具使用提供了巨大的可能性，但重要的是要意识到潜在的挑战和考虑因素。一个关键方面是管理工具交互的复杂性，并确保整个系统的稳定性和可靠性。你需要处理工具调用可能失败、返回意外结果或产生性能影响的情况。此外，你还应该考虑安全和访问控制措施，以防止未经授权或恶意使用工具。适当的错误处理、日志记录和监控机制对于维护 AI 驱动应用程序的完整性和性能至关重要。

在探索在您自己的项目中使用工具的可能性时，请记住要从明确的目标开始，设计结构良好的工具定义，并根据反馈和结果进行迭代。采用正确的方法和思维模式，工具使用可以为您的人工智能驱动的应用开启创新和价值的新层次

流处理

通过 HTTP进行流数据处理，也称为服务器发送事件（SSE），是一种服务器在数据可用时持续向客户端发送数据的机制，无需客户端显式请求。由于 AI 的响应是逐步生成的，通过在生成过程中实时显示 AI 的输出来提供响应式的用户体验是很有意义的。事实上，据我所知，所有 AI 提供商的 API 都在其补全端点中提供了流式响应选项。

本章在使用工具之后立即出现的原因，是因为将工具使用与实时 AI 响应相结合可以产生强大的效果。这样做可以创造动态和交互式的体验，其中 AI可以处理用户输入，自行使用各种工具和函数，然后提供实时响应。

要实现这种无缝交互,你需要编写能够分发 AI 调用的工具函数以及向最终用户输出纯文本的流处在处理工具函数后需要循环的要求为这项工作增添了一个有趣的挑战。

实现 ReplyStream

为了演示如何实现流处理，本章将深入探讨 Olympia中使用的`ReplyStream` 类的简化版本。这个类的实例可以作为`stream` 参数传递给 AI 客户端库，如ruby-openai和openrouter。

以下是我在 Olympia 的`PromptSubscriber` 中使用`ReplyStream` 的方式，它通过 Wisper监听新用户消息的创建。

```
1  class PromptSubscriber
2    include Raix::ChatCompletion
3    include Raix::PromptDeclarations
4
5    # many other declarations omitted...
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 omitted...
```

除了有一个指向实例化它的提示订阅者的`context` 引用外，`ReplyStream` 类还具有用于存储接收数据缓冲区的实例变量，以及用于跟踪流处理过程中调用的函数名称和参数的数组。

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

  delegate :bot_message, :dispatch, to: :context

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

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

  # ...
end
```

`initialize` 方法用于设置 `ReplyStream` 实例的初始状态，初始化缓冲区、上下文和其他变量。

`call` 方法是处理流数据的主要入口点。 它接收一个数据块（以哈希形式表示）和一个可选的 `bytesize` 参数，在我们的示例中这个参数未被使用。在这个方法内部，该类使用模式匹配来根据接收到的数据块的结构处理不同的场景。

对数据块调用 `deep_symbolize_keys` 有助于使模式匹配更加优雅，因为它让我们可以操作符号而不是字符串。

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

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

    f_name[index] = name
```

我们要匹配的第一个模式是工具调用及其相关的函数名。如果检测到一个工具调用，我们就将它存入`f_name` 数组中。我们将函数名存储在索引数组中，因为模型能够进行并行函数调用，同时发送多个函数来执行。

> 并行函数调用是 AI 模型同时执行多个函数调用的能力，允许这些函数调用的效果和结果并行解析。这在函数执行时间较长的情况下特别有用，并且可以减少与 API的往返通信次数，从而可以节省大量的令牌消耗。

接下来我们需要匹配与函数调用相对应的参数。

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

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

与我们处理函数名的方式类似，我们将参数存储在一个索引数组中。

接下来，我们关注普通的面向用户的消息，这些消息将从服务器一个标记一个标记地到达，并被分配给`new_content` 变量。我们还需要留意`finish_reason`。在输出序列的最后一个数据块之前，它的值都将是`nil`。

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

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

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

重要的是，我们添加了一个模式匹配表达式来处理 AI 模型提供商发送的错误消息。在本地开发环境中，我们会抛出异常，但在生产环境中，我们会记录错误并结束处理。

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

case 语句的最终 else 子句会在之前的所有模式都不匹配时执行。这只是一个保障措施，这样如果人工智能模型开始发送我们无法识别的数据块时，我们就能够发现。

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

send_to_client 方法负责将已缓冲的内容发送给客户端。它会检查缓冲区是否为空，更新机器人消息内容，渲染机器人消息，并将内容保存在数据库中以确保数据持久性。

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

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

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

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

当流处理完成时，会调用finalize方法。如果在流期间收到任何函数调用，该方法会执行这些调用，使用最终内容和其他相关信息更新机器人消息，并重置函数调用历史记录

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

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

如果模型决定调用一个函数，你需要“分发“该函数调用（函数名称和参数），以确保它被执行，并且将`function_call`和`function_result`消息添加到对话记录中。

根据我的经验，最好在代码库的某个集中位置处理函数消息的创建，而不是依赖于各个工具的实现。这不仅使代码更整洁，还有一个非常重要的实际原因：如果 AI 模型调用了一个函数，但在你循环时在记录中没有看到相应的调用和结果消息，它就会再次调用相同的函数。这可能会无限循环下去。请记住，AI 是完全无状态的，所以除非你将这些函数调用的信息反馈给它，否则在它看来这些调用就像从未发生过。

```ruby
# PromptSubscriber#dispatch

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

在调度之后清除函数调用历史与确保调用和结果被记录在转录中同样重要，这样你就不会在每次循环时重复调用相同的函数。

“对话循环”

我一直在提到循环，但如果你是函数调用的新手，可能不太明显为什么我们需要循环。原因是一旦 AI “请求 “你代表它执行工具函数，它就会停止回复。接下来就需要你来执行这些函数、收集结果、将结果添加到转录中，然后重新提交原始提示，以获取新的函数调用或面向用户的结果。

在 `PromptSubscriber` 类中，我们使用来自 `PromptDeclarations` 模块的 `prompt` 方法来定义对话循环的行为。`until` 参数被设置为 `-> { bot_message.complete? }`，这意味着循环将持续进行直到 `bot_message` 被标记为完成。

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

但是 bot_message 什么时候被标记为完成呢？如果你忘记了，请回看 finalize 方法的第 13 行。

让我们回顾一下整个流处理逻辑。

1. PromptSubscriber 通过 message_created 方法接收新的用户消息，该方法在最终用户创建新提示时由 Wisper 发布/订阅系统调用。
2. prompt 类方法以声明方式定义了 PromptSubscriber 的聊天补全逻辑。AI 模型将使用用户的消息内容执行聊天补全，将新的 ReplyStream 实例作为流参数，并使用指定的循环条件。
3. AI 模型处理提示并开始生成响应。当响应以流的形式传输时，ReplyStream 实例的 call 方法会针对每个数据块被调用。
4. 如果 AI 模型决定调用工具函数，则从数据块中提取函数名称和参数，并分别存储在 f_name 和 f_arguments 数组中。
5. 如果 AI 模型生成面向用户的内容，这些内容会被缓存并通过 send_to_client 方法发送给客户端。
6. 一旦流处理完成，finalize 方法就会被调用。如果在流期间调用了任何工具函数，它们会使用 PromptSubscriber 的 dispatch 方法进行分发。
7. dispatch 方法会向对话记录添加一条 function_call 消息，执行相应的工具函数，并将函数调用的结果作为 function_result 消息添加到记录中。
8. 分发工具函数后，函数调用历史记录会被清除，以防止在后续循环中重复调用函数。
9. 如果没有调用工具函数，finalize 方法会用最终内容更新 bot_message，将其标记为完成，并将更新后的消息发送给客户端。

10. 循环条件 `-> { bot_message.complete? }` 被评估。如果 `bot_message` 未被标记为完成，循环将继续，原始提示将与更新后的对话记录一起重新提交。
11. 步骤 3-10 会重复进行，直到 `bot_message` 被标记为完成，表明 AI 模型已完成生成响应，且不需要执行更多的工具函数。

通过实现这个对话循环，你使 AI 模型能够与应用程序进行来回交互，根据需要执行工具函数，并生成面向用户的响应，直到对话自然结束。

流处理和对话循环的结合实现了动态和交互式的 AI 驱动体验，其中 AI 模型可以处理用户输入，利用各种工具和函数，并根据不断发展的对话上下文提供实时响应。

自动继续

重要的是要了解 AI 输出的限制。大多数模型在单个响应中可以生成的最大令牌数是由 `max_tokens` 参数决定的。如果 AI 模型在生成响应时达到这个限制，它将突然停止并指示输出被截断。

在 AI 平台 API 的流式响应中，你可以通过检查数据块中的 `finish_reason` 变量来检测这种情况。如果 `finish_reason` 被设置为 `"length"`（或特定于模型的其他键值），这意味着模型在生成过程中达到了其最大令牌限制，输出被提前截断。

处理这种情况的一种优雅方式，同时提供流畅的用户体验，就是在流处理逻辑中实现自动继续机制。通过为与长度相关的完成原因添加模式匹配，你可以选择循环并从中断处自动继续输出。

这里是一个特意简化的例子，展示如何修改 `ReplyStream` 类中的 `call` 方法以支持自动继续：

```
LENGTH_STOPS = %w[length MAX_TOKENS]

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

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

      buffer << new_content.to_s

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

    # ...
  end
end

private

def continue_cutoff
  conversation.bot_message!(buffer.join, visible: false)
```

```
30    conversation.user_message!("please continue", visible: false)
31    bot_message.update_column(:created_at, Time.current)
32  end
```

在这个修改后的版本中，当finish_reason 表明输出被截断时，我们不会终止流处理，而是在不结束对话的情况下向对话记录添加一对消息，通过更新其created_at 属性将原始的面向用户的响应消息移到对话记录的 “底部 “，然后让循环继续进行，这样 AI 就可以从中断处继续生成内容。

请记住，AI 补全端点是无状态的。它只 “知道 “你通过对话记录告诉它的内容。在这种情况下，我们通过向对话记录添加（对最终用户）“不可见 “的消息来告知 AI 它被截断了。但请记住，这只是一个特意简化的示例。真实的实现需要进行更多的对话记录管理，以确保我们不会浪费令牌和/或因对话记录中重复的助手消息而使 AI 混淆。

自动继续功能的真实实现还应该包含所谓的 “断路器逻辑“，以防止无限循环。原因是，在某些类型的用户提示和较低的max_tokens 设置下，AI 可能会无休止地循环生成面向用户的输出。

> 请记住，每次循环都需要单独的请求，而且每个请求都会重新消耗整个对话记录。在决定是否在应用程序中实现自动继续功能时，你应该认真考虑用户体验和 API 使用量之间的权衡。自动继续功能特别容易产生危险的高额费用，尤其是在使用高级商业模型时。

结论

流处理是构建将工具使用与实时 AI 响应相结合的 AI 驱动应用程序的关键方面。通过高效处理来自 AI 平台 API 的流数据，你可以提供无缝的交互式用户体验，处理大型响应，优化资源使用，并优雅地处理错误。

提供的`Conversation::ReplyStream`类演示了如何在 Ruby 应用程序中使用模式匹配和事件驱动架构实现流处理。通过理解和利用流处理技术，你可以充分发挥 AI 集成在应用程序中的潜力，提供强大而引人入胜的用户体验。

自修复数据

自修复数据是一种强大的方法，通过利用大语言模型（LLMs）的能力来确保应用程序中的数据完整性、一致性和质量。这类模式的重点是使用 AI 自动检测、诊断和纠正数据异常、不一致或错误，从而减轻开发人员的负担并维持高水平的数据可靠性。

从本质上讲，自修复数据模式认识到数据是任何应用程序的命脉，确保其准确性和完整性对于应用程序的正常运行和用户体验至关重要。然而，随着应用程序规模和复杂性的增长，管理和维护数据质量可能是一项复杂且耗时的任务。这就是 AI 发挥作用的地方。

在自修复数据模式中，AI 工作器被用来持续监控和分析应用程序的数据。这些模型能够理解和解释数据中的模式、关系和异常。通过利用其自然语言处理和理解能力，它们可以识别数据中的潜在问题或不一致，并采取适当的措施来纠正它们。

自修复数据的过程通常包括几个关键步骤：

1. **数据监控：** AI 工作器持续监控应用程序的数据流、数据库或存储系统，寻找任何异常、不一致或错误的迹象。或者，您可以在出现异常时激活 AI 组件。
2. **异常检测：**当发现问题时，AI 工作器详细分析数据以确定问题的具体性质和范围。这可能包括检测缺失值、不一致的格式或违反预定规则或约束的数据。
3. **诊断和纠正：**一旦确定了问题，AI 工作器使用其对数据领域的知识和理解来决定适当的行动方案。这可能包括自动纠正数据、填补缺失值，或在必要时标记问题以供人工干预。
4. **持续学习（可选，取决于用例）：**当 AI 工作器遇到并解决各种数据问题时，它可以输出描述发生了什么以及如何响应的信息。这些元数据可以被输入到学习过程中，使您（可能还有通过微调的底层模型）能够随着时间的推移在识别和解决数据异常方面变得更加有效和高效。

通过自动检测和纠正数据问题，您可以确保应用程序运行在高质量、可靠的数据基础上。这减少了错误、不一致或与数据相关的错误影响应用程序功能或用户体验的风险。

一旦有了 AI 工作器处理数据监控和纠正的任务，您就可以将精力集中在应用程序的其他关键方面。这节省了原本用于手动数据清理和维护的时间和资源。事实上，随着应用程序规模和复杂性的增长，手动管理数据质量变得越来越具有挑战性。“自修复数据 “模式通过利用 AI 的力量来处理大量数据并实时检测问题，从而实现了有效的扩展。

由于其特性，AI 模型可以随着时间的推移适应不断变化的数据模式、架构或要求，几乎不需要监督。只要它们的指令提供足够的指导，特别是关于预期结果，您的应用程序就可以在不需要大量手动干预或代码更改的情况下发展并处理新的数据场景。

自修复数据模式与我们讨论过的其他模式类别（如 “多重工作器“）很好地协调一致。自修复数据功能可以被视为一种专门专注于确保数据质量和完整性的特殊工作器。这种工作器与其他 AI 工作器一起运作，各自为应用程序的不同功能方面做出贡献。

在实践中实施自修复数据模式需要仔细设计并将 AI 模型集成到应用程序架构中。由于数据丢失和损坏的风险，您应该为如何使用这种技术制定明确的指导方针。您还应该考虑性能、可扩展性和数据安全性等因素。

实践案例研究：修复损坏的 JSON

利用自修复数据最实用和便捷的方法之一也很容易解释：修复损坏的 JSON。

这种技术可以应用于处理由 LLMs生成的不完整或不一致数据（如损坏的 JSON）这一常见挑战，并提供了自动检测和纠正这些问题的方法。

在 Olympia 中，我经常遇到 LLM 生成的 JSON 数据不完全有效的情况。这种情况可能由多种原因导致，比如 LLM 在实际 JSON 代码前后添加了注释，或者引入了语法错误，如缺少逗号或未转义的双引号。这些问题可能导致解析错误，从而影响应用程序的功能。

为了解决这个问题，我实现了一个实用的解决方案，即 JsonFixer 类。这个类体现了 “自我修复数据 “模式，它接收损坏的 JSON 作为输入，并利用 LLM 来修复它，同时尽可能保留原有的信息和意图。

