ChatGPT

高精度なエージェント型RAGシステムの戦略設計

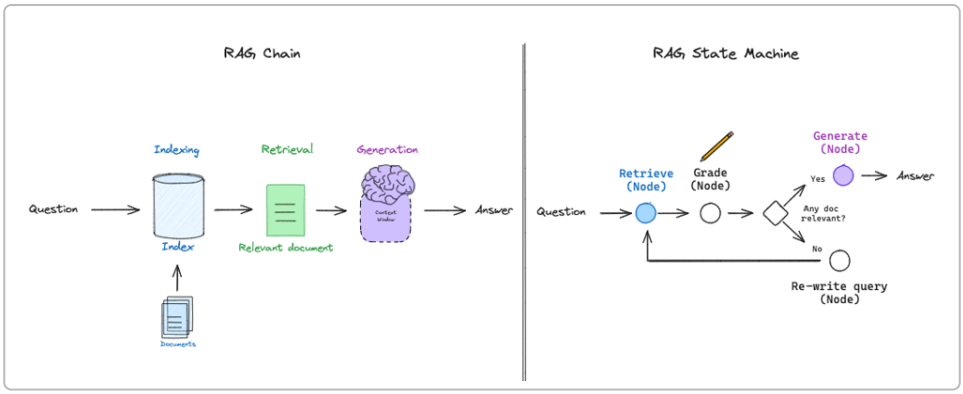
日本語PDF解析の最適化 (PyMuPDF vs pdfplumber)

● **PDFテキスト抽出:** 日本語PDFから正確なテキストを得るには、PyMuPDFとpdfplumberの使い分けが重要です。PyMuPDF (別名fitz)は高速かつ高精度なテキスト抽出が可能で、日本語の文字情報も比較的安定して取得できます [1]。一方でpdfplumberはレイアウト解析に強みがあり、ページ内のテーブル構造や段組みレイアウトを保った抽出が可能です [2]。日本語PDFではテキストが縦書きや複数カラムの場合も多いため、PyMuPDFで全体を抽出しつつ、表や図が含まれるページでは補助的にpdfplumberでレイアウトを解析するのがベストプラクティスです。なお、pdfplumberは非ラテン文字で文字抜けが発生する既知の課題もあり [3]、両者を併用して結果を突合することで精度を高められます。

● **テーブルデータの抽出:** テーブル（表）は日本語ビジネス文書で特に重要な情報源ですが、単純なPDFテキスト抽出ではセルの境界が曖昧になり、隣接セルが結合されたり関係性が失われたりします [4]。これを防ぐため、PDF中の表は**Markdown形式**などに変換して構造を保持するのが有効です [5]。例えば、PyMuPDFでテキスト抽出後にセル間の区切りを検出し、`|`や改行を用いてMarkdownの表形式に整形します。あるいはpdfplumberの`extract_table`機能を用いてセル配列を取得し、そのままCSV/Markdownにする方法もあります。重要なのは**表全体を一つのチャンク**として扱うことです。チャンク分割時に表が分断されると意味が伝わらないため、正規表現等で「表の開始〜終了」を検知して一塊で保持します [5]。このように前処理でテーブルを構造化しておくことで、Embeddingによる検索でも数値や項目名を含むベクトルとして保持でき、質問時に関連セルを見落とす可能性を下げられます。また、最新のツールではChatDOCのようにテーブルを周辺の説明文とセットで抽出し、行・列情報を保持するアプローチも報告されています [6]。これはEmbedding時の意味情報が豊富になり、ユーザ意図にマッチする確率が向上します。

