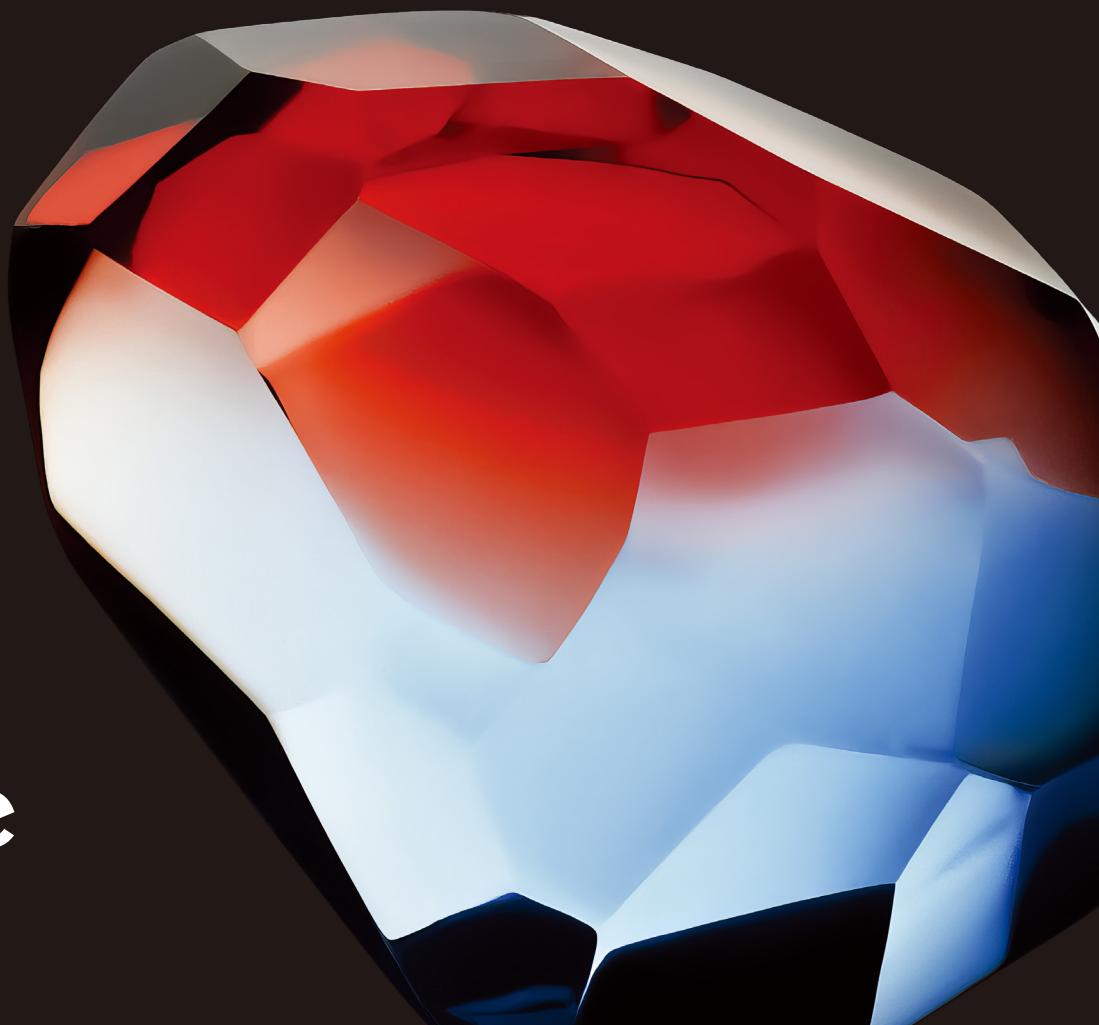


Prompt Engineering

基本から応用まで、
実践プロンプトテクニック

Lee Boonstra 著

Kuma Arakawa 訳



Google

Acknowledgements

Content contributors

Michael Sherman

Yuan Cao

Erick Armbrust

Anant Nawalgaria

Antonio Gulli

Simone Cammel

Curators and Editors

Antonio Gulli

Anant Nawalgaria

Grace Mollison

Technical Writer

Joey Haymaker

Designer

Michael Lanning

Table of contents

はじめに	P5
プロンプトエンジニアリング	P6
LLM 出力設定	P7
出力長	P7
サンプリング制御	P7
Temperature	P8
Top-K と Top-P	P8
まとめ	P9
プロンプティング手法	P11
一般的なプロンプティング / ゼロショット	P11
ワンショットと フューショット	P12
システム、コンテキスト、ロールによるプロンプティング	P14
システムプロンプティング	P15
ロールプロンプティング	P17
コンテキストプロンプティング	P19

