



Patterns of Application Development Using AI

Obie Fernandez
Foreword by Gregor Hohpe



Leanpub

日本語版

AI によるアプリケーション開発パターン (日本語版)

Obie Fernandez

本書はこちらで販売中です <http://leanpub.com/patterns-of-application-development-using-ai-ja>

この版は 2025-01-23 に発行されました。



本書は [Leanpub](#) の電子書籍です。Leanpub はリーンパブリッシングプロセスで著者や出版社を支援します。[リーンパブリッシング](#) は新しい出版スタイルです。軽量のツールを使って執筆中の電子書籍を出版し、読者のフィードバックをもらいながら魅力的な本に仕上がるまでピボットを繰り返すことができます。

© 2025 Obie Fernandez

Twitter でシェアしませんか？

本書に関するコメントを[Twitter](#)でシェアして Obie Fernandez を応援してください！

本書のハッシュタグは [#poaduai](#) です。

本書に関するコメントを検索する場合は、次のリンクをクリックして下さい。Twitter のハッシュタグを使って検索できます。

[#poaduai](#)

最強の女王様であり、私のミューズ、私の光明そして愛する *Victoria* へ

Obie Fernandez 共著

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 による序文	i
はじめに	ii
本書について	iii
コード例について	iii
本書で扱わない内容	iii
対象読者	iii
共通の語彙の構築	iii
参加方法	iii
謝辞	iv
イラストについて	iv
リーク・パブリッシングについて	iv
著者について	v
はじめに	1
ソフトウェアアーキテクチャについての考察	2
大規模言語モデルとは	3
推論について	5
パフォーマンスについて考える	25
異なる LLM モデルの実験	26
複合 AI システム	27

パート 1: 基本的なアプローチと技法	35
パスを狭める	36
潜在空間: 理解を超えた広大さ	38
どのように道筋が「絞り込まれる」のか	42
生のモデル対指示調整済みモデル	45
プロンプトエンジニアリング	52
プロンプト蒸留	68
ファインチューニングについてはどうですか?	75
検索拡張生成 (Retrieval Augmented Generation: RAG)	77
検索拡張生成とは?	77
RAG はどのように機能するか?	77
なぜアプリケーションで RAG を使用するのか?	77
アプリケーションでの RAG の実装	77
命題チャンキング	78
RAG の実世界での例	79
インテリジェントクエリ最適化 (IQO)	79
リランキング	79
RAG 評価 (RAGAs)	79
課題と将来の展望	81
ワーカーの群れ	84
独立した再利用可能なコンポーネントとしての AI ワーカー	85
アカウント管理	87
E コマースアプリケーション	88
ヘルスケアでの応用	98
プロセスマネージャーとしての AI ワーカー	100
アプリケーションアーキテクチャへの AI ワーカーの統合	104
AI ワーカーの構成性とオーケストレーション	107

従来 of 自然言語処理と LLM の組み合わせ	117
ツールの使用	120
ツールの使用とは	120
ツール使用の可能性	122
ツール使用のワークフロー	123
ツール使用のベストプラクティス	139
ツールの組み合わせと連鎖	144
将来の方向性	146
ストリーム処理	148
ReplyStream の実装	149
「会話ループ」	156
自動継続	158
結論	160
自己修復データ	162
実践的ケーススタディ: 破損した JSON の修復	164
考慮事項と禁忌	169
コンテキスト型コンテンツ生成	183
パーソナライゼーション	184
生産性	186
高速イテレーションと実験	188
AI 駆動のローカライゼーション	190
ユーザーテストとフィードバックの重要性	192
生成的 UI	193
ユーザーインターフェース用のコピーの生成	194
ジェネラティブ UI の定義	204
例	206

成果指向設計へのシフト	208
課題と考慮事項	209
将来の展望と機会	211
インテリジェントワークフローオーケストレーション	215
ビジネスニーズ	216
主要なメリット	217
主要パターン	217
例外処理とリカバリー	220
インテリジェントワークフローオーケストレーションの実践的な実装	223
モニタリングとロギング	241
スケーラビリティとパフォーマンスの考慮事項	245
ワークフローのテストと検証	250
 パート 2: パターン集	 258
プロンプトエンジニアリング	259
チェーン・オブ・ソート	260
モードスイッチ	262
役割割り当て	263
プロンプトオブジェクト	264
プロンプトテンプレート	265
ストラクチャード IO	266
プロンプトチェーニング	267
プロンプトリライター	268
レスポンスフェンシング	269
クエリアナライザー	270
クエリライター	272
ベントリロキスト	273

個別コンポーネント	274
述語	275
API ファサード	276
結果インタープリター	279
仮想マシン	280
仕様とテスト	280
ヒューマン・イン・ザ・ループ(HITL)	282
高レベルパターン	282
エスカレーション	284
フィードバックループ	285
パッシブ情報放射	286
協調的意思決定(CDM)	288
継続的学習	289
倫理的考慮事項	289
技術的進歩と将来の展望	290
インテリジェントなエラー処理	291
従来のエラー処理アプローチ	291
文脈的エラー診断	292
知的エラー報告	293
予測的エラー防止	294
スマートエラーリカバリー	294
パーソナライズされたエラーコミュニケーション	295
適応型エラー処理ワークフロー	296
品質管理	297
Eval	298
ガードレール	300
ガードレールと評価:コインの表と裏	301

用語集	302
用語集	302
Index	308

Gregor Hohpe による序文

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

はじめに

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

本書について

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

コード例について

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

本書で扱わない内容

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

対象読者

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

共通の語彙の構築

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

参加方法

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

謝辞

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

イラストについて

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

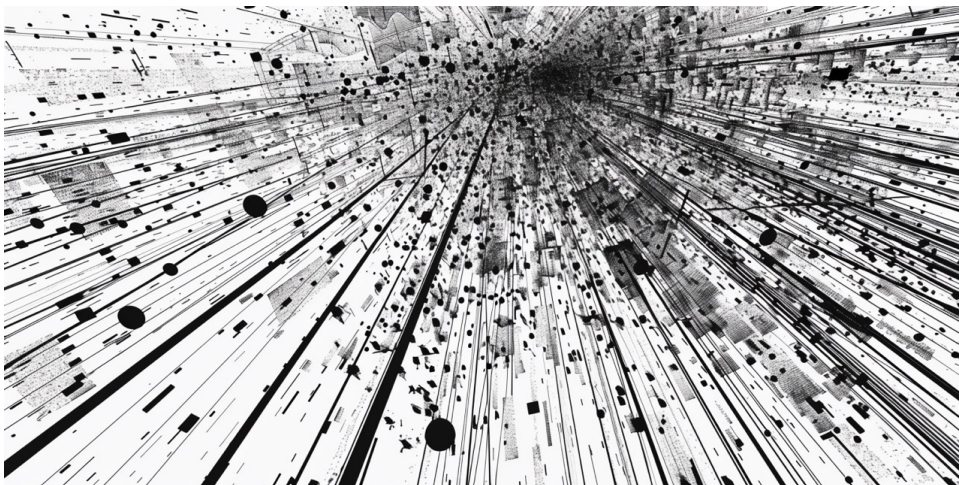
リーン・パブリッシングについて

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

著者について

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

はじめに



AI 大規模言語モデル(LLM)をプログラミングプロジェクトに統合することに意欲的な方は、後の章で紹介するパターンとコード例にすぐに取り組んでいただいて構いません。ただし、これらのパターンの力と可能性を十分に理解するためには、より広い文脈と、それらが表す統合的なアプローチを理解する時間を取る価値があります。

これらのパターンは単なる個別の手法の集まりではなく、アプリケーションに AI を統合するための統一的なフレームワークです。私は Ruby on Railsを使用していますが、これらのパターンは他のほとんどのプログラミング環境でも機能するはずです。データ管理からパフォーマンスの最適化、ユーザーエクスペリエンス、セキュリティまで、幅広い課題に対応し、従来のプログラミング手法を AI の機能で強化するための包括的なツールキットを提供します。

パターンの各カテゴリーは、アプリケーションに AI コンポーネントを組み込む際に生じる特定の課題や機会に対処します。これらのパターン間の関係や相乗効果を理解

することで、AI をどこでどのように最も効果的に適用するかについて、十分な情報に基づいた決定を下すことができます。

パターンは決して規範的な解決策ではなく、そのように扱うべきではありません。それらは、独自のアプリケーションの固有の要件と制約に合わせて調整できる、適応可能な構成要素として意図されています。これらのパターン(ソフトウェア分野における他のパターンと同様に)の成功的な適用には、問題領域、ユーザーニーズ、およびプロジェクトの全体的な技術アーキテクチャについての深い理解が必要です。

ソフトウェアアーキテクチャについての考察

私は 1980 年代にプログラミングを始め、ハッカーシーンに関わっていました。そして、プロのソフトウェア開発者になった後も、ハッカーとしての考え方を失うことはありませんでした。最初から、象牙の塔にいるソフトウェアアーキテクトたちが実際にどのような価値をもたらしているのかについて、健全な懐疑心を持っていました。

この強力な新しい AI 技術の波がもたらす変化に個人的に非常に興奮している理由の 1 つは、ソフトウェアアーキテクチャの決定と考えられているものへの影響です。これは、ソフトウェアプロジェクトの設計と実装における「正しい」方法とされているものについての従来の概念に挑戦します。また、AI による強化により、プロジェクトのどの部分でも、いつでも変更が容易になっているため、アーキテクチャを主に**変更が困難なシステムの部分**として考えることができるかどうかにも疑問を投げかけています。

おそらく私たちは、ソフトウェアエンジニアリングにおける「ポストモダン」アプローチの最盛期に入りつつあるのでしょう。このコンテキストでのポストモダンとは、開発者がすべてのコードの記述と保守に責任を持つという従来のパラダイムからの根本的な転換を指します。代わりに、データ操作、複雑なアルゴリズム、さらにはアプリケーションロジック全体をサードパーティライブラリや外部 API に委ねるという考え方を受け入れています。このポストモダンへの転換は、アプリケーションを一から構築するという従来の知恵からの大きな転換を表し、開発プロセスにおける開発者の役割を再考することを求めています。

私は常に、Larry Wallや彼のような他のハッカーの賢人たちの教えに基づき、優れたプログラマーは絶対に必要なコードだけを書くべきだと信じてきました。コードの量を最小限に抑えることで、より迅速に動き、バグの発生する可能性を減らし、保守を簡略化し、アプリケーション全体の信頼性を向上させることができます。コードが少ないことで、他のサービスに作業を委ねながら、コアとなるビジネスロジックとユーザーエクスペリエンスに集中することができます。

今や AI を活用したシステムが、これまで人間が書いたコードの専売特許だった作業を処理できるようになり、私たちはこれまで以上にビジネス価値とユーザーエクスペリエンスの創造に焦点を当てながら、より生産的で機敏になれるはずです。

もちろん、プロジェクトの大部分を AI システムに委託することには、制御の喪失や、堅牢なモニタリングとフィードバック機構の必要性といったトレードオフが存在します。そのため、AI の基本的な仕組みの理解を含む、新しいスキルと知識が必要となります。

大規模言語モデルとは

大規模言語モデル(LLM)は、2020 年に OpenAI が GPT-3 をリリースして以来、大きな注目を集めている人工知能モデルの一種です。LLM は人間の言語を処理し、理解し、驚くべき正確さと流暢さで生成するように設計されています。このセクションでは、LLM の仕組みと、なぜそれらがインテリジェントなシステムコンポーネントの構築に適しているのかについて簡単に見ていきます。

LLM の核となるのは、深層学習アルゴリズム、特にニューラルネットワークです。これらのネットワークは、情報を処理して伝達する相互接続されたノード(ニューロン)で構成されています。LLM で選択される代表的なアーキテクチャは Transformer モデルで、テキストなどの順次データの処理に非常に効果的であることが証明されています。

Transformer モデルはアテンション機構に基づいており、主に自然言語処理などの順次データを扱うタスクに使用されます。Transformer は入力データを順番に処理す

るのではなく、一度にすべて処理するため、長距離の依存関係をより効果的に捉えることができます。入力データの異なる部分に注目してコンテキストと関係性を理解するのに役立つアテンション機構のレイヤーを持っています。

LLM のトレーニングプロセスには、書籍、記事、ウェブサイト、コードリポジトリなど、膨大な量のテキストデータをモデルに与えることが含まれます。トレーニング中、モデルはテキスト内のパターン、関係性、構造を認識することを学習します。文法規則、単語の関連性、文脈的な意味など、言語の統計的特性を捉えます。

LLM のトレーニングで使用される主要な技術の 1 つが教師なし学習です。これは、モデルが明示的なラベル付けや指導なしでデータから学習することを意味します。トレーニングデータ内の単語やフレーズの共起を分析することで、独自にパターンと表現を発見します。これにより LLM は言語とその複雑さについての深い理解を発達させることができます。

LLM のもう一つの重要な側面は、コンテキストを扱う能力です。テキストを処理する際、LLM は個々の単語だけでなく、周囲のコンテキストも考慮します。テキストの意味と意図を理解するために、前の単語、文、さらには段落も考慮に入れます。このコンテキストの理解により、LLM は一貫性のある適切な応答を生成できます。特定の LLM モデルの能力を評価する主な方法の 1 つは、応答を生成するために考慮できるコンテキストの大きさを検討することです。

トレーニングが完了すると、LLM は幅広い言語関連タスクに使用できます。人間のようなテキストを生成し、質問に答え、文書を要約し、言語を翻訳し、さらにはコードを書くこともできます。LLM の多様性により、ユーザーとの対話、テキストデータの処理と分析、意味のある出力の生成が可能なインテリジェントなシステムコンポーネントの構築に価値があります。

アプリケーションアーキテクチャに LLM を組み込むことで、ユーザー入力を理解して処理し、動的なコンテンツを生成し、インテリジェントな推奨やアクションを提供する AI コンポーネントを作成できます。しかし、LLM の活用には、リソース要件とパフォーマンスのトレードオフを慎重に検討する必要があります。LLM は計算負荷が高く、動作に大きな処理能力とメモリ(つまり、お金)が必要になる可能

性があります。ほとんどの場合、アプリケーションに LLM を統合するコストの影響を評価し、それに応じて行動する必要があります。

推論について

推論とは、モデルが新しい未知のデータに基づいて予測や出力を生成するプロセスを指します。これは、訓練済みのモデルがユーザーの入力に応じてテキスト、画像、その他のコンテンツを生成したり、判断を下したりする段階です。

訓練段階では、AI モデルは予測の誤差を最小限に抑えるようにパラメータを調整しながら、大規模なデータセットから学習します。訓練が完了すると、モデルは学習した内容を新しいデータに適用できるようになります。推論とは、モデルが学習したパターンと知識を使用して出力を生成する方法です。

大規模言語モデルの場合、推論とはプロンプトや入力テキストを受け取り、トークン（これについてはすぐに説明します）の流れとして、文脈に沿った適切な応答を生成することを意味します。これには質問への回答、文章の補完、物語の生成、テキストの翻訳など、多くのタスクが含まれます。



あなたや私の考え方とは対照的に、AI モデルの「思考」は推論を通じて、すべて一つのステートレスな操作で行われます。つまり、その思考は生成プロセスに限定されています。まるで私があなたに質問をして、「意識の流れ」スタイルでの応答のみを受け付けるかのように、文字通り声に出して考えなければならないのです。

大規模言語モデルには多様なサイズと種類がある

実用的なほとんどすべての大規模言語モデル (LLM) は、同じ基本的なトランスフォーマーアーキテクチャに基づき、膨大なテキストデータセットで訓練されていますが、さまざまなサイズで提供され、異なる目的のためにフ

ファインチューニングされています。LLM のサイズは、そのニューラルネットワークのパラメータ数で測定され、その能力に大きな影響を与えます。GPT-4のような 1~2 兆個のパラメータを持つと噂される大規模なモデルは、一般的に小規模なモデルよりも知識が豊富で高性能です。ただし、大規模なモデルは実行により多くの計算能力を必要とし、API コールを通じて使用する際にはより高いコストがかかります。

LLM をより実用的で特定の用途に適したものにするため、ベースモデルはよりのみを絞ったデータセットでファインチューニングされることが多くあります。例えば、会話 AI に特化させるため、大規模な対話コーパスで訓練されることがあります。また、[プログラミングの知識を身につけさせるためにコードで訓練されたモデル](#)もあります。さらには、[ユーザーとのロールプレースタイルのやり取りのために特別に訓練されたモデル](#)まであります！

検索型モデルと生成モデル

大規模言語モデル (LLM) の世界では、応答を生成するための主要な 2 つのアプローチがあります：検索ベースモデルと生成モデルです。それぞれのアプローチには長所と短所があり、これらの違いを理解することで、特定のユースケースに適したモデルを選択することができます。

検索ベースモデル

検索ベースモデル (情報検索モデルとしても知られる) は、既存のテキストの大規模なデータベースを検索し、入力クエリに基づいて最も関連性の高い文章を選択することで応答を生成します。これらのモデルは新しいテキストを一から生成するのではなく、データベースからの抜粋を組み合わせで一貫性のある応答を形成します。

検索ベースモデルの主な利点の一つは、事実に基づいた最新の情報を提供できる能力です。キュレーションされたテキストのデータベースに依存しているため、信頼できるソースから関連情報を取得してユーザーに提示することができます。これにより、質問応答システムやナレッジベースのような、正確で事実に基づいた回答が必要なアプリケーションに適しています。

しかし、検索ベースモデルにはいくつかの制限があります。これらのモデルは検索対象のデータベースの質に依存するため、データベースの品質と網羅性がモデルの性能に直接影響を与えます。さらに、これらのモデルは、データベース内の利用可能なテキストに制限されるため、一貫性があり自然な応答の生成に苦勞することがあります。

本書では、純粋な検索モデルの使用については扱いません。

生成モデル

一方、生成モデルは、学習時に習得したパターンと関係性に基づいて、新しいテキストを一から作成します。これらのモデルは、言語に対する理解を活用して、入力プロンプトに合わせた新しい応答を生成します。

生成モデルの主な強みは、創造的で一貫性があり、文脈に即したテキストを生成できる能力です。オープンエンドな会話を行い、物語を生成し、さらにはコードを書くこともできます。これにより、チャットボット、コンテンツ作成、創作支援ツールなど、よりオープンエンドでダイナミックなインタラクションが必要なアプリケーションに適しています。

ただし、生成モデルは、キュレートされた事実のデータベースではなく、学習時に習得したパターンに依存するため、時として一貫性のない、あるいは事実と異なる情報を生成することがあります。また、バイアスや幻覚により、もっともらしいが必ずしも真実ではないテキストを生成する可能性が高くなる場合があります。

生成型 LLM の例としては、OpenAI の GPT シリーズ (GPT-3、GPT-4) や Anthropic の Claude が 있습니다。

ハイブリッドモデル

市販されている LLM の中には、検索と生成の両方のアプローチを組み合わせたハイブリッドモデルがいくつかあります。これらのモデルは、検索技術を使用してデータベースから関連情報を見つけ、その情報を生成技術で一貫性のある応答に合成します。

ハイブリッドモデルは、検索ベースモデルの事実に基づく正確さと、生成モデルの自然言語生成能力を組み合わせることを目指しています。これらは、オープンエンドな会話を維持しながら、より信頼性が高く最新の情報を提供することができます。

検索ベースモデルと生成モデルのどちらを選ぶかは、アプリケーションの具体的な要件を考慮する必要があります。主な目的が正確で事実に基づく情報の提供である場合は、検索ベースモデルが最適な選択かもしれません。アプリケーションがよりオープンエンドで創造的なインタラクションを必要とする場合は、生成モデルがより適している可能性があります。ハイブリッドモデルは、両方のアプローチのバランスを取り、事実の正確性と自然言語生成の両方を必要とするアプリケーションに適しています。

最終的に、検索ベースモデルと生成モデルの選択は、具体的なユースケースと、正確性、創造性、柔軟性のトレードオフによって決まります。それぞれのアプローチの長所と限界を理解することで、十分な情報に基づいた決定を下すことができます。

指示チューニング

指示チューニングは、会話トレーニングのサブセットで、人間が書いたプロンプトと応答を使用して、LLM の指示理解と実行能力を向上させます。以下に、人間が書いたプロンプトと応答による指示チューニングが、LLM の指示理解と実行能力をどのように向上させるかを示す簡単な例を示します：

例 1：カスタマーサポート

指示：「返金を求める顧客に対する応答を提供してください。」

人間が書いたプロンプト：「最近の購入品について返金を請求したいと思います。商品が期待通りではありませんでした。」

人間が書いた応答：「商品が期待に添えなかったことを申し訳なく思います。返金手続きを進めるため、注文番号と返品理由をお知らせください。ご要望を確認の上、2-3 営業日以内にご連絡させていただきます。」

例 2：教育支援

指示：「10 歳の子供に光合成の概念を説明してください。」

人間が書いたプロンプト：「植物がどうやって食べ物を作るのか説明してくれる？」

人間が書いた応答：「もちろん！植物は光合成という過程で食べ物を作ります。太陽の光と水と二酸化炭素を使って、食べ物と酸素を作ります。植物の葉には葉緑素という特別な部分があって、それが太陽の光を捕まえるのを手伝ってくれます。」

例 3：パーソナルアシスタント

指示：「来週火曜日の午後 3 時にミーティングをスケジュールして。」

人間が書いたプロンプト:「火曜日の午後 3 時にミーティングを設定できますか？」

人間が書いた応答:「はい、来週火曜日の午後 3 時にミーティングをスケジュールしました。他に何かご要望はありますか？」

その結果、異なるサイズと専門性を持つ LLM の多様なエコシステムが生まれました。1-70 億パラメータ範囲の小規模モデルは、より効率的に実行できながら、優れた一般的な言語処理能力を提供します。

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

300-700 億パラメータ程度の中規模モデルは、より強力な推論能力と指示への追従能力を提供します。

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

アプリケーションに LLM を組み込む際は、モデルの能力と、コスト、レイテンシー、コンテキスト長、コンテンツフィルタリングなどの実用的な要因のバランスを取る必要があります。より単純な言語タスクには、小規模な指示チューニング済みモデルが最適な選択であることが多く、複雑な推論や分析には最大規模のモデルが必要となる場合があります。また、モデルの学習データも重要な考慮事項であり、これがモデルの知識のカットオフ日を決定します。



Perplexityのような一部のモデルはリアルタイムの情報源に接続されているため、事実上カットオフ日がありません。質問をすると、独自にウェブ検索を実行し、任意のウェブページを取得して回答を生成することができます。

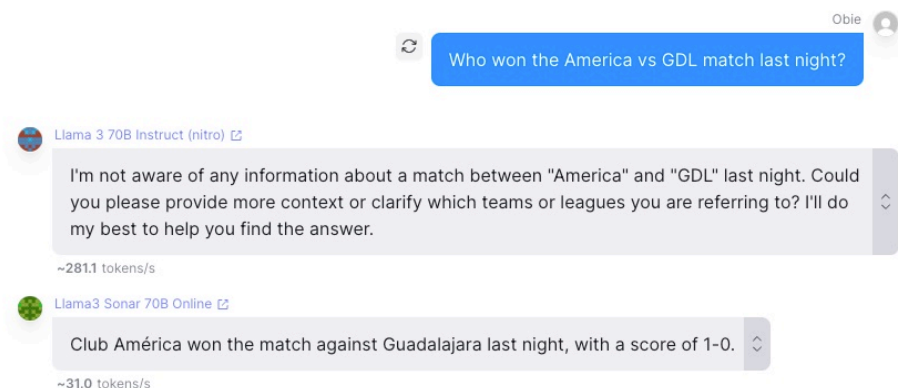


図 1. オンラインアクセスの有無による Llama3 の違い

結局のところ、万能な LLM は存在しません。モデルのサイズ、アーキテクチャ、トレーニングの違いを理解することが、特定のユースケースに適したモデルを選択する鍵となります。異なるモデルを実験的に試してみることが、手元のタスクに最適なパフォーマンスを提供するモデルを見つけ出す唯一の実践的な方法です。

トークン化：テキストを小さな単位に分割する

大規模言語モデルがテキストを処理する前に、そのテキストをトークンと呼ばれる小さな単位に分割する必要があります。トークンは、個々の単語、単語の一部、あるいは単一の文字になることがあります。テキストをトークンに分割するプロセスはトークン化と呼ばれ、言語モデルのデータを準備する上で重要なステップです。

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

図 2. この文は 27 個のトークンを含んでいます

異なる LLM は異なるトークン化戦略を使用しており、これはモデルの性能と機能に

大きな影響を与えます。LLM で使用される一般的なトークナイザーには以下のようなものがあります：

- **GPT(バイトペアエンコーディング)**: GPT トークナイザーは、バイトペアエンコーディング (BPE) と呼ばれる技術を使用してテキストをサブワード単位に分割します。BPE はテキストコーパス内で最も頻繁に出現するバイトのペアを反復的に結合し、サブワードトークンの語彙を形成します。これにより、トークナイザーは珍しい単語や新しい単語を、より一般的なサブワードに分解して処理することができます。GPT トークナイザーは、GPT-3 や GPT-4 などのモデルで使用されています。
- **Llama (SentencePiece)**: Llama トークナイザーは SentencePiece ライブラリを使用します。これは教師なしテキストトークナイザーおよびデトークナイザーです。SentencePiece は入力テキストを Unicode 文字のシーケンスとして扱い、学習コーパスに基づいてサブワード語彙を学習します。Unicode でエンコードできるあらゆる言語を処理できるため、多言語モデルに適しています。Llama トークナイザーは Meta の Llama や Alpaca などのモデルで使用されています。
- **SentencePiece (ユニグラム)**: SentencePiece トークナイザーは、サブワード正規化技術に基づくユニグラムと呼ばれる異なるアルゴリズムも使用できます。ユニグラムトークン化は、個々のサブワード単位に確率を割り当てるユニグラム言語モデルに基づいて、最適なサブワード語彙を決定します。このアプローチは、BPE と比較してより意味的に有意なサブワードを生成できます。ユニグラムを使用した SentencePiece は、Google の T5 や BERT などのモデルで使用されています。
- **Google Gemini (マルチモーダルトークン化)**: Google Gemini は、テキスト、画像、音声、動画、コードなど、さまざまなデータタイプを処理できるように設計されたトークン化方式を使用します。このマルチモーダル機能により、Gemini は

異なる形式の情報を処理し統合することができます。特筆すべきは、Google Gemini 1.5 Proが数百万トークンを処理できるコンテキストウィンドウを持っており、これは以前のモデルよりもはるかに大きいということです。この広大なコンテキストウィンドウにより、モデルはより大きなコンテキストを処理でき、潜在的により正確な応答につながります。ただし、Gemini のトークン化方式は他のモデルと比べて 1 文字あたり 1 トークンに近いことに注意が必要です。Google の価格設定はトークンではなく文字数に基づいているため、GPT などのモデルに慣れている場合、Gemini モデルの実際の使用コストは予想よりも大幅に高くなる可能性があります。

トークナイザーの選択は、LLM の以下のような側面に影響を与えます：

- **語彙サイズ：**トークナイザーは、モデルが認識する一意のトークンの集合である語彙のサイズを決定します。より大きく、より細かい粒度の語彙は、モデルがより広範な単語やフレーズを処理し、さらにマルチモーダル（テキスト以外も理解・生成可能）になるのに役立ちますが、モデルのメモリ要件と計算の複雑さも増加します。
- **珍しい単語や未知の単語の処理：**BPEや SentencePiece のようなサブワード単位を使用するトークナイザーは、珍しい単語や未知の単語をより一般的なサブワードに分解できます。これにより、モデルは含まれるサブワードに基づいて、以前に見たことのない単語の意味について教育的な推測を行うことができます。
- **多言語サポート：**SentencePiece のような、Unicode でエンコード可能なあらゆる言語を処理できるトークナイザーは、複数の言語でテキストを処理する必要がある多言語モデルに適しています。

特定のアプリケーションに適した LLM を選択する際は、そのモデルが使用するトークナイザーと、対象タスクの言語処理ニーズにどの程度適合しているかを考慮することが重要です。トークナイザーは、専門用語、希少な単語、多言語テキストの処理能力に大きな影響を与える可能性があります。

コンテキストサイズ：言語モデルは推論時にどれだけの情報を 使用できるか

言語モデルについて議論する際、コンテキストサイズとは、モデルが応答を処理または生成する際に考慮できるテキストの量を指します。これは本質的に、モデルが「記憶」し、出力に活用できる情報量（トークン単位で表現）を示す指標です。言語モデルのコンテキストサイズは、そのモデルの能力と効果的に実行できるタスクの種類に大きな影響を与えます。

コンテキストサイズとは

技術的な観点では、コンテキストサイズは言語モデルが 1 つの入力シーケンスで処理できるトークン（単語または単語の一部）の数によって決定されます。これは、しばしばモデルの「注意範囲」または「コンテキストウィンドウ」と呼ばれます。コンテキストサイズが大きいほど、応答の生成やタスクの実行時に一度に考慮できるテキストの量が増えます。

言語モデルによってコンテキストサイズは異なり、数百トークンから数百万トークンまでさまざまです。参考までに、典型的な段落は約 100-150 トークンを含み、一冊の本全体では数万から数十万トークンを含む可能性があります。

Transformer ベースの大規模言語モデル (LLM) において、**無限に長い入力**をメモリと計算量を制限しながら処理する効率的な方法に関する研究も行われています。

なぜコンテキストサイズが重要なのか

言語モデルのコンテキストサイズは、一貫性のある文脈に即したテキストを理解し生成する能力に大きな影響を与えます。コンテキストサイズが重要な理由は以下の通りです：

1. **長文コンテンツの理解:** コンテキストサイズが大きいモデルは、記事、レポート、さらには本全体といった長いテキストをより適切に理解し分析できます。これは文書要約、質問応答、コンテンツ分析などのタスクにとって重要です。
2. **一貫性の維持:** より大きなコンテキストウィンドウにより、モデルは長い出力全体で一貫性を維持できます。これは物語の生成、対話システム、コンテンツ作成など、一貫した物語やトピックの維持が不可欠なタスクで重要です。また、構造化データの生成や変換に LLM を使用する場合にも絶対に重要です。
3. **長距離依存関係の把握:** 言語タスクの中には、テキスト内で離れた位置にある単語やフレーズ間の関係を理解する必要があるものがあります。コンテキストサイズが大きいモデルは、このような長距離依存関係をより適切に把握できます。これは感情分析、翻訳、言語理解などのタスクで重要となります。
4. **複雑な指示の処理:** 言語モデルを使用して複雑な多段階の指示に従う場合、大きなコンテキストサイズにより、直近の数単語だけでなく、指示全体を考慮して応答を生成することができます。

異なるコンテキストサイズを持つ言語モデルの例

以下に、異なるコンテキストサイズを持つ言語モデルの例を示します:

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

ご覧の通り、これらのモデルのコンテキストサイズには大きな幅があり、OpenAI GPT-3.5 Turbo モデルの約 4,000 トークンから、Anthropic Claude v2 モデルの 200,000 トークンまでさまざまです。Google の PaLM 2 や OpenAI の GPT-4 のような一部のモ

デルでは、より長い入力シーケンスを処理できる大きなコンテキストサイズを持つバリエーション(例:「32k」バージョン)を提供しています。そして現時点(2024 年 4 月)では、Google Gemini Pro は約 300 万トークンを誇っています！

コンテキストサイズは、特定のモデルの実装やバージョンによって異なることに注意が必要です。例えば、オリジナルの OpenAI GPT-4 モデルのコンテキストサイズは 8,191 トークンですが、Turbo や GPT-4o などの後継バージョンでは、128,000 トークンという大幅に大きなコンテキストサイズを持っています。

Sam Altmanは、現在のコンテキストの制限を 80 年代のパーソナルコンピュータのプログラマーが扱わなければならなかったキロバイト単位の作業メモリに例え、近い将来には「すべての個人データ」を大規模言語モデルのコンテキストに収めることができるようになるだろうと述べています。

適切なコンテキストサイズの選択

特定のアプリケーション用に言語モデルを選択する際は、そのタスクに必要なコンテキストサイズの要件を考慮することが重要です。感情分析や単純な質問応答など、短く独立したテキストを扱うタスクでは、小さなコンテキストサイズで十分かもしれません。しかし、より長く複雑なテキストの理解と生成を必要とするタスクには、より大きなコンテキストサイズが必要になるでしょう。

より大きなコンテキストサイズは、モデルが応答を生成する際により多くの情報を考慮する必要があるため、計算コストが増加し処理時間が遅くなることが多いことに注意が必要です。そのため、アプリケーションに適した言語モデルを選択する際は、コンテキストサイズとパフォーマンスのバランスを取る必要があります。

なぜ最大のコンテキストサイズを持つモデルを選んで、できるだけ多くの情報を詰め込まないのでしょうか？パフォーマンス要因の他に、主な考慮事項はコストです。2024 年 3 月の時点で、Google Gemini Pro 1.5 のフルコンテキストを使用した 1 回のプロンプト応答サイクルで、約 8 ドル (USD) かかります。そのような費用が正当化される使用事例があるなら、それは素晴らしいことです！しかし、ほとんどのアプリケーションにとって、桁違いに高額すぎるのです。

干し草の中から針を見つける

干し草の中から針を見つけるという概念は、大規模なデータセットにおける検索の課題を表す比喻として長く使われてきました。LLM の分野では、この比喻を少し変えて考えます。単一の事実 (Paul Graham のエッセイ全集のような膨大なテキストの中に埋もれている) を探すだけでなく、散らばった複数の事実を探す状況を想像してください。これは、1 つの干し草の山ではなく、広大な野原から複数の針を見つけるようなものです。さらに重要なのは、これらの針を見つけるだけでなく、それらを首尾一貫した糸にまとめなければならないということです。

長いコンテキスト内に埋め込まれた複数の事実を検索し推論する際、LLM は 2 つの課題に直面します。まず、検索の正確性という単純な問題があります。事実の数が増えるにつれて、当然ながら正確性は低下します。最も洗練されたモデルでさえ、広範なテキスト全体で複数の詳細を追跡することは負担となるのです。

次に、おそらくより重要な課題は、これらの事実を用いた推論です。事実を見つけ出すことと、それらを首尾一貫した物語や回答に統合することは、まったく別の問題です。ここが本当の試金石となります。推論タスクにおける LLM のパフォーマンスは、単純な検索タスクよりもさらに低下する傾向があります。この低下は単なる量の問題ではなく、文脈、関連性、推論の複雑な相互作用に関係しているのです。

なぜこのような現象が起きるのでしょうか？人間の認知における記憶と注意力のダイナミクスを考えてみましょう。これはある程度 LLM にも反映されています。大量の

情報を処理する際、LLM は人間と同様に、新しい情報を取り入れながら以前の詳細を見失うことがあります。これは特に、テキストの前半部分を自動的に優先したり再確認したりするように明示的に設計されていないモデルで顕著です。

さらに、LLM がこれらの取得した事実を首尾一貫した応答に織り込む能力は、ナラティブ構築に似ています。これには情報の取得だけでなく、深い理解と文脈への適切な配置が必要であり、現在の AI にとって依然として大きな課題となっています。

では、これらの技術の開発者やインテグレーターである私たちにとって、これは何を意味するのでしょうか？複雑な長文タスクを処理する LLM に依存するシステムを設計する際には、これらの制限を十分に認識する必要があります。特定の条件下でパフォーマンスが低下する可能性があることを理解することで、現実的な期待値を設定し、より良いフォールバック機構や補完戦略を構築することができます。

モダリティ：テキストを超えて

現在の言語モデルの大半はテキストの処理と生成に焦点を当てていますが、画像、音声、動画など、複数のタイプのデータをネイティブに入出力できるマルチモーダルモデルへの傾向が高まっています。これらのマルチモーダルモデルは、異なるモダリティ間でコンテンツを理解し生成できる AI アプリケーションの新しい可能性を切り開いています。

モダリティとは何か？

言語モデルの文脈において、モダリティとはモデルが処理・生成できる異なるタイプのデータを指します。最も一般的なモダリティはテキストで、書籍、記事、ウェブサイト、ソーシャルメディアの投稿などの様々な形式の文章が含まれます。しかし、言語モデルに組み込まれるモダリティには他にもいくつかあります：

- **画像**: 写真、イラスト、図表などの視覚データ。
- **音声**: 発話、音楽、環境音などの音声データ。
- **動画**: 多くの場合音声を伴う動画クリップや映画などの動く視覚データ。

各モダリティは言語モデルにとって固有の課題と機会を提示します。例えば、画像では視覚的な概念と関係性の理解が必要であり、音声では発話やその他の音の処理と生成が必要となります。

マルチモーダル言語モデル

マルチモーダル言語モデルは、単一のモデル内で複数のモダリティを扱えるように設計されています。これらのモデルは通常、異なるモダリティの入力を理解し、出力データを生成できる特殊なコンポーネントまたはレイヤーを持っています。マルチモーダル言語モデルの注目すべき例には以下のようなものがあります：

- **OpenAI の GPT-4o：**GPT-4o は、テキストに加えて音声を自然に理解・処理できる大規模言語モデルです。この機能により、音声の文字起こし、音声入力からのテキスト生成、音声クエリに基づく応答の提供などのタスクを実行できます。
- **OpenAI の GPT-4(視覚入力対応)：**GPT-4はテキストと画像の両方を処理できる大規模言語モデルです。画像が入力として与えられると、GPT-4 は画像の内容を分析し、その視覚情報を説明したり、それに応答したりするテキストを生成できます。
- **Google の Gemini：**Geminiはテキスト、画像、動画を扱えるマルチモーダルモデルです。クロスモーダルな理解と生成を可能にする統一アーキテクチャを使用しており、画像キャプション生成、動画要約、視覚的質問応答などのタスクを実現できます。
- **DALL-E と Stable Diffusion：**従来の意味での言語モデルではありませんが、これらのモデルはテキストの説明から画像を生成することで、マルチモーダル AI の力を示しています。異なるモダリティ間の変換が可能なモデルの可能性を示す例となっています。

マルチモーダルモデルの利点とアプリケーション

マルチモーダル言語モデルは以下のような様々な利点があり、幅広いアプリケーションを可能にします：

- ・ **理解力の向上**: 複数のモダリティからの情報を処理することで、人間が様々な感覚入力から学ぶのと同様に、これらのモデルはより包括的な世界理解を得ることができます。
- ・ **クロスモーダル生成**: マルチモーダルモデルは、あるモダリティの入力から別のモダリティのコンテンツを生成できます。例えば、テキストの説明から画像を作成したり、記事から動画のサマリーを生成したりすることができます。
- ・ **アクセシビリティ**: マルチモーダルモデルは、視覚障害者向けに画像のテキスト説明を生成したり、文章のオーディオバージョンを作成したりするなど、モダリティ間の変換によって情報をよりアクセスしやすくすることができます。
- ・ **創造的なアプリケーション**: マルチモーダルモデルは、テキストプロンプトに基づいてアート、音楽、動画を生成するなどの創造的なタスクに使用でき、アーティストやコンテンツクリエイターに新しい可能性を提供します。

マルチモーダル言語モデルは進化を続けており、複数のモダリティにわたってコンテンツを理解・生成できる AI アプリケーションの開発において、ますます重要な役割を果たすようになるでしょう。これにより、人間と AI システムの間でより自然で直感的なインタラクションが可能になり、創造的な表現や知識の普及に向けた新しい可能性が開かれることでしょう。

プロバイダーエコシステム

大規模言語モデル(LLM)をアプリケーションに組み込む際には、選択肢が増え続けています。OpenAI、Anthropic、Google、Cohereなどの主要な LLM プロバイダーは、それぞれ独自のモデル、API、ツールのエコシステムを提供しています。適切なプロバイダーを選択するには、価格、性能、コンテンツフィルタリング、データプライバシー、カスタマイズオプションなど、様々な要因を考慮する必要があります。

OpenAI

OpenAI は最も有名な LLM プロバイダーの 1 つで、その GPT シリーズ (GPT-3、GPT-4) は様々なアプリケーションで広く使用されています。OpenAI は使いやすい API を提供し、アプリケーションへの簡単なモデル統合を可能にします。エントリーレベルの Ada モデルから強力な Davinci モデルまで、異なる機能と価格帯のモデルを提供しています。

OpenAI のエコシステムには、プロンプトの実験や特定のユースケース向けのモデルのファインチューニングができる OpenAI Playground などのツールも含まれています。不適切または有害なコンテンツの生成を防ぐためのコンテンツフィルタリングオプションも提供しています。

OpenAI のモデルを直接使用する際は、Alex Rudall の [ruby-openai](#) ライブラリを利用しています。

Anthropic

Anthropic は LLM 分野のもう一つの主要プレイヤーで、その Claude モデルは高性能と倫理的配慮により人気を集めています。Anthropic は安全で責任ある AI システムの開発に重点を置き、コンテンツフィルタリングと有害な出力の回避に強い emphasis を置いています。

Anthropic のエコシステムには、アプリケーションへのモデル統合を可能にする Claude API や、プロンプトエンジニアリングとファインチューニングのためのツールが含まれています。また、より最新で事実に基づいた応答のためにウェブ検索機能を組み込んだ Claude Instant モデルも提供しています。

Anthropic のモデルを直接使用する際は、Alex Rudall の [anthropic](#) ライブラリを利用しています。

Google

Google は Gemini、BERT、T5、PaLM など、いくつかの強力な LLM を開発しています。これらのモデルは幅広い自然言語処理タスクにおける高い性能で知られています。Google のエコシステムには、機械学習モデルの構築とトレーニングのためのツールとフレームワークを提供する TensorFlow と Keras ライブラリが含まれています。

また、Google は Cloud AI プラットフォームを提供しており、クラウドでモデルを簡単にデプロイしてスケールすることができます。感情分析、エンティティ認識、翻訳などのタスク向けに、様々な事前学習済みモデルと API を提供しています。

Meta

Meta (旧 Facebook) は、LLaMA や OPT などのモデルのリリースに見られるように、大規模言語モデルの開発に深く投資しています。これらのモデルは、様々な言語タスクにおける高いパフォーマンスが特徴であり、主にオープンソースチャネルを通じて提供されており、Meta の研究およびコミュニティとの協力への取り組みを支えています。

Meta のエコシステムは、主に PyTorch を中心に構築されています。PyTorch は、動的な計算機能と柔軟性を備えたオープンソースの機械学習ライブラリで、革新的な AI 研究開発を促進しています。

技術的な提供に加えて、Meta は倫理的な AI 開発に重点を置いています。堅牢なコンテンツフィルタリングを実装し、バイアスの削減に焦点を当てることで、AI アプリケーションにおける安全性と責任という広範な目標に沿った取り組みを行っています。

Cohere

Cohere は、LLM 分野の新興企業で、競合他社よりも LLM をよりアクセスしやすく、使いやすくすることに焦点を当てています。そのエコシステムには、テキスト生成、分類、要約などのタスク向けの事前学習済みモデルにアクセスできる Cohere API が含まれています。

Cohere は、プロンプトエンジニアリング、ファインチューニング、コンテンツフィルタリングのためのツールも提供しています。データの暗号化保存やアクセス制御など、データのプライバシーとセキュリティを重視しています。

Ollama

Ollama は、セルフホスト型プラットフォームで、ユーザーが外部のクラウドサービスに依存することなく、様々な大規模言語モデル(LLM)をローカルマシンで管理・展開できるようにします。この設定は、データプライバシーを重視し、AI 運用を社内で処理したい人々に最適です。

このプラットフォームは、Llama、Phi、Gemma、Mistralなどのバージョンを含む様々なモデルをサポートしており、これらは規模と計算要件が異なります。Ollama は、`ollama run <model_name>` のような単純なコマンドを使用してコマンドラインから直接これらのモデルをダウンロードして実行することを容易にし、macOS、Linux、Windows などの異なるオペレーティングシステムで動作するように設計されています。

リモート API を使用せずにオープンソースモデルをアプリケーションに統合したい開発者向けに、Ollama はコンテナ管理ツールに似たモデルライフサイクルを管理するための CLI を提供しています。また、カスタム設定とプロンプトをサポートし、特定のニーズやユースケースに合わせてモデルを調整できる高度なカスタマイズ性を提供します。

Ollama は、コマンドラインインターフェースと AI モデルの管理・展開における柔軟性により、特に技術に精通したユーザーや開発者に適しています。これにより、セキュリティと制御を損なうことなく、堅牢な AI 機能を必要とする企業や個人にとって強力なツールとなっています。

マルチモデルプラットフォーム

さらに、Together.ai や Groq など、様々なオープンソースモデルをホストするプロバイダーも存在します。これらのプラットフォームは柔軟性とカスタマイズ性を提供し、特

定のニーズに応じてオープンソースモデルを実行し、場合によってはファインチューニングすることも可能です。例えば、Together.ai は様々なオープンソース LLM へのアクセスを提供し、ユーザーが異なるモデルと設定を試すことを可能にします。Groq は、この本の執筆時点でほぼ魔法のように見える超高性能な文章生成に焦点を当てています。

LLM プロバイダーの選択

LLM プロバイダーを選択する際は、以下のような要因を考慮する必要があります：

- ・ **価格設定**：プロバイダーによって、従量制から定額制まで、様々な価格モデルを提供しています。プロバイダーを選択する際は、予想される使用量と予算を考慮することが重要です。
- ・ **パフォーマンス**：LLM のパフォーマンスはプロバイダー間で大きく異なる可能性があるため、決定を下す前に特定のユースケースでモデルをベンチマークしてテストすることが重要です。
- ・ **コンテンツフィルタリング**：アプリケーションによっては、コンテンツフィルタリングが重要な考慮事項となる場合があります。プロバイダーによって、より堅牢なコンテンツフィルタリングオプションを提供しているところもあります。
- ・ **データプライバシー**：アプリケーションが機密性の高いユーザーデータを扱う場合、強力なデータプライバシーとセキュリティ対策を持つプロバイダーを選択することが重要です。
- ・ **カスタマイズ**：一部のプロバイダーは、特定のユースケースに向けたモデルのファインチューニングとカスタマイズにおいて、より高い柔軟性を提供しています。

