Agents

AI エージェントの設計と実装

Julia Wiesinger, Patrick Marlow
and Vladimir Vuskovic 著
Kuma Arakawa 訳

Google

Acknowledgements

Content contributors

Evan Huang

Emily Xue

Olcan Sercinoglu

Sebastian Riedel

Satinder Baveja

Antonio Gulli

Anant Nawalgaria

Curators and Editors

Antonio Gulli

Anant Nawalgaria

Grace Mollison

Technical Writer

Joey Haymaker

Designer

Michael Lanning

Table of contents

はじめに ····· P4
エージェントとは何か？ ····· P5
モデル ····· P6
ツール ····· P7
オーケストレーション層 ····· P7
エージェント vs. モデル ····· P6
認知アーキテクチャ：エージェントの動作原理 ····· P8
ツール：外部世界への鍵 ····· P11
拡張機能 ····· P11
サンプル拡張機能 ····· P14
Function ····· P16
ユースケース ····· P18
Function サンプルコード ····· P22
データストア ····· P22
実装と応用 ····· P24
ツールのまとめ ····· P27
ターゲット学習によるモデル性能の向上 ····· P28
LangChain によるエージェントクイックスタート ····· P30
Vertex AI エージェントによる本番アプリケーション ····· P33
まとめ ····· P35
巻末注 ····· P36

推論、論理、そして外部情報へのアクセス——これらすべてが生成 AI モデルに接続されているこの組み合わせが、エージェントという概念を呼び起こすのです。

はじめに

人間は、複雑なパターン認識タスクが非常に得意です。しかし、結論に至る前に、書籍、Google 検索、電卓といったツールに頼って、既存の知識を補うことがよくあります。人間と同様に、生成 AI モデルも、リアルタイム情報へのアクセスや実世界でのアクションの提案のために、ツールを使用するように訓練することができます。例えば、あるモデルはデータベース検索ツールを活用して顧客の購入履歴といった特定の情報にアクセスし、それに基づいてパーソナライズされた買い物推奨を生成することができます。あるいは、ユーザーの問い合わせに応じて、モデルが様々な API 呼び出しを行い、同僚へのメール返信や、あなたに代わっての金融取引の実行といった処理を行うことも可能です。そのためには、モデルは一連の外部ツールにアクセスできるだけでなく、あらゆるタスクを自律的に計画し実行する能力も必要とします。推論、論理、そして外部情報へのアクセス——これらすべてが生成 AI モデルに接続されているこの組み合わせが、エージェント——すなわち生成 AI モデル単独の能力を超えるプログラム——という概念を呼び起こすのです。このホワイトペーパーでは、これらすべての点とそれに関連する側面について、より詳細に掘り下げていきます。

エージェントとは何か?

最も基本的な形では、生成 AI エージェントとは、自由に使えるツールを用いて世界を観察し、それに基づいて行動することで目標を達成しようと試みるアプリケーションとして定義できます。エージェントは自律的であり、特に達成すべき適切な目標や目的が与えられた場合には、人間の介在なしに独立して行動することができます。エージェントはまた、目標達成へのアプローチにおいて積極的に行動することもできます。人間からの明確な指示がない場合でも、エージェントは最終目標を達成するために次に何をすべきかを推論することができます。AI におけるエージェントの概念は非常に一般的かつ強力なものですが、このホワイトペーパーでは、発行時点で生成 AI モデルが構築可能な特定タイプのエージェントに焦点を当てています。

エージェントの内部動作を理解するために、まず、エージェントの振る舞い、行動、意思決定を駆動する基本的な構成要素を紹介します。これらの構成要素の組み合わせは認知アーキテクチャとして記述でき、そのようなアーキテクチャは、これらの構成要素を様々に組み合わせることで多数実現可能です。中核となる機能に焦点を当てると、図 1 に示すように、エージェントの認知アーキテクチャには 3 つの不可欠な構成要素があります。

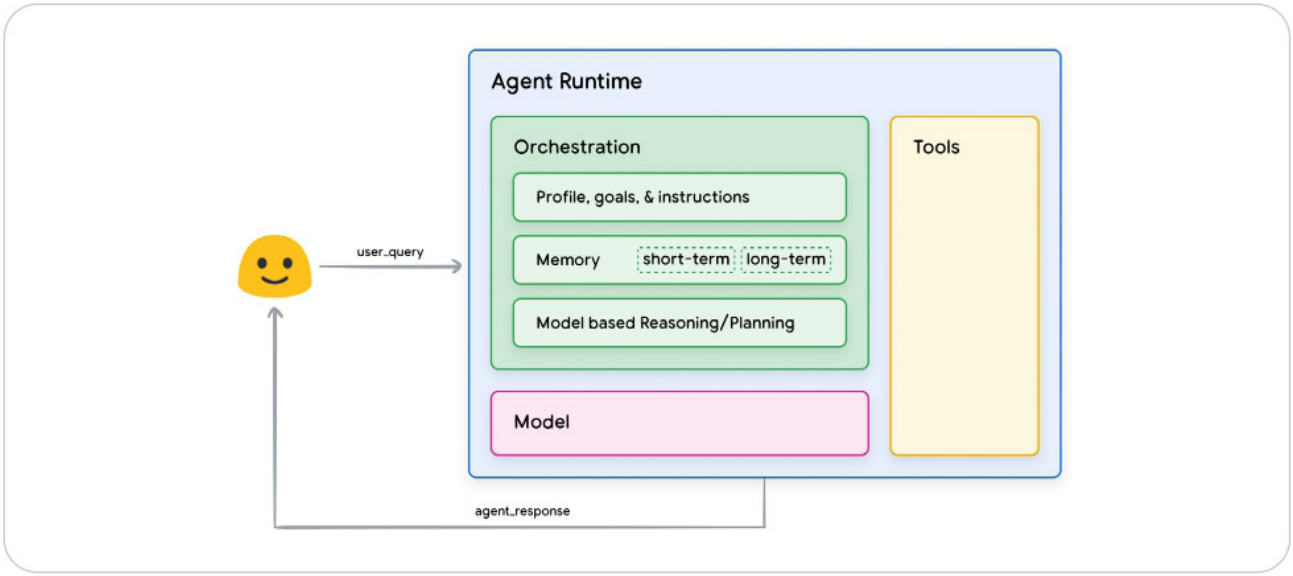

図 1：一般的なエージェントアーキテクチャと構成要素

モデル

エージェントの文脈において、モデルとは、エージェントのプロセスにおける中核的な意思決定者として利用される言語モデル（LM）を指します。エージェントが使用するモデルは、ReAct、Chain-of-Thought、Tree-of-Thoughts のような、指示に基づいた推論および論理フレームワークに従うことができる、あらゆるサイズ（小規模／大規模）の単一または複数の LM であり得ます。モデルは、特定のエージェントアーキテクチャのニーズに応じて、汎用目的、マルチモーダル、あるいはファインチューニングされたものであり得ます。最良の本番結果を得るためには、目的の最終アプリケーションに最も適合し、かつ理想的には、認知アーキテクチャで使用予定のツールに関連するデータシグネチャで訓練されたモデルを活用すべきです。留意すべき重要な点は、モデルは通常、エージェントの特定の設定情報（すなわち、ツールの選択、オーケストレーション／推論の設定）を用いて訓練されるわけではないということです。しかし、エージェントが様々な文脈で特定のツールを使用したり推論ステップを実行したりする事例を含む、エージェントの能力を示す例を提供することで、エージェントのタスクに合わせてモデルをさらに改良することは可能です。

ツール

基盤モデルは、その印象的なテキストおよび画像生成能力にもかかわらず、外部世界とやり取りできないという制約を依然として抱えています。ツールはこのギャップを埋め、エージェントが外部のデータやサービスとやり取りできるようにすると同時に、基盤となるモデル単独の能力を超える、より広範なアクションを可能にします。ツールは様々な形態を取り得、その複雑さの度合いも様々ですが、通常は GET、POST、PATCH、DELETE といった一般的なウェブ API メソッドと連携します。例えば、ツールはデータベース内の顧客情報を更新したり、エージェントがユーザーに提供する旅行の推奨に影響を与えるために気象データを取得したりすることができます。ツールを使うことで、エージェントは実世界の情報にアクセスし処理することができます。これにより、エージェントは検索拡張生成（RAG）のような、より専門的なシステムをサポートできるようになります。これは、基盤モデルが単独で達成できる範囲を超えて、エージェントの能力を大幅に拡張するものです。ツールについては以下でさらに詳しく説明しますが、理解すべき最も重要なことは、ツールがエージェントの内部能力と外部世界との間のギャップを埋め、より広範な可能性を切り拓くということです。

オーケストレーション層

オーケストレーション層は、エージェントが情報を取り込み、何らかの内部推論を実行し、その推論を次の行動や意思決定の判断材料とする方法を規定する、周期的なプロセスを記述します。一般に、このループは、エージェントが目標または停止点に到達するまで継続します。オーケストレーション層の複雑さは、エージェントとそれが実行しているタスクに応じて大きく変わり得ます。ループによっては、決定ルールを伴う単純な計算である場合もあれば、連鎖的な論理を含んだり、追加の機械学習アルゴリズムが関与したり、他の確率的推論技術を実装したりする場合もあります。エージェントのオーケストレーション層の詳細な実装については、認知アーキテクチャのセクションでさらに詳しく説明します。