● **図や画像の解析:** PDF内の画像については、まずPyMuPDFで画像オブジェクトを直接抽出し、ファイルとして保存できます。その上で、もし画像内にテキストが埋め込まれていればOCR（例えばTesseractやEasyOCR）でテキスト化しておき、本文テキストに追記します。グラフや写真で文字がない場合は、その**キャプション（図番号や説明文）**を抽出しておくことが重要です。図表のキャプションやタイトルに質問のキーワードが含まれる場合、それを手掛かりに**該当ページをハイライト表示**できるようになります。AllganizeのRAG評価でも、Parser（パーサ）が「文字・図・表を抽出し読みやすいフォーマットに変換する」能力を独立評価しており [7]、図表情報を無視しないことが高精度回答の前提条件となります。実装上は、PyMuPDFで抽出したテキストにノイズ（不要な改行やページ番号など）が多い場合、正規表現で除去し、日本語の句点や読点で適切に文を繋げてリファインする処理も有効です。これによりEmbeddingの質が向上し、後段の検索精度が安定します。

Agentic RAGのステートマシン設計 (LangGraph + GPT-5-mini)



エージェント型RAG（Agentic RAG）は、従来の「検索→回答」の直線的フローを拡張し、**動的な思考判断**を組み込んだステートマシンで質問に挑みます [8] [9]。特に今回は推論に使用できるモデルが軽量な`gpt-5-mini`に限られるため、LangGraphを用いて**決まった手順内で最大限の思考を引き出す設計**がポイントです。以下に安定動作するフローの一例を示します。

- **① 質問の解析（Query Analysis）：**まずエージェントがユーザ質問を解析し、その**種類や複雑さ**を判断します [10]。例えば質問内容から「単純な事実質問」「複数の情報統合が必要」「最新情報の有無確認」などに分類します。この際、小型モデルでも対応できるようプロンプトを工夫します（例：「この質問は社内データだけで答えられますか？要素に分解すると？」といった指示を与え、Yes/Noや箇条書きで出力させる [10]）。質問の解析結果に応じて次のステップを**ルーティング**します。LangGraphではこのルーティングを条件エッジで表現でき、`StateGraph.add_conditional_edges`で分岐を定義します。具体的には:

```
def route_by_question_type(state: GraphState) -> str:
    q_type = state["question_type"]  # LLMが与えた質問分類結果
    if q_type in ["simple", "known"]:
        return "direct_retrieval"
    elif q_type == "complex":
        return "multi_step_retrieval"
    elif q_type == "current":
        return "web_search"
    else:
        return "direct_retrieval"
graph.add_conditional_edges("analyze_question", route_by_question_type,
                            {"direct_retrieval": "vec_retrieve",
                             "multi_step_retrieval": "vec_retrieve_iterative",
                             "web_search": "web_retrieve"})
```

ここでは例として、**シンプルな質問**なら通常のベクトル検索、**複雑な質問**ならマルチステップ検索（後述）、**最新ニュースなど時事**ならウェブ検索ツールへ、という3分岐を設定しています [11] [12]。このように事前に質問を分析することで、無駄な思考ループを省き小モデルでも効率的に動作させることができます。

- **② 適応的な検索戦略（Adaptive Retrieval）：**エージェントは質問に応じた最適な検索方法で情報取得を試みます [10]。基本はQdrantベクトルDBから関連文書を取得しますが、**工夫として複数検索クエリの発行**と**逐次的な改善**を組み込みます [13] [14]。例えば「質問を言い換えたサブクエリ」をLLMに3種類ほど生成させ、**並列してベクトル検索**を行います [15]。LangGraphでは複数のRetrievalノードを

並列に配置し、検索結果を一つのリストにマージするようなState管理が可能です [16] [17]。これにより、一度の質問で同義語や関連切り口から幅広く文脈を集め、**「単一クエリで見逃す情報を減らす」**効果があります [13]。

また、検索結果に対して**軽量な評価（Retrieval Evaluator）**を挟むのも有効です [18]。例えば、`gpt-5-mini`に簡潔なプロンプトで「この検索結果群は質問に答えるのに十分関連していますか？（Yes/Noで）」と尋ね、Confidenceスコアを付けます [18]。LangGraphではこれを**ドキュメント評価ノード**として組み込み、評価結果が低ければ別経路（例えばWeb検索ノード）に遷移させることも可能です [19]。一例としてCorrective RAG (CRAG)手法では「取得文書に無関連なものが1つでもあればWeb検索を追加実行する」戦略を取っています [20]。今回のように社内ドキュメントが主ですが、どうしても見つからない場合は一般Webから補足情報を取る設計も**保険**として考えられます。特に小型モデルではベクトル検索の結果が多少ずれてもそれを自力で補正できないため、**検索段階でのアダプティブな工夫**が精度に直結します。