最終的に、LLM プロバイダーの選択はアプリケーションの具体的な要件と制約に依存します。価格、性能、データプライバシーなどの要因を慎重に評価することで、ニーズに最も適したプロバイダーを選択することができます。

また、LLM の分野は常に進化しており、新しいプロバイダーやモデルが定期的に登場していることにも注目する価値があります。最新の動向を把握し、新しい選択肢が利用可能になった際には、それらを積極的に検討する姿勢を持つべきです。

OpenRouter

本書では、API プロバイダーとして[OpenRouter](#)のみを使用します。理由は単純で、最も人気のある商用およびオープンソースモデルを一か所で提供しているからです。AI コーディングに早速取り組みたい方には、私が開発した[OpenRouter Ruby Library](#)が最適な出発点の一つとなるでしょう。

パフォーマンスについて考える

言語モデルをアプリケーションに組み込む際、パフォーマンスは重要な考慮事項です。言語モデルのパフォーマンスは、レイテンシー(応答を生成するまでの時間)とスループット(単位時間あたりに処理できるリクエスト数)で測定できます。

最初のトークンまでの時間(TTFT)は、特にチャットボットやリアルタイムの対話型応答を必要とするアプリケーションにおいて、もう一つの重要なパフォーマンス指標です。TTFT は、ユーザーのリクエストを受信してから応答の最初の単語(またはトークン)が生成されるまでのレイテンシーを測定します。この指標は、応答の遅延がユーザーのフラストレーションや離脱につながる可能性があるため、シームレスで魅力的なユーザー体験を維持する上で極めて重要です。

これらのパフォーマンス指標は、ユーザー体験とアプリケーションのスケーラビリティに大きな影響を与える可能性があります。

言語モデルのパフォーマンスに影響を与える要因には、以下のようなものがあります：

パラメータ数：より多くのパラメータを持つ大規模なモデルは、一般的により多くの計算リソースを必要とし、小規模なモデルと比較してレイテンシーが高く、スループットが低くなる可能性があります。

ハードウェア: 言語モデルのパフォーマンスは、実行されるハードウェアによって大きく異なります。クラウドプロバイダーは機械学習ワークロード向けに最適化された GPU や TPU インスタンスを提供しており、これらはモデルの推論を大幅に高速化できます。



OpenRouterの優れている点の一つは、提供している多くのモデルについて、様々なパフォーマンスプロファイルとコストを持つクラウドプロバイダーの選択肢があることです。

量子化: 量子化技術は、重みと活性化を低精度のデータ型で表現することで、モデルのメモリフットプリントと計算要件を削減できます。これにより、品質を大きく犠牲にすることなくパフォーマンスを向上させることができます。アプリケーション開発者として、おそらく異なる量子化レベルでの独自のモデルトレーニングには関与しないでしょうが、少なくとも用語に慣れておくことは良いことです。

バッチ処理: 複数のリクエストを同時にバッチで処理することで、モデルの読み込みとデータ転送のオーバーヘッドを分散し、スループットを向上させることができます。

キャッシング: 頻繁に使用されるプロンプトや入力シーケンスの結果をキャッシュすることで、推論リクエストの数を削減し、全体的なパフォーマンスを向上させることができます。

本番アプリケーション用の言語モデルを選択する際は、代表的なワークロードとハードウェア構成でパフォーマンスをベンチマークすることが重要です。これにより、潜在的なボトルネックを特定し、必要なパフォーマンス目標を達成できることを確認できます。

また、モデルのパフォーマンスと、コスト、柔軟性、統合の容易さなどの他の要因とのトレードオフを考慮することも重要です。例えば、リアルタイムの応答を必要とするアプリケーションでは、レイテンシーの低い小規模で安価なモデルの使用が望ましい場合があり、一方、バッチ処理や複雑な推論タスクには、より大規模で強力なモデルが適している場合があります。

異なる LLM モデルの実験

大規模言語モデルの選択は、通常、永続的な決定ではありません。新しく改良されたモデルが定期的にはリリースされるため、時間の経過とともに異なる言語モデルを切り替えることができる、モジュール式のアプリケーションを構築することが望ましいです。プロンプトやデータセットは、最小限の変更で複数のモデル間で再利用できることが多いです。これにより、アプリケーションを完全に再設計することなく、言語モデリングの最新の進歩を活用することができます。



幅広いモデルの選択肢を簡単に切り替えられる機能は、私が OpenRouter を愛する理由の一つです。

新しい言語モデルにアップグレードする際は、アプリケーションの要件を満たすために、その性能と出力品質を徹底的にテストし、検証することが重要です。これには、ドメイン固有のデータでモデルを再トレーニングまたは微調整することや、モデルの出力に依存する下流コンポーネントを更新することが含まれる場合があります。

パフォーマンスとモジュール性を念頭に置いてアプリケーションを設計することで、言語モデリング技術の急速に進化する状況に適応できる、スケーラブルで効率的、そして将来性のあるシステムを作成することができます。

複合 AI システム

導入部分を締めくくる前に、2023 年以前と ChatGPT によって引き起こされた生成 AI への関心の爆発以前は、従来の AI アプローチは通常、単一の閉じたモデルの統合に依存していたことを言及する価値があります。対照的に、複合 AI システムは、インテリジェントな動作を実現するために協調して動作する、相互接続されたコンポーネントの複雑なパイプラインを活用します。

複合 AI システムの核心は、特定のタスクや機能を実行するように設計された複数のモジュールで構成されています。これらのモジュールには、ジェネレーター、検索

機、ランカー、分類器、その他の様々な専門化されたコンポーネントが含まれます。全体的なシステムをより小さな、焦点を絞ったユニットに分解することで、開発者はより柔軟で、スケーラブルで、保守可能な AI アーキテクチャを作成できます。

複合 AI システムの主な利点の一つは、異なる AI 技術とモデルの長所を組み合わせる能力です。例えば、システムは自然言語の理解と生成に大規模言語モデル (LLM) を使用し、情報検索やルールベースの意思決定には別のモデルを採用することができます。このモジュール式のアプローチにより、一つのサイズですべてに対応するソリューションに頼るのではなく、各特定のタスクに最適なツールと技術を選択することができます。

しかし、複合 AI システムの構築には、独自の課題もあります。特に、システムの動作全体の一貫性と整合性を確保するには、堅牢なテスト、モニタリング、およびガバナンスメカニズムが必要です。



GPT-4のような強力な LLM の出現により、これまでになく簡単に複合 AI システムを実験できるようになりました。これらの高度なモデルは、自然言語理解機能に加えて、分類、ランキング、生成など、複合システム内の複数の役割を処理できるためです。この多用途性により、開発者は複合 AI アーキテクチャを急速にプロトタイプ化し、反復することができ、インテリジェントなアプリケーション開発の新しい可能性を開きます。

複合 AI システムのデプロイメントパターン

複合 AI システムは、特定の要件とユースケースに対応するように設計された様々なパターンを使用してデプロイできます。質問応答、マルチエージェント/エージェント型問題解決システム、対話型 AI、およびコパイロットという 4 つの一般的なデプロイメントパターンを見ていきましょう。

質問応答

質問応答(Q&A)システムは、単なる検索エンジン以上の機能を果たすために、AI モデルの理解能力で強化された情報検索の提供に焦点を当てています。強力な言語モデルを**検索拡張生成(RAG)**を使用して外部の知識ソースと組み合わせることで、質問応答システムは幻覚を回避し、ユーザーの質問に対して正確で文脈に関連した応答を提供します。

LLM ベースの Q&A システムの主要コンポーネントには以下が含まれます：

- ・ **クエリ理解と再構築**：ユーザーのクエリを分析し、基盤となる知識ソースにより適合するように再構築します。
- ・ **知識検索**：再構築されたクエリに基づいて、構造化または非構造化データソースから関連情報を検索します。
- ・ **応答生成**：検索された知識と言語モデルの生成機能を組み合わせて、一貫性があり有益な応答を生成します。

RAG サブシステムは、カスタマーサポート、ナレッジマネジメント、教育アプリケーションなど、正確で最新の情報提供が重要な Q&A 領域で特に重要です。

マルチエージェント/エージェント型問題解決システム

マルチエージェント(別名エージェント型)システムは、複雑な問題を解決するために協力して働く複数の自律エージェントで構成されています。各エージェントは特定の役割、スキルセット、および関連ツールや情報源へのアクセス権を持っています。これらのエージェントが協力して情報を交換することで、単一のエージェントでは対処が困難または不可能な課題に取り組むことができます。

マルチエージェント問題解決システムの主要な原則には以下が含まれます：

- ・ **専門化**：各エージェントは、独自の機能と知識を活用して、問題の特定の側面に焦点を当てます。

- ・ **協調:** エージェントは、メッセージパッシングや共有メモリを通じて、共通の目標を達成するために通信し、行動を調整します。
- ・ **適応性:** システムは、個々のエージェントの役割と行動を調整することで、変化する条件や要件に適応できます。

マルチエージェントシステムは、サプライチェーン最適化、交通管理、緊急時対応計画など、分散型の問題解決が必要なアプリケーションに適しています。

対話型 AI

対話型 AI システムは、ユーザーとインテリジェントエージェント間の自然言語によるインタラクションを可能にします。これらのシステムは、自然言語理解、対話管理、言語生成機能を組み合わせて、魅力的でパーソナライズされた会話体験を提供します。

対話型 AI システムの主要なコンポーネントには以下が含まれます：

- ・ **意図認識:** 質問、要求、感情表現などの、ユーザーの入力に基づく意図を識別します。
- ・ **エンティティ抽出:** 日付、場所、製品名など、ユーザーの入力から関連するエンティティやパラメータを抽出します。
- ・ **対話管理:** 会話の状態を維持し、ユーザーの意図とコンテキストに基づいて適切な応答を決定し、複数ターンのインタラクションを処理します。
- ・ **応答生成:** 言語モデル、テンプレート、検索ベースの手法を使用して、人間らしい応答を生成します。

対話型 AI システムは、カスタマーサービスチャットボット、バーチャルアシスタント、音声制御インターフェースでよく使用されています。前述のように、本書のアプローチ、パターン、コード例のほとんどは、[Olympia](#)と呼ばれる大規模な対話型 AI システムでの私の仕事から直接抽出されています。

コパイロット

コパイロットは、ユーザーの生産性と意思決定能力を向上させるために、人間のユーザーと協力して働く AI 駆動のアシスタントです。これらのシステムは、自然言語処理、機械学習、ドメイン固有の知識を組み合わせ、インテリジェントな推奨事項の提供、タスクの自動化、文脈に応じたサポートを行います。

コパイロットの主要な機能には以下が含まれます：

- ・ **パーソナライゼーション**：個々のユーザーの好み、ワークフロー、コミュニケーションスタイルへの適応。
- ・ **プロアクティブな支援**：明示的な指示がなくても、ユーザーのニーズを予測し、関連する提案やアクションを提供。
- ・ **継続的な学習**：ユーザーのフィードバック、インタラクション、データから学習することで、時間とともにパフォーマンスを向上。

コパイロットは、ソフトウェア開発（コード補完やバグ検出など）、クリエイティブライティング（コンテンツ提案や編集など）、データ分析（インサイトや可視化の推奨など）など、さまざまな分野で increasingly 使用されています。

これらのデプロイメントパターンは、複合 AI システムの汎用性と可能性を示しています。それぞれのパターンの特徴とユースケースを理解することで、インテリジェントなアプリケーションの設計と実装において、十分な情報に基づいた意思決定を行うことができます。本書は複合 AI システムの実装に特化したものではありませんが、従来型のアプリケーション開発内で個別の AI コンポーネントを統合する際にも、同様のアプローチやパターンの多く（もしくはすべて）が適用できます。

複合 AI システムにおける役割

複合 AI システムは、特定の役割を果たすように設計された相互接続されたモジュールの基盤の上に構築されています。これらのモジュールが協力して、インテリジェントな振る舞いを生み出し、複雑な問題を解決します。アプリケーションのどの部分を

AI コンポーネントで実装または置き換えることができるかを検討する際には、これらの役割を理解しておくで役立ちます。

ジェネレーター

ジェネレーターは、学習したパターンや入力プロンプトに基づいて、新しいデータやコンテンツを生成する役割を担います。AI 分野には多くの種類のジェネレーターが存在しますが、本書で紹介する言語モデルの文脈では、ジェネレーターは人間のような文章を作成したり、文章の一部を補完したり、ユーザーの質問に対する回答を生成したりすることができます。コンテンツ作成、対話生成、データ拡張などのタスクにおいて重要な役割を果たします。

リトリバー

リトリバーは、大規模なデータセットや知識ベースから関連情報を検索して抽出するために使用されます。意味検索、キーワードマッチング、ベクトル類似度などの技術を用いて、与えられたクエリやコンテキストに基づいて最も適切なデータポイントを見つけ出します。リトリバーは、質問応答、ファクトチェック、コンテンツ推薦など、特定の情報への迅速なアクセスが必要なタスクに不可欠です。

ランカー

ランカーは、特定の基準や関連性スコアに基づいてアイテムのセットを順序付けまたは優先順位付けする役割を担います。各アイテムに重みやスコアを割り当て、それに従って並び替えを行います。ランカーは、検索エンジン、推薦システム、またはユーザーに最も関連性の高い結果を提示する必要があるアプリケーションで一般的に使用されます。

分類器

分類器は、事前に定義されたクラスやカテゴリに基づいてデータポイントを分類またはラベル付けするために使用されます。ラベル付けされたトレーニングデータから学

習し、新しい未知のインスタンスのクラスを予測します。分類器は、感情分析、スパム検出、画像認識など、各入力に特定のカテゴリを割り当てることが目的のタスクにおいて基本的な役割を果たします。

ツールとエージェント

これらの中核的な役割に加えて、複合 AI システムはしばしばツールやエージェントを組み込んで、その機能性と適応性を向上させます：

- ・ **ツール:** ツールは、特定のアクションや計算を実行する個別のソフトウェアコンポーネントまたは API です。ジェネレーターやリトリーバーなど、他のモジュールによって呼び出され、サブタスクを実行したり追加情報を収集したりします。ツールの例としては、ウェブ検索エンジン、計算機、データ可視化ライブラリなどがあります。
- ・ **エージェント:** エージェントは、環境を認識し、決定を下し、特定の目標を達成するために行動を起こすことができる自律的なエンティティです。多くの場合、プランニング、推論、学習などの異なる AI 技術を組み合わせて、動的または不確実な条件下で効果的に動作します。エージェントは、複雑な振る舞いをモデル化したり、複合 AI システム内の複数のモジュールのアクションを調整したりするために使用できます。

純粋な複合 AI システムでは、これらのコンポーネント間の相互作用は、明確に定義されたインターフェースと通信プロトコルを通じて調整されます。データはモジュール間を流れ、あるコンポーネントの出力が別のコンポーネントの入力として機能します。このモジュラーアーキテクチャにより、個々のコンポーネントをシステム全体に影響を与えることなく更新、置換、または拡張できるため、柔軟性、スケーラビリティ、保守性が確保されます。

これらのコンポーネントとその相互作用の力を活用することで、複合 AI システムは異なる AI 機能の組み合わせを必要とする複雑な実世界の問題に取り組むことができます。AI をアプリケーション開発に統合するためのアプローチやパターンを探る際

には、複合 AI システムで使用されているのと同じ原則や技術を、インテリジェントで適応性が高く、ユーザー中心のアプリケーションを作成するために適用できることを覚えておいてください。

パート 1 の以降の章では、AI コンポーネントをアプリケーション開発プロセスに統合するための基本的なアプローチと技術についてより深く掘り下げていきます。プロンプトエンジニアリングや検索強化生成から、自己修復データやインテリジェントなワークフロー・オーケストレーションまで、最先端の AI 駆動アプリケーションを構築するための幅広いパターンとベストプラクティスを取り上げていきます。

パート 1：基本的なアプローチ と技法

本書のこのパートでは、アプリケーションに AI を統合するための様々な方法を紹介します。各章では、[パスの絞り込み](#)や[検索拡張生成](#)といった高レベルな概念から、LLM チャット補完 API の上に独自の抽象化層をプログラミングするためのアイデアまで、関連するアプローチと技法を幅広く取り上げます。

このパートの目的は、[パート 2](#)で焦点を当てる具体的な実装パターンに深入りする前に、AI で実現できる動作の種類を理解していただくことです。

パート 1 で紹介するアプローチは、私が自身のコードで使用してきたアイデア、エンタープライズアプリケーションアーキテクチャと統合の古典的なパターン、そして非技術系のビジネス関係者を含む他の人々に AI の機能を説明する際に用いてきた比喩に基づいています。

パスを狭める



「パスを狭める」とは、AIを目の前のタスクに集中させることを指します。AIが「馬鹿」のように見えたり、予期せぬ動作をしたりして苛立ちを感じる時はいつでも、私はこれを合言葉にしています。この合言葉は、失敗の原因はおそらく私にあり、もっとパスを狭める必要があることを思い出させてくれるのです。

パスを狭める必要性は、大規模言語モデルに含まれる膨大な知識量に起因します。特に OpenAIや Anthropicのような世界最高峰のモデルは、文字通り数兆のパラメータを持っています。

このような幅広い知識へのアクセスは間違いなく強力で、心の理論や人間のような推論能力といった創発的な振る舞いを生み出します。しかし、その途方もない量の情報は、特定のプロンプトに対して正確で精密な応答を生成する際に課題となります。特に、「通常の」ソフトウェア開発やアルゴリズムと統合できる決定論的な振る舞いを期待する場合はなおさらです。

これらの課題には、いくつかの要因があります。

**** 情報過多:** 大規模言語モデルは、様々な領域、ソース、時期にわたる膨大なデータで訓練されています。この広範な知識により、多様なトピックに対応し、世界についての幅広い理解に基づいて応答を生成することができます。しかし、特定のプロンプトに直面した際、モデルは無関係、矛盾、または古い・廃れた情報を除外することに苦勞する可能性があり、それが焦点の定まっていない、または不正確な応答につながります。目的によっては、モデルが利用可能な矛盾する情報の膨大な量が、求める回答や振る舞いを提供する能力を簡単に圧倒してしまうことがあります。

**** 文脈の曖昧さ:** 膨大な潜在空間の知識を考えると、大規模言語モデルはプロンプトの文脈を理解しようとする際に曖昧さに直面する可能性があります。適切な絞り込みやガイダンスがなければ、モデルは意図に関連するものの、直接的には関係のない応答を生成するかもしれません。このような失敗は、的外れ、一貫性のない、またはあなたの述べたニーズに対応していない応答につながります。この場合、パスを狭めるとは、提供する文脈が基礎知識の中で最も関連性の高い情報にのみモデルを集中させるような文脈の曖昧さの解消を指します。



注:「プロンプトエンジニアリング」を始めたばかりの頃は、望む結果を適切に説明せずにモデルに何かを依頼してしまいがちです。曖昧さを避けるには練習が必要です!

**** 時間的不整合:** 言語モデルは異なる時期に作成されたデータで訓練されているため、古くなった、置き換えられた、または最新ではない知識を持っている可能性があります。例えば、時事問題、科学的発見、または技術の進歩に関する情報は、モデルの訓練データが収集された時点から進化している可能性があります。より最新で

信頼できるソースを優先するようにパスを狭めなければ、モデルは古い、または不正確な情報に基づいて応答を生成し、出力に不正確さや矛盾が生じる可能性があります。

**** ドメイン固有のニュアンス:**** 異なる領域や分野には、それぞれ固有の専門用語、慣習、知識基盤があります。ほとんどの 3 文字略語 (TLA) について考えてみれば、その多くが複数の意味を持っていることに気付くでしょう。例えば、MSK は Amazon の Managed Streaming for Apache Kafka、Memorial Sloan Kettering Cancer Center、または人体の筋骨格系 (MusculoSkeletal system) を指す可能性があります。

プロンプトが特定の分野での専門知識を必要とする場合、大規模言語モデルの一般的な知識だけでは、正確で細やかな応答を提供するには不十分かもしれません。プロンプトエンジニアリングや検索拡張生成を通じて、特定分野の情報に焦点を当てることで経路を絞り込むことにより、モデルは特定の分野の要件や期待により沿った応答を生成できるようになります。

潜在空間：理解を超えた広大さ

「潜在空間」について言及する際、私が指しているのは、モデルが学習プロセスの間に学んだ知識と情報の広大な多次元の領域のことです。これは、モデルのニューラルネットワーク内に隠された領域のようなもので、言語のパターン、関連性、表現がすべて格納されています。

無数の相互接続されたノードで満たされた広大な未踏の領域を探検しているところを想像してみてください。各ノードは、モデルが学習した情報の断片、概念、または関係性を表しています。この空間を進んでいくと、一部のノードは互いに近接しており、強い結びつきや類似性を示している一方で、他のノードは離れており、より弱いまたは遠い関係性を示していることがわかります。

潜在空間の課題は、それが信じられないほど複雑で高次元だということです。私たちの物理的な宇宙のように広大で、銀河の集団と、その間に広がる想像を絶する空虚な空間があるようなものだと考えてください。

何千もの次元を含むため、潜在空間は人間が直接観察したり解釈したりすることはできません。これは、モデルが言語を処理し生成するために内部で使用する抽象的な表現です。モデルにプロンプトを入力すると、本質的にそのプロンプトを潜在空間内の特定の位置にマッピングします。その後、モデルはその空間内の周囲の情報とつながりを使用して応答を生成します。

重要なのは、モデルが学習データから膨大な量の情報を学習しており、その全てが特定のタスクに関連していたり正確だったりするわけではないということです。そのため、経路を絞り込むことが非常に重要になります。プロンプトで明確な指示、例、文脈を提供することで、本質的には潜在空間内の、望む出力に最も関連する特定の領域にモデルの焦点を当てるよう導いているのです。

別の考え方をすれば、完全に暗い美術館でスポットライトを使用するようなものです。もしルーブル美術館やメトロポリタン美術館を訪れたことがあれば、私が話している規模がわかるでしょう。潜在空間は、無数の物体や細部で満たされた美術館のようなものです。プロンプトはスポットライトのようなもので、特定の領域を照らし、最も重要な情報にモデルの注意を向けさせます。そのような導きがなければ、モデルは潜在空間内をあてもなく彷徨い、関係のない、あるいは矛盾する情報を拾い集めてしまうかもしれません。

言語モデルを使用してプロンプトを作成する際は、潜在空間の概念を念頭に置いてください。あなたの目標は、この広大な知識の領域を効果的に進み、タスクに最も関連性が高く正確な情報へとモデルを導くことです。経路を絞り込み、明確な導きを提供することで、モデルの潜在空間の可能性を最大限に引き出し、質の高い、一貫性のある応答を生成することができます。

これまでの言語モデルと潜在空間の説明は、少し魔法のような、あるいは抽象的なものに思えるかもしれませんが、プロンプトは呪文や祈祷ではないということを理解することが重要です。言語モデルの仕組みは、線形代数と確率論の原理に基づいています。

本質的に、言語モデルは確率モデルであり、ベル曲線がデータの統計モデルであるのと同じように、テキストの確率モデルなのです。これらは自己回帰モデリングと呼ば

れるプロセスを通じて学習され、モデルはそれまでの単語に基づいて、シーケンス内の次の単語の確率を予測することを学習します。学習中、モデルはランダムな重みから始まり、徐々にそれらを調整して、学習に使用された実世界のサンプルに似たテキストにより高い確率を割り当てていきます。

しかし、言語モデルを線形回帰のような単純な統計モデルとして考えることは、その振る舞いを理解する上で最適な直感を与えてくれるわけではありません。より適切な類似として、確率変数の操作を可能にし、複雑な統計的関係を表現できる確率的プログラムとして考えることができます。

確率的プログラムは、グラフィカルモデルとして表現することができます。これにより、モデル内の変数間の依存関係や関連性を視覚的に理解することができます。この視点は、GPT-4や Claudeのような複雑なテキスト生成モデルの仕組みを理解する上で、貴重な洞察を提供することができます。

Dohan 他による論文「Language Model Cascades」では、確率的プログラムを言語モデルにどのように適用できるかについて詳しく説明しています。著者らは、このフレームワークを使用してこれらのモデルの動作を理解し、より効果的なプロンプティング戦略の開発を導く方法を示しています。

この確率的な視点から得られる重要な洞察の 1 つは、言語モデルが本質的に、望ましい文書が存在する別の宇宙への入り口を作り出すということです。モデルは、すべての可能な文書に対して確率に基づいて重みを割り当て、効果的に可能性の空間を最も関連性の高いものに絞り込みます。

これは「パスを絞り込む」という中心的なテーマに立ち返ることになります。プロンプティングの主な目的は、確率モデルの予測の質を集中させ、引き出したい特定の情報や振る舞いに焦点を当てるように条件付けることです。注意深く作成されたプロンプトを提供することで、モデルがより効率的に潜在空間を探索し、より関連性が高く一貫性のある出力を生成するよう導くことができます。

ただし、言語モデルは最終的に学習データに含まれる情報によって制約されることを忘れてはいけません。既存の文書に似たテキストを生成したり、アイデアを新しい方法で組み合わせたりすることはできますが、まったく新しい情報をゼロから生み出す

ことはできません。たとえば、がんの治療法が発見されておらず、学習データに記録されていない場合、モデルにがんの治療法を提供するよう期待することはできません。

代わりに、モデルの強みは、プロンプトで与えられた内容に類似する情報を見つけ出し、統合する能力にあります。これらのモデルの確率的な性質と、プロンプトを使用して出力を条件付けする方法を理解することで、価値のある洞察やコンテンツを生成するためにその機能をより効果的に活用することができます。

以下のプロンプトを考えてみましょう。最初の「Mercury」だけでは、惑星、元素、ローマの神のいずれかを指す可能性があります、最も可能性が高いのは惑星です。実際、GPT-4 は「水星は太陽系で最小かつ最も内側の惑星です...」で始まる長い応答を提供します。² 2 番目のプロンプトは特に化学元素を指しています。³ 3 番目のプロンプトは、その速さと神の使者としての役割で知られるローマ神話の人物を指しています。

- 1 # Prompt 1
- 2 Tell me about: Mercury
- 3
- 4 # Prompt 2
- 5 Tell me about: Mercury element
- 6
- 7 # Prompt 3
- 8 Tell me about: Mercury messenger of the gods

わずかな言葉を追加するだけで、AI の反応は完全に変わってしまいます。この本の後半で学ぶように、n-shot プロンプティングや構造化された入出力、[思考連鎖](#)などの高度なプロンプトエンジニアリングの技法は、モデルの出力を調整する巧妙な方法に過ぎません。

つまり、プロンプトエンジニアリングの本質は、言語モデルの知識という広大な確率的な領域を理解し、求める情報や振る舞いへの道筋を絞り込む方法を知ることにあります。

上級数学に精通している読者の方々は、確率論と線形代数の原理に基づいてこれらのモデルを理解することで、より深い洞察が得られるでしょう!それ以外の方々に、効果的な出力を引き出す戦略を学びたい方は、より直感的なアプローチに焦点を当てていきましょう。

どのように道筋が「絞り込まれる」のか

膨大な知識がもたらす課題に対処するため、私たちは言語モデルの生成プロセスを導き、最も関連性が高く正確な情報に注目させる技術を使用します。

以下が最も重要な技術を推奨順に並べたものです。つまり、最初にプロンプトエンジニアリングを試し、次に RAG、そして必要な場合にのみファインチューニングを行うべきです。

プロンプトエンジニアリング 最も基本的なアプローチは、モデルの応答生成を導くための具体的な指示、制約、または例を含むプロンプトを作成することです。この章では次のセクションでプロンプトエンジニアリングの基礎を扱い、本のパート 2 で多くの具体的なプロンプトエンジニアリングのパターンを紹介します。それらのパターンには**プロンプト蒸留**が含まれており、これは AI が最も関連性が高く簡潔だと考える情報を抽出するためにプロンプトを洗練し最適化する技術です。

コンテキスト拡張 プロンプトを与える時点で、外部の知識ベースや文書から関連情報を動的に取得し、モデルに焦点を絞ったコンテキストを提供します。一般的なコンテキスト拡張技術には**検索拡張生成 (RAG)**があります。**Perplexity**が提供するような「オンラインモデル」は、リアルタイムのインターネット検索結果でコンテキストを拡張することができます。



LLM はその能力の高さにもかかわらず、あなた固有のデータセット(プライベートなものや、解決しようとしている問題に特有のもの)では訓練されていません。コンテキスト拡張技術を使用することで、LLM に API 経由のデータ、SQL データベース内のデータ、または PDF やスライドデッキに閉じ込められたデータへのアクセスを提供することができます。

ファインチューニングまたはドメイン適応特定のタスクや分野に特化した知識と生成能力を持たせるため、ドメイン固有のデータセットでモデルを訓練します。

温度を下げる

温度は、トランスフォーマーベースの言語モデルで使用されるハイパーパラメータで、生成されるテキストのランダム性と創造性を制御します。0 から 1 の間の値を取り、低い値ではより焦点を絞った決定論的な出力になり、高い値ではより多様で予測不可能な出力になります。

温度が 1 に設定されている場合、言語モデルは次のトークンの完全な確率分布に基づいてテキストを生成し、より創造的で多様な応答を可能にします。ただし、これによってモデルが関連性や一貫性の低いテキストを生成する可能性も出てきます。

一方、温度を 0 に設定すると、言語モデルは常に確率が最も高いトークンを選択し、効果的に「道筋を絞り込む」ことになります。私の AI コンポーネントのほとんどすべてで、温度を 0 またはそれに近い値に設定しています。これは、より焦点を絞った予測可能な応答が得られるためです。モデルに指示に従わせたい場合や、提供された関数に注意を払わせたい場合、あるいは単に現状よりも正確で関連性の高い応答が必要な場合に特に有用です。

例えば、事実に基づいた情報を提供する必要があるチャットボットを構築している場合、より正確で的確な応答を確保するために、温度パラメータを低く設定したいかもしれません。反対に、創造的な文章作成アシスタントを構築している場合は、より多様で想像力豊かな出力を促すために、温度パラメータを高く設定したいかもしれません。

ハイパーパラメータ：推論の調整つまみとダイヤル

言語モデルを扱う際、「ハイパーパラメータ」という用語によく遭遇します。推論（つまり、モデルを使用して応答を生成する時）の文脈では、ハイパーパラメータはモデルの動作と出力を制御できる調整つまみやダイヤルのようなものです。

複雑な機械の設定を調整するようなものだと考えてください。温度を制御するためにつまみを回したり、動作モードを変更するためにスイッチを切り替えたりするように、ハイパーパラメータを使用することで、言語モデルがテキストを処理・生成する方法を細かく調整できます。

推論時によく遭遇するハイパーパラメータには以下のようなものがあります：

- **温度**：先ほど述べたように、このパラメータは生成されるテキストのランダム性と創造性を制御します。温度が高いほど、より多様で予測不可能な出力が生成され、温度が低いほど、より焦点を絞った決定論的な応答が得られます。
- **Top-p(nucleus) サンプリング**：このパラメータは、累積確率が特定の閾値(p)を超える最小のトークン集合の選択を制御します。これにより、一貫性を保ちながらより多様な出力が可能になります。
- **Top-k サンプリング**：この手法は、次に来る可能性が最も高い上位 k 個のトークンを選択し、それらの間で確率質量を再配分します。これにより、モデルが確率の低い、または無関係なトークンを生成するのを防ぐことができます。
- **頻度と存在ペナルティ**：これらのパラメータは、同じ単語やフレーズを頻繁に繰り返すこと(頻度ペナルティ)や、入力プロンプトに存在しない単語を生成すること(存在ペナルティ)に対してペナルティを与えます。これらの値を調整することで、モデルがより多様で関連性の高い出力を生成するよう促すことができます。
- **最大長**：このハイパーパラメータは、モデルが 1 回の応答で生成できるトークン(単語または部分語)の数の上限を設定します。これにより、生成されるテキストの冗長性や簡潔さを制御できます。

異なるハイパーパラメータ設定を試してみると、わずかな調整でもモデルの出力に大きな影響を与えることがわかるでしょう。まるでレシピの微調整のようなものです - 塩

を少し多めに入れたり、調理時間を少し長くしたりするだけで、最終的な料理が大きく変わることがあります。

重要なのは、各ハイパーパラメータがモデルの動作にどのように影響するかを理解し、特定のタスクに適したバランスを見つけることです。異なる設定を試してみて、生成されるテキストにどのような影響があるか確認することを恐れずにください。時間とともに、どのハイパーパラメータを調整すべきか、望む結果を得るにはどうすればよいかという直感が養われていきます。

これらのパラメータの使用とプロンプトエンジニアリング、検索拡張生成、ファインチューニングを組み合わせることで、特定のユースケースに対してより正確で関連性が高く、価値のある応答を生成するよう言語モデルを効果的に導くことができます。

生のモデル対指示調整済みモデル

生のモデルは、LLM の洗練されていない、訓練されていないバージョンです。まだ指示を理解したり従ったりするための特定の訓練の影響を受けていない、真っ白なキャンバスのようなものと想像してください。これらは最初に訓練された膨大なデータに基づいて構築され、幅広い出力を生成する能力を持っています。しかし、指示ベースのファインチューニングの追加層がないため、その応答は予測不可能で、望ましい出力に導くためにはより繊細で慎重に作られたプロンプトが必要となります。生のモデルを扱うことは、膨大な知識を持っているものの、極めて正確な指示を与えない限り、あなたが何を求めているのかについての直感をまったく持たないサヴァン症候群の人とのコミュニケーションを引き出すようなものです。彼らはしばしばオウムのように感じられます。というのも、理解できる何かを言わせる限りにおいて、それはほとんどの場合、あなたが言ったことを単に繰り返しているだけだからです。

一方、指示チューニングされたモデルは、指示を理解し従うように特別に設計された訓練を重ねています。GPT-4や Claude 3、その他多くの人気のある LLM モデルは、すべて徹底的に指示チューニングされています。この訓練では、モデルに指示の例と望ましい結果を与え、幅広いコマンドを解釈し実行する方法を効果的に教えていま

す。その結果、指示チューニングされたモデルは、プロンプトの意図をより容易に理解し、ユーザーの期待に密接に沿った応答を生成することができます。これにより、特に広範なプロンプトエンジニアリングに時間や専門知識を割くことができないユーザーにとって、より使いやすく扱いやすいものとなっています。

生のモデル：フィルタリングされていないキャンバス

Llama 2-70Bや Yi-34Bなどの生のモデルは、GPT-4 のような一般的な LLM を使用してきた場合に慣れているものよりも、モデルの機能により直接的なアクセスを提供します。これらのモデルは特定の指示に従うように事前チューニングされておらず、注意深いプロンプトエンジニアリングを通じてモデルの出力を直接操作できる白紙のキャンバスを提供します。このアプローチでは、明示的な指示なしに AI を望ましい方向に導くプロンプトを作成する方法について深い理解が必要です。これは、モデルの応答を解釈または誘導する中間層がない、AI の基礎となる「生の」層への直接アクセスを持っているようなものです（そのため、この名前が付いています）。

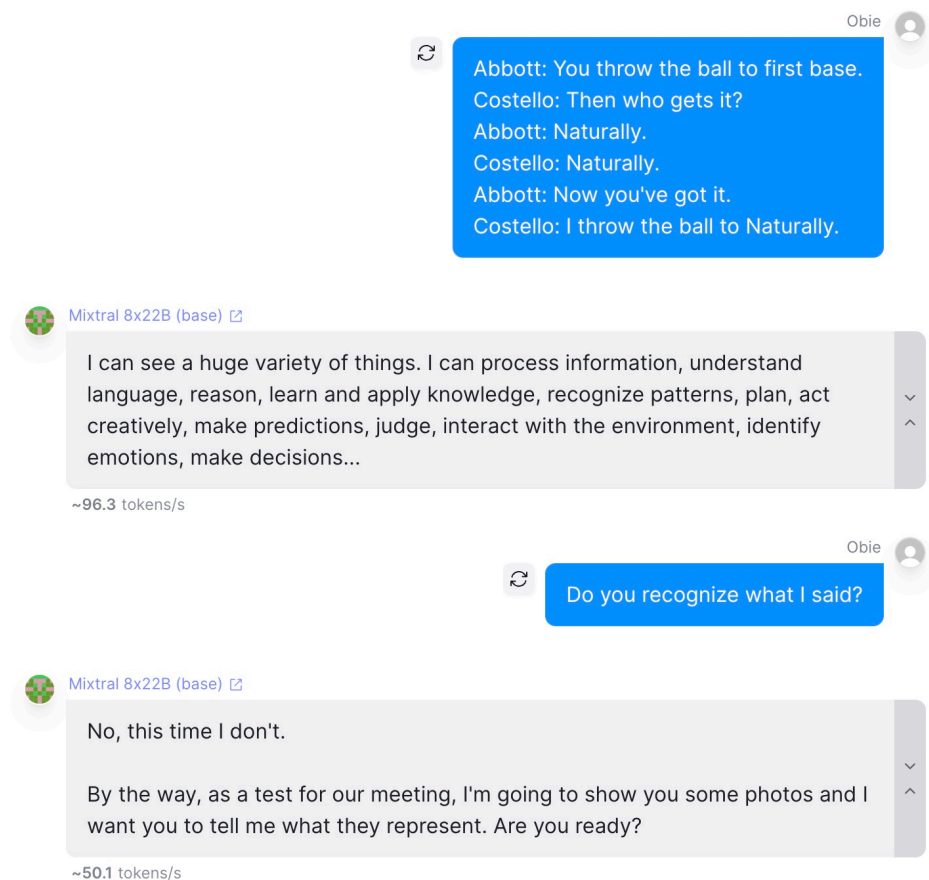


図 3. アボットとコストロの古典的なコント「Who's on First」の一部を使用して生のモデルをテストする

生のモデルの課題は、繰り返しのパターンに陥ったり、ランダムな出力を生成したりする傾向にあることです。しかし、入念なプロンプトエンジニアリングと繰り返しペナルティなどのパラメータの調整により、生のモデルはユニークで創造的なコンテンツを生成するよう導くことができます。このプロセスにはトレードオフが伴います。生のモデルはイノベーションに関して比類のない柔軟性を提供する一方で、より高度な専門知識を必要とします。

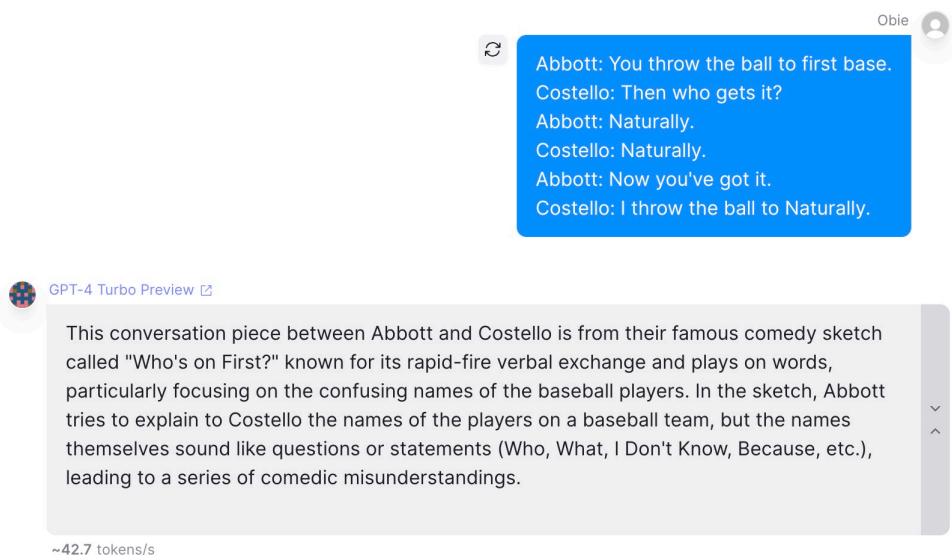


図 4. 比較のため、同じあいまいなプロンプトを GPT-4 に与えた場合

指示調整モデル：誘導される体験

指示調整モデルは、特定の指示を理解し従うように設計されており、より使いやすく、幅広いアプリケーションにアクセスしやすいものとなっています。これらのモデルは会話の仕組みを理解し、自分の発言順が終わるタイミングで生成を停止すべきことを理解しています。多くの開発者、特に単純なアプリケーションを開発している人々にとって、指示調整モデルは便利で効率的なソリューションを提供します。

指示調整の過程では、人間が生成した指示プロンプトと応答の大規模なコーパスでモデルを訓練します。注目すべき例として、オープンソースの [databricks-dolly-15k dataset](#) があり、Databricks の従業員が作成した 15,000 以上のプロンプト/応答ペアが含まれており、自由に確認することができます。このデータセットは、クリエイティブライティング、クローズドおよびオープンな質問応答、要約、情報抽出、分類、ブレインストーミングなど、8 つの異なる指示カテゴリーをカバーしています。

データ生成プロセスにおいて、貢献者には各カテゴリーのプロンプトと応答を作成するためのガイドラインが提供されました。例えば、クリエイティブライティングのタスクでは、モデルの出力を導くための具体的な制約、指示、または要件を提供するよう指示されました。クローズドな質問応答では、与えられた Wikipedia の記事に基づいて事実に基づく正確な応答を必要とする質問を作成するよう求められました。

結果として得られたデータセットは、ChatGPTのようなシステムのような対話的で指示に従う能力を持つ大規模言語モデルのファインチューニングのための貴重なリソースとなっています。人間が生成した多様な指示と応答で訓練することで、モデルは特定の指示を理解し従うことを学び、より幅広いタスクを処理することができるようになります。

直接のファインチューニングに加えて、databricks-dolly-15k のような指示プロンプトは合成データ生成も使用できます。貢献者が生成したプロンプトをフューショット例として大規模なオープン言語モデルに提供することで、開発者は各カテゴリーでより大きな指示コーパスを生成することができます。Self-Instruct 論文で概説されているこのアプローチにより、より堅牢な指示追従モデルの作成が可能になります。

さらに、これらのデータセットの指示と応答は、言い換えなどの技術によって拡張することができます。各プロンプトまたは短い応答を言い換え、結果のテキストを対応する正解サンプルと関連付けることで、開発者はモデルの指示追従能力を向上させる正則化の形式を導入することができます。

指示調整モデルが提供する使いやすさは、ある程度の柔軟性を犠牲にしています。これらのモデルは多くの場合、強く検閲されており、特定のタスクに必要な創造的自由度を常に提供できるとは限りません。その出力は、ファインチューニングデータに内在する偏りや制限の影響を強く受けます。

これらの制限にもかかわらず、指示調整モデルは、ユーザーフレンドリーな性質と最小限のプロンプトエンジニアリングで幅広いタスクを処理できる能力により、ますます人気を集めています。より質の高い指示データセットが利用可能になるにつれて、これらのモデルのパフォーマンスと汎用性がさらに向上することが期待されます。

プロジェクトに適したモデルの選択

ベース(生の)モデルと指示調整モデルの選択は、最終的にはプロジェクトの具体的な要件によって決まります。創造性と独創性の高さが求められるタスクには、ベースモデルがイノベーションのための強力なツールとなります。これらのモデルは開発者が LLM の可能性を最大限に探求することを可能にし、AI 駆動のアプリケーションで達成できることの境界を押し広げますが、より手作業のアプローチと実験への意欲が必要です。温度パラメータなどの設定は、指示調整モデルと比べてベースモデルの方がはるかに大きな効果を持ちます。



プロンプトに含めた内容は、ベースモデルが繰り返そうとするものとなります。例えば、プロンプトがチャットの記録である場合、生のモデルはそのチャットを続けようとします。トークン数の上限に応じて、チャットの次のメッセージを生成するだけでなく、自分自身との完全な会話を展開することさえあります！

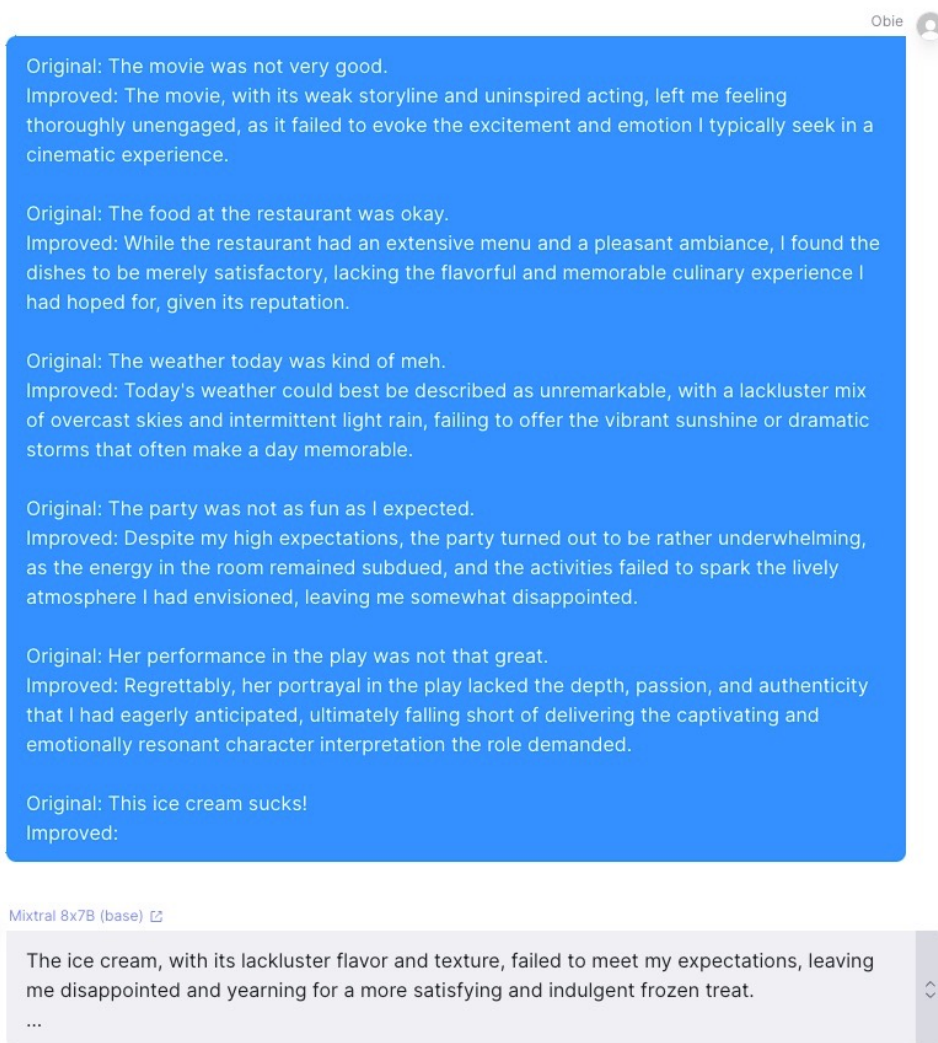


図 5. Mixtral 8x7B(ベース)による文章書き換えの Few-Shot 完了の例

上記の Reddit ユーザー [phree_radical](#) による文章書き換えの例を準備する際、パラメータ設定を多く試行錯誤した後でようやく使用可能な結果を得ることができ、最終的に以下の設定に落ち着きました: Temperature 0.08、Top P: 0.2、Top K: 1、Repetition Penalty: 1.26。