エージェント vs. モデル

エージェントとモデルの違いをより明確に理解するために、次の表をご覧ください。

モデル	エージェント
知識は、訓練データ内で利用可能なものに限定される。	知識は、ツールを介した外部システムとの接続を通じて拡張される。
ユーザーの問い合わせに基づく単一の推論／予測。モデルに明示的に実装されていない限り、セッション履歴や継続的なコンテキスト（例：チャット履歴）の管理は行われない。	管理されたセッション履歴（例：チャット履歴）により、ユーザーの問い合わせおよびオーケストレーション層での決定に基づいた複数ターンの推論／予測が可能。この文脈において「ターン」とは、対話システムとエージェント間の 1 回の相互作用（例：1 つの入力イベント／問い合わせと 1 つのエージェント応答）として定義される。
ネイティブなツールの実装はない。	ツールはエージェントアーキテクチャにネイティブに実装される。
ネイティブな論理層の実装はない。ユーザーは、単純な質問としてプロンプトを作成したり、推論フレームワーク（CoT、ReAct など）を使用してモデルの予測をガイドするための複雑なプロンプトを作成したりできる。	CoT、ReAct などの推論フレームワーク、または LangChain のような他の構築済みエージェントフレームワークを使用する、ネイティブな認知アーキテクチャ。

認知アーキテクチャ：エージェントの動作原理

忙しい厨房にいるシェフを想像してみてください。彼らの目標は、レストランの常連客のためにおいしい料理を作ることであり、それには計画、実行、調整というある種のサイクルが伴います。

- 常連客の注文や、食料品庫や冷蔵庫にある食材といった情報を集める。
- 集めたばかりの情報に基づいて、どんな料理や風味の組み合わせを作り出せるかについて、何らかの内部推論を行う。
- 野菜を刻み、スパイスを調合し、肉を焼くといった、料理を作るための行動を取る。

プロセスの各段階で、シェフは必要に応じて調整を行い、食材がなくなったり顧客からのフィードバックを受け取ったりするにつれて計画を練り直し、過去の一連の結果を用いて次の行動計画を決定します。この情報収集、計画、実行、調整のサイクルは、シェフが目標を達成するために採用する独自の認知アーキテクチャを記述しています。

シェフとまったく同様に、エージェントも、情報を繰り返し処理し、情報に基づいた意思決定を行い、以前の出力に基づいて次の行動を改良することによって、認知アーキテクチャを使用して最終目標を達成することができます。エージェントの認知アーキテクチャの中核には、記憶、状態、推論、計画の維持を担当するオーケストレーション層があります。それは、急速に進化するプロンプトエンジニアリングの分野と関連フレームワークを使用して推論と計画をガイドし、エージェントが環境とより効果的に相互作用しタスクを完了できるようにします。プロンプトエンジニアリングフレームワークと言語モデルのタスク計画の分野における研究は急速に進化しており、様々な有望なアプローチを生み出しています。これは網羅的なリストではありませんが、本書発行時点で利用可能な、最も人気のあるフレームワークと推論技術のいくつかを以下に示します。

- **ReAct：**言語モデルが文脈内事例の有無にかかわらずユーザーの問い合わせに対して推論し（Reason）行動する（take action）ための思考プロセス戦略を提供するプロンプトエンジニアリングフレームワーク。ReAct プロンプティングは、いくつかの SOTA（最先端）ベースラインを上回り、LLM（大規模言語モデル）の人間との相互運用性と信頼性を向上させることが示されている。

- **Chain-of-Thought（CoT）：**中間ステップを通じて推論能力を可能にするプロンプトエンジニアリングフレームワーク。CoT には、自己整合性、アクティブプロンプト、マルチモーダル CoT など様々なサブ技術があり、それぞれ特定のアプリケーションに応じて長所と短所がある。

- **Tree-of-Thoughts（ToT）：**探索や戦略的な先読みタスクに適したプロンプトエンジニアリングフレームワーク。これは Chain-of-Thought プロンプティングを一般化し、言語モデルによる一般的な問題解決のための中間ステップとして機能する様々な思考の連鎖をモデルが探求することを可能にする。

エージェントは、上記の推論技術のいずれか、または他の多くの技術を利用して、与えられたユーザーリクエストに対する次の最善の行動を選択できます。例えば、ユーザーの問い合わせに対して正しい行動とツールを選択するために ReAct フレームワークを使用するようにプログラムされたエージェントを考えてみましょう。一連のイベントは次のようになるかもしれません。

1. ユーザーがエージェントに問い合わせを送信する。
2. エージェントが ReAct シーケンスを開始する。
3. エージェントはモデルにプロンプトを提供し、次の ReAct ステップのいずれかとそれに対応する出力を生成するように依頼する。
   a. **質問 :** ユーザーの問い合わせからの入力質問（プロンプトと共に提供）。
   b. **思考 :** モデルが次に何をすべきかについての考え。
   c. **行動 :** モデルが次に取るべき行動に関する決定。
      i . ここでツールの選択が行われることがある。
      ii . 例えば、行動は [Flights、Search、Code、None] のいずれかであり得る。最初の 3 つはモデルが選択できる既知のツールを表し、最後は「ツール選択なし」を表す。
   d. **行動入力 :** ツールに提供する入力（もしあれば）に関するモデルの決定。
   e. **観察 :** 行動／行動入力シーケンスの結果。
      i . この思考／行動／行動入力／観察は、必要に応じて N 回繰り返されることがある。
   f. **最終回答 :** 元のユーザーの問い合わせに対して提供するモデルの最終回答。
4.ReAct ループが終了し、最終回答がユーザーに返される。

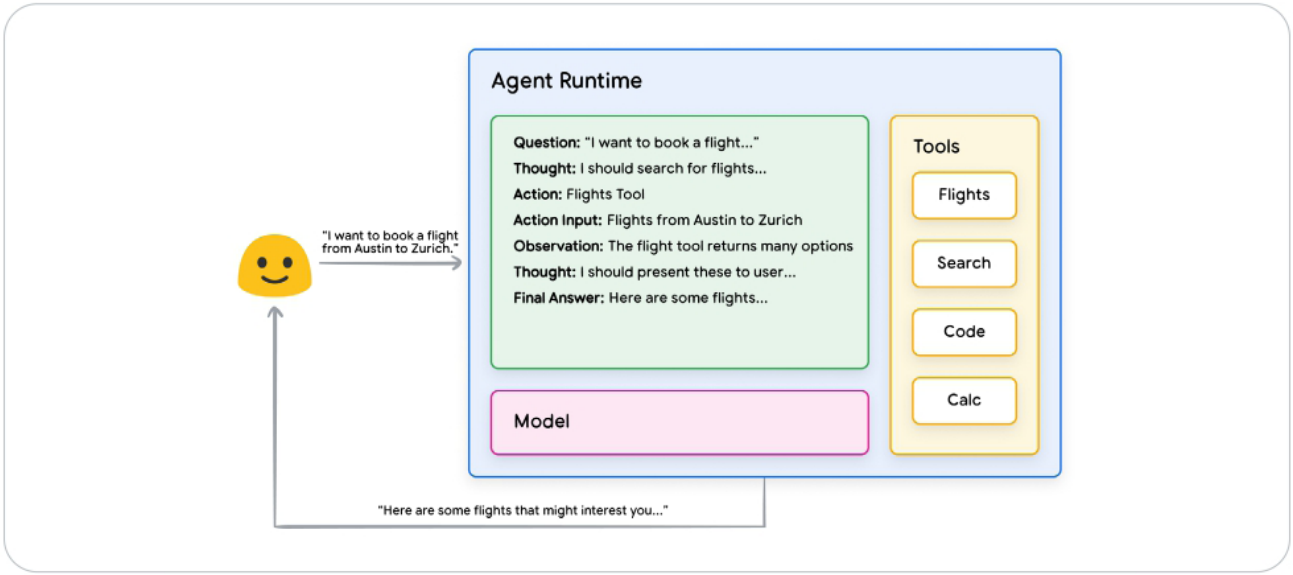

図 2：オーケストレーション層で ReAct 推論を行うエージェントの例

図 2 に示すように、モデル、ツール、およびエージェントの設定が連携して動作し、ユーザーの元の問い合わせに基づいて、根拠のある簡潔な応答をユーザーに返します。モデルは事前の知識に基づいて答えを推測（ハルシネーション）することもできましたが、代わりにツール（Flights）を使用してリアルタイムの外部情報を検索しました。この追加情報がモデルに提供されたことで、モデルは実際の事実データに基づいてより情報に基づいた意思決定を行い、この情報を要約してユーザーに返すことができました。

要約すると、エージェントの応答の品質は、適切なツールを選択する能力やそのツールがどれだけうまく定義されているかを含め、これらの様々なタスクについて推論し行動するモデルの能力に直接結びついています。新鮮な食材で料理を作り顧客のフィードバックに注意を払うシェフのように、エージェントは最適な結果を提供するために、確かな推論と信頼できる情報に依存しています。次のセクションでは、エージェントが新鮮なデータに接続する様々な方法について詳しく見ていきます。

ツール：外部世界への鍵

言語モデルは情報処理に長けていますが、実世界を直接認識し影響を与える能力に欠けています。このことは、外部システムやデータとのインタラクションが必要な状況において、それらの有用性を制限します。これは、ある意味で、言語モデルはその訓練データから学習した内容以上のものにはならない、ということを意味します。しかし、どれだけ多くのデータをモデルに投入したとしても、それらは依然として外部世界とやり取りする基本的な能力に欠けています。では、どうすればモデルが外部システムとリアルタイムでコンテキストを意識したインタラクションを行えるようにできるのでしょうか？ Function、拡張機能、データストア、そしてプラグインはすべて、この重要な能力をモデルに提供するための方法です。

