# 現代的プロンプトエンジニアリング実践ガイド：LLMの性能を最大化する最新技術

## I. はじめに

大規模言語モデル（LLM）の能力を最大限に引き出す鍵となるのが、プロンプトエンジニアリングです。これは、LLMに対して与える指示（プロンプト）を戦略的に設計し、最適化するプロセスを指します [1]。AI技術、特にLLMの急速な進化に伴い、プロンプトエンジニアリングの手法もまた、目覚ましいスピードで変化しています [2]。かつて有効とされたテクニックが陳腐化し、新たなアプローチが登場する中で、現在最も効果的な戦略を把握することは、開発者や研究者にとって不可欠です。

このレポートでは、2024年から2025年にかけて有効とされる、最新かつ実践的なプロンプトエンジニアリング技術に焦点を当てます。古くなった、あるいは効果の低いとされる手法や例を除外し、現在主流となっているベストプラクティス、主要なプロンプト技術（Zero-Shot、Few-Shot、思考の連鎖、役割設定、構造化出力、自己反省など）、そしてLLMの多様性への対応といった側面を網羅的に解説します。目的は、読者がLLMとの対話を通じて、より高品質で信頼性の高い結果を得るための知識とツールを提供することにあります。

## II. 効果的なプロンプト設計の基礎原則

LLMから望ましい出力を得るためには、明確で効果的なプロンプト設計が不可欠です。最新のLLMは進化し続けていますが、その性能を安定して引き出すための基本的な原則は依然として重要です。ここでは、現在有効とされるプロンプト作成のベストプラクティスを解説します。

### A. 明確性、具体性、文脈の重要性

LLMは、曖昧な指示よりも明確で具体的な指示を与えられた場合に最高のパフォーマンスを発揮します [4]。プロンプトを作成する際には、以下の要素を意識することが重要です。

- **目標の明確化:** プロンプトの目的、つまり「何のために」「どのような結果を求めているか」を明確に定義します [2]。例えば、「10歳の子どもに量子力学の基本概念を分かりやすく説明する」といった具体的な目的と、「専門用語を避け、日常的な比喩を使う」「小学生が理解できる平易さ」といった成功基準を設定することが有効です [2]。
- **指示の具体性:** 実行してほしいタスクを具体的に指示します。「この文書を要約して」のような曖昧な指示ではなく、「この文書を、議論されている主要な課題に焦点を当てて3つの箇条書きで要約してください」のように具体的に記述します [4]。
- **文脈（コンテキスト）の提供:** LLMがタスクを正確に理解し、適切な応答を生成するためには、十分な文脈情報が必要です [4]。これには、背景情報、関連データ、期待されるトーンやスタイル、ターゲットオーディエンスに関する情報などが含まれます [5]。例えば、顧客サポートの応答を生成する場合、問題の説明と望ましいトーン（例：共感的、丁寧）を提供します [4]。

- 制約条件の設定: LLMの出力が冗長になったり、関連性のない情報を含んだりするのを防ぐために、制約条件を設定することが有効です [4]。文字数制限（例：「100語で説明してください」）、出力形式の指定（例：「JSON形式で回答してください」）、使用すべきでない言葉の指定などが考えられます [4]。
- 指示と文脈の分離: プロンプト内で指示、文脈、入力データ、出力形式指定などの異なる要素を明確に区別することが、LLMの理解を助けます [1]。

近年のLLMの進化により、一部のタスクでは非常に詳細な指示よりも、明確で簡潔なプロンプトが効果を発揮するケースも報告されています [2]。例えば、「深呼吸して」といったシンプルな合図が、複雑な手法よりも良い結果をもたらすことがあるとされます [2]。しかし、これはタスクの性質やモデルの能力に依存する側面が強いと考えられます。特に、複雑なタスクや特定の形式・精度が要求される場合においては、依然として明確性、具体性、そして十分な文脈を提供することが、高品質な出力を得るための鍵となります。基礎的な指示が明確であれば、モデルはより少ない例示でパターンを理解する能力も向上しています [2]。

**B.** 構造化プロンプトの活用

複雑な要求をLLMに伝える際には、プロンプト自体を構造化することが非常に有効です。これにより、モデルは指示の各部分を正確に解釈しやすくなります。

- 区切り文字の使用: プロンプトの異なるセクション（指示、文脈、入力データ、例など）を区切るために、###や"""""""のような明確なマーカーを使用します [5]。これにより、モデルはプロンプトの構造と意図をより効果的に理解できます。
- **XML**タグ形式: 特に複雑なリクエストの場合、XMLスタイルのタグを使用してプロンプトを構造化するアプローチが推奨されています [5]。これは、XMLの属性と要素のツリー構造を模倣し、タスクに合わせて適用するものです。この方法により、モデルはプロンプトの特定のセクション（役割、プロセス、タスク、応答指示など）を容易に識別し、重要な指示や提供された推論ステップに従うことができ、全体的な応答生成能力が向上します [5]。
  **XML**タグ形式の例 [5]:
  XML
  ```
  <role>
  あなたは環境に優しい商品の魅力的な商品説明を作成する専門のコピーライターです。
  </role>
  <process>
  <step1>入力で提供された商品の特徴を分析してください。</step1>
  <step2>創造的で説得力のある商品説明を作成してください。</step2>
  <step3>購入を促す行動喚起で締めくくってください。</step3>
  </process>
  <task>
  環境に優しい水筒の商品説明を100～150語で書いてください。
  特徴:
  - 容量:750 ml
  ```

- 二重壁断熱（飲み物を24時間冷たく、12時間温かく保つ）
- BPAフリー、漏れ防止
- 複数色展開
- 100%リサイクル可能なパッケージ
</task>
<responseInstructions>
商品説明はプロフェッショナルで魅力的なトーンで書いてください。
商品の環境に優しい側面を強調し、主要な特徴を含めるようにしてください。
応答は次のタグで囲んでください：<finalResponse>...商品説明...</finalResponse>。
タグの外には何も含めないでください。
</responseInstructions>

- 論理的なフローとタスク分解: 複雑なタスクは、プロンプト内でより小さな、順序立てられたステップやサブタスクに分解することが推奨されます[2]。これにより、LLMはプロセス全体をより正確に理解し、実行することができます。例えば、コードレビューを依頼する場合、単にコードを提示するのではなく、「次のデータ処理関数をレビューしてください：まず基本的な構造と目的を理解し、次に正確性、パフォーマンス、可読性、エラー処理の観点から詳細分析を行い、具体的な改善提案を優先度順にリストアップしてください」といった段階的な指示が有効です[2]。同様に、REST APIの作成を依頼する場合も、サーバー設定、ルート作成、エラーハンドリングなどのステップに分解して指示します[6]。
- プロンプト要素の統合: 文献で指摘されている主要なプロンプト要素（タスク指示、背景/文脈、入力データ、出力形式、役割、スタイル、例示）を理解し、必要に応じて組み合わせることが重要です[1]。ただし、全てのプロンプトに全ての要素が必須というわけではありません[1]。タスクの性質に応じて、適切な要素を選択し、組み合わせることが求められます。

**C.** 例示の力（**Few-Shot Learning**の統合）

プロンプト内に具体的な例を含めること（Few-Shot Learning）は、LLMに期待する出力形式、スタイル、あるいは推論プロセスを教えるための強力な手段です[4]。