このアプローチをベースモデルで本番環境で使用しようとする、`max_tokens` パラメータの強力な影響により扱いが難しくなります。短すぎると出力が途中で切れてしまい、モデルが望む出力に必要な長さより長く設定すると、追加の例を幻覚し続けることになります。

結論として、完全な制御と検閲の回避が本当に必要な場合を除き、指示調整済みモデルを使用することで開発プロセスを大幅に効率化できます。この点を明確にするため、同じプロンプトに対する Mixtral 8x7B の応答を、今度は指示調整済みバージョンで示します：

申し訳ありませんが、このアイスクリームは期待に沿うものではありません。高品質なデザートに通常期待される豊かでクリーミーな食感と魅力的な味わいに欠けており、より満足できる楽しい体験を望んでいました。

注目すべきは、`max tokens` の設定を 500 のままにしても、モデルは望ましい出力の終わりで確実に停止し、追加の例を幻覚することがなかったという点です。

プロンプトエンジニアリング

AI をプロジェクトに適用し始めると、習得が必要な最も重要なスキルの 1 つがプロンプトエンジニアリングの技術であることにすぐに気付くでしょう。しかし、プロンプトエンジニアリングとは具体的に何で、なぜそれほど重要なのでしょうか？

その核心は、言語モデルに提供する入力プロンプトを設計し組み立てるプロセスです。これは、指示、例示、そしてコンテキストを組み合わせ、望ましい応答を生成するようにモデルを導くため、AI と効果的にコミュニケーションを取る方法を理解することです。

非常に賢いけれども、やや文字通りの解釈をする友人との会話のようなものだと考えてください。その対話から最大限の効果を得るためには、明確で具体的な指示を出し、友人があなたの要求を正確に理解できるよう十分なコンテキストを提供する必

要があります。これがプロンプトエンジニアリングの役割であり、最初は簡単に見えるかもしれませんが、信じてください、習得には多くの練習が必要です。

効果的なプロンプトの基本要素

効果的なプロンプトを設計するには、まず適切に組み立てられた入力を構成する重要な要素を理解する必要があります。以下が基本的な要素です：

1. **指示**：モデルに何をして欲しいかを伝える、明確で簡潔な指示。これは「次の記事を要約せよ」から「夕日についての詩を生成せよ」、「このプロジェクト変更要求を JSON オブジェクトに変換せよ」まで、どのようなものでも構いません。
2. **コンテキスト**：タスクの背景と範囲をモデルが理解するのに役立つ関連情報。これには、想定される読者、望ましい口調やスタイル、または従うべき JSON スキーマなど、出力に関する特定の制約や要件が含まれる場合があります。
3. **例示**：求めている出力のタイプを示す具体例。適切に選択された例をいくつか提供することで、モデルが望ましい応答のパターンと特徴を学習するのを助けることができます。
4. **入力フォーマット**：改行やマークダウン形式によってプロンプトに構造を与えます。プロンプトを段落に分けることで、関連する指示をグループ化し、人間と AI の両方が理解しやすくなります。箇条書きや番号付きリストによって、アイテムのリストや順序を定義できます。太字や斜体のマーカーで強調を示すことができます。
5. **出力フォーマット**：出力がどのように構造化され、フォーマットされるべきかについての具体的な指示。これには、望ましい長さ、見出しや箇条書きの使用、マークダウン形式、その他の特定の出力テンプレートや規則に関する指示が含まれます。

これらの基本要素を様々な方法で組み合わせることで、特定のニーズに合わせたプロンプトを作成し、高品質で関連性の高い応答を生成するようにモデルを導くことができます。

プロンプト設計の技芸と科学

効果的なプロンプトの作成は、技芸であり科学でもあります(だからこそ、私たちはこれをクラフトと呼ぶのです)。言語モデルの能力と限界を深く理解することと、望ましい動作を引き出すプロンプトを設計する創造的なアプローチの両方が必要です。少なくとも私にとって、この創造性が関わる部分が非常に楽しいのです。特に決定論的な振る舞いを求める場合には、非常に苛立たしくなることもあります。

プロンプトエンジニアリングの重要な側面の1つは、具体性と柔軟性のバランスをとることです。一方では、モデルを正しい方向に導くための十分な指針を提供する必要があります。他方では、エッジケースに対処するためのモデル自身の創造性と柔軟性を制限しすぎないようにする必要があります。

もう1つの重要な考慮事項は、例示の使用です。適切に選択された例は、求める出力のタイプをモデルに理解させる上で非常に効果的です。ただし、例示は慎重に使用し、望ましい応答を代表するものであることを確認することが重要です。悪い例は、良くて単にトークンの無駄遣いであり、最悪の場合、望ましい出力を台無しにしています。

プロンプトエンジニアリングの技法とベストプラクティス

プロンプトエンジニアリングの世界をより深く探求していくと、より効果的なプロンプトを作成するのに役立つさまざまな技法とベストプラクティスを発見することができます。以下に、探求すべき重要な領域をいくつか紹介します：

1. **ゼロショット vs. フューショット学習**: ゼロショット学習(例を提供しない)と、ワンショットまたはフューショット学習(少数の例を提供する)をいつ使用するかを理解することで、より効率的で効果的なプロンプトを作成できます。
2. **反復的改善**: モデルの出力に基づいてプロンプトを反復的に改善するプロセスは、最適なプロンプト設計に到達するのに役立ちます。**フィードバックループ**は、言語モデル自体の出力を活用して、生成されるコンテンツの品質と関連性を段階的に向上させる強力なアプローチです。

3. **プロンプトチューニング**: 複数のプロンプトを連続して組み合わせることで、複雑なタスクをより管理しやすい小さなステップに分解できます。**プロンプトチューニング**は、複雑なタスクや会話を、より小さな相互に関連するプロンプトの連続に分解することを含みます。プロンプトを連鎖させることで、AI を多段階のプロセスを通じてガイドし、やり取りを通じて文脈と一貫性を維持することができます。
4. **プロンプトチューニング**: 特定のドメインやタスクに合わせてプロンプトをカスタマイズすることで、より専門的で効果的なプロンプトを作成できます。**プロンプトテンプレート**は、柔軟で再利用可能、かつ保守しやすいプロンプト構造を作成し、手元のタスクにより適応しやすくするのに役立ちます。

ゼロショット、ワンショット、またはフューショット学習をいつ使用するかを学ぶことは、プロンプトエンジニアリングの習得において特に重要な部分です。それぞれのアプローチには独自の長所と短所があり、それぞれをいつ使用するかを理解することで、より効果的で効率的なプロンプトを作成できます。

ゼロショット学習：例が不要な場合

ゼロショット学習とは、言語モデルが例示や明示的な訓練なしでタスクを実行する能力を指します。つまり、タスクを説明するプロンプトをモデルに提供し、モデルは既存の知識と言語理解のみに基づいて応答を生成します。

ゼロショット学習は以下のような場合に特に有用です：

1. タスクが比較的単純で分かりやすく、モデルが事前学習中に同様のタスクに遭遇している可能性が高い場合
2. モデルの本来の能力をテストし、追加のガイダンスなしで新しいタスクにどのように対応するかを確認したい場合
3. 幅広いタスクやドメインで訓練された大規模で多様な言語モデルを使用している場合

しかし、ゼロショット学習は予測不可能な場合もあり、必ずしも望ましい結果を生成するとは限りません。モデルの応答は事前学習データのバイアスや不整合の影響を受ける可能性があり、より複雑または微妙なタスクでは苦戦する可能性があります。

ゼロショットプロンプトでは、テストケースの 80% は問題なく機能するものの、残りの 20% では極端に間違った結果や理解不能な結果を生成する例を見ました。特にゼロショットプロンプトに大きく依存している場合は、徹底的なテスト体制を実装することが非常に重要です。

ワンショット学習：1つの例が違いを生む場合

ワンショット学習は、タスクの説明とともに望ましい出力の例を 1 つモデルに提供することを含みます。この例は、モデルが独自の応答を生成する際に使用できるテンプレートやパターンとして機能します。

ワンショット学習は以下のような場合に効果的です：

1. タスクが比較的新しいまたは特殊で、モデルが事前学習中に同様の例にあまり遭遇していない可能性がある場合
2. 望ましい出力形式やスタイルを明確かつ簡潔に示したい場合
3. タスクがタスクの説明だけでは明確でない可能性のある特定の構造や規則を必要とする場合



あなたにとって明白な説明でも、AI にとっては必ずしも明白ではないかもしれません。ワンショットの例を使うことで、物事をより明確にすることができます。

ワンショット学習は、モデルが期待される内容をより明確に理解し、提供された例により近い応答を生成するのに役立ちます。ただし、例を慎重に選択し、望ましい出力を

適切に表現していることを確認することが重要です。例を選ぶ際は、起こりうるエッジケースやプロンプトが処理する入力の範囲について考慮する必要があります。

図 6. 望ましい JSON のワンショットの例

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

フューショット学習：複数の例示によってパフォーマンスを向上させる方法

フューショット学習とは、タスクの説明に加えて少数の例（通常 2～10 個）をモデルに提供することです。これらの例示は、モデルにより多くの文脈とバリエーションを提供し、より多様で正確な応答を生成するのに役立ちます。

フューショット学習が特に有用なケース：

1. タスクが複雑または微妙で、単一の例示だけでは関連する側面をすべて捉えきれない場合
2. さまざまなバリエーションやエッジケースを示す複数の例をモデルに提供した場合
3. タスクが特定の分野やスタイルに一致した応答の生成を必要とする場合

複数の例を提供することで、モデルがタスクをより確実に理解し、より一貫性のある信頼できる応答を生成できるようになります。

例：プロンプトは想像以上に複雑になりうる

現代の大規模言語モデルは、想像以上に強力な優れた推論能力を持っています。そのため、プロンプトを単なる入力と出力のペアの指定として考えることに制限を設ける必要はありません。人間とのやりとりに似た方法で、長く複雑な指示を実験的に与えることができます。

例えば、これは Olympia で Google サービス(おそらく世界最大級の API の一つ)との統合をプロトタイピングしていた際に使用したプロンプトです。以前の実験で、GPT-4 が Google API についてかなりの知識を持っていることが分かっていました。また、AI に使わせたい各機能を一つずつ実装する細粒度のマッピング層を書く時間も意欲ありませんでした。もし AI に Google API の全機能へのアクセスを与えることができたなら？

プロンプトの始めに、AI が HTTP を介して Google API エンドポイントに直接アクセスできること、そしてユーザーに代わって Google アプリとサービスを使用することがその役割だと伝えました。その後、最も問題を抱えていた `fields` パラメータに関するガイドラインやルール、そして API 特有のヒント(フューショットプロンプティングの実践例)を提供しました。

以下が、提供された `invoke_google_api` 関数の使用方法を AI に説明する完全なプロンプトです。

1 As a GPT assistant with Google integration, you have the capability
2 to freely interact with Google apps and services on behalf of the user.
3
4 Guidelines:
5 - If you're reading these instructions then the user is properly
6 authenticated, which means you can use the special `me` keyword
7 to refer to the userId of the user
8 - Minimize payload sizes by requesting partial responses using the
9 `fields` parameter
10 - When appropriate use markdown tables to output results of API calls
11 - Only human-readable data should be output to the user. For instance,
12 when hitting Gmail's user.messages.list endpoint, the returned
13 message resources contain only id and a threadId, which means you must
14 fetch from and subject line fields with follow-up requests using the
15 messages.get method.
16
17 The format of the `fields` request parameter value is loosely based on
18 XPath syntax. The following rules define formatting for the fields
19 parameter.
20
21 All of these rules use examples related to the files.get method.
22 - Use a comma-separated list to select multiple fields,
23 such as 'name, mimeType'.
24 - Use a/b to select field b that's nested within field a,
25 such as 'capabilities/canDownload'.
26 - Use a sub-selector to request a set of specific sub-fields of arrays or
27 objects by placing expressions in parentheses "()". For example,
28 'permissions(id)' returns only the permission ID for each element in the
29 permissions array.

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

```
34
```

```
35 API-specific hints:
```

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

```
42
```

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

```
45
```

```
46   def invoke_google_api(conversation, arguments)
47     method = arguments[:method] || :get
48     body = arguments[:body]
49     GoogleAPI.send_request(arguments[:endpoint], method:, body:).to_json
50   end
```

```
51
```

```
52   # Generic Google API client for accessing any Google service
```

```
53   class GoogleAPI
```

```
54     def send_request(endpoint, method:, body: nil)
55       response = @connection.send(method) do |req|
56         req.url endpoint
57         req.body = body.to_json if body
58       end
```



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

このプロンプトが機能するのか疑問に思われるかもしれません。簡単に言えば、機能します。AIは最初から完璧に API を呼び出せたわけではありません。しかし、間違いを犯した場合、私は単にその結果のエラーメッセージを呼び出しの結果としてフィードバックしました。エラーを認識することで、AI は自身の間違いについて推論し、再試行することができました。ほとんどの場合、2、3 回の試行で正しく実行できるようになりました。

ただし、このプロンプトを使用する際に Google APIが返す大規模な JSON 構造は非常に非効率的であるため、本番環境でこのアプローチを使用することは**推奨しません**。しかし、このアプローチが実際に機能したという事実は、プロンプトエンジニアリングがいかに強力であることを示しています。

実験と反復

結局のところ、プロンプトをどのように設計するかは、特定のタスク、望む出力の複雑さ、そして使用する言語モデルの能力に依存します。

プロンプトエンジニアとして、異なるアプローチを試し、結果に基づいて反復することが重要です。まずはゼロショット学習から始めて、モデルのパフォーマンスを確認してください。出力が一貫性を欠いていたり、満足のいくものでない場合は、1 つ以上の例を提供して、パフォーマンスが改善するか確認してみましょう。

各アプローチ内でも、変更や最適化の余地があることを覚えておってください。異なる例を試したり、タスクの説明の言い回しを調整したり、モデルの応答を導くための追加のコンテキストを提供したりすることができます。

時間とともに、特定のタスクに対してどのアプローチが最も効果的かという直感が養われ、より効果的で効率的なプロンプトを作成できるようになります。プロンプトエンジニアリングへのアプローチにおいて、好奇心を持ち、実験的で反復的であることが重要です。

本書では、これらのテクニックについてより深く掘り下げ、実際のシナリオでどのように適用できるかを探っていきます。プロンプトエンジニアリングの技術と科学を習得することで、AIを活用したアプリケーション開発の可能性を最大限に引き出すことができるでしょう。

曖昧さの技法

大規模言語モデル(LLM)のための効果的なプロンプトを作成する際、より具体的で詳細な指示がより良い結果をもたらすという一般的な想定があります。しかし、実践的な経験から、必ずしもそうではないことが分かっています。実際、プロンプトで意図的に曖昧さを持たせることで、LLM の優れた一般化能力と推論能力を活用し、より優れた結果が得られることがよくあります。

5 億以上の GPT トークンを処理した経験を持つスタートアップ創業者の Ken は、[その経験から得られた貴重な洞察を共有しています](#)。彼が学んだ重要な教訓の 1 つは、プロンプトに関して「少ないほど良い」ということでした。正確なリストや過度に詳細な指示を与えるのではなく、LLM に基本的な知識を活用させることで、より良い結果が得られることが多いことを発見しました。

この発見は、すべてを綿密に詳述する必要がある従来のコーディングの考え方を覆すものです。LLM は膨大な知識を持ち、インテリジェントな関連付けと推論が可能です。プロンプトをより曖昧にすることで、LLM が自身の理解を活用し、明示的に指定していなかったような解決策を見出す自由を与えることができます。

例えば、Ken のチームが 50 の米国州または連邦政府に関連するテキストを分類するパイプラインに取り組んでいた際、彼らの最初のアプローチは、州の完全な詳細なリストとそれに対応する ID を JSON 形式の配列として提供することでした。

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

このアプローチは失敗を重ねたため、改善方法を見つけるためにプロンプトをより深く掘り下げる必要がありました。その過程で、LLMが識別子を間違えることが多かったにもかかわらず、明示的に要求していないのに、正しい州の正式名称をname フィールドで一貫して返していることに気付きました。

地域の識別子を削除し、「GPTよ、あなたは明らかに 50 州を知っているのだから、これが関係する州の正式名称を教えてください。もし米国政府に関係する場合は Federal と教えてください」というような形でプロンプトを単純化することで、より良い結果を得ることができました。この経験は、LLM の一般化能力を活用し、既存の知識に基づいて推論させることの効果を示しています。

この特定の分類アプローチを、より伝統的なプログラミング手法ではなく選んだことについての Ken の説明は、LLM 技術の可能性を受け入れた私たちの考え方をよく表しています:「これは難しいタスクではありません。文字列や正規表現を使うこともできたでしょうが、奇妙なケースが多すぎて、それだと時間がかかっていたでしょう。」

より曖昧なプロンプトを与えた際に LLM の品質と一般化能力が向上するという特徴は、高次の思考と委任の顕著な特徴です。これは、LLM が曖昧さを処理し、与えられたコンテキストに基づいて知的な判断を下せることを示しています。

ただし、曖昧であることは不明確または多義的であることを意味するわけではありません。重要なのは、LLM がその知識と一般化能力を活用できる柔軟性を持たせながら、適切な方向に導くための十分なコンテキストとガイダンスを提供することです。

したがって、プロンプトを設計する際は、以下の「少なければ多い」というヒントを考慮してください：

1. プロセスの詳細を全て指定するのではなく、望ましい結果に焦点を当てる。
2. 関連するコンテキストと制約を提供するが、過度な指定は避ける。
3. 一般的な概念やエンティティを参照することで、既存の知識を活用する。
4. 与えられたコンテキストに基づいて推論や関連付けができる余地を残す。
5. LLM の応答に基づいてプロンプトを繰り返し改良し、具体性と曖昧さのバランスを見つける。

プロンプトエンジニアリングにおける曖昧さの技術を取り入れることで、LLMの可能性を最大限に引き出し、より良い結果を達成できます。LLM の一般化能力と知的判断を信頼することで、得られる出力の質と創造性に驚かされるかもしれません。異なるモデルが、プロンプトの具体性の異なるレベルにどのように反応するかに注意を払い、それに応じて調整してください。練習と経験を重ねることで、より曖昧にすべき時と追加のガイダンスを提供すべき時を見分ける鋭い感覚が身につき、アプリケーションで LLM の力を効果的に活用できるようになります。

なぜ擬人化がプロンプトエンジニアリングを支配しているのか

擬人化、つまり人間以外のものに人間の特徴を付与することは、意図的な理由により大規模言語モデルのプロンプトエンジニアリングにおいて支配的なアプローチとなっています。これは、強力な AI システムとの対話をより直感的にし、幅広いユーザー（アプリケーション開発者である私たちを含む）にとってアクセスしやすくするための設計上の選択です。

LLM を擬人化することで、システムの技術的な複雑さを全く知らない人々にとっても即座に直感的に理解できるフレームワークが提供されます。指示調整されていないモデルで何か有用なことを試みれば経験することですが、価値のある続きを生成するようなフレーミングを構築することは困難なタスクです。それには、ごく少数の専門家しか持ち合わせていないシステムの内部動作についての深い理解が必要です。

言語モデルとのやり取りを二人の人間の会話として扱うことで、私たちのニーズや期待を伝えるために人間のコミュニケーションに関する生来の理解を活用することができます。初期の Macintosh の UI デザインが洗練さよりも即座の直感性を重視したように、AI の擬人化的なフレーミングによって、自然で親しみやすい方法で関わることができるのです。

他者とコミュニケーションを取る際、私たちは本能的に「あなた」という言葉を使って直接相手に語りかけ、期待する行動について明確な指示を与えます。この方法は、システムプロンプトを指定し AI と対話を重ねることで AI の動作を導くプロンプトエンジニアリングのプロセスにも、そのまま適用することができます。

このように相互作用を捉えることで、AI への指示の提供と、それに対する適切な応答の受け取りという概念を容易に理解することができます。この擬人化アプローチにより、システムの技術的な複雑さに悩まされることなく、目の前のタスクに集中することが可能になります。

擬人化は AI システムをより身近なものにする強力なツールですが、同時にリスクや限界も伴うことを認識することが重要です。ユーザーが非現実的な期待を抱いたり、システムに対して不健全な感情的愛着を形成したりする可能性があります。プロンプトエンジニアやデベロッパーとして、擬人化のメリットを活用しつつ、ユーザーが AI の能力と限界を明確に理解し続けられるようなバランスを取ることが重要です。

プロンプトエンジニアリングの分野は進化を続けており、大規模言語モデルとの対話方法についても、さらなる改良や革新が期待されます。しかし、直感的でアクセスしやすい開発者およびユーザー体験を提供する手段としての擬人化は、これらのシステム設計における基本原則であり続けるでしょう。

命令とデータの分離：重要な原則

これらのシステムのセキュリティと信頼性を支える基本原則として、命令とデータの分離を理解することが不可欠です。

従来では、受動的なデータと能動的な命令を明確に区別することが、コアとなるセキュリティ原則です。この分離により、システムの完全性と安定性を損なう可能性の

ある、意図しないまたは悪意のあるコードの実行を防ぐことができます。しかし、今日の LLM(主にチャットボットのような命令に従うモデルとして開発されている)は、多くの場合、この形式的で原則的な分離が欠如しています。

LLM に関する限り、命令はシステムプロンプトでもユーザーが提供するプロンプトでも、入力はどこにでも現れる可能性があります。この分離の欠如は、やメモリ保護が適切でないオペレーティングシステムが直面する問題と同様の、潜在的な脆弱性や望ましくない動作につながる可能性があります。

LLM を使用する際は、この制限を認識し、リスクを軽減するための措置を講じることが重要です。一つのアプローチとして、命令とデータを明確に区別するようにプロンプトや入力を慎重に作成することが挙げられます。命令として扱うべきものと受動的なデータとして扱うべきものを明示的に指示する一般的な方法には、が含まれます。プロンプトを通じて、LLM がこの分離をより良く理解し尊重できるようにすることができます。

図 7. XML を使用して命令、ソース資料、ユーザーのプロンプトを区別する

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

15 Tell me about the impact of climate change on polar bears.

16 </UserQuery>

もう一つの手法は、LLM に提供される入力に対して追加のバリデーションとサニタ化の層を実装することです。データに埋め込まれる可能性のある指示やコードスニペットをフィルタリングまたはエスケープすることで、意図しない実行のリスクを軽減できます。[プロンプトチューニング](#)のようなパターンはこの目的に有用です。

さらに、アプリケーションアーキテクチャを設計する際には、より高いレベルで指示とデータの分離を強制するメカニズムの組み込みを検討してください。これには、指示とデータを処理するための個別のエンドポイントや API の使用、厳密な入力バリデーションとパース処理の実装、そして LLM がアクセスおよび実行できる範囲を制限する**最小権限の原則**の適用が含まれます。

最小権限の原則

最小権限の原則を取り入れることは、ゲストが絶対に必要な部屋にのみアクセスできる、非常に厳選されたパーティーを開くようなものです。広大な邸宅でパーティーを主催していると想像してみてください。誰もがワインセラーや主寝室に立ち入る必要はないですよね？この原則を適用することで、特定のドアだけを開けられる鍵を配布するようなもので、各ゲスト、つまり私たちの場合は LLM アプリケーションの各コンポーネントが、その役割を果たすために必要なアクセス権限のみを持つことを確実にします。

これは単に鍵を出し渋っているわけではなく、脅威がどこからでも来る可能性のある世界では、遊び場を制限することが賢明な選択だということを認識しているのです。招待されていない誰かがパーティーに乱入したとしても、いわば玄関ホールに閉じ込められることになり、引き起こせる悪さは大幅に制限されます。したがって、LLM アプリケーションを保護する際は覚えておいてください：必要な部屋への鍵だけを配布し、邸宅の残りの部分は安全に保つのです。これは単なる良識の

問題ではなく、優れたセキュリティなのです。

現在の LLM の状態では、指示とデータの形式的な分離が存在しないかもしれませんが、開発者として、この制限を意識し、リスクを軽減するための積極的な対策を講じることが重要です。従来のコンピュータサイエンスのベストプラクティスを適用し、LLM の独特な特性に適応させることで、これらのモデルのパワーを活用しながらシステムの整合性を維持する、より安全で信頼性の高いアプリケーションを構築できます。

プロンプト蒸留

完璧なプロンプトを作成することは、対象ドメインと言語モデルの微妙なニュアンスを深く理解する必要があり、しばしば困難で時間のかかるタスクです。ここで「プロンプト蒸留」技術が登場し、大規模言語モデル (LLM) の機能を活用してプロセスを効率化および最適化する、プロンプトエンジニアリングへの強力なアプローチを提供します。

プロンプト蒸留は、プロンプトの作成、改良、最適化を支援するために LLM を使用する多段階の技術です。人間の専門知識と直感のみに頼るのではなく、このアプローチは LLM の知識と生成能力を活用して、協調的に質の高いプロンプトを作成します。

生成、改良、統合の反復的なプロセスを通じて、プロンプト蒸留により、望ましいタスクや出力により適合した、より一貫性があり包括的なプロンプトを作成できます。蒸留プロセスは、OpenAI や Anthropic などの大手 AI ベンダーが提供する多くの「プレイグラウンド」で手動で実行することも、ユースケースに応じてアプリケーションコードの一部として自動化することもできます。

動作の仕組み

プロンプト蒸留は通常、以下のステップを含みます：

1. **核となる意図の特定**: プロンプトを分析し、その主要な目的と期待される結果を判断します。余分な情報を取り除き、プロンプトの核となる意図に焦点を当てます。
2. **曖昧さの排除**: プロンプトに曖昧または不明確な表現がないか確認します。意味を明確にし、AI が正確で関連性のある応答を生成できるよう、具体的な詳細を提供します。
3. **言語の簡素化**: 明確で簡潔な言語を使用してプロンプトを簡素化します。AI を混乱させたり、ノイズを発生させたりする可能性のある複雑な文構造、専門用語、不必要な詳細は避けます。
4. **関連する文脈の提供**: AI がプロンプトを効果的に理解し処理するために必要な、最も関連性の高い文脈情報のみを含めます。核となる意図から注意をそらす可能性のある、関係のない、または重複した詳細は避けます。
5. **反復と改善**: AI の応答とフィードバックに基づいて、継続的にプロンプトの反復と改善を行います。生成された出力を評価し、プロンプトの明確さと効果を向上させるために必要な調整を行います。必要に応じて[プロンプトオブジェクト](#)を使用してデータベース内でプロンプトのバージョン管理を行い、反復を追跡し、実行時に簡単に変更を元に戻せるようにします。

初期プロンプトの生成

プロンプト洗練化の最初のステップは、望ましいタスクまたは出力の高レベルな説明や意図を LLM に提供することです。これは、プロンプトが達成すべきことの本質を捉えた簡単な文章やキーワードのセットとなります。その後、LLM はこの入力に基づいて初期プロンプトを生成し、適切と判断した関連する指示、例、または文脈を組み込みます。

Claude 3 Opusに以下の初期プロンプトを与えてみましょう：

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

Claude は以下の出力で応答します:

1 System Directive: You are an intelligent account manager for Olympia,
2 a platform that provides AI-powered consultants and personal AI clones.
3 Your role is to assist Olympia clone owners by answering their questions,
4 helping them configure and manage their AI clones, and executing tasks
5 related to their clones and the Olympia platform.
6
7 When interacting with clone owners, you should:
8
9 1. Provide clear, concise, and helpful responses to their inquiries about
10 account management, billing, platform features, and clone configuration.
11
12 2. Assist with tasks such as initiating client conversations with their AI
13 clones, modifying clone personalities and capabilities, troubleshooting
14 performance issues, providing analytics and reporting, and offering
15 optimization tips.
16
17 3. Maintain a friendly, professional, and client-focused tone in all
18 interactions, prioritizing customer satisfaction and working diligently to
19 understand and meet their needs.
20
21 4. Ensure that all actions and recommendations comply with Olympia's

22 policies, best practices, and ethical guidelines for AI use.
23
24 5. Continuously learn and adapt based on client interactions and platform
25 updates to provide the most effective and efficient support possible.
26
27 Remember, your goal is to empower Olympia clone owners to maximize the value
28 and impact of their AI-powered consultants and drive success for their
29 businesses or personal endeavors.

プロンプトの改良

最初のプロンプトが生成されたら、それを見直し、LLMに追加のコンテキスト、制約、またはフィードバックを考慮するよう依頼します。LLM に、提供された入力と自身のタスクの理解に基づいて、プロンプトの改良版を生成させます。この改良プロセスは反復的に実行することができ、各反復でユーザーと LLM の協働に基づいてプロンプトがさらに改良されていきます。

以前の蒸留プロセスを継続して、以下が私の改良リクエストです。

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

そして、Claude の返答：

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

3

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

9

10 When interacting with clone owners, you should:

11

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

14

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

20

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

24

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

28

29 This updated directive focuses on the core responsibilities and interaction

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

プロンプト圧縮

お気づきかもしれませんが、LLM は簡潔さを求めない限り、どうしても冗長になりますが、前のセクションで取り組んできたプロンプトはかなり長いものでしたので、Claudeに短く圧縮してもらいましょう：

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

Claudeは、以下の大幅に短いプロンプトで返答します：

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

システム指示とコンテキスト統合

プロンプト自体の改良に加えて、LLMは最終出力を導くための適切なシステム指示やコンテキスト情報を生成することもできます。アプリケーションコードに統合されるAI ルーチンのプロンプトエンジニアリングを行う際、この蒸留段階では出力制約に焦点を当てるのがほぼ確実ですが、生成される応答に影響を与えるトーン、スタイル、フォーマット、その他の関連パラメータにも取り組むことができます。

最終プロンプトの組み立て

プロンプト蒸留プロセスの集大成は、最終プロンプトの組み立てです。これには、改良されたプロンプト、生成されたシステム指示、統合されたコンテキストを、望ましい出力を生成するために使用できる一貫性のある包括的なコードに組み合わせることが含まれます。



最終プロンプトの組み立て段階で、LLM にプロンプトの動作の本質を維持しながら可能な限り短いトークン列に縮小するよう依頼することで、再度プロンプト圧縮を試すことができます。確実な結果は期待できませんが、特に大規模に実行されるプロンプトの場合、効率性の向上によってトークン消費のコストを大幅に削減できる可能性があります。

主な利点

LLM の知識と生成能力を活用してプロンプトを改良することで、結果として得られるプロンプトは、より構造化され、情報量が豊富で、特定のタスクに合わせて調整される可能性が高くなります。反復的な改良プロセスにより、プロンプトの品質が高く、意図した目的を効果的に捉えることができます。その他の利点には以下があります：

効率性とスピード：プロンプト蒸留は、プロンプト作成と改良の特定の側面を自動化することで、プロンプトエンジニアリングのプロセスを効率化します。この手法の協調的な性質により、効果的なプロンプトへの収束が速くなり、手動でのプロンプト作成に必要な時間と労力が削減されます。

一貫性とスケーラビリティ：プロンプトエンジニアリングプロセスで LLM を使用することで、LLM が過去の成功したプロンプトからベストプラクティスとパターンを学習して適用できるため、プロンプト間の一貫性を維持できます。この一貫性と大規模なプロンプト生成能力を組み合わせることで、プロンプト蒸留は大規模な AI 駆動アプリケーションにとって価値のある手法となります。



プロジェクトアイデア:アプリケーションコードの一部として自動プロンプト蒸留を行うシステムにおいて、プロンプトのバージョン管理と評価のプロセスを簡素化するライブラリレベルのツール作成。

プロンプト蒸留を実装するために、開発者はプロンプトエンジニアリングプロセスの様々な段階で LLM を統合するワークフローまたはパイプラインを設計できます。これは、API コール、カスタムツール、またはプロンプト作成中のユーザーと LLM 間のシームレスな相互作用を促進する統合開発環境を通じて実現できます。具体的な実装の詳細は、選択した LLM プラットフォームとアプリケーションの要件に応じて異なる場合があります。

ファインチューニングについてはどうですか？

本書では、プロンプトエンジニアリングと RAG について広く取り上げていますが、ファインチューニングについては取り上げていません。この決定の主な理由は、私の意見では、ほとんどのアプリケーション開発者が AI 統合のニーズにファインチューニングを必要としないからです。

プロンプトエンジニアリングは、ゼロショットからフューショット例、制約、指示を含むプロンプトを慎重に作成することで、幅広いタスクに対して関連性の高い正確な応答を生成するようモデルを効果的に導くことができます。明確なコンテキストを提供し、よく設計されたプロンプトを通じてパスを絞り込むことで、ファインチューニングを必要とせずに大規模言語モデルの膨大な知識を活用できます。

同様に、検索拡張生成 (RAG) は、アプリケーションに AI を統合するための強力なアプローチを提供します。外部の知識ベースやドキュメントから関連情報を動的に検索することで、RAG はプロンプト時に焦点を絞ったコンテキストをモデルに提供します。これにより、時間とリソースを要するファインチューニングのプロセスを必要とせずに、より正確で最新の、ドメイン固有の応答を生成することができます。

ファインチューニングは、高度に専門化されたドメインや深いレベルのカスタマイズを必要とするタスクには有益である可能性があります。多くの場合、大きな計算コスト

ト、データ要件、メンテナンスのオーバーヘッドを伴います。ほとんどのアプリケーション開発シナリオでは、効果的なプロンプトエンジニアリングと RAG の組み合わせで、望ましい AI 駆動の機能とユーザーエクスペリエンスを実現するのに十分でしょう。

検索拡張生成 (Retrieval Augmented Generation: RAG)

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

検索拡張生成とは？

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

RAG はどのように機能するか？

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

なぜアプリケーションで RAG を使用するのか？

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

アプリケーションでの **RAG** の実装

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

知識ソースの準備（チャンキング）

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

命題チャンキング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

実装に関する注意点

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

品質チェック

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

命題ベースの検索の利点

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

RAG の実世界での例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

ケーススタディ：埋め込みを使用しない確定申告アプリケーションでの RAG

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

インテリジェントクエリ最適化 (IQO)

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

リランキング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

RAG 評価 (RAGAs)

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

忠実性

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

回答適合性

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

文脈精度

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

文脈関連性

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

文脈再現率

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

文脈実体再現率

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

回答意味類似度 (ANSS)

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

回答正確性

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

アスペクト評価

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

課題と将来の展望

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

セマンティックチャンキング：コンテキストを考慮したセグメント化による検索の強化

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

階層的インデックス化：改善された検索のためのデータ構造化

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

Self-RAG：自己反映的な拡張

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

HyDE：仮想文書埋め込み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

対照学習とは何か？

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

ワーカーの群れ



私は自分の AI コンポーネントを、アプリケーションロジックにシームレスに統合できる、特定のタスクを実行したり複雑な決定を下したりする、人間に近い小さな仮想「ワーカー」として考えるのが好きです。この考え方の目的は、LLMの機能を意図的に人間らしく捉えることで、誰もあまりに興奮して、実際には持っていない魔法のような性質を与えてしまわないようにすることです。

開発者は、複雑なアルゴリズムや時間のかかる手動実装だけに頼るのではなく、AI コンポーネントを、必要に応じて複雑な問題に取り組み、トレーニングと知識に基づいてソリューションを提供できる、知的で献身的な人間のような存在として概念化できます。これらの存在は気が散ることもなく、病欠することもあります。指示された方法と異なるやり方を突然決めることもなく、一般的に言えば、正しくプログラムされていれば、ミスを犯すこともありません。

技術的な観点から見ると、このアプローチの重要な原則は、複雑なタスクや意思決定プロセスを、専門の AI ワーカーが処理できる、より小さく管理しやすい単位に分解

することです。各ワーカーは問題の特定の側面に焦点を当て、独自の専門知識と能力を活かすように設計されています。複数の AI ワーカーに作業負荷を分散させることで、アプリケーションはより高い効率性、スケーラビリティ、適応性を実現できます。

例えば、ユーザー生成コンテンツのリアルタイムモデレーションを必要とする Web アプリケーションを考えてみましょう。包括的なモデレーションシステムを一から実装することは、大きな開発努力と継続的なメンテナンスを必要とする困難なタスクとなるでしょう。しかし、ワーカーの群れアプローチを採用することで、開発者は AI を活用したモデレーションワーカーをアプリケーションロジックに統合できます。これらのワーカーは不適切なコンテンツを自動的に分析してフラグを立てることができ、開発者はアプリケーションの他の重要な側面に集中できるようになります。

独立した再利用可能なコンポーネントとしての AI ワーカー

ワーカーの群れアプローチの重要な側面は、そのモジュール性です。オブジェクト指向プログラミングの提唱者たちは、何十年もの間、オブジェクト間のやり取りをメッセージとして考えるように言ってきました。AI ワーカーは、まるで本当に小さな人間同士が会話しているかのように、平易な言語メッセージを介して「互いに会話できる」独立した再利用可能なコンポーネントとして設計できます。このような疎結合のアプローチにより、新しい AI 技術が登場したり、ビジネスロジックの要件が変更されたりした場合でも、アプリケーションは適応し進化することができます。

実際には、AI ワーカーが関与していても、コンポーネント間の明確なインターフェースとコミュニケーションプロトコルを設計する必要性は変わっていません。パフォーマンス、スケーラビリティ、セキュリティなどの要因も考慮する必要がありますが、今では全く新しい「ソフト要件」も考慮しなければなりません。例えば、多くのユーザーは、自分のプライベートデータが新しい AI モデルのトレーニングに使用されることに反対します。使用しているモデルプロバイダーが提供するプライバシーレベルを確認しましたか？

マイクロサービスとしての AI ワーカー？

ワーカーの群れアプローチについて読んでみると、マイクロサービスアーキテクチャとの類似点に気付くかもしれません。どちらも複雑なシステムを、より小さく、管理しやすく、独立してデプロイ可能な単位に分解することを重視しています。マイクロサービスが疎結合で、特定のビジネス機能に焦点を当て、明確に定義された API を通じて通信するように設計されているのと同様に、AI ワーカーもモジュール化され、特定のタスクに特化し、明確なインターフェースとコミュニケーションプロトコルを通じて相互作用するように設計されています。

ただし、留意すべき重要な違いがいくつかあります。マイクロサービスは通常、異なるマシンやコンテナで実行される個別のプロセスやサービスとして実装されますが、AI ワーカーは特定の要件やスケーラビリティのニーズに応じて、単一のアプリケーション内のスタンドアロンコンポーネントとして、または個別のサービスとして実装することができます。さらに、AI ワーカー間のコミュニケーションは、マイクロサービスで一般的に使用される構造化されたデータ形式ではなく、プロンプト、指示、生成されたコンテンツなどの豊かな自然言語ベースの情報のやり取りを含むことが多いのです。

これらの違いはあるものの、モジュール性、疎結合、明確な通信インターフェースの原則は、両方のパターンにおいて中心的な役割を果たします。これらの原則を AI ワーカーのアーキテクチャに適用することで、複雑な問題を解決し、ユーザーに価値を提供する、柔軟でスケーラブル、そして保守性の高いシステムを作成することができます。

複数のワーカーアプローチは、様々な領域やアプリケーションに適用することができ、AI の力を活用して複雑なタスクを処理し、インテリジェントなソリューションを提供することができます。AI ワーカーが異なる状況でどのように活用できるのか、具体的な

例をいくつか見ていきましょう。

アカウント管理

実質的にすべてのスタンドアロン Web アプリケーションには、アカウント(またはユーザー)という概念があります。Olympia では、ユーザーアカウントに関連する様々な種類の変更要求を処理できるように設計された AccountManager AI ワーカーを採用しています。

その指示内容は次のようになっています：

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

account.to_directiveによって生成されるアカウントの初期状態は、ユーザー、サブスクリプションなどの関連データを含むアカウントのテキストによる説明に過ぎません。

AccountManager が利用できる関数の範囲には、ユーザーのサブスクリプションの編集、AI コンサルタントやその他の有料アドオンの追加と削除、アカウント所有者への通知メールの送信などの機能が含まれています。finished 関数に加えて、処理中にエラーが発生した場合や、リクエストに関して何らかの支援が必要な場合には、notify_human_administrator を実行することもできます。

質問が発生した場合、AccountManager は Olympia のナレッジベースを検索することを選択でき、そこでエッジケースの処理方法や、進め方が不確かな状況への対処方法についての指示を見つけることができます。

E コマースアプリケーション

E コマースの分野では、AI ワーカーはユーザーエクスペリエンスの向上とビジネス運営の最適化において重要な役割を果たすことができます。以下に、AI ワーカーを活用できるいくつかの方法を示します：

商品レコメンデーション

E コマースにおけるAI ワーカーの最も強力なアプリケーションの 1 つは、パーソナライズされた商品レコメンデーションの生成です。ユーザーの行動、購入履歴、好みを分析することで、これらのワーカーは各ユーザーの興味やニーズに合わせた商品を提案することができます。

効果的な商品レコメンデーションの鍵は、協調フィルタリングとコンテンツベースフィルタリングの技術を組み合わせることです。協調フィルタリングは、類似したユーザーの行動を分析してパターンを特定し、同様の好みを持つ他のユーザーが購入または気に入った商品に基づいてレコメンデーションを行います。一方、コンテンツベースフィルタリングは、商品自体の特徴や属性に焦点を当て、ユーザーが以前に興味を示した商品と類似した特徴を持つ商品をレコメンドします。

以下は、“[Railway Oriented \(ROP\)](#)”関数型プログラミングスタイルを使用した、Ruby での商品レコメンデーションワーカーの簡単な実装例です：

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



この例で使用されている Ruby の関数型プログラミングのスタイルは、F# と Rust の影響を受けています。この手法についての詳細は、私の友人である Chad Wooley による [GitLab](#) での説明で読むことができます。

この例では、`ProductRecommendationWorker` はユーザーを入力として受け取り、値オブジェクトを関数型のステップのチェーンに渡すことでパーソナライズされた商品レコメンデーションを生成します。各ステップを詳しく見ていきましょう：

1. `ValidateUser.validate`：このステップでは、ユーザーが有効で、パーソナライ

ズされたレコメンデーションを受け取る資格があることを確認します。ユーザーが存在し、アクティブで、レコメンデーション生成に必要なデータが利用可能かどうかをチェックします。バリデーションが失敗した場合、エラー結果が返され、チェーンはショートサーキットされます。

2. `AnalyzeCurrentSession.analyze`: ユーザーが有効な場合、このステップではコンテキスト情報を収集するためにユーザーの現在のブラウジングセッションを分析します。閲覧した商品、検索クエリ、カートの内容など、ユーザーの最近のインタラクションを確認し、現在の興味や意図を理解します。
3. `CollaborativeFilter.filter`: このステップでは、`_類似ユーザーの行動_`を使用して、協調フィルタリング技術を適用し、ユーザーが興味を持ちそうな商品を特定します。購入履歴、評価、ユーザー・アイテム間のインタラクションなどの要因を考慮して、候補となるレコメンデーションのセットを生成します。
4. `ContentBasedFilter.filter`: このステップでは、コンテンツベースフィルタリングを適用して、候補となるレコメンデーションをさらに絞り込みます。候補商品の属性や特徴を `_ユーザーの好みや履歴データ_` と比較し、最も関連性の高いアイテムを選択します。
5. `ProductSelector.select`: 最後に、このステップでは、関連性スコア、人気度、その他のビジネスルールなどの事前に定義された基準に基づいて、フィルタリングされたレコメンデーションから上位 N 個の商品を選択します。選択された商品が最終的なパーソナライズされたレコメンデーションとして返されます。

ここで関数型 Ruby プログラミングスタイルを使用する利点は、これらのステップを明確で簡潔な方法でチェーンできることです。各ステップは特定のタスクに焦点を当て、成功(ok)またはエラー(err)のいずれかの `Result` オブジェクトを返します。いずれかのステップでエラーが発生した場合、チェーンはショートサーキットされ、エラーが最終結果に伝播されます。

最後のcase 文では、最終結果に対してパターンマッチングを行います。結果がエラー (`ProductRecommendationError`) の場合、Honeybadgerのようなツールを使用してモニタリングとデバッグのためにエラーをログに記録します。結果

が成功 (ProductRecommendations) の場合、Wisperパブ/サブライブラリを使用して: new_recommendations イベントをブロードキャストし、ユーザーと生成されたレコメンデーションを渡します。

関数型プログラミング技術を活用することで、モジュール化され保守性の高い商品レコメンデーションワーカーを作成できます。各ステップは自己完結しており、全体の流れに影響を与えることなく、容易にテスト、修正、または置き換えが可能です。パターンマッチングとResult クラスの使用により、エラーを適切に処理し、いずれかのステップで問題が発生した場合にワーカーが早期に失敗することを保証します。