さらに**Iterative Retrieval**にも対応します。例えば複雑な質問の場合、最初の検索結果から部分的な答えを得てもまだ不足していれば、**追加の再検索**を行います [21]。LangGraphでループを作るには、`graph.add_edge("nodeX", "nodeY")`を適切に設定し、ループ脱出条件をエッジ条件にします。疑似コードで示すと:

```
graph LR
  A[初回検索] --> B[回答生成]
  B --> C{回答十分?}
  C -- 不十分 --> D[クエリ改善]
  D --> A
  C -- 十分 --> E[回答確定]
```

上記のように**検索→回答→評価→再検索**のループを1回ないし必要に応じて複数回回すことができます。実装上は、一度目の回答生成後に**自己改善ノード**で回答を評価し [22]、「不足している情報」や「追加で参照すべき文書のヒント」を小モデルに生成させます。例えば「回答に足りないのはどんな情報か？次にどの資料のどんな部分を探すべきか」を考えさせ、その内容で再度ベクトル検索クエリをリライトします [21]。LangGraphでは**状態 (`GraphState`) に現在のiteration回数や前回の回答内容**を保持できるので、2回目の検索ではそれらを入力に含めることで**文脈を加味した検索**が可能です。「前の回答では○○が曖昧なので、△△に関するデータを再度探す」といった**エージェントの自主的な再検索**が、より正確な根拠集めに繋がります [10] [21]。注意点として、`gpt-5-mini`のトークン予算を考慮しループ回数は過剰にしない（最大1回の改善に留める等）、評価もYes/Noや箇条書きなど**簡潔なフォーマット**で出力させるようにします [23]。特にLangChainのPydantic出力パーサを使えば、例えば「relevant\": true/false」のJSONを吐かせるだけで判断でき、小型モデルでもミスが少なくなります [23]。こうした状態機械的なフロー制御によって、軽量モデル上でも**安定して自律的な情報検索→回答改善**が行えるのがLangGraph＋Agentic RAGの強みです [24]。

- **③ 回答生成と自己反省 (Generation & Reflection):** 最終的な関連コンテキストが集まったら、LLMに回答生成させます。プロンプトテンプレートはシンプルに「# 文脈: {context}\n# 質問: {question}\n# 回答:」という形式で十分ですが、**表形式のデータが含まれる場合はMarkdownで整形して提示**すると良いです（モデルが表を読み取りやすくなるため）。生成された回答については、別のエージェント（あるいは同じ`gpt-5-mini`にロールを変えて）で**自己評価**させます [22]。評価基準はAllganizeの評価指標に合わせ、「正確性 (質問に正しく答えているか)」「網羅性 (必要な情報が欠けていないか)」「根拠との合致 (出典コンテキストに基づいているか)」などを1〜5段階でスコアリングします。実装上、Systemメッセージに評価者の視点を与え、Assistantとしてスコアのみ出力させることで簡易なスコアリングが可能です。

Reflectionの結果スコアが低ければ、**改善アクション**をエージェントが検討します。例えば「スコア3/5、不正確: ○○の数値が異なります」とフィードバックが返ってきたら、それをStateに記録し、先述のループに戻

して**クエリリライト**を行います（「○○に関する表を再検索する」等）[25]。LangGraphではこのフィードバックを状態に持たせたまま再度フローを実行できるため、モデルが**どの点を修正すべきか理解した上で回答をアップデート**できます[22]。なお、小型モデルでは自己評価自体の信頼性も高くない場合があるため、可能であればここだけOpenAIの高性能モデル（例えばGPT-4相当）をスポットで使うのも現実解です。ただし予算制限が厳しければ、`gpt-5-mini`で簡易基準の評価 (Yes/No判定や1-5スコア) を行い、人手でスポットチェックする運用も検討すべきです。

