エージェント システムの設計パターン

エージェントシステムの構築には、LLM コール、データ取得、および外部アクションがどのように連携して流れるかを調整する必要があります。エージェント システムの設計パターンは、 **決定論的なチェーン** から、動的な決定を下すことができる (内部で複数の LLM 呼び出しが含まれる場合があります) 単一 **エージェント** システム、複数の特殊なエージェントを調整する **マルチエージェント** アーキテクチャまで、 **複雑さと自律性の連続** 性で考えることができます。

ツール呼び出し

設計パターンに飛び込む前に、ツール呼び出しを、単純なものから複雑なものまで、あらゆるエージェント システムで使用できる基本的な機能として理解することが重要です。ツール呼び出しは、エージェント システムが外部関数、データソース、またはサービスと対話できるようにするメカニズムです。これにより、次のことが可能になります。

- SQLクエリ、CRMフェッチ、ベクトルインデックス取得などのライブデータルックアップ。
- Eメールの送信やレコードの更新などのアクション。
- Python関数またはAPIによる任意の論理または変換。

このように、ツール・コールは、LLMが外部データや API を「認識」するための強力なメカニズムを提供します。これは、選択したデザイン・パターンに関係なくです。

エージェント ツールの詳細については、「AI エージェント ツール」を参照してください。

以下のセクションでは、3 つのエージェント システム設計パターンについて説明します。各パターンは、ツール呼び出しをさまざまな程度で活用できます。

生成 AI アプリの設計パターンの比較

生成AI アプリ (エージェント) の設計パターンは、複雑さの順に表示されます。

デザインパターン	いつ使用するか	長所	短所
決定論的チェーン	• 明確に定義されたタスク - 基本的なRAGなどの静的パイプライン - その場での意思決定が不要	• とてもシンプル - 監査が容易	-柔軟性 - 適応するためにコードの変更が必要
シングルエージェントシステム	• 同じドメイン内の中程度から複雑なクエリ - 複数の専門エージェントのオーバーヘッドのないダイナミックな意思決定。	-フレキシブル - マルチエージェントよりもシンプル - 良い「デフォルト」	• 予測しにくい - ツールコールの繰り返しや誤りを防ぐ必要がある
マルチエージェントシステム	大規模または機能横断的なドメイン。複数の「専門家」エージェント。明確なロジックまたは会話のコンテキスト。高度なリフレクションパターン。	• 高度なモジュール - 大規模ドメインへの拡張	• オーケストレーションが複雑 - トレースとデバッグが困難

シングルエージェントシステム

シングルエージェントシステムは、受信リクエストを処理するために、1つの調整されたロジックフロー(多くの場合、複数のLLMコールをオーケストレーション)を備えています。エージェントは次のことができます。

1. ユーザークエリなどのリクエストや、会話履歴などの関連するコンテキストを受け入れます。
2. 最適な応答方法に関する理由付け (必要に応じて、外部データまたはアクションのツールを呼び出すかどうかを決定します)。
3. 必要に応じて、目的が達成されるか、特定の条件 (有効なデータの受信やエラーの解決など) が満たされるまで、LLM (および/または同じツール) を繰り返し呼び出して、反復します。
4. ツールの出力を会話に統合します。
5. まとまりのある応答を出力として返します。

多くのユースケースでは、LLM推論の1ラウンド(多くの場合、ツール呼び出しを使用)で十分です。ただし、より高度なエージェントは、目的の結果に到達するまで複数のステップをループできます。

エージェントが 1 つしかない場合でも、内部で複数の LLM とツール コール (計画、生成、検証など) を使用でき、すべてがこの 1 つの統一されたフローによって管理されます。

例: ヘルプ デスク アシスタント

- ユーザーが簡単な質問(「返品ポリシーは何ですか」)をした場合、エージェントはLLMの知識から直接回答する場合があります。
- ユーザーが注文ステータスを希望する場合、エージェントは関数 `lookup_order(customer_id, order_id)` を呼び出します。そのツールが「無効な注文番号」と応答した場合、エージェントは再試行するか、ユーザーに正しいIDの入力を求めることができ、最終的な回答が得られるまで続行できます。