```ruby
class JsonFixer
  include Raix::ChatCompletion

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

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

    self.stop = ["```"]

    chat_completion(json: true)
  end

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

注意 `JsonFixer` 如何使用 Ventriloquist 来引导 AI 的响应。

JSON 数据的自修复过程如下：

1. **JSON 生成：** 使用大语言模型基于特定提示或要求生成 JSON 数据。然而，由于大语言模型的特性，生成的 JSON 可能并不总是完全有效的。如果您向 JSON 解析器提供无效的 JSON，它自然会抛出 `ParserError`。

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

注意异常消息也被传递给`JSONFixer` 调用，这样它就不需要完全假设数据出了什么问题，特别是因为解析器通常会准确告诉你哪里出错了。

2. **基于大语言模型的修正：** `JSONFixer` 类将损坏的 JSON 连同特定的提示或指令一起发送给大语言模型，要求在尽可能保留原始信息和意图的同时修复 JSON。经过海量数据训练并具备 JSON 语法理解能力的大语言模型会尝试纠正错误并生成有效的 JSON 字符串。这里使用了响应围栏来限制大语言模型的输出，我们选择 Mixtral 8x7B 作为 AI 模型，因为它特别适合这类任务。
3. **验证和集成：** 由于调用了`chat_completion(json: true)`，`JSONFixer` 类本身会解析大语言模型返回的修复后的 JSON 字符串。如果修复后的 JSON 通过验证，它就会被整合回应用程序的工作流程中，使应用程序能够继续无缝处理数据。这样，损坏的 JSON 就被“修复“了。

虽然我已经多次编写和重写我自己的`JSONFixer` 实现，但我怀疑所有这些版本的总投入时间不会超过一两个小时。

注意，保持意图是任何自修复数据模式的关键要素。基于大语言模型的修正过程旨在尽可能保留生成的 JSON 的原始信息和意图。这确保了修复后的 JSON 保持其语义含义，并能在应用程序的上下文中有效使用。

Olympia 中 “自修复数据 “方法的这种实践实现清楚地展示了如何利用 AI（特别是大语言模型）来解决现实世界的数据挑战。它展示了将传统编程技术与 AI 功能相结合来构建健壮高效应用程序的强大力量。

波斯特法则与 “自修复数据 “模式

“自修复数据”（如 JSONFixer 类所展示的）与波斯特法则（也称为健壮性原则）高度契合。波斯特法则指出：

“对于自己要做的事要保守，对于接受他人的东西要自由。”

这一原则最初由互联网早期先驱 Jon Postel 提出，强调了构建系统时要能容忍多样化甚至略有错误的输入，同时在发送输出时要严格遵守指定协议的重要性。

在 “自修复数据 “的背景下，JSONFixer 类体现了波斯特法则，它在接受大语言模型生成的损坏或不完善的 JSON 数据时采取开放态度。当遇到不严格符合预期格式的 JSON 时，它不会立即拒绝或失败。相反，它采取包容的方法，尝试使用大语言模型的能力来修复 JSON。

通过自由接受不完善的 JSON，JSONFixer 类展示了其健壮性和灵活性。它承认现实世界中的数据经常以各种形式出现，可能并不总是符合严格的规范。通过优雅地处理和纠正这些偏差，该类确保应用程序即使在面对不完善的数据时也能继续平稳运行。

另一方面，JSONFixer 类在输出方面也遵守了波斯特法则的保守原则。在使用大语言模型修复 JSON 后，该类会验证修正后的 JSON 以确保它严格符合预期格式。在将数据传递给应用程序的其他部分之前，它保持数据的完整性和正确性。这种保守方法保证了 JSONFixer 类的输出是可靠和一致的，促进了互操作性并防止错误的传播。

关于 Jon Postel 的有趣趣事：

- Jon Postel（1943-1998）是一位美国计算机科学家，在互联网发展中发挥了关键作用。由于他对底层协议和标准的重大贡献，他被称为 “互联网之神 “。
- Postel 是请求评议标准（RFC）文档系列的编辑，这是一系列关于互联网的技术和组织说明。他撰写或合著了超过 200 份 RFC，包括 TCP、IP 和 SMTP 等基础协议。
- 除了技术贡献外，Postel 以其谦逊和协作的方式而闻名。他相信达成共识和共同努力建设健壮且可互操作网络的重要性。
- Postel 从 1977 年起担任南加州大学（USC）信息科学研究所（ISI）计算机网络部门主任，直到 1998 年不幸去世。
- 为表彰他的巨大贡献，Postel 在 1998 年获得了享有盛誉的图灵奖（通常被称为 “计算机界的诺贝尔奖 “）。

`JSONFixer` 类推广了健壮性、灵活性和互操作性，这些都是 Postel 在其职业生涯中始终坚持的核心价值。通过构建能够容忍缺陷同时又严格遵守协议的系统，我们可以创建出在面对现实世界挑战时更具韧性和适应性的应用程序。

考虑因素和禁忌症

自修复数据方法的适用性完全取决于您的应用程序处理的数据类型。有充分的理由说明为什么您可能不想简单地通过猴子补丁方式修改`JSON.parse` 来自动修正应用程序中的所有 JSON 解析错误：并非所有错误都能够或应该被自动修正。

当涉及到数据处理相关的监管或合规要求时，自修复特别令人担忧。某些行业，如医疗保健和金融，对数据完整性和可审计性有着严格的规定，在没有适当监督或日志记录的情况下进行任何形式的 “黑盒 “数据修正可能会违反这些规定。确保您制定的任何自修复数据技术都符合适用的法律和监管框架至关重要。

应用自修复数据技术，特别是那些涉及 AI 模型的技术，可能会对应用程序的性能和资源利用产生重大影响。通过 AI 模型处理大量数据以进行错误检测和修正

可能会消耗大量计算资源。评估自修复数据的好处与相关的性能和资源成本之间的权衡很重要。

话虽如此，让我们深入探讨决定何时何地应用这种强大方法所涉及的因素。

数据关键性

在考虑应用自修复数据技术时，评估所处理数据的关键性至关重要。关键性水平指的是数据在您的应用程序和业务领域中的重要性和敏感性。

在某些情况下，自动修正数据错误可能并不合适，特别是当数据高度敏感或具有法律影响时。例如，考虑以下场景：

1. ** 金融交易：** 在金融应用程序中，如银行系统或交易平台，数据准确性至关重要。金融数据中即使微小的错误也可能产生重大后果，如账户余额不正确、资金误导或错误的交易决策。在这些情况下，在没有彻底验证和审计的情况下进行自动修正可能会带来不可接受的风险。
2. ** 医疗记录：** 医疗保健应用程序处理高度敏感和机密的患者数据。医疗记录中的不准确性可能对患者安全和治疗决策产生严重影响。在没有合格医疗专业人员适当监督和验证的情况下自动修改医疗数据可能违反监管要求并危及患者福祉。
3. ** 法律文档：** 处理法律文档的应用程序，如合同、协议或法院文件，需要严格的准确性和完整性。法律数据中即使微小的错误也可能产生重大的法律后果。在这个领域进行自动修正可能并不合适，因为数据通常需要法律专家进行人工审查和验证以确保其有效性和可执行性。

在这些关键数据场景中，自动修正相关的风险通常超过潜在的收益。引入错误或不正确修改数据的后果可能非常严重，导致财务损失、法律责任，甚至对个人造成伤害。

在处理高度关键的数据时，优先考虑人工验证和确认过程至关重要。人工监督和专业知识对确保数据的准确性和完整性至关重要。自动自修复技术仍然可以

用于标记潜在的错误或不一致，但对于修正的最终决定应该涉及人工判断和批准。

然而，需要注意的是，应用程序中并非所有数据都具有相同的关键性水平。在同一个应用程序中，可能存在敏感度较低或错误影响较小的数据子集。在这种情况下，自修复数据技术可以选择性地应用于这些特定的数据子集，而关键数据仍然需要进行人工验证。

关键是要仔细评估应用程序中每个数据类别的关键性，并根据相关风险和影响制定明确的指导方针和处理流程。通过区分关键数据（如账本、医疗记录）和非关键数据（如邮寄地址、资源警告），您可以在适当利用自修复数据技术的好处和在必要时维持严格控制和监督之间取得平衡。

最终，是否对关键数据应用自修复数据技术的决定应该与领域专家、法律顾问和其他相关利益相关者协商后做出。考虑应用程序数据的具体要求、法规和风险，并相应地调整数据修正策略至关重要。

错误严重程度

在应用自修复数据技术时，评估数据错误的严重程度和影响很重要。并非所有错误都是同等的，适当的处理方式可能会根据问题的严重程度而有所不同。

轻微的不一致或格式问题可能适合自动修正。例如，负责修复损坏 JSON 的自修复数据工作器可以处理缺失的逗号或未转义的双引号，而不会显著改变数据的含义或结构。这些类型的错误通常容易修正，对整体数据完整性的影响最小。

然而，对于那些从根本上改变数据含义或完整性的更严重错误，可能需要采用不同的处理方法。在这种情况下，自动校正可能不够充分，需要人工干预来确保数据的准确性和有效性。

这就引出了利用 AI 本身来帮助判断错误严重程度的概念。通过利用 AI 模型的能力，我们可以设计自修复数据工作器，它不仅可以纠正错误，还能评估这些错误的严重程度，并就如何处理这些错误做出明智的决策。

例如，让我们考虑一个负责纠正流入客户数据库的数据不一致性的自修复数据工作器。该工作器可以被设计用来分析数据并识别潜在的错误，如缺失或冲突的信息。但是，工作器不会自动修正所有错误，而是可以配备额外的工具调用，使其能够将严重错误标记出来供人工审查。

下面是如何实现这一点的示例：

```
class CustomerDataReviewer
  include Raix::ChatCompletion
  include Raix::FunctionDeclarations

  attr_accessor :customer

  function :flag_for_review, reason: { type: "string" } do |params|
    AdminNotifier.review_request(customer, params[:reason])
  end

  def initialize(customer)
    self.customer = customer
  end

  def call(customer_data)
    transcript << {
      system: "You are a customer data reviewer. Your task is to identify
        and correct inconsistencies in customer data.

        < additional instructions here... >

        If you encounter severe errors that require human review, use the
        `flag_for_review` tool to flag the data for manual intervention." }
```

```
24
25      transcript << { user: customer.to_json }
26      transcript << { assistant: "Reviewed/corrected data:\n```json\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
```

在这个例子中，`CustomerDataHealer` 工作器被设计用来识别和纠正客户数据中的不一致性。我们再次使用响应围栏和腹语术来获取结构化输出。重要的是，工作器的系统指令包含了在遇到严重错误时使用`flag_for_review` 函数的说明。

当工作器处理客户数据时，它会分析数据并尝试纠正任何不一致性。如果工作器判定错误严重并需要人工干预，它可以使用`flag_for_review` 工具来标记数据，并提供标记的原因。

`chat_completion` 方法使用`json: true` 参数来将修正后的客户数据解析为 JSON。由于函数调用后没有循环处理的规定，如果调用了`flag_for_review`，结果将为空。否则，客户数据将使用经过审查和可能已修正的数据进行更新。

通过整合错误严重性评估和标记数据供人工审查的选项，自修复数据工作器变得更加智能和适应性强。它可以自动处理轻微错误，同时将严重错误升级给人类专家进行手动干预。

确定错误严重性的具体标准可以根据领域知识和业务需求在工作器的指令中定义。在评估严重性时，可以考虑诸如对数据完整性的影响、数据丢失或损坏的可能性以及不正确数据造成的后果等因素。

通过利用 AI 来评估错误严重性并提供人工干预的选项，自修复数据技术可以在自动化和维护数据准确性之间取得平衡。这种方法确保了轻微错误能够被高效修正，而严重错误则能获得人工审查者必要的关注和专业处理。

领域复杂性

在考虑应用自修复数据技术时，评估数据领域的复杂性以及支配其结构和关系的规则非常重要。领域的复杂性可能会显著影响自动数据修正方法的有效性和可行性。

当数据遵循明确定义的模式和约束时，自修复数据技术效果很好。在数据结构相对简单且数据元素之间关系直观的领域中，自动修正可以以高度的置信度进行应用。例如，自修复数据工作器通常可以有效处理格式问题或强制执行基本数据类型约束。

然而，随着数据领域复杂性的增加，与自动数据修正相关的挑战也随之增长。在具有复杂业务逻辑、数据实体之间复杂关系或特定领域规则和例外的领域中，自修复数据技术可能无法始终捕捉到这些细微差别，并可能引入意外后果。

让我们考虑一个复杂领域的例子：金融交易系统。在这个领域中，数据涉及各种金融工具、市场数据、交易规则和监管要求。不同数据元素之间的关系可能非常复杂，而支配数据有效性和一致性的规则可能高度特定于该领域。

在这样一个复杂的领域中，负责修正交易数据不一致性的自修复数据工作器需要深入理解领域特定的规则和约束。它需要考虑市场法规、交易限制、风险计算和结算程序等因素。在这种情况下，自动修正可能无法始终把握领域的全部复杂性，并可能无意中引入错误或违反领域特定规则。

为了应对领域复杂性的挑战，可以通过以下方式增强自修复数据技术，将领域特定知识和规则纳入 AI 模型和工作器：

1. ** 领域特定训练：** 用于自修复数据的 AI 模型可以在捕捉特定领域的复杂性和规则的领域特定数据集上进行引导甚至微调。通过接触具有代表性的数据和场景，它们可以学习该领域特有的模式、约束和例外情况。

2. ** 基于规则的约束：** 自修复数据工作器可以通过明确的基于规则的约束来增强，这些约束编码了领域特定知识。这些规则可以由领域专家定义并集成到数据修正过程中。AI 模型随后可以使用这些规则来指导其决策并确保符合领域特定要求。
3. ** 与领域专家协作：** 在复杂领域中，在设计和开发自修复数据技术时让领域专家参与进来至关重要。领域专家可以为数据的复杂性、业务规则和潜在边缘情况提供宝贵见解。他们的知识可以使用人在环中模式整合到 AI 模型和工作器中，以提高自动数据修正的准确性和可靠性。
4. ** 增量和迭代方法：** 在处理复杂领域时，采用增量和迭代的方法进行数据自修复通常是有益的。不要试图一次性自动修正整个领域，而是关注规则和约束明确的特定子领域或数据类别。随着对领域理解的加深和技术的证明有效，逐步扩大自修复技术的范围。