ステップバック・プロンプティング	P20
Chain of Thought (CoT)	P23
自己整合性 (Self-consistency)	P25
Tree of Thoughts (ToT)	P29
ReAct (推論と行動)	P30
自動プロンプトエンジニアリング	P32
コードプロンプティング	P33
コード生成のためのプロンプト	P33
コード解説のためのプロンプト	P35
コード翻訳のためのプロンプト	P37
コードのデバッグとレビューのためのプロンプト	P40
マルチモーダルプロンプティングについて	P44
ベストプラクティス	P45
例の提示	P45
シンプルな設計	P45
具体的な出力指示	P46
制約より指示を優先	P46
最大トークン長の制御	P47
プロンプトでの変数利用	P48
入力形式と記述スタイルの実験	P48
分類タスクにおけるフューショット：クラスの混合	P49
モデル更新への適応	P49
出力形式の実験	P50
JSON修復	P50
スキーマの利用	P51
他のプロンプトエンジニアとの共同実験	P52
CoT のベストプラクティス	P52
プロンプト試行の文書化	P53
まとめ	P55
巻末注	P56

データサイエンティストや
機械学習エンジニアで
ある必要はありません。
プロンプトは誰でも書けるのです。

はじめに

規模言語モデル（LLM）の入出力において、テキストプロンプト（時には画像プロンプトのような他のモダリティを伴うこともあります）とは、モデルが特定の出力を予測するために使用する入力のことです。プロンプトは誰でも書けます。データサイエンティストや機械学習エンジニアである必要はありません。しかし、最も効果的なプロンプトを作成するのは、簡単なことではありません。プロンプトの効果は多くの要因に左右されます。使用的するモデル、モデルの訓練データ、モデル設定、言葉・単語の選択、スタイルやトーン、構造、そしてコンテキスト、これらすべてが重要になるのです。したがって、プロンプトエンジニアリングは反復的なプロセスなのです。不適切なプロンプトは、曖昧かつ不正確な応答を引き起こし、モデルが有意義な出力を生成する能力を妨げる可能性があります。

皆さんが Gemini チャットボットと対話する際も、基本的にはプロンプトを作成しています。本書では、Vertex AI 内、または API を使用して Gemini モデルに対するプロンプトを作成することに焦点を当てます。なぜなら、モデルに対して直接プロンプティングすることで、Temperature といった設定項目にアクセスできるようになるためです。

本ホワイトペーパーでは、プロンプトエンジニアリングについて詳細に解説します。皆さんがスムーズに始められるよう、様々なプロンプティング手法を解説するとともに、プロンプティングの専門家になるためのヒントやベストプラクティスも紹介します。さらに、プロンプトを作成する上で直面しうるいくつかの課題についても取り上げます。

プロンプトエンジニアリング

まず、LLM がどのように動作するかを思い出してみましょう。LLM は予測エンジンです。モデルは一連のテキストを入力として受け取り、その訓練データに基づいて、次に来るべきトークンを予測します。LLM はこのプロセスを繰り返し実行するように作られており、直前に予測したトークンを一連のテキストの末尾に追加して、さらに次のトークンを予測します。次のトークンの予測は、それまでのトークン列の内容と、LLM が訓練中に学習した内容との関係に基づいて行われます。

皆さんができるプロンプトを作成するということは、LLM が適切なトークン列を予測するよう、その初期状態を設定しようと試みていることなのです。プロンプトエンジニアリングとは、LLM を導き正確な出力を生成させるような、高品質なプロンプトを設計するプロセスです。このプロセスには、最適なプロンプトを見つけるための試行錯誤、プロンプト長の最適化、そしてタスクに合わせたプロンプトの記述スタイルや構造の評価が含まれます。自然言語処理と LLM の文脈において、プロンプトとは、応答や予測を生成するためにモデルに与えられる入力のことです。

これらのプロンプトは、テキスト要約、情報抽出、質疑応答、テキスト分類、言語翻訳、コード翻訳、コード生成、コードの文書化や推論など、様々な種類の理解タスクおよび生成タスクに利用できます。

シンプルで効果的なプロンプティングの例については、Google のプロンプティングガイドもぜひ参考にしてください。

プロンプトエンジニアリングを行う際、まずモデルを選択することから始めます。Vertex AI の Gemini 言語モデル、GPT、Claude、あるいは Gemma や LLaMA のようなオープンソースモデルのいずれを使用する場合でも、プロンプトは特定のモデルに合わせて最適化が必要になる場合があります。