いつ使用するか:

- さまざまなユーザークエリが予想されますが、それでもまとまりのあるドメインまたは製品領域内にあります。
- 特定のクエリや条件では、顧客データをいつフェッチするかを決定するなど、ツールの使用が正当化される場合があります。
- 決定論的チェーンよりも柔軟性が必要ですが、タスクごとに個別の専門エージェントは必要ありません。

利点：

- エージェントは、呼び出すツール(存在する場合)を選択することで、新しいクエリや予期しないクエリに適応できます。
- エージェントは、LLMの繰り返し呼び出しやツールの呼び出しをループして、完全なマルチエージェント設定を必要とせずに結果を絞り込むことができます。
- この設計パターンは、多くの場合、エンタープライズのユースケースのスイートスポットであり、マルチエージェントセットアップよりもデバッグが簡単で、動的ロジックと限られた自律性が可能です。

考慮 事項：

- ハードコードされたチェーンと比較して、ツール呼び出しの繰り返しや無効な呼び出しから保護する必要があります。(無限ループは、ツール呼び出しのシナリオで発生する可能性があるため、反復制限またはタイムアウトを設定します。
- アプリケーションが根本的に異なるサブドメイン(財務、DevOps、マーケティングなど)にまたがっている場合、1つのエージェントが扱いにくくなったり、機能要件が過負荷になったりする可能性があります。
- エージェントの集中力と関連性を維持するために、慎重に設計されたプロンプトと制約は依然として必要です。

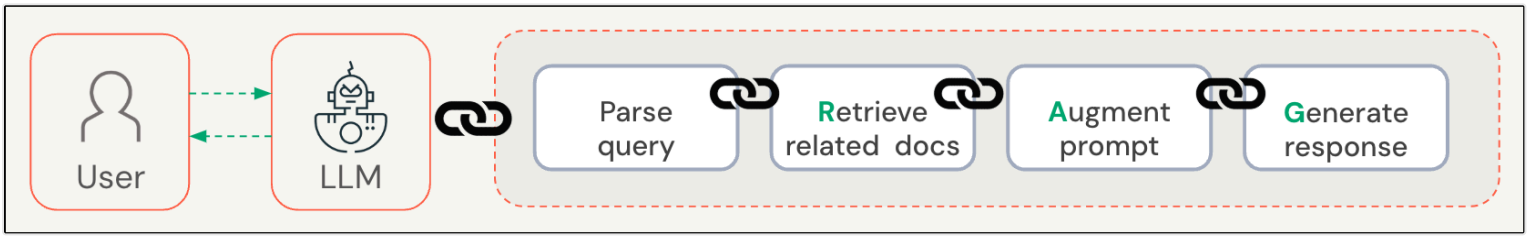
決定論的チェーン (ハードコードされたステップ)

このパターンでは、開発者は、どのコンポーネントを、どの順序で、どのパラメーターを使用して呼び出すかを定義します。 *どのツール* を *どの順序* で呼び出すかについての動的な意思決定はありません。システムは、すべてのリクエストに対して事前定義されたワークフローに従うため、非常に予測可能です。

一般に「チェーン」と呼ばれるフローは、基本的に次のような固定された一連のステップです。

1. 常にユーザーの要求を取得し、関連するコンテキストのベクトル インデックスから取得します。
2. そのコンテキストとユーザーの要求を組み合わせて、最終的な LLM プロンプトを作成します。
3. LLM を呼び出し、応答を返します。

例:基本的なRAGチェーン



決定論的なRAGチェーンは、常に次のことを行います。

1. 受信ユーザーの要求を使用して、ベクトル インデックスから top-k の結果を取得します (取得)。
2. 取得したチャンクをプロンプトテンプレートにフォーマットします (augment)。
3. その拡張されたプロンプトを LLM に渡します (生成)。
4. LLM の応答を返します。