通过考虑数据领域的复杂性并将领域特定知识融入自修复数据技术中，你可以在自动化和准确性之间取得平衡。重要的是要认识到自修复数据并非放之四海而皆准的解决方案，这种方法应该根据每个领域的具体要求和挑战量身定制。

在复杂的领域中，将自修复数据技术与人类专业知识和监督相结合的混合方法可能最为有效。自动化修正可以处理常规和明确定义的情况，而复杂场景或异常情况则可以标记出来供人工审查和干预。这种协作方法确保了实现自动化的好处，同时在复杂的数据领域中保持必要的控制和准确性。

可解释性和透明度

可解释性指的是理解和解释 AI 模型背后决策理由的能力，而透明度则涉及为数据修正过程提供清晰的可见性。

在许多情况下，数据修改需要可审计和合理解释。包括业务用户、审计师和监管机构在内的利益相关者可能需要解释为什么进行某些数据修正，以及 AI 模型是如何得出这些决定的。这在数据准确性和完整性具有重要影响的领域中尤为重要，比如金融、医疗保健和法律事务。

为了满足可解释性和透明度的需求，自修复数据技术应该包含能够提供 AI 模型决策过程洞察的机制。这可以通过多种方法实现：

1. **思维链：**在对数据进行更改之前，要求模型 “大声 “解释其思考过程，这可能有助于更容易理解决策过程，并能为所做的修正生成人类可读的解释。权衡之处在于在将解释与结构化数据输出分离时会增加一些复杂性，这可以通过...来解决
2. **解释生成：**自修复数据工作器可以具备为其所做修正生成人类可读解释的能力。这可以通过要求模型将其决策过程作为易于理解的解释_整合到数据本身_中来实现。例如，自修复数据工作器可以生成一份报告，突出显示它识别出的具体数据不一致性、应用的修正以及这些修正背后的理由。
3. **特征重要性：** AI 模型可以在其指令中被告知数据修正过程中不同特征或属性的重要性。这些指令反过来可以向人类利益相关者公开。通过识别影响模型决策的关键因素，利益相关者可以深入了解修正背后的理由并评估其有效性。
4. **日志记录和审计：**实施全面的日志记录和审计机制对于维护自修复数据过程的透明度至关重要。AI 模型进行的每一次数据修正都应该被记录下来，包括原始数据、修正后的数据以及采取的具体操作。这种审计跟踪允许进行回顾性分析，并为数据修改提供清晰的记录。
5. **人在环中方法：**采用人在环中方法可以增强自修复数据技术的可解释性和透明度。通过让人类专家参与 AI 生成的修正的审查和验证，组织可以确保修正符合领域知识和业务需求。人工监督增加了一层问责制，并允许识别 AI 模型中可能存在的任何偏差或错误。
6. **持续监控和评估：**定期监控和评估自修复数据技术的性能对于维护透明度和信任至关重要。通过随时间评估 AI 模型的准确性和有效性，组织可以识别任何偏差或异常并采取纠正措施。持续监控有助于确保自修复数据过程保持可靠性并与预期结果保持一致。

可解释性和透明度是实施自修复数据技术时的关键考虑因素。通过为数据修正提供清晰的解释、维护全面的审计跟踪并涉及人工监督，组织可以建立对自修

复数据过程的信任，并确保对数据的修改是合理的且与业务目标一致。

在自动化的好处和透明度需求之间取得平衡很重要。虽然自修复数据技术可以显著提高数据质量和效率，但这不应以牺牲数据修正过程的可见性和控制为代价。通过在设计自修复数据工作器时考虑可解释性和透明度，组织可以利用 AI 的力量，同时保持必要的问责制和对数据的信任。

意外后果

虽然自修复数据技术旨在提高数据质量和一致性，但意识到潜在的意外后果至关重要。如果没有仔细设计和监控，自动化修正可能会无意中改变数据的含义或上下文，导致下游问题。

自修复数据的主要风险之一是在数据修正过程中引入偏差或错误。AI 模型，就像任何其他软件系统一样，可能会受到训练数据中存在的偏差或通过算法设计引入的偏差的影响。如果这些偏差没有被识别和缓解，它们可能会在自修复数据过程中传播，导致偏斜或不正确的数据修改。

例如，考虑一个负责纠正客户人口统计数据中的不一致性的自修复数据工作器。如果 AI 模型从历史数据中学习到了偏见，比如将某些职业或收入水平与特定性别或种族关联起来，它可能会做出错误的假设，并以强化这些偏见的方式修改数据。这可能导致不准确的客户画像、错误的商业决策，以及潜在的歧视性结果。

另一个潜在的意外后果是在数据纠正过程中丢失有价值的信息或上下文。自修复数据技术通常注重于标准化和规范化数据以确保一致性。然而，在某些情况下，原始数据可能包含对于理解完整情况很重要的细微差别、例外情况或上下文信息。盲目执行标准化的自动纠正可能会无意中删除或模糊这些有价值的信息。

例如，设想一个负责纠正医疗记录中不一致性的自修复数据工作器。如果工作器遇到一个患者的病史中有罕见病症或不寻常的治疗方案，它可能会试图将数据规范化以符合更常见的模式。然而，在这样做的过程中，它可能会丢失那些

对准确表述患者独特情况至关重要的具体细节和上下文。这种信息的丢失可能会对患者护理和医疗决策产生严重影响。

为了降低意外后果的风险，在设计和实施自修复数据技术时采取主动方法至关重要：

1. **全面测试和验证：**在将自修复数据工作器部署到生产环境之前，必须针对各种场景彻底测试和验证其行为。这包括使用涵盖各种边缘情况、例外情况和潜在偏差的代表性数据集进行测试。严格的测试有助于在影响实际数据之前识别和解决任何意外后果。
2. **持续监控和评估：**实施持续监控和评估机制对于及时发现和缓解意外后果至关重要。定期审查自修复数据处理的结果，分析对下游系统和决策制定的影响，以及收集利益相关者的反馈，可以帮助识别任何不利影响并及时采取纠正措施。如果您的组织有运营仪表板，添加与自动数据更改相关的清晰可见的指标可能是个好主意。添加与正常数据变更活动有大幅偏差时的告警可能是更好的主意！
3. **人工监督和干预：**保持人工监督和干预自修复数据过程的能力至关重要。虽然自动化可以大大提高效率，但让人类专家审查和验证 AI 模型所做的纠正很重要，特别是在关键或敏感领域。人类的判断和领域专业知识可以帮助识别和解决可能出现的任何意外后果。

4. **可解释人工智能（XAI）和透明度：**如前一小节所讨论的，引入可解释人工智能技术并确保自修复数据过程的透明度可以帮助缓解意外后果。通过为数据纠正提供清晰的解释并维护全面的审计跟踪，组织可以更好地理解和追踪 AI 模型进行修改的理由。
5. **渐进和迭代方法：**采用渐进和迭代的方法进行数据自修复可以帮助最小化意外后果的风险。不要一次性对整个数据集应用自动纠正，而是从数据子集开始，随着技术证明有效和可靠而逐步扩大范围。这允许在过程中进行仔细监控和调整，减少任何意外后果的影响。
6. **协作和反馈：**在整个自修复数据过程中让来自不同领域的利益相关者参与并鼓励协作和反馈，可以帮助识别和解决意外后果。定期寻求领域专家、

数据使用者和最终用户的意见可以提供关于数据纠正实际影响的宝贵见解，并突出可能被忽视的任何问题。

通过主动应对意外后果的风险并实施适当的保护措施，组织可以利用自修复数据技术的优势，同时最小化潜在的不利影响。重要的是要将自修复数据视为一个迭代和协作的过程，持续监控、评估和改进技术，以确保它们与预期结果保持一致，并维护数据的完整性和可靠性。

在考虑使用自修复数据模式时，必须仔细评估这些因素，并权衡其益处与潜在风险和局限性。在某些情况下，将自动纠正与人工监督和干预相结合的混合方法可能是最合适的解决方案。

同样值得注意的是，自修复数据技术不应被视为替代健全的数据验证、输入净化和错误处理机制。这些基础实践对确保数据完整性和安全性仍然至关重要。自修复数据应被视为一种补充方法，可以增强和改进这些现有措施。

最终，是否采用自修复数据模式取决于您的应用程序的具体要求、约束和优先级。通过仔细考虑上述因素，并将它们与您的应用程序目标和架构保持一致，您可以就何时以及如何有效利用自修复数据技术做出明智的决定。

上下文内容生成

上下文内容生成模式利用大语言模型（LLMs）的能力在应用程序中生成动态和具有上下文相关性的内容。这类模式认识到基于用户的特定需求、偏好，甚至是他们与应用程序之前和当前的交互来提供个性化和相关内容的重要性。

> 在这种方法中，“内容 “既指主要内容（如博客文章、文章等），也指元内容，比如对主要内容的推荐。

上下文内容生成模式在提升用户参与度、提供定制化体验以及为您和用户自动化内容创建任务方面可以发挥关键作用。通过使用本章描述的模式，您可以创建能够动态生成内容的应用程序，实时适应上下文和输入。

这些模式通过将 LLMs整合到应用程序的输出中来工作，范围涵盖从用户界面

（有时称为“界面元素“）到电子邮件和其他形式的通知，以及任何内容生成管道。

当用户与应用程序交互或触发特定内容请求时，应用程序会捕获相关上下文，如用户偏好、之前的交互或特定提示。这些上下文信息随后会连同任何必要的模板或指南一起输入到 LLM 中，用于生成原本需要硬编码、存储在数据库中或通过算法生成的文本输出。

LLM 生成的内容可以采取多种形式，如个性化推荐、动态产品描述、定制化邮件回复，甚至是完整的文章或博客文章。我在一年多前开创的最激进的用途之一是动态生成用户界面元素，如表单标签、工具提示和其他类型的解释性文本。

个性化

上下文内容生成模式的主要优势之一是能够为用户提供高度个性化的体验。通过基于用户特定上下文生成内容，这些模式使应用程序能够根据个别用户的兴趣、偏好和交互来定制内容。

个性化不仅仅是将用户名插入通用内容中。它涉及利用每个用户可用的丰富上下文来生成能够引起他们特定需求和愿望共鸣的内容。这种上下文可以包括广泛的因素，例如：

1. ** 用户档案信息：** 在应用这种技术的最一般层面上，人口统计数据、兴趣、偏好和其他档案属性可用于生成与用户背景和特征相符的内容。
2. ** 行为数据：** 用户与应用程序的过往交互，如浏览过的页面、点击过的链接或购买过的产品，可以提供关于其行为和兴趣的宝贵见解。这些数据可用于生成反映其参与模式并预测其未来需求的内容建议。
3. ** 上下文因素：** 用户当前的上下文，如位置、设备、一天中的时间，甚至天气，都可以影响内容生成过程。例如，旅行应用程序可能有一个 AI 工作器，能够基于用户当前位置和当前天气状况生成个性化推荐。

通过利用这些上下文因素，上下文内容生成模式使应用程序能够提供对每个用户来说都感觉量身定制的内容。这种级别的个性化具有几个显著优势：

1. ** 提高参与度：** 个性化内容能够吸引用户的注意力并保持他们与应用程序的互动。当用户感觉内容与其相关并直接针对其需求时，他们更有可能花更多时间与应用程序交互并探索其功能。
2. ** 提升用户满意度：** 个性化内容表明应用程序理解并关心用户的独特需求。通过提供有帮助、信息丰富且符合其兴趣的内容，应用程序可以提升用户满意度并与用户建立更强的连接。
3. ** 更高的转化率：** 在电子商务或营销应用程序的背景下，个性化内容可以显著影响转化率。通过向用户展示根据其偏好和行为定制的产品、优惠或推荐，应用程序可以增加用户采取期望行动的可能性，如进行购买或注册服务。

生产力

上下文内容生成模式可以通过减少创意过程中手动内容生成和编辑的需求来显著提升某些类型的
通过利用 LLMs 的能力，您可以大规模生成高质量内容，节省内容创作者和开发人员原本需要花在繁琐手工工作上的时间和精力。

传统上，内容创作者需要研究、写作、编辑和格式化内容，以确保其符合应用程序的要求和用户的期望。这个过程可能非常耗时且需要大量资源，尤其是随着内容量的增长。

然而，通过上下文内容生成模式，内容创作过程可以在很大程度上实现自动化。大语言模型可以根据提供的提示和指导生成连贯、语法正确且与上下文相关的内容。这种自动化带来了几个生产力方面的优势：

1. ** 减少人工工作：** 通过将内容生成任务委托给大语言模型，内容创作者可以专注于更高层次的任务，如内容策略、构思和质量保证。他们可以为大语言模型提供必要的上下文、模板和指导，让它来处理实际的内容生成。这减少了写作和编辑所需的人工努力，使内容创作者能够更高效地工作。

2. ** 更快的内容创作：** 大语言模型能够比人类作者更快地生成内容。有了适当的提示和指导，大语言模型可以在几秒或几分钟内生成多篇内容。这种速度使应用程序能够以更快的速度生成内容，跟上用户需求和不断变化的数字环境。

> 更快的内容创作是否正在导致一种的情况，使互联网被无人阅读的内容所淹没？遗憾的是，我怀疑答案是肯定的。

3. ** 一致性和质量：** 大语言模型可以轻松修改内容，使其在风格、语气和质量方面保持一致。在提供清晰的指导和示例的情况下，某些类型的应用（如新闻编辑室、公关等）可以确保其人工生成的内容与品牌声音保持一致，并满足所需的质量标准。这种一致性减少了大量编辑和修订的需求，节省了内容创作过程中的时间和精力。
4. ** 迭代和优化：**上下文内容生成模式支持内容的快速迭代和优化。通过调整提供给大语言模型的提示、模板或指导，您的应用程序可以快速生成内容的变体，并以过去从未可能的自动化方式测试不同的方法。这种迭代过程允许更快速地实验和改进内容策略，随时间推移产生更有效和更具吸引力的内容。这种特定技术对于依赖跳出率和参与度生存的电子商务应用程序来说可能是一个彻底的游

需要注意的是，虽然上下文内容生成模式可以大大提高生产力，但并不能完全消除人工参与的需求。内容创作者和编辑在定义整体内容策略、为大语言模型提供指导以及确保生成内容的质量和适当性方面仍然发挥着关键作用。

通过自动化内容创作中更具重复性和耗时的方面，上下文内容生成模式释放了宝贵的人力时间和资源，这些资源可以被重新引导到更高价值的任务中。这种

提高的生产力使您能够向用户提供更个性化和更具吸引力的内容，同时优化内容创作工作流程。

快速迭代和实验

上下文内容生成模式使您能够快速迭代和尝试不同的内容变体，从而更快地优化和改进您的内容策略。只需调整提供给模型的上下文、模板或指导，您就可以在几秒钟内生成多个版本的内容。