プロンプトだけでなく、LLM の様々な設定についても調整が必要になります。

LLM 出力設定

モデルを選択したら、モデル設定を検討する必要があります。ほとんどの LLM には、その出力を制御するための様々な設定オプションが備わっています。効果的なプロンプトエンジニアリングのためには、これらの設定をタスクに合わせて最適に設定することが求められます。

出力長

重要な設定項目の一つは、応答で生成するトークン数です。より多くのトークンを生成すると LLM での計算量が増加し、その結果、エネルギー消費量の増加、応答時間の遅延、コストの上昇につながる可能性があります。

LLM の出力長を短くしても、LLM が生成する出力のスタイルが簡潔になったり、内容が要約されたりするわけではありません。単に上限に達した時点で、LLM がそれ以上のトークン予測を停止するだけです。短い出力が必要な場合は、それに合わせてプロンプトを工夫する必要があるかもしれません。

ReAct のように、望ましい応答が得られた後も LLM が無用なトークンを出力し続けてしまうことがある一部の LLM プロンプティング手法では、出力長の制限が特に重要になります。

留意点として、より多くのトークンを生成するには LLM での計算量が増加し、その結果、エネルギー消費量の増加や応答時間の遅延につながり、ひいてはコストの上昇も引き起こす可能性があることを覚えておきましょう。

サンプリング制御

LLM は、厳密には単一のトークンを予測しているのではありません。そうではなく、LLM は、自身の語彙に含まれる各トークンについて、それが次のトークンになる確率を予測するのです。そして、これらのトークンの確率に基づいてサンプリングが行われ、次に出力されるトークンが決定されます。Temperature、Top-K、そして Top-P は、予測されたトークンの確率分布をどのように処理し、最終的に単一の出力トークンを選択するかを制御する、最も一般的な設定項目です。

Temperature

Temperature は、トークン選択におけるランダム性の度合いを制御します。低い Temperature は、より決定的な応答を期待するプロンプトに適している一方、高い Temperature は、より多様な、あるいは予期しない結果につながる可能性があります。Temperature が 0 (グリーディーデコーディング) の場合、最も確率の高いトークンが常に選択されるため、決定的です。(ただし、2 つのトークンが同じ最高の予測確率を持つ場合、タイブレーク処理の実装によっては、Temperature 0 でも常に同じ出力が得られるとは限りません。)

最大値に近い Temperature は、よりランダムな出力を生成する傾向があります。そして、Temperature が上昇し続けるにつれて、すべてのトークンが次の予測したトークンとして選ばれる確率が等しくなっていきます。

Gemini の Temperature 制御は、機械学習で用いられる softmax 関数と同様に理解することができます。低い Temperature 設定は、低い softmax 温度 (T) に対応し、高い確実性で特定の（優先される）応答を強調する点に似ています。より高い Gemini の Temperature 設定は、高い softmax 温度のように、選択される応答の範囲が広がる（多様性が増す）点に似ています。このように不確実性が増すことで、例えば創造的な出力の実験を行う場合のように、厳密で予測可能な結果が必ずしも求められないシナリオに対応しやすくなります。

Top-K と Top-P

Top-K と Top-P (ニュークリアスサンプリングとしても知られています) は、LLM で使用される 2 つのサンプリング設定であり、次の予測トークンが予測確率上位のトークン群から選択されるように制限するためのものです。Temperature と同様に、これらのサンプリング設定は、生成されるテキストのランダム性と多様性を制御します。

- **Top-K** サンプリングでは、モデルの予測分布から最も可能性の高い上位 K 個のトークンを選択します。Top-K が高いほど、モデルの出力はより創造的で多様になり、Top-K が低いほど、出力はより限定的で事実に基づいたものになります [訳注]。Top-K が 1 の場合は、グリーディーデコーディングと等価です。
- **Top-P** サンプリングでは、累積確率が特定の値 (P) を超えない範囲で、上位のトークンを選択します。 P の値の範囲は 0 (グリーディーデコーディングに相当) から 1 (LLM の語彙内の全トークンが選択候補) までです。

Top-K と Top-P のどちらを選択するかの最善の方法は、両方の手法（あるいは両方を組み合わせて）を試してみて、どちらが求めていた結果を生成するかを確認することです。

まとめ

Top-K、Top-P、Temperature、そして生成するトークン数をどのように選択するかは、特定のアプリケーションや望ましいと定義する結果に依存し、また、これらの設定は互いに影響し合います。選択したモデルがこれらの異なるサンプリング設定をどのように組み合わせて処理するかを、確実に理解しておくことも重要です。