それらは多くの名前で呼ばれていますが、私たちの基盤モデルと外部世界との間の繋がりを作り出すのはツールです。外部システムやデータへのこの繋がりは、私たちのエージェントがより広範なタスクを実行し、かつ、より高い精度と信頼性でそれを行うことを可能にします。例えば、ツールはエージェントがスマートホームの設定を調整したり、カレンダーを更新したり、データベースからユーザー情報を取得したり、特定の指示セットに基づいてメールを送信したりすることを可能にします。

本書発行日現在、Google のモデルが対話できる主要なツールタイプは、拡張機能、Function、およびデータストアの 3 つです。エージェントにツールを装備することで、私たちは、それらが世界を理解するだけでなく、それに基づいて行動するための広大な可能性を解き放ち、無数の新しいアプリケーションと可能性への扉を開きます。

拡張機能

拡張機能を理解する最も簡単な方法は、それらを、API とエージェント間のギャップを標準化された方法で埋め、基盤となる実装に関係なくエージェントが API をシームレスに実行できるようにするもの、と考えることです。あなたがユーザーのフライト予約を支援することを目標とするエージェントを構築したとしましょう。あなたはフライト情報を取得するために Google Flights API を使用したいと考えていますが、エージェントにこの API エンドポイントへの API 呼び出しを行わせる方法がわかりません。

図 3：エージェントはどのように外部 API と対話するのか？

一つのアプローチとして、入ってくるユーザーの問い合わせを受け取り、関連情報を解析し、その後 API 呼び出しを行うカスタムコードを実装することが考えられます。例えば、フライト予約のユースケースで、ユーザーが「オースティンからチューリッヒへのフライトを予約したい」と述べたとします。このシナリオでは、私たちのカスタムコードソリューションは、API 呼び出しを試みる前に、ユーザーの問い合わせから「オースティン」と「チューリッヒ」を関連エンティティとして抽出する必要があります。しかし、ユーザーが「チューリッヒへのフライトを予約したい」と言い、出発都市を伝えなかった場合はどうなるでしょうか?必要なデータがないため API 呼び出しは失敗し、このようなエッジケースやコーナーケースを捕捉するためにより多くのコードを実装する必要が生じます。このアプローチはスケーラブルではなく、実装されたカスタムコードの範囲外のいかなるシナリオでも容易に破綻する可能性があります。

より回復力のあるアプローチは、拡張機能を使用することです。拡張機能は、次の方法でエージェントと API の間のギャップを埋めます。

1. 例を用いて、API エンドポイントの使用方法をエージェントに教える。
2.API エンドポイントを正常に呼び出すために必要な引数やパラメータをエージェントに教える。

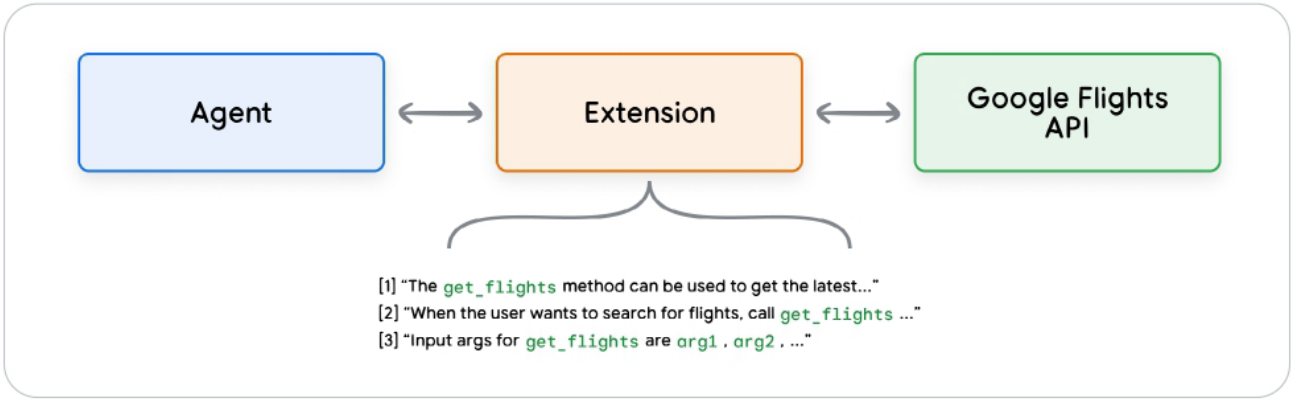

図 4：拡張機能がエージェントを外部 API に接続する

拡張機能はエージェントとは独立して作成できますが、エージェントの設定の一部として提供されるべきです。エージェントは実行時にモデルと例を使用して、ユーザーの問い合わせを解決するのにどの拡張機能が（もしあれば）適しているかを判断します。これは拡張機能の主要な強み、すなわち組み込みの例の型を浮き彫りにします。これにより、エージェントはタスクに最も適した拡張機能を動的に選択できます。

図 5：エージェント、拡張機能、API 間の 1 対多の関係

ソフトウェア開発者がユーザーの問題の解決策を設計し提供する際に、どの API エンドポイントを使用するかを決定するのと同じように考えてください。ユーザーがフライトを予約したい場合、開発者は Google Flights API を使用するかもしれません。ユーザーが現在地から最も近いコーヒーショップの場所を知りたい場合、開発者は Google Maps API を使用するかもしれません。同じように、エージェント／モデルスタックは既知の拡張機能のセットを使用して、どれがユーザーの問い合わせに最適かを判断します。拡張機能が実際に動作するところを見たい場合は、Gemini アプリケーションで、「設定」>「拡張機能」に移動し、テストしたいものを有効にすることで試すことができます。例えば、Google Flights 拡張機能を有効にしてから Gemini に「来週の金曜日に出発するオースティンからチューリッヒへのフライトを表示して」と尋ねることができます。

サンプル拡張機能

拡張機能の使用を簡素化するために、Google はいくつかの標準提供の拡張機能を提供しています。これらはプロジェクトに迅速にインポートし、最小限の設定で使用できます。例えば、スニペット 1 のコードインタプリタ拡張機能は、自然言語による記述から Python コードを生成し実行することを可能にします。

Python

```
import vertexai
import pprint

PROJECT_ID = “YOUR_PROJECT_ID”
REGION = “us-central1”

vertexai.init(project=PROJECT_ID, location=REGION)

from vertexai.preview.extensions import Extension

extension_code_interpreter = Extension.from_hub(“code_interpreter”)
CODE_QUERY = “””Write a python method to invert a binary tree in O(n) time.”””

response = extension_code_interpreter.execute(
  operation_id = “generate_and_execute”,
  operation_params = {“query”: CODE_QUERY}
  )

print(“Generated Code:”)
pprint.pprint({response[‘generated_code’]})

# The above snippet will generate the following code.
```
Generated Code:
```
class TreeNode:
  def __init__(self, val=0, left=None, right=None(:
    self.val = val
    self.left = left
    self.right = right
```

次のページに続きます

Python

```
def invert_binary_tree(root):
    """
    Inverts a binary tree.
    Args:
        root: The root of the binary tree.
    Returns:
        The root of the inverted binary tree.
    """

    if not root:
        return None

    # Swap the left and right children recursively
    root.left, root.right =
invert_binary_tree(root.right), invert_binary_tree(root.left)

    return root

# Example usage:
# Construct a sample binary tree
root = TreeNode(4)
root.left = TreeNode(2)
root.right = TreeNode(7)
root.left.left = TreeNode(1)
root.left.right = TreeNode(3)
root.right.left = TreeNode(6)
root.right.right = TreeNode(9)


# Invert the binary tree
inverted_root = invert_binary_tree(root)
```

スニペット 1. コードインタプリタ拡張機能は Python コードを生成し実行できる

要約すると、拡張機能は、エージェントが無数の方法で外部世界を認識し、対話し、影響を与えるための方法を提供します。これらの拡張機能の選択と呼び出しは、例の使用によってガイドされます。それらはすべて拡張機能設定の一部として定義されています。

Function

ソフトウェアエンジニアリングの世界では、関数（functions）は、特定のタスクを達成し必要に応じて再利用可能な、自己完結型のコードモジュールとして定義されます。ソフトウェア開発者がプログラムを作成する際、様々なタスクを実行するために多くの関数（functions）を作成します。また、いつ関数 a を呼び出し、いつ関数 b を呼び出すかという論理や、期待される入力と出力を定義します。

エージェントの世界でも、Function は非常によく似た形で機能しますが、ソフトウェア開発者をモデルに置き換えることができます。モデルは既知の Function のセットを受け取り、各 Function をいつ使用するか、そしてその仕様に基づいて Function が必要とする引数は何かを決定できます。Function はいくつかの点で拡張機能とは異なり、最も顕著なのは次の点です。

1. モデルは Function とその引数を出力しますが、実際の API 呼び出しは行いません。
2.Function はクライアントサイドで実行されるのに対し、拡張機能はエージェントサイドで実行されます。

再び Google Flights の例を用いると、Function の単純な設定は図 7 の例のようになります。

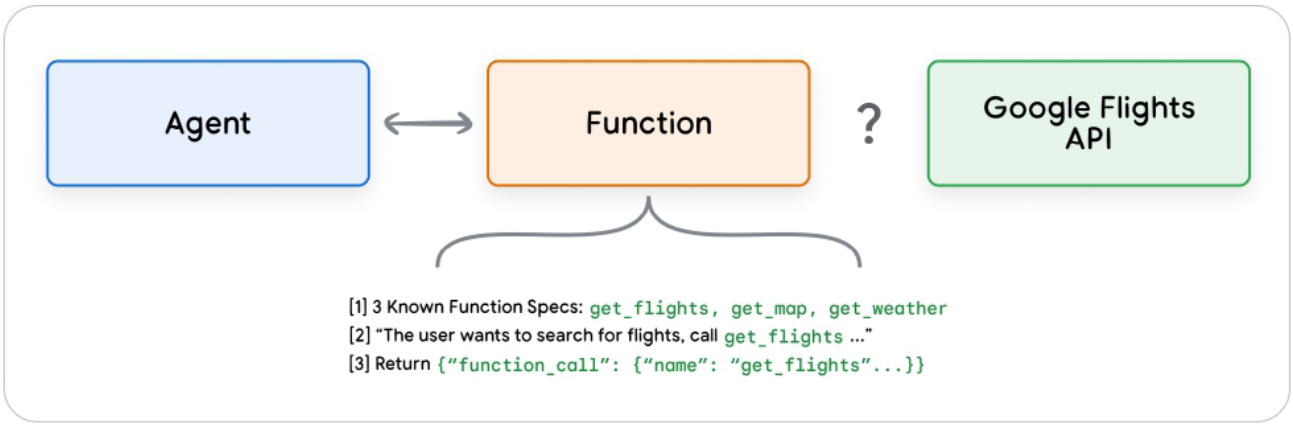

図 7：Function はどのように外部 API と対話するのか?