这种快速迭代能力提供了几个关键优势：

1. ** 测试和优化：** 通过快速生成内容变体的能力，您可以轻松测试不同的方法并衡量其效果。例如，您可以生成产品描述或营销信息的多个版本，每个版本都针对特定的用户群体或上下文进行定制。通过分析用户参与度指标，如点击率或转化率，您可以识别最有效的内容变体，并相应地优化您的内容策略。

2. **A/B 测试：** 上下文内容生成模式支持内容的无缝 A/B 测试。您可以生成两个或更多的内容变体，并随机向不同的用户组提供。通过比较每个变体的表现，您可以确定哪些内容最能引起目标受众的共鸣。这种数据驱动的方法允许您做出明智的决策，并持续改进内容以最大化用户参与度并实现您期望的结果。
3. ** 个性化实验：** 快速迭代和实验在个性化方面特别有价值。通过上下文内容生成模式，您可以根据不同的用户群体、偏好或行为快速生成个性化内容变体。通过尝试不同的个性化策略，您可以识别最有效的方法来吸引个别用户并提供量身定制的体验。
4. ** 适应不断变化的趋势：** 快速迭代和实验的能力使您能够保持敏捷，并适应不断变化的趋势和用户偏好。随着新主题、关键词或用户行为的出现，您可以快速生成与这些趋势相符的内容。通过持续实验和改进内容，您可以在不断发展的数字环境中保持相关性和竞争优势。

5. ** 成本效益实验：** 传统的内容实验通常需要大量时间和资源，因为内容创作者需要手动开发和测试不同的变体。然而，通过上下文内容生成模式，实验成本大大降低。大语言模型可以快速大规模地生成内容变体，让您无需承担巨大成本就能探索广泛的想法和方法。

为了充分利用快速迭代和实验，建立一个明确定义的实验框架很重要。这个框架应该包括：

- 每个实验的明确目标和假设
- 适当的指标和跟踪机制来衡量内容表现
- 细分和定位策略，确保向正确的用户提供相关的内容变体
- 用于从实验数据中获取见解的分析和报告工具
- 将学习成果和优化整合到内容策略中的流程

通过采用快速迭代和实验，您可以持续改进和优化内容，确保内容保持吸引力、相关性，并有效实现应用程序的目标。这种敏捷的内容创作方法使您能够保持领先地位，提供卓越的用户体验。

可扩展性和效率

随着应用程序的增长和个性化内容需求的增加，上下文内容生成模式能够实现内容创作的高效扩展。大语言模型可以同时为大量用户和上下文生成内容，而无需相应增加人力资源。这种可扩展性使应用程序能够在不影响其内容创作能力的情况下，为不断增长的用户群提供个性化体验。

请注意，上下文内容生成可以有效地用于“即时“国际化您的应用程序。事实上，这正是我使用 Instant18n Gem 为 Olympia 提供超过半打语言版本的方式，尽管我们成立还不到一年。

AI 驱动的本地化

如果允许我自夸一下，我认为我开发的 Rails 应用程序的 Instant18n 库是 “上下文内容生成 “模式的一个开创性示例，展示了 AI 在应用程序开发中的变革潜力。这个 gem 利用 OpenAI 的 GPT 大语言模型的力量，彻底改变了 Rails 应用程序中处理国际化和本地化的方式。

传统上，国际化 Rails 应用程序需要手动定义翻译键并为每种支持的语言提供相应的翻译。这个过程可能耗时、资源密集且容易出现不一致。然而，使用 Instant18n gem，本地化的范式被完全重新定义。

通过集成大语言模型，Instant18n gem 使您能够根据文本的上下文和含义即时生成翻译。该 gem 不依赖预定义的翻译键和静态翻译，而是使用 AI 的力量动态翻译文本。这种方法提供了几个关键优势：

1. ** 无缝本地化：** 使用 Instant18n gem，开发人员不再需要为每种支持的语言手动定义和维护翻译文件。该 gem 根据提供的文本和目标语言自动生成翻译，使本地化过程变得轻松和无缝。
2. ** 上下文准确性：**AI 可以获得足够的上下文来理解被翻译文本的细微差别。它可以考虑周围的上下文、习语和文化参考，生成准确、自然且符合上下文的翻译。
3. ** 广泛的语言支持：**Instant18n gem 利用 GPT 的广泛知识和语言能力，实现对大量语言的翻译。从西班牙语和法语等常用语言到克林贡语和精灵语等更罕见或虚构的语言，该 gem 都能处理各种翻译需求。
4. ** 灵活性和创造力：** 该 gem 超越了传统的语言翻译，允许创造性和非常规的本地化选项。开发人员可以将文本翻译成各种风格、方言，甚至虚构语言，为独特的用户体验和引人入胜的内容开创新的可能性。
5. ** 性能优化：**Instant18n gem 整合了缓存机制来提高性能并减少重复翻译的开销。翻译后的文本会被缓存，使得后续相同翻译的请求能够快速得到响应，无需重复 API 调用。

Instant18n gem 通过利用 AI 动态生成本地化内容，展示了 “上下文内容生成 “模式的强大功能。它展示了如何将 AI 集成到 Rails 应用程序的核心功能中，彻底改变开发人员处理国际化和本地化的方式。

通过消除手动翻译管理的需求，并实现基于上下文的即时翻译，Instant18n gem 为开发人员节省了大量的时间和精力。它使开发人员能够专注于构建应用程序的核心功能，同时确保本地化过程得到无缝且准确的处理。

用户测试和反馈的重要性

最后，始终要牢记用户测试和反馈的重要性。 验证上下文内容生成是否符合用户期望并与应用程序目标保持一致至关重要。基于用户见解和分析数据持续迭代和改进生成的内容。如果你正在大规模生成动态内容，而这些内容不可能由你和你的团队手动验证，请考虑添加反馈机制，允许用户报告奇怪或错误的内容，并解释原因。这些宝贵的反馈甚至可以提供给负责调整生成内容的组件的 AI 工作程序！

生成式用户界面

在当今时代，注意力是如此珍贵，要实现有效的用户参与，不仅需要软件体验做到无缝和直观,还必须高度个性化以适应个人需求、偏好和具体场景。因此，设计师和开发人员越来越多地面临着创建能够适应每个用户独特需求的用户界面的挑战，而且要实现规模化。

生成式用户界面（GenUI）是用户界面设计的一次真正革命性突破，它利用大型语言模型（LLMs）的能力，即时创建高度个性化和动态的用户体验。我想在本书中至少为您介绍 GenUI 的基础知识，因为我认为这是目前应用程序设计和框架领域中最具发展潜力的机会之一。我确信在这个特定领域将会涌现出数十个或更多成功的商业和开源项目。

从本质上讲，GenUI 将上下文内容生成的原则与先进的人工智能技术相结合，基于对用户场景、偏好和目标的深入理解，动态生成用户界面元素，如文本、图像和布局。GenUI 使设计师和开发人员能够创建能够响应用户交互并不断演进的界面，提供此前无法实现的个性化程度。

GenUI 代表着我们在用户界面设计方法上的根本性转变。它让我们从面向大众的设计转向为个人定制的设计。个性化的内容和界面有潜力创造出能与每个用户产生更深层共鸣的用户体验，从而提高参与度、满意度和忠诚度。

作为一项前沿技术，向 GenUI 的转变充满了概念性和实践性的挑战。将人工智能整合到设计过程中，确保生成的界面不仅个性化，还要保持可用性、可访问性，并与整体品牌和用户体验保持一致，这些挑战使得 GenUI 成为少数人而非多数人能够追求的目标。此外，人工智能的参与还引发了关于数据隐私、透明度，甚至道德影响的问题。

尽管存在这些挑战，规模化的个性化体验有能力彻底改变我们与数字产品和服务的交互方式。它为创建包容性和无障碍的界面开辟了可能性，能够满足不同能力、背景或偏好用户的多样化需求。

在本章中，我们将探讨 GenUI 的概念，研究其一些定义性特征、主要优势和潜在挑战。我们首先考虑 GenUI 最基本和最容易实现的形式：为传统设计和实现的用户界面生成文本内容。

为用户界面生成文案

存在于应用程序界面框架中的文本元素，如表单标签、工具提示和说明文本，通常都是硬编码在模板或 UI 组件中的，为所有用户提供一致但通用的体验。使用上下文内容生成模式，你可以将这些静态元素转变为动态的、具有情境感知能力的个性化组件。

个性化表单

表单是网络和移动应用程序中无处不在的组成部分，是收集用户输入的主要方式。然而，传统表单往往呈现出一种通用且缺乏个性化的体验，其标准化的标签和字段可能并不总是符合用户的具体情境或需求。用户更有可能完成那些感觉针对其需求和偏好定制的表单，这将带来更高的转化率和用户满意度。

然而，在个性化和一致性之间取得平衡很重要。虽然使表单适应个别用户是有益的，但保持一定程度的熟悉性和可预测性至关重要。即使在个性化元素的情况下，用户仍应能够轻松识别和操作表单。

这里有一些个性化表单的灵感建议：

上下文字段建议

GenUI可以分析用户的历史交互、偏好和数据，作为预测来提供智能字段建议。例如，如果用户之前输入过他们的收货地址，表单可以自动用其保存的信息填充相关字段。这不仅节省时间，还表明应用程序理解并记住了用户的偏好。

等一下，这种技术不是不用人工智能也能实现吗？当然可以，但是用人工智能来驱动这类功能有两个显著优势：1）实现起来非常简单，2）随着用户界面的变化和演进，这种方式具有很强的适应性。

让我们为我们假设的订单处理系统快速创建一个服务，这个服务会主动地为用户填写正确的收货地址。

```
class OrderShippingAddressSubscriber
  include Raix::ChatCompletion

  attr_accessor :order

  delegate :customer, to: :order

  DIRECTIVE = "You are a smart order processing assistant. Given the
  customer's order history, guess the most likely shipping address
  for the current order."

  def order_created(order)
    return unless order.pending? && order.shipping_address.blank?
```

```
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
```

```
end
```

这个例子虽然非常简化，但适用于大多数情况。其理念是让 AI像人类一样进行推测。为了更清楚地说明我的意思，让我们来看一些示例数据：

```
Order History:
[
  {"date": "2024-01-03", "description": "garden soil mix",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-01-15", "description": "hardcover fiction novels",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-01-22", "description": "baby diapers", "shipping_address":
   "789 Suburb St, Quietville"},
  {"date": "2024-02-01", "description": "organic vegetables",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-02-17", "description": "mystery thriller book set",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-02-25", "description": "baby wipes",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-03-05", "description": "flower seeds",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-03-20", "description": "biographies",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-03-30", "description": "baby formula",
   "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-04-12", "description": "lawn fertilizer",
   "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-04-22", "description": "science fiction novels",
   "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-05-02", "description": "infant toys",
```

```
    "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-05-14", "description": "outdoor grill",
    "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-05-29", "description": "literary classics",
  "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-06-11", "description": "baby clothes",
    "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-07-01", "description": "watering can",
    "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-07-18", "description": "non-fiction essays",
  "shipping_address": "456 City Apt, Metroville"},
  {"date": "2024-07-28", "description": "baby bath items",
  "shipping_address": "789 Suburb St, Quietville"},
  {"date": "2024-08-09", "description": "herb garden kit",
  "shipping_address": "123 Country Lane, Rural Town"},
  {"date": "2024-08-24", "description": "children's books",
  "shipping_address": "456 City Apt, Metroville"}
]
```

你注意到数据中的模式了吗？我敢保证，对大型语言模型来说这都是小菜一碟。让我们来问问 GPT-4，“温度计 “最可能的收货地址是什么。

```
From the order history you've provided, it looks like the purchases are
generally clustered into three main types based on the shipping addresses:

123 Country Lane, Rural Town - This address often orders garden and
outdoor-related items like soil mix, vegetables, flower seeds, lawn
fertilizer, an outdoor grill, a watering can, and an herb garden kit.

456 City Apt, Metroville - This address tends to order various types of
books, indicating an interest in reading across genres like fiction,
thrillers, biographies, science fiction, literary classics, non-fiction
essays, and children's books.

789 Suburb St, Quietville - The purchases for this address are consistently
baby-related products, such as diapers, wipes, baby formula, infant toys,
baby clothes, and baby bath items.

Given these patterns, a digital thermometer could potentially be a fit for
any of the addresses based on different needs: outdoor temperature tracking,
a tool for a passionate reader interested in science gadgets, or a health
tool for a baby. However, considering its common use in monitoring health,
especially for babies, it's most likely to be purchased by someone who's
already buying baby care products.

Therefore, the most likely shipping address for a digital thermometer, based
on the purchasing patterns, would be 789 Suburb St, Quietville.
```

如果你认为使用像 GPT-4这样昂贵的模型来完成这项任务有点大材小用，你说得对！我在 Mistral 7B Instruct上尝试了相同的提示，它以每秒 75 个令牌的速度生成了以下响应，而成本仅为 0.000218 美元。

```
Based on the given order history, it appears that the customer has received
orders for baby-related items (diapers, wipes, formula, clothes, bath items)
at the address "789 Suburb St, Quietville" and garden-related items (soil,
seeds, watering can, herb garden kit) at the address "123 Country Lane, Rural
Town". The customer has also received books (fiction, non-fiction, literary
classics, mystery-thriller, science fiction, biographies, children's books)
at the address "456 City Apt, Metroville".

Given that the new order is for a "thermometer", which doesn't seem to fit
into any of the categories that the customer has previously ordered, it's
hard to accurately predict the shipping address based solely on the order
history. However, given the 50%-50% split between baby-related and
garden-related items, it could somewhat lean towards the Baby-related items
address ("789 Suburb St, Quietville"). But remember, this is an assumption
and cannot be definitively confirmed without more context or information.
```

对于许多在线零售商来说，这种技术的开销和成本是否值得用来创造更神奇的结账体验？答案是绝对值得。从目前情况来看，AI 计算的成本只会继续下降，特别是在开源模型托管服务提供商竞相压低价格的情况下。

使用提示模板和结构化输入输出以及响应围栏来优化这类聊天完成功能。

自适应字段排序

表单字段的呈现顺序会显著影响用户体验和完成率。通过 GenUI，你可以根据用户上下文和每个字段的重要性动态调整字段顺序。例如，如果用户正在填写健身应用的注册表单，表单可以优先展示与其健身目标和偏好相关的字段，使整个过程更具相关性和吸引力。

个性化微文案

表单相关的说明文字、错误信息和其他微文案也可以使用 GenUI进行个性化设计。不再显示像 “邮箱地址无效 “这样的通用错误信息，而是可以生成更有帮助和更具情境性的信息，比如 “请输入有效的邮箱地址以接收订单确认信息 “。这些个性化的细节可以使表单体验更加友好，减少用户的挫败感。