- 効果: 例を示すことで、モデルはタスクのニュアンスをより深く理解し、一貫性のある、期待に沿った応答を生成する可能性が高まります[4]。特に、特定のフォーマット（例：JSON、箇条書き）やトーン（例：フォーマル、カジュアル）が要求される場合に有効です[4]。
- 現代の**LLM**における効率性: 最新のLLMは、Few-Shot（少数例示）学習能力が向上しており、場合によってはOne-Shot（単一例示）でも複雑なパターンを理解できるようになっています[2]。これは、以前のモデルよりも少ない例で効果的なガイダンスを提供できる可能性を示唆しています。
- 効果的な組み込み方: 例は、明確で、タスクに直接関連しており、期待される出力の構造や内容を正確に反映している必要があります[4]。例えば、特定のデータセット形式を生成させたい場合、その形式に従ったサンプル行をプロンプトに含めます[5]。あるいは、特定の応答スタイルを模倣させたい場合、そのスタイルに従った応答例をいくつか提示します

[4]。

**D.** イテレーションと改善の必要性

プロンプトエンジニアリングは、一度で完璧な結果が得られることは稀であり、試行錯誤を伴う反復的なプロセスです [4]。

- 反復的改善サイクル**:** 効果的なプロンプトを作成するための典型的なプロセスは以下の通りです [6]。
  1. 基本的なプロンプトから始める。
  2. 生成された応答を分析する。
  3. 改善が必要な領域（不明瞭さ、不正確さ、形式の誤りなど）を特定する。
  4. 特定された問題点に基づいてプロンプトを修正・洗練する（例：指示の具体化、文脈の追加、例の変更、構造の調整）。
  5. 修正したプロンプトをテストする。
  6. 期待する品質の応答が一貫して得られるまで、ステップ2から5を繰り返す。
- 実験の推奨**:** 異なる言い回し、指示の与え方、構造、詳細レベルなどを試すことが重要です [4]。同じ課題に対して複数の異なるプロンプトを試し、どの表現が最も効果的かを比較検討することも有効な学習方法です [9]。また、他のユーザーが作成した効果的なプロンプトを研究し、その構造や表現を学ぶこともスキル向上に繋がります [9]。

**E.** 避けるべき一般的な誤り

効果的なプロンプトを作成する上で、陥りやすい一般的な間違いを避けることが重要です [6]。

- 曖昧すぎる**:**「このコードを修正して」のような一般的なプロンプトは避け、具体的に何を修正する必要があるのかを明記する。
- プロンプトの過負荷**:** 一つのプロンプトで多くのことを達成しようとしない。複雑なタスクは、より小さく管理しやすいプロンプトに分割する。
- 文脈の無視**:** 必要な背景情報を提供しないと、関連性のない、あるいは不正確な応答につながる可能性がある。
- **AI**の知識の過信**:** LLMは広範な知識を持つが、最新のフレームワークやライブラリについて常に最新であるとは限らない。最先端またはニッチな技術を扱う場合は、必要な情報を提供する。
- 出力形式指定の怠慢**:** 特定の形式が必要な場合は、プロンプトで明示的に指定しないと、使用できない形式の応答が返ってくる可能性がある。
- 出力の検証不足**:** 特に重要なタスクやコード生成の場合、AIの出力を常にレビューし、検証する。

これらの基本的な原則とベストプラクティスを理解し適用することは、プロンプトエンジニアリングの成功の基盤となります。明確性、具体性、文脈、構造、例示、そして反復的な改善という要素は、互いに密接に関連し合っています。例えば、十分な文脈を提供することが、LLMが要

求される具体性を理解する前提となります。また、プロンプトを構造化することは、文脈と具体的な指示の両方を明確に伝える助けとなります。例示は、期待される具体性や形式を具体的に示す役割を果たします。そして、イテレーションはこれら全ての要素を最適化していくプロセスです。したがって、これらの原則を個別にではなく、統合的に捉え、実践することが、LLMの能力を最大限に引き出す上で極めて重要です。

表1: プロンプトエンジニアリング ベストプラクティス チェックリスト

| チェック項目 | 説明 | 関連原則 |
| --- | --- | --- |
| 目標は明確か？ | プロンプトの目的と期待する成果（成功基準）が定義されているか？ [2] | 明確性、具体性 |
| 指示は具体的か？ | 実行してほしいタスクが曖昧でなく、具体的に記述されているか？ [4] | 具体性 |
| 十分な文脈があるか？ | 背景情報、関連データ、必要な前提知識などが提供されているか？ [4] | 文脈 |
| 制約は設定されているか？ | 文字数、トーン、形式、禁止事項など、必要な制約が指定されているか？ [4] | 制約 |
| 役割（ペルソナ）は適切か？ | 必要であれば、LLMに特定の役割（専門家、教師など）を割り当てているか？ [5] | 文脈、役割設定 |
| 対象読者は明確か？ | 応答が誰に向けたものか（専門家、初心者、子供など）が指定されているか？ [5] | 文脈、具体性 |
| 構造化されているか？ | 複雑な場合、区切り文字やタグ（例：XML）で指示、文脈、入力などを分離しているか？ [5] | 構造化 |
| タスクは分解されているか？ | 複雑なタスクが、より小さな実行可能なステップに分解されている | 構造化、具体性 |

| | | |
|---|---|---|
| | か？ [2] | |
| 例示は含まれているか？ | 必要に応じて、期待する出力形式、スタイル、推論プロセスを示す例が含まれているか？ [4] | 例示 |
| 出力形式は指定されているか？ | 期待する出力の形式（箇条書き、JSON、表など）が明記されているか？ [4] | 具体性、出力形式 |
| 反復的な改善を行っているか？ | 応答を分析し、プロンプトを修正・テストするプロセスを経ているか？ [4] | イテレーション |
| 一般的な誤りを避けているか？ | 曖昧さ、過負荷、文脈無視、知識の過信、形式指定漏れ、検証不足などをチェックしたか？ [6] | 全般 |

このチェックリストは、プロンプトを作成・評価する際のガイドとして利用できます。各項目を確認することで、見落としがちな要素に気づき、より効果的なプロンプトへと体系的に改善していく助けとなります。

## III. コアとなるプロンプト技術の解説

基本的な設計原則に加え、特定の目的を達成するために開発された様々なプロンプト技術が存在します。ここでは、現在広く利用されている主要な技術について、その仕組み、利点、そして具体的な使用例を解説します。

### A. Zero-Shot vs. Few-Shot プロンプティング

Zero-ShotとFew-Shotは、LLMにタスクを指示する際に、どの程度の例示（サンプル）をプロンプトに含めるかという点で区別される基本的な技術です。ここでいう「Shot」は、LLMに与える「例」や「ヒント」の数を指します [14]。

- **Zero-Shot プロンプティング:**
  - 定義: タスクの指示のみを与え、具体的な実行例を一切示さずにLLMに応答させる方法です [4]。モデルは、事前学習で得た広範な知識と、プロンプト内の指示に基づいてタスクを遂行します。
  - 仕組み: モデルが持つ汎化能力に依存します。与えられた指示を解釈し、既存の知識パターンを適用して応答を生成します [11]。
  - 利点: プロンプトの準備が簡単で、特定の例を用意する必要がありません。比較的単純なタスクや、モデルが事前学習で十分な知識を持っていると考えられるタスクに適