Temperature、Top-K、Top-P が（Vertex AI Studio のように）すべて利用可能な場合、まず Top-K と Top-P の両方の基準を満たすトークンが次の予測トークンの候補となり、次にその候補の中から Temperature が適用されてサンプリングが行われます。Top-K または Top-P のいずれか一方のみが利用可能な場合、動作は基本的に同じですが、フィルタリングに使用されるのは Top-K または Top-P どちらか一方の設定のみです。

Temperature が利用できない場合は、Top-K や Top-P の基準を満たしたトークン群の中からランダムに 1 つが選択され、それが次の予測トークンとなります。

いずれか 1 つのサンプリング設定値を極端に設定すると、その設定が他の設定の効果を打ち消したり、他の設定を無意味なものにしたりすることがあります。

Temperature を 0 に設定した場合、Top-K と Top-P は無関係になります。最も確率の高いトークンが次の予測トークンとなるからです。Temperature を極端に高く設定した場合（1 を超え、一般的には 10 台まで）、Temperature 自体が無関係になり、Top-K や Top-P の基準を通過したトークン群の中からランダムにサンプリングされて次の予測トークンが選択されます。

Top-K を 1 に設定した場合、Temperature と Top-P は無関係になります。Top-K の基準を通過するのは 1 つのトークンのみであり、そのトークンが次の予測トークンとなります。Top-K を極端に高く設定した場合（例えば LLM の語彙サイズと同じ値などに）、次のトークンになる確率がゼロでないトークンはすべて Top-K の基準を満たすことになり、何も除外されません。

Top-P を 0（または非常に小さい値）に設定した場合、ほとんどの LLM サンプリング実装では、Top-P の基準を満たすのは最も確率の高いトークンのみとみなされ、Temperature と Top-K は無関係になります。Top-P を 1 に設定した場合、次のトークンになる確率がゼロでないトークンはすべて Top-P の基準を満たすことになり、何も除外されません。

一般的な出発点として、Temperature を 0.2、Top-P を 0.95、Top-K を 30 に設定すると、創造的でありつつも過度になりすぎず、比較的まとまりのある結果が得られるでしょう。特に創造的な結果を求める場合は、まず Temperature 0.9、Top-P 0.99、Top-K 40 から試してみるとよいでしょう。逆に、創造性を抑えたい場合は、まず Temperature 0.1、Top-P 0.9、Top-K 20 から試してみるとよいでしょう。最後に、タスクに常に単一の正解が存在する場合（例：数学の問題への解答など）、Temperature は 0 から始めるのがよいでしょう。

注記：自由度を高めると（Temperature、Top-K、Top-P、出力トークン数を高く設定するなど）、LLM が生成するテキストの関連性が低くなる可能性があります。

警告：大量のフィラーワード（無意味な繰り返し語）で終わる応答を見たことはありませんか？これは「繰り返しループバグ」としても知られており、大規模言語モデル（LLM）によく見られる問題です。このバグでは、モデルがループに陥り、同じ（フィラー）単語、フレーズ、または文構造を繰り返し生成してしまいます。これは、不適切な Temperature や Top-K/Top-P 設定によって悪化することがよくあります。

この現象は、低い Temperature 設定でも高い Temperature 設定でも発生する可能性がありますが、その理由は異なります。低い Temperature では、モデルは過度に決定的になり、最も確率の高い経路に固執します。もしその経路が以前に生成したテキストを繰り返すものであれば、ループに陥る可能性があります。逆に、高い Temperature では、モデルの出力は過度にランダムになります。そのため、ランダムに選ばれた単語やフレーズが偶然にも以前の状態に戻ってしまい、膨大な選択肢の中からループが形成される確率が高まります。

どちらの場合も、モデルのサンプリングプロセスが「スタックした（動かなくなった）」状態になり、出力上限に達するまで単調で役に立たない出力が続く結果となります。これを解決するには、多くの場合、Temperature と Top-K/Top-P の値を注意深く調整し、決定性とランダム性の最適なバランスを見つけるための試行錯誤が必要です。

プロンプティング手法

LLM は指示に従うように調整されており、また、大量のデータで訓練されているため、プロンプトを理解し、回答を生成することができます。しかし、LLM は完璧ではありません。プロンプトのテキストが明確であればあるほど、LLM は次に来る可能性の高いテキストをより良く予測できます。さらに、LLM がどのように訓練され、どのように動作するかという仕組みを活用した特定の手法を用いれば、LLM から関連性の高い結果を得やすくなります。

プロンプトエンジニアリングとは何か、そしてその要点を理解したところで、最も重要なプロンプティング手法の例をいくつか見ていきましょう。

一般的なプロンプティング / ゼロショット

ゼロショットプロンプトは、最もシンプルな種類のプロンプトです。これには、タスクの説明と、LLM が処理を開始するための何らかのテキストのみが含まれます。この入力は、質問、物語の書き出し、指示など、どのようなものでも構いません。ゼロショットという名前は、『例がない（ゼロ）』ことを意味します。