以上のように、エージェントの**思考ステートマシン**を構築することで、「検索する・しない」「どのデータソースを使う」「結果が不十分なら再試行」といった判断をすべてLLM任せにせずフローとして保証できます[26]。特にLangGraphはステートフルな情報共有とループ・条件分岐をサポートするため[24]、`gpt-5-mini`単体では難しい高度な推論も、**適切に誘導された段階的処理**で実現可能になります。これが**Naive　RAGを超える正答率**を叩き出す秘訣です。

表(Table)・画像(Image)を含む情報への対応策

日本語RAGデータセットでは、回答根拠が段落テキスト以外に**表**や**画像**に含まれるケースが多々あります。この場合、単純なテキストベースのベクトル検索だけでは限界があるため、Agentic RAGとして工夫すべきポイントを述べます。

● **表データへのアクセス:** 前述の通り、PDF解析段階で表をMarkdown化しておくことは大前提です。その上で、**表質問専用の検索戦略**を導入します。例えば、ユーザ質問に「〜の表では...」「表中の数値...」といったキーワードがあれば、エージェントは通常のテキスト検索に加えて**メタデータフィルタ**や**正規表現検索**を使って該当テーブルを直接探しに行きます。具体的には、ドキュメントをインデックス化する際に各チャンクに`metadata={"type": "table"}`のような属性を付与し、質問解析で「表参照」が推測されたらVector DBの検索時に`filter={"type": "table"}`を適用します。これにより通常段落よりもテーブルチャンクを優先して取得できます。取得後、LLMに渡す前に**表のヘッダ行やキー列**をもう一度ハイライトするようプロンプトを整形すると、モデルが数値の意味を取り違えにくくなります（例えば：「以下は○○の表です。各列のヘッダは…」と説明を添える）。

また、表質問では**直接計算や比較**が必要なケースもあります。高度な戦略として、最新研究のTableRAGに倣い、**表を一種のデータベースとしてクエリする**方法も考えられます。TableRAGでは、テーブルを構造化データベース（例：SQLite）に変換し、エージェントが質問から適切なSQLクエリを自動生成してテーブルから直接答えを取得する手法が提案されています[27]。例えば「売上表から2023年Q1の合計を求めよ」という質問に対し、`SELECT`文を生成して値を合算するようなプロセスです。このアプローチを実装すれば、LLMが表データを誤読するリスクや計算ミスを大幅に減らせます。ただし、今回の課題スコープでSQL実行エージェントまで作り込むのは現実的でない場合、**疑似的に計算**させる工夫でも対応可能です。例えば、表が抽出されたテキストをそのまま渡さず「表中の該当箇所を抜き出して回答せよ」と指示し、モデルに必要セルだけ抽取・計算させるようなfew-shotプロンプトを用意します（1件例示して「このように答えて」と示す）。いずれにせよ、表付きの文書では**表全体を正しくEmbeddingし、必要ならばプログラム的アプローチを組み合わせる**ことで、段落ベースでは30%しか正解できなかった問いにも対応できるようになります[28]。実際、Allganize社のソリューションでも「特許出願中の技術で表から高精度回答を生成」と言及されており[29]、表に対する特化戦略が性能差を生んでいます。プロ仕様のRAGでは、この部分への対策が合否を分けるでしょう。

● **画像情報へのアクセス:** 画像も表同様に、索引作成時に`metadata={"type": "image"}`を付けたり、キャプション文字列を別途Embeddingするなどして検索可能にしておきます。質問が「図を参照」している場合、該当キャプションや周辺説明文が引っ掛かるよう、**画像周辺テキストを重点的にEmbedding**します。例えばPDFから画像ファイル名や「Figure 3: XXX」の説明文を取得し、それ自体を一つのドキュメントとしてVector DBに登録します。こうすることで、「図3に示されたグラフの傾向は？」といった質問でも**関連する説明文**を引ける可能性が上がります。さらに、Agentic RAGなら**画像内容を要約するツール**を組み込むことも検討でき