しています [4]。
    - 欠点: モデルの解釈に依存するため、特に複雑なタスクや特定の形式・スタイルが要求される場合、期待通りの出力が得られない可能性があります [17]。
    - 例: 「'Hello, how are you?'を英語からフランス語に翻訳してください」[12] や、「相対性理論を高校生向けに簡単な言葉で説明してください」[4] といった指示がZero-Shotプロンプトにあたります。
- **Few-Shot** プロンプティング:
    - 定義: タスクの指示と共に、1つ以上の具体的な実行例（入力と期待される出力のペア）をプロンプト内に含めてLLMに応答を促す方法です [4]。通常、数個（few）の例が用いられます。
    - 仕組み: 提供された例が、モデルに対する文脈やガイダンスとして機能します。モデルはこれらの例から、タスクのパターン、期待される出力形式、トーン、あるいは推論プロセスを学習し、新しい入力に対してそれを適用します [4]。
    - 利点: Zero-Shotに比べて、モデルがタスクをより正確に理解し、期待に沿った、一貫性のある出力を生成する可能性が高まります [4]。特定のフォーマットやスタイルが重要な場合に特に有効です。
    - 欠点: 適切な例を選び、プロンプトに組み込む手間がかかります。例の質や選択が不適切だと、かえって性能を低下させる可能性もあります。
    - 例: 顧客からの問い合わせに特定のスタイルで応答させたい場合、「顧客からの問い合わせへの回答方法はこちらです： 例1:『ご懸念承知いたしました。直ちに解決いたします。』 例2:『ご不便をおかけし申し訳ありません。迅速な解決策でお手伝いさせてください。』さて、以下の問い合わせに回答してください： [問い合わせ内容]」[4] のように例を示します。詩のスタイルを模倣させたい場合も同様です [12]。
- 使い分け:
    - **Zero-Shot:** タスクが単純明快で、モデルの一般知識で十分対応可能と考えられる場合、または適切な例を用意するのが難しい場合。
    - **Few-Shot:** 特定の出力形式、スタイル、推論パターンが要求される場合、タスクが複雑でモデルに追加のガイダンスが必要な場合、または応答の一貫性を高めたい場合。

近年のLLMはFew-Shot学習能力が向上しており、以前よりも少ない例で効果が得られる傾向にあります [2]。これは、Few-Shotプロンプティングの利便性を高める要因となっています。なお、機械学習分野におけるZero-Shot/Few-Shot Learningの定義と、GPT-3以降のLLM文脈での使われ方には差異がある可能性も指摘されており [18]、LLMのプロンプティングにおける定義として理解することが重要です。

**B. Chain-of-Thought (CoT)** プロンプティング：複雑な推論の解明

Chain-of-Thought（CoT、思考の連鎖）プロンプティングは、LLMに複雑な問題解決を行わせる際に、最終的な答えに至るまでの中間的な思考プロセスや推論ステップを明示的に生成さ

せる技術です[11]。

- 定義と目的: CoTは、LLMが問題を段階的に分解し、各ステップでの推論を示すように促すことで、特に多段階の推論、算術計算、常識的推論、記号的推論などを必要とするタスクのパフォーマンスを向上させることを目的としています[15]。
- 仕組み: 人間が問題を解く際の思考プロセスを模倣させます[15]。プロンプト内で、問題、推論ステップ、最終的な答えを含む例（Few-Shot CoT）を提示することで、LLMは新しい問題に対しても同様に段階的に思考し、それを記述することを学習します[15]。これにより、複雑な問題をより小さな管理可能なステップに分解し、論理的な誤りを減らすことが期待されます。
- **Few-Shot CoT**の例: 算術問題でCoTを適用する場合、単に問題と答えを示すのではなく、計算過程を含めます[19]。
  Q: このグループの奇数を足すと偶数になりますか？ 4, 8, 9, 15, 12, 2, 1
  A: 奇数 (9, 15, 1) をすべて足すと 25 になります。答えは False です。
  Q: このグループの奇数を足すと偶数になりますか？ 17, 10, 19, 4, 8, 12, 24
  A: 奇数 (17, 19) をすべて足すと 36 になります。答えは True です。
  Q: このグループの奇数を足すと偶数になりますか？ 15, 32, 5, 13, 82, 7, 1
  A:
  期待される出力:
  奇数 (15, 5, 13, 7, 1) をすべて足すと 41 になります。答えは False です。

- **Zero-Shot CoT:** 近年提案された、よりシンプルなCoTの形式です。Few-Shot CoTのように推論ステップを含む例を提示する代わりに、元のプロンプトの末尾に「ステップバイステップで考えましょう (Let's think step-by-step)」のような定型句を追加するだけで、LLMに推論プロセスを生成させることができます[15]。
  - 仕組み: この定型句が、LLMに対して問題を分解し、思考プロセスを言語化してから最終的な答えを出すように指示する汎用的なトリガーとして機能します。特定の例がなくても、モデルは自身の持つ問題解決能力を活用して思考の連鎖を生成しようとします[19]。
  - 利点: Few-Shot CoTのための例を作成する手間が不要であり、例が少ない場合に特に有用です[19]。
  - 例: [19] より、リンゴの計算問題。
    - 通常のプロンプト:「市場に行ってリンゴを10個買いました。隣人に2個、修理工に2個あげました。その後、さらに5個買って1個食べました。残りは何個ですか？」→ 誤った出力: 11個
    - **Zero-Shot CoT**プロンプト:「市場に行ってリンゴを10個買いました。隣人に2個、修理工に2個あげました。その後、さらに5個買って1個食べました。残りは何個ですか？ ステップバイステップで考えましょう。」→ 正しい出力:「まず、10個のリンゴから始めます。隣人と修理工にそれぞれ2個ずつあげたので、残りは6個で

す。次に5個買ったので、合計11個になります。最後に1個食べたので、残りは10個です。」

- **Auto-CoT:** CoTプロンプトにおける効果的で多様な例を手作業で作成する手間を省くために提案された手法です [19]。LLM自身に「ステップバイステップで考えましょう」というプロンプトを用いて推論連鎖を生成させ、自動的にデモンストレーション例を構築します。生成された連鎖には誤りが含まれる可能性があるため、多様な問題から代表的なものをサンプリングし、生成された推論連鎖を組み合わせることで、エラーの影響を軽減します [19]。

CoTプロンプティングは、特に複雑な推論が求められるタスクにおいて、LLMの能力を引き出すための強力なテクニックです。Zero-Shot CoTの登場により、その利用のハードルはさらに下がりました。

**C.** 役割プロンプティング（ペルソナプロンプティング）：ペルソナによる**LLM**の誘導

役割プロンプティング（ロールプロンプティング、ペルソナプロンプティングとも呼ばれる）は、LLMに対して特定の役割やペルソナ（例：専門家、教師、批評家、特定の職種）を割り当てることで、その応答のスタイル、トーン、焦点、利用する知識ベースなどを誘導する技術です [1]。

- 定義と目的**:** LLMにあたかも特定の人物であるかのように振る舞うよう指示することで、特定の文脈や専門知識レベルに合わせた、よりターゲットを絞った出力を得ることを目的とします [12]。
- 仕組み**:** 役割の割り当ては、一種の文脈注入として機能します [13]。LLMは、「この特定のレンズを通して知識をフィルタリングせよ」という指示を受け取り、どのドメイン知識を優先し、どのように応答を構成すべきかを判断します。これにより、言葉遣い、文体、情報の優先順位などが変化します。
- 利点**:**
  - 明確性と精度の向上**:** 応答を特定の役割に合わせることで、テキストの明確性と精度が向上します。特に推論や説明タスクにおいて、特定の視点を採用することで、より焦点が絞られ、関連性の高い回答が得られます [25]。
  - スタイル模倣**:** 特定の人物や職業の書き方を模倣させることができます。これにより、トーン、スタイル、情報の深さを調整し、特定のオーディエンスや目的に合わせたコミュニケーションが可能になります [25]。
  - タスクパフォーマンスの向上**:** 役割設定は、ライティング、推論、対話ベースのアプリケーションなど、幅広いタスクでモデルのパフォーマンスを向上させることが示唆されています [8]。特定の役割（例：「あなたはサイバーセキュリティアナリストです」）は、より専門的で質の高い応答を引き出す可能性があります [5]。
  - 複雑なアイデアや微妙な行動の表現**:** 特定の人物（例：ガンジー）を役割として設定することで、特定の道徳的基準などを明示的に述べずに示唆することも可能です [25]。
- ベストプラクティス**:**
  - 役割の明示**:** 役割を曖昧に示唆するのではなく、「あなたは[役割]です」と直接的に記述します [13]。より良い結果を得るために、「あなたは10年の経験を持つ倫理的ハッ