いつ使用するか:

- 予測可能なワークフローを持つ明確に定義されたタスクの場合。
- 一貫性と監査が最優先事項である場合。
- オーケストレーションの決定のための複数の LLM 呼び出しを回避して待機時間を最小限に抑える場合。

利点：

- 最高の予測可能性と監査可能性。
- 通常、待機時間が短くなります (オーケストレーションのための LLM 呼び出しが少なくなります)。
- テストと検証が簡単にできます。

考慮 事項：

- 多様なリクエストや予期しないリクエストを処理するための柔軟性が限られている。

- ロジック分岐が大きくなると、複雑になり、保守が難しくなる可能性があります。
- 新しい機能に対応するために、大幅なリファクタリングが必要になる場合があります。

マルチエージェントシステム

マルチエージェントシステムには、メッセージを交換したり、タスクで協力したりする2つ以上の専門エージェントが含まれます。各エージェントは、独自のドメインまたはタスクの専門知識、コンテキスト、および潜在的に異なるツールセットを持っています。別の「コーディネーター」(別のLLMまたはルールベースのルーターなど)は、リクエストを適切なエージェントに送るか、あるエージェントから別のエージェントにいつハンドオフするかを決定します。

例: 専門エージェントを持つエンタープライズ・アシスタント

- **顧客サポートエージェント:** CRMの検索、返品、および配送を処理します。
- **アナリティクス エージェント:** SQLクエリとデータ要約に重点を置いています。
- **スーパーバイザ/ルータ:** 特定のユーザークエリに最適なエージェント、またはいつ切り替えるかを選択します。

各サブエージェントは、独自のドメイン (`lookup_customer_account` や `run_sql_query` など) 内でツール呼び出しを実行でき、多くの場合、独自のプロンプトや会話履歴が必要です。

いつ使用するか:

- コーディングエージェントや財務エージェントなど、明確な問題領域やスキルセットがあります。
- 各エージェントは、会話履歴またはドメイン固有のプロンプトにアクセスする必要があります。
- 非常に多くのツールがあるため、それらすべてを 1 つのエージェントのスキーマに適合させるのは現実的ではありません。各エージェントはサブセットを所有できます。
- 専門のエージェント間での反省、批評、またはコラボレーションを実装したいと考えています。

利点：

- このモジュラーアプローチは、各エージェントを、狭いドメインに特化した別々のチームによって開発または保守できることを意味します。
- 1 人のエージェントではまとまりのある管理に苦労する可能性のある大規模で複雑なエンタープライズワークフローを処理できます。
- 高度なマルチステップまたはマルチパースペクティブ推論を促進します - たとえば、1つのエージェントが回答を生成し、別のエージェントがそれを検証します。

考慮 事項：

- エージェント間のルーティングの戦略に加えて、複数のエンドポイント間でのログ記録、トレ-　　゙
  バッグのオーバーヘッドが必要です。

- 多くのサブエージェントとツールがある場合、どのエージェントがどのデータまたは APIにアクセスできるかを決定するのは複雑になる可能性があります。
- エージェントは、慎重に制約を受けなければ、解決せずにタスクを無期限にバウンスすることができます。
  - 無限ループのリスクは、単一エージェントのツール呼び出しにも存在しますが、複数エージェントのセットアップはデバッグの複雑さをさらに増やします。

実践的なアドバイス

どの設計パターンを選択するかに関係なく、安定した保守可能なエージェント システムを開発するための次のベスト プラクティスを検討してください。

1. **簡単に始めましょう:** 単純なチェーンのみが必要な場合は、決定論的チェーンをすばやく構築できます。
2. **徐々に複雑さを増していきます。** より動的なクエリや柔軟なデータソースが必要な場合は、ツールコールを備えたシングルエージェントシステムに移行してください。
3. **マルチエージェント化:** 明確に異なるドメインやタスクがある場合、複数の会話コンテキストがある場合、または1人のエージェントのプロンプトには大きすぎる大規模なツールセットがある場合に限ります。