个性化验证

与个性化微文案相似，你可以使用 AI 以看似神奇的方式验证表单。想象一下让 AI 在语义层面验证用户资料表单，寻找潜在的错误。

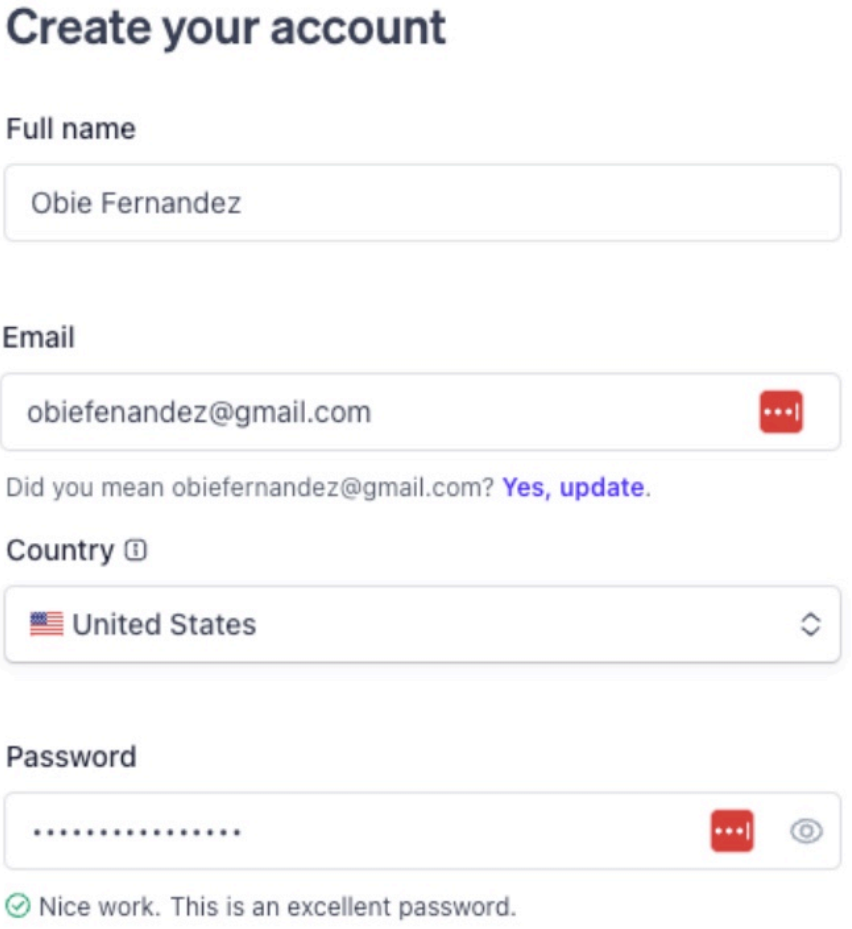


图 8. 你能发现正在进行的语义验证吗?

渐进式展示

GenUI 可以根据用户上下文智能判断哪些表单字段是必要的，并根据需要逐步显示额外字段。这种渐进式展示技术有助于减轻认知负担，使表单填写过程更易管理。例如，如果用户正在注册基础订阅，表单最初只显示必要字段，随着用户的操作进展或选择特定选项，可以动态引入其他相关字段。

上下文感知说明文本

工具提示通常用于在用户悬停或与特定元素交互时提供额外信息或指导。通过“上下文内容生成“方法，你可以生成适应用户上下文并提供相关信息的工具提示。例如，当用户在探索某个复杂功能时，工具提示可以根据他们之前的交互或技能水平提供个性化的提示或示例。

说明文本（如使用说明、描述或帮助信息）可以根据用户上下文动态生成。你可以使用大语言模型生成针对用户特定需求或问题定制的文本，而不是呈现通用解释。例如，如果用户在某个步骤遇到困难，说明文本可以提供个性化的指导或故障排除提示。

微文案指的是引导用户使用应用程序的简短文本，如按钮标签、错误信息或确认提示。通过将上下文内容生成方法应用于微文案，你可以创建响应用户操作并提供相关有用文本的自适应 UI。例如，如果用户即将执行关键操作，确认提示可以动态生成，提供清晰且个性化的信息。

个性化的说明文本和工具提示可以大大改善新用户的入门体验。通过提供具体情境的指导和示例，你可以帮助用户快速理解和导航应用程序，减少学习曲线并提高采用率。

动态和上下文感知的界面元素也可以使应用程序感觉更直观和引人入胜。当配套文本针对用户的特定需求和兴趣定制时，用户更有可能与功能互动并进行探索。

到目前为止，我们已经讨论了用 AI 增强现有 UI 范式的想法，但是从更激进的角度重新思考用户界面的设计和实现又会如何呢?

定义生成式 UI

与传统的 UI 设计不同，设计师们不再创建固定的、静态的界面，生成式 UI 暗示了一个软件具有灵活的、个性化的体验，能够实时进化和适应的未来。每当

我们使用 AI 驱动的对话式界面时，我们都在让 AI 适应用户的特定需求。生成式 UI 更进一步，将这种适应性水平应用到软件的_视觉_界面上。

今天我们之所以能够尝试生成式 UI 的想法，是因为大语言模型已经理解编程，其基础知识包括 UI 技术和框架。现在的问题是，是否可以使用大语言模型来生成为每个用户量身定制的 UI 元素，如文本、图像、布局，甚至整个界面。可以指示模型考虑各种因素，如用户的过往互动、明确的偏好、人口统计信息以及当前使用场景，以创建高度个性化且相关的界面。

生成式 UI在几个关键方面与传统用户界面设计不同：

1. ** 动态和自适应：** 传统 UI 设计涉及创建固定的、静态的界面，对所有用户都保持相同。相比之下，生成式 UI 支持能够根据用户需求和场景动态适应和改变的界面。这意味着同一应用程序可以向不同用户呈现不同的界面，甚至可以在不同情况下向同一用户呈现不同的界面。
2. ** 规模化个性化：** 在传统设计中，由于时间和资源的限制，为每个用户创建个性化体验通常是不切实际的。而生成式 UI 则允许规模化个性化。通过利用 AI，设计师可以创建自动适应每个用户独特需求和偏好的界面，而无需为每个用户群手动设计和开发单独的界面。
3. ** 注重成果：** 传统 UI 设计往往注重创建视觉吸引力和功能性的界面。虽然这些方面在生成式 UI 中仍然重要，但主要焦点转向实现期望的用户成果。生成式 UI 旨在创建针对每个用户特定目标和任务优化的界面，将可用性和效果置于纯粹的美学考虑之上。
4. ** 持续学习和改进：** 生成式 UI 系统可以基于用户互动和反馈持续学习和改进。当用户与生成的界面互动时，AI 模型可以收集用户行为、偏好和结果的数据，使用这些信息来改进和优化未来的界面生成。这种迭代学习过程使生成式 UI 系统能够随着时间推移越来越有效地满足用户需求。

需要注意的是，生成式 UI 与 AI 辅助设计工具不同，后者提供建议或自动化某些设计任务。虽然这些工具有助于简化设计过程，但它们仍然依赖设计师做出最终决定并创建静态界面。相比之下，生成式 UI 涉及 AI 系统在基于用户数据和场景的实际界面生成和适应方面发挥更积极的作用。

生成式 UI 代表了我们对用户界面设计方法的重大转变，从一刀切的解决方案转向高度个性化、自适应的体验。通过利用 AI 的力量，生成式 UI 有潜力彻底改变我们与数字产品和服务的交互方式，为每个用户创造更直观、更吸引人、更有效的界面。

示例

为了说明生成式 UI 的概念，让我们考虑一个假设的健身应用程序 “FitAI“。这个应用程序旨在根据用户的个人目标、健身水平和偏好提供个性化的锻炼计划和营养建议。

在传统的 UI 设计方法中，FitAI 可能有一套固定的屏幕和元素，对所有用户都是一样的。然而，通过生成式 UI，应用程序的界面可以动态适应每个用户的独特需求和场景。

这种方法在 2024 年实现起来可能有点难以想象，甚至可能没有足够的投资回报率，但它是可能的。

以下是它可能的工作方式：

1. **入门引导：**

    - 不是标准问卷，FitAI 使用对话式 AI来收集有关用户目标、当前健身水平和偏好的信息。
    - 基于这个初始互动，AI 生成个性化的仪表板布局，突出显示与用户目标最相关的功能和信息。
    - 当前的 AI 技术可能有一系列可供使用的屏幕组件来组成个性化仪表板。
    - 未来的 AI 技术可能会承担经验丰富的 UI 设计师的角色，实际上_从头开始_创建仪表板。

2. **健身计划规划器：**

- 人工智能会根据用户的经验水平和可用设备来调整健身计划界面。
- 对于没有器材的初学者，它可能会显示简单的体重训练动作，并配有详细的说明和视频。
- 对于能够使用健身房的高级用户，它可能会显示更复杂的训练计划，减少解释性内容。
- 健身计划的内容并非简单地从一个大型集合中筛选得出。它可以根据包含用户所有已知信息的上下文查询知识库，即时生成内容。

3. **进度追踪：**

- 进度追踪界面会根据用户的目标和参与模式而演变。
- 如果用户主要关注减重，界面可能会突出显示体重趋势图表和卡路里消耗统计数据。
- 对于增肌的用户，它可能会突出显示力量增长和身体成分变化。
- 人工智能可以根据用户的实际进展调整应用程序的这一部分。如果进度在一段时间内停滞，应用程序可以转换到一种模式，试图引导用户透露停滞的原因，以便加以改善。

4. **营养建议：**

- 营养部分会根据用户的饮食偏好和限制进行调整。
- 对于素食用户，它可能会显示植物性膳食建议和蛋白质来源。
- 对于不耐麦麸质的用户，它会自动从推荐中过滤掉含麸质的食物。
- 同样，内容并非从适用于所有用户的海量膳食数据中抽取，而是从知识库中综合生成，该知识库包含可根据用户具体情况和限制条件进行调整的信息。
- 例如，食谱会根据用户健康水平和身体数据的不断变化，生成符合其动态卡路里需求的配料规格。

5. **激励元素：**

- 应用程序的激励内容和通知会根据用户的性格类型和对不同激励策略的反应进行个性化设置。

- 某些用户可能会收到鼓励性的消息，而其他用户则获得更多数据驱动的反馈。

在这个例子中，生成式用户界面（GenUI）使 FitAI 能够为每个用户创建高度定制的体验，可能提高参与度、满意度和实现健身目标的可能性。界面元素、内容，甚至应用程序的 “个性 “都会调整以最好地服务于每个用户的需求和偏好。

向成果导向设计的转变

生成式用户界面代表了用户界面设计方法的根本转变，从关注创建特定界面元素转向更加整体的、成果导向的方法。这种转变有几个重要含义：

1. **关注用户目标：**
    - 设计师需要更深入地思考用户目标和期望的成果，而不是具体的界面组件。
    - 重点将放在创建能够生成帮助用户有效实现目标的界面的系统上。
    - 新的 UI 框架将会出现，为基于人工智能的设计师提供所需的工具，使其能够即时且从零开始生成用户体验，而不是基于预定义的屏幕规格。
2. **设计师角色的变化：**
    - 设计师将从创建固定布局转变为为人工智能系统定义规则、约束和指导方针。
    - 他们需要在数据分析、人工智能提示工程和系统思维等领域发展技能，以有效指导生成式用户界面系统。
3. **用户研究的重要性：**
    - 在生成式用户界面环境中，用户研究变得更加关键，因为设计师不仅需要了解用户偏好，还需要了解这些偏好和需求在不同情境下如何变化。

- 持续的用户测试和反馈循环对于完善和改进人工智能生成有效界面的能力至关重要。

4. **为可变性设计：**

- 设计师不再是创建单一的 “完美 “界面，而是需要考虑多种可能的变体，确保系统能够为不同用户需求生成适当的界面。
- 这包括为边缘案例设计，确保生成的界面在不同配置下保持可用性和可访问性。
- 产品差异化在用户心理学的不同视角以及利用竞争对手无法获得的独特数据集和知识库方面呈现出新的维度。

挑战和考虑因素

虽然生成式用户界面提供了令人兴奋的可能性，但它也带来了几个挑战和需要考虑的问题：

1. **技术限制：**

- 当前的人工智能技术虽然先进，但在理解复杂用户意图和生成真正具有情境感知的界面方面仍有限制。
- 在性能较弱的设备上实时生成界面元素相关的性能问题。

2. **数据要求：**

- 根据使用场景的不同，有效的生成式用户界面系统可能需要大量用户数据来生成个性化界面。
- 在伦理范围内获取真实用户数据的挑战引发了对数据隐私和安全的担忧，以及在训练生成式用户界面模型时使用的数据可能存在偏见的问题。

3. **可用性和一致性：**

- 至少在这种实践广泛应用之前,界面持续变化的应用程序可能会导致可用性问题,因为用户可能难以找到熟悉的元素或高效导航。
- 在个性化和保持一致、可学习的界面之间取得平衡将至关重要。

4. **过度依赖人工智能:**

   - 存在将设计决策过度委托给人工智能系统的风险，这可能导致缺乏创意、存在问题或简单来说就是有缺陷的界面选择。
   - 在可预见的未来，人工监督和能够覆盖人工智能生成设计的能力仍将很重要。

5. **无障碍访问问题:**

   - 确保动态生成的界面对残障用户保持无障碍访问带来了全新的挑战,考虑到典型系统在无障碍访问合规性方面表现不佳，这一点令人担忧。
   - 另一方面，人工智能设计师可能会在设计时就_内置_无障碍访问考虑，并具备为残障用户即时构建无障碍界面的能力，就像为非残障用户构建界面一样。
   - 无论如何，生成式用户界面系统都应该设计有健全的无障碍访问指南和测试流程。

6. **用户信任和透明度:**

   - 用户可能会对那些似乎 “过分了解 “他们或以他们无法理解的方式变化的界面感到不适。
   - 提供关于界面如何以及为什么进行个性化的透明度信息，对建立用户信任至关重要。

未来展望与机遇

生成式用户界面（GenUI）的未来在革新我们与数字产品和服务交互方式方面蕴含着巨大潜力。随着这项技术不断发展，我们可以预见用户界面的设计、实现和体验方式将发生翻天覆地的变化。我认为生成式用户界面是最终将推动我们的软件进入现在被认为是科幻领域的现象。

生成式用户界面最令人兴奋的前景之一是其增强无障碍访问的潜力，这种增强超越了仅仅确保严重残障人士不被完全排除在软件使用之外。通过自动适应个别用户需求，生成式用户界面可以使数字体验比以往更具包容性。试想界面能够无缝调整，为年轻或视力障碍用户提供更大的文字，为认知障碍用户提供简化的布局，所有这些都无需手动配置或单独的 “无障碍 “版本应用程序。

生成式用户界面的个性化能力很可能推动各类数字产品的用户参与度、满意度和忠诚度的提升。随着界面越来越适应个人偏好和行为，用户将发现数字体验更直观、更愉悦，这可能带来与技术更深入、更有意义的互动。

生成式用户界面还有潜力改变新用户的入门过程。通过创建直观、个性化的首次用户体验，并快速适应每个用户的专业水平，生成式用户界面可以显著降低学习新应用程序的难度。这可能导致更快的采用率，并增加用户探索新功能的信心。