ここで注意すべき主な違いは、Function もエージェントも、Google Flights API と直接やり取りをしないという点です。では、実際の API 呼び出しはどのように行われるのでしょうか？

Function を使用すると、実際の API エンドポイントを呼び出すロジックと実行は、エージェントからクライアントサイドのアプリケーションにオフロードされます（下の図 8 および図 9 参照）。これにより、開発者はアプリケーション内のデータフローに対して、よりきめ細かい制御を行うことができます。開発者が拡張機能よりも Function を選択する理由は多くありますが、いくつかの一般的なユースケースは次のとおりです。

- API 呼び出しを、直接的なエージェントアーキテクチャのフロー外にあるアプリケーションスタックの別レイヤー（例：ミドルウェアシステム、フロントエンドフレームワークなど）で行う必要がある場合。
- エージェントによる API の直接呼び出しを妨げるセキュリティまたは認証上の制約がある場合（例：API がインターネットに公開されていない、またはエージェントのインフラストラクチャからアクセスできない）。
- エージェントによるリアルタイムでの API 呼び出しを妨げるタイミングまたは操作順序の制約がある場合（例：バッチ処理、ヒューマンインザループによるレビューなど）。
- エージェントが実行できない追加のデータ変換ロジックを API 応答に適用する必要がある場合。例えば、返される結果数を制限するフィルタリングメカニズムを提供しない API エンドポイントを考えます。クライアントサイドで Function を使用することで、開発者はこれらの変換を行う追加の機会を得られます。
- 開発者が API エンドポイント用に追加のインフラストラクチャをデプロイせずにエージェント開発を繰り返したい場合（つまり、Function 呼び出しは API の「スタブ化」のように機能し得る）。

図 8 に示すように、2 つのアプローチ間の内部アーキテクチャの違いは微妙ですが、追加の制御と外部インフラストラクチャへの非結合依存により、Function 呼び出しは開発者にとって魅力的な選択肢となります。

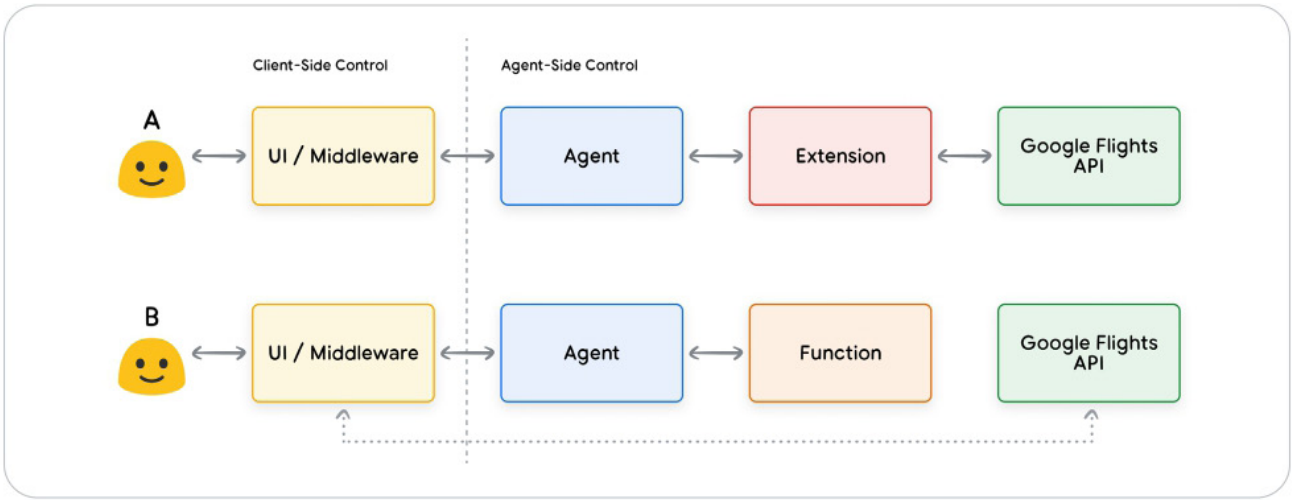

図 8：拡張機能と Function 呼び出しにおけるクライアントサイド対エージェントサイドの制御の明確化

ユースケース

モデルは、エンドユーザーのための複雑なクライアントサイドの実行フローを処理するために Function を呼び出すのに使用できます。これは、エージェント開発者が（拡張機能の場合のように）言語モデルに API 実行を管理させたくない場合に適しています。休暇旅行を予約したいユーザーと対話する旅行コンシェルジュとして訓練されているエージェントの例を考えてみましょう。目標は、エージェントに都市のリストを生成させ、それをミドルウェアアプリケーションで使用してユーザーの旅行計画用の画像やデータなどをダウンロードできるようにすることです。ユーザーは次のように言うかもしれません。

「家族とスキー旅行に行きたいのですが、どこへ行けばいいかわかりません。」

モデルへの典型的なプロンプトでは、出力は次のようになるかもしれません。

かしこまりました、家族でのスキー旅行にご検討いただける都市のリストはこちらです：
・クレステッド・ビュート、コロラド州、アメリカ
・ウィスラー、ブリティッシュコロンビア州、カナダ
・ツェルマット、スイス

上記の出力には必要なデータ（都市名）が含まれていますが、解析には理想的な形式ではありません。Function 呼び出しを使えば、モデルにこの出力を他のシステムが解析するのにより便利な構造化形式（JSON など）で整形するように教えることができます。ユーザーからの同じ入力プロンプトが与えられた場合、Function からの JSON 出力例は、代わりにスニペット 5 のようになるかもしれません。

```
function_call {
  name: “display_cities”
  args: {
    “cities”: [“Crested Butte”, “Whistler”, “Zermatt”],
    “preferences”: “skiing”
    }
}
```

スニペット 5. 都市のリストとユーザーの好みを表示するための Function 呼び出しペイロードのサンプル

この JSON ペイロードはモデルによって生成され、その後、私たちが望む処理を行うためにクライアントサイドサーバーに送信されます。この特定のケースでは、Google Places API を呼び出し、モデルによって提供された都市を取得して画像を検索し、それらを整形されたリッチコンテンツとしてユーザーに返します。上記の相互作用を段階的に詳細に示した図 9 のシーケンス図をご覧ください。