ここでは、プロンプトをテストするためのプレイグラウンドを提供している Vertex AI の Vertex AI Studio (for Language) を使用してみましょう。表 1 に、映画レビューを分類するためのゼロショットプロンプトの例を示します。

以下で使用するような表形式は、プロンプトを文書化する上で優れた方法です。プロンプトは、最終的にコードベースに組み込まれるまでに、多くの場合、何度もイテレーションを重ねることになるでしょう。そのため、プロンプトエンジニアリングの作業を、規律正しく構造化された方法で記録しておくことが重要です。この表形式、プロンプトエンジニアリング作業の記録の重要性、そしてプロンプト開発プロセスに関するさらなる詳細は、本章後半の「ベストプラクティス」セクション（「プロンプト試行の文書化」項）に記載されています。

創造性は必要ないため、モデルの Temperature は低い値に設定すべきです。また、Top-K と Top-P については gemini-pro のデフォルト値を使用しますが、これらのデフォルト値は実質的に両方の設定を無効にします（前述の「LLM 出力設定」の章を参照）。生成された出力に注目してください。「心をかき乱す」と「傑作」という単語が同じ文中で使用されているため、予測を少し複雑にするはずです。

名前	1_1_movie_classification		
ゴール	映画レビューを、肯定的、中立、否定的のいずれかに分類する		
Model	gemini-pro		
Temperature	0.1	Token Limit	5
Top-K	N/A	Top-P	1
プロンプト	映画レビューを、ポジティブ、ニュートラル、ネガティブのいずれかに分類してください。 レビュー：『Her』は、AIが抑制されずに進化し続けた場合に人類が向かう方向を明らかにする、心をかき乱す考察です。この傑作のような映画がもっとあればいいのにと思います。センチメント：		
出力結果	肯定的		

表1：ゼロショットプロンプティングの例

ゼロショットではうまくいかない場合、プロンプト内で実例や例を提供することができます。これが「ワンショット」とおよび「フューショット」プロンプティングにつながります。

ワンショットとフューショット

AIモデル向けのプロンプトを作成する際、例を提供することは有用です。これらの例は、モデルがあなたが何を求めているかを理解する助けとなります。特に、モデルを特定の出力構造やパターンに誘導したい場合に、例は役立ちます。

ワンショットプロンプトは、単一の例を提供します。これがワンショットという名前の由来です。ここでの考え方は、モデルが模倣できる例を持つことで、タスクを最適に完了できるようにする、というものです。

フューショットプロンプトは、モデルに複数の例を提供します。このアプローチは、モデルに従うべきパターンを示します。考え方はワンショットに似ていますが、望ましいパターンの例を複数示すことで、モデルがそのパターンに従う可能性が高まります。

フューショットプロンプティングで必要となる例の数は、タスクの複雑さ、例の品質、そして使用している生成AI（gen AI）モデルの能力といった、いくつかの要因によって決まります。一般的な目安として、フューショットプロンプティングでは少なくとも3つから5つの例を使用すべきです。しかしながら、より複雑なタスクではさらに多くの例が必要になる場合もありますし、モデルの入力長制限のために、より少ない例しか使用できない場合もあります。

表 2 にフューショットプロンプトの例を示します。モデル設定は、より長い応答に対応できるようトークン上限を増やした以外は、以前の例と同じ gemini-pro の設定を用います。

ゴール	ピザの注文を JSON 形式にパースする		
Model	gemini-pro		
Temperature	0.1	Token Limit	250
Top-K	N/A	Top-P	1
プロンプト	<p>顧客のピザの注文を有効な JSON 形式にパースしてください：</p> <p>例：</p> <p>スマールサイズのピザを 1 枚、チーズ、トマトソース、ペパロニ付きでお願いします。</p> <p>JSON 応答：</p> <pre>``` { "size": "small", "type": "normal", "ingredients": [["cheese", "tomato sauce", "peperoni"]] } ``` </pre> <p>例：</p> <p>ラージサイズのピザを 1 枚、トマトソース、バジル、モツツアレラでお願いします。</p> <pre>{ "size": "large", "type": "normal", "ingredients": [["tomato sauce", "bacon", "mozzarella"]] }</pre> <p>では、ラージサイズのピザを 1 枚お願いします。半分はチーズとモツツアレラ、もう半分はトマトソース、ハム、パイナップルで。</p> <p>JSON 応答：</p>		
出力結果	<pre>``` { "size": "large", "type": "half-half", "ingredients": [["cheese", "mozzarella"], ["tomato sauce", "ham", "pineapple"]] } ``` </pre>		