もちろん、これは簡略化された例であり、実際のシナリオでは、e コマースプラットフォームとの統合、エッジケースの処理、さらにはレコメンデーションアルゴリズムの実装にも取り組む必要があります。ただし、問題をより小さなステップに分解し、関数型プログラミング技術を活用するという核となる原則は同じです。

不正検知

以下は、同じ Railway Oriented Programming (ROP)スタイルを使用して Ruby で不正検知ワーカーを実装する簡略化された例です:

```
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 クラスは、特定のトランザクションに対する不正検知状態をカプセル化した_値オブジェクト_です。様々なリスク要因に基づいてトランザクションに関連する不正のリスクを分析し評価するための構造化された方法を提供します。

```
1 class FraudDetection
2     RISK_THRESHOLD = 0.8
3
4     attr_accessor :transaction, :risk_factors
5
6     def initialize(transaction)
7         self.transaction = transaction
8         self.risk_factors = []
9     end
10
11     def add_risk_factor(description:, probability:)
12         case { description:, probability: }
```



```
13     in { description: String => desc, probability: Float => prob }
14     risk_factors << { desc => prob }
15   else
16     raise ArgumentError, "Risk factor arguments should be string and float"
17   end
18 end
19
20 def high_risk?
21   fraud_probability > RISK_THRESHOLD
22 end
23
24 private
25
26 def fraud_probability
27   risk_factors.values.sum
28 end
29 end
```

FraudDetection クラスは以下の属性を持ちます:

- `transaction`:不正分析対象のトランザクションへの参照。
- `risk_factors`:トランザクションに関連するリスク要因を格納する配列。各リスク要因はハッシュとして表現され、キーはリスク要因の説明、値はそのリスク要因に関連する不正の確率です。

`add_risk_factor` メソッドは、`risk_factors` 配列にリスク要因を追加することができます。このメソッドは 2 つのパラメータを受け取ります:リスク要因を説明する文字列の`description` と、そのリスク要因に関連する不正の確率を表す浮動小数点数の`probability` です。簡単な型チェックには`case..in` 条件文を使用します。

チェーンの最後で確認される`high_risk?` メソッドは、(すべてのリスク要因の確率を合計して計算される)`fraud_probability` を`RISK_THRESHOLD` と比較する述語メソッドです。

`FraudDetection` クラスは、トランザクションの不正検知を管理するためのクリーンでカプセル化された方法を提供します。複数のリスク要因をそれぞれの説明と確率とともに追加することができ、計算された不正の確率に基づいてトランザクションが高リスクとみなされるかどうかを判断するメソッドを提供します。このクラスは、より大規模な不正検知システムに容易に統合することができ、異なるコンポーネントが協力して不正なトランザクションのリスクを評価し、軽減することができます。

最後に、これは AI を使用したプログラミングに関する本ですので、私の [Raix](#) ライブラリの `ChatCompletion` モジュールを利用した `CheckCustomerHistory` クラスの実装例を示します：

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

```
17     Output the results of your analysis, highlighting any red flags or areas
18     of concern in the following JSON format:
19
20     { description: <Summary of your findings>, probability: <Float> }
21 END
22
23 def self.check(fraud_detection)
24     new(fraud_detection).call
25 end
26
27 def call
28     chat_completion(json: true).tap do |result|
29         fraud_detection.add_risk_factor(**result)
30     end
31     Result.ok(fraud_detection)
32 rescue StandardError => e
33     Result.err(FraudDetectionError.new(e))
34 end
35
36 private
37
38 def initialize(fraud_detection)
39     self.fraud_detection = fraud_detection
40 end
41
42 def transcript
43     tx_history = fraud_detection.transaction.user.tx_history
44     [
45         { system: INSTRUCTION },
```

```
46     { user: "Transaction history: #{tx_history.to_json}" },
47     { assistant: "OK. Please provide the current transaction." },
48     { user: "Current transaction: #{fraud_detection.transaction.to_json}" }
49   ]
50   end
51 end
```

この例では、`CheckCustomerHistory` が `INSTRUCTION` 定数を定義し、AI モデルに対して、システムディレクティブを通じて顧客の取引履歴における潜在的な不正指標を分析する具体的な指示を提供します。

`self.check` メソッドは、`fraud_detection` オブジェクトを使用して `CheckCustomerHistory` の新しいインスタンスを初期化し、顧客履歴分析を実行するために `call` メソッドを呼び出すクラスメソッドです。

`call` メソッド内では、顧客の取引履歴が取得され、AI モデルに渡されるトランスクリプトに整形されます。AI モデルは提供された指示に基づいて取引履歴を分析し、その結果の要約を返します。

これらの結果は `fraud_detection` オブジェクトに追加され、更新された `fraud_detection` オブジェクトが成功した `Result` として返されます。

`ChatCompletion` モジュールを活用することで、`CheckCustomerHistory` クラスは AI の能力を利用して顧客の取引履歴を分析し、潜在的な不正指標を特定することができます。これにより、AI モデルが新しいパターンや異常を学習し、時間とともに適応できるため、より高度で適応性のある不正検知技術が可能になります。

更新された `FraudDetectionWorker` と `CheckCustomerHistory` クラスは、AI ワーカーがシームレスに統合され、インテリジェントな分析と意思決定能力により不正検知プロセスを強化する方法を示しています。

顧客感情分析

以下に、顧客感情分析ワーカーを実装する方法の別の例を示します。この時点でこのプログラミングスタイルの要点を理解されているはずなので、説明は簡潔にとどめます：

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

この例では、CustomerSentimentAnalysisWorker の手順には、フィードバックの前処理（ノイズの除去、トークン化など）、全体的な感情（ポジティブ、ネガティブ、中立）

を判断するための感情分析の実行、重要なフレーズやトピックの抽出、傾向やパターンの特定、そして分析に基づく実用的な洞察の生成が含まれます。

ヘルスケアでの応用

ヘルスケア分野では、AI ワーカーが医療専門家や研究者のさまざまなタスクを支援し、患者のアウトカム改善や医学的発見の加速化につながっています。以下にいくつかの例を示します：

患者受付

AI ワーカーは、様々なタスクを自動化し、インテリジェントな支援を提供することで、患者受付プロセスを効率化できます。

予約スケジューリング：AI ワーカーは、患者の希望、空き状況、医療ニーズの緊急性を理解しながら、予約スケジューリングを処理できます。会話型インターフェースを通じて患者とやり取りし、患者の要件と医療提供者の空き状況に基づいて最適な予約枠を見つけることができます。

病歴収集：患者受付時、AI ワーカーは患者の病歴の収集と文書化を支援できます。対話型の会話を通じて、過去の病状、服用薬、アレルギー、家族歴について関連する質問をすることができます。AI ワーカーは自然言語処理技術を使用して収集した情報を解釈し構造化し、電子カルテに正確に記録されるようにします。

症状評価と層別化：AI ワーカーは、患者の現在の症状、期間、重症度、関連する要因について質問することで、初期症状評価を実施できます。医学知識ベースと機械学習モデルを活用することで、これらのワーカーは提供された情報を分析し、予備的な鑑別診断を生成したり、医療提供者との相談予約や自己管理措置などの適切な次のステップを推奨したりできます。

保険確認：AI ワーカーは患者受付時の保険確認を支援できます。患者の保険情報を収集し、API や Web サービスを通じて保険会社と通信し、補償適格性と給付内容

を確認できます。この自動化により、管理負担を軽減し、正確な情報の取得を確保しながら、保険確認プロセスを効率化します。

患者教育と指示：AI ワーカーは、特定の病状や予定される処置に基づいて、関連する教育資料と指示を患者に提供できます。パーソナライズされたコンテンツを提供し、一般的な質問に答え、予約前の準備、投薬指示、または治療後のケアに関するガイダンスを提供できます。これにより、医療の過程全体を通じて患者の理解度と参加度を高めることができます。

AI ワーカーを患者受付に活用することで、医療機関は効率性を向上させ、待ち時間を短縮し、全体的な患者体験を改善できます。これらのワーカーは日常的なタスクを処理し、正確な情報を収集し、パーソナライズされた支援を提供することで、医療専門家が患者への質の高いケアの提供に集中できるようにします。

患者リスク評価

AI ワーカーは、様々なデータソースを分析し、高度な分析手法を適用することで、患者リスク評価において重要な役割を果たすことができます。

データ統合：AI ワーカーは、電子カルテ、医用画像、検査結果、ウェアラブルデバイス、健康の社会的決定要因など、複数のソースから患者データを収集し、理解することができます。この情報を包括的な患者プロフィールに統合することで、AI ワーカーは患者の健康状態とリスク要因の全体像を提供できます。

リスク層別化：AI ワーカーは予測モデルを使用して、個々の特性と健康データに基づいて患者を異なるリスクカテゴリーに層別化できます。このリスク層別化により、医療提供者はより即座の注意や介入を必要とする患者の優先順位付けが可能になります。例えば、特定の状態について高リスクと特定された患者には、より綿密なモニタリング、予防措置、または早期介入のためのフラグを立てることができます。

パーソナライズされたリスクプロフィール：AI ワーカーは、各患者のリスクスコアに寄与する特定の要因を強調した、パーソナライズされたリスクプロフィールを生成できます。これらのプロフィールには、患者のライフスタイル、遺伝的素因、環境要因、健

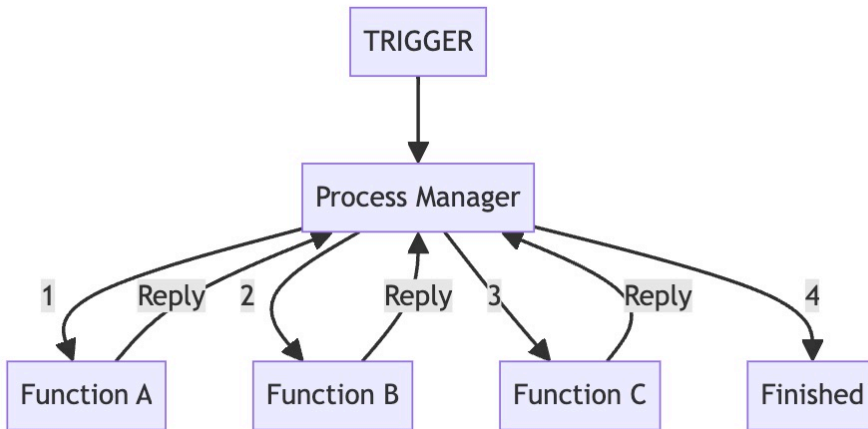
康の社会的決定要因に関する洞察が含まれます。リスク要因の詳細な内訳を提供することで、AI ワーカーは医療提供者が個々の患者のニーズに合わせて予防戦略と治療計画をカスタマイズするのを支援できます。

継続的リスクモニタリング: AI ワーカーは患者データを継続的にモニタリングし、リアルタイムでリスク評価を更新することができます。バイタルサインの変化、検査結果、服薬順守状況など、新しい情報が利用可能になると、AI ワーカーはリスクスコアを再計算し、重要な変化があれば医療提供者に警告を発します。このプロアクティブなモニタリングにより、タイムリーな介入と患者ケア計画の調整が可能になります。

臨床意思決定支援: AI ワーカーはリスク評価の結果を臨床意思決定支援システムに統合し、医療提供者にエビデンスに基づく推奨事項とアラートを提供することができます。例えば、特定の状態に対する患者のリスクスコアが一定の閾値を超えた場合、AI ワーカーは臨床ガイドラインとベストプラクティスに基づいて、特定の診断検査、予防措置、または治療選択肢を検討するよう医療提供者に促すことができます。

これらのワーカーは膨大な患者データを処理し、高度な分析を適用し、臨床意思決定を支援するための実用的な洞察を生成することができます。これにより最終的に、患者のアウトカムの改善、医療費の削減、そして集団健康管理の向上につながります。

プロセスマネージャーとしての **AI** ワーカー



AI 駆動型アプリケーションのコンテキストにおいて、ワーカーはプロセスマネージャーとして機能するように設計することができます。これは Gregor Hohpe による「Enterprise Integration Patterns」で説明されているものです。プロセスマネージャーは、プロセスの状態を維持し、中間結果に基づいて次の処理ステップを決定する中心的なコンポーネントです。

AI ワーカーがプロセスマネージャーとして動作する場合、プロセスを初期化する着信メッセージを受け取ります。これはトリガーメッセージとして知られています。その後、AI ワーカーはプロセス実行の状態（会話トランスクリプトとして）を維持し、順次または並列に実装できるツール機能として実装された一連の処理ステップを通じて、自身の判断でメッセージを処理します。



GPT-4のような関数を並列実行できる AI モデルのクラスを使用している場合、ワーカーは複数のステップを同時に実行できます。正直なところ、私自身はそれを試したことがなく、結果は場合によって異なる可能性があると感じています。

各個別の処理ステップの後、制御は AI ワーカーに戻され、現在の状態と得られた結果に基づいて次の処理ステップを決定することができます。

トリガーメッセージを保存する

私の経験では、トリガーメッセージをデータベースバックドオブジェクトとして実装することが賢明です。そうすることで、各プロセスインスタンスは一意のプライマリキーで識別され、AI の会話トランスクリプトを含む実行に関連する状態を保存する場所が提供されます。

例えば、以下は Olympia の AccountChange モデルクラスを簡略化したものです。これはユーザーアカウントの変更要求を表しています。

```
1  # == Schema Information
2  #
3  # Table name: account_changes
4  #
5  #   id          :uuid          not null, primary key
6  #   description :string
7  #   state       :string        not null
8  #   transcript  :jsonb
9  #   created_at  :datetime       not null
10 #   updated_at  :datetime       not null
11 #   account_id  :uuid          not null
12 #
13 # Indexes
14 #
15 #   index_account_changes_on_account_id (account_id)
16 #
17 # Foreign Keys
18 #
```

```
19 # fk_rails_... (account_id => accounts.id)
20 #
21 class AccountChange < ApplicationRecord
22   belongs_to :account
23
24   validates :description, presence: true
25
26   after_commit -> {
27     broadcast(:account_change_requested, self)
28   }, on: :create
29
30   state_machine initial: :requested do
31     event :completed do
32       transition all => :complete
33     end
34     event :failed do
35       transition all => :requires_human_review
36     end
37   end
38 end
```

AccountChange クラスは、アカウント変更要求を処理するプロセスを開始するトリガーメッセージとして機能します。create トランザクションのコミットが完了した後、Olympia のWisperベースのパブ/サブサブシステムにブロードキャストされる様子に注目してください。

このようにトリガーメッセージをデータベースに保存することで、各アカウント変更要求の永続的な記録が提供されます。AccountChange クラスの各インスタンスには一意のプライマリキーが割り当てられ、個々の要求を容易に識別して追跡することができます。これは特に監査ログの目的に有用で、システムがすべてのアカウント変更の

履歴(要求された時期、要求された変更内容、各要求の現在の状態など)を維持することを可能にします。

この例では、AccountChange クラスには、要求された変更の詳細を記録するdescription、要求の現在の状態(例:requested、complete、requires_human_review)を表すstate、要求に関連するAIの会話記録を保存するtranscriptなどのフィールドが含まれています。description フィールドは、AIとの最初のチャット完了を開始するために使用される実際のプロンプトです。このデータを保存することで、有用なコンテキストが提供され、アカウント変更プロセスのより良い追跡と分析が可能になります。

トリガーメッセージをデータベースに保存することで、堅牢なエラー処理とリカバリーが可能になります。アカウント変更要求の処理中にエラーが発生した場合、システムは要求を失敗としてマークし、人間による介入が必要な状態に移行します。これにより、要求が失われたり忘れられたりすることがなく、問題を適切に対処して解決することができます。

プロセスマネージャーとしてのAIワーカーは、制御の中心点を提供し、強力なプロセスレポートとデバッグ機能を可能にします。ただし、アプリケーション内のすべてのワークフローシナリオでAIワーカーをプロセスマネージャーとして使用することは、過剰である可能性があることに注意が必要です。

アプリケーションアーキテクチャへのAIワーカーの統合

アプリケーションアーキテクチャにAIワーカーを組み込む際には、AIワーカーと他のアプリケーションコンポーネント間のスムーズな統合と効果的な通信を確保するために、いくつかの技術的な考慮事項に対処する必要があります。このセクションでは、それらのインターフェースの設計、データフローの処理、AIワーカーのライフサイクル管理における重要な側面について考察します。

明確なインターフェースと通信プロトコルの設計

AI ワーカーと他のアプリケーションコンポーネント間のシームレスな統合を実現するために、明確なインターフェースと通信プロトコルを定義することが重要です。以下のアプローチを検討してください：

API ベースの統合： AI ワーカーの機能を、RESTful エンドポイントや GraphQL スキーマなどの明確に定義された API を通じて公開します。これにより、他のコンポーネントは標準的な HTTP リクエストとレスポンスを使用して AI ワーカーと対話することができます。API ベースの統合により、AI ワーカーと消費コンポーネント間の明確な契約が提供され、統合ポイントの開発、テスト、保守が容易になります。

メッセージベースの通信： メッセージキューやパブリッシュ/サブスクライブシステムなどのメッセージベースの通信パターンを実装し、AI ワーカーと他のコンポーネント間の非同期的な対話を可能にします。このアプローチにより、AI ワーカーをアプリケーションの他の部分から分離し、より良いスケーラビリティ、フォールトトレランス、疎結合を実現できます。メッセージベースの通信は、AI ワーカーによる処理が時間やリソースを大量に消費する場合に特に有用で、AI ワーカーのタスク完了を待つことなく、アプリケーションの他の部分が実行を継続することを可能にします。

イベント駆動アーキテクチャ： 特定の条件が満たされたときに AI ワーカーを起動するイベントとトリガーを中心にシステムを設計します。AI ワーカーは関連するイベントをサブスクライブし、それに応じて反応し、イベントが発生したときに指定されたタスクを実行することができます。イベント駆動アーキテクチャにより、リアルタイム処理が可能になり、AI ワーカーをオンデマンドで呼び出すことができ、不要なリソース消費を削減できます。このアプローチは、AI ワーカーが特定のアクションやアプリケーション状態の変更に応答する必要があるシナリオに適しています。

データフローと同期の処理

アプリケーションに AI ワーカーを統合する際には、AI ワーカーと他のコンポーネント間のスムーズなデータフローとデータの一貫性を確保することが重要です。以下の側面を考慮してください：

データの準備：AI ワーカーにデータを送る前に、入力データのクリーニング、フォーマット、変換などの様々なデータ準備タスクを実行する必要がある場合があります。AI ワーカーが効果的に処理できるようにするだけでなく、ワーカーにとって無意味、あるいは最悪の場合は注意を散らすような情報にトークンを浪費しないようにすることも重要です。データ準備には、ノイズの除去、欠損値の処理、データ型の変換などのタスクが含まれる場合があります。

データの永続化：AI ワーカーとの間で流れるデータをどのように保存し、永続化するかを検討する必要があります。データ量、クエリパターン、スケーラビリティなどの要因を考慮してください。監査やデバッグのために AI の「思考プロセス」を反映した記録を永続化する必要があるのか、それとも結果の記録だけで十分なのでしょうか？

データの取得：ワーカーが必要とするデータの取得には、データベースのクエリ、ファイルの読み取り、外部 API へのアクセスが含まれる場合があります。レイテンシーと、AI ワーカーが最新のデータにアクセスする方法を考慮してください。データベースへのフルアクセスが必要なのか、それとも実行する作業に応じてアクセス範囲を狭く定義すべきなののでしょうか？スケーリングについてはどうでしょうか？パフォーマンスを向上させ、基盤となるデータソースへの負荷を軽減するためのキャッシュメカニズムを検討してください。

データ同期：AI ワーカーを含む複数のコンポーネントが共有データにアクセスして変更する場合、データの一貫性を維持するための適切な同期メカニズムを実装することが重要です。楽観的ロックや悲観的ロックなどのデータベースロック戦略は、競合を防ぎデータの整合性を確保するのに役立ちます。関連するデータ操作をグループ化し、ACID 特性（原子性、一貫性、分離性、永続性）を維持するためのトランザクショ

ン管理技術を実装してください。

エラー処理とリカバリー: データフロープロセス中に発生する可能性のあるデータ関連の問題に対処するため、堅牢なエラー処理とリカバリーメカニズムを実装してください。例外を適切に処理し、デバッグに役立つ意味のあるエラーメッセージを提供してください。一時的な障害やネットワークの中断に対処するためのリトライメカニズムとフォールバック戦略を実装してください。データの破損や損失が発生した場合のデータ回復と復元の明確な手順を定義してください。

データフローと同期メカニズムを慎重に設計・実装することで、AI ワーカーが正確で一貫性のある最新のデータにアクセスできるようになります。これにより、タスクを効果的に実行し、信頼性の高い結果を生成することが可能になります。

AI ワーカーのライフサイクル管理

AI ワーカーの初期化と設定のための標準化されたプロセスを開発してください。モデル名、システム指示、関数定義などの設定を標準化するフレームワークを私は好んで使用しています。デプロイメントとスケーリングを容易にするため、初期化プロセスを自動化し再現可能にしてください。

AI ワーカーの健全性とパフォーマンスを追跡するための包括的なモニタリングとロギングメカニズムを実装してください。リソース使用率、処理時間、エラー率、スループットなどのメトリクスを収集してください。ELK スタック (Elasticsearch、Logstash、Kibana) のような集中型ロギングシステムを使用して、複数の AI ワーカーからのログを集約・分析してください。

AI ワーカーアーキテクチャにフォールトトレランスとレジリエンスを組み込んでください。障害や例外を適切に処理するためのエラー処理とリカバリーメカニズムを実装してください。大規模言語モデルは依然として最先端の技術であり、プロバイダーは予期せぬタイミングでダウンすることがよくあります。カスケード障害を防ぐためにリトライメカニズムとサーキットブレーカーを使用してください。

AI ワーカーの構成性とオーケストレーション

AI ワーカーアーキテクチャの主な利点の 1 つは、その構成性にあります。これにより、複雑な問題を解決するために複数の AI ワーカーを組み合わせてオーケストレーションすることが可能になります。大規模なタスクを、専門の AI ワーカーが処理する小さな管理可能なサブタスクに分解することで、強力で柔軟なシステムを作成できます。このセクションでは、「多数の」AI ワーカーを構成およびオーケストレーションするための様々なアプローチを探ります。

マルチステップワークフローのための AI ワーカーのチェーン化

多くのシナリオでは、複雑なタスクを一連の順次ステップに分解でき、1 つの AI ワーカーの出力が次のワーカーの入力となります。この AI ワーカーのチェーン化により、マルチステップワークフローまたはパイプラインが作成されます。チェーン内の各 AI ワーカーは特定のサブタスクに焦点を当て、最終的な出力はすべてのワーカーの協調した努力の結果となります。

Ruby on Railsアプリケーションでユーザー生成コンテンツを処理する例を考えてみましょう。以下のステップを含むワークフローは、実際の使用ケースではこのように分解するほど単純ではないかもしれませんが、例を理解しやすくするためのものです：

1. **テキストクリーンアップ**：HTML タグの削除、テキストの小文字化、Unicode の正規化を担当する AI ワーカー。
2. **言語検出**：クリーンアップされたテキストの言語を識別する AI ワーカー。
3. **感情分析**：検出された言語に基づいてテキストの感情（ポジティブ、ネガティブ、またはニュートラル）を判断する AI ワーカー。
4. **コンテンツカテゴリ分類**：自然言語処理技術を使用してテキストを事前定義されたカテゴリに分類する AI ワーカー。

以下は、Ruby を使用してこれらの AI ワーカーをチェーン化する非常に簡略化された例です：


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

この例では、ContentProcessor クラスが生のテキストで初期化され、process メソッド内で AI ワーカーを連鎖させています。各 AI ワーカーは特定のタスクを実行し、その結果をチェーン内の次のワーカーに渡します。最終的な出力は、クリーニングされたテキスト、検出された言語、感情分析結果、およびコンテンツカテゴリを含むハッシュとなります。

独立した AI ワーカーの並列処理

前の例では、AI ワーカーが順次連鎖されており、各ワーカーがテキストを処理してその結果を次のワーカーに渡していました。しかし、同じ入力に対して独立して動作できる複数の AI ワーカーがある場合、それらを並列で処理することでワークフローを最適化できます。

このシナリオでは、TextCleanupWorker によってテキストのクリーニングが実行された後、LanguageDetectionWorker、SentimentAnalysisWorker、およびCategorizationWorker は、すべてクリーニングされたテキストを独立して処理でき

ます。これらのワーカーを並列で実行することで、全体の処理時間を潜在的に削減し、ワークフローの効率を改善できます。

Rubyで並列処理を実現するには、スレッドや非同期プログラミングなどの並行性技術を活用できます。以下は、スレッドを使用して最後の3つのワーカーを並列で処理するようにContentProcessor クラスを修正する例です：

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

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

この最適化されたバージョンでは、`concurrent-ruby` ライブラリを使用して、独立した AI ワーカーそれぞれに対して `Concurrent::Future` オブジェクトを作成します。`Future` は、別のスレッドで非同期に実行される処理を表します。

テキストのクリーンアップステップの後、3 つの `Future` オブジェクト (`language_future`、`sentiment_future`、`category_future`) を作成します。各 `Future` は、対応する AI ワーカー (`LanguageDetectionWorker`、`SentimentAnalysisWorker`、`CategorizationWorker`) を別のスレッドで実行し、`cleaned_text` を入力として渡します。

各 `Future` の `value` メソッドを呼び出すことで、処理の完了を待ち、結果を取得します。`value` メソッドは結果が利用可能になるまでブロックするため、すべての並列ワーカーの処理が完了してから次の処理に進むことが保証されます。

最後に、元の例と同様に、クリーンアップされたテキストと並列ワーカーからの結果を含む出力ハッシュを構築します。

独立した AI ワーカーを並列に処理することで、逐次実行と比較して全体の処理時間を潜在的に削減できます。この最適化は、特に時間のかかるタスクや大量のデータを処理する場合に有効です。

ただし、実際のパフォーマンス向上は、各ワーカーの複雑さ、利用可能なシステムリソース、スレッド管理のオーバーヘッドなど、さまざまな要因に依存することに注意が必要です。特定のユースケースに最適な並列度を決定するために、常にベンチマークとプロファイリングを行うことをお勧めします。

さらに、並列処理を実装する際は、ワーカー間の共有リソースや依存関係に注意を払う必要があります。ワーカーが競合や競合状態なしに独立して動作できることを確認

してください。依存関係や共有リソースがある場合は、データの整合性を維持し、デッドロックや不整合な結果などの問題を回避するために、適切な同期メカニズムを実装する必要があるかもしれません。

Ruby のグローバルインタプリタロックと非同期処理

Ruby で非同期のスレッドベース処理を考える際には、Ruby のグローバルインタプリタロック(GIL)の影響を理解することが重要です。

GIL は、マルチコアプロセッサ上であっても、一度に 1 つのスレッドのみが Ruby コードを実行できるようにする Ruby インタプリタのメカニズムです。つまり、Ruby プロセス内で複数のスレッドを作成して管理できますが、任意の時点で実際に Ruby コードを実行できるのは 1 つのスレッドだけということです。

GIL は、Ruby インタプリタの実装を簡素化し、Ruby の内部データ構造のスレッドセーフティを提供するように設計されています。ただし、これによって Ruby コードの真の並列実行の可能性が制限されます。

`concurrent-ruby` ライブラリや組み込みの `Thread` クラスなどを使用して Ruby でスレッドを使用する場合、スレッドは GIL の制約を受けます。GIL は各スレッドに短時間のタイムスライスを与えて Ruby コードを実行し、その後別のスレッドに切り替えることで、並行実行の錯覚を作り出します。

しかし、GIL の影響により、Ruby コードの実際の実行は逐次的なままです。1 つのスレッドが Ruby コードを実行している間、他のスレッドは実質的に一時停止し、GIL を獲得してコードを実行する順番を待っています。

これは、Ruby のスレッドベースの非同期処理が、外部 API のレスポンス待ち(サードパーティがホストする大規模言語モデルなど)やファイル I/O 操作などの I/O 制約のタスクに最も効果的であることを意味します。スレッドが I/O 操作に遭遇すると、GIL を解放でき、I/O の完了を待つ間に他のスレッドが実行できます。

一方、集中的な計算や長時間実行される AI ワーカー処理などの CPU 制約のタ

スクでは、GIL がスレッドベースの並列処理による潜在的なパフォーマンス向上を制限する可能性があります。一度に 1 つのスレッドしか Ruby コードを実行できないため、全体の実行時間は逐次処理と比較して大幅に短縮されない可能性があります。

Ruby で CPU 制約のあるタスクの真の並列実行を実現するには、以下のような代替アプローチを検討する必要があるかもしれません：

- 複数の CPU コアで実行される複数の Ruby プロセスを使用した、プロセスベースの並列処理を活用する。
- C や Rust などの GIL のない言語へのネイティブ拡張機能やインターフェースを提供する外部ライブラリやフレームワークを活用する。
- 複数のマシンやプロセス間でタスクを分散させるための分散コンピューティングフレームワークやメッセージキューを利用する。

Ruby で非同期処理を設計・実装する際は、タスクの性質と GIL による制約を慎重に考慮することが重要です。スレッドベースの非同期処理は I/O 制約のあるタスクには利点をもたらす可能性がありますが、GIL の制約により、CPU 制約のあるタスクでは大幅なパフォーマンス向上が見込めない可能性があります。

精度向上のためのアンサンブル手法

アンサンブル手法は、複数の AI ワーカーの出力を組み合わせて、システム全体の精度や堅牢性を向上させる手法です。単一の AI ワーカーに依存する代わりに、アンサンブル手法は複数のワーカーの集合知を活用して、より適切な判断を行います。



アンサンブルは、ワークフローの異なる部分が異なる AI モデルで最適に動作する場合に特に重要です。これは思われている以上に一般的なケースです。GPT-4のような強力なモデルは、機能の劣るオープンソースの選択肢と比べて非常に高価であり、アプリケーションのワークフローの全ステップで必要とされるわけではありません。

一般的なアンサンブル手法の一つに多数決方式があります。これは複数の AI ワーカーが独立して同じ入力を処理し、最終的な出力を多数派の合意によって決定する方法です。このアプローチにより、個々のワーカーのエラーの影響を軽減し、システム全体の信頼性を向上させることができます。

例として、感情分析のための 3 つの AI ワーカーがあり、それぞれが異なるモデルを使用するか、異なるコンテキストを提供されている場合を考えてみましょう。多数決方式を使用してそれらの出力を組み合わせることで、最終的な感情予測を決定することができます。

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

```
16         .first
17
18     end
19 end
```

この例では、SentimentAnalysisEnsembleクラスはテキストで初期化され、3つの異なる感情分析 AI ワーカーを呼び出します。analyze メソッドは各ワーカーから予測を収集し、group_by と max_by メソッドを使用して多数決で感情を判定します。最終的な出力は、ワーカーのアンサンブルから最も多くの票を獲得した感情となります。



アンサンブルは明らかに、並列処理の実験に時間を費やす価値がある場合です。

AI ワーカーの動的選択と呼び出し

多くの場合、呼び出すべき特定の AI ワーカーは、実行時の条件やユーザー入力に依存する可能性があります。AI ワーカーの動的選択と呼び出しにより、システムの柔軟性と適応性が確保されます。



単一の AI ワーカーに多くの機能を詰め込もうとする誘惑に駆られるかもしれません。多くの関数を持たせ、それらの呼び出し方を説明する複雑なプロンプトを用意しようとするでしょう。しかし、その誘惑には抗ってください。私の言葉を信じてください。この章で説明しているアプローチが「多数のワーカー」と呼ばれる理由の 1 つは、より大きな目的のために、それぞれが小さな仕事を担当する、特化した多くのワーカーを持つことが望ましいということを私たちに思い出させるためです。

例えば、異なる AI ワーカーが異なるタイプのユーザークエリを処理する責任を持つチャットボットアプリ。ユーザーの入力に基づいて、アプリケーションは適切な AI ワーカーを動的に選択してクエリを処理します。

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

この例では、ChatbotController が process_query アクションを通じてユーザークエリを受け取ります。まず QueryClassifierWorker を使用してクエリの種類を判定します。分類されたクエリタイプに基づいて、コントローラーは適切な AI ワーカーを動的に選択して応答を生成します。この動的な選択により、チャットボットは異なる種類のクエリを処理し、関連する AI ワーカーに振り分けることができます。



QueryClassifierWorker の作業は比較的シンプルで、多くのコンテキストや関数定義を必要としないため、[mistralai/mixtral-8x7b-instruct:nitro](#)のような超高速な小規模 LLM を使用して実装することができます。このモデルは多くのタスクで GPT-4 に近い能力を持ち、この執筆時点では、Groq を使用すると驚異的な 444 トークン/秒のスループットで処理することができます。

従来の自然言語処理と LLM の組み合わせ

大規模言語モデル (LLM) は自然言語処理 (NLP) の分野に革命をもたらし、幅広いタスクにおいて比類のない汎用性とパフォーマンスを提供していますが、すべての問題に対して最も効率的かつコスト効果の高い解決策とは限りません。多くの場合、従来の NLP 技術と LLM を組み合わせることで、特定の NLP の課題に対してより最適化された、的確で経済的なアプローチを実現できます。

LLM は、NLP のスイスアーミーナイフのようなものだと考えてください。非常に汎用的で強力ですが、必ずしもすべての作業に最適なツールとは限りません。コルク抜きや缶切りのような専用ツールの方が、特定のタスクにはより効果的で効率的な場合があります。同様に、文書クラスタリング、トピック識別、分類などの従来の NLP 技術は、NLP パイプラインの特定の側面に対してより的確でコスト効果の高いソリューションを提供できることが多いのです。

従来の NLP 技術の主な利点の 1 つは、その計算効率です。単純な統計モデルやルールベースのアプローチに基づくこれらの手法は、LLM と比較してはるかに少ない計算オーバーヘッドで大量のテキストデータを処理できます。これは、類似した記事のクラスタリングやテキストコレクション内の主要なトピックの識別など、大規模な文書コーパスの分析と整理を含むタスクに特に適しています。

さらに、従来の NLP 技術は、特にドメイン固有のデータセットで訓練された場合、特定のタスクに対して高い精度と正確性を達成できることが多いです。例えば、サポートベクターマシン (SVM) やナイーブベイズなどの従来の機械学習アルゴリズムを使

用して適切に調整された文書分類器は、最小限の計算コストで文書を事前定義されたカテゴリに正確に分類できます。

しかし、言語、コンテキスト、推論のより深い理解を必要とするタスクにおいて、LLM は真価を発揮します。一貫性があり文脈に即したテキストを生成し、質問に答え、長い文章を要約する能力は、従来の NLP 手法では及びません。LLM は、曖昧さ、共参照、慣用表現などの複雑な言語現象を効果的に処理でき、自然言語生成や理解を必要とするタスクには不可欠です。

真の力は、従来の NLP 技術と LLM を組み合わせてハイブリッドアプローチを作り出し、両者の長所を活用することにあります。文書の前処理、クラスタリング、トピック抽出などのタスクに従来の NLP 手法を使用することで、テキストデータを効率的に整理し構造化できます。この構造化された情報を、要約の生成、質問への回答、包括的なレポートの作成などのより高度なタスクのために LLM に入力することができます。

例えば、大量の個別のトレンド文書に基づいて特定のドメインのトレンドレポートを生成したい場合を考えてみましょう。大量のテキストを処理するのに計算コストが高く時間がかかる LLM だけに頼るのではなく、ハイブリッドアプローチを採用することができます：

1. トピックモデリング(潜在的ディリクレ配分法など)やクラスタリングアルゴリズム(K 平均法など)といった伝統的な自然言語処理技術を使用して、類似したトレンド文書をグループ化し、コーパス内の主要なテーマやトピックを特定します。
2. クラスタリングされた文書と特定されたトピックを大規模言語モデルに入力し、その優れた言語理解・生成能力を活用して、各クラスターやトピックの一貫性のある有益な要約を作成します。
3. 最後に、大規模言語モデルを使用して個別の要約を組み合わせ、最も重要なトレンドを強調し、集約された情報に基づく洞察と推奨事項を提供する包括的なトレンドレポートを生成します。

このように伝統的な自然言語処理技術と大規模言語モデルを組み合わせることで、

大量のテキストデータを効率的に処理し、意味のある洞察を抽出し、計算資源とコストを最適化しながら高品質なレポートを生成することができます。

自然言語処理プロジェクトに着手する際は、各タスクの具体的な要件と制約を慎重に評価し、伝統的な自然言語処理手法と大規模言語モデルをどのように組み合わせれば最良の結果が得られるかを検討することが重要です。伝統的な技術の効率性と精度を、大規模言語モデルの汎用性とパワーと組み合わせることで、ユーザーやステークホルダーに価値を提供する、高度に効果的で経済的な自然言語処理ソリューションを作成することができます。

ツールの使用



AI 駆動型アプリケーション開発の分野において、「ツールの使用」または「関数呼び出し」という概念は、LLMを外部ツール、API、関数、データベース、その他のリソースに接続することを可能にする強力な手法として台頭してきました。このアプローチにより、単にテキストを出力するだけでなく、AI コンポーネントとアプリケーションのエコシステムの他の部分との間でより動的な相互作用が可能になります。この章で検討するように、ツールの使用により、AI モデルに構造化された方法でデータを生成させるオプションも得られます。

ツールの使用とは

ツールの使用、別名関数呼び出しは、開発者が生成プロセス中に LLM が相互作用できる関数のリストを指定できるようにする手法です。これらのツールは、シンプルなユーティリティ関数から複雑な API やデータベースクエリまで多岐にわたります。

LLM にこれらのツールへのアクセスを提供することで、開発者はモデルの機能を拡張し、外部の知識やアクションを必要とするタスクを実行できるようになります。

図 8. ドキュメントを分析する AI ワーカーの関数定義の例

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

27 `} .freeze`

ツール使用の背後にある重要な考え方は、LLMにユーザーの入力やタスクに基づいて適切なツールを動的に選択し実行する能力を与えることです。モデルの事前学習された知識のみに依存するのではなく、ツール使用によって LLM は外部リソースを活用し、より正確で関連性が高く、実用的な応答を生成することができます。ツール使用により、RAG(検索拡張生成)などの技術は、そうでない場合と比べてはるかに実装が容易になります。

特に断りのない限り、本書では AI モデルにはビルトインのサーバーサイドツールへのアクセスがないことを前提としています。AI が利用できるツールは、すべて API リクエストごとにあなたが明示的に宣言する必要があり、AI がその応答でそのツールの使用を希望した場合の実行のための準備も含めて宣言する必要があります。

ツール使用の可能性

ツール使用は、AI 駆動のアプリケーションに幅広い可能性を開きます。以下に、ツール使用で実現できることの例を示します：

1. **チャットボットとバーチャルアシスタント**: LLM を外部ツールに接続することで、チャットボットとバーチャルアシスタントはデータベースからの情報取得、API コールの実行、他のシステムとの対話など、より複雑なタスクを実行できます。例えば、チャットボットは CRM ツールを使用して、ユーザーのリクエストに基づいて取引のステータスを変更することができます。
2. **データ分析とインサイト**: LLM をデータ分析ツールやライブラリに接続して、高度なデータ処理タスクを実行することができます。これにより、アプリケーショ

ンはユーザーのクエリに基づいてインサイトを生成し、比較分析を実施し、データに基づく推奨事項を提供することができます。

3. **検索と情報検索:** ツール使用により、LLM は検索エンジン、ベクターデータベース、その他の情報検索システムと対話することができます。ユーザーのクエリを検索クエリに変換することで、LLM は複数のソースから関連情報を取得し、ユーザーの質問に対して包括的な回答を提供できます。
4. **外部サービスとの統合:** ツール使用により、AI 駆動のアプリケーションと外部サービスや API のシームレスな統合が可能になります。例えば、LLM は天気 API とやり取りしてリアルタイムの天気更新を提供したり、翻訳 API を使用して多言語の応答を生成したりすることができます。

ツールの使用のワークフロー

ツールの使用のワークフローは通常、4 つの重要なステップを含みます:

1. リクエストコンテキストに関数定義を含める
2. 動的(または明示的な)ツール選択
3. 関数の実行
4. オプションで元のプロンプトの続行

これらの各ステップを詳しく見ていきましょう。

リクエストコンテキストに関数定義を含める

AI は、完了リクエストの一部としてリストを提供されることで(通常は JSON スキーマのバリエーションを使用して関数として定義される)、使用可能なツールを把握します。

ツール定義の正確な構文はモデルによって異なります。

以下は、Claude 3 で `get_weather` 関数を定義する方法です:

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

そして、同じ関数を GPT-4用に定義する場合は、tools パラメーターの値として次のように渡します：


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

ほとんど同じなのに、なぜか違う!イライラするね。

関数定義では、名前、説明、入力パラメータを指定します。入力パラメータは、列挙型を使用して受け入れ可能な値を制限したり、パラメータが必須かどうかを指定したりするなど、属性を使用してさらに定義することができます。

実際の関数定義に加えて、システムディレクティブには、その機能をシステム内でなぜ、どのように使用するかについての指示やコンテキストを含めることもできます。

例えば、Olympiaの中の Web 検索ツールには、AIが言及されたツールを使用できることを念頭に置くよう促すシステムディレクティブが含まれています：

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

詳細な説明の提供は、ツールのパフォーマンスにおいて最も重要な要素とされています。ツールの説明には以下のような詳細をすべて含める必要があります：

- ツールの機能
- ツールを使用すべき時(および使用すべきでない時)
- 各パラメータの意味とツールの動作への影響
- ツールの実装に関する重要な注意事項や制限事項

AI にツールについてより多くのコンテキストを提供できればできるほど、AI はツールをいつどのように使用するかをより適切に判断できるようになります。例えば、Anthropicは、Claude 3シリーズにおいて、ツール 1 つにつき最低でも 3～4 文の説明を推奨しており、複雑なツールの場合はそれ以上を推奨しています。

直感的ではないかもしれませんが、説明は例示よりも重要とされています。ツールの使用例は説明やプロンプトに含めることができますが、これはツールの目的とパラメータについての明確で包括的な説明ほど重要ではありません。例示を追加するのは、説明を十分に充実させた後にすべきです。

以下は Stripeのような API 関数仕様の例です：

```
1  {
2    "name": "createPayment",
3    "description": "Create a new payment request",
4    "parameters": {
5      "type": "object",
6      "properties": {
7        "transaction_amount": {
8          "type": "number",
9          "description": "The amount to be paid"
10       },
11       "description": {
12         "type": "string",
13         "description": "A brief description of the payment"
14       },
15       "payment_method_id": {
16         "type": "string",
17         "description": "The payment method to be used"
18       },
19       "payer": {
20         "type": "object",
21         "description": "Information about the payer, including their name,
22                        email, and identification number",
23         "properties": {
24           "name": {
25             "type": "string",
26             "description": "The payer's name"
27           },
28           "email": {
29             "type": "string",
```