ます。もし図がグラフで数値軸が読み取れるなら、OCRで数値データを抽出→簡単な説明文にする、といったカスタムツールをエージェントに持たせるのです。LangChainのツール機能をLangGraphに組み込み、`tool_name="GraphOCRTool"`のようなノードを用意しておけば、LLMが「画像から読み取る必要あり」と判断した際にそのツールを呼び出す動作も可能です。もっとも`gpt-5-mini`では画像内容の高度な説明は困難なので、そこまで凝らずとも**「画像あり」質問だと検知したら、事前に用意した画像説明を出力に含める**くらいのロジックでも十分差別化できます。

● **マルチモーダルEmbedding:** 余談ですが、Vector DB側でテキストと画像を**マルチモーダル埋め込み**する手もあります。例えばTCLIPや統合Transformerを用いて画像をベクトル化し、テキストと同じ空間で近傍検索する技術も登場しています [30]。今回手元のembeddingモデルはtext-embedding-3-smallとテキスト限定ですが、将来的に図版の内容もEmbeddingできれば「画像そのもの」を検索して根拠取得→それをLLMに説明させることが可能になるでしょう。現時点ではまず**図表をテキストに変換し尽くすこと**が最優先ですが、エージェントに画像解析AIを組み合わせる拡張性も頭に入れておくと、よりプロアクティブなRAGシステム設計になります。

デバッグ機能付き評価パイプラインの設計 (Evaluator実装)

高精度なRAGシステムを完成させるには、自身の開発中に**弱点を可視化し分析するための評価パイプライン**が不可欠です。AllganizeのLeaderboardでは、正解回答との類似度評価やLLMによる正誤判定（Voting）を用いて自動評価しています [31]。これに倣いつつ、さらに「なぜ間違えたか？」まで解明できるダッシュボード的機能を備えた評価基盤を構築しましょう。ポイントは以下の4点です。

1. **検索根拠のトレーサビリティ:** 各質問に対し、エージェントが**どのドメインのどの文書の何ページからどの段落を引いたか**を記録・表示します。具体的には、ドキュメントごとにメタデータとして`{"domain": "金融", "doc": "例.pdf", "page": 5, "chunk_index": 12}`のような情報を持たせ、回答に引用したコンテキストにはそのメタ情報を埋め込みます。回答後の評価段階でこのメタ情報を集約し、UI上で「**出典**」としてハイパーリンク付きで提示します。例えば、「出典:　　金融ドメイン/例.pdf 5ページ【該当箇所をハイライト】」という具合です。ユーザから見れば、回答の根拠箇所を一目で参照でき、信頼性を検証できます。また開発者視点では、**正解のページを引けていたのに回答を間違えた**のか、**根拠となるページ自体を外していた**のかが区別できます。前者であれば読解/生成ステップの問題、後者なら検索ステップの問題なので、改善アプローチが異なります。例えば、後者が多ければEmbeddingやクエリ改善ロジックを見直す必要がありますし、前者が多ければLLMプロンプトや出力形式（例えば表読み取り能力）を改善する必要がある、といった分析が可能になります。

2. **エージェントの思考ログ (Trace) の可視化:** LangGraphおよびLangChainには実行トレースを取得する仕組みがあります。LangSmith等を使えば、各ノードでLLMがどんなメッセージをやりとりし、どのツールを呼び出したか記録できます。これを**時系列のフローチャート**として表示することで、エージェントの判断過程を再現可能にします。例えば「Q1:　質問分析→ベクトル検索実行→回答生成→自己評価(不十分)→クエリ書き換え→再検索→回答更新→完了」という一連の流れをログツリーで表示します [32] [33]。具体的な実装としては、LangGraphの各ノード関数内で`logger.info`等でステップ名と要約を出力し、それを収集してHTML上にタイムライン表示するようにします。またはLangSmithのトレースJSONをパースして可視化しても良いでしょう。これにより、**どの段階で判断ミスが起きたか**（例えば「第一検索で本来不要な文書を選択してしまった」「自己評価が甘く誤答をそのまま出力した」等）が突き止めやすくなります。特にAgentic RAGでは分岐やループがあるため、ブラックボックスにしないことが重要です。思考ログを見れば**エージェント内のLLMの内的判断**（検索結果へのコメントや、再検索理由）が分かり、調整すべきプロンプト箇所を的確に見つけられます。