ユースケースが小さなものから始める場合(単純なRAGチェーンなど)、ハードコードされたチェーンから始めます。要件の進化に応じて、動的な意思決定のためのツール呼び出しロジックを追加したり、タスクを複数の専用エージェントにセグメント化したりすることもできます。実際には、多くの現実世界のエージェント システムはパターンを組み合わせています。たとえば、主に決定論的なチェーンを使用しますが、必要に応じて LLM が 1 つのステップで特定の API を動的に呼び出すことができるようにします。

Mosaic AI Agent Framework は、どのパターンを選択しても依存せず、アプリケーションの成長に合わせてデザインパターンを簡単に進化させることができます。

開発ガイダンス

- **プロンプト**
  - プロンプトは明確で最小限に抑えて、矛盾する指示を避け、情報を散らし、幻覚を減らします。
  - エージェントが必要とするツールとコンテキストのみを提供し、無制限の API や無関係な大規模なコンテキストのセットを提供しないでください。
- **ロギングとオブザーバビリティ**
  - 各ユーザーリクエスト、エージェントプラン、およびツールコールの詳細なログを実装します。MLflow Tracing は、デバッグのために構造化ログをキャプチャするのに役立ちます。
  - ログは安全に保存し、会話データ内の個人を特定できる情報(PII)に注意してください。

- **モデルの更新とバージョンのピン留め**
  - LLM の動作は、プロバイダーがバックグラウンドでモデルを更新すると変化する可能性があります。バージョンの固定と頻繁な回帰テストを使用して、エージェントロジックの堅牢性と安定性を維持します。
  - MLflow と Mosaic AI エージェント評価を組み合わせると、エージェントのバージョン管理と、品質とパフォーマンスの定期的な評価を効率化できます。

テストとイテレーションのガイダンス

- **エラー処理とフォールバックロジック**
  - ツールまたは LLM の障害を計画します。タイムアウト、形式が正しくない応答、または空の結果があると、ワークフローが中断される可能性があります。再試行戦略、フォールバック ロジック、または高度な機能が失敗した場合のより単純なフォールバック チェーンを含めます。
- **反復プロンプトエンジニアリング**
  - 時間の経過とともに、プロンプトとチェーンロジックが洗練されることが予想されます。(Git と MLflow を使用して) 各変更をバージョン管理して、バージョン間でパフォーマンスをロールバックまたは比較できるようにします。
  - DSPyのようなフレームワークを検討して、エージェントシステム内のプロンプトやその他のコンポーネントをプログラムで最適化します。

本番運用 ガイダンス

- **レイテンシとコストの最適化**
  - LLM またはツール呼び出しが追加されるたびに、トークンの使用量と応答時間が増加します。可能な場合は、ステップを組み合わせるか、繰り返しクエリをキャッシュして、パフォーマンスとコストを管理しやすくします。
- **セキュリティとサンドボックス化**
  - エージェントがレコードを更新したり、コードを実行したりできる場合は、それらのアクションをサンドボックス化するか、必要に応じて人間の承認を強制します。これは、企業環境や規制された環境で意図しない損害を避けるために重要です。
  - Databricks では、サンドボックス実行に Unity Catalog ツールを推奨しています。Unity Catalog 関数ツールとエージェント コード ツールを参照してください。Unity Catalog を使用すると、任意のコードの実行を分離し、悪意のあるアクターがエージェントを騙して、他の要求を妨害したり盗聴したりするコードを生成および実行することを防ぎます。

これらのガイドラインに従うことで、ツールの誤呼び出し、LLM パフォーマンスのドリフト、予期しないコストの急増など、最も一般的な障害モードの多くを軽減し、より信頼性が高くスケーラブルなエージェント システムを構築できます。

2025年3月10日に最終更新

この記事は役に立ちましたか?