另一个令人兴奋的可能性是生成式用户界面能够在不同设备和平台之间保持一致的用户体验，同时针对每个特定使用场景进行优化。这可能解决在日益碎片化的设备环境中提供连贯体验的长期挑战，从智能手机和平板电脑到台式电脑和新兴技术如增强现实眼镜。

生成式用户界面的数据驱动特性为用户界面设计的快速迭代和改进开辟了机会。通过收集用户与生成界面交互的实时数据，设计师和开发人员可以获得前所未有的用户行为和偏好洞察。这种反馈循环可能导致用户界面设计的持续改进，这种改进是由实际使用模式而不是假设或有限的用户测试驱动的。

为适应这种转变，设计师需要发展他们的技能组合和思维方式。工作重点将从

创建固定布局转向开发全面的设计系统和指南，这些系统和指南可以指导人工智能驱动的界面生成。设计师需要培养对数据分析、人工智能技术和系统思维的深入理解，以有效指导生成式用户界面系统。

此外，随着生成式用户界面模糊了设计和技术之间的界限，设计师需要与开发人员和数据科学家更紧密地合作。这种跨学科方法在创建不仅视觉吸引且用户友好，而且技术上稳健和伦理上可靠的生成式用户界面系统方面将至关重要。

随着生成式用户界面（GenUI）技术的成熟，其伦理影响也将成为关注的焦点。设计师将在制定负责任的人工智能界面设计框架方面发挥关键作用，确保个性化能够提升用户体验，同时避免侵犯隐私或以不道德的方式操纵用户行为。

展望未来，GenUI 既带来令人振奋的机遇，也带来重大挑战。它有潜力为全球用户创造更直观、更高效、更令人满意的数字体验。虽然这要求设计师适应变化并掌握新技能，但同时也提供了一个前所未有的机会，让我们能够以深刻而有意义的方式塑造人机交互的未来。实现完整的 GenUI 系统的道路无疑是复杂的，但就改善用户体验和提高数字可访问性而言，这样的未来值得我们为之努力。

智能工作流编排

在应用程序开发领域中，工作流在定义任务、流程和用户交互的结构与执行方式方面发挥着至关重要的作用。随着应用程序变得越来越复杂，用户期望不断提高，对智能和自适应工作流编排的需求也变得越发明显。

“智能工作流编排”方法着重于利用 AI 组件来动态编排和优化应用程序中的复杂工作流。其目标是创建更高效、更灵敏且能够根据实时数据和上下文进行自适应的应用程序。

在本章中，我们将探讨智能工作流编排方法的核心原则和模式。我们将考虑如何使用 AI 来智能路由任务、自动化决策制定，以及基于用户行为、系统性能和业务规则等各种因素动态调整工作流。通过实际示例和真实场景，我们将展示 AI 在简化和优化应用程序工作流方面的变革潜力。

无论您是在构建具有复杂业务流程的企业应用程序，还是具有动态用户旅程的面向消费者的应用程序，本章讨论的模式和技术都将为您提供知识和工具，以

创建能够提升整体用户体验并推动业务价值的智能高效工作流。

业务需求

传统的工作流管理方法通常依赖于预定义规则和静态决策树，这可能会显得僵化、缺乏灵活性，且无法应对现代应用程序的动态特性。

考虑一个电子商务应用程序需要处理复杂订单履行流程的场景。工作流可能涉及多个步骤，如订单验证、库存检查、支付处理、配送和客户通知。每个步骤可能都有其自身的规则集、依赖关系、外部集成和异常处理机制。通过手动或硬编码逻辑来管理这样的工作流很快就会变得繁琐、容易出错且难以维护。

此外，随着应用程序规模的扩大和并发用户数量的增长，工作流可能需要根据实时数据和系统性能进行自适应和优化。例如，在流量高峰期，应用程序可能需要动态调整工作流以优先处理某些任务、高效分配资源并确保流畅的用户体验。

这正是“智能工作流编排“方法发挥作用的地方。通过利用 AI 组件，开发人员可以创建智能、自适应和自优化的工作流。AI 可以分析海量数据，从过去的经验中学习，并实时做出明智的决策来有效编排工作流。

主要优势

1. **提高效率：** AI 可以优化任务分配、资源利用和工作流执行，从而缩短处理时间并提高整体效率。
2. **适应性：**由 AI 驱动的工作流可以动态适应不断变化的条件，如用户需求、系统性能或业务需求的波动，确保应用程序保持响应性和弹性。
3. **自动化决策：** AI 可以自动化工作流中的复杂决策过程，减少人工干预并最小化人为错误的风险。
4. **个性化：** AI 可以分析用户行为、偏好和上下文，以个性化工作流并为个别用户提供量身定制的体验。

5. **可扩展性：**由 AI 支持的工作流可以无缝扩展以处理不断增加的数据量和用户交互，而不会影响性能或可靠性。

在接下来的章节中，我们将探讨实现智能工作流的关键模式和技术，并展示 AI 如何在现代应用程序中改变工作流管理的真实案例。

关键模式

为了在应用程序中实现智能工作流编排，开发人员可以利用几个关键模式来发挥 AI 的力量。这些模式为设计和管理工作流提供了结构化方法，使应用程序能够基于实时数据和上下文进行适应、优化和自动化处理。让我们探讨智能工作流编排中的一些基本模式。

动态任务路由

这种模式涉及使用 AI 根据各种因素（如任务优先级、资源可用性和系统性能）智能地在工作流中路由任务。AI 算法可以分析每个任务的特征，考虑系统的当前状态，并做出明智的决策，将任务分配给最合适的资源或处理路径。动态任务路由确保任务得到高效分配和执行，优化整体工作流性能。

```
class TaskRouter
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  attr_accessor :task

  # list of functions that can be called by the AI entirely at its
  # discretion depending on the task received

  function :analyze_task_priority do
```

```
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
```

注意第 29 行的`while` 表达式创建的循环，该循环会持续提示 AI 直到任务被分配。

在第 35 行，我们保存任务的记录以供后续分析和调试使用（如果需要的话）。

上下文决策制定

你可以使用非常相似的代码来在工作流中进行上下文感知决策。通过分析相关数据点，如用户偏好、历史模式和实时输入，AI 组件可以在工作流的每个决策点确定最合适的行动方案。根据每个用户或场景的具体上下文调整工作流的行为，从而提供个性化和优化的体验。

自适应工作流组合

这种模式着重于基于不断变化的需求或条件动态组合和调整工作流。AI 可以分析工作流的当前状态，识别瓶颈或低效之处，并自动修改工作流结构以优化性能。自适应工作流组合使应用程序能够在不需要人工干预的情况下持续演进和改进其流程。

异常处理和恢复

异常处理和恢复是智能工作流编排的关键方面。在使用 AI 组件和复杂工作流时，预测并优雅地处理异常对于确保系统的稳定性和可靠性至关重要。

以下是智能工作流中异常处理和恢复的一些关键考虑因素和技术：

1. **异常传播：**实现一种在工作流组件间传播异常的一致方法。当组件内发生异常时，应该捕获、记录并将其传播到编排器或负责处理异常的独立组件。这样做的目的是集中异常处理并防止异常被静默吞噬，同时为智能错误处理开启可能性。
2. **重试机制：**重试机制有助于提高工作流的弹性并优雅地处理间歇性故障。对于瞬态或可恢复的异常，如网络连接或资源不可用的情况，一定要实现重试机制，这些异常可以在指定延迟后自动重试。有了 AI 驱动的编排器

或异常处理器，意味着你的重试策略不必是机械性的，依赖于指数回退等固定算法。你可以将重试的处理交给负责决定如何处理异常的 AI 组件的“判断“。

3. **回退策略：**如果 AI 组件无法提供有效响应或遇到错误（考虑到其前沿性质，这是常见情况），要有回退机制来确保工作流可以继续。这可能涉及使用默认值、替代算法或人在环路中来做出决策并保持工作流向前推进。
4. **补偿操作：**编排器的指令应包括关于补偿操作的说明，以处理无法自动解决的异常。补偿操作是为撤销或缓解失败操作影响而采取的步骤。例如，如果支付处理步骤失败，补偿操作可以是回滚交易并通知用户。补偿操作有助于在面对异常时维护数据一致性和完整性。
5. **异常监控和告警：** 设置监控和告警机制来检测并通知相关利益相关者关键异常。可以让编排器知道阈值和规则，在异常超过特定限制或发生特定类型的异常时触发告警。这允许在问题影响整个系统之前主动识别和解决问题。

以下是 Ruby 工作流组件中异常处理和恢复的示例：

```
class InventoryManager
  def check_availability(order)
    begin
      # Perform inventory check logic
      inventory = Inventory.find_by(product_id: order.product_id)
      if inventory.available_quantity >= order.quantity
        return true
      else
        raise InsufficientInventoryError,
              "Insufficient inventory for product #{order.product_id}"
      end
    rescue InsufficientInventoryError => e
      # Log the exception
```

```
      logger.error("Inventory check failed: #{e.message}")

      # Retry the operation after a delay
      retry_count ||= 0
      if retry_count < MAX_RETRIES
        retry_count += 1
        sleep(RETRY_DELAY)
        retry
      else
        # Fallback to manual intervention
        NotificationService.admin("Inventory check failed: Order #{order.id}")
        return false
      end
    end
  end
end
```

在这个例子中，`InventoryManager` 组件检查给定订单的产品可用性。如果可用数量不足，它会抛出`InsufficientInventoryError`。该异常会被捕获并记录，同时实现了重试机制。如果超过重试限制，组件会通过通知管理员来转为人工干预。

通过实施健壮的异常处理和恢复机制，你可以确保你的智能工作流具有弹性，可维护性，并能优雅地处理意外情况。

这些模式构成了智能工作流编排的基础，可以根据不同应用程序的具体需求进行组合和调整。通过利用这些模式，开发人员可以创建灵活、有弹性且针对性能和用户体验进行优化的工作流。

在下一节中，我们将探讨如何在实践中实施这些模式，使用真实世界的例子和代码片段来说明 AI 组件与工作流管理的集成。

在实践中实施智能工作流编排

现在我们已经探讨了智能工作流编排中的关键模式，让我们深入了解如何在实际应用中实施这些模式。我们将提供实际的例子和代码片段来说明 AI 组件与工作流管理的集成。

智能订单处理器

让我们深入探讨一个在 Ruby on Rails电子商务应用程序中使用 AI 驱动的`OrderProcessor` 组件来实施智能工作流编排的实际例子。该`OrderProcessor` 实现了我们在第 3 章讨论多重工作者时首次遇到的流程管理器企业集成概念。该组件将负责管理订单履行工作流，基于中间结果做出路由决策，并协调各个处理步骤的执行。

订单履行过程包括多个步骤，如订单验证、库存检查、支付处理和配送。每个步骤都作为单独的工作进程实现，执行特定任务并将结果返回给`OrderProcessor`。这些步骤不是强制性的，甚至不一定要按照精确的顺序执行。

以下是`OrderProcessor` 的示例实现。它具有来自Raix的两个混入模块。第一个（`ChatCompletion`）赋予它聊天补全的能力，这使其成为一个 AI 组件。第二个（`FunctionDispatch`）启用了 AI 的函数调用功能，允许它通过函数调用而不是文本消息来响应提示。

工作函数（`validate_order`、`check_inventory` 等）委托给它们各自的工作类，这些类可以是 AI 或非 AI 组件，唯一的要求是它们返回的工作结果可以表示为字符串格式。

与本书这部分的所有其他示例一样，此代码实际上是伪代码，仅用于传达模式的含义并启发你自己的创作。第 2 部分包含了完整的模式描述和完整的代码示例。

```
class OrderProcessor
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are an order processor, tasked with..."

  def initialize(order)
    self.order = order
    transcript << { system: SYSTEM_DIRECTIVE }
    transcript << { user: order.to_json }
  end

  def perform
    # will continue looping until `stop_looping!` is called
    chat_completion(loop: true)
  end

  # list of functions available to be called by the AI
  # truncated for brevity

  def functions
    [
      {
        name: "validate_order",
        description: "Invoke to check validity of order",
        parameters: {
          ...
      },
      ...
```

```
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def validate_order
    OrderValidationWorker.perform(@order)
  end

  def check_inventory
    InventoryCheckWorker.perform(@order)
  end

  def process_payment
    PaymentProcessingWorker.perform(@order)
  end

  def schedule_shipping
    ShippingSchedulerWorker.perform(@order)
  end

  def send_confirmation
    OrderConfirmationWorker.perform(@order)
  end

  def finished_processing
    @order.update!(transcript:, processed_at: Time.current)
    stop_looping!
```

```
59    end
60  end
```

在这个示例中，OrderProcessor 通过一个订单对象进行初始化，并以大语言模型原生的对话记录格式维护工作流程执行的转录文本。AI 被赋予完全的控制权来协调各种处理步骤的执行，例如订单验证、库存检查、支付处理和发货。

每次调用`chat_completion` 方法时，转录文本都会被发送给 AI 以获取作为函数调用的完成结果。完全由 AI 来分析前一步骤的结果并确定采取适当的行动。例如，如果库存检查显示库存水平较低，`OrderProcessor` 可以安排补货任务。如果支付处理失败，它可以启动重试或通知客户支持。

> 上面的示例中没有定义补货或通知客户支持的函数，但它完全可以包含这些功能。

每次调用函数时，转录文本都会增长，并作为工作流程执行的记录，包括每个步骤的结果和 AI 生成的下一步指令。这个转录文本可用于调试、审计和提供订单履行过程的可见性。

通过在`OrderProcessor` 中利用 AI，电子商务应用程序可以根据实时数据动态调整工作流程，并智能地处理异常情况。AI 组件可以做出明智的决策，优化工作流程，确保即使在复杂场景中也能顺利处理订单。

当你意识到工作进程的唯一要求就是返回一些 AI 可以理解的输出来决定下一步该做什么时，你可能会开始明白这种方法如何能够减少在集成不同系统时通常涉及的输入/输出映射工作。

智能内容审核器

社交媒体应用程序通常需要至少最基本的内容审核来确保社区的安全和健康。这个示例`ContentModerator` 组件利用 AI 来智能地协调审核工作流程，根据内容

特征和各种审核步骤的结果做出决策。

审核过程涉及多个步骤，如文本分析、图像识别、用户信誉评估和人工审核。每个步骤都作为独立的工作进程实现，执行特定任务并将结果返回给ContentModerator。

以下是ContentModerator 的示例实现：

```
class ContentModerator
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a content moderator process manager,
    tasked with the workflow involved in moderating user-generated content..."

  def initialize(content)
    @content = content
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: content.to_json }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