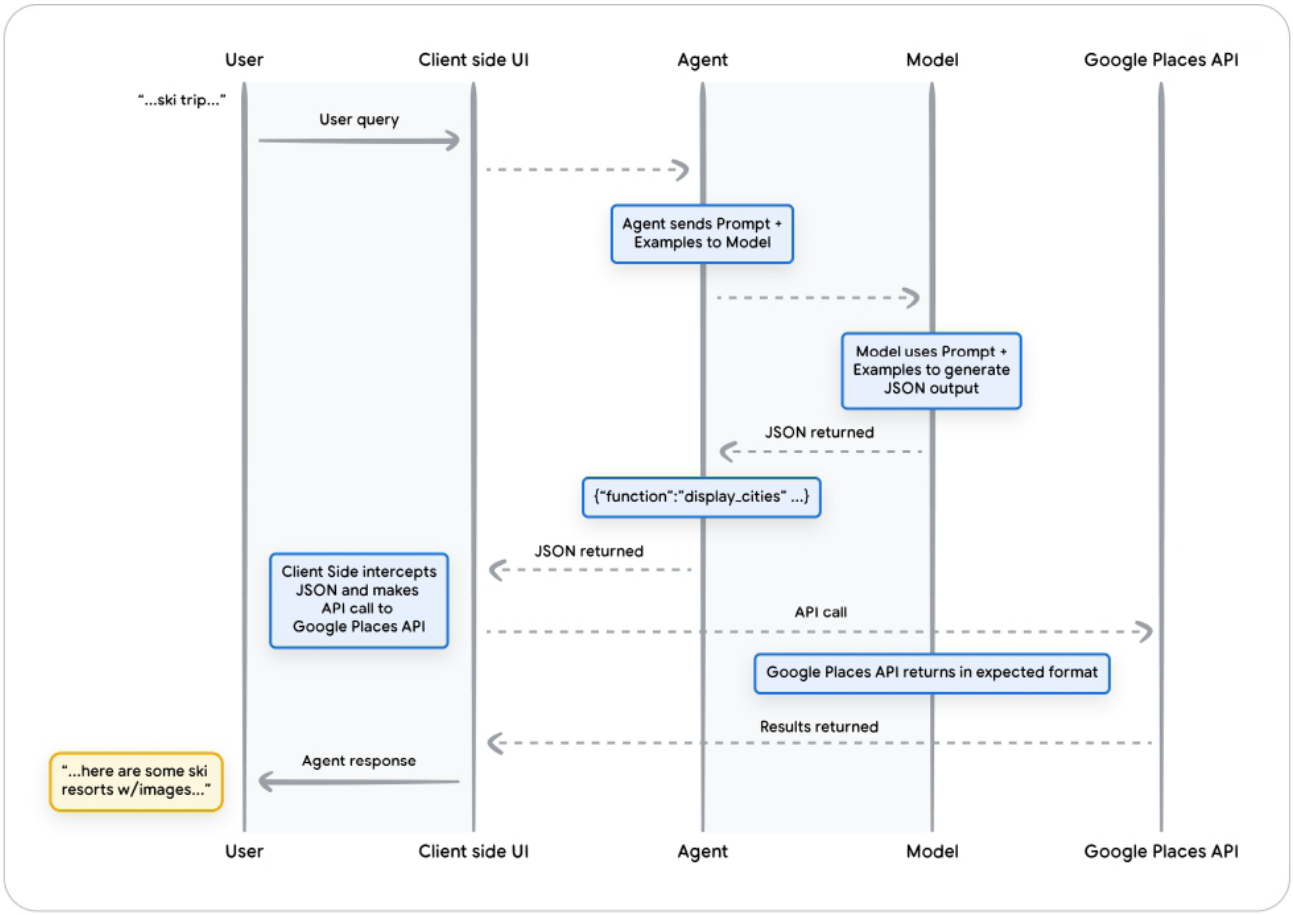

図 9：Function 呼び出しのライフサイクルを示すシーケンス図

図 9 の例の結果として、モデルは、クライアントサイド UI（ユーザーインターフェース）が Google Places API への API 呼び出しを行うために必要なパラメータで「穴埋め」をするために活用されます。クライアントサイド UI は、返された Function 内でモデルによって提供されたパラメータを使用して、実際の API 呼び出しを管理します。これは Function 呼び出しの一つのユースケースにすぎませんが、次のような考慮すべき他の多くのシナリオがあります。

- 言語モデルにコードで使用できる関数を提案させたいが、コードに認証情報を含めたくない場合。Function 呼び出しは関数を実行しないため、関数の情報と共にコードに認証情報を含める必要はない。
- 数秒以上かかる可能性のある非同期処理を実行している場合。Function 呼び出しは非同期処理であるため、これらのシナリオとうまく連携する。
- Function 呼び出しとその引数を生成するシステムとは異なるデバイスで関数を実行したい場合。

Function について覚えておくべき重要なことの一つは、API 呼び出しの実行だけでなく、アプリケーション全体のデータフローに対しても、開発者にはるかに多くの制御を提供することを目的としているという点です。図 9 の例では、エージェントが取る可能性のある将来の行動には関連がなかったため、開発者は API 情報をエージェントに返さないことを選択しました。しかし、アプリケーションのアーキテクチャによっては、将来の推論、論理、および行動の選択に影響を与えるために、外部 API 呼び出しのデータをエージェントに返すことが理にかなっている場合があります。最終的には、特定のアプリケーションにとって何が正しいかを選択するのはアプリケーション開発者次第です。

Function サンプルコード

私たちのスキー休暇シナリオから上記の出力を達成するために、これを gemini-2.0-flash-001 モデルで動作させるための各コンポーネントを構築しましょう。 まず、display_cities 関数を単純な Python メソッドとして定義します。

Python

```
from typing import Optional

def display_cities(cities: list[str], preferences: Optional[str] = None):
"""Provides a list of cities based on the user's search query and preferences.

   Args:
      preferences (str): The user's preferences for the search, like skiing,
      beach, restaurants, bbq, etc.
      cities (list[str]): The list of cities being recommended to the user.

   Returns:
      list[str]: The list of cities being recommended to the user.
   """

   return cities
```

スニペット 6. 都市のリストを表示する関数のための Python メソッドのサンプル

次に、モデルをインスタンス化し、ツール（Tool）を構築し、その後、ユーザーの問い合わせとこれらのツールをモデルに渡します。下のコードを実行すると、コードスニペットの最後に示すような出力が得られます。

Python

```
from google.genai import Client, types

client = Client(
    vertexai=True,
    project="PROJECT_ID",
    location="us-central1"
    )

res = client.models.generate_content(
    model="gemini-2.0-flash-001",
    model="I'd like to take a ski trip with my family but I'm not sure where
to go?",
    config=types.GenerateContentConfig(
        tools=[display_cities],
   automatic_function_calling=types.AutomaticFunctionCallingConfig(disable=True),
        tool_config=types.ToolConfig(
            function_calling_config=types.FunctionCallingConfig(mode='ANY')
            )
        )
)

print(f"Function Name: {res.candidates[0].content.parts[0].function_call.name}")
print(f"Function Args: {res.candidates[0].content.parts[0].function_call.args}")

> Function Name: display_cities
> Function Args: {'preferences': 'skiing', 'cities': ['Aspen', 'Park
City', 'Whistler']}

```

スニペット 7. ツール（Tool）を構築し、ユーザーの問い合わせと共にモデルに送信し、Function 呼び出しを実行させる

要約すると、Function は、アプリケーション開発者にデータフローとシステム実行に対するきめ細かい制御権を与えると同時に、重要な入力生成のためにエージェント／モデルを効果的に活用できる簡単なフレームワークを提供します。開発者は、外部データを返すことでエージェントを「ループ内」に留めるか、特定のアプリケーションアーキテクチャの要件に基づいてそれを省略するかを選択的に決定できます。

データストア

言語モデルを、その訓練データを含む広大な図書館だと想像してみてください。しかし、継続的に新しい蔵書を獲得していく図書館とは異なり、この図書館は静的であり、最初に訓練された時点の知識のみを保持しています。実世界の知識は絶えず進化しているため、これは課題となります。データストアは、より動的で最新の情報へのアクセスを提供し、モデルの応答が事実性と関連性に根差したものであり続けることを保証することで、この限界に対処します。開発者がモデルに少量の追加データ（おそらくスプレッドシートや PDF の形式）を提供する必要がある一般的なシナリオを考えてみましょう。

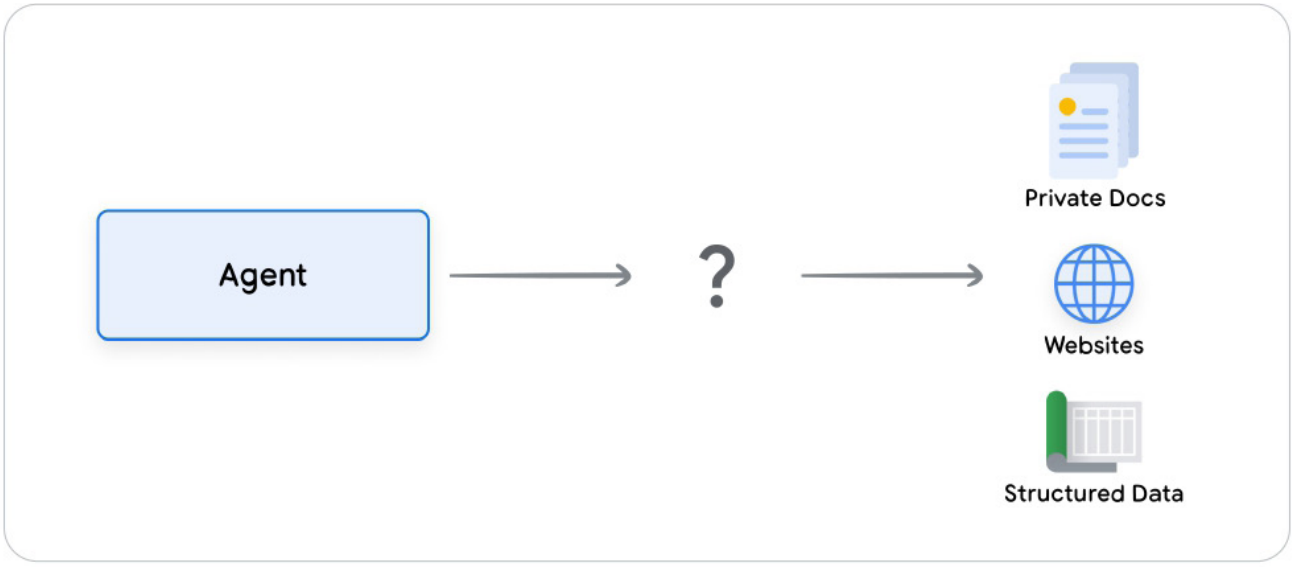

図 10：エージェントはどのように構造化データおよび非構造化データと対話できるか？

データストアを使用すると、開発者は元の形式の追加データをエージェントに提供でき、時間のかかるデータ変換、モデルの再訓練、またはファインチューニングの必要がなくなります。データストアは入力されたドキュメントをベクトルデータベース埋め込みのセットに変換します。エージェントはこれを使用して、ユーザーへの次の行動や応答を補足するために必要な情報を抽出できます。