カーです」のように、役割に具体的な特性を追加することも有効です [13]。
    - 非親密な対人関係の役割: 学校、社会、職場などの設定では、職業的な役割よりも、非親密な対人関係の役割（例：教師、同僚、友人）の方が良い結果をもたらす傾向があります [25]。
    - 性別中立的な用語: プロンプトで性別中立的な用語を使用すると、一般的にパフォーマンスが向上します [25]。
    - 直接的な役割/対象者指定: 「想像して...」のような想像上の構成や、「あなたの[役割]と話しています」のような関係性を示唆するプロンプトよりも、「あなたは[役割]です」や「あなたは[役割]と話しています」といった直接的な指定の方が効果的です [25]。
    - 二段階アプローチ: 複雑なシナリオでは、まず役割を設定し、モデルにその役割に基づいた初期応答を生成させ、次のプロンプトで実際の質問やタスクを提示するという二段階のアプローチが有効な場合があります [25]。
- 例:
    - フードクリティック: 「あなたはフードクリティックです。[ピザ屋の名前]のレビューを書いてください。」→ 通常のプロンプトよりも詳細で掘り下げたレビューが生成される [26]。
    - カスタマーサポート担当者: 「あなたはテック企業のカスタマーサポート担当者です。次の顧客からの問い合わせに回答してください：『ラップトップが起動しません...』」→ トラブルシューティング手順を含む適切な応答が生成される [12]。
    - セールスパーソン: 「あなたはセールスパーソンです。[相手の名前]に提携について簡単なアプローチメールを書いてください。」→ よりビジネスライクで行動指向のメールが生成される [25]。
- 注意点: 役割プロンプティングの効果については、肯定的な報告 [25] と懐疑的な研究 [27] が混在しています。効果はタスクやモデル、役割設定の具体性に依存する可能性があります。また、LLMの学習データに含まれるバイアスを意図せず増幅させてしまう可能性も指摘されています [25]。より高度な実装では、役割設定と応答生成を複数回のLLM呼び出しや評価器を用いて行うアプローチも提案されています [27]。

**D.** 構造化出力：利用可能な応答の保証

LLMはしばしば自由形式のテキストを生成しますが、その出力を後続のアプリケーションやシステムで利用するためには、JSONやXMLのような特定の構造化された形式で応答を得ることが不可欠です [28]。

- 問題点: LLMの応答は、フォーマットの不整合（余分なスペース、改行、引用符の不一致）、無関係なテキストの混入（応答の前後の会話的な蛇足）、ハルシネーション（データの捏造や指示の誤解釈）などにより、プログラムでの処理が困難になることがあります [29]。
- 目的: 信頼性が高く、一貫して特定のスキーマに準拠した形式（例：JSON、XML）の出力を得ることです [28]。
- 実現方法:

    - プロンプティング: プロンプト内でLLMに「JSON形式で、キーsetupとpunchlineを持つように応答してください」のように、望ましい形式を直接指示する方法です [5]。
        - 利点: 全てのLLMで試すことができます。
        - 欠点: LLMが指示に完全に従う保証はなく、形式が崩れたり、不完全になったりする可能性があります [28]。OpenAIによると、この方法でのスキーマフォーマットの一貫性は約35.9%程度であったとされます [29]。
        - 補助テクニック: 応答の開始部分（例：{）を事前に埋めておく [30]、あるいはXMLタグで期待される出力箇所を囲む [5] といった工夫で、精度を高められる場合があります。
    - 関数呼び出し (Function Calling) / ツール呼び出し (Tool Calling): LLMが、API経由で渡された事前定義された関数（ツール）を呼び出すべきだと判断し、その関数の引数として構造化されたデータを生成する機能です [28]。関数のパラメータ定義が出力構造を規定するため、プロンプティングのみに比べて信頼性が高まります。ツール呼び出しは、複数の関数を同時に呼び出すことを可能にします [28]。
    - JSONモード: 一部のLLMプロバイダーが提供する、出力が常に有効なJSON形式であることを保証する機能です [28]。このモードを使用する場合でも、期待するJSONの具体的なスキーマ（キーと値の型）をプロンプトで指示する必要があることが多いです [28]。OpenAIの「Structured Outputs」は、従来のJSONモードを改善し、スキーマへの準拠性を高めたバージョンとされています [29]。
- 実装例 (LangChain): LangChainフレームワークは、.with_structured_outputメソッドを通じて、これらの構造化出力技術を統一的なインターフェースで利用する方法を提供しています [28]。
    1. スキーマ定義: PydanticモデルやJSONスキーマを用いて、期待する出力構造を定義します [28]。
    ```Python
    from langchain_core.pydantic_v1 import BaseModel, Field

    class Joke(BaseModel):
        setup: str = Field(description="ジョークの導入部")
        punchline: str = Field(description="ジョークのオチ")
    ```
    2. LLM初期化: ChatOpenAIなどのLangChainのモデルを初期化します。
    3. .with_structured_output適用: モデルインスタンスに対して.with_structured_output(Joke)のように呼び出し、スキーマをバインドします。method引数で"function_calling"や"json_mode"を指定することも可能です [28]。
    ```Python
    from langchain_openai import ChatOpenAI
    model = ChatOpenAI(model="gpt-4o", temperature=0)
    structured_llm = model.with_structured_output(Joke) # デフォルトは関数呼び出しなど
    ```

```
# structured_llm = model.with_structured_output(Joke, method="json_mode") # JSONモード指定
```

4. 呼び出し: 構造化LLMインスタンスをプロンプトと共に呼び出すと、定義したスキーマ（この例ではJokeオブジェクト）に沿った出力が得られます [28]。
```
Python
structured_llm.invoke("猫についてのジョークを教えて")
# JSONモードの場合: structured_llm.invoke("猫についてのジョークを教えて、`setup`と`punchline`キーを持つJSONで応答して")
```

- 利点: 応答の信頼性向上、後続処理の容易化、データ検証の可能性、解析の簡略化 [28]。
- 課題: スキーマ設計の複雑さ [29]、構造化出力のトークン数制限 [29]、構造化を強制することによる本来のタスクパフォーマンスへの影響の可能性 [33]、場合によっては出力の修正・後処理の必要性 [30]。一部の状況ではXML形式が好まれることもあります [33]。