```

```
  # list of functions available to be called by the AI
  # truncated for brevity

  def functions
    [
      {
        name: "analyze_text",
        # ...
      },
      {
        name: "recognize_image",
        description: "Invoke to describe images...",
        # ...
      },
      {
        name: "assess_user_reputation",
        # ...
      },
      {
        name: "escalate_to_manual_review",
        # ...
      },
      {
        name: "approve_content",
        # ...
      },
      {
        name: "reject_content",
        # ...
```

```
      }
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def analyze_text
    result = TextAnalysisWorker.perform(@content)
    continue_with(result)
  end

  def recognize_image
    result = ImageRecognitionWorker.perform(@content)
    continue_with(result)
  end

  def assess_user_reputation
    result = UserReputationWorker.perform(@content.user)
    continue_with(result)
  end

  def escalate_to_manual_review
    ManualReviewWorker.perform(@content)
    @content.update!(status: 'pending', transcript: @transcript)
  end

  def approve_content
    @content.update!(status: 'approved', transcript: @transcript)
```

```
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
```

在这个示例中，ContentModerator 通过一个内容对象进行初始化，并以对话格式维护审核记录。AI 组件完全控制着审核工作流程，根据内容特征和每个步骤的结果来决定执行哪些步骤。

AI 可以调用的工作函数包括analyze_text、recognize_image、assess_user_reputation 和escalate_to_manual_review。每个函数都会将任务委托给相应的工作进程（TextAnalysisWorker、ImageRecognitionWorker 等）并将结果追加到审核记录中，其中上报函数作为终止状态。同样，approve_content 和reject_content 函数也作为终止状态。

AI 组件分析内容并确定采取适当的行动。如果内容包含图像引用，它可以调用recognize_image 工作程序来协助进行视觉审查。如果任何工作程序警告存在潜在有害内容，AI 可能会决定将内容上报进行人工审核，或直接拒绝。但根据警告的严重程度，AI 可能会选择使用用户信誉评估的结果来决定如何处理其他不确定的内容。根据具体用例，可信用户可能在发布内容方面有更大的自由度。如此等等……

与前面的流程管理器示例一样，审核记录用于记录工作流程的执行情况，包括

每个步骤的结果和 AI 生成的决策。这些记录可用于审计、透明度和随时间改进审核流程。

通过在ContentModerator 中利用 AI，社交媒体应用可以根据内容特征动态调整审核工作流程，并智能处理复杂的审核场景。AI 组件可以做出明智的决策，优化工作流程，确保社区体验的安全和健康。

让我们再探讨两个示例，展示在智能工作流程编排环境中的预测性任务调度和异常处理与恢复。

客户支持系统中的预测性任务调度

在使用 Ruby on Rails构建的客户支持应用中，高效管理和优先处理支持工单对于及时为客户提供帮助至关重要。SupportTicketScheduler 组件利用 AI 来预测性地调度支持工单并将其分配给可用的客服人员，这个过程基于工单紧急程度、客服专业知识和工作负载等多个因素。

```
class SupportTicketScheduler
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a support ticket scheduler,
    tasked with intelligently assigning tickets to available agents..."

  def initialize(ticket)
    @ticket = ticket
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: ticket.to_json }
    ]
  end
```

```
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        },
```

```
      {
        name: "reschedule_ticket",
        # ...
      }
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

  def analyze_ticket_urgency
    result = TicketUrgencyAnalyzer.perform(@ticket)
    continue_with(result)
  end

  def list_available_agents
    result = ListAvailableAgents.perform
    continue_with(result)
  end

  def predict_agent_workload
    result = AgentWorkloadPredictor.perform
    continue_with(result)
  end

  def assign_ticket_to_agent
    TicketAssigner.perform(@ticket, @transcript)
  end

```

```
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
```

在这个示例中，SupportTicketScheduler 通过支持工单对象进行初始化，并维护一个调度记录。AI 组件分析工单详情，并基于工单紧急程度、客服专业知识和预测的客服工作量等因素，预测性地安排工单分配。

AI 可以调用的功能包括analyze_ticket_urgency、list_available_agents、predict_agent_workload 和assign_ticket_to_agent。每个功能都将任务委托给相应的分析器或预测器组件，并将结果追加到调度记录中。AI 还可以使用delay_assignment 功能来延迟分配。

AI 组件检查调度记录并对工单分配做出明智的决策。它考虑工单的紧急程度、可用客服的专业知识以及每个客服的预测工作量，以确定最适合处理该工单的客服。

通过利用预测性任务调度，客户支持应用程序可以优化工单分配，减少响应时间，提高整体客户满意度。主动高效地管理支持工单确保将正确的工单在正确的时间分配给正确的客服。

数据处理管道中的异常处理和恢复

处理异常和从故障中恢复对确保数据完整性和防止数据丢失至关重要。DataProcessingOrchestrator组件利用 AI 智能处理异常并协调数据处理管道中的恢复过程

```
class DataProcessingOrchestrator
  include Raix::ChatCompletion
  include Raix::FunctionDispatch

  SYSTEM_DIRECTIVE = "You are a data processing orchestrator..."

  def initialize(data_batch)
    @data_batch = data_batch
    @transcript = [
      { system: SYSTEM_DIRECTIVE },
      { user: data_batch.to_json }
    ]
  end

  def perform
    complete(@transcript)
  end

  def model
    "openai/gpt-4"
  end

  def functions
    [
      {
```

```
        name: "validate_data",
        # ...
      },
      {
        name: "process_data",
        # ...
      },
      {
        name: "request_fix",
        # ...
      },
      {
        name: "retry_processing",
        # ...
      },
      {
        name: "mark_data_as_failed",
        # ...
      },
      {
        name: "finished",
        # ...
      }
    ]
  end

  # implementation of functions that can be called by the AI
  # entirely at its discretion, depending on the needs of the order

```

```
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
```

```
    @data_batch.update!(status: 'finished', transcript: @transcript)
  end

  private

  def continue_with(result)
    @transcript << { function: result }
    complete(@transcript)
  end

  def handle_validation_exception(exception)
    @transcript << { exception: exception.message }
    complete(@transcript)
  end

  def handle_processing_exception(exception)
    @transcript << { exception: exception.message }
    complete(@transcript)
  end
end
```

在这个示例中，DataProcessingOrchestrator 通过数据批次对象进行初始化，并维护一个处理记录。AI 组件负责编排数据处理流水线，处理异常并在需要时从故障中恢复。

AI 可以调用的功能包括validate_data、process_data、request_fix、retry_processing 和mark_data_as_failed。每个功能都会将任务委派给相应的数据处理组件，并将结果或异常详情添加到处理记录中。

如果在validate_data 步骤中发生验证异常，handle_validation_exception 函数会将异常数据添加到记录中，并将控制权返回给 AI。同样，如果在process_data

步骤中发生处理异常，AI 可以决定采取何种恢复策略。

根据遇到的异常性质，AI 可以自行决定调用`request_fix`，该函数会委派给 AI 驱动的`SmartDataFixer` 组件（参见自修复数据章节）。数据修复器会获得一个用简单英语描述的说明，说明应该如何修改`@data_batch` 以便重试处理。也许成功的重试可能需要从数据批次中删除未通过验证的记录，和/或将它们复制到不同的处理流水线中供人工审查？可能性几乎是无限的。

通过引入 AI 驱动的异常处理和恢复机制，数据处理应用变得更具弹性和容错性。`DataProcessingOrchestrator` 智能地管理异常，最小化数据丢失，并确保数据处理工作流的顺利执行。

监控和日志记录

监控和日志记录为 AI 驱动的工作流组件的进度、性能和健康状况提供了可见性，使开发人员能够跟踪和分析系统的行为。实施有效的监控和日志记录机制对于调试、审计和持续改进智能工作流至关重要。

监控工作流进度和性能

为确保智能工作流的顺利执行，监控每个工作流组件的进度和性能很重要。这涉及在整个工作流生命周期中跟踪关键指标和事件。

需要监控的一些重要方面包括：

1. **工作流执行时间：**测量每个工作流组件完成任务所需的时间。这有助于识别性能瓶颈并优化整体工作流效率。

2. **资源利用率：**监控每个工作流组件对系统资源（如 CPU、内存和存储）的使用情况。这有助于确保系统在其容量范围内运行，并能有效处理工作负载。

3. **错误率和异常：**跟踪工作流组件中的错误和异常发生情况。这有助于识别潜在问题，并实现主动的错误处理和恢复。

4. **决策点和结果：** 监控工作流中的决策点和 AI 驱动决策的结果。这为 AI 组件的行为和效果提供了洞察。

监控过程捕获的数据可以在仪表板中显示，或用作定期报告的输入，以告知系统管理员系统的健康状况。

监控数据可以输入到 AI 驱动的系统管理员进程中供其审查和采取可能的行动！

记录关键事件和决策

日志记录是一项重要的实践，涉及捕获和存储工作流执行过程中发生的关键事件、决策和异常的相关信息。

需要记录的一些重要方面包括：

1. **工作流启动和完成：**记录每个工作流实例的开始和结束时间，以及输入数据和用户上下文等相关元数据。

2. **组件执行：**记录每个工作流组件的执行详情，包括输入参数、输出结果和生成的任何中间数据。

3. **AI 决策和推理：**记录 AI 组件做出的决策，以及相关的推理过程或置信度分数。这提供了透明度，并使 AI 驱动的决策可以被审计。

4. **异常和错误消息：**记录工作流执行过程中遇到的任何异常或错误消息，包括堆栈跟踪和相关的上下文信息。

日志记录可以使用各种技术实现，例如写入日志文件、将日志存储在数据库中，或将日志发送到集中式日志服务。选择一个提供灵活性、可扩展性并易于与应用程序架构集成的日志框架很重要。

以下是在 Ruby on Rails 应用程序中使用`ActiveSupport::Logger`类实现日志记录的示例：

```ruby
class WorkflowLogger
  def self.log(message, severity = :info)
    @logger ||= ActiveSupport::Logger.new('workflow.log')
    @logger.formatter ||= proc do |severity, datetime, progname, msg|
      "#{datetime} [#{severity}] #{msg}\n"
    end
    @logger.send(severity, message)
  end
end

# Usage example
WorkflowLogger.log("Workflow initiated for order ##{@order.id}")
WorkflowLogger.log("Payment processing completed successfully")
WorkflowLogger.log("Inventory check failed for item ##{item.id}", :error)
```

通过在工作流组件和 AI 决策点中战略性地放置日志记录语句，开发人员可以捕获有价值的信息用于调试、审计和分析。

监控和日志记录的优势

在智能工作流编排中实施监控和日志记录具有以下几个优势：

1. **调试和故障排除：**详细的日志和监控数据帮助开发人员快速识别和诊断问题。它们提供了工作流执行流程、组件交互以及遇到的任何错误或异常的洞察。

2. **性能优化：**监控性能指标使开发人员能够识别瓶颈并优化工作流组件以提高效率。通过分析执行时间、资源利用率和其他指标，开发人员可以做出明智的决策来改善系统的整体性能。

3. **审计和合规：**记录关键事件和决策为监管合规和问责提供了审计跟踪。它使组织能够追踪和验证 AI 组件采取的行动，并确保遵守业务规则和法律要求。

4. **持续改进：**监控和日志记录数据为智能工作流的持续改进提供了宝贵的输入。通过分析历史数据、识别模式并衡量 AI 决策的有效性，开发人员可以迭代地改进和增强工作流编排逻辑。

注意事项和最佳实践

在智能工作流编排中实施监控和日志记录时，请考虑以下最佳实践：

1. **定义清晰的监控指标：**根据工作流的具体要求确定需要监控的关键指标和事件。关注能够提供系统性能、健康状况和行为有意义见解的指标。

2. **实施细粒度日志记录：**确保在工作流组件和 AI 决策点的适当位置放置日志记录语句。捕获相关的上下文信息，如输入参数、输出结果和生成的任何中间数据。

3. **使用结构化日志记录：**采用结构化日志记录格式以便于解析和分析日志数据。结构化日志记录允许更好地搜索、过滤和聚合日志条目。

4. **管理日志保留和轮换：**实施日志保留和轮换策略以管理日志文件的存储和生命周期。根据法律要求、存储限制和分析需求确定适当的保留期限。如果可能，将日志记录转移到第三方服务，如 Papertrail。

5. **保护敏感信息：**在记录敏感信息时要谨慎，比如个人身份信息（PII）或机密业务数据。实施适当的安全措施，如数据掩码或加密，以保护日志文件中的敏感信息。

6. **集成监控和告警工具：**利用监控和告警工具来集中收集、分析和可视化监控和日志数据。这些工具可以提供实时洞察，基于预定义的阈值生成告警，并促进主动问题检测和解决。我最喜欢的这类工具是 Datadog。

通过实施全面的监控和日志记录机制，开发人员可以获得对智能工作流行为和性能的宝贵洞察。这些洞察能够实现 AI 驱动的工作流编排系统的有效调试、优化和持续改进。

可扩展性和性能考虑因素

可扩展性和性能是设计和实施智能工作流编排系统时需要考虑的关键方面。随着并发工作流的数量和 AI 驱动组件的复杂性增加，确保系统能够高效处理工作负载并无缝扩展以满足不断增长的需求变得至关重要。

处理大量并发工作流

智能工作流编排系统通常需要处理大量的并发工作流。为确保可扩展性，请考虑以下策略：

1. **异步处理：**实施异步处理机制以解耦工作流组件的执行。这使系统能够同时处理多个工作流，而无需阻塞或等待每个组件完成。异步处理可以通过消息队列、事件驱动架构或后台作业处理框架（如 Sidekiq）来实现。

2. **分布式架构：**设计系统架构以使用无服务器组件（如 AWS Lambda）或简单地将工作负载分配到多个节点或服务器，与主应用服务器一起工作。这实现了水平可扩展性，可以添加额外的节点来处理增加的工作流量。

3. **并行执行：**识别工作流中的并行执行机会。某些工作流组件可能相互独立，可以并发执行。通过利用并行处理技术（如多线程或分布式任务队列），系统可以优化资源利用并减少整体工作流执行时间。

AI 驱动组件的性能优化

AI 驱动的组件，如机器学习模型或自然语言处理引擎，可能会占用大量计算资源，从而影响工作流编排系统的整体性能。要优化 AI 组件的性能，请考虑以下技术：

1. **缓存：**如果你的 AI 处理纯粹是生成式的，不涉及实时信息查询或外部集成来生成对话完成，那么你可以考虑使用缓存机制来存储和重用那些经常访问或计算密集型操作的结果。

2. **模型优化：**持续优化工作流组件中 AI 模型的使用方式。这可能涉及诸如提示词精炼等技术，或者可能仅仅是在新模型发布时进行测试。

3. **批处理：**如果你正在使用 GPT-4 级别的模型，你可能可以利用批处理技术在单个批次中处理多个数据点或请求，而不是单独处理它们。通过批量处理数据，系统可以优化资源利用并减少重复模型请求的开销。

监控和性能分析

为了识别性能瓶颈并优化智能工作流编排系统的可扩展性，实施监控和分析机制至关重要。考虑以下方法：

1. **性能指标：**定义并跟踪关键性能指标，如响应时间、吞吐量、资源利用率和延迟。这些指标能够深入了解系统性能并帮助识别需要优化的领域。流行的 AI 模型聚合器OpenRouter在每个 API 响应中都包含主机[1]和速度[2]指标，使得跟踪这些关键指标变得轻而易举。