3. **エラー原因の自動分類:**　回答が不正解だった場合、その原因をエージェント自身に分析させ、自動でタグ付けする機能を組み込みます。具体的には、評価段階で別のLLM（もしくは同モデル）に以下のような指示を与えます:「質問・生成回答・正解・参照コンテキスト」を渡し、「誤答の原因」を次か

ら選んで出力: 検索失敗(正解載ってるページ未取得)、読解失敗(ページ在ったが解釈ミス)、幻想(根拠無い内容を生成)、不完全(一部は合っているが抜け漏れ)」。 LLMに自己分析させることで、人手で分類する手間を省きます。例えば正解が「表の数値AとBの差は10」で、モデル回答が「差は15」だった場合、コンテキストには正しいA,Bが含まれていた→読解ミス、と分類できます。逆にコンテキスト自体にA,B載ってなければ→検索ミス、という具合です。この判断ロジックはルールベースでも部分的に可能です。例えば**正解アノテーションとして各質問に「該当ページ」情報**がデータセットに含まれているなら [34] [35]、それとエージェントの取得ページを比較し未取得なら自動で「検索失敗」とタグ付けできます。残りの分類も、回答文と正解文の類似度（ベクトル類似やキーワード一致）や、回答文と参照コンテキストの含有率チェックなどである程度ルール判定可能です。ただ、小型LLMにテキスト比較させて決めさせる方が実装コストは低いでしょう。これらのエラー分類結果は、最終的に**統計レポート**として集計し、「どのタイプのミスが多いか」を分析します。例えば「検索失敗が全体の60%を占める」と分かればRetriever強化が急務ですし、「幻想型が多い」ならプロンプトで根拠外発言を抑制する措置（例えば引用ルール徹底など）を追加する、といった具合に対策に繋げられます。

4. **コンテキスト形式別の集計:** 最後に、各質問の根拠コンテキストが**段落か表か画像か**をラベル付けしておき、正解率をタイプ別に集計します。これにより「段落ベース質問の正解率は90%だが、表ベースは30%しかない」等の性能偏差が数値で把握できます。データセットに各QAペアの根拠種別ラベルが無い場合でも、正解テキストを解析して簡易に推定できます（例：正解文に数字が多ければ表由来、○○図とあれば画像由来など）。評価パイプラインで自動ラベル付けし、最終的に**ヒートマップや棒グラフ**でタイプ別精度を可視化するとよいでしょう。こうした分析結果は、開発サイクルの中で「次に何を改善すべきか」を定量的に示してくれます。例えば表質問だけ極端に低ければ、前述のTable対策に注力すべきですし、段落は高いのにまだ全体精度が80%程度なら他の要因（ループの有効性や自己評価プロンプト）がボトルネックかもしれません。

以上のような評価パイプラインは、一種の**RAGダッシュボード**として機能します。質問→回答の生成結果だけでなく、その背後でエージェントが何を参照しどう判断したかまで追跡できるため、単なるスコア以上の学びが得られます。実装には多少手間がかかりますが、最終的に採用担当者へ提出する際にも、このような**分析可視化ツール込み**で成果物を示せれば「プロダクションレベルでの品質管理まで考慮している」と高く評価されるはずです。

トークン予算内で思考の質を担保するプロンプト工夫

最後に、限られたOpenAI API予算（約\$30）で高度なAgentic動作を実現するための**プロンプト最適化戦略**について述べます。小型モデル `gpt-5-mini` を多数回呼び出す場合、各呼び出しで無駄なトークンを極力省きつつ、必要な思考プロセスは犠牲にしない工夫が求められます。