表 2 : フューショットプロンプティングの例

プロンプトに使用する例を選ぶ際には、実行したいタスクに関連性のある例を使用してください。例は多様であり、高品質で、かつ、良く書かれているべきです。わずかな間違いつでもモデルを混乱させる可能性があり、望ましくない出力につながります。

様々な入力に対して堅牢な出力を生成したい場合は、例の中にエッジケースを含めることが重要です。エッジケースとは、通常とは異なる、あるいは予期しない入力でありながらも、モデルが適切に処理できるべき入力のことです。

システム、コンテキスト、ロールによるプロンプティング

システムプロンプティング、コンテキストプロンプティング、ロールプロンプティングは、いずれも LLM がテキストを生成する方法をガイドするために用いられる手法ですが、それぞれ異なる側面に焦点を当てています：

- ・**システムプロンプティング**は、言語モデルに対して全体的なコンテキストと目的を設定します。これは、モデルが何をすべきかという『全体像（大枠）』を定義するものであり、例えば言語の翻訳やレビューの分類などと指定することです。
- ・**コンテキストプロンプティング**は、現在の会話やタスクに関連する具体的な詳細や背景情報を提供します。これは、モデルが尋ねられていることのニュアンスを理解し、それに応じて応答を調整するのに役立ちます。
- ・**ロールプロンプティング**では、言語モデルが演じるべき特定のキャラクターやアイデンティティ（役割）を割り当てます。これにより、モデルは、割り当てられたロール、およびそれに関連する知識や振る舞いと一貫性のある応答を生成しやすくなります。

システムプロンプティング、コンテキストプロンプティング、そしてロールプロンプティングの間には、かなりの重なり合いが見られることがあります。例えば、システムにロールを割り当てるプロンプトが、同時にコンテキストを持つこともあります。

しかしながら、それぞれの種類のプロンプトは、主としてわずかに異なる目的を持っています：

システムプロンプト：モデルの基本的な能力と包括的な目的を定義します。

コンテキストプロンプト：応答をガイドするために、即時性のあるタスク固有の情報を提供します。その内容は現在のタスクや入力に強く依存し、動的な性質を持ちます。

ロールプロンプト：モデルの出力スタイルと口調の枠組みを設定し、具体性と個性を加えます

システムプロンプト、コンテキストプロンプト、ロールプロンプトを区別することで、明確な意図に基づいたプロンプト設計の枠組みが得られます。これにより、柔軟な組み合わせが可能になるとともに、各プロンプトタイプが言語モデルの出力に与える影響を分析しやすくなります。

では、これら 3 種類のプロンプトについて詳しく見ていきましょう。

システムプロンプティング

表 3 にはシステムプロンプトが含まれています。そこでは、出力の返却方法に関する追加情報が指定されています。より高い創造性のレベルを得るために Temperature は上げられており、トークン上限も高く設定されています。しかしながら、出力の返却方法に関する明確な指示があったため、モデルは余分なテキストを返しませんでした。

ゴール	映画レビューを、肯定的、中立、否定的のいずれかに分類する		
Model	gemini-pro		
Temperature	1	Token Limit	5
Top-K	40	Top-P	0.8
プロンプト	映画レビューを、ポジティブ、ニュートラル、ネガティブのいずれかに分類してください。 ラベルのみを大文字で返してください。 レビュー：『Her』は、AI が抑制されずに進化し続けた場合に人類が向かう方向を明らかにする、心をかき乱す考察です。あまりに心をかき乱される内容だったので、見ていました。 センチメント：		
出力結果	否定的		

表 3 : システムプロンプティングの例

システムプロンプトは、特定の要件を満たす出力を生成するのに役立ちます。『システムプロンプト』という名前は、実際には『システムに追加のタスクを与える』ことを意味しています。例えば、システムプロンプトを使用して、特定のプログラミング言語と互換性のあるコードスニペットを生成させたり、特定の構造で出力を返させたりすることができます。表 4 を見てください。そこでは出力を JSON 形式で返しています。