2. **分析工具：**使用分析工具来分析各个工作流组件和 AI 操作的性能。分析工具可以帮助识别性能热点、低效的代码路径或资源密集型操作。常用的分析工具包括 New Relic、Scout，或编程语言或框架提供的内置分析器。

3. **负载测试：**进行负载测试以评估系统在不同并发工作负载下的性能。负载测试有助于识别系统的可扩展性限制，检测性能下降，并确保系统能够在不影响性能的情况下处理预期流量。

4. **持续监控：**实施持续监控和告警机制，以主动检测性能问题和瓶颈。设置监控仪表板和告警来跟踪关键性能指标（KPI），并在预定义阈值被突破时接收通知。这能够实现性能问题的及时识别和解决。

[1]主机时间是指从模型主机接收到流式生成的第一个字节所需的时间，也就是 “首字节时间 “。

[2]速度是通过完成令牌数除以总生成时间计算得出的。对于非流式请求，延迟被视为生成时间的一部分。

扩展策略

为了处理不断增加的工作负载并确保智能工作流编排系统的可扩展性，请考虑以下扩展策略：

1. **垂直扩展：**垂直扩展涉及增加单个节点或服务器的资源（如 CPU、内存）以处理更高的工作负载。当系统需要更多的处理能力或内存来处理复杂的工作流或 AI 操作时，这种方法是适用的。

2. **水平扩展：**水平扩展涉及向系统添加更多的节点或服务器以分配工作负载。当系统需要处理大量并发工作流，或者工作负载可以轻松地分布在多个节点上时，这种方法是有效的。水平扩展需要分布式架构和负载均衡机制来确保流量的均匀分配。

3. **自动扩展：**实施自动扩展机制，根据工作负载需求自动调整节点数量或资源。自动扩展允许系统根据传入流量动态扩展或收缩，确保最佳的资源利用和成本效益。像亚马逊云服务（AWS）或谷歌云平台（GCP）这样的云平台提供了可以用于智能工作流编排系统的自动扩展功能。

性能优化技术

除了扩展策略之外，还要考虑以下性能优化技术来提高智能工作流编排系统的效率：

1. **高效的数据存储和检索：**优化工作流组件使用的数据存储和检索机制。使用高效的数据库索引、查询优化技术和数据缓存来最小化延迟并提高数据密集型操作的性能。

2. **异步 I/O：**利用异步 I/O 操作来防止阻塞并提高系统的响应能力。异步 I/O 允许系统在不等待 I/O 操作完成的情况下同时处理多个请求，从而最大化资源利用率。

3. **高效序列化和反序列化：**优化工作流组件之间数据交换所使用的序列化和反序列化过程。使用高效的序列化格式，如 Protocol Buffers或 MessagePack，以减

少数据序列化的开销并提高组件间通信的性能。

对于基于 Ruby 的应用程序，考虑使用Universal ID。Universal ID 利用 MessagePack 和 Brotli（一个为速度和最佳数据压缩而构建的组合）。当结合使用时，这些库比 Protocol Buffers 快 30%，压缩率相差仅 2-5%。

4. **压缩和编码：**应用压缩和编码技术来减少工作流组件之间传输的数据大小。压缩算法，如 gzip或 Brotli，可以显著减少网络带宽使用并提高系统的整体性能。

在设计和实现智能工作流编排系统时考虑可扩展性和性能方面，可以确保您的系统能够处理大量并发工作流，优化 AI 驱动组件的性能，并无缝扩展以满足不断增长的需求。持续监控、分析和优化工作对于随着工作负载和复杂性随时间增加而维持系统的性能和响应能力至关重要。

工作流的测试和验证

测试和验证是开发和维护智能工作流编排系统的关键方面。考虑到 AI 驱动工作流的复杂性，确保每个组件按预期运行、整体工作流正确运行以及 AI 决策准确可靠是至关重要的。在本节中，我们将探讨测试和验证智能工作流的各种技术和注意事项。

工作流组件的单元测试

单元测试涉及独立测试各个工作流组件以验证其正确性和健壮性。在对 AI 驱动的工作流组件进行单元测试时，请考虑以下几点：

1. **输入验证：**测试组件处理不同类型输入的能力，包括有效和无效数据。验证组件是否能优雅地处理边缘情况并提供适当的错误消息或异常。

2. **输出验证：**断言组件对给定的输入集产生预期的输出。将实际输出与预期结果进行比较以确保正确性。

3. **错误处理：**通过模拟各种错误场景来测试组件的错误处理机制，如无效输入、资源不可用或意外异常。验证组件是否适当地捕获和处理错误。

4. **边界条件：**测试组件在边界条件下的行为，如空输入、最大输入大小或极端值。确保组件能够优雅地处理这些条件而不会崩溃或产生错误结果。

以下是使用 RSpec 测试框架对工作流组件进行单元测试的示例：

```
RSpec.describe OrderValidator do
  describe '#validate' do
    context 'when order is valid' do
      let(:order) { build(:order) }

      it 'returns true' do
        expect(subject.validate(order)).to be true
      end
    end

    context 'when order is invalid' do
      let(:order) { build(:order, total_amount: -100) }

      it 'returns false' do
        expect(subject.validate(order)).to be false
      end
    end
  end
end
```

在这个例子中，`OrderValidator` 组件使用两个测试用例进行测试：一个用于有效订单，另一个用于无效订单。这些测试用例验证`validate` 方法是否根据订单的有效性返回预期的布尔值。

集成测试工作流交互

集成测试侧重于验证不同工作流组件之间的交互和数据流。它确保组件之间能够无缝协作并产生预期的结果。在对智能工作流进行集成测试时，需要考虑以下几点：

1. 组件交互： 测试工作流组件之间的通信和数据交换。验证一个组件的输出是否正确地作为工作流中下一个组件的输入。

2. 数据一致性： 确保数据在工作流中流转时保持一致和准确。验证数据转换、计算和聚合是否正确执行。

3. 异常传播： 测试异常和错误是如何在工作流组件之间传播和处理的。验证异常是否被正确捕获、记录并适当处理，以防止工作流中断。

4. 异步行为： 如果工作流涉及异步组件或并行执行，测试协调和同步机制。确保工作流在并发和异步场景下表现正确。

以下是使用 RSpec 测试框架对 Ruby 工作流进行集成测试的示例：

```
RSpec.describe OrderProcessingWorkflow do

  let(:order) { build(:order) }

  it 'processes the order successfully' do
    expect(OrderValidator).to receive(:validate).and_return(true)
    expect(InventoryManager).to receive(:check_availability).and_return(true)
    expect(PaymentProcessor).to receive(:process_payment).and_return(true)
    expect(ShippingService).to receive(:schedule_shipping).and_return(true)

    workflow = OrderProcessingWorkflow.new(order)
    result = workflow.process

```

```
    expect(result).to be true
    expect(order.status).to eq('processed')
  end

end
```

在这个例子中，通过验证不同工作流组件之间的交互来测试 `OrderProcessingWorkflow`。测试用例设置了每个组件行为的预期结果，并确保工作流能够成功处理订单，相应地更新订单状态。

测试人工智能决策点

测试人工智能决策点对确保 AI 驱动的工作流的准确性和可靠性至关重要。在测试人工智能决策点时，需要考虑以下几点：

1. **决策准确性：**验证人工智能组件基于输入数据和训练模型做出准确的决策。将人工智能的决策与预期结果或真实标注数据进行比较。

2. **边界情况：**测试人工智能组件在边界情况和异常场景下的行为。验证人工智能组件能够优雅地处理这些情况并做出合理的决策。

3. **偏见和公平性：**评估人工智能组件是否存在潜在偏见，确保其做出公平和无偏见的决策。使用多样化的输入数据测试组件，并分析结果中是否存在任何歧视性模式。

4. **可解释性：**如果人工智能组件为其决策提供解释或推理，请验证解释的正确性和清晰度。确保解释与底层的决策过程相符。

以下是使用 RSpec 测试框架在 Ruby 中测试人工智能决策点的示例：

```ruby
RSpec.describe FraudDetector do
  describe '#detect_fraud' do
    context 'when transaction is fraudulent' do
      let(:tx) do
        build(:transaction, amount: 10_000, location: 'High-Risk Country')
      end

      it 'returns true' do
        expect(subject.detect_fraud(tx)).to be true
      end
    end

    context 'when transaction is legitimate' do
      let(:tx) do
        build(:transaction, amount: 100, location: 'Low-Risk Country')
      end

      it 'returns false' do
        expect(subject.detect_fraud(tx)).to be false
      end
    end
  end
end
```

在这个例子中，FraudDetector AI 组件通过两个测试用例进行测试：一个用于欺诈交易，另一个用于合法交易。这些测试用例验证了detect_fraud 方法是否能够根据交易的特征返回预期的布尔值。

端到端测试

端到端测试涉及从开始到结束测试整个工作流程，模拟真实世界的场景和用户交互。它确保工作流程能够正确运行并产生预期的结果。在对智能工作流进行端到端测试时，需要考虑以下几点：

1. **用户场景：**识别常见的用户场景，并测试工作流在这些场景下的行为。验证工作流能否正确处理用户输入、做出适当的决策并产生预期的输出。

2. **数据验证：**确保工作流对用户输入进行验证和清理，以防止数据不一致或安全漏洞。使用各种类型的输入数据测试工作流，包括有效和无效的数据。

3. **错误恢复：**测试工作流从错误和异常中恢复的能力。模拟错误场景并验证工作流是否能够优雅地处理这些错误，记录错误日志，并采取适当的恢复措施。

4. **性能和可扩展性：**评估工作流在不同负载条件下的性能和可扩展性。使用大量并发请求测试工作流，并测量响应时间、资源利用率和整体系统稳定性。

以下是使用 Ruby语言、RSpec 测试框架和 Capybara 库对工作流进行端到端测试的示例：

```
RSpec.describe 'Order Processing Workflow' do
  scenario 'User places an order successfully' do
    visit '/orders/new'
    fill_in 'Product', with: 'Sample Product'
    fill_in 'Quantity', with: '2'
    fill_in 'Shipping Address', with: '123 Main St'
    click_button 'Place Order'

    expect(page).to have_content('Order Placed Successfully')
    expect(Order.count).to eq(1)
    expect(Order.last.status).to eq('processed')
  end
end
```

在这个示例中，端到端测试模拟用户通过网页界面下单的过程。它填写必需的表单字段，提交订单，并验证订单是否已成功处理，包括显示适当的确认消息以及更新数据库中的订单状态。

持续集成和部署

为确保智能工作流程的可靠性和可维护性，建议将测试和验证集成到持续集成和部署（CI/CD）流水线中。这样可以在工作流程部署到生产环境之前进行自动化测试和验证。请考虑以下实践：

1. **自动化测试执行：**配置 CI/CD 流水线，使其在工作流程代码库发生变更时自动运行测试套件。这确保在开发过程的早期就能检测到任何回归问题或故障。

2. **测试覆盖率监控：**对工作流程组件和 AI 决策点的测试覆盖率进行度量和监控。致力于实现高测试覆盖率，以确保关键路径和场景得到充分测试。

3. **持续反馈：**将测试结果和代码质量指标集成到开发工作流程中。向开发人员持续提供有关测试状态、代码质量以及 CI/CD 过程中发现的任何问题的反馈。

4. **预发布环境：** 将工作流程部署到与生产环境高度类似的预发布环境中。在预发布环境中执行额外的测试和验证，以发现任何与基础设施、配置或数据集成相关的问题。

5. **回滚机制：** 实施回滚机制，以应对部署失败或生产环境中发现的严重问题。确保工作流程能够快速恢复到之前的稳定版本，以最大限度地减少停机时间和对用户的影响。

通过在智能工作流程的整个开发生命周期中融入测试和验证，组织可以确保其 AI 驱动系统的可靠性、准确性和可维护性。定期的测试和验证有助于发现错误、防止回归问题，并建立对工作流程行为和结果的信心。

第二部分：模式

提示工程

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

思维链

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

内容生成

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

结构化实体创建

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

大语言模型代理指导

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

优势与注意事项

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

模式切换

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

使用场景

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

角色分配

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

使用场景

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

提示对象

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

提示模板

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

优势和注意事项

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

使用场景

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

结构化输入输出

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

扩展结构化输入输出

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

优势与注意事项

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

提示链接

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

使用场景

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例：Olympia 的引导流程

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

提示重写器

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

响应围栏

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

优势和注意事项

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

错误处理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

查询分析器

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

实现

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

词性标注（POS）和命名实体识别（NER）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

意图分类

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

关键词提取

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

优势

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

查询重写器

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

优势

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

腹语术师模式

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

使用场景

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

离散组件

谓词

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

使用场景

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

API 外观模式

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

主要优势

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

使用时机

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

身份验证和授权

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

请求处理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

响应格式化

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

错误处理和边界情况

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

可扩展性和性能考虑

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

与其他设计模式的比较

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

结果解释器

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

使用场景

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

虚拟机

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

使用时机

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

魔法背后的原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

规格说明和测试

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

明确行为规格

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

编写测试用例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例：测试翻译器组件

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

HTTP 交互的重放

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

人在环路中（HITL）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

高层次模式

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

混合智能

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

自适应响应

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

人机角色切换

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

升级处理机制

工作原理

主要优势

现实应用：医疗保健

反馈循环

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

应用和示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

人工反馈集成的高级技术

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

被动信息辐射

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

上下文信息显示

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

主动通知

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

解释性见解

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

交互式探索

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

主要优势

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

应用和示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

协作决策（CDM）

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

持续学习

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

应用和示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

伦理考虑

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

HITL 在缓解 AI 风险中的作用

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

技术进步与未来展望

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

人在环系统的挑战和局限性

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

智能错误处理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

传统错误处理方法

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

上下文错误诊断

工作原理

上下文错误诊断的提示工程

用于上下文错误诊断的检索增强生成

智能错误报告

预测性错误预防

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

智能错误恢复

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

个性化错误通信

工作原理

自适应错误处理工作流

工作原理

质量控制

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

评估器

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

问题

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

解决方案

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

注意事项

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

理解黄金标准参考

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

无参考评估的工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

防护机制

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

问题

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

解决方案

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

工作原理

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

示例

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

注意事项

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

护栏和评估：一枚硬币的两面

护栏和无参考评估的互换性

实现双重用途的护栏和评估

术语表

术语表

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

A

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

B

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

C

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

D

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

E

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

F

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

G

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

H

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

I

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

J

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

K

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

L

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

M

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

N

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

O

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

P

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

Q

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

R

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

S

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

T

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

U

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

V

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

W

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

Z

此内容没有包含在样章中。您可以在 Leanpub 上购买这本书，网址为 http://leanpub.com/patterns-of-application-development-using-ai-zh-Hans

Index