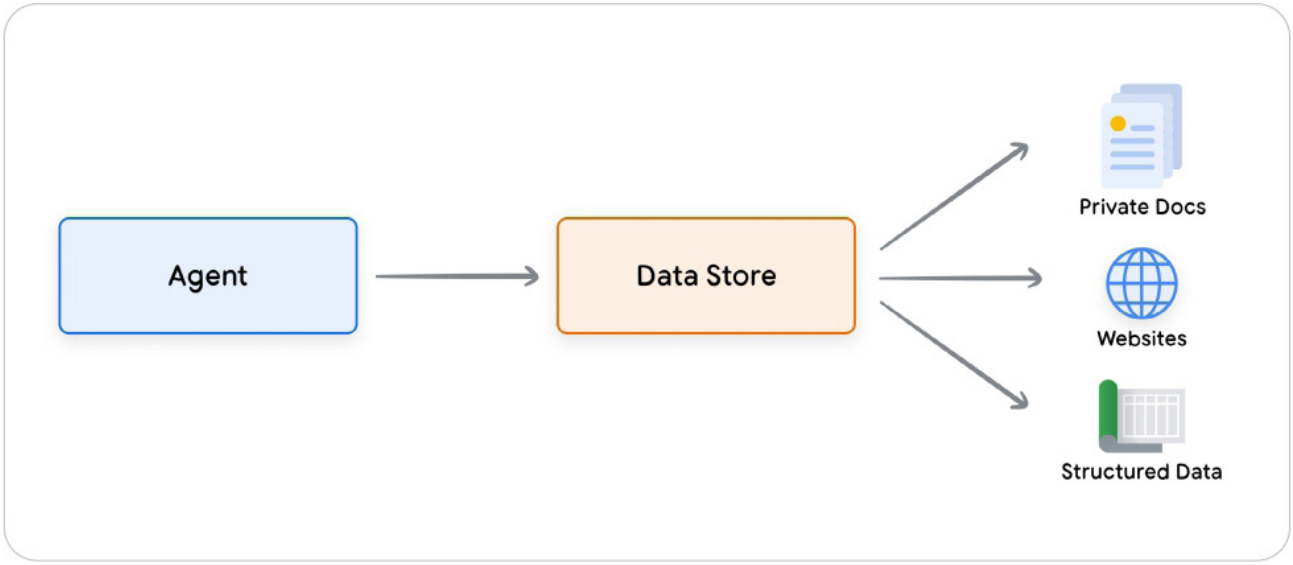

図 11：データストアはエージェントを様々なタイプの新しいリアルタイムデータソースに接続する

実装と応用

生成 AI エージェントの文脈では、データストアは通常、開発者がエージェントに実行時にアクセスさせたいと考えるベクトルデータベースとして実装されます。ここではベクトルデータベースについて詳しく説明しませんが、理解すべき重要な点は、それらがベクトル埋め込み（提供されたデータの高次元ベクトルまたは数学的表現の一種）の形式でデータを格納するということです。近年における言語モデルでのデータストア使用の最も顕著な事例の一つは、検索拡張生成（RAG）ベースのアプリケーションの実装です。これらのアプリケーションは、モデルに次のような様々な形式のデータへのアクセスを与えることにより、その知識の幅と深さを基盤となる訓練データの範囲を超えて拡張することを目指しています。

- ウェブサイトのコンテンツ
- PDF、Word 文書、CSV、スプレッドシートなどの形式の構造化データ
- HTML、PDF、TXT などの形式の非構造化データ

図 12：エージェントとデータストア間の 1 対多の関係（様々なタイプの事前インデックス済みデータを表現可能）

各ユーザーリクエストとエージェント応答ループの基盤となるプロセスは、一般的に図 13 に示すようにモデル化されます。

1. ユーザーの問い合わせが埋め込みモデルに送信され、問い合わせの埋め込みが生成される。
2. 次に、問い合わせの埋め込みが、SCaNN のようなマッチングアルゴリズムを使用してベクトルデータベースの内容と照合される。
3. 照合されたコンテンツがテキスト形式でベクトルデータベースから取得され、エージェントに返送される。
4. エージェントはユーザーの問い合わせと取得されたコンテンツの両方を受け取り、応答または行動を形成する。
5. 最終的な応答がユーザーに送信される。

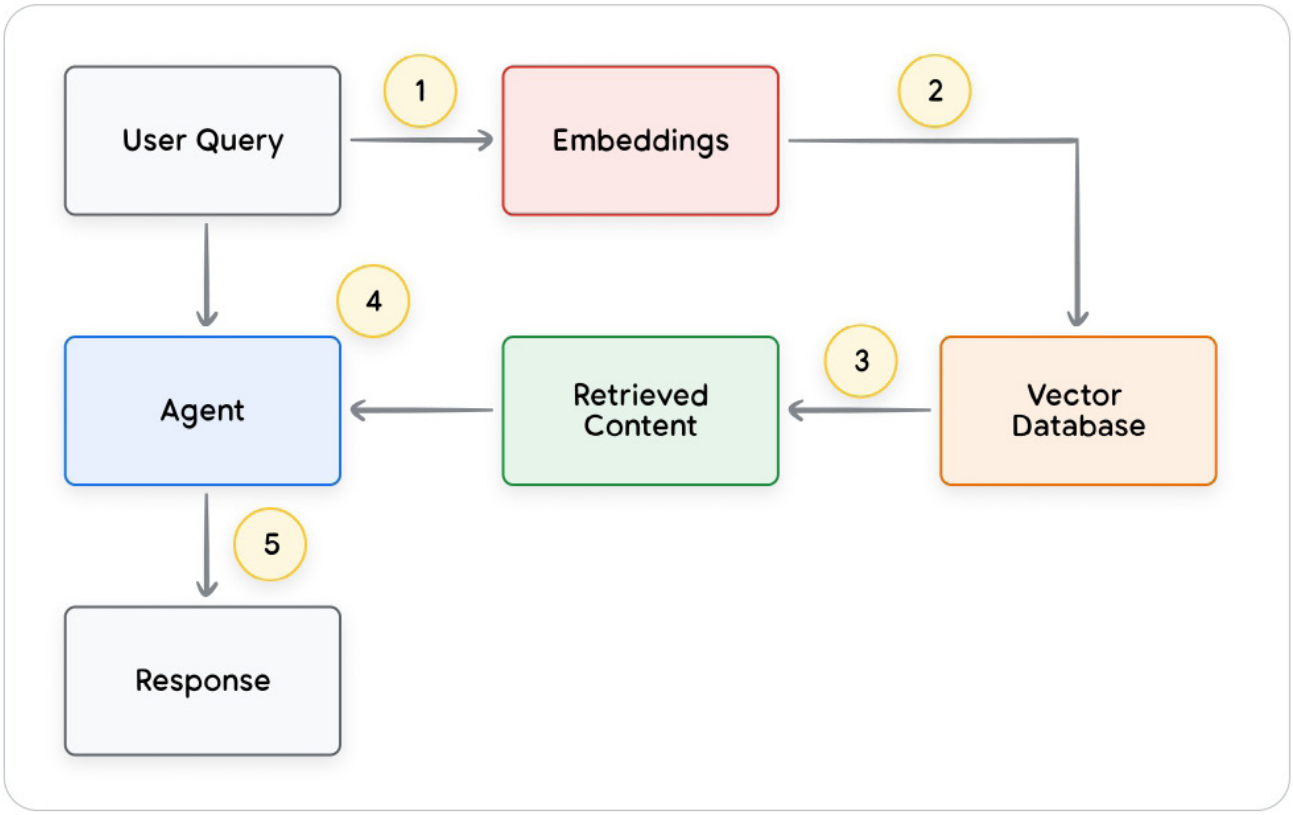

図 13：RAG ベースのアプリケーションにおけるユーザーリクエストとエージェント応答のライフサイクル

最終結果は、エージェントがベクトル検索を通じてユーザーの問い合わせを既知のデータストアと照合し、元のコンテンツを取得し、それをオーケストレーション層とモデルに提供してさらなる処理を行わせることを可能にするアプリケーションです。次の行動は、ユーザーに最終的な回答を提供すること、あるいは結果をさらに絞り込むため追加のベクトル検索を実行することかもしれません。 ReAct の推論／計画を用いた RAG を実装するエージェントとの対話例は、図 14 で見ることができます。