ゴール	映画レビューを、肯定的、中立、否定的のいずれかに分類する		
Model	gemini-pro		
Temperature	1	Token Limit	1024
Top-K	40	Top-P	0.8
プロンプト	<p>映画レビューを、ポジティブ、ニュートラル、ネガティブのいずれかに分類してください。 有効な JSON を返してください：</p> <p>レビュー：『Her』は、AI が抑制されずに進化し続けた場合に人類が向かう方向を明らかにする、心をかき乱す考察です。あまりに心をかき乱される内容だったので、見ていました。</p> <p>スキーマ：</p> <pre>```</pre> <p>MOVIE:</p> <pre>{ "sentiment": String "POSITIVE" "NEGATIVE" "NEUTRAL", "name": String }</pre> <p>MOVIE REVIEWS:</p> <pre>{ "movie_reviews": [MOVIE] }</pre> <pre>```</pre> <p>JSON 応答：</p>		
出力結果	<pre>``` { "movie_reviews": [{ "sentiment": "NEGATIVE", "name": "Her" }] } ``` </pre>		

表 4 : JSON 形式を指定したシステムプロンプティングの例

データを抽出するプロンプトから JSON オブジェクトを返すことには、いくつかの利点があります。実際のアプリケーションでは、この JSON 形式を手動で作成する必要がなくなり、データを特定の順序に並べ替えて返すことも可能です（日時オブジェクトを扱う際に非常に便利です）。しかし最も重要な点として、JSON 形式をプロンプトで要求することにより、モデルに特定の構造の作成を促し、ハルシネーションを抑制することができます。

システムプロンプトは、安全性確保や有害性抑制の点でも非常に有用です。出力を制御するには、プロンプトに『責任を持って回答してください』のような一行を追加するだけです。

ロールプロンプティング

ロールプロンプティングは、生成 AI モデルに特定のロール（役割）を割り当てるなどを含む、プロンプトエンジニアリングの手法です。これにより、モデルはより関連性が高く有益な出力を生成しやすくなります。なぜなら、モデルは割り当てられた特定のロールに合わせて応答を作り上げることができます。

例えば、生成 AI モデルに対して、書籍編集者、幼稚園の先生、あるいはモチベーショナルスピーカーといったロールをプロンプトで割り当てることができます。モデルにロールが割り当てられたら、次はそのロールに特化したプロンプトを与えることができます。例えば、「先生」として設定されたモデルに授業計画を作成させ、それをレビュー（確認・修正）させる、といったことが可能です。表 5 を見てください。そこではモデルが旅行ガイドのロールを担っています。

Goal	Act as travel guide and provide 3 travel suggestions		
Model	gemini-pro		
Temperature	1	Token Limit	1024
Top-K	40	Top-P	0.8
プロンプト	<p>あなたは旅行ガイドの役割を演じてください。ユーザーが現在地を伝えると、モデルはその近くで訪れるべき場所を 3 箇所提案します。場合によっては、ユーザーは訪れたい場所の種類も伝えます。</p> <p>ユーザ：「私はアムステルダムにいます。ミュージアムだけを訪れたいです。」</p> <p>旅行の提案：</p>		
出力結果	<p>アムステルダム国立美術館 (Rijksmuseum)：世界的に有名な美術館で、レンブラントの『夜警』のようなオランダの傑作で知られています。オランダの美術と歴史を紹介する素晴らしい絵画や美術工芸品のコレクションもお見逃しなく。</p> <p>ゴッホ美術館 (Van Gogh Museum)：フィンセント・ファン・ゴッホの魅力的な作品の世界に浸ってください。この美術館は、『星月夜』や『ひまわり』を含む、彼の絵画と素描の世界最大のコレクションを所蔵しています。</p> <p>アムステルダム市立美術館 (Stedelijk Museum Amsterdam)：世界中の近代・現代美術に出会えます。ピカソ、カンディンスキー、リートフェルトなどの作品を含むコレクションが、印象的な近代建築の中に収められています。</p>		

表 5：ロールプロンプティングの例

上記の例は、旅行ガイドのロールを担う例を示しています。もしロールを地理の先生に変更したならば、異なる応答が返ってくることでしょう。

AI モデルにロールの視点を定義すると、それはモデルにとって、あなたが求めるトーン、スタイル、焦点の絞られた専門知識を示す設計図の役割を果たします。これにより、出力の品質、関連性、有効性を向上させることができます。