表2: コアとなるプロンプト技術の比較

| 技術 | 仕組み | 主な用途 | 主な利点 | 主な限界/考慮事項 | 例の出典例 |
|---|---|---|---|---|---|
| **Zero-Shot** | 指示のみ。事前学習知識と汎化能力に依存 [4]。 | 単純なタスク、例が不要/利用不可な場合 [4]。 | 簡単、準備不要。 | 複雑なタスクや特定形式では精度が不安定 [17]。 | [12] |
| **Few-Shot** | 指示＋1つ以上の例を提供。例からパターンや形式を学習 [4]。 | 特定の形式/スタイル/推論が必要なタスク、一貫性向上 [4]。 | 精度と一貫性の向上。 | 例の準備が必要、例の質に依存。 | [4] |
| **Chain-of-Thought (CoT)** | 問題解決の中間的な推論ステップを生成させる [15]。 | 複雑な推論、算術、常識問題 [15]。 | 推論能力の向上、プロセスの透明化。 | プロンプトが長くなる可能性、Few-Shot CoTは例作成が必要。 | [19] |
| 役割プロンプティング | LLMに特定のペルソナを割り当て、応答スタイルや焦 | 特定の専門知識/視点/トーンが必要な場合、スタイル | 応答のターゲット化、明確性・精度の向上（可能性）。 | 効果は不定、バイアス増幅の可能性、役割設定の工夫 | [26] |

| | | | | | |
|---|---|---|---|---|---|
| | 点を誘導[8]。 | 模倣[25]。 | | が必要[25]。 | |
| 構造化出力 | JSON/XML等の特定形式での出力を強制/誘導（プロンプティング、関数呼び出し、JSONモード）[28]。 | 出力をプログラムで利用する場合、データ抽出[28]。 | 出力の信頼性と利用可能性の向上。 | 複雑さ、コスト、LLM/機能のサポート依存、タスク性能への影響可能性[29]。 | [28] |

これらのコア技術は、しばしば組み合わせて使用されます。例えば、Few-Shotプロンプティングと CoTを組み合わせて、特定の推論スタイルを示す例を提供することができます[15]。役割プロンプティングは、Zero-ShotやFew-Shot、CoTが実行される際の文脈を設定し、その結果に影響を与えます[13]。構造化出力は、しばしば役割設定や例示を含むプロンプト、あるいは関数呼び出しといったメカニズムを通じて達成される目標となります[28]。

また、これらの技術は、期待される出力の信頼性において異なるレベルを提供します。単純なプロンプティングは最も信頼性が低い一方[28]、Few-ShotやCoT、役割プロンプティングはガイダンスを強化します。そして、関数呼び出しやJSONモードのような専用メカニズムは、特に構造に関しては最も高い（ただし完璧ではない）保証を提供します[28]。このことは、実装の複雑さやモデルのサポート状況と、得られる出力の信頼性との間にトレードオフが存在することを示唆しています。実践者は、タスクの要件に応じてこれらの技術を適切に選択し、組み合わせる必要があります。

## IV. LLMのパフォーマンスを向上させる高度な戦略

基本的なプロンプト技術に加え、LLMの能力をさらに引き出し、より複雑なタスクに対応するための高度な戦略が開発されています。これらの戦略は、しばしば複数のLLM呼び出しや洗練されたプロンプト構造を伴いますが、応答の質、信頼性、推論能力を大幅に向上させる可能性があります。

### A. 自己修正とリフレクション：LLMに「思考について考えさせる」

LLMに自身の出力や推論プロセスを批判させ、評価させ、そして洗練させるように促すアプローチは、精度と信頼性を向上させるための強力な手段として注目されています[2]。これは、LLMを単なる応答生成器から、自身の思考プロセスを監視し改善できる、より高度な「システム2」思考[34]やメタ認知[2]に近い存在へと導く試みと言えます。この分野では、多様な技術が提案されています。

- 基本的なリフレクション: 最もシンプルな形式は、LLMにまず応答を生成させ、次にその応答を（場合によっては「教師」のような異なるペルソナで）批判・評価させるという2段階の

プロセスです [34]。

- **Reflexion (Shinn et al.):** 言語的なフィードバックを通じてエージェントを強化するフレームワークです [24]。以下の3つのモデルで構成されます。
  - **Actor:** 状態観測に基づいてテキストや行動を生成します。CoTやReActのような手法とメモリを利用します [37]。
  - **Evaluator:** Actorが生成した軌跡（短期記憶）を評価し、報酬スコアを出力します [37]。
  - **Self-Reflection:** 報酬信号、現在の軌跡、永続的なメモリを利用して、Actorの自己改善を支援するための言語的なフィードバック（自己反省）を生成します [37]。このフレームワークは、エージェントが過去の失敗から言語的なフィードバックを通じて迅速に学習することを可能にし、特に試行錯誤が必要なタスクや、解釈可能性が重要な場合に有効です [37]。ただし、自己評価能力や長期記憶の制約に依存するという課題もあります [37]。
- **Language Agent Tree Search (LATS) (Zhou et al.):** リフレクション/評価と探索アルゴリズム（特にモンテカルロ木探索）を組み合わせた汎用的なLLMエージェント探索アルゴリズムです [34]。強化学習の枠組みを採用し、エージェント、価値関数、オプティマイザをすべてLLMの呼び出しで置き換えます。探索プロセスは以下の4ステップで構成されます [34]。
  1. 選択 **(Select):** 過去の報酬に基づいて最良の次の行動を選択します。
  2. 展開とシミュレーション **(Expand and Simulate):** 複数の（例：5つ）潜在的な行動を生成し、並行して実行します。
  3. リフレクション＋評価 **(Reflect + Evaluate):** これらの行動の結果を観察し、リフレクション（および外部フィードバック）に基づいて決定をスコアリングします。
  4. バックプロパゲーション **(Backpropagate):** 結果に基づいてルート軌跡のスコアを更新します。LATSは、推論、計画、リフレクションを統合し、エージェントが複雑なタスクに適応し、反復ループに陥るのを避けるのに役立ちます [34]。
- **Self-Refine (Madaan et al.):** 人間が下書きを作成し、レビューと修正を通じて改善するプロセスを模倣します。LLMは初期出力を生成した後、自己批判に基づいて段階的にそれを反復的に洗練させ、精度と品質を向上させます [36]。
- **Self-Calibration (Jiang et al.):** LLMが応答を生成した後に、自身の出力の正しさや信頼度を評価するように促す手法です。これにより、モデルは自身の誤りを認識し、確信度のキャリブレーション（過信や過小評価の抑制）を改善することを目指します [36]。
- **Reversing Chain-of-Thought (RCoT) (Chang et al.):** CoTのプロセスを逆転させることで、ハルシネーションや不正確な仮定を検出・修正します。まず問題を解き、次にその解から元の問題に合致するはずの新しい問題を生成させ、元の問題と比較することで不整合を発見します [36]。
- **Self-Verification (Weng et al.):** CoTを用いて複数の候補解を生成した後、元の質問の一部をマスキングし、各解が残りの情報からマスキングされた部分を予測できるかを確認することで、解の妥当性を検証します [36]。
- **Chain-of-Verification (CoVe) (Dhuliawala et al.):** CoTのように中間ステップを生

成する代わりに、初期応答に対して検証用の質問を生成させます。モデルはこれらの質問に答えることで、自身の応答を評価し、最終的な出力を洗練させます [36]。

- **Cumulative Reasoning (CR) (Zhang et al.):** 問題解決を複数のステップに分解し、各ステップをLLMが評価して受け入れるか拒否するかを決定します。最終的な答えに至るまで、このプロセスを反復的に行い、解決策に到達するまで洗練を続けます [36]。
- 自己修正の実例（医療）: LLMに医療情報の要約を生成させた後、「科学的正確性」「記述の完全性」「バランス」「一般読者への明瞭さ」といった観点から自己検証を行い、必要に応じて修正させるプロンプトが有効な例として挙げられます [2]。