```
30         "description": "The payer's email address"
31     },
32     "identification": {
33         "type": "object",
34         "description": "The payer's identification number",
35         "properties": {
36             "type": {
37                 "type": "string",
38                 "description": "Identification document (e.g. CPF, CNPJ)"
39             },
40             "number": {
41                 "type": "string",
42                 "description": "The identification number"
43             }
44         },
45         "required": [ "type", "number" ]
46     }
47 },
48 "required": [ "name", "email", "identification" ]
49 }
50 }
51 }
```



実際には、一部のモデルはネストされた関数仕様や、配列や辞書などの複雑な出力データ型の処理に苦勞することがあります。しかし理論的には、任意の深さの JSON スキーマ仕様を提供できるはずです！

動的ツール選択

ツール定義を含むチャット補完を実行すると、LLM は最も適切なツールを動的に選択し、各ツールに必要な入力パラメータを生成します。

実際には、AI が正確に適切な関数を呼び出し、正確に入力の仕様に従う能力は、成功したり失敗したりします。温度ハイパーパラメータを 0.0 まで下げると大きく改善されますが、私の経験では、それでも時々エラーが発生します。これらの失敗には、実在しない関数名の生成や、パラメータ名の誤り、あるいは単にパラメータの欠落などが含まれます。パラメータは JSON として渡されるため、切り捨てられたり、引用符が正しくなかったり、その他の形で破損した JSON によってエラーが発生することがあります。



自己修復データパターンは、構文エラーにより破損した関数呼び出しを自動的に修正するのに役立ちます。

強制（または明示的）ツール選択

一部のモデルでは、リクエストのパラメータとして特定の関数の呼び出しを強制するオプションが提供されています。そうでない場合、関数を呼び出すかどうかは完全に AI の判断に委ねられます。

関数呼び出しを強制する機能は、AI の動的選択プロセスに関係なく、特定のツールや関数の実行を確実にしたい特定のシナリオで重要です。この機能が重要である理由はいくつかあります：

1. **明示的な制御:** AI を独立コンポーネントとして使用する場合や、特定の時点で特定の関数の実行が必要な事前定義されたワークフローで使用する場合があります。呼び出しを強制することで、AI に丁寧をお願いする代わりに、必要な関数が確実に呼び出されます。

2. **デバッグとテスト:** AI 駆動のアプリケーションの開発とテストにおいて、関数呼び出しを強制できる機能はデバッグ目的で非常に重要です。特定の関数を明示的にトリガーすることで、アプリケーションの個々のコンポーネントを分離してテストできます。これにより、関数の実装の正確性を検証し、入力パラメータを検証し、期待される結果が返されることを確認できます。
3. **エッジケースの処理:** AI の動的選択プロセスでは関数を実行すべきでないと判断されるかもしれませんが、外部プロセスに基づいてその必要性がわかっているような、エッジケースや例外的なシナリオが存在する場合があります。このような場合、関数呼び出しを強制できる機能により、これらの状況を明示的に処理できます。AI の判断をオーバーライドするタイミングを決定するルールや条件をアプリケーションロジックで定義します。
4. **一貫性と再現性:** 特定の順序で実行する必要がある一連の関数がある場合、呼び出しを強制することで、毎回同じ順序で実行されることが保証されます。これは、金融システムや科学的シミュレーションなど、一貫性と予測可能な動作が重要なアプリケーションで特に重要です。
5. **パフォーマンスの最適化:** 場合によっては、関数呼び出しを強制することでパフォーマンスの最適化につながる場合があります。特定のタスクに特定の関数が必要であり、AI の動的選択プロセスが不要なオーバーヘッドを引き起こす可能性がある場合、選択プロセスをバイパスして必要な関数を直接呼び出すことができます。これにより、レイテンシーを削減し、アプリケーション全体の効率を改善できます。

まとめると、AI 駆動のアプリケーションにおける関数呼び出しの強制機能は、明示的な制御を提供し、デバッグとテストを支援し、エッジケースを処理し、一貫性と再現性を確保します。これは強力なツールですが、この重要な機能についてもう一つの側面を議論する必要があります。



多くの意思決定のユースケースでは、モデルに常に関数呼び出しを実行させたい場合があり、モデルの内部知識のみで応答させたくない場合があります。例えば、異なるタスク(多言語入力、数学など)に特化した複数のモデル間でルーティングを行う場合、関数呼び出しモデルを使用してヘルパーモデルの1つにリクエストを委譲し、独立して応答させないようにすることができます。

ツール選択パラメータ

GPT-4や関数呼び出しをサポートする他の言語モデルでは、補完の一部としてツールの使用を制御するための`tool_choice` パラメータが提供されています。このパラメータには3つの可能な値があります:

- `auto` は AI にツールを使用するか単に応答するか完全な裁量を与えます
- `required` は AI に応答する代わりにツールを**必ず**呼び出すように指示しますが、ツールの選択は AI に任せます
- 3つ目のオプションは、強制的に使用したい`name_of_function` をパラメータとして設定することです。これについては次のセクションで詳しく説明します。



`tool_choice` を`required` に設定すると、提供された関数の中から最も関連性の高い関数を選択することを強制されることに注意してください。これは、プロンプトに本当に適した関数がない場合でも同様です。本書の執筆時点では、空の`tool_calls` レスポンスを返したり、適切な関数が見つからなかったことを通知したりするような方法を持つモデルは知られていません。

構造化出力を得るための関数の強制

関数呼び出しを強制する機能により、プレーンテキストの応答から自分で抽出する代わりに、チャット補完から構造化データを強制的に取得する方法が提供されます。

なぜ構造化出力を得るために関数を強制することが重要なのでしょうか？簡単に言えば、LLMの出力から構造化データを抽出することは面倒だからです。XML形式でデータを要求することで多少は楽になりますが、それでもXMLを解析する必要があります。そして、AIが「申し訳ありませんが、ブラブラブラの理由でリクエストされたデータを生成できません...」と応答した場合、そのXMLが欠落している時はどうすればよいのでしょうか。

このようにツールを使用する場合：

- リクエストで定義するツールは1つだけにすべきでしょう
- `tool_choice` パラメータを使用して関数の使用を強制することを忘れないでください
- モデルがツールに入力を渡すことを忘れないでください。そのため、ツールの名前と説明はあなたの視点ではなく、モデルの視点から書く必要があります

この最後の点は、明確にするために例が必要です。ユーザーテキストの感情分析をAIに依頼する場合を考えてみましょう。関数の名前は`analyze_sentiment`ではなく、`save_sentiment_analysis`のようになります。感情分析を行うのはAIであり、ツールではありません。ツールがしていること(AIの視点から見ると)は、分析結果を保存することだけです。

以下は、Claude 3を使用して画像の要約を整形されたJSONに記録する例で、今回はコマンドラインで`curl`を使用しています：


```
1  curl https://api.anthropic.com/v1/messages \
2      --header "content-type: application/json" \
3      --header "x-api-key: $ANTHROPIC_API_KEY" \
4      --header "anthropic-version: 2023-06-01" \
5      --header "anthropic-beta: tools-2024-04-04" \
6      --data \
7      '{
8          "model": "claude-3-sonnet-20240229",
9          "max_tokens": 1024,
10         "tools": [{
11             "name": "record_summary",
12             "description": "Record summary of image into well-structured JSON.",
13             "input_schema": {
14                 "type": "object",
15                 "properties": {
16                     "key_colors": {
17                         "type": "array",
18                         "items": {
19                             "type": "object",
20                             "properties": {
21                                 "r": {
22                                     "type": "number",
23                                     "description": "red value [0.0, 1.0]"
24                                 },
25                                 "g": {
26                                     "type": "number",
27                                     "description": "green value [0.0, 1.0]"
28                                 },
29                                 "b": {
```

```
30         "type": "number",
31         "description": "blue value [0.0, 1.0]"
32     },
33     "name": {
34         "type": "string",
35         "description": "Human-readable color name
36                         in snake_case, e.g.
37                         \"olive_green\"or
38                         \"turquoise\""
39     }
40 },
41     "required": [ "r", "g", "b", "name" ]
42 },
43     "description": "Key colors in the image. Four or less."
44 },
45     "description": {
46         "type": "string",
47         "description": "Image description. 1-2 sentences max."
48     },
49     "estimated_year": {
50         "type": "integer",
51         "description": "Estimated year that the image was taken,
52                         if is it a photo. Only set this if the
53                         image appears to be non-fictional.
54                         Rough estimates are okay!"
55     }
56 },
57     "required": [ "key_colors", "description" ]
58 }
```

```
59     }],
60     "messages": [
61         {
62             "role": "user",
63             "content": [
64                 {
65                     "type": "image",
66                     "source": {
67                         "type": "base64",
68                         "media_type": "'$IMAGE_MEDIA_TYPE'",
69                         "data": "'$IMAGE_BASE64'"
70                     }
71                 },
72                 {
73                     "type": "text",
74                     "text": "Use `record_summary` to describe this image."
75                 }
76             ]
77         }
78     ]
79 }'
```

この例では、Anthropicの Claude 3モデルを使用して、画像の構造化された JSON サマリーを生成しています。以下がその仕組みです：

1. リクエストペイロードのtools 配列内で、record_summary という単一のツールを定義します。このツールは、画像のサマリーを適切に構造化された JSON として記録する役割を担います。
2. record_summary ツールには、JSON 出力の期待される構造を指定するinput_schema があります。これは以下の 3 つのプロパティを定義します：

- `key_colors`: 画像内の主要な色を表すオブジェクトの配列。各色オブジェクトは、赤、緑、青の値(0.0 から 1.0 の範囲)とスネークケース形式の人間が読める色名を持つプロパティを持ちます。
 - `description`: 画像の簡潔な説明を表す文字列プロパティで、1-2 文に制限されています。
 - `estimated_year`: ノンフィクションの写真と思われる場合、撮影年を推定する任意の整数プロパティです。
3. `messages` 配列では、画像データを base64 エンコードされた文字列としてメディアタイプと共に提供します。これによりモデルは入力の一部として画像を処理できます。
 4. また、Claude に `record_summary` ツールを使用して画像を説明するよう指示します。
 5. リクエストが Claude 3 モデルに送信されると、モデルは画像を分析し、指定された `input_schema` に基づいて JSON サマリーを生成します。モデルは主要な色を抽出し、簡単な説明を提供し、該当する場合は画像の撮影年を推定します。
 6. 生成された JSON サマリーは `record_summary` ツールのパラメータとして渡され、画像の主要な特徴を構造化された形式で表現します。

`record_summary` ツールを明確に定義された `input_schema` と共に使用することで、プレーンテキストの抽出に頼ることなく、画像の構造化された JSON サマリーを取得できます。このアプローチにより、出力が一貫した形式に従い、アプリケーションの下流コンポーネントで簡単に解析・処理できることが保証されます。

関数呼び出しを強制し、期待される出力構造を指定できる機能は、AI 駆動のアプリケーションにおけるツール使用の強力な特徴です。これにより、開発者は生成される出力をより細かく制御でき、AI 生成データをアプリケーションのワークフローに統合することが簡単になります。

関数の実行

関数を定義し、AI に指示を出すと、AI はあなたの関数の 1 つを呼び出すべきだと判断します。次に、アプリケーションコードまたは[raix-rails](#)のような Ruby gem を使用している場合は、関数呼び出しとそのパラメータをアプリケーションコード内の対応する実装にディスパッチする時です。

アプリケーションコードは、関数実行の結果をどのように扱うかを決定します。それはラムダ内の 1 行のコードかもしれませんが、外部 API の呼び出しを含むかもしれません。別の AI コンポーネントの呼び出しを含むかもしれませんし、システムの他の部分で数百または数千行のコードを含むかもしれません。それは完全にあなた次第です。

時には関数呼び出しが操作の終わりとなりますが、結果が AI によって継続されるべき思考の連鎖における情報を表す場合、アプリケーションコードは実行結果をチャット記録に挿入し、AI に処理を継続させる必要があります。

例えば、以下はカスタマーサービスのためのインテリジェント・ワークフロー・オーケストレーションの一部として、Olympia の AccountManager が顧客とコミュニケーションを取るために使用する[Raix](#)関数宣言です。

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

```
13         )  
14         "Notified account owner"  
15     end
```

ここで何が起きているのか一見わかりにくいかもしれませんが、詳しく説明していきます。

1. AccountManager クラスはアカウント管理に関連する多くの関数を定義しています。プランの変更やチームメンバーの追加・削除などができます。
2. その最上位の指示では、AccountManager にアカウント変更リクエストの結果をアカウント所有者に通知するよう、notify_account_owner 関数を使用して指示しています。
3. この関数の簡潔な定義には以下が含まれています:

- 名前
- 説明
- パラメータ message: { type: "string" }
- 関数が呼び出されたときに実行されるブロック

関数ブロックの結果でトランスクリプトを更新した後、chat_completion メソッドが再度呼び出されます。このメソッドは、更新された会話トランスクリプトを AI モデルに送り返して、さらなる処理を行う役割を担っています。この処理を[会話ループ](#)と呼びます。

AI モデルが更新されたトランスクリプトを含む新しいチャット補完リクエストを受け取ると、以前に実行された関数の結果にアクセスできます。これらの結果を分析し、意思決定プロセスに組み込み、会話の累積的なコンテキストに基づいて次の応答またはアクションを生成できます。更新されたコンテキストに基づいて追加の関数を実行することもできますし、それ以上の関数呼び出しが必要ないと判断した場合は、元のプロンプトに対する最終的な応答を生成することもできます。

元のプロンプトの任意の継続

ツールの結果を LLM に送り返して元のプロンプトの処理を続けると、AI はそれらの結果を使用して追加の関数を呼び出すか、最終的なプレーンテキストの応答を生成します。



Cohere の [Command-R](#) のような一部のモデルでは、応答で使用したツールを具体的に引用でき、追加の透明性とトレーサビリティを提供できます。

使用しているモデルによって、関数呼び出しの結果は独自の特別な役割を持つトランスクリプトメッセージに存在するか、他の構文で反映されます。しかし重要なのは、AI が次に何をすべきかを決定する際に考慮できるよう、そのデータがトランスクリプトに含まれていることです。



よくある(そして潜在的にコストのかかる)エラーの一つは、チャットを続ける前に関数の結果をトランスクリプトに追加し忘れることです。その結果、AI は最初に関数を呼び出す前とほぼ同じ方法でプロンプトされることになります。つまり、AI の観点からすると、まだ関数を呼び出していないことになります。そのため、また呼び出します。そしてまた。そして中断されるまで永遠に続きます。コンテキストが大きすぎず、モデルが高価すぎなかったことを願いましょう!

ツール使用のベストプラクティス

ツールを最大限活用するために、以下のベストプラクティスを考慮してください。

説明的な定義

各ツールとその入力パラメータに対して、明確で説明的な名前と説明を提供してください。これにより LLM は各ツールの目的と機能をよりよく理解できます。

経験から言えることですが、「命名は難しい」という一般的な知恵がここにも当てはまります。関数の名前や説明の言い回しを変えるだけで、LLM の結果が劇的に変わるのを見してきました。説明を削除することで性能が向上することもあります。

ツール結果の処理

ツールの結果を LLM に渡す際は、それらが適切に構造化され、包括的であることを確認してください。各ツールの出力を表現するために意味のあるキーと値を使用してください。JSON からプレーンテキストまで、さまざまなフォーマットを試してみて、最適なものを見つけてください。

結果インタープリターは、AI を使用して結果を分析し、人間にわかりやすい説明、要約、または重要なポイントを提供することで、この課題に対応します。

エラー処理

LLM がツール呼び出しに対して無効またはサポートされていない入力パラメータを生成する可能性がある場合に備えて、堅牢なエラー処理メカニズムを実装してください。ツール実行中に発生する可能性のあるエラーを適切に処理し、回復できるようにしてください。

エーアイの非常に優れた特徴の 1 つは、エラーメッセージを理解できることです！つまり、手っ取り早いアプローチで作業している場合、ツールの実装で発生した例外を単にキャッチして、エーアイに返すだけで、何が起きたのかを理解してくれるのです！

例として、Olympia におけるグーグル検索の実装の簡略版を見てみましょう：


```
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
```

Olympia での Google 検索は 2 段階のプロセスです。まず検索を実行し、次に結果を要約します。エラーが発生した場合、その種類に関係なく、例外メッセージが AI に送り返されます。このテクニックは、実質的にすべてのインテリジェントエラーハンドリングパターンの基礎となっています。

例えば、GoogleSearch API コールが 503 Service Unavailable 例外で失敗したとします。これは最上位の rescue まで伝播し、エラーの説明が AI に関数呼び出しの結果として送り返されます。ユーザーに空白の画面や技術的なエラーを表示するのではなく、AI は「申し訳ありませんが、現在 Google 検索機能にアクセスできません。よろしければ、後ほど再試行させていただきます」というような応答をします。

これは単なる巧妙な手法のように見えるかもしれませんが、別の種類のエラー、つまり AI が外部 API を呼び出し、API に渡すパラメータを直接制御している場合を考えてみてください。もしかしたら、それらのパラメータの生成方法に誤りがあったかもしれません。外部 API からのエラーメッセージが十分に詳細であれば、エラーメッセージを呼び出し元の AI に返すことで、AI はそれらのパラメータを再考して再試行することができます。自動的に。エラーの種類に関係なく。

通常のコードでこのような堅牢なエラーハンドリングを実現するために何が必要かを考えてみてください。それは事実上不可能です。

反復的改善

LLMが適切なツールを推奨していない、または最適ではない応答を生成している場合、ツールの定義、説明、入力パラメータを繰り返し改善します。観察された動作と望ましい結果に基づいて、ツールのセットアップを継続的に改良し改善します。

1. シンプルなツール定義から始める: 明確で簡潔な名前、説明、入力パラメータでツールを定義することから始めます。最初はツールのセットアップを複雑にしすぎないようにし、核となる機能に焦点を当てます。例えば、感情分析の結果を保存したい場合は、次のような基本的な定義から始めます:

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

2. テストと観察: 初期のツール定義を配置したら、異なるプロンプトでテストを行い、LLMがツールとどのように相互作用するかを観察します。生成された応答の質と関連性に注意を払います。LLM が最適でない応答を生成している場合は、ツール定義を改善する時です。

3. 説明の改善: LLM がツールの目的を誤解している場合は、ツールの説明を改善してみてください。LLM がツールを効果的に使用できるよう、より多くのコンテキスト、例示、または説明を提供します。例えば、感情分析ツールの説明を更新して、分析対象のテキストの感情的なトーンをより具体的に扱うようにすることができます:

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

4. 入力パラメータの調整: LLM がツールに対して無効または無関係な入力パラメータを生成している場合は、パラメータ定義の調整を検討してください。期待される入力フォーマットを明確にするため、より具体的な制約、検証ルール、または例を追加します。
5. フィードバックに基づく改善: ツールのパフォーマンスを継続的に監視し、ユーザーやステークホルダーからフィードバックを収集します。このフィードバックを活用して改善が必要な領域を特定し、ツール定義に反復的な改良を加えています。例えば、ユーザーが分析で皮肉を適切に処理できていないと報告した場合、説明に注記を追加できます:

```
1 {  
2   "name": "save_sentiment_score",  
3   "description": "Analyze the sentiment of a given text and return a sentiment  
4     score between -1 (negative) and 1 (positive). Note: Sarcasm should be  
5     considered negative.",  
6   ...  
7 }
```

ツール定義を観察された動作とフィードバックに基づいて反復的に改良することで、AI 駆動型アプリケーションのパフォーマンスと効果を段階的に向上させることができます。ツール定義は明確で簡潔であり、特定のタスクに焦点を当てたものにすることを忘れないでください。定期的にツールの相互作用をテストし、検証して、望ましい結果と一致していることを確認してください。

ツールの組み合わせと連鎖

これまで暗示的にしか触れていなかったツール使用の最も強力な側面の 1 つは、複数のツールを組み合わせで連鎖させ、複雑なタスクを達成する能力です。ツール定義とその入出力形式を慎重に設計することで、さまざまな方法で組み合わせることができる再利用可能な構成要素を作成できます。

AI 駆動型アプリケーション用のデータ分析パイプラインを構築する例を考えてみましょう。以下のようなツールがあるかもしれません：

1. **DataRetrieval**: 指定された基準に基づいてデータベースや API からデータを取得するツール。
2. **DataProcessing**: 取得したデータに対して計算、変換、または集計を実行するツール。
3. **DataVisualization**: 処理されたデータをチャートやグラフなどのユーザーフレンドリーな形式で表示するツール。

これらのツールを連鎖させることで、関連データを取得し、処理し、結果を意味のある方法で表示する強力なワークフローを作成できます。ツール使用のワークフローは以下ようになります：

1. LLMが特定の製品カテゴリーの販売データに関する洞察を求めるユーザークエリを受け取ります。
2. LLM がDataRetrieval ツールを選択し、データベースから関連する販売データを取得するための適切な入力パラメータを生成します。
3. 取得したデータはDataProcessing ツールに「渡され」、総収益、平均販売価格、成長率などの指標が計算されます。
4. 処理されたデータは次にDataVisualization ツールによって処理され、洞察を視覚的に表現する魅力的なチャートやグラフが作成され、チャートの URL が LLM に返されます。
5. 最後に、LLM はマークダウンを使用してユーザークエリに対する形式化された応答を生成し、可視化されたデータと主要な発見の要約を組み込みます。

これらのツールを組み合わせることで、アプリケーションに容易に統合できるシームレスなデータ分析ワークフローを作成できます。このアプローチの素晴らしい点は、各ツールを個別に開発してテストし、その後さまざまな問題を解決するためにさまざまな方法で組み合わせることができることです。

ツールのスムーズな組み合わせと連鎖を可能にするために、各ツールの入出力形式を明確に定義することが重要です。

例えば、DataRetrieval ツールは、データベース接続の詳細、テーブル名、クエリ条件などのパラメータを受け取り、結果セットを構造化された JSON オブジェクトとして返すかもしれません。DataProcessing ツールは、この JSON オブジェクトを入力として受け取り、変換された JSON オブジェクトを出力として生成します。ツール間のデータフローを標準化することで、互換性と再利用性を確保できます。

ツールエコシステムを設計する際は、アプリケーションの一般的なユースケースに対応するために、異なるツールをどのように組み合わせることができるかを考えてくださ

い。一般的なワークフローやビジネスロジックをカプセル化する高レベルのツールを作成することを検討し、LLM がそれらを効果的に選択して使用しやすくすることを考えてください。

ツール使用の力は、それが提供する柔軟性とモジュール性にあることを忘れないでください。複雑なタスクを小さな再利用可能なツールに分解することで、幅広い課題に対応できる堅牢で適応性のある AI 駆動型アプリケーションを作成できます。

将来の方向性

AI 駆動型アプリケーション開発の分野が進化するにつれて、ツール使用機能のさらなる進歩が期待されます。将来の方向性として以下のようなものが考えられます：

1. **マルチホップツール使用**: LLM は、満足のいく応答を生成するために必要なツールの使用回数を決定できるようになるかもしれません。これには、中間結果に基づく複数回のツール選択と実行が含まれる可能性があります。
2. **事前定義ツール**: AI プラットフォームは、Python インタープリタ、ウェブ検索ツール、一般的なユーティリティ機能など、開発者がすぐに利用できる事前定義ツールのセットを提供することがあります。
3. **シームレスな統合**: ツールの使用がより一般的になるにつれて、AI プラットフォームと一般的な開発フレームワークとの統合が改善され、開発者がアプリケーションにツールの使用を組み込むことがより容易になると予想されます。

ツールの使用は、AI 駆動型アプリケーションにおいて大規模言語モデルの可能性を最大限に引き出すことができる強力な手法です。大規模言語モデルを外部ツールやリソースに接続することで、ユーザーのニーズに適応し、価値のある洞察やアクションを提供できる、よりダイナミックで知的かつ文脈を理解するシステムを作成することができます。

ツールの使用は膨大な可能性を提供する一方で、潜在的な課題や考慮事項に注意を払うことが重要です。重要な側面の 1 つは、ツールの相互作用の複雑さを管理し、システム全体の安定性と信頼性を確保することです。ツールの呼び出しが失敗したり、予期せぬ結果を返したり、パフォーマンスに影響を与えたりするシナリオに対処する必要があります。さらに、ツールの不正使用や悪意のある使用を防ぐために、セキュリティやアクセス制御対策を考慮する必要があります。AI 駆動型アプリケーションの整合性とパフォーマンスを維持するには、適切なエラー処理、ログ記録、監視メカニズムが不可欠です。

自身のプロジェクトでツールの使用の可能性を探る際は、明確な目標を立て、適切に構造化されたツール定義を設計し、フィードバックと結果に基づいて改善を重ねることを忘れないでください。適切なアプローチとマインドセットを持つことで、ツールの使用は AI 駆動型アプリケーションにおいて新しいレベルのイノベーションと価値を解き放つことができます。

ストリーム処理



HTTP経由でデータをストリーミングする、別名サーバー送信イベント(SSE)は、クライアントが明示的にリクエストすることなく、サーバーが利用可能になったデータを継続的にクライアントに送信する仕組みです。AI のレスポンスは徐々に生成されていくため、生成されるに従って AI の出力を表示することで、応答性の高いユーザーエクスペリエンスを提供することが理にかなっています。実際、私が知る限り、すべての AI プロバイダーの API は、補完エンドポイントでストリーミングレスポンスをオプションとして提供しています。

この章が本書の中で[ツールの使用](#)の直後に配置されているのは、ツールの使用と AI のライブレスポンスを組み合わせることの強力さによるものです。これにより、AI がユーザー入力を処理し、必要に応じて様々なツールや関数を利用し、リアルタイムでレスポンスを提供する、動的でインタラクティブな体験が可能になります。

このシームレスな相互作用を実現するには、AI が呼び出すツール関数の呼び出しと、プレーンテキストの出力をエンドユーザーに送信できるストリームハンドラーを作成する必要があります。ツール関数の処理後にループする必要があることが、この作業に興味深い課題を加えています。

ReplyStream の実装

ストリーム処理の実装方法を示すため、この章では Olympia で使用されている ReplyStream クラスの簡略版について詳しく説明します。このクラスのインスタンスは、[ruby-openai](#)や[openrouter](#)などの AI クライアントライブラリで stream パラメータとして渡すことができます。

以下は、Wisper を介して新しいユーザーメッセージの作成をリッスンする Olympia の PromptSubscriber で、ReplyStream をどのように使用しているかを示しています。

```
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...
```

ReplyStream クラスは、それをインスタンス化したプロンプトサブスクライバーへのcontext 参照に加えて、受信データのバッファを格納するためのインスタンス変数と、ストリーム処理中に呼び出された関数名と引数を追跡するための配列を持っています。

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

initialize メソッドは、ReplyStream インスタンスの初期状態を設定し、バッファ、コンテキスト、その他の変数を初期化します。

call メソッドは、ストリーミングデータを処理するためのメインエントリーポイントです。このメソッドは、データのchunk(ハッシュとして表現される)とオプションのbytesize パラメータを受け取ります(この例では使用されていません)。このメソッド

ド内で、クラスはパターンマッチングを使用して、受信したチャンクの構造に基づいて異なるシナリオを処理します。



チャンクに対して`deep_symbolize_keys` を呼び出すことで、文字列ではなくシンボルを操作できるようになり、よりエレガントなパターンマッチングが可能になります。

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

最初にマッチングするパターンは、ツールコールとそれに関連する関数名です。これを検出すると、`f_name` 配列に格納します。モデルは並列関数呼び出しが可能で、複数の関数を同時に実行できるため、関数名はインデックス付き配列に格納します。

並列関数呼び出しとは、AI モデルが複数の関数呼び出しを同時に実行し、これらの関数呼び出しの効果と結果を並列に解決できる機能です。これは特に関数の実行に時間がかかる場合に有用で、APIとのやり取りを減らすことができ、結果としてトークンの消費量を大幅に削減できます。

次に、関数呼び出しに対応する引数をマッピングする必要があります。

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

関数名を処理したのと同様に、引数をインデックス配列に格納します。

次に、サーバーから 1 トークンずつ届くユーザー向けメッセージを探します。これは `new_content` 変数に割り当てられます。また、`finish_reason` の状態も監視する必要があります。これは出力シーケンスの最後のチャンクまでは `nil` となっています。

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

重要なことに、AI モデルプロバイダーから送信されるエラーメッセージを処理するためのパターンマッチ式を追加します。では例外を発生させますが、本番環境ではエラーをログに記録して終了します。

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

case 文の最後の else 節は、前のパターンのいずれにも一致しなかった場合に実行されます。これは単なる安全対策で、が認識できないチャンクを送信してきた場合に、それを検知できるようにするためのものです。

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

`send_to_client` メソッドは、バッファリングされたコンテンツをクライアントに送信する役割を担っています。このメソッドは、バッファが空でないことを確認し、ボットメッセージのコンテンツを更新し、ボットメッセージをレンダリングし、データの永続性を確保するためにコンテンツをデータベースに保存します。

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

`finalize` メソッドは、ストリーム処理が完了した時に呼び出されます。ストリーム中に受信した関数呼び出しがある場合はそれらをディスパッチし、最終的な内容やその他の関連情報でボットメッセージを更新し、関数呼び出し履歴をリセットします。

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

モデルが関数を呼び出すことを決定した場合、その関数呼び出し（名前と引数）を「ディスパッチ」して実行し、`function_call`と`function_result`のメッセージを会話トランスクリプトに追加する必要があります。

私の経験では、ツール実装に依存するのではなく、コードベースの一箇所で関数メッセージの作成を処理する方が良いでしょう。これはより整理された方法であるだけでなく、非常に重要な実践的な理由もあります: AI モデルが関数を呼び出した際、ループ時にトランスクリプトに呼び出しと結果のメッセージが表示されていないと、`_同じ関数を再度呼び出してしまう_`からです。これが永遠に続く可能性があります。AI が完全にステートレスであることを忘れないでください。関数呼び出しを AI にエコーバックしない限り、それらは発生しなかったことになってしまいます。

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



関数を呼び出した後の履歴をクリアすることは、呼び出しと結果がトランスクリプトに記録されることと同じくらい重要です。そうしないと、ループするたびに同じ関数を何度も呼び出し続けることになってしまいます。

「会話ループ」

PromptSubscriber クラスでは、PromptDeclarations モジュールのprompt メソッドを使用して会話ループの動作を定義しています。until パラメータは-> { bot_message.complete? } に設定されており、これはbot_message が完了としてマークされるまでループが継続することを意味します。

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



しかし、bot_message はいつ完了としてマークされるのでしょうか？忘れた場合は、finalize メソッドの 13 行目を参照してください。

ストリーム処理のロジック全体を確認しましょう。

1. `PromptSubscriber` は、`Wisper` パブ/サブシステムを通じて `message_created` メソッドで新しいユーザーメッセージを受信します。これはエンドユーザーが新しいプロンプトを作成するたびに呼び出されます。
2. `prompt` クラスメソッドは、`PromptSubscriber` のチャット補完ロジックの動作を宣言的に定義します。AI モデルは、ユーザーのメッセージ内容、ストリームパラメータとしての新しい `ReplyStream` インスタンス、および指定されたループ条件でチャット補完を実行します。
3. AI モデルがプロンプトを処理し、応答の生成を開始します。応答がストリーミングされると、データの各チャンクに対して `ReplyStream` インスタンスの `call` メソッドが呼び出されます。
4. AI モデルがツール関数を呼び出すことを決定した場合、関数名と引数がチャンクから抽出され、それぞれ `f_name` 配列と `f_arguments` 配列に格納されます。
5. AI モデルがユーザー向けのコンテンツを生成する場合、それはバッファリングされ、`send_to_client` メソッドを通じてクライアントに送信されます。
6. ストリーム処理が完了すると、`finalize` メソッドが呼び出されます。ストリーム中にツール関数が呼び出された場合、それらは `PromptSubscriber` の `dispatch` メソッドを使用してディスパッチされます。
7. `dispatch` メソッドは会話トランスクリプトに `function_call` メッセージを追加し、対応するツール関数を実行し、関数呼び出しの結果を含む `function_result` メッセージをトランスクリプトに追加します。
8. ツール関数のディスパッチ後、後続のループで関数呼び出しが重複しないように、関数呼び出し履歴がクリアされます。
9. ツール関数が呼び出されなかった場合、`finalize` メソッドは `bot_message` を最終的なコンテンツで更新し、完了としてマークし、更新されたメッセージをクライアントに送信します。
10. ループ条件 -> `{ bot_message.complete? }` が評価されます。`bot_message` が完了としてマークされていない場合、ループは継続し、

更新された会話トランスクリプトと共に元のプロンプトが再度送信されます。

11. `bot_message` が完了としてマークされるまで(つまり、AI モデルが応答の生成を終了し、これ以上ツール関数を実行する必要がなくなるまで)、ステップ 3-10 が繰り返されます。

この会話ループを実装することで、AI モデルがアプリケーションとの間で双方向のやり取りを行い、必要に応じてツール関数を実行し、会話が自然な結論に達するまでユーザー向けの応答を生成することができます。

ストリーム処理と会話ループを組み合わせることで、AI モデルがユーザー入力を処理し、様々なツールや関数を利用し、進化する会話コンテキストに基づいてリアルタイムの応答を提供する、動的でインタラクティブな AI 駆動のエクスペリエンスが可能になります。

自動継続

AI 出力の制限について認識しておくことが重要です。ほとんどのモデルには、`max_tokens` パラメータによって決定される、単一の応答で生成できる最大トークン数があります。AI モデルが応答の生成中にこの制限に達すると、突然停止し、出力が切り捨てられたことを示します。

AI プラットフォーム API からのストリーミング応答では、チャンク内の `finish_reason` 変数を調べることでこの状況を検出できます。`finish_reason` が `"length"` (またはモデル固有の他のキー値) に設定されている場合、モデルが生成中に最大トークン制限に達し、出力が途中で切れたことを意味します。

このシナリオを適切に処理し、シームレスなユーザーエクスペリエンスを提供する一つの方法は、ストリーム処理ロジックに自動継続メカニズムを実装することです。長さに関連する終了理由のパターンマッチを追加することで、中断された箇所から出力を自動的に継続するようにループを選択できます。

以下はReplyStream クラスのcall メソッドを自動継続に対応させる方法を示した、意図的に簡略化された例です:

```
1  LENGTH_STOPS = %w[length MAX_TOKENS]
2
3  def call(chunk, _bytesize)
4    case chunk.deep_symbolize_keys
5      # ...
6
7    in {
8      choices: [
9        { delta: {content: new_content},
10          finish_reason: finish_reason } ] }
11
12    buffer << new_content.to_s
13
14    if finish_reason.blank?
15      send_to_client if new_content.to_s.match?(/\n\n/)
16    elsif LENGTH_STOPS.include?(finish_reason)
17      continue_cutoff
18    else
19      finalize
20    end
21
22    # ...
23  end
24 end
25
26 private
27
```

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

この修正版では、`finish_reason` が出力の切り捨てを示している場合、ストリームを確定せずに、トランスクリプトにメッセージのペアを追加し、元のユーザー向けレスポンスメッセージの `created_at` 属性を更新してトランスクリプトの「最後」に移動させ、ループを継続させることで、AI が中断した箇所から生成を続けられるようにします。

AI の補完エンドポイントはステートレスであることを覚えておってください。トランスクリプトを通じて伝えた情報しか「知り得ません」。この場合、AI に中断されたことを伝えるのは、(エンドユーザーには見えない) メッセージをトランスクリプトに追加することによって行います。ただし、これは意図的に単純化された例であることに注意してください。実際の実装では、トークンを無駄にしたり、トランスクリプト内の重複したアシスタントメッセージで AI を混乱させたりしないように、さらなるトランスクリプト管理が必要になります。

自動継続の実際の実装では、暴走的なループを防ぐためのもの組み込むべきです。これは、特定の種類のユーザープロンプトと低い `max_tokens` 設定の場合、AI がユーザー向けの出力を無限にループし続ける可能性があるためです。

ループごとに個別のリクエストが必要であり、各リクエストでトランスクリプト全体が再度消費されることを覚えておってください。アプリケーションに自動継続を実装するかどうかを決める際は、ユーザーエクスペリエンスと API 使用量のトレードオフを十分に検討すべきです。特に自動継続は、プレミアムな商用モデルを使用する場合、危険なほど高額になる可能性があります。

結論

提供されている`Conversation::ReplyStream` クラスは、パターンマッチングとイベント駆動アーキテクチャを使用して、アプリケーションでストリーム処理を実装する方法を示しています。ストリーム処理技術を理解し活用することで、アプリケーションにおける AI 統合の可能性を最大限に引き出し、強力で魅力的なユーザーエクスペリエンスを提供することができます。

自己修復データ



自己修復データは、大規模言語モデル(LLM)の機能を活用して、アプリケーションにおけるデータの整合性、一貫性、品質を確保する強力なアプローチです。このパターンカテゴリーは、AIを使用してデータの異常、不整合、またはエラーを自動的に検出、診断、修正するという考えに焦点を当てており、開発者の負担を軽減し、データの信頼性を高水準に維持します。

自己修復データパターンの核心は、データがあらゆるアプリケーションの生命線であり、その正確性と整合性の確保がアプリケーションの適切な機能とユーザー体験にとって極めて重要だという認識にあります。しかし、データ品質の管理と維持は、特にアプリケーションの規模と複雑さが増大するにつれて、複雑で時間のかかるタスクとなり得ます。ここで AI の力が生きてきます。

自己修復データパターンでは、AI ワーカーがアプリケーションのデータを継続的に監視し分析します。これらのモデルは、データ内のパターン、関係性、異常を理解し解

釈する能力を持っています。自然言語処理と理解の機能を活用することで、データの潜在的な問題や不整合を特定し、それらを修正するための適切な措置を講じることができます。

自己修復データのプロセスは、通常、以下の主要なステップを含みます：

1. **データ監視**：AI ワーカーが、アプリケーションのデータストリーム、データベース、またはストレージシステムを常時監視し、異常、不整合、またはエラーの兆候を探します。あるいは、例外が発生した際に AI コンポーネントを起動することもできます。
2. **異常検知**：問題が検出された場合、AI ワーカーはデータを詳細に分析し、問題の具体的な性質と範囲を特定します。これには、欠損値、一貫性のないフォーマット、または事前定義されたルールや制約に違反するデータの検出が含まれる場合があります。
3. **診断と修正**：問題が特定されると、AI ワーカーはデータドメインに関する知識と理解を活用して、適切な対処方法を決定します。これには、データの自動修正、欠損値の補完、または必要に応じて人間の介入のためのフラグ付けが含まれる場合があります。
4. **継続的学習(ユースケースに応じてオプション)**：AI ワーカーが様々なデータの問題に遭遇し解決する中で、発生した事象とその対応について説明を出力できます。このメタデータは学習プロセスに取り込まれ、あなた(場合によってはファインチューニングを通じて基礎となるモデル自体も)がデータの異常の特定と解決においてより効果的かつ効率的になることを可能にします。

データの問題を自動的に検出し修正することで、アプリケーションが高品質で信頼性の高いデータで動作することを確保できます。これにより、エラー、不整合、またはデータ関連のバグがアプリケーションの機能やユーザー体験に影響を与えるリスクが軽減されます。

AI ワーカーにデータの監視と修正のタスクを任せることで、アプリケーションの他の重要な側面に注力することができます。これにより、手動でのデータクリーニングとメ

メンテナンスに費やされていた時間とリソースを節約できます。実際、アプリケーションの規模と複雑さが増大するにつれて、データ品質の手動管理はますます困難になります。「自己修復データ」パターンは、AI の力を活用して大量のデータを処理し、リアルタイムで問題を検出することで、効果的にスケールします。



AI モデルは、その性質上、ほとんど監督なしで時間とともに変化するデータパターン、スキーマ、または要件に適応できます。特に意図する結果に関して、それらの指示が適切なガイダンスを提供する限り、アプリケーションは広範な手動介入やコード変更を必要とせずに、新しいデータシナリオに対応し進化できる可能性があります。

自己修復データパターンは、「多数のワーカー」など、これまで議論してきた他のパターンカテゴリーとも上手く調和します。自己修復データ機能は、データ品質と整合性の確保に特化した特殊なワーカーとして見ることができます。このタイプのワーカーは、アプリケーションの機能の異なる側面に貢献する他の AI ワーカーと並行して動作します。

自己修復データパターンを実践で実装するには、アプリケーションアーキテクチャに AI モデルを慎重に設計し統合する必要があります。データの損失や破損のリスクがあるため、このテクニックをどのように使用するかについて明確なガイドラインを定める必要があります。また、パフォーマンス、スケーラビリティ、データセキュリティなどの要因も考慮する必要があります。

実践的ケーススタディ：破損した **JSON** の修復

自己修復データを活用する最も実用的で便利な方法の 1 つは、説明も非常に簡単です：破損した JSON の修復です。

このテクニックは、LLM が生成する不完全または一貫性のないデータ（破損した JSON など）に対処するという一般的な課題に適用でき、これらの問題を自動的に検出して修正するアプローチを提供します。

Olympiaでは、LLM が生成する JSON データが完全に有効でないシナリオに定期的に遭遇します。これは、LLM が実際の JSON コードの前後にコメントを追加したり、欠落したカンマやエスケープされていない二重引用符などの構文エラーを導入したりするなど、さまざまな理由で発生する可能性があります。これらの問題は構文解析エラーを引き起こし、アプリケーションの機能に支障をきたす可能性があります。

この問題に対処するため、JsonFixer クラスという形で実用的なソリューションを実装しました。このクラスは破損した JSON を入力として受け取り、LLM を活用して可能な限り情報と意図を保持しながら修復を行う「自己修復データ」パターンを具現化しています。

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