● **システムメッセージの活用とFew-shot最小化:** プロンプトの基本方針として、**Systemメッセージに包括的な指示を書き込み、Userには具体的質問やデータのみ渡す**形にします。Systemに「あなたは優秀なアシスタント。以下の手順で考えなさい…」と役割思考を詰め込んでおけば、毎回のUserプロンプトで同じ指示を繰り返す必要がありません。LangChainなら一度定義したSystemプロンプトを会話に維持できます。またFew-shot例示も、必要最低限の1件だけに留めます。例えばクエリを書き換えるエージェントなら、**1つの例だけ**「Q: ... → Rewritten Q: ...」のように示し、後はモデルにパターンを汲ませます。複数例を与えるとそれだけで数百トークン消費するため、**「ワンショット提示 + 要点の抽出」**で十分です。必要なら訓練済みの方策をSystemに文章で書いても良いでしょう（例：「もし検索結果が関連薄なら、質問を言い換えて再検索してください」など）。こうした**プロンプト内方策エンジニアリング**でFew-shotをテキストルールに置き換えると、大幅なトークン節約になります。

● **スクラッチパッドの圧縮:** エージェントが段階的に思考するReActスタイルでは、本来LLMがツール使用計画や中間結論を**長々と説明**することがあります。しかし小型モデルでそれをやらせると、トークン浪費の割に信頼度も低い推論が出る恐れがあります。そこで、LangGraphの設計段階で**中間出力を極力構造化**し、モ

デルに冗長な独白をさせないようにします。例えば「検索結果評価」のノードでは、モデルに「各ドキュメントについてRelevant or IrrelevantをJSONで返せ」と指示すれば、`[{"doc": 1, "eval": "Relevant"}, ...]`程度の短い出力で済みます[23]。これを文字で「Doc1は質問と関連があります。Doc2は関係ありません…」と書かせると一気にトークンを消費します。同様に、自己反省のフィードバックも簡潔なフォーマットで出させ、次の入力にそのまま使えるようにします。**チェイン・オブ・ソート(Chain-of-Thought)の圧縮**とはつまり、モデルの思考結果を人間可読な文章ではなく機械可読な要約で受け取り、それを次のステップに使い回すことです。LangGraphのステートで保持する情報も、長文ではなくキーとなる単語やスコアだけ保持すれば、次回LLM呼び出し時にSystemやUserメッセージに詰め込むトークンを抑えられます。

● **条件付きでステップ省略:** 予算節約のためには、「常に同じフローを回さず、必要なときだけ追加思考する」というメリハリも重要です。例えば**Query Analyzer**が「これはシンプル質問」と判定した場合、自己反省ループはスキップして即回答を返す設定にします。LangGraph上では、分析ノードから直接回答生成ノードへENDに繋ぐエッジを用意し、Reflectionノードをバイパスできます。同様に、検索結果の評価が高スコアならクエリリライトを飛ばす、といった**条件付きスキップ**を組み込むのです。これにより平均ステップ数が減り、トークン使用総量も下がります。実際、Adaptive RAGの考え方では質問の難易度に応じて「**Retrievalなし**」「**一回Retrieval**」「**繰り返しRetrieval**」を使い分けています[36][37]。このようなフロー分岐を明示的に設計することで、軽量モデルでも**必要十分な場合は余計なループを回さない**スマートな挙動が可能になります。

● **OpenAI APIの活用と監視:** OpenAIのAPIでは、`gpt-5-mini`相当であれば比較的安価でしょうが、トークン数×リクエスト回数が積もると\$30に届く可能性があります。これを防ぐため、**ベクトル検索などLLM不要部分は極力Pythonで処理**し、LLM呼び出し回数を減らします。また、OpenAIの`max_tokens`を適切に設定して**無駄な長文回答を出させない**こと、`temperature=0`で確定的に振る舞わせ**再試行によるトークン浪費を防ぐ**ことも重要です。さらにLangChainの`CallbackManager`等を用いて各呼び出しのトークン使用量をログり、進捗に応じて動的にフローを簡略化することも考えられます（例えば、すでに予算の80%を使ったらReflectionをスキップする等の措置）。