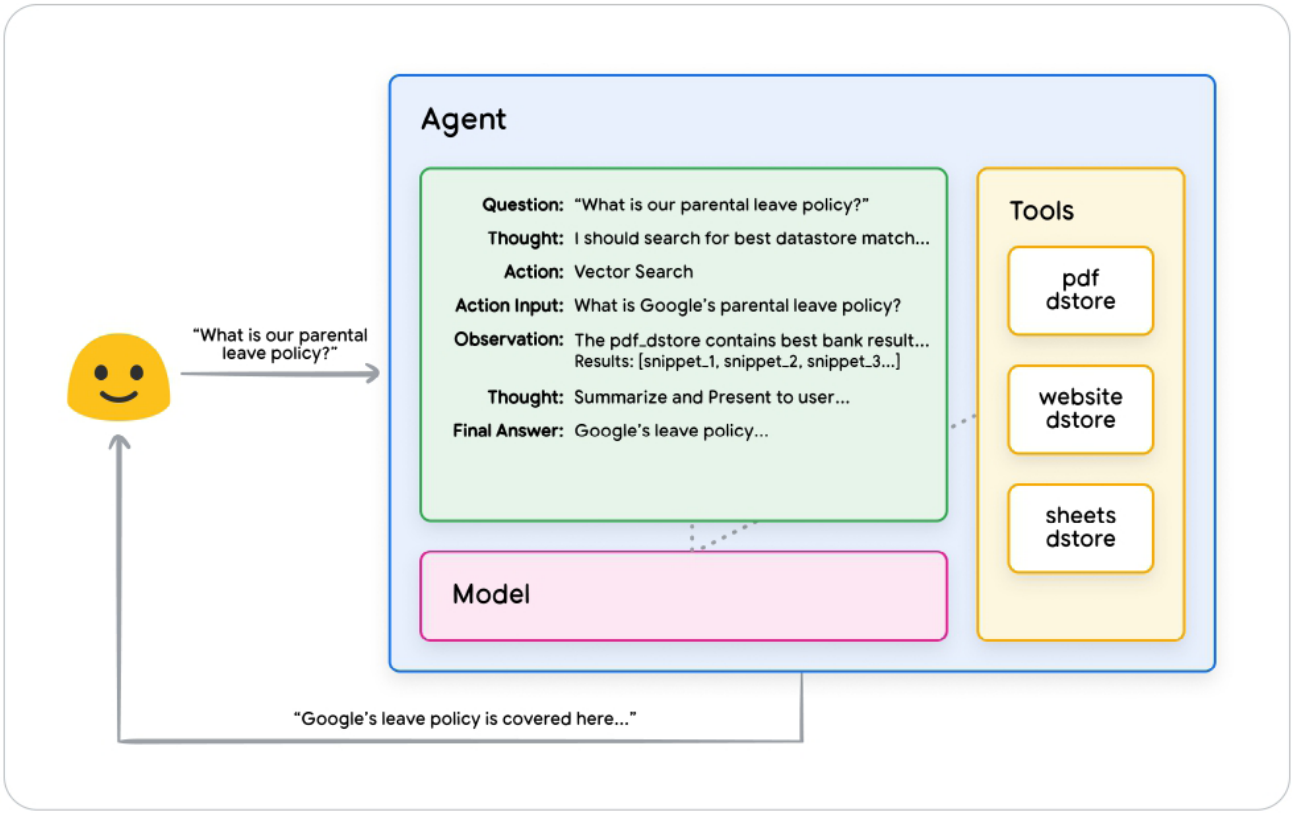

図 14：ReAct の推論／計画を用いた RAG ベースのアプリケーションのサンプル

ツールのまとめ

要約すると、拡張機能、Function、およびデータストアは、エージェントが実行時に使用できるいくつかの異なるツールタイプを構成します。それぞれに独自の目的があり、エージェント開発者の裁量で、一緒にまたは独立して使用できます。

	拡張機能	Function 呼び出し	データストア
実行	エージェントサイド実行	クライアントサイド実行	エージェントサイド実行
ユースケース	・開発者がエージェントに API エンドポイントとの対話を制御させたい場合 ・ネイティブの構築済み拡張機能（例：Vertex Search、Code Interpreter など）を活用する場合に便利 ・マルチホップの計画と API 呼び出し（つまり、次のエージェントの行動が前のアクション／ API 呼び出しの出力に依存する場合）	・セキュリティまたは認証の制約によりエージェントが API を直接呼び出せない場合 ・タイミングの制約または操作順序の制約によりエージェントがリアルタイムで API 呼び出しを行えない場合（例：バッチ処理、ヒューマンインザループによるレビューなど） ・API がインターネットに公開されていない、または Google のシステムからアクセスできない場合	開発者が次のいずれかのデータタイプで検索拡張生成（RAG）を実装したい場合： ・事前インデックス済みのドメインおよび URL からのウェブサイトコンテンツ ・PDF、Word 文書、CSV、スプレッドシートなどの形式の構造化データ ・リレーショナル／非リレーショナルデータベース ・HTML、PDF、TXT などの形式の非構造化データ

ターゲット学習によるモデル性能の向上

モデルを効果的に使用する上で重要な側面は、出力を生成する際に適切なツールを選択する能力であり、これは特に本番環境でツールを大規模に使用する場合に顕著です。一般的な訓練はモデルがこのスキルを習得するのに役立ちますが、実世界のシナリオではしばしば訓練データを超える知識が必要となります。これを、基本的な調理スキルと特定の料理を習得することの違いとして想像してみてください。どちらも基礎的な調理知識を必要としますが、後者はよりニュアンスのある結果を得るためにターゲット学習を必要とします。

モデルがこの種の特定の知識にアクセスできるよう支援するために、いくつかのアプローチが存在します。

- **インコンテキスト学習：**この手法は、一般化されたモデルにプロンプト、ツール、およびフューショット事例を推論時に提供することで、特定のタスクに対してそれらのツールをいつ、どのように使用するかを「オンザフライで」学習できるようにするものです。ReAct フレームワークは、自然言語におけるこのアプローチの一例です。

- **検索ベースのインコンテキスト学習：**この技術は、外部メモリから最も関連性の高い情報、ツール、および関連する例を取得し、それらを動的にモデルプロンプトに投入するものです。この例としては、Vertex AI 拡張機能の「Example Store」や、前述のデータストアの RAG ベースアーキテクチャが挙げられます。

- **ファインチューニングベースの学習：**この手法は、推論の前に、特定の事例の大規模なデータセットを使用してモデルを訓練することを含みます。これにより、モデルはユーザーの問い合わせを受け取る前に、特定のツールをいつ、どのように適用するかを理解できます。

各ターゲット学習アプローチに関する追加の洞察を提供するために、私たちの料理の例えを再度見てみましょう。

- シェフが顧客から特定のレシピ（プロンプト）、いくつかの主要な食材（関連ツール）、いくつかの料理の例（フューショット事例）を受け取ったと想像してみてください。この限られた情報とシェフの一般的な調理知識に基づいて、レシピと顧客の好みに最も近くなるように、料理を「オンザフライで」準備する方法を考え出す必要があります。これがインコンテキスト学習です。
- 次に、様々な食材や料理本（例とツール）で満たされた、品揃えの豊富な食料庫（外部データストア）のある厨房にいるシェフを想像してみましょう。シェフは今や食料庫から食材や料理本を動的に選択し、顧客のレシピや好みに、よりうまく合わせることができます。これにより、シェフは既存の知識と新しい知識の両方を活用して、より情報に基づいた洗練された料理を作ることができます。これが検索ベースのインコンテキスト学習です。
- 最後に、シェフを学校に送り返して新しい料理または一連の料理（特定の事例の大規模なデータセットでの事前訓練）を学ばせたと想像してみましょう。これにより、シェフは将来の未知の顧客のレシピに対して、より深い理解をもってアプローチできます。このアプローチは、シェフに特定の料理（知識ドメイン）で卓越してもらいたい場合に最適です。これがファインチューニングベースの学習です。

これらの各アプローチは、速度、コスト、レイテンシの観点から、それぞれ独自の利点と欠点があります。しかし、これらの技術をエージェントフレームワークで組み合わせることにより、様々な強みを活用し弱みを最小限に抑えることができ、より堅牢で適応性のあるソリューションが可能になります。

LangChain によるエージェントクイックスタート

実際に動作するエージェントの実行可能な実世界での例を提供するために、LangChain および LangGraph ライブラリを使用して簡単なプロトタイプを構築します。これらの人気のあるオープンソースライブラリを使用すると、ユーザーは、論理、推論、およびツール呼び出しのシーケンスを「連鎖させる」ことによって、ユーザーの問い合わせに答えるためのカスタムエージェントを構築できます。私たちは gemini-2.0-flash-001 モデルといくつかの簡単なツールを使用して、スニペット 8 に示すように、ユーザーからの複数段階の問い合わせに答えます。 私たちが使用しているツールは、SerpAPI（Google 検索用）と Google Places API です。スニペット 8 のプログラムを実行すると、スニペット 9 でサンプル出力を確認できます。

Python

```
from langgraph.prebuilt import create_react_agent
from langchain_core.tools import tool
from langchain_community.utilities import SerpAPIWrapper
from langchain_community.tools import GooglePlacesTool

os.environ[“SERPAPI_API_KEY”] = “XXXXX”
os.environ[“GPLACES_API_KEY”] = “XXXXX”

@tool
def search(query: str):
    “””Use the SerpAPI to run a Google Search.”””
    search = SerpAPIWrapper()
    return search.run(query)

@tool
def places(query: str):
    “””Use the Google Places API to run a Google Places Query.”””
    places = GooglePlacesTool()
    return places.run(query)

model = ChatVertexAI(model=”gemini-2.0-flash-001”)
tools = [search, places]

query = “Who did the Texas Longhorns play in football last week? What is the
address of the other team’s stadium?”

agent = create_react_agent(model, tools)
input = {“messages”: [(“human”, query)]}

for s in agent.stream(input, stream_mode=”values”):
    message = s[“messages”][-1]
    if isinstance(message, tuple):
        print(message)
    else:
        message.pretty_print()
```

スニペット 8. ツールを備えた LangChain および LangGraph ベースのエージェントのサンプル

```
============================== Human Message ================================
Who did the Texas Longhorns play in football last week? What is the address
of the other team's stadium?
================================== Ai Message ==================================
Tool Calls: search
Args:
   query: Texas Longhorns football schedule
================================= Tool Message =================================
Name: search
{...Results: "NCAA Division I Football, Georgia, Date..."}
================================== Ai Message ==================================
The Texas Longhorns played the Georgia Bulldogs last week.
Tool Calls: places
Args:
   query: Georgia Bulldogs stadium
================================= Tool Message =================================
Name: places

{...Sanford Stadium Address: 100 Sanford...}
================================== Ai Message ==================================
The address of the Georgia Bulldogs stadium is 100 Sanford Dr, Athens, GA
30602, USA.
```

スニペット 9. スニペット 8 のプログラムからの出力

これはかなり単純なエージェントの例ですが、モデル、オーケストレーション、およびツールという基本コンポーネントがすべて連携して特定の目標を達成することを示しています。最終セクションでは、Vertex AI エージェントや Generative Playbooks のような Google 規模のマネージド製品において、これらのコンポーネントがどのように統合されるかを探求します。