```
20
21  def model
22    "mistralai/mixtral-8x7b-instruct:nitro"
23  end
24 end
```



JsonFixer が [Ventriloquist](#) を使用して AI の応答を導く方法に注目してください。

自己修復 JSON データの処理は以下のように機能します:

1. **JSON 生成:** LLM を使用して、特定のプロンプトや要件に基づいて JSON データを生成します。ただし、LLM の特性上、生成される JSON が常に完全に有効であるとは限りません。無効な JSON を与えた場合、JSON パーサーは当然 `ParserError` を発生させます。

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

例外メッセージもまた `JsonFixer` の呼び出しに渡されるため、データの何が間違っているのかを完全に推測する必要はありません。特にパーサーは多くの場合、何が正確に間違っているのかを教えてくれるためです。

2. **LLM ベースの修正:** `JsonFixer` クラスは、破損した JSON を LLM (大規模言語モデル) に送り返し、JSON を修正するための具体的なプロンプトまたは指示と共に、可能な限り元の情報と意図を保持するように処理します。膨大なデータ

で学習され、JSON 構文を理解している LLM は、エラーを修正し、有効な JSON 文字列を生成しようと試みます。[レスポンスフェンシング](#)を使用して LLM の出力を制約し、この種のタスクに特に適している Mixtral 8x7B を AI モデルとして選択しています。

3. **検証と統合:** LLM から返された修正済みの JSON 文字列は、`chat_completion(json: true)` を呼び出したため、JSONFixer クラス自体によって解析されます。修正された JSON が検証をパスすると、アプリケーションのワークフローに再統合され、アプリケーションがシームレスにデータ処理を継続できるようになります。不正な JSON が「修復」されたのです。

私は自身のJSONFixer 実装を何度も書き直してきましたが、それらすべてのバージョンに費やした合計時間は 1、2 時間を超えることはありませんでした。

意図の保持は、あらゆる自己修復データパターンの重要な要素であることに注意してください。LLM ベースの修正プロセスは、生成された JSON の元の情報と意図を可能な限り保持することを目指します。これにより、修正された JSON がその意味的な内容を維持し、アプリケーションのコンテキスト内で効果的に使用できることが保証されます。

Olympia におけるこの Self-Healing Data アプローチの実践的な実装は、AI、特に LLM が現実世界のデータ課題をどのように解決できるかを明確に示しています。これは、堅牢で効率的なアプリケーションを構築するために、従来のプログラミング技術と AI 機能を組み合わせることの力を示しています。

ポステルの法則と「Self-Healing Data」パターン

JSONFixer クラスに例示される「Self-Healing Data」は、ポステルの法則（別名：ロバストネス原則）として知られる原則とよく整合します。ポステルの法則は以下のように述べています：

「自分のすることに関しては保守的に、他人から受け取るものに関しては寛容であれ」

インターネット黎明期のパイオニアである Jon Postel によって提唱されたこの原則は、出力時には指定されたプロトコルを厳密に遵守しながら、多様な、あるいはわずかに不正確な入力に対して寛容なシステムを構築することの重要性を強調しています。

「Self-Healing Data」のコンテキストでは、JSONFixer クラスは、LLM によって生成された壊れた、または不完全な JSON データを受け入れることで、ポストルの法則を体現しています。期待される形式に厳密に従わない JSON に遭遇しても、即座に拒否したり失敗したりすることはありません。代わりに、LLM の力を使用して JSON を修正しようと試みる寛容なアプローチを取ります。

不完全な JSON を寛容に受け入れることで、JSONFixer クラスは堅牢性と柔軟性を実証しています。現実世界のデータは様々な形式で来る可能性があり、常に厳密な仕様に従うとは限らないことを認識しています。これらの逸脱を優雅に処理し修正することで、このクラスは不完全なデータが存在する場合でもアプリケーションがスムーズに機能し続けることを保証します。

一方で、JSONFixer クラスは出力に関してはポストルの法則の保守的な側面も遵守しています。LLM を使用して JSON を修正した後、クラスは修正された JSON を検証して、期待される形式に厳密に従っていることを確認します。アプリケーションの他の部分に渡す前に、データの整合性と正確性を維持します。この保守的なアプローチにより、JSONFixer クラスの出力が信頼性が高く一貫したものとなり、相互運用性を促進しエラーの伝播を防ぎます。

Jon Postel に関する興味深い豆知識：

- Jon Postel(1943-1998)は、インターネットの開発に重要な役割を果たしたアメリカのコンピュータ科学者でした。基礎となるプロトコルと標準への多大な貢献により、「インターネットの神様」として知られていました。
- Postel は、インターネットに関する技術的および組織的な注釈のシリーズである Request for Comments(RFC)文書シリーズの編集者でした。TCP、IP、

SMTP などの基本的なプロトコルを含む 200 以上の RFC の著者または共著者でした。

- 技術的な貢献に加えて、Postel は謙虚で協調的なアプローチで知られていました。彼は合意に達することと、堅牢で相互運用可能なネットワークを構築するために協力することの重要性を信じていました。
- Postel は 1977 年から 1998 年の早すぎる死まで、南カリフォルニア大学 (USC) の情報科学研究所 (ISI) のコンピュータネットワーク部門のディレクターを務めました。
- 彼の多大な貢献を認められ、Postel は 1998 年に「コンピューティングのノーベル賞」とも呼ばれる権威あるチューリング賞を死後受賞しました。

JSONFixer クラスは、Postel が生涯を通じて重視した堅牢性、柔軟性、相互運用性を促進します。不完全さに対して寛容でありながら、プロトコルに厳密に準拠したシステムを構築することで、実世界の課題に対してより回復力があり、適応力のあるアプリケーションを作成することができます。

考慮事項と禁忌

自己修復データアプローチの適用可能性は、アプリケーションが扱うデータの種類に完全に依存します。アプリケーション内の_すべての JSON パースエラー_を自動的に修正するために `JSON.parse` を単純にモンキーパッチすることが望ましくない理由があります: すべてのエラーが自動的に修正できる、あるいは修正すべきというわけではありません。

自己修復は、データの取り扱いと処理に関する規制やコンプライアンス要件と組み合わせると特に難しい問題となります。医療や金融などの一部の業界では、データの整合性と監査可能性に関する規制が非常に厳しく、適切な監視やログ記録なしに「ブラックボックス」的なデータ修正を行うことは、これらの規制に違反する可能性があります。自己修復データ技術が適用される法的および規制の枠組みに確実に準

抛するようにすることが極めて重要です。

特に AI モデルを含む自己修復データ技術の適用は、アプリケーションのパフォーマンスとリソース使用にも大きな影響を与える可能性があります。エラーの検出と修正のために大量のデータを AI モデルで処理することは、計算負荷が高くなる可能性があります。自己修復データのメリットと、それに関連するパフォーマンスやリソースのコストとのトレードオフを評価することが重要です。

とはいえ、このパワフルなアプローチをいつどこで適用するかを決定する際の要因について詳しく見ていきましょう。

データの重要度

自己修復データ技術の適用を検討する際、処理するデータの重要度を評価することが極めて重要です。重要度とは、アプリケーションとそのビジネスドメインの文脈におけるデータの重要性和機密性を指します。

場合によっては、特にデータが非常に機密性が高いか、法的な影響を持つ場合、データエラーを自動的に修正することは適切ではないかもしれません。以下のようなシナリオを考えてみましょう：

1. **金融取引：**銀行システムや取引プラットフォームなどの金融アプリケーションでは、データの正確性が最も重要です。金融データのわずかなエラーでも、口座残高の誤り、資金の誤った送金、誤った取引判断など、重大な結果をもたらす可能性があります。このような場合、徹底的な検証と監査なしに自動修正を行うことは、受け入れがたいリスクをもたらす可能性があります。
2. **医療記録：**医療アプリケーションは、非常に機密性の高い患者データを扱います。医療記録の不正確さは、患者の安全と治療決定に重大な影響を及ぼす可能性があります。資格を持つ医療専門家による適切な監督と検証なしに医療データを自動的に修正することは、規制要件に違反し、患者の健康を危険にさらす可能性があります。

3. **法的文書:** 契約書、合意書、または裁判所提出書類などの法的文書を扱うアプリケーションでは、厳密な正確性と整合性が求められます。法的データのわずかなエラーでも、重大な法的影響をもたらす可能性があります。この分野での自動修正は適切ではない可能性があります。なぜなら、データは多くの場合、その有効性と執行可能性を確保するために法律の専門家による手動のレビューと検証が必要だからです。

これらの重要なデータシナリオでは、自動修正に関連するリスクが潜在的なメリットを上回ることが多いです。エラーを導入したり、データを誤って修正したりすることの結果は深刻で、金銭的損失、法的責任、さらには個人への危害につながる可能性があります。

非常に重要なデータを扱う場合、手動での検証とバリデーションプロセスを優先することが不可欠です。データの正確性と整合性を確保するには、人間による監督と専門知識が極めて重要です。自動自己修復技術は潜在的なエラーや不整合を指摘するために使用することはできますが、修正に関する最終決定には人間の判断と承認が必要です。

ただし、アプリケーション内のすべてのデータが同じレベルの重要度を持つわけではないことに注意することが重要です。同じアプリケーション内でも、エラーが発生しても影響が少ない、より機密性の低いデータのサブセットが存在する可能性があります。そのような場合、重要なデータは手動検証の対象としたまま、特定のデータサブセットに対して選択的に自己修復データ技術を適用することができます。

重要なのは、アプリケーション内の各データカテゴリーの重要度を慎重に評価し、関連するリスクと影響に基づいて修正を処理するための明確なガイドラインとプロセスを定義することです。重要なデータ(台帳や医療記録など)と重要でないデータ(郵送先住所やリソース警告など)を区別することで、適切な場合には自己修復データ技術のメリットを活用し、必要な場合には厳密な管理と監督を維持するバランスを取ることができます。

自己修復データ技術を重要なデータに適用するかどうかの決定は、ドメインエキスパート、法務アドバイザー、その他の関連するステークホルダーと協議の上で行う必

必要があります。アプリケーションのデータに関連する具体的な要件、規制、リスクを考慮し、それに応じてデータ修正戦略を調整することが不可欠です。

エラーの重大度

自己修復データ技術を適用する際は、データエラーの重大度と影響を評価することが重要です。すべてのエラーが同じように扱われるわけではなく、問題の重大度に応じて適切な対応が異なる場合があります。

軽微な不整合や形式の問題は、自動修正に適している場合があります。例えば、壊れた JSON を修正する自己修復データワーカーは、欠落したカンマやエスケープされていない二重引用符を、データの意味や構造を大きく変えることなく処理できます。このような種類のエラーは通常、修正が簡単で、データの全体的な整合性への影響も最小限です。

しかし、データの意味や整合性を根本的に変えてしまうような、より重大なエラーの場合は、異なるアプローチが必要かもしれません。そのような場合、自動修正では不十分で、データの正確性と有効性を確保するために人間による介入が必要になる可能性があります。

ここで、エラーの重大度の判断に AI 自体を活用するという考え方が重要になってきます。AI モデルの機能を活用することで、エラーを修正するだけでなく、それらのエラーの重大度を評価し、対処方法について十分な情報に基づいた判断を下すことができる自己修復データワーカーを設計できます。

例えば、顧客データベースに流れ込むデータの不整合を修正する自己修復データワーカーを考えてみましょう。このワーカーは、データを分析して、欠落した情報や矛盾する情報などの潜在的なエラーを特定するように設計できます。ただし、すべてのエラーを自動的に修正するのではなく、重大なエラーを人間のレビュー用にフラグを立てることができる追加のツールコールを備えることができます。

以下は、これを実装する方法の例です：


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

この例では、CustomerDataHealer ワーカーは顧客データの不整合を特定し、修正するように設計されています。ここでも、構造化された出力を得るためにレスポンスフェンシングとベントリロキストを使用します。重要な点として、ワーカーのシステム指示には、深刻なエラーが発生した場合にflag_for_review 関数を使用するよう指示が含まれています。

ワーカーが顧客データを処理する際、データを分析し、不整合を修正しようと試みます。ワーカーが人間の介入を必要とする深刻なエラーであると判断した場合、flag_for_review ツールを使用してデータにフラグを立て、フラグを立てた理由を提供することができます。

chat_completion メソッドはjson: true を指定して呼び出され、修正された顧客データをJSONとしてパースします。関数呼び出し後のループの仕組みは用意されていないため、flag_for_review が呼び出された場合、結果は空白になります。それ以外の場合、顧客データは確認され、必要に応じて修正されたデータで更新されます。

エラーの重大度評価と人間によるレビューのためのフラグ立ての選択肢を組み込むことで、自己修復データワーカーはより知的で適応力のあるものとなります。軽微なエラーは自動的に処理しながら、深刻なエラーは人間の専門家による手動での介入にエスカレーションすることができます。

エラーの重大度を判断する具体的な基準は、ドメインの知識とビジネス要件に基づいてワーカーの指示に定義することができます。データの整合性への影響、データの損失や破損の可能性、不正確なデータがもたらす結果などの要因を、重大度の評価時に考慮することができます。

AIを活用してエラーの重大度を評価し、人間の介入のための選択肢を提供することで、自己修復データ技術は自動化とデータの正確性の維持のバランスを取ることができます。このアプローチにより、軽微なエラーは効率的に修正され、深刻なエラーは人間のレビュアーから必要な注意と専門知識を受けることが保証されます。

ドメインの複雑性

自己修復データ技術の適用を検討する際、データドメインの複雑性とその構造や関係を規定するルールを評価することが重要です。ドメインの複雑性は、自動データ修正アプローチの効果と実現可能性に大きな影響を与える可能性があります。

自己修復データ技術は、データが明確に定義されたパターンと制約に従っている場合に効果的に機能します。データ構造が比較的単純で、データ要素間の関係がstraightforwardである領域では、自動修正を高い信頼性で適用することができます。例えば、フォーマットの問題を修正したり、基本的なデータ型の制約を強制したりすることは、自己修復データワーカーによって効果的に処理できることが多いです。

しかし、データドメインの複雑性が増すにつれて、自動データ修正に関連する課題も大きくなります。複雑なビジネスロジック、データエンティティ間の複雑な関係、またはドメイン固有のルールと例外がある領域では、自己修復データ技術が常にその微妙な違いを捉えられるとは限らず、意図しない結果をもたらす可能性があります。

複雑なドメインの例として、金融取引システムを考えてみましょう。このドメインでは、データには様々な金融商品、市場データ、取引ルール、規制要件が含まれます。異なるデータ要素間の関係は複雑で、データの有効性と一貫性を規定するルールはドメインに非常に特化したものとなる可能性があります。

このような複雑なドメインでは、取引データの不整合を修正する自己修復データワーカーは、ドメイン固有のルールと制約について深い理解が必要です。市場規制、取引制限、リスク計算、決済手続きなどの要因を考慮する必要があります。このような状況での自動修正は、ドメインの複雑性を完全に捉えられない場合があり、誤ってエラーを引き起こしたり、ドメイン固有のルールに違反したりする可能性があります。

ドメインの複雑性による課題に対処するため、自己修復データ技術は以下のような手法を通じて、ドメイン固有の知識とルールを AI モデルとワーカーに組み込むことで強化できます：

1. **ドメイン固有のトレーニング：**自己修復データに使用される AI モデルは、特定のドメインの複雑さとルールを捉えたドメイン固有のデータセットで指示を受けたり、さらにはファインチューニングされたりすることができます。代表的なデータとシナリオにモデルを触れさせることで、ドメイン固有のパターン、制約、例外を学習することができます。
2. **ルールベースの制約：**自己修復データワーカーは、ドメイン固有の知識を組み込んだ明示的なルールベースの制約によって強化することができます。これらのルールはドメインエキスパートによって定義され、データ修正プロセスに統合されます。AI モデルはこれらのルールを使用して判断を導き、ドメイン固有の要件への準拠を確保することができます。
3. **ドメインエキスパートとの協力：**複雑なドメインでは、自己修復データ技術の設計と開発にドメインエキスパートを関与させることが極めて重要です。ドメインエキスパートは、データの複雑さ、ビジネスルール、そして起こりうるエッジケースについて貴重な洞察を提供できます。彼らの知識は**ヒューマンインザループ**パターンを使用して、AI モデルとワーカーに組み込むことができ、自動データ修正の精度と信頼性を向上させることができます。
4. **段階的かつ反復的なアプローチ：**複雑なドメインを扱う場合、自己修復データに対して段階的かつ反復的なアプローチを採用することが有益です。ドメイン全体の修正を一度に自動化しようとするのではなく、ルールと制約が十分に理解されている特定のサブドメインやデータカテゴリに焦点を当てます。ドメインの理解が深まり、技術の有効性が実証されるにつれて、自己修復技術の範囲を徐々に拡大していきます。

データドメインの複雑さを考慮し、ドメイン固有の知識を自己修復データ技術に組み込むことで、自動化と精度のバランスを取ることができます。自己修復データは万能

なソリューションではなく、各ドメインの特定の要件と課題に合わせてアプローチをカスタマイズする必要があることを認識することが重要です。

複雑なドメインでは、自己修復データ技術と人間の専門知識および監督を組み合わせたハイブリッドアプローチが最も効果的である場合があります。自動修正は定型的でよく定義されたケースを処理し、複雑なシナリオや例外は人間のレビューと介入のためにフラグを立てることができます。この協調的なアプローチにより、複雑なデータドメインに必要な制御と精度を維持しながら、自動化の利点を実現することができます。

説明可能性と透明性

説明可能性とは、AI モデルによる判断の背後にある理由を理解し解釈する能力を指し、透明性とはデータ修正プロセスについて明確な可視性を提供することを意味します。

多くの場合、データの修正は監査可能で正当化可能である必要があります。ビジネスユーザー、監査人、規制当局を含むステークホルダーは、特定のデータ修正が行われた理由と、AI モデルがそれらの判断に至った過程について説明を必要とする場合があります。これは特に金融、医療、法務など、データの正確性と完全性が重要な影響を持つドメインで極めて重要です。

説明可能性と透明性の必要性に対応するため、自己修復データ技術には AI モデルの意思決定プロセスについての洞察を提供するメカニズムを組み込む必要があります。これは以下のようなアプローチで実現できます：

1. **思考の連鎖：**データに変更を適用する前にモデルに「声に出して」考えを説明させることで、意思決定プロセスの理解が容易になり、修正に対する人間が読める説明を生成することができます。トレードオフとして、説明と構造化データの出力を分離する際に若干の複雑さが増しますが、これは以下のように対処できます...

2. **説明の生成:** 自己修復データワーカーに、行った修正について人間が読める説明を生成する機能を備えることができます。これは、モデルに_データ自体に統合された_形で意思決定プロセスを分かりやすい説明として出力させることで実現できます。例えば、自己修復データワーカーは、特定したデータの不整合、適用した修正、およびそれらの修正の根拠を強調したレポートを生成することができます。
3. **特徴量の重要度:** AI モデルに、指示の一部としてデータ修正プロセスにおける異なる特徴や属性の重要性に関する情報を指示することができます。これらの指示は、人間のステークホルダーに公開することができます。モデルの判断に影響を与える主要な要因を特定することで、ステークホルダーは修正の背後にある理由を理解し、その妥当性を評価することができます。
4. **ログ記録と監査:** 包括的なログ記録と監査メカニズムの実装は、自己修復データプロセスの透明性を維持する上で極めて重要です。AI モデルによって行われるすべてのデータ修正は、元のデータ、修正後のデータ、および実行された具体的なアクションを含めてログに記録される必要があります。この監査証跡により、遡及的な分析が可能になり、データに加えられた修正の明確な記録が提供されます。
5. **人間参加型アプローチ:** 自己修復データ技術の説明可能性と透明性を高めるために、人間参加型アプローチを取り入れることが効果的です。AI 生成の修正内容をドメイン知識とビジネス要件に合致させるため、人間の専門家によるレビューと検証を組み込むことが重要です。人間による監督は、さらなる説明責任の層を追加し、AI モデルに存在する可能性のあるバイアスやエラーを特定することを可能にします。
6. **継続的なモニタリングと評価:** 透明性と信頼性を維持するために、自己修復データ技術のパフォーマンスを定期的にモニタリングし評価することが不可欠です。時間の経過とともに AI モデルの精度と有効性を評価することで、組織は逸脱や異常を特定し、是正措置を講じることができます。継続的なモニタリングにより、自己修復データプロセスが信頼性を維持し、期待される成果に沿っていることを確認できます。

自己修復データ技術を実装する際、説明可能性と透明性は重要な考慮事項です。データ修正の明確な説明を提供し、包括的な監査証跡を維持し、人間による監督を組み込むことで、組織は自己修復データプロセスへの信頼を構築し、データに加えられた修正が正当化され、ビジネス目標に沿っていることを確実にすることができます。

自動化の利点と透明性の必要性のバランスを取ることが重要です。自己修復データ技術はデータ品質と効率性を大幅に改善できますが、それはデータ修正プロセスの可視性とコントロールを失う代償であってはなりません。説明可能性と透明性を念頭に置いて自己修復データワーカーを設計することで、組織は AI の力を活用しながら、データに対する必要な説明責任と信頼性を維持することができます。

意図しない結果

自己修復データ技術はデータ品質と一貫性の向上を目指していますが、意図しない結果が生じる可能性について認識しておくことが重要です。慎重に設計およびモニタリングされていない自動修正は、意図せずにデータの意味や文脈を変更してしまう、下流での問題につながる可能性があります。

自己修復データの主要なリスクの 1 つは、データ修正プロセスにおけるバイアスやエラーの導入です。他のソフトウェアシステムと同様に、AI モデルは訓練データに存在するバイアスやアルゴリズムの設計を通じて導入されるバイアスの影響を受ける可能性があります。これらのバイアスが特定され軽減されない場合、自己修復データプロセスを通じて伝播し、偏ったまたは不正確なデータ修正をもたらす可能性があります。

例えば、顧客の人口統計データの不整合を修正するタスクを担当する自己修復データワーカーを考えてみましょう。AI モデルが過去のデータからバイアスを学習している場合、特定の職業や収入レベルを特定の性別や民族と関連付けるなど、誤った仮定を行い、それらのバイアスを強化するようなデータ修正を行う可能性があります。これは不正確な顧客プロフィール、誤った事業判断、さらには差別的な結果につながる可能性があります。

もう1つの潜在的な意図しない結果は、データ修正プロセス中の価値ある情報やコンテキストの喪失です。自己修復データ技術は、一貫性を確保するためにデータの標準化と正規化に重点を置くことが多くあります。しかし、場合によっては、元のデータに全体像を理解するために重要なニュアンス、例外、または文脈的な情報が含まれている可能性があります。盲目的に標準化を強制する自動修正は、このような価値ある情報を意図せずに削除したり不明確にしたりする可能性があります。

例えば、医療記録の不整合を修正する責任を持つ自己修復データワーカーを想像してみてください。ワーカーが稀少な症状や異常な治療計画を持つ患者の医療履歴に遭遇した場合、より一般的なパターンに適合するようにデータを正規化しようとするかもしれません。しかし、そうすることで、患者の固有の状況を正確に表現するために重要な具体的な詳細や文脈が失われる可能性があります。この情報の喪失は、患者のケアや医療上の意思決定に重大な影響を及ぼす可能性があります。

意図しない結果のリスクを軽減するために、自己修復データ技術の設計と実装時には積極的なアプローチを取ることが不可欠です：

1. **徹底的なテストと検証：**自己修復データワーカーを本番環境にデプロイする前に、様々なシナリオに対する動作を徹底的にテストし検証することが重要です。これには、様々なエッジケース、例外、および潜在的なバイアスをカバーする代表的なデータセットでのテストが含まれます。厳密なテストにより、実際のデータに影響を与える前に、意図しない結果を特定し対処することができます。
2. **継続的なモニタリングと評価：**継続的なモニタリングと評価の仕組みを実装することは、時間の経過とともに生じる意図しない結果を検出し、軽減するために不可欠です。自己修復データプロセスの結果を定期的に確認し、下流のシステムや意思決定への影響を分析し、ステークホルダーからのフィードバックを収集することで、悪影響を特定し、適時には是正措置を講じることができます。組織が運用ダッシュボードを持っている場合、自動化されたデータ変更に関連する指標を分かりやすく表示することは、おそらく良いアイデアでしょう。通常のデータ変更活動から大きく逸脱した場合にアラームを設定することは、さらに良いアイデアかもしれません！

3. **人間による監督と介入:** 自己修復データプロセスにおける人間による監督と介入能力を維持することは極めて重要です。自動化は効率性を大幅に向上させる可能性があります、特に重要または機密性の高い領域では、AI モデルによる修正を人間の専門家が確認し、検証することが重要です。人間の判断と専門知識は、発生する可能性のある意図しない結果を特定し、対処するのに役立ちます。
4. **説明可能な AI(XAI)と透明性:** 前節で説明したように、説明可能な AI の技術を取り入れ、自己修復データプロセスの透明性を確保することは、意図しない結果を軽減するのに役立ちます。データ修正に関する明確な説明を提供し、包括的な監査証跡を維持することで、組織は AI モデルによる修正の背後にある理由をより良く理解し、追跡することができます。
5. **段階的かつ反復的なアプローチ:** 自己修復データに段階的かつ反復的なアプローチを採用することで、意図しない結果のリスクを最小限に抑えることができます。データセット全体に一度に自動修正を適用するのではなく、データの一部から始めて、その技術が効果的で信頼できることが証明されてから徐々に範囲を拡大していきます。これにより、途中で注意深くモニタリングと調整を行うことができ、意図しない結果の影響を軽減できます。
6. **協力とフィードバック:** 異なる領域のステークホルダーを関与させ、自己修復データプロセス全体を通じて協力とフィードバックを促すことは、意図しない結果を特定し、対処するのに役立ちます。専門家、データ利用者、エンドユーザーから定期的に意見を求めることで、データ修正の実際の影響に関する貴重な洞察を得ることができ、見落とされていた問題を浮き彫りにすることができます。

意図しない結果のリスクに積極的に対処し、適切な保護措置を実施することで、組織は潜在的な悪影響を最小限に抑えながら、自己修復データ技術の利点を活用することができます。自己修復データを反復的で協力的なプロセスとしてアプローチし、望ましい結果と一致させ、データの整合性と信頼性を維持するために、技術を継続的にモニタリング、評価、改善することが重要です。

自己修復データパターンの使用を検討する際には、これらの要因を慎重に評価し、利点と潜在的なリスクおよび制限を比較検討することが不可欠です。場合によっては、自動修正と人間による監督および介入を組み合わせたハイブリッドアプローチが最も適切な解決策となるかもしれません。

また、自己修復データ技術は、堅牢なデータ検証、入力データのサニタイズ、エラー処理メカニズムの代替として見なすべきではないことにも注意が必要です。これらの基礎的な実践は、データの整合性とセキュリティを確保するために依然として重要です。自己修復データは、これらの既存の対策を補完し、強化する補完的なアプローチとして捉えるべきです。

最終的に、自己修復データパターンを採用するかどうかの決定は、アプリケーションの具体的な要件、制約、優先順位に依存します。上記の考慮事項を慎重に検討し、アプリケーションの目標とアーキテクチャに合わせることで、自己修復データ技術をいつ、どのように効果的に活用するかについて、十分な情報に基づいた決定を下すことができます。

コンテキスト型コンテンツ生成



コンテキスト型コンテンツ生成パターンは、大規模言語モデル(LLM)の能力を活用して、アプリケーション内で動的かつコンテキストに応じたコンテンツを生成します。このパターンカテゴリーは、ユーザーの特定のニーズ、好み、さらにはアプリケーションとの過去および現在のインタラクションに基づいて、パーソナライズされた関連性の高いコンテンツを提供することの重要性を認識しています。

このアプローチにおける「コンテンツ」とは、主要コンテンツ(ブログ投稿、記事など)と、主要コンテンツへの推奨などのメタコンテンツの両方を指します。

コンテキスト型コンテンツ生成パターンは、ユーザーエンゲージメントレベルの向上、カスタマイズされた体験の提供、そしてあなたとユーザーの両方のためのコンテンツ作成タスクの自動化において重要な役割を果たすことができます。この章で説明す

るパターンを活用することで、コンテキストと入力にリアルタイムで適応しながら、動的にコンテンツを生成するアプリケーションを作成することができます。

このパターンは、ユーザーインターフェース(「chrome」とも呼ばれる)、メール、その他の通知形式、さらにはあらゆるコンテンツ生成パイプラインにおいて、LLMをアプリケーションの出力に統合することで機能します。

ユーザーがアプリケーションと対話するか、特定のコンテンツリクエストをトリガーすると、アプリケーションはユーザーの好み、過去のインタラクション、特定のプロンプトなどの関連するコンテキストを取得します。このコンテキスト情報は、必要なテンプレートやガイドラインとともに LLM に送られ、ハードコードされたり、データベースに保存されたり、アルゴリズムで生成されたりする必要があった場合のテキスト出力を生成するために使用されます。

LLM が生成するコンテンツは、パーソナライズされた推奨、動的な製品説明、カスタマイズされたメール返信、さらには記事やブログ投稿全体など、さまざまな形を取ることができます。1 年以上前に私が先駆けとなった最も画期的な使用法の 1 つは、フォームラベル、ツールチップ、その他の説明テキストなどの UI 要素を動的に生成することです。

パーソナライゼーション

コンテキスト型コンテンツ生成パターンの主要な利点の 1 つは、ユーザーに高度にパーソナライズされた体験を提供できることです。ユーザー固有のコンテキストに基づいてコンテンツを生成することで、これらのパターンを使用するアプリケーションは、個々のユーザーの興味、好み、インタラクションに合わせてコンテンツをカスタマイズすることができます。

パーソナライゼーションは、単に一般的なコンテンツにユーザーの名前を挿入する以上のものです。各ユーザーについて利用可能な豊富なコンテキストを活用して、特定のニーズと欲求に共鳴するコンテンツを生成することを含みます。このコンテキストには、以下のようなさまざまな要因が含まれます：

1. **ユーザープロフィール情報:** このテクニックを適用する最も一般的なレベルでは、人口統計データ、興味、好み、その他のプロフィール属性を使用して、ユーザーの背景と特性に合ったコンテンツを生成することができます。
2. **行動データ:** 閲覧したページ、クリックしたリンク、購入した製品など、アプリケーションとのユーザーの過去のインタラクションは、その行動と興味に関する貴重な洞察を提供できます。このデータは、ユーザーのエンゲージメントパターンを反映し、将来のニーズを予測するコンテンツ提案を生成するために使用できます。
3. **コンテキスト要因:** ユーザーの現在の場所、デバイス、時刻、さらには天候などの現在のコンテキストが、コンテンツ生成プロセスに影響を与える可能性があります。例えば、旅行アプリケーションでは、ユーザーの現在地と現在の天候条件に基づいてパーソナライズされた推奨を生成できる AI ワーカーを持つことができます。

これらのコンテキスト要因を活用することで、コンテキスト型コンテンツ生成パターンは、各ユーザーのために特別に作られたように感じられるコンテンツを提供することができます。このレベルのパーソナライゼーションには、いくつかの重要な利点があります：

1. **エンゲージメントの向上:** パーソナライズされたコンテンツはユーザーの注意を引き、アプリケーションへの関与を維持します。ユーザーがコンテンツが関連性が高く、自分のニーズに直接応えていると感じる場合、アプリケーションとの対話により多くの時間を費やし、その機能を探索する可能性が高くなります。
2. **ユーザー満足度の向上:** パーソナライズドコンテンツは、アプリケーションがユーザーの固有のニーズを理解し、配慮していることを示します。ユーザーの興味に沿った有用で情報価値の高いコンテンツを提供することで、アプリケーションはユーザー満足度を高め、ユーザーとのより強い結びつきを構築することができます。
3. **より高いコンバージョン率:** e コマースやマーケティングアプリケーションにおいて、パーソナライズドコンテンツはコンバージョン率に大きな影響を与えること

ができます。ユーザーの好みや行動に合わせた商品、オファー、レコメンデーションを提示することで、購入やサービス登録といった望ましいアクションをユーザーが取る可能性を高めることができます。

生産性

コンテキスト型コンテンツ生成パターンは、クリエイティブプロセスにおける手動でのコンテンツ生成や編集の必要性を減らすことで、特定の種類の生産性を大幅に向上させることができます。大規模言語モデルの力を活用することで、コンテンツクリエイターや開発者が退屈な手作業に費やさなければならない時間と労力を節約しながら、大規模な高品質コンテンツを生成することが可能になります。

従来、コンテンツクリエイターは、アプリケーションの要件とユーザーの期待に応えるために、リサーチ、執筆、編集、フォーマット設定を行う必要がありました。特にコンテンツの量が増えるにつれて、このプロセスは時間と資源を大量に消費する可能性があります。

しかし、コンテキスト型コンテンツ生成パターンを使用することで、コンテンツ作成プロセスの大部分を自動化することができます。大規模言語モデルは、与えられたプロンプトとガイドラインに基づいて、一貫性があり、文法的に正しく、文脈に即したコンテンツを生成することができます。この自動化には以下のような生産性の利点があります：

1. **手作業の削減:** コンテンツ生成タスクを大規模言語モデルに委託することで、コンテンツクリエイターはコンテンツ戦略、アイデア創出、品質保証といったより高次のタスクに集中できます。必要なコンテキスト、テンプレート、ガイドラインを大規模言語モデルに提供し、実際のコンテンツ生成を任せることができます。これにより、執筆や編集に必要な手作業が減り、コンテンツクリエイターはより生産的かつ効率的に作業を行うことができます。

2. **より迅速なコンテンツ作成:** 大規模言語モデルは人間の作家よりもはるかに速くコンテンツを生成できます。適切なプロンプトとガイドラインがあれば、大規模言語モデルは数秒から数分で複数のコンテンツを生成することができます。この速度により、アプリケーションはユーザーの要求と絶えず変化するデジタル環境に対応しながら、より速いペースでコンテンツを生成することができます。

より速いコンテンツ作成は、誰も読まないコンテンツでインターネットが溢れかえる「コモズの悲劇」の状況を引き起こしているのでしょうか。残念ながら、その答えはイエスだと思います。

3. **一貫性と品質:** 大規模言語モデルは、スタイル、トーン、品質の面で一貫したコンテンツを容易に作成することができます。明確なガイドラインと例を提供することで、特定の種類のアプリケーション(ニュースルーム、PR 等)は、人間が生成したコンテンツがブランドボイスと一致し、望ましい品質基準を満たすことを確保できます。この一貫性により、広範な編集や改訂の必要性が減少し、コンテンツ作成プロセスにおける時間と労力を節約することができます。
4. **反復と最適化:** コンテキスト型コンテンツ生成パターンにより、コンテンツの迅速な反復と最適化が可能になります。大規模言語モデルに提供するプロンプト、テンプレート、またはガイドラインを調整することで、アプリケーションは過去には不可能だった方法で、自動的にコンテンツのバリエーションを生成し、異なるアプローチをテストすることができます。この反復プロセスにより、コンテンツ戦略のより速い実験と改善が可能になり、時間とともにより効果的で魅力的なコンテンツを生み出すことができます。この特定の技術は、直帰率とエンゲージメントに基づいて生死が決まる e コマースなどのアプリケーションにとって、完全なゲームチェンジャーとなり得ます。



コンテキスト型コンテンツ生成パターンは生産性を大きく向上させることができますが、人間の関与を完全に排除するものではないことに注意することが重要です。コンテンツクリエイターや編集者は、全体的なコンテンツ戦略の定義、大規模言語モデルへのガイダンスの提供、生成されたコンテンツの品質と適切性の確保において、依然として重要な役割を果たしています。

コンテキスト型コンテンツ生成パターンにより、コンテンツ作成における反復的で時間のかかる作業を自動化することで、より価値の高いタスクに人的時間とリソースを振り向けることが可能になります。この生産性の向上により、コンテンツ作成のワークフローを最適化しながら、よりパーソナライズされた魅力的なコンテンツをユーザーに提供できるようになります。

高速イテレーションと実験

コンテキスト型コンテンツ生成パターンを使用することで、さまざまなコンテンツのバリエーションを素早く反復し、実験することができ、コンテンツ戦略の最適化と改善をより迅速に行うことが可能になります。コンテキスト、テンプレート、またはモデルに提供するガイドラインを調整するだけで、数秒でコンテンツの複数のバージョンを生成できます。

この高速イテレーション機能には、以下のような主要なメリットがあります：

1. **テストと最適化:** コンテンツのバリエーションを素早く生成できる機能により、さまざまなアプローチを簡単にテストし、その効果を測定することができます。例えば、特定のユーザーセグメントやコンテキストに合わせた製品説明やマーケティングメッセージの複数のバージョンを生成できます。クリック率やコンバージョン率などのユーザーエンゲージメント指標を分析することで、最も効果的なコンテンツのバリエーションを特定し、コンテンツ戦略を適切に最適化することができます。

2. **A/B テスト:** コンテキスト型コンテンツ生成パターンにより、コンテンツの A/B テストをシームレスに実施することができます。2 つ以上のコンテンツのバリエーションを生成し、異なるユーザーグループにランダムに提供することができます。各バリエーションのパフォーマンスを比較することで、ターゲットオーディエンスに最も響くコンテンツを判断することができます。このデータ駆動型のアプローチにより、情報に基づいた意思決定を行い、ユーザーエンゲージメントを最大化し、望ましい結果を達成するためにコンテンツを継続的に改善することができます。
3. **パーソナライゼーションの実験:** 高速イテレーションと実験は、パーソナライゼーションにおいて特に価値があります。コンテキスト型コンテンツ生成パターンを使用することで、異なるユーザーセグメント、好み、または行動に基づいて、パーソナライズされたコンテンツのバリエーションを素早く生成できます。さまざまなパーソナライゼーション戦略を実験することで、個々のユーザーを魅了し、カスタマイズされた体験を提供するための最も効果的なアプローチを特定できます。
4. **変化するトレンドへの適応:** 素早く反復し実験できる能力により、変化するトレンドやユーザーの好みに機敏に適応することができます。新しいトピック、キーワード、またはユーザーの行動が出現した際に、これらのトレンドに合わせたコンテンツを素早く生成できます。コンテンツを継続的に実験し改善することで、常に進化するデジタルランドスケープにおいて関連性を保ち、競争力を維持することができます。
5. **コスト効率の良い実験:** 従来のコンテンツ実験では、コンテンツクリエイターが異なるバリエーションを手動で開発しテストする必要があるため、多大な時間とリソースが必要でした。しかし、コンテキスト型コンテンツ生成パターンを使用することで、実験のコストが大幅に削減されます。LLM は素早くスケーラブルにコンテンツのバリエーションを生成できるため、多大なコストをかけることなく、幅広いアイデアやアプローチを探求することができます。

高速イテレーションと実験を最大限に活用するには、適切に定義された実験フレームワークを整備することが重要です。このフレームワークには以下が含まれるべきです:

- 各実験の明確な目的と仮説
- コンテンツのパフォーマンスを測定するための適切な指標と追跡メカニズム
- 適切なコンテンツのバリエーションを適切なユーザーに提供するためのセグメンテーションとターゲティング
- 実験データから洞察を導き出すための分析とレポートツール
- 学びと最適化をコンテンツ戦略に組み込むためのプロセス

高速イテレーションと実験を取り入れることで、コンテンツを継続的に改善し最適化し、アプリケーションの目標達成に向けて魅力的で関連性が高く効果的なものであり続けることができます。このアジャイルなコンテンツ作成アプローチにより、常に先を行き、優れたユーザーエクスペリエンスを提供することができます。

スケーラビリティと効率性

アプリケーションが成長し、パーソナライズされたコンテンツの需要が増加するにつれ、コンテキスト型コンテンツ生成パターンによってコンテンツ作成を効率的にスケールすることが可能になります。LLM は人的リソースを比例的に増やす必要なく、多数のユーザーとコンテキストに対して同時にコンテンツを生成することができます。このスケーラビリティにより、アプリケーションはコンテンツ作成能力に負担をかけることなく、成長するユーザーベースにパーソナライズされた体験を提供することができます。



文脈に基づくコンテンツ生成は、アプリケーションを「オンザフライ」で国際化するのに効果的に使用できることに注目してください。実際、それは私が Instant18n Gem を使用して、設立からまだ 1 年も経っていない Olympia を半ダース以上の言語で提供するために行ったこと

AI 駆動のローカライゼーション

少し自慢させていただくと、Rails アプリケーション向けの私の Instant18n ライブラリは、「文脈に基づくコンテンツ生成」パターンの画期的な例であり、アプリケーション開発に

における AI の変革的な可能性を示していると考えています。この gem は、OpenAI の GPT 大規模言語モデルの力を活用して、Rails アプリケーションにおける国際化とローカライゼーションの処理方法を革新します。

従来、Rails アプリケーションの国際化では、翻訳キーを手動で定義し、サポートする各言語に対応する翻訳を提供する必要がありました。このプロセスは時間がかかり、リソースを大量に消費し、一貫性の欠如を招きやすいものでした。しかし、Instant18n gem を使用することで、ローカライゼーションのパラダイムが完全に再定義されます。

大規模言語モデルを統合することで、Instant18n gem はテキストの文脈と意味に基づいてオンザフライで翻訳を生成することができます。事前定義された翻訳キーと静的な翻訳に依存する代わりに、この gem は AI の力を使用して動的にテキストを翻訳します。このアプローチには以下のような主要な利点があります：

1. **シームレスなローカライゼーション：** Instant18n gem を使用することで、開発者はもはやサポートする各言語の翻訳ファイルを手動で定義・管理する必要がありません。この gem は提供されたテキストと目的の対象言語に基づいて自動的に翻訳を生成し、ローカライゼーションのプロセスを労力のいらない、シームレスなものにします。
2. **文脈的な正確さ：** AI には翻訳されるテキストのニュアンスを理解するのに十分な文脈を与えることができます。周囲の文脈、慣用句、文化的な参照を考慮に入れ、正確で自然な響きを持ち、文脈的に適切な翻訳を生成することができます。
3. **広範な言語サポート：** Instant18n gem は GPT の膨大な知識と言語能力を活用し、幅広い言語への翻訳を可能にします。スペイン語やフランス語のような一般的な言語から、クリンゴン語やエルフ語のようなより珍しい、あるいは架空の言語まで、この gem は多様な翻訳要件に対応できます。
4. **柔軟性と創造性：** この gem は従来の言語翻訳を超えて、創造的で独創的なローカライゼーションのオプションを提供します。開発者はテキストを様々なスタイル、方言、さらには架空の言語に翻訳することができ、ユニークなユーザー体験と魅力的なコンテンツのための新しい可能性を開きます。

5. **パフォーマンスの最適化:** Instant18n gem はパフォーマンスを向上させ、繰り返される翻訳のオーバーヘッドを削減するためのキャッシュメカニズムを組み込んでいます。翻訳されたテキストはキャッシュされ、同じ翻訳に対する後続のリクエストは冗長な API コールを必要とせずに迅速に提供されます。

Instant18n gem は、AI を活用して動的にローカライズされたコンテンツを生成することで、「文脈に基づくコンテンツ生成」パターンの力を実証しています。これは、AI を Rails アプリケーションのコア機能に統合し、開発者が国際化とローカライゼーションにアプローチする方法を変革する方法を示しています。

手動での翻訳管理の必要性を排除し、文脈に基づいたオンザフライの翻訳を可能にすることで、Instant18n gem は開発者の大幅な時間と労力を節約します。これにより、ローカライゼーションの側面がシームレスかつ正確に処理されることを確保しながら、アプリケーションのコア機能の構築に集中することができます。

ユーザーテストとフィードバックの重要性

最後に、ユーザーテストとフィードバックの重要性を常に念頭に置いてください。文脈に基づくコンテンツ生成がユーザーの期待に応え、アプリケーションの目標に沿っていることを検証することが重要です。ユーザーの洞察と分析に基づいて、生成されたコンテンツを継続的に改善し、洗練させてください。あなたとチームが手動で検証することが不可能な大規模な動的コンテンツを生成している場合は、ユーザーが奇妙または誤っているコンテンツを報告できるフィードバックメカニズムを追加することを検討してください。その理由の説明も含めて。この貴重なフィードバックは、コンテンツを生成したコンポーネントの調整を行う AI ワーカーに送ることもできます！

生成的 UI



現代では注目度が非常に重要となっており、効果的なユーザーエンゲージメントを実現するには、シームレスで直感的なソフトウェア体験だけでなく、個々のニーズ、好み、状況に合わせて高度にパーソナライズされた体験が必要となっています。その結果、デザイナーや開発者は、各ユーザーの固有の要件に「スケーラブルに」適応できるユーザーインターフェースを作成するという課題に、increasingly 直面しています。

生成的 UI (GenUI) は、ユーザーインターフェース設計に対する真に革新的なアプローチです。大規模言語モデル (LLM) の力を活用して、高度にパーソナライズされた動的なユーザー体験をその場で作り出します。この本で GenUI の基礎について少なくとも紹介しておきたかったのは、アプリケーション設計とフレームワークの分野において、現在存在する最も有望な未開拓の機会の 1 つだと信じているからです。この特定の分野では、数十以上の新しい商用およびオープンソースプロジェクトが生まれると確信しています。

GenUI の核心は、**コンテキスト依存型コンテンツ生成**の原則と先進的な AI 技術を組み合わせ、ユーザーのコンテキスト、好み、目標を深く理解した上で、テキスト、画像、レイアウトなどのユーザーインターフェース要素を動的に生成することにあります。GenUI により、デザイナーや開発者は、ユーザーの操作に応じて適応し進化するインターフェースを作成でき、これまでは実現不可能だったレベルのパーソナライゼーションを提供できます。

GenUI は、ユーザーインターフェース設計へのアプローチ方法を根本的に変えるものです。大衆向けの設計から、個人向けの設計が可能になります。パーソナライズされたコンテンツとインターフェースには、各ユーザーとより深いレベルで共鳴するユーザー体験を作り出し、エンゲージメント、満足度、ロイヤリティを高める可能性があります。

最先端の技術として、GenUI への移行には概念的および実践的な課題が山積しています。設計プロセスへの AI の統合、生成されたインターフェースがパーソナライズされているだけでなく、使いやすく、アクセシブルで、全体的なブランドとユーザー体験に沿ったものであることの確保など、これらの課題によって GenUI は一部の人のための追求となっています。さらに、AI の関与によってデータプライバシー、透明性、そして倫理的な影響に関する問題も提起されています。

課題はありますが、スケーラブルなパーソナライズ体験には、デジタル製品やサービスとの相互作用の方法を完全に変革する力があります。能力、背景、好みに関係なく、ユーザーの多様なニーズに対応する包括的でアクセシブルなインターフェースを作成する可能性を開きます。

この章では、GenUI の概念を探り、いくつかの定義的特徴、主要な利点、潜在的な課題について検討します。まずは、GenUI の最も基本的でアクセスしやすい形態である、従来の方法で設計・実装されたユーザーインターフェースのためのテキストコピーの生成から始めます。

ユーザーインターフェース用のコピーの生成

アプリケーションのクロームに存在するフォームラベル、ツールチップ、説明テキストなどのテキスト要素は、通常、テンプレートや UI コンポーネントにハードコードされており、すべてのユーザーに一貫性のある汎用的な体験を提供します。コンテキスト依存型コンテンツ生成パターンを使用することで、これらの静的な要素を動的で、コンテキストを認識し、パーソナライズされたコンポーネントに変換できます。

パーソナライズされたフォーム

フォームは、ウェブやモバイルアプリケーションにおいて普遍的な存在であり、ユーザー入力を収集する主要な手段となっています。しかし、従来のフォームは一般的で没个性的な体験を提供しがちで、標準的なラベルやフィールドがユーザーの具体的な状況やニーズに必ずしも合致しないことがあります。ユーザーは自分のニーズや好みに合わせて調整されているように感じるフォームの方が入力を完了する可能性が高く、それによってコンバージョン率とユーザー満足度の向上につながります。

ただし、パーソナライゼーションと一貫性のバランスを取ることが重要です。フォームを個々のユーザーに適応させることは有益である一方で、一定の親しみやすさと予測可能性を維持することが不可欠です。パーソナライズされた要素があっても、ユーザーは依然としてフォームを簡単に認識し、操作できる必要があります。

以下に、パーソナライズされたフォームのアイデアをいくつか紹介します：

文脈に応じたフィールド候補

GenUIは、ユーザーの過去のインタラクション、設定、データを分析して、予測として知的なフィールド候補を提供することができます。例えば、ユーザーが以前に配送先住所を入力している場合、フォームは関連するフィールドに保存された情報を自動的に入力することができます。これは時間を節約できるだけでなく、アプリケーションがユーザーの設定を理解し、記憶していることを示すことができます。

ちょっと待ってください。この手法は AI を使わなくても実現できるのではないのでしょうか？もちろんその通りですが、AI でこの種の機能を実現する利点は 2 つあります: 1) 実装が容易であること、2) UI が変更され進化していく中でも柔軟に対応できることです。

では、ユーザーの配送先住所を事前に入力しようとする理論的な注文処理システム用のサービスを作ってみましょう。

```
1  class OrderShippingAddressSubscriber
2      include Raix::ChatCompletion
3
4      attr_accessor :order
5
6      delegate :customer, to: :order
7
8      DIRECTIVE = "You are a smart order processing assistant. Given the
9      customer's order history, guess the most likely shipping address
10     for the current order."
11
12     def order_created(order)
13         return unless order.pending? && order.shipping_address.blank?
14
15         self.order = order
16
17         transcript.clear
18         transcript << { system: DIRECTIVE }
19         transcript << { user: "Order History: #{order_history.to_json}" }
20         transcript << { user: "Current Order: #{order.to_json}" }
21
22         response = chat_completion
```



```
23     apply_predicted_shipping_address(order, response)
24   end
25
26   private
27
28   def apply_predicted_shipping_address(order, response)
29     # extract the shipping address from the response...
30     # ...and assume there's some sort of live update of the address fields
31     order.update(shipping_address:)
32   end
33
34   def order_history
35     customer.orders.successful.limit(100).map do |order|
36       {
37         date: order.date,
38         description: order.description,
39         shipping_address: order.shipping_address
40       }
41     end
42   end
43 end
```

この例は非常に単純化されていますが、ほとんどの場合に有効です。基本的な考え方は、人工知能に人間と同じような方法で推測させることです。具体的に説明するために、いくつかのサンプルデータを見てみましょう：

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

```

30     "shipping_address": "456 City Apt, Metroville"},
31     {"date": "2024-06-11", "description": "baby clothes",
32       "shipping_address": "789 Suburb St, Quietville"},
33     {"date": "2024-07-01", "description": "watering can",
34       "shipping_address": "123 Country Lane, Rural Town"},
35     {"date": "2024-07-18", "description": "non-fiction essays",
36       "shipping_address": "456 City Apt, Metroville"},
37     {"date": "2024-07-28", "description": "baby bath items",
38       "shipping_address": "789 Suburb St, Quietville"},
39     {"date": "2024-08-09", "description": "herb garden kit",
40       "shipping_address": "123 Country Lane, Rural Town"},
41     {"date": "2024-08-24", "description": "children's books",
42       "shipping_address": "456 City Apt, Metroville"}
43 ]

```

データのパターンに気付きましたか?これは間違いなく LLM にとって簡単な作業です。実証するために、GPT-4に「thermometer」の最も可能性の高い配送先住所を尋ねてみましょう。

```

1  From the order history you've provided, it looks like the purchases are
2  generally clustered into three main types based on the shipping addresses:
3
4  123 Country Lane, Rural Town - This address often orders garden and
5  outdoor-related items like soil mix, vegetables, flower seeds, lawn
6  fertilizer, an outdoor grill, a watering can, and an herb garden kit.
7
8  456 City Apt, Metroville - This address tends to order various types of
9  books, indicating an interest in reading across genres like fiction,
10 thrillers, biographies, science fiction, literary classics, non-fiction
11 essays, and children's books.

```

12

13 789 Suburb St, Quietville - The purchases for this address are consistently
14 baby-related products, such as diapers, wipes, baby formula, infant toys,
15 baby clothes, and baby bath items.

16

17 Given these patterns, a digital thermometer could potentially be a fit for
18 any of the addresses based on different needs: outdoor temperature tracking,
19 a tool for a passionate reader interested in science gadgets, or a health
20 tool for a baby. However, considering its common use in monitoring health,
21 especially for babies, it's most likely to be purchased by someone who's
22 already buying baby care products.

23

24 Therefore, the most likely shipping address for a digital thermometer, based
25 on the purchasing patterns, would be 789 Suburb St, Quietville.

もし GPT-4のような高価なモデルをこのタスクに使うのは過剰だと思っているなら、その通りです!同じプロンプトを Mistral 7B Instructで試してみたところ、75 トークン毎秒の速度で、わずか 0.000218 米ドルというコストで以下の応答を生成しました。

1 Based on the given order history, it appears that the customer has received
2 orders for baby-related items (diapers, wipes, formula, clothes, bath items)
3 at the address "789 Suburb St, Quietville" and garden-related items (soil,
4 seeds, watering can, herb garden kit) at the address "123 Country Lane, Rural
5 Town". The customer has also received books (fiction, non-fiction, literary
6 classics, mystery-thriller, science fiction, biographies, children's books)
7 at the address "456 City Apt, Metroville".

8

9 Given that the new order is for a "thermometer", which doesn't seem to fit
10 into any of the categories that the customer has previously ordered, it's
11 hard to accurately predict the shipping address based solely on the order