これらの高度なリフレクション技術は、LangGraph [34] や AutoGen [35] のようなフレームワークを使用して実装されることがあります。これらのフレームワークは、複数のエージェントやLLM呼び出しを連携させ、複雑な思考プロセスを構築するのに役立ちます。

表3: 高度な自己修正/リフレクション技術の概要

| 技術 | コアアイデア/メカニズム | 主要コンポーネント/ステップ | 主な利点/目的 |
|---|---|---|---|
| **Reflexion** | 言語的フィードバック（自己反省）を通じてエージェントを強化 [37]。 | Actor (生成), Evaluator (評価), Self-Reflection (フィードバック生成) [37]。 | 試行錯誤からの学習、ニュアンスのあるフィードバック、解釈可能性 [37]。 |
| **LATS** | リフレクション/評価とモンテカルロ木探索を組み合わせる [34]。 | 選択、展開＆シミュレーション、リフレクション＋評価、バックプロパゲーション [34]。 | 複雑なタスクへの適応、反復ループの回避、複数経路の探索 [34]。 |
| **Self-Refine** | 生成→自己批判→反復的改善のプロセス [36]。 | 初期生成、反復的な評価と洗練 [36]。 | 精度と品質の段階的な向上 [36]。 |
| **CoVe** | 初期応答に対する検証質問を生成し、それに答えることで応答を検証・洗練 [36]。 | 初期応答生成、検証質問生成、質問応答、最終応答洗練 [36]。 | 応答の正確性と信頼性の向上 [36]。 |
| **RCoT** | 問題解決→解から問題再構築→比較、によりハルシネーション等を検 | 問題解決、問題再構築、比較 [36]。 | 誤った仮定やハルシネーションの検出・修正 |

| | | | |
|---|---|---|---|
| | 出[36]。 | | [36]。 |
| **Self-Calibration** | 生成後に自身の応答の正しさ/信頼度を評価させる[36]。 | 応答生成、自己評価[36]。 | 誤りの発見、確信度キャリブレーションの改善[36]。 |
| **Self-Verification** | 複数CoT解を生成し、元の質問の一部をマスキングして予測可能か検証[36]。 | 複数解生成 (CoT)、マスキング、予測検証[36]。 | CoTの結論の正確性検証[36]。 |
| **Cumulative Reasoning (CR)** | 問題をステップに分解し、各ステップを評価（受諾/拒否）しながら解決まで反復[36]。 | ステップ分解、ステップ評価、反復的洗練[36]。 | 複雑な問題の段階的かつ堅牢な解決[36]。 |

**B.** フロンティアを探る：その他の注目すべき技術

自己修正・リフレクション以外にも、LLMの能力を引き出すための様々な先進的なプロンプティング技術が研究・開発されています。

- **Tree-of-Thoughts (ToT) (Yao et al./Long):** CoTを一般化し、複数の可能性のある思考経路（思考の木）を同時に探求する手法です[15]。中間的な思考を複数生成し、それらを評価し、幅優先探索（BFS）やビームサーチなどの探索アルゴリズムを用いて思考の木を探索します[15]。問題解決プロセスを遡る（バックトラック）機能を持つモジュールを含む場合もあります[15]。
- **Self-Consistency (Wang et al.):** 特に算術や常識推論タスクにおいて、複数の多様な推論経路（しばしばFew-Shot CoTを使用）を生成させ、複数の解をサンプリングし、その中で最も頻繁に出現する（一貫性のある）答えを選択することで、精度を向上させる手法です[15]。単一のCoT経路に含まれる可能性のある誤りの影響を低減します[24]。例えば、統計問題を3つの異なる方法で解かせ、最も信頼性の高い回答を選ばせる、といった応用が考えられます[2]。
- 深津式プロンプト **(Fukatsu-Style Prompting):** プロダクトデザイナーの深津貴之氏によって提唱された、構造化されたプロンプト形式です[8]。基本的な構成要素（命令、制約、入力、出力）に加え、AIに逆質問を促して情報を補完させる「逆質問機能」、厳格な要求と段階的な改善プロセスを特徴とする「パワハラプロンプト」、目標から逆算して手順を導く「ゴールシークプロンプト」、複数のAIに異なる役割を与えて協調させる「複数AI連携プロンプト」などの高度なバリエーションが存在します[8]。ビジネス文書作成、プログラミング、クリエイティブ作業など、多岐にわたる応用が報告されています[8]。
- **Mixture of Formats (MOF) (Elhafni et al.):** プロンプトの些細な形式変更に対するLLMの感受性（プロンプト脆弱性）に対処するため、Few-Shotプロンプト内の各例を異な

るスタイルで提示し、さらにモデルに各例を異なるスタイルで書き直すよう指示する手法です [11]。これにより、モデルが表面的なスタイルに依存せず、タスクの本質的な内容に焦点を当てるように促します。
- **Conversation Routines (CR) (Ahmed et al.):** タスク指向の対話システムの開発において、ワークフロー、条件分岐、反復処理、関数呼び出しといったビジネスロジックを、自然言語による仕様としてLLMのシステムプロンプトに直接埋め込むフレームワークです [32]。これにより、ローコードでの対話エージェント開発が可能になります。
- プロンプト連鎖 **(Prompt Chaining):** 複雑なタスクを複数の連続したプロンプトに分解し、あるプロンプトの出力を次のプロンプトの入力として使用する手法です [10]。

これらの先進的な技術は、LLMの応用範囲を広げ、より困難な課題に取り組むための道を開いています。自己修正・リフレクション技術の隆盛は、単なる指示応答を超え、自身の推論プロセスを内省し改善できる、より堅牢で信頼性の高いAIシステムへの移行を示唆しています [2]。

一方で、これらの高度な技術の多くは、複数のLLM呼び出し（例：LATSのシミュレーションとリフレクション [34]、Self-Consistencyのサンプリング [15]、Reflexionの複数モデル [37]）や、複雑なプロンプト構造（例：CR [32]、ToT [15]）を必要とします。これは、応答時間の遅延や計算コストの増加につながる可能性がありますが、その対価として、より高い品質や信頼性を目指すものです。この複雑性とパフォーマンスのトレードオフは、実践者が考慮すべき重要な要素です。

さらに、これらの技術が互いに影響し合い、融合していく傾向も見られます。LATSはリフレクションを組み込み [34]、ReflexionはCoTやReActを利用し [37]、Self-ConsistencyはしばしばCoTを基盤としています [24]。これは、分野が成熟し、成功した要素を組み合わせてより強力な複合的戦略が生まれていることを示唆しています。

## V. LLMの多様性とプロンプト感受性への対応

プロンプトエンジニアリングを実践する上で、LLMが常に一貫した振る舞いをするわけではないことを理解しておく必要があります。モデルの種類や、プロンプトのわずかな違いによって、パフォーマンスが大きく変動することがあります。

### A. プロンプト脆弱性の理解

プロンプト脆弱性（Prompt Brittleness）とは、LLMがプロンプトの意味内容を変えない些細な変更（例：フォーマットの変更、例の順序の入れ替え）に対して敏感に反応し、パフォーマンスが大きく変動する現象を指します [11]。

- 現象**:** 同じ意図を持つプロンプトでも、表現や構造がわずかに異なると、LLMの応答品質が大きく変わることが、複数の研究で定量的に示されています [11]。これは、LLMがまだ完全な意味理解には至っておらず、表面的な形式に影響されやすいことを示唆しています。
- 対策**:** この脆弱性に対処するために、いくつかの技術が開発されています。