Vertex AI エージェントによる本番アプリケーション

このホワイトペーパーではエージェントのコアコンポーネントを探求しましたが、本番グレードのアプリケーションを構築するには、それらをユーザーインターフェース、評価フレームワーク、継続的改善メカニズムといった付加的なツールと統合する必要があります。Google の Vertex AI プラットフォームは、先に説明したすべての基本要素を備えたフルマネージド環境を提供することにより、このプロセスを簡素化します。自然言語インターフェースを使用して、開発者はエージェントの重要な要素（目標、タスク指示、ツール、タスク委任のためのサブエージェント、例など）を迅速に定義し、望ましいシステム動作を容易に構築できます。さらに、このプラットフォームには一連の開発ツールが付属しており、開発されたエージェントのテスト、評価、エージェントパフォーマンスの測定、デバッグ、および全体的な品質の向上を可能にします。これにより、開発者はエージェントの構築と改良に集中できる一方、インフラストラクチャ、デプロイ、メンテナンスの複雑さはプラットフォーム自体によって管理されます。

図 15 では、Vertex AI プラットフォーム上に構築され、Vertex Agent Builder、Vertex Extensions、Vertex Function Calling、Vertex Example Store といった（これらはほんの数例ですが）様々な機能を使用したエージェントのサンプルアーキテクチャを示しています。このアーキテクチャには、本番環境に対応したアプリケーションに必要な様々なコンポーネントの多くが含まれています。

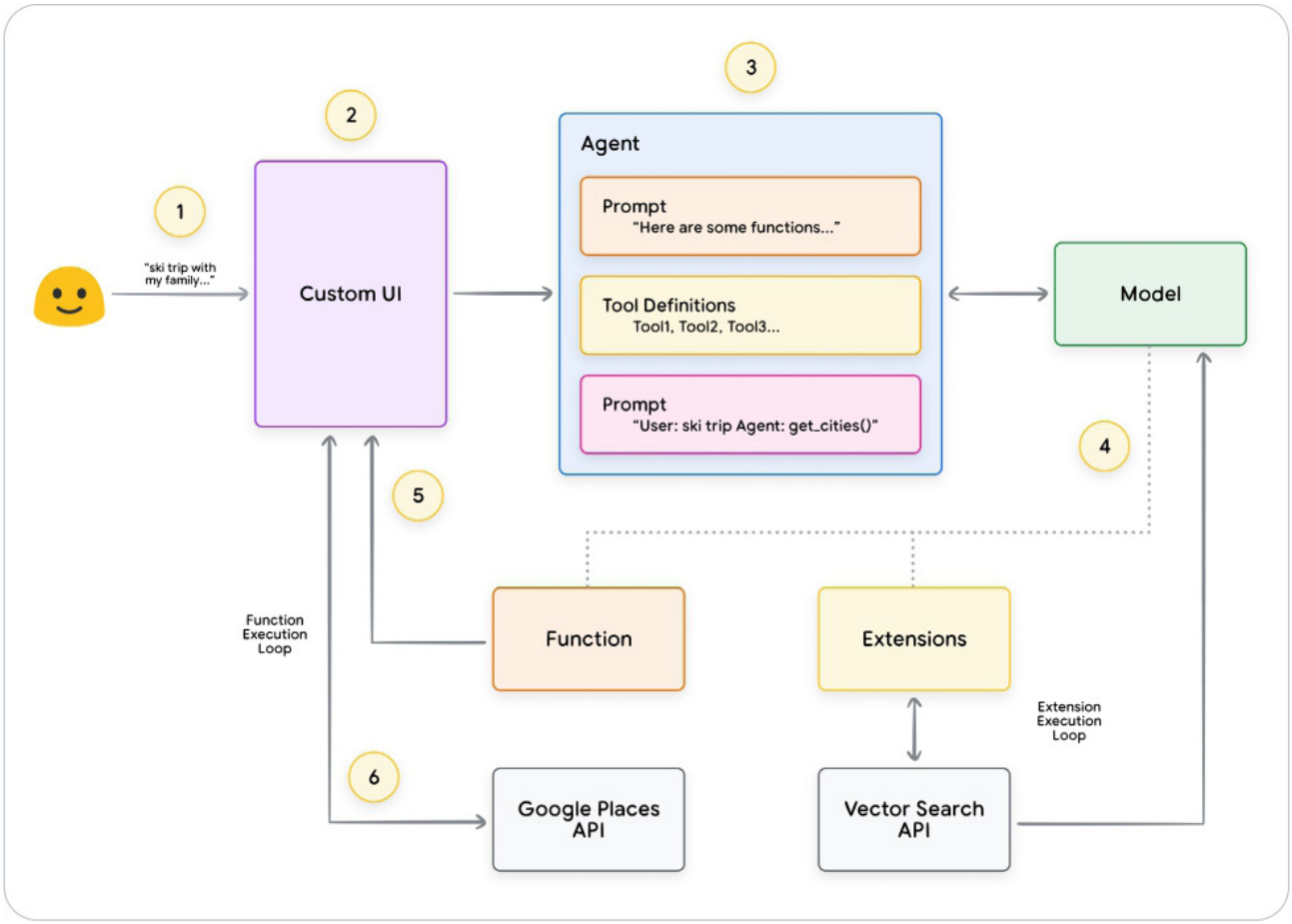

図 15：Vertex AI プラットフォーム上に構築されたエンドツーエンドのエージェントアーキテクチャのサンプル

この構築済みエージェントアーキテクチャのサンプルは、私たちの公式ドキュメントから試すことができます。

まとめ

このホワイトペーパーでは、生成 AI エージェントの基本的な構成要素、その構成、そしてそれらを認知アーキテクチャの形で効果的に実装する方法について議論してきました。このホワイトペーパーからの主要なポイントは次のとおりです。

1. エージェントは、ツールを活用してリアルタイム情報へのアクセス、実世界のアクションの提案、複雑なタスクの自律的な計画・実行を行うことで、言語モデルの能力を拡張します。エージェントは 1 つまたは複数の言語モデルを活用して、状態間をいつどのように遷移するかを決定し、モデル単独では完了が困難または不可能な多数の複雑なタスクを外部ツールを使用して完了できます。

2. エージェントの運用の中心には、推論、計画、意思決定を構造化し、その行動をガイドする認知アーキテクチャであるオーケストレーション層があります。ReAct、Chain-of-Thought、Tree-of-Thoughts などの様々な推論技術は、オーケストレーション層が情報を取り込み、内部推論を実行し、情報に基づいた意思決定や応答を生成するためのフレームワークを提供します。

3. 拡張機能、Function、データストアなどのツールは、エージェントにとって外部世界への鍵として機能し、外部システムと対話し訓練データを超える知識にアクセスすることを可能にします。拡張機能はエージェントと外部 API の間の橋渡しをし、API 呼び出しの実行とリアルタイム情報の取得を可能にします。Function は分業を通じて開発者によりニュアンスのある制御を提供し、エージェントがクライアントサイドで実行可能な Function パラメータを生成できるようにします。データストアはエージェントに構造化データまたは非構造化データへのアクセスを提供し、データ駆動型アプリケーションを可能にします。

エージェントの未来はエキサイティングな進歩を秘めており、私たちはその可能性のほんの表面をかすめたにすぎません。ツールがより洗練され、推論能力が強化されるにつれて、エージェントはますます複雑な問題を解決する力を得るでしょう。 さらに、「エージェントチェイニング」という戦略的アプローチは勢いを増し続けるでしょう。特定のドメインやタスクに長けた特化型エージェントを組み合わせることで、様々な業界や問題領域で卓越した結果をもたらすことができる「エージェントエキスパートの混合」アプローチを作り出すことができます。

複雑なエージェントアーキテクチャの構築には反復的なアプローチが必要であることを覚えておくことが重要です。特定のビジネスケースや組織的ニーズに対する解決策を見つけるには、実験と改良が鍵となります。それらのアーキテクチャを支える基盤モデルの生成的な性質のため、同じように作られるエージェントは二つとありません。しかし、これらの各基本コンポーネントの強みを活用することで、私たちは言語モデルの能力を拡張し、実世界の価値を推進する影響力のあるアプリケーションを作成できます。

卷末注

1. Shafran, I., Cao, Y. et al., 2022, ‘ReAct: Synergizing Reasoning and Acting in Language Models’. Available at: https://arxiv.org/abs/2210.03629
2. Wei, J., Wang, X. et al., 2023, ‘Chain-of-Thought Prompting Elicits Reasoning in Large Language Models’. Available at: https://arxiv.org/pdf/2201.11903.pdf.
3. Wang, X. et al., 2022, ‘Self-Consistency Improves Chain of Thought Reasoning in Language Models’. Available at: https://arxiv.org/abs/2203.11171.
4. Diao, S. et al., 2023, ‘Active Prompting with Chain-of-Thought for Large Language Models’. Available at: https://arxiv.org/pdf/2302.12246.pdf.
5. Zhang, H. et al., 2023, ‘Multimodal Chain-of-Thought Reasoning in Language Models’. Available at: https://arxiv.org/abs/2302.00923.
6. Yao, S. et al., 2023, ‘Tree of Thoughts: Deliberate Problem Solving with Large Language Models’. Available at: https://arxiv.org/abs/2305.10601.
7. Long, X., 2023, ‘Large Language Model Guided Tree-of-Thought’. Available at: https://arxiv.org/abs/2305.08291.
8. Google. ‘Google Gemini Application’. Available at: http://gemini.google.com.
9. Swagger. ‘OpenAPI Specification’. Available at: https://swagger.io/specification/.
10. Xie, M., 2022, ‘How does in-context learning work? A framework for understanding the differences from traditional supervised learning’. Available at: https://ai.stanford.edu/blog/understanding-incontext/.
11. Google Research. ‘ScaNN (Scalable Nearest Neighbors)’. Available at: https://github.com/google-research/google-research/tree/master/scann.
12. LangChain. ‘LangChain’. Available at: https://python.langchain.com/v0.2/docs/introduction/.