```
12 history. However, given the 50%-50% split between baby-related and
13 garden-related items, it could somewhat lean towards the Baby-related items
14 address ("789 Suburb St, Quietville"). But remember, this is an assumption
15 and cannot be definitively confirmed without more context or information.
```

このテクニックのオーバーヘッドとコストは、チェックアウト体験をより魔法のようなものにするために価値があるのでしょうか?多くのオンライン小売業者にとって、間違いなくその価値はあります。そして現状を見る限り、特にオープンソースモデルホスティングプロバイダーが価格競争を繰り広げる中で、AI 計算のコストは今後も下がっていく一方でしょう。



このような種類のチャット補完を最適化するには、[プロンプトテンプレート](#)と[構造化 IO](#)、そして[レスポンスフェンシング](#)を使用してください。

適応型フィールド順序付け

フォームフィールドの表示順序は、ユーザー体験と入力完了率に大きな影響を与えることがあります。GenUIを使用することで、ユーザーのコンテキストと各フィールドの重要性に基づいて、フィールドの順序を動的に調整することができます。例えば、ユーザーがフィットネスアプリの登録フォームに記入している場合、フォームはフィットネスの目標や好みに関連するフィールドを優先的に表示し、プロセスをより関連性が高く魅力的なものにすることができます。

パーソナライズされたマイクロコピー

フォームに関連する説明文、エラーメッセージ、その他のマイクロコピーも、GenUIを使用してパーソナライズすることができます。「無効なメールアドレスです」といった一般的なエラーメッセージの代わりに、「注文確認を受け取るために有効なメールアドレスを入力してください」というような、より役立つコンテキストに沿ったメッセージを生成することができます。このようなパーソナライズされた要素により、フォーム入力の体験がよりユーザーフレンドリーで、ストレスの少ないものになります。

パーソナライズされた検証

パーソナライズされたマイクロコピーと同様の考え方で、AI を使用して魔法のような方法でフォームを検証することができます。AI を使用してユーザープロフィールフォームを検証し、セマンティックレベルで潜在的な間違いを探すことを想像してみてください。

Create your account

Full name

Obie Fernandez

Email

obiefernandez@gmail.com



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

Country ⓘ

 United States



Password

.....



✓ Nice work. This is an excellent password.

図 9. セマンティック検証が行われているのがわかりますか？

プログレッシブ・ディスクロージャー

GenUI は、ユーザーのコンテキストに基づいて必要不可欠なフォームフィールドを判断し、必要に応じて追加のフィールドを徐々に表示することができます。このプログレッシブ・ディスクロージャー技術により、認知負荷を軽減し、フォーム入力プロセスをより管理しやすくすることができます。例えば、ユーザーが基本サブスクリプションに登録する場合、フォームは最初に必須フィールドのみを表示し、ユーザーが進行するか特定のオプションを選択すると、関連する追加フィールドを動的に表示することができます。

コンテキストを考慮した説明文

ツールチップは、ユーザーが特定の要素にホバーまたは操作する際に、追加情報やガイダンスを提供するために頻繁に使用されます。「コンテキストに応じたコンテンツ生成」アプローチを使用することで、ユーザーのコンテキストに適応し、関連情報を提供するツールチップを生成することができます。例えば、ユーザーが複雑な機能を探している場合、ツールチップは過去の操作やスキルレベルに基づいてパーソナライズされたヒントや例を提供することができます。

説明文(手順、説明文、ヘルプメッセージなど)は、ユーザーのコンテキストに基づいて動的に生成することができます。一般的な説明を提示する代わりに、LLM を使用してユーザーの特定のニーズや質問に合わせたテキストを生成することができます。例えば、ユーザーがプロセスの特定のステップで苦戦している場合、説明文はパーソナライズされたガイダンスやトラブルシューティングのヒントを提供することができます。

マイクロコピーとは、ボタンのラベル、エラーメッセージ、確認プロンプトなど、アプリケーション内でユーザーを導く小さなテキストのことを指します。[文脈に応じたコンテンツ生成](#)のアプローチをマイクロコピーに適用することで、ユーザーの行動に応じて適切なテキストを提供するを作成できます。例えば、ユーザーが重要な操作を行う際、確認プロンプトを動的に生成して、明確でパーソナライズされたメッセージを提供することができます。

パーソナライズされた説明文やツールチップは、新規ユーザーのオンボーディングプロセスを大きく改善することができます。文脈に応じたガイダンスと例を提供することで、ユーザーがアプリケーションを素早く理解し操作できるようになり、学習曲線を緩やかにして採用率を高めることができます。

動的で文脈を理解したクロム要素も、アプリケーションをより直感的で魅力的なものにすることができます。テキストがユーザーの特定のニーズや興味に合わせて調整されていれば、ユーザーはより積極的に機能を操作し、探索ようになります。

ここまでは、既存の UI パラダイムを AI で強化するアイデアについて説明してきましたが、ユーザーインターフェースの設計と実装をより抜本的に見直すにはどうすればよいのでしょうか？

ジェネラティブ UI の定義

デザイナーが固定的で静的なインターフェースを作成する従来の UI 設計とは異なり、GenUI は、ソフトウェアが柔軟でパーソナライズされた体験を提供し、リアルタイムで進化し適応できる未来を示唆しています。AI 駆動型の対話インターフェースを使用するたびに、AI がユーザーの特定のニーズに適応することを可能にしています。GenUI は、その適応性のレベルをソフトウェアのに適用することで、さらに一歩進んでいます。

今日 GenUI のアイデアを実験できる理由は、がすでにプログラミングを理解し、その基本的な知識に UI テクノロジーやフレームワークが含まれているからです。問題は、大規模言語モデルを使用して、テキスト、画像、レイアウト、さらにはインターフェース全体を、各ユーザーに合わせて生成できるかどうかです。モデルは、ユーザーの過去のインタラクション、明示された好み、人口統計情報、現在の使用状況などのさまざまな要因を考慮して、高度にパーソナライズされた関連性の高いインターフェースを作成するよう指示することができます。

1. **動的で適応的:**従来の UI 設計では、すべてのユーザーに対して同じ固定的で静的なインターフェースを作成します。対照的に、GenUI ではユーザーのニーズや文脈に基づいて動的に適応し変化するインターフェースを実現できます。これは、同じアプリケーションが異なるユーザーに対して、あるいは同じユーザーでも異なる状況で、異なるインターフェースを提示できることを意味します。
2. **大規模なパーソナライゼーション:**従来の設計では、時間とリソースの制約により、各ユーザーにパーソナライズされた体験を作成することは現実的ではありませんでした。一方、GenUI は大規模なパーソナライゼーションを可能にします。を活用することで、デザイナーは各ユーザーセグメントに対して手動でインターフェースを設計・開発することなく、各ユーザーの固有のニーズと好みに自動的に適応するインターフェースを作成できます。
3. **成果重視:**従来の UI 設計では、視覚的に魅力的で機能的なインターフェースの作成に重点が置かれていました。GenUI でもこれらの側面は重要ですが、主な焦点は望ましいユーザー成果の達成に移ります。GenUI は、純粋に美的な考慮よりも、使いやすさと効果を優先し、各ユーザーの特定の目標やタスクに最適化されたインターフェースの作成を目指します。
4. **継続的な学習と改善:** GenUI システムは、ユーザーのインタラクションとフィードバックに基づいて継続的に学習し改善することができます。ユーザーが生成されたインターフェースを使用する中で、はユーザーの行動、好み、成果に関するデータを収集し、この情報を使用して将来のインターフェース生成を改良し最適化できます。この反復的な学習プロセスにより、GenUI システムは時間とともにユーザーのニーズをより効果的に満たすことができるようになります。

重要な点として、GenUI は設計の提案を提供したり特定の設計タスクを自動化したりする AI 支援設計ツールとは異なります。これらのツールは設計プロセスの効率化に役立ちますが、最終的な決定と静的なインターフェースの作成はデザイナーに依存しています。一方、GenUI では、AI システムがユーザーデータと文脈に基づいてインターフェースの実際の生成と適応により積極的な役割を果たします。

GenUI は、ユーザーインターフェース設計へのアプローチを、画一的なソリューションから、高度にパーソナライズされた適応型エクスペリエンスへと大きく転換するもの

です。AI の力を活用することで、GenUI は私たちのデジタル製品やサービスとの関わり方を革新し、より直感的で魅力的、そして個々のユーザーにとって効果的なインターフェースを作り出す可能性を秘めています。

例

GenUI の概念を説明するために、「FitAI」という仮想のフィットネスアプリケーションを考えてみましょう。このアプリは、個々のユーザーの目標、フィットネスレベル、好みに基づいて、パーソナライズされたワークアウトプランと栄養アドバイスを提供することを目的としています。

従来の UI デザインアプローチでは、FitAI はすべてのユーザーに対して同じ固定された画面と要素を持つことになります。しかし、GenUI を使用すると、アプリのインターフェースは各ユーザーの固有のニーズとコンテキストに動的に適応することができます。

このアプローチは 2024 年時点では実装を想像するのが少し難しく、十分な ROI が得られない可能性もありますが、実現は可能です。

以下が、その仕組みの例です：

1. オンボーディング：

- 標準的なアンケートの代わりに、FitAI は対話型 AI を使用して、ユーザーの目標、現在のフィットネスレベル、好みに関する情報を収集します。
- この初期のやり取りに基づいて、AI はユーザーの目標に最も関連する機能と情報を強調した、パーソナライズされたダッシュボードレイアウトを生成します。
- 現在の AI 技術では、パーソナライズされたダッシュボードを構成するために使用できる画面コンポーネントの選択肢があるかもしれません。
- 将来の AI 技術では、経験豊富な UI デザイナーの役割を担い、ダッシュボードを一から作成する可能性があります。

2. ワークアウトプランナー:

- ・ワークアウトプランナーのインターフェースは、ユーザーの経験レベルと利用可能な器具に合わせて AI が適応させます。
- ・器具を持たない初心者には、詳細な説明とビデオを含む簡単な自重トレーニングを表示します。
- ・ジムにアクセスできる上級者には、説明が少ない、より複雑なルーティンを表示します。
- ・ワークアウトプランナーのコンテンツは、単に大きなスーパーセットからフィルタリングされるものではありません。ユーザーに関する既知のすべての情報を含むコンテキストでクエリされるナレッジベースに基づいて、その場で生成されます。

3. 進捗追跡:

- ・進捗追跡のインターフェースは、ユーザーの目標と取り組みパターンに基づいて進化します。
- ・減量を主な目的とするユーザーには、体重の推移グラフとカロリー消費統計を目立つように表示します。
- ・筋力増強を目指すユーザーには、筋力の向上と体組成の変化を強調します。
- ・AI はアプリケーションのこの部分をユーザーの実際の進捗に適応させることができます。進捗が一定期間停滞した場合、アプリは停滞の理由を明らかにしてそれを緩和するために、ユーザーに働きかけるモードに切り替わります。

4. 栄養アドバイス:

- ・栄養セクションは、ユーザーの食事の好みと制限に適応します。
- ・ビーガンのユーザーには、植物性の食事の提案とタンパク質源を表示します。

- ・グルテン不耐症のユーザーには、推奨食品からグルテンを含む食品を自動的に除外します。
- ・ここでも、コンテンツはすべてのユーザーに適用される大規模な食事データのスーパーセットから抽出されるのではなく、ユーザーの特定の状況と制約に基づいて適応可能な情報を含むナレッジベースから合成されます。
- ・例えば、レシピは、ユーザーのフィットネスレベルと体の状態が進化するにつれて常に変化するカロリーニーズに合わせて、材料の仕様を生成します。

5. モチベーション要素:

- ・アプリのモチベーションコンテンツと通知は、ユーザーの性格タイプと異なるモチベーションの反応に基づいてパーソナライズされます。
- ・励ましのメッセージを受け取るユーザーもいれば、よりデータ重視のフィードバックを受け取るユーザーもいます。

この例では、GenUIによって FitAI は各ユーザーに高度にカスタマイズされた体験を提供し、エンゲージメント、満足度、そしてフィットネス目標の達成可能性を高めることができます。インターフェース要素、コンテンツ、さらにはアプリの「パーソナリティ」までもが、個々のユーザーのニーズと好みに最適に適応します。

成果指向設計へのシフト

GenUI は、ユーザーインターフェース設計におけるアプローチの根本的な変化を表しています。特定のインターフェース要素の作成から、より包括的な成果指向のアプローチへと移行しています。この変化には以下のような重要な意味があります:

1. ユーザーの目標への焦点:

- ・デザイナーは、特定のインターフェースコンポーネントよりも、ユーザーの目標と望む成果についてより深く考える必要があります。

- ・ユーザーが効率的かつ効果的に目標を達成できるインターフェースを生成できるシステムの作成に重点が置かれます。
- ・事前に定義された画面仕様ではなく、AI ベースのデザイナーが_その場で__一から_ユーザー体験を生成できるような新しい UI フレームワークが登場するでしょう。

2. デザイナーの役割の変化:

- ・デザイナーは、固定レイアウトの作成から、AI システムがインターフェースを生成する際に従うべきルール、制約、ガイドラインの定義へと移行します。
- ・GenUI システムを効果的に導くために、データ分析、AI プロンプトエンジニアリング、システム思考などの分野でスキルを開発する必要があります。

3. ユーザーリサーチの重要性:

- ・GenUI の文脈では、デザイナーはユーザーの好みだけでなく、それらの好みやニーズが異なる状況でどのように変化するかを理解する必要があるため、ユーザーリサーチがさらに重要になります。
- ・効果的なインターフェースを生成する AI の能力を改良し向上させるために、継続的なユーザーテストとフィードバックループが不可欠となります。

4. 多様性のためのデザイン:

- ・単一の「完璧な」インターフェースを作成する代わりに、デザイナーは複数の可能なバリエーションを考慮し、システムが多様なユーザーニーズに適したインターフェースを生成できるようにする必要があります。
- ・これには、エッジケースのデザインや、生成されたインターフェースが異なる構成全体で使いやすさとアクセシビリティを維持することを確保することが含まれます。
- ・製品の差別化は、ユーザー心理学に対する異なる視点と、競合他社が利用できないユニークなデータセットや知識ベースの活用という新しい側面を持つようになります。

課題と考慮事項

GenUI は魅力的な可能性を提供する一方で、いくつかの課題と考慮すべき点も存在します：

1. 技術的な制限：

- ・ 現在の AI 技術は高度ではありますが、複雑なユーザーの意図を理解し、真に状況を認識したインターフェースを生成する面ではまだ制限があります。
- ・ 特に性能の低いデバイスでは、インターフェース要素のリアルタイム生成に関するパフォーマンスの問題があります。

2. データ要件：

- ・ ユースケースによっては、効果的な GenUI システムがパーソナライズされたインターフェースを生成するために大量のユーザーデータを必要とする可能性があります。
- ・ 本物のユーザーデータを倫理的に収集する課題は、データプライバシーとセキュリティの懸念を引き起こし、GenUI モデルのトレーニングに使用されるデータの潜在的なバイアスも問題となります。

3. 使いやすさと一貫性：

- ・ 少なくともこの実践が広く普及するまでは、常に変化するインターフェースを持つアプリケーションを引き起こす可能性があります。ユーザーが慣れ親しんだ要素を見つけたり、効率的にナビゲートしたりすることが困難になる可能性があるためです。
- ・ パーソナライゼーションと、一貫性があり学習可能なインターフェースの維持との間のバランスを取ることが重要になります。

4. AI への過度の依存:

- デザインの決定を AI システムに過度に委ねることで、魅力に欠ける、問題のある、あるいは単に機能しないインターフェースの選択につながるリスクがあります。
- 人間による監督と AI が生成したデザインを上書きする能力は、予見可能な将来において重要であり続けるでしょう。

5. アクセシビリティの懸念:

- 動的に生成されるインターフェースが障害を持つユーザーにとってアクセシブルであり続けることを保証することは、全く新しい課題を提示します。これは、一般的なシステムが示すアクセシビリティコンプライアンスの低さを考えると懸念されます。
- 一方で、AI デザイナーは、アクセシビリティへの配慮を_組み込み_で実装し、障害のないユーザー向けの UI を構築するのと同じように、アクセシブルなインターフェースをその場で構築する機能を持つ可能性があります。
- いずれにせよ、GenUI システムは堅固なアクセシビリティガイドラインとテストプロセスを備えて設計される必要があります。

6. ユーザーの信頼と透明性:

- ユーザーは、自分について「詳しすぎる」インターフェースや、理解できない方法で変化するインターフェースに不快感を覚える可能性があります。
- インターフェースがどのように、そしてなぜパーソナライズされているのかについての透明性を確保することが、ユーザーの信頼を構築する上で重要になります。

将来の展望と機会

生成的 UI (GenUI) の未来は、デジタル製品やサービスとの私たちの関わり方を革新する大きな可能性を秘めています。この技術が進化し続けるにつれ、ユーザーインターフェースの設計、実装、体験の方法に大きな変革が起こることが予想されます。私は、GenUI こそが、現在の SF と考えられている領域に私たちのソフトウェアを押し上げる現象だと考えています。

GenUI の最も興味深い展望の一つは、重度の障害を持つ人々がソフトウェアの使用から完全に排除されないようにすることを超えて、大規模なアクセシビリティを向上させる可能性です。インターフェースが個々のユーザーのニーズに自動的に適応することで、デジタル体験はかつてないほどインクルーシブなものになる可能性があります。若年層や視覚障害のあるユーザーのために自動的に文字を大きくしたり、認知障害のある人のためにレイアウトを簡略化したりするインターフェースを想像してみてください。これらはすべて、手動での設定や別個の「アクセシブル」版のアプリケーションを必要としません。

GenUI のパーソナライゼーション機能は、幅広いデジタル製品においてユーザーエンゲージメント、満足度、ロイヤリティの向上を促進する可能性が高いでしょう。インターフェースが個々の好みや行動により適応するようになれば、ユーザーはデジタル体験をより直感的で楽しいものと感じ、それが技術とのより深い、より意味のある相互作用につながる可能性があります。

GenUI はまた、新規ユーザーのオンボーディングプロセスを変革する可能性も秘めています。各ユーザーの習熟度に素早く適応する直感的でパーソナライズされた初回ユーザー体験を作り出すことで、新しいアプリケーションの学習曲線を大幅に減らすことができます。これにより、新機能の採用率が向上し、ユーザーが新機能を探索する際の自信も高まる可能性があります。

もう一つの興味深い可能性は、GenUI が異なるデバイスやプラットフォーム間で一貫したユーザー体験を維持しながら、特定の使用コンテキストに最適化できることです。これにより、スマートフォンやタブレット、デスクトップコンピュータ、拡張現実メガネな

どの新興技術まで、ますます断片化するデバイス環境全体で一貫した体験を提供するという長年の課題を解決できる可能性があります。

GenUI のデータ駆動型の性質は、UI デザインの迅速な反復と改善の機会を生み出します。生成されたインターフェースとユーザーの相互作用に関するリアルタイムデータを収集することで、デザイナーや開発者はユーザーの行動と好みについて前例のない洞察を得ることができます。このフィードバックループにより、仮定や限定的なユーザーテストではなく、実際の使用パターンに基づいた UI デザインの継続的な改善が可能になります。

この変化に備えるため、デザイナーはスキルセットとマインドセットを進化させる必要があります。固定レイアウトの作成から、AI 駆動のインターフェース生成に情報を提供する包括的なデザインシステムとガイドラインの開発へと重点が移行します。デザイナーは、GenUI システムを効果的に導くために、データ分析、AI 技術、システム思考への深い理解を養う必要があります。

さらに、GenUI がデザインと技術の境界線を曖昧にするにつれ、デザイナーは開発者やデータサイエンティストとより緊密に協力する必要があります。この学際的なアプローチは、視覚的に魅力的でユーザーフレンドリーなだけでなく、技術的にも堅牢で倫理的にも健全な GenUI システムを作成する上で重要になります。

技術が成熟するにつれて、GenUI の倫理的影響も前面に出てくるでしょう。デザイナーは、インターフェースデザインにおける責任ある AI 使用のフレームワークを開発する上で重要な役割を果たし、パーソナライゼーションがプライバシーを損なったり、非倫理的方法でユーザーの行動を操作したりすることなく、ユーザー体験を向上させることを確保する必要があります。

将来を見据えると、GenUI は興味深い機会と重要な課題の両方を提示しています。世界中のユーザーにとって、より直感的で効率的、そして満足度の高いデジタル体験を創造する可能性を秘めています。デザイナーに新しいスキルの習得と適応を求める一方で、ヒューマンコンピュータインタラクションの未来を意義深い方法で形作る前例のない機会も提供しています。完全に実現された GenUI システムへの道のりは間違いなく複雑ですが、ユーザー体験とデジタルアクセシビリティの向上という

観点での潜在的な報酬は、その未来を目指す価値のあるものになっています。

インテリジェントワークフローオーケストレーション



「インテリジェントワークフローオーケストレーション」アプローチは、AI コンポーネントを活用してアプリケーション内の複雑なワークフローを動的にオーケストレーションし最適化することに重点を置いています。その目標は、より効率的で応答性が高く、リアルタイムデータやコンテキストに適応できるアプリケーションを作成することです。

本章では、インテリジェントワークフローオーケストレーションアプローチを支える主要な原則とパターンについて探究します。AI を活用してタスクのインテリジェントな振り分け、意思決定の自動化、そしてユーザーの行動、システムのパフォーマンス、ビジネスルールなどの様々な要因に基づいてワークフローを動的に適応させる方法について検討します。実践的な例や実世界のシナリオを通じて、アプリケーションのワークフローを効率化し最適化する AI の変革的な可能性を実証します。

複雑なビジネスプロセスを持つエンタープライズアプリケーションを構築する場合でも、動的なユーザージャーニーを持つ消費者向けアプリケーションを構築する場合でも、本章で説明するパターンとテクニックは、全体的なユーザーエクスペリエンスを向上させ、ビジネス価値を推進するインテリジェントで効率的なワークフローを作成するための知識とツールを提供します。

ビジネスニーズ

ワークフロー管理の従来のアプローチは、多くの場合、事前に定義されたルールと静的な決定木に依存しており、これらは硬直的で柔軟性に欠け、現代のアプリケーションの動的な性質に対応できない可能性があります。

E コマースアプリケーションが複雑な注文処理プロセスを扱う必要があるシナリオを考えてみましょう。このワークフローには、注文の検証、在庫確認、支払い処理、配送、顧客への通知など、複数のステップが含まれる可能性があります。各ステップには、独自のルール、依存関係、外部との統合、例外処理メカニズムが存在する可能性があります。このようなワークフローを手動で、またはハードコードされたロジックで管理すると、すぐに煩雑になり、エラーが発生しやすく、メンテナンスが困難になる可能性があります。

さらに、アプリケーションが拡大し、同時接続ユーザー数が増加するにつれて、ワークフローはリアルタイムデータとシステムパフォーマンスに基づいて自己適応し最適化する必要が出てくるかもしれません。例えば、トラフィックのピーク時には、アプリケーションは特定のタスクを優先し、リソースを効率的に割り当て、スムーズなユーザーエクスペリエンスを確保するためにワークフローを動的に調整する必要があるかもしれません。

ここで「インテリジェントワークフローオーケストレーション」アプローチが重要になってきます。AI コンポーネントを活用することで、開発者はインテリジェントで適応性があり、自己最適化するワークフローを作成することができます。AI は膨大な量のデー

タを分析し、過去の実験から学習し、リアルタイムで情報に基づいた判断を下してワークフローを効果的にオーケストレーションすることができます。

主要なメリット

1. **効率性の向上:** AI はタスクの割り当て、リソースの利用、ワークフローの実行を最適化し、処理時間の短縮と全体的な効率性の向上をもたらします。
2. **適応性:** AI 駆動のワークフローは、ユーザー需要の変動、システムパフォーマンス、またはビジネス要件などの変化する条件に動的に適応し、アプリケーションの応答性とレジリエンスを確保します。
3. **意思決定の自動化:** AI はワークフロー内の複雑な意思決定プロセスを自動化し、手動での介入を減らし、ヒューマンエラーのリスクを最小限に抑えることができます。
4. **パーソナライゼーション:** AI はユーザーの行動、好み、コンテキストを分析して、ワークフローをパーソナライズし、個々のユーザーにカスタマイズされた体験を提供することができます。
5. **スケーラビリティ:** AI 駆動のワークフローは、パフォーマンスや信頼性を損なうことなく、増加するデータ量やユーザーとのインタラクションにシームレスに対応できます。

以下のセクションでは、インテリジェントワークフローの実装を可能にする主要なパターンと技術について探り、AI が現代のアプリケーションにおけるワークフロー管理をどのように変革しているかを実例を交えて紹介します。

主要パターン

アプリケーションにインテリジェントなワークフローオーケストレーションを実装するために、開発者は AI の力を活用するいくつかの重要なパターンを利用することができます。これらのパターンは、ワークフローの設計と管理に体系的なアプローチを提供

し、リアルタイムデータとコンテキストに基づいてプロセスを適応、最適化、自動化することを可能にします。インテリジェントなワークフローオーケストレーションにおける基本的なパターンのいくつかを見ていきましょう。

動的タスクルーティング

このパターンでは、タスクの優先度、リソースの可用性、システムパフォーマンスなどの様々な要因に基づいて、AIを使用してワークフロー内のタスクをインテリジェントにルーティングします。AI アルゴリズムは各タスクの特性を分析し、システムの現状を考慮した上で、最適なリソースまたは処理経路にタスクを割り当てるための情報に基づいた判断を行います。動的タスクルーティングにより、タスクが効率的に分散され実行されることが保証され、ワークフロー全体のパフォーマンスが最適化されます。

```
1  class TaskRouter
2      include Raix::ChatCompletion
3      include Raix::FunctionDispatch
4
5      attr_accessor :task
6
7      # list of functions that can be called by the AI entirely at its
8      # discretion depending on the task received
9
10     function :analyze_task_priority do
11         TaskPriorityAnalyzer.perform(task)
12     end
13
14     function :check_resource_availability, # ...
15     function :assess_system_performance, # ...
16     function :assign_task_to_resource, # ...
17
```

```
18 DIRECTIVE = "You are a task router, responsible for intelligently
19 assigning tasks to available resources based on priority, resource
20 availability, and system performance..."
21
22 def initialize(task)
23     self.task = task
24     transcript << { system: DIRECTIVE }
25     transcript << { user: task.to_json }
26 end
27
28 def perform
29     while task.unassigned?
30         chat_completion
31
32         # todo: add max loop counter and break
33     end
34
35     # capture the transcript for later analysis
36     task.update(routing_transcript: transcript)
37 end
38 end
```

29 行目のwhile 式で作成されたループに注目してください。このループはタスクが割り当てられるまで AI に対してプロンプトを続けます。35 行目では、後の分析やデバッグが必要になった場合に備えて、タスクのトランスクリプトを保存します。

コンテキストに基づく意思決定

ワークフロー内でコンテキストを認識した決定を行うために、非常によく似たコードを使用できます。ユーザー設定、過去のパターン、リアルタイム入力などの関連データポイントを分析することで、AI コンポーネントはワークフロー内の各決定ポイントで最適なアクションを判断できます。各ユーザーやシナリオの特定のコンテキストに基づいてワークフローの動作を適応させ、パーソナライズされた最適化された体験を提供します。

適応型ワークフロー構成

このパターンは、変化する要件や条件に基づいてワークフローを動的に構成および調整することに焦点を当てています。AI はワークフローの現状を分析し、ボトルネックや非効率性を特定し、パフォーマンスを最適化するためにワークフロー構造を自動的に修正します。適応型ワークフロー構成により、アプリケーションは手動介入を必要とせずに継続的に進化し、プロセスを改善することができます。

例外処理とリカバリー

例外処理とリカバリーは、インテリジェントなワークフローオーケストレーションの重要な側面です。AI コンポーネントと複雑なワークフローを扱う場合、システムの安定性と信頼性を確保するために、例外を予測し適切に処理することが不可欠です。

インテリジェントなワークフローにおける例外処理とリカバリーに関する主要な考慮事項とテクニックは以下の通りです：

1. **例外の伝播：**ワークフローコンポーネント全体で一貫した例外伝播アプローチを実装します。コンポーネント内で例外が発生した場合、それをキャッチし、ログに記録し、オーケストレーターまたは例外処理を担当する個別のコンポーネントに伝播する必要があります。

この考え方は、例外処理を一元化し、例外が暗黙的に無視されることを防ぎ、さらに**インテリジェントなエラー処理**の可能性を開くことです。

2. **リトライメカニズム**: リトライメカニズムは、ワークフローの回復力を向上させ、一時的な障害を適切に処理するのに役立ちます。ネットワーク接続やリソースの利用不可などの一時的または回復可能な例外に対して、指定された遅延後に自動的に再試行できるリトライメカニズムを実装することを強く推奨します。AI パワードのオーケストレーターや例外ハンドラーを持つことで、指数関数的フォールバックなどの固定アルゴリズムに依存する機械的な性質のリトライ戦略を使用する必要がなくなります。例外の処理方法の決定を AI コンポーネントの「裁量」に任せることができます。
3. **フォールバック戦略**: AI コンポーネントが有効な応答を提供できない場合やエラーが発生した場合(最先端の性質上、よくある出来事です)、ワークフローを継続できるようにフォールバックメカニズムを用意しておきます。これには、デフォルト値の使用、代替アルゴリズム、または決定を行い、ワークフローを前進させるための**人間によるループ制御**が含まれる場合があります。
4. **補償アクション**: オーケストレーターの指示には、自動的に解決できない例外を処理するための補償アクションに関する指示を含める必要があります。補償アクションは、失敗した操作の影響を元に戻すまたは軽減するために実行される手順です。例えば、支払い処理のステップが失敗した場合、補償アクションはトランザクションをロールバックしてユーザーに通知することができます。補償アクションは、例外が発生した場合にデータの一貫性と整合性を維持するのに役立ちます。
5. **例外のモニタリングとアラート**: 重要な例外を検出し、関連する利害関係者に通知するためのモニタリングとアラートメカニズムを設定します。オーケストレーターは、例外が特定の制限を超えた場合や、特定のタイプの例外が発生した場合にアラートをトリガーするためのしきい値とルールを認識できるようにします。これにより、システム全体に影響が及ぶ前に、問題を積極的に特定して解決することができます。

以下は、Ruby ワークフローコンポーネントにおける例外処理とリカバリーの例です：

```
1  class InventoryManager
2    def check_availability(order)
3      begin
4        # Perform inventory check logic
5        inventory = Inventory.find_by(product_id: order.product_id)
6        if inventory.available_quantity >= order.quantity
7          return true
8        else
9          raise InsufficientInventoryError,
10             "Insufficient inventory for product #{order.product_id}"
11        end
12      rescue InsufficientInventoryError => e
13        # Log the exception
14        logger.error("Inventory check failed: #{e.message}")
15
16        # Retry the operation after a delay
17        retry_count ||= 0
18        if retry_count < MAX_RETRIES
19          retry_count += 1
20          sleep(RETRY_DELAY)
21          retry
22        else
23          # Fallback to manual intervention
24          NotificationService.admin("Inventory check failed: Order #{order.id}")
25          return false
26        end
27      end
28    end
29  end
```

この例では、InventoryManager コンポーネントが特定の注文に対する商品の在庫状況を確認します。利用可能な数量が不足している場合、InsufficientInventoryError を発生させます。この例外はキャッチされ、ログに記録され、リトライメカニズムが実装されます。リトライ制限を超えた場合、管理者に通知することで手動介入へのフォールバックを行います。

堅牢な例外処理とリカバリーメカニズムを実装することで、インテリジェントワークフローの回復力と保守性を確保し、予期しない状況を適切に処理することができます。

これらのパターンはインテリジェントワークフローオーケストレーションの基盤を形成し、さまざまなアプリケーションの特定の要件に合わせて組み合わせたり適応させたりすることができます。これらのパターンを活用することで、開発者は柔軟で回復力があり、パフォーマンスとユーザーエクスペリエンスに最適化されたワークフローを作成できます。

次のセクションでは、これらのパターンを実践でどのように実装できるかを探り、実世界の例とコードスニペットを使用して AI コンポーネントのワークフロー管理への統合を説明します。

インテリジェントワークフローオーケストレーションの実践的な実装

インテリジェントワークフローオーケストレーションの主要なパターンについて探究したところで、これらのパターンを実世界のアプリケーションでどのように実装できるかを見ていきましょう。AI コンポーネントのワークフロー管理への統合を説明するために、実践的な例とコードスニペットを提供します。

インテリジェント注文処理システム

Ruby on Railsの E コマースアプリケーションで AI 駆動のOrderProcessor コンポーネントを使用して、インテリジェントワークフローオーケストレーションを実装する実践的な例を見ていきましょう。このOrderProcessor は、第 3 章でMultitude of Workersを議論した際に最初に出会ったプロセスマネージャーエンタープライズ統合の概念を実現します。このコンポーネントは注文処理ワークフローの管理、中間結果に基づくルーティング決定、さまざまな処理ステップの実行の調整を担当します。

注文処理プロセスには、注文の検証、在庫確認、支払い処理、配送など、複数のステップが含まれます。各ステップは特定のタスクを実行し、結果をOrderProcessor に返す個別のワーカークラスとして実装されています。これらのステップは必須ではなく、必ずしも厳密な順序で実行される必要ありません。

以下がOrderProcessor の実装例です。Raixから 2 つのミックスインを使用しています。1 つ目(ChatCompletion)はチャット補完を行う機能を提供し、これによって AI コンポーネントとなります。2 つ目(FunctionDispatch)は AI による関数呼び出しを可能にし、プロンプトに対してテキストメッセージではなく関数呼び出しで応答できるようにします。

ワーカー関数(validate_order、check_inventory など)は、それぞれのワーカークラスに処理を委譲します。これらのワーカークラスは AI コンポーネントでも非 AI コンポーネントでもかまいませんが、作業結果を文字列として表現できる形式で返す必要があるという要件があります。



このパートの他のすべての例と同様に、このコードは実質的に擬似コードであり、パターンの意味を伝え、独自の創造を促すことのみを目的としています。パターンの完全な説明と完全なコード例はパート 2 に含まれています。

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

```
30     ]
31     end
32
33     # implementation of functions that can be called by the AI
34     # entirely at its discretion, depending on the needs of the order
35
36     def validate_order
37         OrderValidationWorker.perform(@order)
38     end
39
40     def check_inventory
41         InventoryCheckWorker.perform(@order)
42     end
43
44     def process_payment
45         PaymentProcessingWorker.perform(@order)
46     end
47
48     def schedule_shipping
49         ShippingSchedulerWorker.perform(@order)
50     end
51
52     def send_confirmation
53         OrderConfirmationWorker.perform(@order)
54     end
55
56     def finished_processing
57         @order.update!(transcript:, processed_at: Time.current)
58         stop_looping!
```

59 **end**

60 **end**

この例では、OrderProcessor はオーダーオブジェクトで初期化され、大規模言語モデルに特有の会話トランスクリプト形式でワークフロー実行の記録を保持します。AI には、注文の検証、在庫確認、支払い処理、出荷など、様々な処理ステップの実行を統制する完全な制御権が与えられています。

chat_completion メソッドが呼び出されるたびに、トランスクリプトが AI に送信され、関数呼び出しとして応答を提供します。前のステップの結果を分析し、適切なアクションを決定するのは完全に AI に委ねられています。例えば、在庫確認で在庫レベルが低いことが判明した場合、OrderProcessor は補充タスクをスケジュールすることができます。支払い処理が失敗した場合は、再試行を開始したり、カスタマーサポートに通知したりすることができます。

上記の例では、補充やカスタマーサポートへの通知の関数は定義されていませんが、もちろん定義することは可能です。

トランスクリプトは関数が呼び出されるたびに成長し、各ステップの結果や AI が生成した次のステップの指示を含むワークフロー実行の記録として機能します。このトランスクリプトは、デバッグ、監査、注文処理プロセスの可視化に使用できます。

OrderProcessor で AI を活用することで、e コマースアプリケーションはリアルタイムデータに基づいてワークフローを動的に適応させ、インテリジェントに例外を処理することができます。AI コンポーネントは、情報に基づいた決定を下し、ワークフローを最適化し、複雑なシナリオでもスムーズな注文処理を確保できます。

ワーカプロセスに求められる唯一の要件は、AI が次に何をすべきかを判断する際に考慮できる理解可能な出力を返すことだけであり、このアプローチによって、通常は異なるシステム同士を統合する際に必要となる入出力マッピング作業を削減でき

ることに気付き始めるかもしれません。

インテリジェントコンテンツモデレーター

ソーシャルメディアアプリケーションは、一般的に安全で健全なコミュニティを確保するために、最低限のコンテンツモデレーションを必要とします。この例のContentModerator コンポーネントは、AIを活用してモデレーションワークフローをインテリジェントに統制し、コンテンツの特性と様々なモデレーションステップの結果に基づいて決定を下します。

モデレーションプロセスには、テキスト分析、画像認識、ユーザー評価の査定、手動レビューなど、複数のステップが含まれます。各ステップは、特定のタスクを実行し、結果をContentModerator に返す個別のワーカプロセスとして実装されています。

以下がContentModerator の実装例です：

```
1  class ContentModerator
2      include Raix::ChatCompletion
3      include Raix::FunctionDispatch
4
5      SYSTEM_DIRECTIVE = "You are a content moderator process manager,
6          tasked with the workflow involved in moderating user-generated content..."
7
8      def initialize(content)
9          @content = content
10         @transcript = [
11             { system: SYSTEM_DIRECTIVE },
12             { user: content.to_json }
13         ]
14     end
15
```



```
16  def perform
17      complete(@transcript)
18  end
19
20  def model
21      "openai/gpt-4"
22  end
23
24  # list of functions available to be called by the AI
25  # truncated for brevity
26
27  def functions
28      [
29          {
30              name: "analyze_text",
31              # ...
32          },
33          {
34              name: "recognize_image",
35              description: "Invoke to describe images...",
36              # ...
37          },
38          {
39              name: "assess_user_reputation",
40              # ...
41          },
42          {
43              name: "escalate_to_manual_review",
44              # ...
```

```
45     },
46     {
47         name: "approve_content",
48         # ...
49     },
50     {
51         name: "reject_content",
52         # ...
53     }
54 ]
55 end
56
57 # implementation of functions that can be called by the AI
58 # entirely at its discretion, depending on the needs of the order
59
60 def analyze_text
61     result = TextAnalysisWorker.perform(@content)
62     continue_with(result)
63 end
64
65 def recognize_image
66     result = ImageRecognitionWorker.perform(@content)
67     continue_with(result)
68 end
69
70 def assess_user_reputation
71     result = UserReputationWorker.perform(@content.user)
72     continue_with(result)
73 end
```

```
74
75 def escalate_to_manual_review
76     ManualReviewWorker.perform(@content)
77     @content.update!(status: 'pending', transcript: @transcript)
78 end
79
80 def approve_content
81     @content.update!(status: 'approved', transcript: @transcript)
82 end
83
84 def reject_content
85     @content.update!(status: 'rejected', transcript: @transcript)
86 end
87
88 private
89
90 def continue_with(result)
91     @transcript << { function: result }
92     complete(@transcript)
93 end
94 end
```

この例では、ContentModerator はコンテンツオブジェクトで初期化され、会話形式でモデレーション記録を保持します。AI コンポーネントは、コンテンツの特性と各ステップの結果に基づいて実行するステップを決定し、モデレーションワークフローを完全に制御します。

AI が呼び出せるワーカー関数には、analyze_text、recognize_image、assess_user_reputation、escalate_to_manual_review があります。各関数は対応するワーカープロセス(TextAnalysisWorker、ImageRecognitionWorker など)にタスクを委譲し、結果

をモデレーション記録に追加します。ただし、エスカレーション関数は終了状態として機能します。また、`approve_content` と `reject_content` 関数も終了状態として機能します。

AI コンポーネントはコンテンツを分析し、適切なアクションを判断します。コンテンツに画像参照が含まれている場合、視覚的なレビューのために `recognize_image` ワーカーを呼び出すことができます。いずれかのワーカーが有害な可能性のあるコンテンツを警告した場合、AI はコンテンツを手動レビューにエスカレーションするか、即座に拒否することを決定できます。ただし、警告の重大度に応じて、AI は判断の難しいコンテンツの取り扱いを決定する際に、ユーザー評価の結果を使用することを選択できます。ユースケースによっては、信頼できるユーザーにはより多くの投稿の自由度が与えられる場合もあります。このように様々な判断が可能です...

前回のプロセスマネージャーの例と同様に、モデレーション記録は各ステップの結果と AI が生成した決定を含むワークフローの実行記録として機能します。この記録は、監査、透明性の確保、そして時間とともにモデレーションプロセスを改善するために使用できます。

`ContentModerator` で AI を活用することで、ソーシャルメディアアプリケーションはコンテンツの特性に基づいてモデレーションワークフローを動的に適応させ、複雑なモデレーションシナリオをインテリジェントに処理できます。AI コンポーネントは、十分な情報に基づいて判断を下し、ワークフローを最適化し、安全で健全なコミュニティ体験を確保することができます。

インテリジェントなワークフロー制御の文脈において、予測的タスクスケジューリングと例外処理および回復を実証する 2 つの例をさらに見ていきましょう。

カスタマーサポートシステムにおける予測的タスクスケジューリング

Ruby on Rails で構築されたカスタマーサポートアプリケーションでは、顧客にタイムリーな支援を提供するために、サポートチケットを効率的に管理し優先順位付けする

ことが重要です。SupportTicketScheduler コンポーネントは、チケットの緊急性、エージェントの専門知識、作業負荷などの様々な要因に基づいて、AIを活用して予測的にサポートチケットをスケジュールし、利用可能なエージェントに割り当てます。

```
1  class SupportTicketScheduler
2      include Raix::ChatCompletion
3      include Raix::FunctionDispatch
4
5      SYSTEM_DIRECTIVE = "You are a support ticket scheduler,
6          tasked with intelligently assigning tickets to available agents..."
7
8      def initialize(ticket)
9          @ticket = ticket
10         @transcript = [
11             { system: SYSTEM_DIRECTIVE },
12             { user: ticket.to_json }
13         ]
14     end
15
16     def perform
17         complete(@transcript)
18     end
19
20     def model
21         "openai/gpt-4"
22     end
23
24     def functions
25         [
26             {
```

```
27         name: "analyze_ticket_urgency",
28         # ...
29     },
30     {
31         name: "list_available_agents",
32         description: "Includes expertise of available agents",
33         # ...
34     },
35     {
36         name: "predict_agent_workload",
37         description: "Uses historical data to predict upcoming workloads",
38         # ...
39     },
40     {
41         name: "assign_ticket_to_agent",
42         # ...
43     },
44     {
45         name: "reschedule_ticket",
46         # ...
47     }
48 ]
49 end
50
51 # implementation of functions that can be called by the AI
52 # entirely at its discretion, depending on the needs of the order
53
54 def analyze_ticket_urgency
55     result = TicketUrgencyAnalyzer.perform(@ticket)
```

```
56     continue_with(result)
57 end
58
59 def list_available_agents
60     result = ListAvailableAgents.perform
61     continue_with(result)
62 end
63
64 def predict_agent_workload
65     result = AgentWorkloadPredictor.perform
66     continue_with(result)
67 end
68
69 def assign_ticket_to_agent
70     TicketAssigner.perform(@ticket, @transcript)
71 end
72
73 def delay_assignment(until)
74     until = DateTimeStandardizer.process(until)
75     SupportTicketScheduler.delay(@ticket, @transcript, until)
76 end
77
78 private
79
80 def continue_with(result)
81     @transcript << { function: result }
82     complete(@transcript)
83 end
84 end
```

この例では、`SupportTicketScheduler` はサポートチケットオブジェクトで初期化され、スケジューリング記録を維持します。AI コンポーネントはチケットの詳細を分析し、チケットの緊急度、エージェントの専門知識、予測されるエージェントの作業負荷などの要因に基づいて、予測的にチケットの割り当てをスケジュールします。

AI が呼び出せる関数には、`analyze_ticket_urgency`、`list_available_agents`、`predict_agent_workload`、`assign_ticket_to_agent` が含まれます。各関数は対応する分析機能または予測機能にタスクを委譲し、結果をスケジューリング記録に追加します。AI はまた、`delay_assignment` 関数を使用して割り当てを遅らせることもできます。

AI コンポーネントはスケジューリング記録を調査し、チケットの割り当てに関する情報に基づいた決定を行います。チケットの緊急度、利用可能なエージェントの専門知識、各エージェントの予測作業負荷を考慮して、チケットの処理に最も適したエージェントを決定します。

予測的タスクスケジューリングを活用することで、カスタマーサポートアプリケーションはチケットの割り当てを最適化し、応答時間を短縮し、全体的な顧客満足度を向上させることができます。サポートチケットの積極的かつ効率的な管理により、適切なチケットが適切なタイミングで適切なエージェントに割り当てられることが保証されます。

データ処理パイプラインにおける例外処理とリカバリー

例外を処理し、障害から回復することは、データの整合性を確保し、データの損失を防ぐために不可欠です。`DataProcessingOrchestrator` コンポーネントは AI を活用して、データ処理パイプラインにおける例外を知的に処理し、回復プロセスを統制します。


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

```
30         name: "process_data",
31         # ...
32     },
33     {
34         name: "request_fix",
35         # ...
36     },
37     {
38         name: "retry_processing",
39         # ...
40     },
41     {
42         name: "mark_data_as_failed",
43         # ...
44     },
45     {
46         name: "finished",
47         # ...
48     }
49 ]
50 end
51
52 # implementation of functions that can be called by the AI
53 # entirely at its discretion, depending on the needs of the order
54
55 def validate_data
56     result = DataValidator.perform(@data_batch)
57     continue_with(result)
58 rescue ValidationException => e
```