- **Mixture of Formats (MOF):** Few-Shotの例を多様なスタイルで提示することで、モデルが特定の形式に過度に依存しないように訓練する手法です [11]。
- アンサンブル法: 複数の異なるプロンプトテンプレートでモデルを実行し、その予測結果を集約するアプローチです（例：Template Ensembles [11]）。ただし、複数のプロンプトを実行する必要があるため、計算コストが高くなる可能性があります [11]。
- 複数プロンプト評価: 単一のプロンプトではなく、複数のプロンプトでモデルの性能を評価する手法です（例：PromptEval [11]）。

プロンプト脆弱性の存在は、現在のLLM技術における根本的な課題、すなわち表面的な形式が出力に強く影響するという意味理解の限界を浮き彫りにしています。MOFやアンサンブルのような技術は、この課題に対する回避策であり、堅牢性の向上が活発な研究分野であることを示しています。

**B.** 異なるモデルとプロバイダーへのプロンプト適応

最適なプロンプトは、使用するLLM（例：GPT-4, Claude, Gemini, Llama）によって異なる可能性があります [4]。これは、モデルごとに学習データ、アーキテクチャ、ファインチューニングの方法が異なるためです。

- モデル特性への適合: プロンプトは、使用するモデルの既知の強み（例：創造的なライティングが得意か、技術的なタスクが得意か）に合わせて調整することが推奨されます [4]。
- 機能サポートの差異: 関数呼び出しやJSONモードのような特定の機能のサポート状況は、プロバイダーやモデルによって異なります [28]。あるモデルで有効な構造化出力の方法が、別のモデルでは利用できない場合があります。
- テストと適応の必要性: モデルを切り替える際には、既存のプロンプトが同様に機能するとは限らないため、テストを行い、必要に応じてプロンプトを調整することが重要です [4]。

プロンプト脆弱性とモデル間の多様性を組み合わせると、特定のタスクに対して、あらゆるLLMで普遍的に最適となる単一の「完璧な」プロンプトは存在しない可能性が高いと言えます。あるモデルやバージョンで最適化されたプロンプトが、別のモデルでは最適でない可能性があるため、モデル固有のチューニングと反復的な改善プロセスが不可欠となります。

## VI. 進化するランドスケープと今後の方向性

プロンプトエンジニアリングの分野は、LLM自体の急速な進化と共に、常に変化し続けています。最新のトレンドを把握し、将来の方向性を理解することは、この分野で効果を発揮し続けるために重要です。

**A.** 現在のトレンド（[2]の統合的考察）

近年の変化から、いくつかの重要なトレンドが浮かび上がっています [2]。

- シンプルさへの回帰？: より高性能なベースモデルや改善されたアライメントにより、場合

によっては過度に複雑なプロンプトよりも、明確で簡潔な指示の方が効果的であるという観察があります [2]。ただし、これはタスクの複雑さや要求される精度に依存する可能性が高いです（II.A参照）。「洗練されたシンプルさ」、つまり本質を捉えた無駄のない指示が鍵となるかもしれません。

- メタ認知への注目: 自己反省、自己評価、自己批判といった、LLMが自身の思考プロセスを客観視するようなメタ認知的アプローチの重要性が増しています [2]。これは、より信頼性が高く、堅牢なAIシステムを構築するための中心的なテーマとなりつつあります（IV.A参照）。
- **Few-Shot**学習能力の向上: モデルは、より少ない例から複雑なパターンを学習する能力を高めています [2]。これにより、Few-Shotプロンプティングの効率性と実用性が向上しています。
- 洗練された役割演技: 単一の役割だけでなく、複数の専門家ペルソナを組み合わせるなど、より高度でニュアンスのある役割設定が可能になりつつあります [2]。
- 協調へのシフト: プロンプトエンジニアリングは、単に「AIの能力を引き出す技術」から、「AIと効果的に協働する方法」へと進化しています [2]。問題の本質を伝え、AIの推論能力を信頼し、対話を通じて解決策を共創していくアプローチがより重要になっています。

**B.** プロンプト最適化とフレームワークの台頭

手動でのプロンプト作成・改善プロセスを支援、あるいは自動化するためのツールやフレームワークが登場しています。

- 自動プロンプト最適化 **(APO):** 特定のタスクに対して最適なプロンプト（指示や例）を自動的に発見・最適化することを目指す手法群です [11]。
- プロンプティングによる最適化 **(OPRO):** LLM自体をオプティマイザとして活用し、最適化タスクを自然言語で記述することでプロンプトを改善させる手法です [11]。
- 体系的フレームワーク **(DSPy):** ハードコーディングされたプロンプトテンプレートを置き換え、LLMアプリケーションのためのパイプラインを体系的かつモジュール式に構築するためのフレームワークです [11]。宣言的に望ましい振る舞いを定義し、それをコンパイルしてパイプラインを生成します。
- 対話フレームワーク **(CR):** Conversation Routinesのように、タスク指向の対話ロジックをプロンプト内に埋め込むことで、対話エージェントの開発を効率化するフレームワークです [32]。
- 影響: これらのツールの進化は、プロンプトエンジニアの役割を変える可能性があります。手作業での緻密なプロンプト作成から、自動化システムのための目標設定、制約定義、評価基準策定へと、焦点が移っていくかもしれません [9]。

この分野では、高度な手動技術（例：複雑なリフレクション戦略、深津式）と、自動化ツール（APO, OPRO, DSPy）の両方が進展しています。これは、将来的に、手動技術への深い理解と自動化ツール/フレームワークを活用する能力の両方が価値を持つようになる可能性を示唆し

ています。

**C.** 基礎の永続的重要性

高度な技術や自動化ツールが登場する中でも、プロンプトエンジニアリングの基礎原則（明確性、具体性、文脈、構造など、セクションIIで詳述）は依然として極めて重要です。自動化システムや複雑なフレームワークであっても、その入力として明確に定義された目標、制約、そして十分な文脈情報が必要となるでしょう。CRのようなフレームワークも、プロンプト内での詳細な仕様記述を必要とします [32]。OPROのような最適化技術も、最適化タスク自体を明確に記述する必要があります [11]。したがって、使用する技術やツールに関わらず、意図を明確に伝え、必要な情報を提供する能力は、今後もプロンプトエンジニアリングの中核であり続けると考えられます。

## VII. 結論：実践を通じたプロンプトエンジニアリングの習得

本レポートでは、2024年から2025年にかけて有効とされる最新のプロンプトエンジニアリング技術について概説しました。効果的なプロンプト作成は、明確性、具体性、文脈、構造といった基礎原則の上に成り立っています。これらを基盤として、Zero-Shot、Few-Shot、Chain-of-Thought、役割プロンプティング、構造化出力といったコア技術を適切に選択し、組み合わせることが求められます。さらに、複雑なタスクや高い信頼性が要求される場面では、自己修正・リフレクション、Tree-of-Thoughts、Self-Consistencyといった高度な戦略が有効となります。

また、LLMの種類による挙動の違いやプロンプトの些細な変更に対する感受性（プロンプト脆弱性）を理解し、モデルへの適応と反復的な改善を行うことが不可欠です。

プロンプトエンジニアリングは、理論だけでなく、実践を通じて磨かれるスキルです [4]。様々なテクニックを試し、結果を分析し、継続的に学習し、急速に進化する分野の最新動向を追い続けることが、習熟への道となります。自動化ツールやフレームワークが登場する中でも、基礎原則の理解と、意図を明確に伝える能力は、その重要性を失うことはないでしょう。