● **圧縮プロンプトの例:** 最後に、思考プロセスの圧縮を実現するプロンプト構造の例を示します。自己評価と再検索提案を**一度のLLM呼出でまとめて**行う方法です。通常は「回答生成→評価→提案」と2段階に分けますが、小型モデルでは一度で済ませた方が対話コストが減ります。例えば以下のようにします:

```
**System:** あなたは厳密な評論家兼助手です。まずユーザ質問に対する回答を与え、その後にその回答の評価(5点満点)と改善提案をJSONで出力しなさい。
**User:** 質問: ...\n[コンテキスト情報]\n(まず回答し、続けて評価してください)
```

このプロンプトに対し、モデルは**回答→{"score":3, "reason":"○○が不足", "suggestion":"△△を参照せよ"}**のように一度に出力します。パーサーでJSON部分を抽出し、スコアが閾値未満なら`suggestion`を用いて追加検索・回答改善を行います。こうすれば**1ターン分のトークンで回答とその評価・計画を済ませる**ことができ、効率的です。もっとも、プロンプトが複雑になるほど小型モデルは破綻しやすいので、この例では回答と評価を明示的に区切る工夫（例えば回答後に特殊トークンを書くルールなど）が必要でしょう。

以上のように、**思考ステップの取捨選択**と**出力フォーマットの工夫**によって、トークンコストを最小化しつつエージェントの賢さを引き出すことが可能です。特に**「小さく賢く産む」**発想でプロンプトを設計できれば、予算内でも120点満点の性能を目指せるでしょう。実装段階では細かなチューニングの連続になりますが、その過程自体がLLMエンジニアとしての腕の見せ所です。ぜひこれら戦略を盛り込み、採用担当者が唸るような完成度のRAGシステムを構築してください。成功を祈っています！[26][38]

---

1 Which is faster at extracting text from a PDF: PyMuPDF or PyPDF2?
https://www.reddit.com/r/learnpython/comments/11ltkqz/which_is_faster_at_extracting_text_from_a_pdf/

2 How to Extract Text from PDF in Python - YouTube
https://www.youtube.com/watch?v=Ddk8bA6OWjQ

3 pdfplumber characters missing (for Chinese character) · Issue #1022
https://github.com/jsvine/pdfplumber/issues/1022

4 5 6 28 Improving table extraction of enterprise documents in RAG systems : r/Rag
https://www.reddit.com/r/Rag/comments/1lpp417/improving_table_extraction_of_enterprise/

7 29 31 ■お知らせ■ 日本語RAG性能を評価した「Allganize RAG Leaderboard」を本日公開
https://blog-ja.allganize.ai/rag_leaderboard/

8 Agentic RAG With LangGraph - Qdrant
https://qdrant.tech/documentation/agentic-rag-langgraph/

9 10 21 26 38 LangGraphで学ぶAgentic RAG解説
https://zenn.dev/egghead/articles/22966db52b604d

11 12 36 37 Adaptive RAG | LangChain OpenTutorial
https://langchain-opentutorial.gitbook.io/langchain-opentutorial/17-langgraph/02-structures/07-langgraph-adaptive-rag

13 14 15 16 17 22 24 25 LangGraphで作る賢いRAGエージェントの実装方法｜BRANU 開発部広報
https://note.com/branu_dev_pr/n/n52fe230b8b79

18 19 20 23 32 33 Self-Reflective RAG with LangGraph
https://blog.langchain.com/agentic-rag-with-langgraph/

27 TableRAG: A Retrieval Augmented Generation Framework for Heterogeneous Document Reasoning
https://arxiv.org/html/2506.10380v1

30 Scaling RAG QA with Large Docs, Tables, and 30K+ Chunks
https://community.deeplearning.ai/t/scaling-rag-qa-with-large-docs-tables-and-30k-chunks/826199

34 35 RAG-Evaluation-Dataset-JAに関する備忘録 #RAG-Evaluation-Dataset-JA - Qiita
https://qiita.com/onoyu1012/items/88dfa8ed0827325e4536