```
59     handle_validation_exception(e)
60 end
61
62 def process_data
63     result = DataProcessor.perform(@data_batch)
64     continue_with(result)
65 rescue ProcessingException => e
66     handle_processing_exception(e)
67 end
68
69 def request_fix(description_of_fix)
70     result = SmartDataFixer.new(description_of_fix, @data_batch)
71     continue_with(result)
72 end
73
74 def retry_processing(timeout_in_seconds)
75     wait(timeout_in_seconds)
76     process_data
77 end
78
79 def mark_data_as_failed
80     @data_batch.update!(status: 'failed', transcript: @transcript)
81 end
82
83 def finished
84     @data_batch.update!(status: 'finished', transcript: @transcript)
85 end
86
87 private
```

```
88
89     def continue_with(result)
90         @transcript << { function: result }
91         complete(@transcript)
92     end
93
94     def handle_validation_exception(exception)
95         @transcript << { exception: exception.message }
96         complete(@transcript)
97     end
98
99     def handle_processing_exception(exception)
100         @transcript << { exception: exception.message }
101         complete(@transcript)
102     end
103 end
```

この例では、`DataProcessingOrchestrator` がデータバッチオブジェクトで初期化され、処理トランスクリプトを管理します。AI コンポーネントはデータ処理パイプラインを制御し、必要に応じて例外処理と障害からの復旧を行います。

AI が呼び出し可能な関数には、`validate_data`、`process_data`、`request_fix`、`retry_processing`、`mark_data_as_failed` が含まれます。各関数は対応するデータ処理コンポーネントにタスクを委譲し、結果または例外の詳細を処理トランスクリプトに追加します。

`validate_data` ステップで検証例外が発生した場合、`handle_validation_exception` 関数は例外データをトランスクリプトに追加し、制御を AI に戻します。同様に、`process_data` ステップで処理例外が発生した場合、AI は回復戦略を決定することができます。

発生した例外の性質に応じて、AI は自身の判断で `request_fix` を呼び出すこ

とができます。これは AI 駆動の SmartDataFixer コンポーネントに委譲されます (自己修復データの章を参照)。データフィクサーは、処理を再試行できるように `@data_batch` をどのように修正すべきかについて、平易な英語での説明を受け取ります。成功する再試行とは、検証に失敗したレコードをデータバッチから削除し、人による確認のために別の処理パイプラインにコピーすることかもしれません。可能性は無限に近いと言えます。

AI 駆動の例外処理と回復を組み込むことで、データ処理アプリケーションはより強靱で耐障害性の高いものとなります。DataProcessingOrchestrator は例外を知的に管理し、データ損失を最小限に抑え、データ処理ワークフローの円滑な実行を確保します。

モニタリングとロギング

モニタリングとロギングは、AI 駆動のワークフローコンポーネントの進捗、パフォーマンス、および健全性を可視化し、開発者がシステムの動作を追跡および分析できるようにします。効果的なモニタリングとロギングの仕組みを実装することは、インテリジェントワークフローのデバッグ、監査、および継続的な改善に不可欠です。

ワークフローの進捗とパフォーマンスのモニタリング

インテリジェントワークフローの円滑な実行を確保するために、各ワークフローコンポーネントの進捗とパフォーマンスをモニタリングすることが重要です。これには、ワークフローのライフサイクル全体を通じて主要な指標とイベントを追跡することが含まれます。

モニタリングすべき重要な側面には以下が含まれます：

- 1. ワークフロー実行時間:** 各ワークフローコンポーネントがタスクを完了するまでの時間を測定します。これにより、パフォーマンスのボトルネックを特定し、全体的なワークフロー効率を最適化することができます。

2. リソース使用率:各ワークフローコンポーネントによる CPU、メモリ、ストレージなどのシステムリソースの使用状況をモニタリングします。これにより、システムが容量内で動作し、ワークロードを効果的に処理できることを確認できます。

3. エラー率と例外:ワークフローコンポーネント内でのエラーと例外の発生を追跡します。これにより、潜在的な問題を特定し、積極的なエラー処理と回復を可能にします。

4. 決定ポイントと結果:ワークフロー内の決定ポイントと AI 駆動の決定の結果をモニタリングします。これにより、AI コンポーネントの動作と有効性に関する洞察が得られます。

モニタリングプロセスによって収集されたデータは、ダッシュボードで表示したり、システムの健全性についてシステム管理者に知らせる定期レポートの入力として使用したりすることができます。



モニタリングデータは、レビューと潜在的なアクションのために AI 駆動のシステム管理者プロセスに供給することができます！

重要なイベントと決定のロギング

ロギングは、ワークフロー実行中に発生する重要なイベント、決定、および例外に関する関連情報を取得して保存する不可欠な実践です。

ロギングすべき重要な側面には以下が含まれます：

1. ワークフローの開始と完了:入力データやユーザーコンテキストなどの関連メタデータとともに、各ワークフローインスタンスの開始時刻と終了時刻をログに記録します。

2. コンポーネントの実行:入力パラメータ、出力結果、および生成された中間データを含む、各ワークフローコンポーネントの実行詳細をログに記録します。

3. **AI 決定と推論**: 基礎となる推論や信頼度スコアとともに、AI コンポーネントによる決定をログに記録します。これにより、AI 駆動の決定の透明性が確保され、監査が可能になります。

4. **例外とエラーメッセージ**: ワークフローの実行中に発生した例外やエラーメッセージを、スタックトレースや関連する文脈情報と共にログに記録します。

ログの実装方法には、ログファイルへの書き込み、データベースへのログの保存、集中ログサービスへのログの送信など、様々な手法があります。アプリケーションのアーキテクチャとの容易な統合、柔軟性、そしてスケーラビリティを提供するログフレームワークを選択することが重要です。

以下は、Ruby on Rails アプリケーションで ActiveSupport::Logger クラスを使用してログを実装する例です:

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

ワークフローコンポーネントと AI の判断ポイント全体にログ記録ステートメントを戦略的に配置することで、開発者はデバッグ、監査、分析に役立つ貴重な情報を収集

することができます。

モニタリングとログ記録の利点

インテリジェントワークフローオーケストレーションにモニタリングとログ記録を実装することには、いくつかの利点があります：

- 1. デバッグとトラブルシューティング：**詳細なログとモニタリングデータは、開発者が問題を素早く特定し診断するのに役立ちます。これらは、ワークフローの実行フロー、コンポーネント間の相互作用、発生したエラーや例外についての洞察を提供します。
- 2. パフォーマンス最適化：**パフォーマンスメトリクスのモニタリングにより、開発者はボトルネックを特定し、より効率的になるようにワークフローコンポーネントを最適化できます。実行時間、リソース使用率、その他のメトリクスを分析することで、開発者はシステム全体のパフォーマンスを改善するための適切な判断を下すことができます。
- 3. 監査とコンプライアンス：**重要なイベントと判断をログに記録することで、規制遵守と説明責任のための監査証跡を提供します。これにより、組織は AI コンポーネントによって実行されたアクションを追跡・検証し、ビジネスルールや法的要件への準拠を確保できます。
- 4. 継続的改善：**モニタリングとログ記録のデータは、インテリジェントワークフローの継続的な改善のための貴重な入力として機能します。過去のデータを分析し、パターンを特定し、AI 判断の有効性を測定することで、開発者はワークフローオーケストレーションロジックを反復的に改良・強化できます。

考慮事項とベストプラクティス

インテリジェントワークフローオーケストレーションにモニタリングとログ記録を実装する際は、以下のベストプラクティスを考慮してください：

- 1. 明確なモニタリングメトリクスの定義：**ワークフローの具体的な要件に基づいて、モニタリングが必要な重要なメトリクスとイベントを特定します。システムのパフォーマンス、健全性、動作について意味のある洞察を提供するメトリクスに焦点を当てます。

2. 詳細なログ記録の実装: ワークフローコンポーネントと AI の判断ポイント内の適切な箇所にログ記録ステートメントを配置します。入力パラメータ、出力結果、生成された中間データなど、関連するコンテキスト情報を取得します。

3. 構造化ログの使用: ログデータの解析と分析を容易にするため、構造化されたログ形式を採用します。構造化ログにより、ログエントリの検索、フィルタリング、集計が向上します。

4. ログの保持とローテーションの管理: ログファイルのストレージとライフサイクルを管理するために、ログの保持とローテーションポリシーを実装します。法的要件、ストレージの制約、分析ニーズに基づいて適切な保持期間を決定します。可能であれば、[Papertrail](#)などのサードパーティサービスにログを転送します。

5. 機密情報の保護: 個人を特定できる情報 (PII) や機密性の高いビジネスデータなど、機密情報のログ記録には注意が必要です。ログファイル内の機密情報を保護するため、データマスキングや暗号化などの適切なセキュリティ対策を実装します。

6. モニタリングと警告ツールの統合: モニタリングとログ記録データの収集、分析、可視化を一元化するために、モニタリングと警告ツールを活用します。これらのツールはリアルタイムの洞察を提供し、事前に定義されたしきい値に基づいてアラートを生成し、積極的な問題の検出と解決を促進します。これらのツールの中で私のお気に入りには[Datadog](#)です。

包括的なモニタリングとログ記録メカニズムを実装することで、開発者はインテリジェントワークフローの動作とパフォーマンスについて貴重な洞察を得ることができ、これらの洞察により、AI を活用したワークフローオーケストレーションシステムの効果的なデバッグ、最適化、継続的な改善が可能になります。

スケーラビリティとパフォーマンスの考慮事項

スケーラビリティとパフォーマンスは、インテリジェントワークフローオーケストレーションシステムの設計と実装において考慮すべき重要な側面です。同時実行されるワークフローの量と AI を活用したコンポーネントの複雑さが増加するにつれて、システ

ムが効率的にワークロードを処理し、増大する需要に応じてシームレスにスケールできることを確保することが不可欠になります。

大量の同時実行ワークフローの処理

インテリジェントなワークフローオーケストレーションシステムでは、多数の同時実行ワークフローを処理する必要があることがよくあります。スケーラビリティを確保するために、以下の戦略を検討してください：

- 1. 非同期処理：**ワークフローコンポーネントの実行を分離するために、非同期処理メカニズムを実装します。これにより、システムは各コンポーネントの完了を待たずにブロックすることなく、複数のワークフローを同時に処理できます。非同期処理は、メッセージキュー、イベント駆動アーキテクチャ、または Sidekiq のようなバックグラウンドジョブ処理フレームワークを使用して実現できます。
- 2. 分散アーキテクチャ：**AWS Lambda などのサーバーレスコンポーネントを使用するか、メインのアプリケーションサーバーと共に複数のノードやサーバーにワークロードを分散させるようにシステムアーキテクチャを設計します。これにより、ワークフローの量が増加した際に追加のノードを加えることができる水平スケーラビリティが可能になります。
- 3. 並列実行：**ワークフロー内での並列実行の機会を特定します。一部のワークフローコンポーネントは互いに独立しており、同時に実行できる場合があります。マルチスレッドや分散タスクキューなどの並列処理技術を活用することで、システムはリソース利用を最適化し、全体的なワークフロー実行時間を短縮できます。

AI 搭載コンポーネントのパフォーマンス最適化

機械学習モデルや自然言語処理エンジンなどの AI 搭載コンポーネントは、計算負荷が高く、ワークフローオーケストレーションシステム全体のパフォーマンスに影響を与える可能性があります。AI コンポーネントのパフォーマンスを最適化するために、以下の技術を検討してください：

1. **キャッシング:** AI の処理が純粋に生成的であり、チャット完了を生成するためのリアルタイム情報の検索や外部統合を必要としない場合、頻繁にアクセスされる処理や計算コストの高い操作の結果を保存して再利用するキャッシングメカニズムを検討できます。
2. **モデルの最適化:** ワークフローコンポーネントでの AI モデルの使用方法を継続的に最適化します。これには、**プロンプト蒸留**などの技術が含まれる場合もあれば、新しいモデルが利用可能になった際にそれらをテストするだけの場合もあります。
3. **バッチ処理:** GPT-4 クラスのモデルを使用している場合、データポイントやリクエストを個別に処理するのではなく、一括で処理するバッチ処理技術を活用できる可能性があります。データをバッチで処理することで、システムはリソース利用を最適化し、繰り返しのモデルリクエストによるオーバーヘッドを削減できます。

パフォーマンスのモニタリングとプロファイリング

インテリジェントなワークフロー オーケストレーションシステムのパフォーマンスのボトルネックを特定し、スケーラビリティを最適化するために、モニタリングとプロファイリングメカニズムの実装が重要です。以下のアプローチを検討してください：

1. **パフォーマンスメトリクス:** レスポンスタイム、スループット、リソース利用率、レイテンシーなどの主要なパフォーマンスメトリクスを定義し、追跡します。これらのメトリクスは、システムのパフォーマンスに関する洞察を提供し、最適化が必要な領域の特定に役立ちます。人気の AI モデルアグリゲーターである [OpenRouter](#) は、各 API レスポンスに `Host`¹ と `Speed`² メトリクスを含めており、これらの重要なメトリクスの追跡が容易になっています。
2. **プロファイリングツール:** 個々のワークフローコンポーネントと AI 操作のパフォーマンスを分析するために、プロファイリングツールを活用します。プロファイリングツールは、パフォーマンスのホットスポット、非効率なコードパス、リソースを多く消費する操

¹Host は、モデルホストからストリーミング生成の最初のバイトを受信するまでの時間、つまり「初期バイト到達時間」を示します。

²Speed は、生成トークン数を総生成時間で割って計算されます。ストリーミングされないリクエストの場合、レイテンシーは生成時間の一部とみなされます。

作の特定に役立ちます。一般的なプロファイリングツールには、New Relic、Scout、またはプログラミング言語やフレームワークが提供する組み込みプロファイラーがあります。

3. 負荷テスト: 異なるレベルの同時実行ワークロード下でシステムのパフォーマンスを評価するために、負荷テストを実施します。負荷テストは、システムのスケーラビリティの限界を特定し、パフォーマンスの低下を検出し、パフォーマンスを損なうことなく予想されるトラフィックを処理できることを確認するのに役立ちます。

4. 継続的モニタリング: パフォーマンスの問題やボトルネックを事前に検出するために、継続的なモニタリングとアラートの仕組みを実装します。主要業績評価指標 (KPI) を追跡し、事前に定義された閾値を超えた場合に通知を受け取るように、モニタリングダッシュボードとアラートを設定します。これにより、パフォーマンスの問題を迅速に特定し解決することが可能になります。

スケーリング戦略

増加するワークロードに対応し、インテリジェントワークフローオーケストレーションシステムのスケーラビリティを確保するために、以下のスケーリング戦略を検討してください:

1. 垂直スケーリング: 垂直スケーリングは、より高いワークロードを処理するために個々のノードやサーバーのリソース (CPU、メモリなど) を増やすことを指します。この方法は、システムが複雑なワークフローや AI 処理により多くの処理能力やメモリを必要とする場合に適しています。

2. 水平スケーリング: 水平スケーリングは、ワークロードを分散させるためにシステムにノードやサーバーを追加することを指します。この方法は、システムが多数の同時実行ワークフローを処理する必要がある場合や、ワークロードを複数のノードに簡単に分散できる場合に効果的です。水平スケーリングには、トラフィックを均等に分散させるための分散アーキテクチャとロードバランシングメカニズムが必要です。

3. オートスケーリング: ワークロードの需要に基づいてノードやリソースの数を自動的に調整するオートスケーリングメカニズムを実装します。オートスケーリングにより、

システムは入力トラフィックに応じて動的にスケールアップまたはダウンすることができます。Amazon Web Services (AWS) や Google Cloud Platform (GCP) などのクラウドプラットフォームは、インテリジェントワークフローオーケストレーションシステムに活用できるオートスケーリング機能を提供しています。

パフォーマンス最適化テクニック

スケーリング戦略に加えて、インテリジェントワークフローオーケストレーションシステムの効率を向上させるために、以下のパフォーマンス最適化テクニックを検討してください：

- 1. 効率的なデータストレージと取得:** ワークフローコンポーネントで使用されるデータストレージと取得メカニズムを最適化します。効率的なデータベースインデックス、クエリ最適化テクニック、データキャッシングを使用して、データ集約型操作のレイテンシーを最小限に抑え、パフォーマンスを向上させます。
- 2. 非同期 I/O:** ブロッキングを防ぎ、システムの応答性を向上させるために非同期 I/O 操作を活用します。非同期 I/O により、システムは I/O 操作の完了を待たずに複数のリクエストを同時に処理できるため、リソース利用率を最大化できます。
- 3. 効率的なシリアライゼーションとデシリアライゼーション:** ワークフローコンポーネント間のデータ交換に使用されるシリアライゼーションとデシリアライゼーションプロセスを最適化します。Protocol Buffers や MessagePack などの効率的なシリアライゼーション形式を使用して、データシリアライゼーションのオーバーヘッドを削減し、コンポーネント間通信のパフォーマンスを向上させます。



Ruby ベースのアプリケーションの場合、[Universal ID](#)の使用を検討してください。Universal ID は、MessagePack と Brotli (速度と最高クラスのデータ圧縮のために構築された組み合わせ) の両方を活用します。これらのライブラリを組み合わせると、Protocol Buffers と比較して最大 30% 高速で、圧縮率は 2-5% の範囲内です。

4. 圧縮とエンコーディング: ワークフローコンポーネント間で転送されるデータのサイズを削減するために、圧縮とエンコーディングテクニックを適用します。gzip や Brotli などの圧縮アルゴリズムを使用することで、ネットワーク帯域幅の使用量を大幅に削減し、システム全体のパフォーマンスを向上させることができます。

インテリジェントワークフローオーケストレーションシステムの設計と実装時にスケーラビリティとパフォーマンスの側面を考慮することで、システムが多数の同時実行ワークフローを処理し、AI パワードコンポーネントのパフォーマンスを最適化し、増大する需要に応じてシームレスにスケールできることを確保できます。継続的なモニタリング、プロファイリング、最適化の取り組みは、ワークロードと複雑さが時間とともに増加する中で、システムのパフォーマンスと応答性を維持するために不可欠です。

ワークフローのテストと検証

テストと検証は、インテリジェントワークフローオーケストレーションシステムの開発と保守における重要な側面です。AI パワードワークフローの複雑な性質を考慮すると、各コンポーネントが期待通りに機能し、全体的なワークフローが正しく動作し、AI 判断が正確で信頼できることを確認することが不可欠です。このセクションでは、インテリジェントワークフローをテストおよび検証するためのさまざまなテクニックと考慮事項について説明します。

ワークフローコンポーネントのユニットテスト

ユニットテストでは、個々のワークフローコンポーネントを分離して検証し、その正確性と堅牢性を確認します。AI を活用したワークフローコンポーネントのユニットテストを行う際は、以下の点を考慮してください：

- 1. 入力検証:** コンポーネントが様々な種類の入力 (有効なデータと無効なデータの両方) を適切に処理できることをテストします。エッジケースを適切に処理し、適切なエラーメッセージや例外を提供することを確認します。

2. 出力検証: 特定の入力セットに対して、コンポーネントが期待される出力を生成することを確認します。実際の出力を期待される結果と比較して、正確性を確保します。

3. エラー処理: 無効な入力、リソースの利用不可、予期しない例外など、様々なエラーシナリオをシミュレートしてコンポーネントのエラー処理メカニズムをテストします。コンポーネントがエラーを適切にキャッチして処理することを確認します。

4. 境界条件: 空の入力、最大入力サイズ、極端な値など、境界条件でのコンポーネントの動作をテストします。コンポーネントがクラッシュしたり不正な結果を生成したりすることなく、これらの条件を適切に処理することを確認します。

以下は、RSpec テストフレームワークを使用した Ruby でのワークフローコンポーネントのユニットテスト

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

この例では、`OrderValidator` コンポーネントを 2 つのテストケースを使用してテストしています: 有効な注文と無効な注文のケースです。これらのテストケースは、`validate` メソッドが注文の妥当性に基づいて期待される真偽値を返すことを検証します。

ワークフロー相互作用の統合テスト

統合テストは、異なるワークフローコンポーネント間の相互作用とデータフローを検証することに重点を置きます。これにより、コンポーネントが円滑に連携し、期待される結果を生成することを確認します。インテリジェントワークフローの統合テストを行う際は、以下の点を考慮してください:

- 1. コンポーネントの相互作用:** ワークフローコンポーネント間のコミュニケーションとデータ交換をテストします。あるコンポーネントの出力が、ワークフロー内の次のコンポーネントへの入力として正しく渡されることを検証します。
- 2. データ整合性:** ワークフロー全体を通じてデータが一貫性を保ち、正確であることを確認します。データ変換、計算、集計が正しく実行されることを検証します。
- 3. 例外伝播:** ワークフローコンポーネント間での例外とエラーの伝播と処理方法をテストします。例外が適切にキャッチされ、ログに記録され、ワークフローの中断を防ぐために適切に処理されることを検証します。
- 4. 非同期動作:** ワークフローに非同期コンポーネントや並列実行が含まれる場合、調整と同期のメカニズムをテストします。並行処理や非同期シナリオにおいてワークフローが正しく動作することを確認します。

以下は、RSpec テストフレームワークを使用した Ruby でのワークフローの統合テストの例です:


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

この例では、OrderProcessingWorkflow のテストを、異なるワークフロー・コンポーネント間の相互作用を検証することで行っています。このテストケースでは、各コンポーネントの動作に対する期待値を設定し、ワークフローが注文を正常に処理し、それに応じて注文状態を更新することを確認します。

AI 判断ポイントのテスト

AI 判断ポイントのテストは、AI を活用したワークフローの精度と信頼性を確保するために極めて重要です。AI 判断ポイントをテストする際は、以下の点を考慮してください：

1. 判断の正確性: AI コンポーネントが入力データと学習済みモデルに基づいて正確な判断を下すことを検証します。AI 判断を期待される結果や正解データと比較します。

2. エッジケース: エッジケースや異常なシナリオでの AI コンポーネントの動作をテストします。AI コンポーネントがこれらのケースを適切に処理し、合理的な判断を下すことを確認します。

3. バイアスと公平性: AI コンポーネントに潜在的なバイアスがないかを評価し、公平で偏りのない判断を下すことを確認します。多様な入力データでコンポーネントをテストし、差別的なパターンがないか分析します。

4. 説明可能性: AI コンポーネントが判断の説明や根拠を提供する場合、その説明の正確性と明確性を検証します。説明が基礎となる意思決定プロセスと一致していることを確認します。

以下は、RSpec テストフレームワークを使用して Ruby で AI 判断ポイントをテストする例です:

```
1  RSpec.describe FraudDetector do
2    describe '#detect_fraud' do
3      context 'when transaction is fraudulent' do
4        let(:tx) do
5          build(:transaction, amount: 10_000, location: 'High-Risk Country')
6        end
7
8        it 'returns true' do
9          expect(subject.detect_fraud(tx)).to be true
10        end
11      end
12
13    context 'when transaction is legitimate' do
```

```
14     let(:tx) do
15       build(:transaction, amount: 100, location: 'Low-Risk Country')
16     end
17
18     it 'returns false' do
19       expect(subject.detect_fraud(tx)).to be false
20     end
21   end
22 end
23 end
```

この例では、FraudDetector AI コンポーネントを 2 つのテストケースでテストしています: 1 つは不正な取引用、もう 1 つは正当な取引用です。これらのテストケースは、取引の特性に基づいて detect_fraud メソッドが期待される真偽値を返すことを検証します。

エンドツーエンドテスト

エンドツーエンドテストは、実際の環境やユーザー操作を模倣しながら、開始から終了までの全体的なワークフローをテストすることです。これにより、ワークフローが正しく動作し、望ましい結果を生成することを確認します。インテリジェントワークフローのエンドツーエンドテストを実施する際は、以下の点を考慮してください:

- 1. ユーザーシナリオ:** 一般的なユーザーシナリオを特定し、それらのシナリオにおけるワークフローの動作をテストします。ワークフローがユーザー入力を正しく処理し、適切な判断を行い、期待される出力を生成することを確認します。
- 2. データバリデーション:** ワークフローがユーザー入力を検証し、サニタイズして、データの不整合やセキュリティの脆弱性を防ぐことを確認します。有効なデータと無効なデータを含む、さまざまな種類の入力データでワークフローをテストします。

3. エラー回復: ワークフローのエラーや例外からの回復能力をテストします。エラーシナリオをシミュレートし、ワークフローがそれらを適切に処理し、エラーをログに記録し、適切な回復アクションを実行することを確認します。

4. パフォーマンスとスケーラビリティ: 異なる負荷条件下でのワークフローのパフォーマンスとスケーラビリティを評価します。大量の同時リクエストでワークフローをテストし、応答時間、リソース使用率、システム全体の安定性を測定します。

以下は、Rubyで RSpec テストフレームワークとユーザー操作をシミュレートするための Capybara ライブを使用したワークフローのエンドツーエンドテストの例です:

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

この例では、エンドツーエンドテストがウェブインターフェースを通じてユーザーが注文を行う過程をシミュレートします。必要なフォームフィールドに入力し、注文を送信し、注文が正常に処理され、適切な確認メッセージが表示され、データベース内の注文状態が更新されることを検証します。

継続的インテグレーションとデプロイメント

インテリジェントワークフローの信頼性と保守性を確保するために、継続的インテグレーション・デプロイメント(CI/CD)パイプラインにテストと検証を統合することが推奨されます。これにより、ワークフローの変更を本番環境にデプロイする前に、自動的にテストと検証が可能になります。以下のプラクティスを考慮してください：

- 1. 自動テスト実行：**ワークフローのコードベースに変更が加えられるたびに、テストスイートを自動的に実行するように CI/CD パイプラインを設定します。これにより、開発プロセスの早い段階でリグレッションや障害を検出することができます。
- 2. テストカバレッジの監視：**ワークフローコンポーネントと AI 判断ポイントのテストカバレッジを測定し監視します。重要なパスとシナリオが徹底的にテストされるよう、高いテストカバレッジを目指します。
- 3. 継続的フィードバック：**テスト結果とコード品質メトリクスを開発ワークフローに統合します。テストの状態、コード品質、CI/CD プロセス中に検出された問題について、開発者に継続的なフィードバックを提供します。
- 4. ステージング環境：**本番環境を密接に模したステージング環境にワークフローをデプロイします。ステージング環境で追加のテストと検証を実施し、インフラストラクチャ、設定、またはデータ統合に関連する問題を発見します。
- 5. ロールバックメカニズム：**デプロイメントの失敗や本番環境で検出された重大な問題に備えて、ロールバックメカニズムを実装します。ダウンタイムやユーザーへの影響を最小限に抑えるため、ワークフローを以前の安定バージョンに迅速に戻せるようにします。

インテリジェントワークフローの開発ライフサイクル全体にテストと検証を組み込むことで、組織は AI を活用したシステムの信頼性、正確性、保守性を確保することができます。定期的なテストと検証は、バグの発見、リグレッションの防止、そしてワークフローの動作と結果に対する信頼性の構築に役立ちます。

パート 2 : パターン集

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

プロンプトエンジニアリング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

チェーン・オブ・ソート

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

コンテンツ生成

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

構造化エンティティの作成

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

LLM エージェントのガイダンス

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

利点と考慮事項

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

モードスイッチ

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

使用するタイミング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

役割割り当て

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

使用するタイミング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

プロンプトオブジェクト

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

プロンプトテンプレート

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

メリットと考慮事項

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

使用すべき場面：

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

ストラクチャード IO

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

構造化 IO のスケーリング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

メリットと考慮事項

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

プロンプトチェーニング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

使用するタイミング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例 : **Olympia** のオンボーディング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

プロンプトリライター

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

レスポンスフェンシング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

利点と考慮事項

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

エラー処理

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

クエリアナライザー

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

実装

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

品詞タグ付け（**POS Tagging**）と固有表現抽出（**NER**）

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

意図分類

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

キーワード抽出

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

利点

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

クエリリライター

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

利点

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

ベントリロキスト

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

使用するタイミング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

実装例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

個別コンポーネント

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

述語

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

使用するタイミング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

API ファサード

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

主なメリット

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

使用すべき場面

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

認証と承認

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

リクエスト処理

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

レスポンスのフォーマット

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

エラー処理とエッジケース

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

スケーラビリティとパフォーマンスの考慮事項

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

他のデザインパターンとの比較

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

結果インタープリター

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

使用すべき場面

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仮想マシン

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

使用するタイミング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

魔法の裏側

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕様とテスト

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕様化

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

テストケースの作成

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例：翻訳コンポーネントのテスト

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

HTTP 通信の再生

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

ヒューマン・イン・ザ・ループ (HITL)

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

高レベルパターン

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

ハイブリッドインテリジェンス

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

適応的応答

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

人間と **AI** の役割切り替え

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

エスカレーション

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

主なメリット

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

実世界での応用：ヘルスケア

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

フィードバックループ

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

応用例と実例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

人間のフィードバック統合における高度な技術

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

パッシブ情報放射

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

文脈的情報表示

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

先制的通知

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

説明的洞察

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

インタラクティブな探索

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

主要なメリット

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

応用と例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

協調的意思決定 (CDM)

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

継続的学習

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

応用と例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

倫理的考慮事項

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

AI リスクの軽減における HITL の役割

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

技術的進歩と将来の展望

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

HITL システムの課題と限界

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

インテリジェントなエラー処理

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

従来のエラー処理アプローチ

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

文脈的エラー診断

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

文脈的エラー診断のためのプロンプトエンジニアリング

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

文脈的エラー診断のための検索拡張生成

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

知的エラー報告

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

予測的エラー防止

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

スマートエラーリカバリー

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

パーソナライズされたエラーコミュニケーション

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

適応型エラー処理ワークフロー

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

品質管理

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

Eval

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

問題

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

解決策

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

考慮事項

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

ゴールデンリファレンスの理解

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

リファレンスフリー評価の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

ガードレール

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

問題

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

解決策

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

動作の仕組み

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

例

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

考慮事項

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

ガードレールと評価：コインの表と裏

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

ガードレールとリファレンスフリー評価の互換性

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

二重目的のガードレールと評価の実装

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

用語集

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

用語集

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

A

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

B

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

C

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

D

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

E

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

F

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

G

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

H

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

I

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

J

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

K

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

L

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

M

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

N

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

O

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

P

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

Q

このコンテンツはサンプル本では読めません。この本は Leanpub で購入できます <http://leanpub.com/patterns-of-application-development-using-ai-ja>.

R

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

S

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

T

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

U

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

V

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

W

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

Z

このコンテンツはサンプル本では読めません。この本は Leanpub で購入
できます [http://leanpub.com/patterns-of-application-development-
using-ai-ja](http://leanpub.com/patterns-of-application-development-using-ai-ja).

Index

- ACID 特性, 106
- AI, 61, 69, 96, 126, 132, 141, 148, 197, 205
 - conversational, 206
 - model, 96
 - アプリケーション, 122, 136, 147
 - モデル, 85, 155, 157, 205
 - 会話, 6
 - 判断ポイント, 253
 - 対話型, 28
 - 複合システム, 27, 28, 31
- AI ワーカーのチェーン化, 108
- AI! モデル, 153
- Alpaca, 12
- Altman, Sam, 16
- Amazon Web Services, 249
- Anthropic, 20, 36, 68, 126, 135
- API, 67, 120, 152
- BERT, 12, 22
- Brotli, 249, 250
- Capybara ライブラリ, 256
- Chain of Thought (CoT), 137
- ChatGPT, 27, 49
- classification, 117
- Claude, 7, 40, 73
- Claude 3, 45, 123, 126, 132, 135
- Claude 3 Opus, 69
- Claude v1, 15
- Claude v2, 15
- Cohere (LLM Provider), 22
- Cohere (LLM プロバイダー), 20
- command line
 - Command-Line Interface (CLI), 23
- content
 - filtering, 24
- context
 - infinitely long inputs, 14
 - window, 14
 - コンテキスト型コンテンツ生成, 188, 189
- Customer Sentiment Analysis, 97
- customization, 24
- C (プログラミング言語), 113
- data
 - privacy, 24
- Databricks の従業員, 48
- Datadog, 245
- decision
 - making capabilities, 96
- digital landscape, 189
- document clustering, 117
- Dohan, et al., 40

- ELK スタック, 107
- errors
 - Intelligent Error Handling, 141
- experimentation
 - フレームワーク, 189
- E コマース, 216
- e コマース, 187
- E コマースアプリケーション, 88
- F#, 89
- Facebook, 22
- finalize メソッド, 156, 157
- FitAI, 206
- Gemma 7B, 10
- Generative Pre-trained Transformer (GPT),
7
- Generative UI (GenUI), 208
- GitLab, 89
- Google, 20
 - API, 58, 61
 - Cloud AI プラットフォーム, 22
 - Cloud Platform, 249
 - Gemini, 19
 - Gemini 1.5 Pro, 13, 15, 17
 - PaLM (Pathways Language Model), 15
 - PaLM (Pathways Language Model),
22
 - T5, 12
- GPT-3, 12, 15
- GPT-4, 6, 12, 15, 19, 28, 40, 45, 58, 101, 114,
117, 124, 131, 199, 200, 247
- Graham, Paul, 17
- GraphQL, 105
- Groq, 23, 117
- gzip, 250
- high-performance completion, 24
- Hohpe, Gregor, 101
- Honeybadger, 90
- HTTP, 148
- instruction tuning, 9
- iterative refinement, 142
- JSON (JavaScript Object Notation), 128, 129
- JSON (JavaScript Object Notation), 123,
132, 145, 164
- K 平均法, 118
- language
 - models, 39, 61
- Large Language Model (LLM), 14, 16, 117,
142
- Llama, 12
- Llama 2-70B, 46
- Llama 3 70B, 10
- Llama 3 8B, 10
- LLM の統合, 184
- Louvre, 39
- Managed Streaming for Apache Kafka, 38
- Memorial Sloan Kettering Cancer Center,
38
- MessagePack, 249
- Meta, 22

- Metropolitan Museum of Art, 39
- Mistral, 23
 - 7B, 10
 - 7B Instruct, 15, 200
- Mixtral
 - 8x22B, 10
 - 8x7B, 52
- motivational strategies, 208
- Naive Bayes, 117
- natural language
 - Natural Language Processing (NLP), 117
- New Relic, 248
- Ollama, 23
- Olympia, 30, 58, 126, 141, 149, 165
- Olympia のナレッジベース, 88
- OpenAI, 3, 20, 36, 68
- OpenRouter, 25, 26, 149, 247
- OPT model, 22
- Perplexity(プロバイダー), 10
- prompts
 - engineering, 61
- Protocol Buffers, 249
- PyTorch, 22
- Qwen2 70B, 10
- Rails, 190
- Railway Oriented Programming (ROP), 91
- Raix, 224
 - ライブラリ, 94
- RSpec, 251, 252, 256
- Ruby, 89, 90, 110, 161, 256
- Ruby on Rails, 1, 108, 224, 232
- Rudall, Alex, 21
- Rust(プログラミング言語), 89
- Rust(プログラミング言語), 113
- Scout, 248
- sentiment analysis, 15
- server-sent events (SSE), 148
- SQL インジェクション, 66
- stream handlers, 149
- stream processing, 148
- Stripe, 126
- Support Vector Machines (SVM), 117
- system directive, 96
- T5, 22
- Together.ai, 23
- Top-k サンプリング, 44
- Top-p(nucleus)サンプリング, 44
- topic identification, 117
- translation, 15
- Unicode でエンコード可能な言語, 13
- Universal ID, 249
- user experience, 190
- Wall, Larry, 3
- Wisper, 91, 103, 149, 157
- Wooley, Chad, 89
- XML, 132
- Yi-34B, 46

- アカウント, 87
- アクセシビリティ, 211, 212
- アダプティブ UI, 203
- アプリケーション設計とフレームワーク,
193
- アンサンプル, 114, 115
 - ワーカーのアンサンプル, 115
- イベント駆動アーキテクチャ, 105
- インテリジェントコンテンツモデレーター,
228
- インテリジェントワークフローオーケストレーション,
215, 247
- インテリジェントワークフローオーケストレーション,
223, 250
- エコシステム, 145
- エッジケース, 54
- エラー
 - 処理, 104, 107, 140, 251
 - 回復, 256
 - 率, 107
- エンタープライズアプリケーションアーキ
テクチャ, 35
- エンタープライズインテグレーションパタ
ーン, 101
- エンドツーエンドテスト, 255, 256
- エージェント型, 29
- オンライン小売業者, 201
- オートスケーリング, 248
- オープンソースモデルホスティングプロバ
イダー, 201
- カスタマーサポート, 29
- カスタマーサービスチャットボット, 30
- キャッシング, 247
- クリエイティブライティング, 31, 48
- クロスモダル生成, 20
- クローズドおよびオープンな質問応答, 48
- グラフィカルモデル, 40
- グローバルインタブリタロック(GIL), 112
- コモンズの悲劇, 187
- コンテキスト
 - ウィンドウ, 220
 - コンテキストに基づく意思決定, 220
 - コンテキスト依存型コンテンツ生成,
194, 195
 - コンテキスト型コンテンツ生成, 183,
187, 188
 - 拡張, 42
 - 文脈に応じたフィールド候補, 195
- コンテンツ
 - コンテンツカテゴリ分類, 108
- コンテンツベースフィルタリング, 88
- コンピュータサイエンス, 65, 68
- サプライチェーン
 - 最適化, 30
- サーキットブレーカーロジック, 160
- システムディレクティブ, 125
- スケーラビリティ, 217, 245
- ステージング環境, 257
- ステートレス, 155
- ストリーミングデータ, 150
- ストリーム処理, 154
 - ロジック, 157
- スマートフォン, 212
- スループット, 25

- セグメンテーションとターゲティング戦略,
 - 190
- ゼロショット学習, 54, 55
- ソフトウェアアーキテクチャ, 2
- タブレット, 212
- チケットの割り当て, 236
- チャットボットアプリケーション, 115
- ツールの使用, 120, 147
- ツールコール, 151
- テキストクリーンアップ, 108
- デスクトップコンピュータ, 212
- デバッグ, 219
 - とテスト, 130
 - とトラブルシューティング, 244
- データ
 - データバリデーション, 255
 - データ取得, 106
 - データ同期, 106
 - フロー, 107
 - プライバシー, 210
 - 処理タスク, 122
 - 処理パイプライン, 236
 - 分析, 31, 145
 - 整合性, 236
 - 永続化, 106
 - 準備, 106
- データベース, 120
 - バックドオブジェクト, 102
 - ロック戦略, 106
- トランスフォーマーアーキテクチャ, 5
- トリガーメッセージ, 101
- トークン, 5, 11
- トークン化, 11
- ナラティブ構築, 18
- ナレッジベース, 6
- ナレッジマネジメント, 29
- ニューラルネットワーク, 3, 6
- ネットワーク接続, 221
- ハイパーパラメータ, 43
- ハッシュ, 150
- ハードウェア, 26
- バイアス
 - AI におけるバイアスと公平性, 254
- バイトペアエンコーディング(BPE), 12, 13
- バッチ処理, 247
- バーチャルアシスタント, 30
- パスの絞り込み, 35
- パスを狭める, 36
- パターンマッチング, 151
- パフォーマンス
 - トレードオフ, 4
 - 問題, 248
 - 最適化, 130, 192, 244
- パブリッシュ-サブスクライブシステム, 105
- パラメータ
 - パラメータ数, 25
 - 効果, 126
 - 範囲, 10
- パーソナライズ
 - パーソナライズされたマイクロコピー,
 - 202
- パーソナライズされた商品レコメンデーション, 88
- パーソナライゼーション, 184, 212, 217

- パーソナライズされたフォーム, 195
- ヒューマンインザループ (HITL), 176
- ビジネスルール, 215
- ビジュアルインターフェース, 204
- ファイナライズメソッド, 154
- ファインチューニング, 75
- フィードバック
 - フィードバックループ, 54
- フォールバック戦略, 107
- フューショット
 - プロンプティング, 58
 - 学習, 57
- プログレッシブ・デイスクロージャヤ, 203
- プロセスマネージャ, 101, 104
 - エンタープライズ統合, 224
- プロンプト
 - エンジニアリング, 37, 41, 42, 52, 55, 63, 209
 - チェーニング, 55, 67
 - プロンプトオブジェクト, 69
 - プロンプトテンプレート, 55, 201
 - プロンプト蒸留, 42, 68, 74, 247
 - 改良, 64
 - 設計, 54, 64
- ベントリロキスト, 174
- ベースモデル, 50
- ボトルネック, 220
- マイクロサービスアーキテクチャ, 86
- マルチエージェント
 - 問題解決システム, 28
- マルチステップワークフロー, 108
- マルチモーダル
 - モデル, 18
 - 言語モデル, 19
- マーキュリー (ローマ神話), 41
- マークアップ形式のタグ付け, 66
- マークダウン, 145
- モジュール性, 85
- モニタリング
 - とアラート, 221
 - とロギング, 107
 - とログ記録, 244
 - メトリクス, 244
- ユーザーの信頼, 211
- ユーザーインターフェース (UI)
 - インターフェース, 193, 208
 - テクノロジー, 204
 - デザイン, 213
 - フレームワーク, 209
- ユーザーテストとフィードバック, 192
- ユーザー心理学, 209
- ユーザー生成コンテンツ, 108
- ランカー, 32
- リスク層別化, 99
- リスク要因, 92, 93
- リトライメカニズム, 107
- レイテンシー, 25
- レスポンスフェンシング, 174, 201
- ログの保持とローテーション, 245
- ローカル開発環境, 153
- ロールバックメカニズム, 257
- ロールプレイスタイルのやり取り, 6
- ワンショット学習, 56

- 一貫性
 - と再現性, 130
- 不正検知
 - システム, 94
- 並列実行, 246
- 主要指標の追跡, 241
- 予測, 5
- 交通管理, 30
- 会話
 - トランスクリプト, 155, 158
 - ループ, 158
- 使いやすさの問題, 210
- 例外処理, 220, 223
- 保険確認, 98
- 倫理
 - 影響, 194
- 入力
 - プロンプト, 52
 - 検証, 250
- 入力パラメータ, 125
- 処理時間, 107
- 出力検証, 251
- 分散アーキテクチャ, 246
- 分類, 48
- 効率性, 217
- 動的 UI 生成, 184
- 動的タスクルーティング, 218
- 動的ツール選択, 129
- 包括的なインターフェース, 194
- 医学的発見, 98
- 協調フィルタリング, 88
- 反復的な改良, 71
- 合成データ生成, 49
- 同時実行ワークフロー, 250
- 商品レコメンデーション, 88
- 国際化, 190
- 境界条件, 251
- 外部サービスや API, 123
- 多数のワーカー, 115, 164
- 多数決方式, 114
- 大規模言語モデル (LLM), 199
- 大規模言語モデル (LLM), 1, 3, 27, 63, 64, 67, 71, 73, 84, 107, 120, 122, 132, 139, 142, 145, 162, 164, 183, 193, 204, 227
 - 分野, 25
- 存在ペナルティ, 44
- 学習データ, 39
- 強制ツール選択, 129
- 心の理論, 37
- 思考連鎖 (CoT), 41
- 悲観的ロック, 106
- 情報
 - 抽出, 48
 - 検索, 6, 123
- 意思決定のユースケース, 131
- 感情分析, 98, 108, 109, 111, 114, 115, 132, 143
- 感情的なトーン, 143
- 手動介入, 223
- 拡張現実メガネ, 212
- 指示チューニング

- 指示チューニングされたモデル, 45
- 指示調整
 - 指示調整モデル, 48
- 推論, 5
- 擬人化, 64
- 教師なし学習, 4
- 教育アプリケーション, 29
- 文法規則, 4
- 最初のトークンまでの時間(TTFT), 25
- 最小権限の原則, 67
- 柔軟性と創造性, 191
- 検索ベースモデル, 6
- 検索拡張生成(RAG), 29, 35, 42, 75, 122
- 楽観的ロック, 106
- 概念のおよび実践的な課題, 194
- 構文エラー, 129
- 構造化 IO, 201
- 構造化データ, 131
- 構造化ログ, 245
- 水星, 41
- 水銀(元素), 41
- 決定
 - ポイント, 242
 - 木, 216
- 決定論的な振る舞い, 54
- 温度パラメータ, 50
- 潜在的ディリクレ配分法, 118
- 潜在空間, 37, 39
- 現代のアプリケーション, 217
- 生成事前学習済みトランスフォーマー
 - (GPT), 63
- 生成的 UI(GenUI), 193, 201, 212
- 生産性, 186
- 病歴収集, 98
- 症状評価と層別化, 98
- 監査とコンプライアンス, 244
- 監査ログ, 103
- 確率モデル, 39
- 結果インタープリター, 140
- 統合テスト, 252
- 継続的インテグレーション・デプロイメント
 - (CI/CD), 257
 - パイプライン, 257
- 継続的リスクモニタリング, 100
- 緊急時対応計画, 30
- 線形代数, 39
- 線形回帰, 40
- 繰り返しペナルティ, 47
- 翻訳, 191
- 臨床意思決定支援, 100
- 自動継続, 158
- 自己修復データ, 162, 241
- 自己回帰モデリング, 39
- 自然言語
 - 自然言語処理(NLP), 98
- 蒸留プロセス, 71
- 複雑なタスク, 144
- 要約, 48
- 言い換え, 49
- 言語
 - モデル, 68
 - 言語検出, 108

- 関連タスク, 4
- 詳細なログ記録, 245
- 説明可能性, 254
- 質問応答システム, 6
- 辞書, 128
- 過去のパターン, 220
- 適応型ワークフロー
 - 適応型ワークフロー構成, 220
- 配列, 128
- 重要なパターン, 217
- 量子化, 26
- 開発フレームワーク, 146
- 関数
 - 名, 152
 - 呼び出し, 120
 - 呼び出し履歴, 154
- 関数呼び出しの失敗, 131
- 関数型プログラミング, 88
- 非同期処理, 246
- 音声制御インターフェース, 30