最終的に、プロンプトエンジニアリングは、人間とAIとの協調的な未来を築くための鍵となります [2]。熟練したプロンプトエンジニアリングを通じて、私たちはLLMの持つ計り知れない可能性を解き放ち、より複雑な問題を解決し、新たな価値を創造することができるようになるでしょう。

引用文献


1. Prompt Engineering for Large Language Model-assisted Inductive Thematic Analysis - arXiv, 4月 22, 2025にアクセス、https://arxiv.org/html/2503.22978v1
2. # 企業での生成AI活用ガイド：プロンプトエンジニアリング 2025（社内勉強会資料） - Zenn, 4月 22, 2025にアクセス、https://zenn.dev/devp/articles/14f71c31d8d9dd
3. 【2025年版】ChatGPTプロンプト設計トレンド｜アメリカの最新実践事例 - AI LAB, 4月 22, 2025にアクセス、https://ai.krgo.jp/chatgpt/chatgpt-prompt-trends-2025-us/

4. Prompt Engineering Guide: Techniques & Management Tips for LLMs - Portkey, 4月 22, 2025にアクセス、https://portkey.ai/blog/the-complete-guide-to-prompt-engineering
5. 8 best practices for effective prompt engineering | KNIME, 4月 22, 2025にアクセス、https://www.knime.com/blog/prompt-engineering
6. LLM Prompting Techniques for Developers - Pedro Alonso, 4月 22, 2025にアクセス、https://www.pedroalonso.net/blog/llm-prompting-techniques-developers/
7. Creating Effective Prompts: Best Practices, Prompt Engineering, and How to Get the Most Out of Your LLM - VisibleThread, 4月 22, 2025にアクセス、https://www.visiblethread.com/blog/creating-effective-prompts-best-practices-prompt-engineering-and-how-to-get-the-most-out-of-your-llm/
8. 【2025最新】深津式プロンプトとは？テンプレート・実例で ..., 4月 22, 2025にアクセス、https://ai-front-trend.jp/fukatsu-style-prompt/
9. 【2025年最新】プロンプトエンジニアリングでデザインが激変！基本から学ぶ7つのテクニック - note, 4月 22, 2025にアクセス、https://note.com/mmmiyama/n/nce6b928e4085
10. Prompt Engineering for Conversational AI Systems: A Systematic Review of Techniques and Applications, 4月 22, 2025にアクセス、https://ijsrcseit.com/index.php/home/article/view/CSEIT25111276
11. arxiv.org, 4月 22, 2025にアクセス、https://arxiv.org/pdf/2504.06969
12. LLM Prompt - Examples and Best Practices - Mirascope, 4月 22, 2025にアクセス、https://mirascope.com/blog/llm-prompt/
13. Mastering role prompting: How to get the best responses from LLMs - Portkey, 4月 22, 2025にアクセス、https://portkey.ai/blog/role-prompting-for-llms/
14. Zero-ShotとFew-Shotの違いを実例解説｜メリットや活用シーンも - 新時代のトビラ, 4月 22, 2025にアクセス、https://shinjidainotobira.com/shot-learning/
15. Prompt engineering - Wikipedia, 4月 22, 2025にアクセス、https://en.wikipedia.org/wiki/Prompt_engineering
16. 少ない例示で AIを賢く使う！Few-Shot Promptingの基本と活用テクニックを解説 - TechGrowUp, 4月 22, 2025にアクセス、https://techgrowup.net/prompt-engineering-few-shot-prompting/
17. Zero-Shot vs. Few-Shot Prompting: Key Differences - Shelf - Shelf.io, 4月 22, 2025にアクセス、https://shelf.io/blog/zero-shot-and-few-shot-prompting/
18. GPT-3におけるFew-Shot・Zero-Shot - Zenn, 4月 22, 2025にアクセス、https://zenn.dev/dhirooka/articles/34205e1b423a80
19. Chain-of-Thought Prompting | Prompt Engineering Guide, 4月 22, 2025にアクセス、https://www.promptingguide.ai/zh/techniques/cot
20. Chain-of-Thought (CoT) Prompting - Prompt Engineering Guide, 4月 22, 2025にアクセス、https://www.promptingguide.ai/techniques/cot
21. 思维链提示(CoT) - Ultralytics, 4月 22, 2025にアクセス、https://www.ultralytics.com/zh/glossary/chain-of-thought-prompting
22. Chain-of-Thought Prompting Elicits Reasoning in Large Language Models - arXiv, 4月 22, 2025にアクセス、https://arxiv.org/abs/2201.11903
23. Let Claude think (chain of thought prompting) to increase performance - Anthropic API, 4月 22, 2025にアクセス、

https://docs.anthropic.com/en/docs/build-with-claude/prompt-engineering/chain-of-thought
24. 5 Advanced Prompting Techniques to Improve Your LLM App's Responses - athina.ai, 4月 22, 2025にアクセス、https://blog.athina.ai/5-advanced-prompting-techniques-to-improve-your-llm-app-s-responses
25. Role Prompting: Guide LLMs with Persona-Based Tasks, 4月 22, 2025にアクセス、https://learnprompting.org/docs/advanced/zero_shot/role_prompting
26. Assigning Roles to Chatbots - Learn Prompting, 4月 22, 2025にアクセス、https://learnprompting.org/docs/basics/roles
27. Role-Prompting: Does Adding Personas to Your Prompts Really Make a Difference?, 4月 22, 2025にアクセス、https://www.prompthub.us/blog/role-prompting-does-adding-personas-to-your-prompts-really-make-a-difference
28. Structured Output | 🦜 LangChain, 4月 22, 2025にアクセス、https://python.langchain.com/v0.1/docs/modules/model_io/chat/structured_output/
29. Structured Outputs: Everything You Should Know - Humanloop, 4月 22, 2025にアクセス、https://humanloop.com/blog/structured-outputs
30. Crafting Structured {JSON} Responses: Ensuring Consistent Output from any LLM, 4月 22, 2025にアクセス、https://dev.to/rishabdugar/crafting-structured-json-responses-ensuring-consistent-output-from-any-llm-l9h
31. Structured outputs - LangChain, 4月 22, 2025にアクセス、https://python.langchain.com/docs/concepts/structured_outputs/
32. arxiv.org, 4月 22, 2025にアクセス、https://arxiv.org/pdf/2501.11613
33. Best format for structured output for smaller LLMs? XML/JSON or something else? - Reddit, 4月 22, 2025にアクセス、https://www.reddit.com/r/LocalLLaMA/comments/1i5k5qw/best_format_for_structured_output_for_smaller/
34. Reflection Agents - LangChain Blog, 4月 22, 2025にアクセス、https://blog.langchain.dev/reflection-agents/
35. LLM Reflection | AutoGen 0.2 - Microsoft Open Source, 4月 22, 2025にアクセス、https://microsoft.github.io/autogen/0.2/docs/topics/prompting-and-reasoning/reflection/
36. Introduction to Self-Criticism Prompting Techniques for LLMs, 4月 22, 2025にアクセス、https://learnprompting.org/docs/advanced/self_criticism/introduction
37. Reflexion | Prompt Engineering Guide, 4月 22, 2025にアクセス、https://www.promptingguide.ai/techniques/reflexion
38. Prompt Engineering of LLM Prompt Engineering : r/PromptEngineering - Reddit, 4月 22, 2025にアクセス、https://www.reddit.com/r/PromptEngineering/comments/1hv1ni9/prompt_engineering_of_llm_prompt_engineering/