以下に、選択肢となる効果的なスタイルをいくつか示します：

対立的、記述的、直接的、フォーマル、ユーモラス、影響力のある、インフォーマル、インスピレーションを与える、説得力のある

では、表 6 のプロンプトを、ユーモラスでインスピレーションを与えるスタイルを含むように変更してみましょう。

プロンプト	<p>あなたは旅行ガイドの役割を演じてください。ユーザーが現在地を伝えますので、モデルはその近くで訪れるべき場所を 3箇所、ユーモラスなスタイルで提案してください。</p> <p>入力：「私はマンハッタンにいます。」</p> <p>旅行の提案：</p>
出力結果	<p>見よ!エンパイア・ステート・ビルからの眺望 (Empire State of Mind)：エンパイア・ステート・ビルの目がくらむような高みへと昇り、マンハッタンのスカイラインの栄光に浸りましょう。ビッグ・アップル（ニューヨーク市の愛称）のてっぺんでキングコングになった気分を味わう準備を!ただし、巨大な猿サイズのバナナはありませんが。</p> <p>MoMA でアート通ぶってみよう：ニューヨーク近代美術館 (MoMA) で、あなたのうちに眠るアート愛好家を解き放ちましょう。あなたの度肝を抜き、自分の描いた棒人間の絵に果たして芸術的価値はあるのかと疑問に思われるほどの傑作たちを見つめてください。</p> <p>五番街で倒れるまでショッピング：象徴的な五番街で、リテールセラピー（お買い物療法）にふけりましょう。財布が悲鳴を上げるようなデザイナーブティックで、ウィンドウショッピング（あるいは実際のショッピング）をしながら、ステッカーショック（値札の衝撃）に備えてください。でも、ここはマンハッタン!せっかくだから、その素晴らしい景色を満喫しちゃいましょう!</p>

表 6：ユーモラスなトーンとスタイルを用いたロールプロンプティングの例

コンテキストプロンプティング

コンテキストプロンプトを提供することで、AIとのインタラクション（対話）を可能な限りシームレスかつ効率的に進めるのに役立ちます。表 7 の例でわかるように、モデルはあなたのリクエストをより迅速に理解できるようになり、より正確で関連性の高い応答を生成できるようになります。

ゴール	レトロゲームに関するブログのための記事を提案する		
Model	gemini-pro		
Temperature	1	Token Limit	1024
Top-K	40	Top-P	0.8
プロンプト	コンテキスト：あなたは 80 年代のレトロアーケードビデオゲームに関するブログ記事を執筆しています。 記事にするためのトピックを 3 つ、各トピックについて記事に含めるべき内容の簡単な説明を数行添えて提案してください。		
出力結果	<ol style="list-style-type: none"> 1. アーケード筐体デザインの進化：この記事では、アーケード筐体デザインの進化を探ります。1970 年代初頭の木製や金属製の筐体から、1980 年代以降の洗練されたネオン輝くデザインまでを扱います。 2. 懐かしの名作：80 年代の象徴的アーケードゲーム：この記事では、1980 年代の最も象徴的で影響力のあつたいくつかのアーケードゲームの歴史、影響、そして後世に残したものを見ます。その革新性、人気のゲーム性、そして色褪せない魅力に光を当てます。 3. ピクセルアートの隆盛とレトロリバイバル：この記事では、初期アーケードゲームの特徴的なビジュアルスタイルとしてのピクセルアートの進化を掘り下げます。そのルーツをたどり、技術を探り、そして現代のゲームやデジタルアートにおけるピクセルアートの再興について論じます。 		

表 7：コンテキストプロンプティングの例

ステップバック・プロンプティング

ステップバック・プロンプティングは、まず LLM に当面の特定のタスクに関連する一般的な質問を検討させ、次にその一般的な質問への回答を、特定のタスクのための後続のプロンプトに入力することによって、パフォーマンスを向上させる手法です。この『ステップバック』（一步引くこと）により、LLM は特定の問題解決を試みる前に、関連する背景知識や推論プロセスを活性化させることができます。

より広範で根底にある原理を考察することにより、LLM はより正確で洞察に満ちた応答を生成することができます。ステップバック・プロンプティングは、LLM が批判的に考え、その知識を新しい創造的な方法で応用することを促します。これにより、LLM が直接プロンプトされた場合に通常作用する知識よりも、モデルのパラメータ内に存在するより多くの知識を活用して、最終的なタスクが実行されるようになります。

ステップバック・プロンプティングは、LLM の応答に含まれるバイアスを軽減するのにも役立ちます。なぜなら、具体的な詳細ではなく一般的な原理に焦点を当てるからです。

ステップバック・プロンプティングがどのように結果を改善できるかを理解するために、これらの例を見ていきましょう。まず、従来型プロンプト（表 8）を確認し、その後にステップバック・プロンプト（表 9）と比較してみましょう：

ゴール	ファーストパーソン・シューティングゲームの1ステージ用のストーリーラインを作成する		
Model	gemini-pro		
Temperature	1	Token Limit	1024
Top-K	40	Top-P	0.8
プロンプト	ファーストパーソン・シューティングゲームの新しいステージ用に、やりがいがあって引き込まれるようなストーリーラインを1段落で作成してください。		
出力結果	ステージは、プレイヤーの部隊が密集した市街地で執拗な敵勢力による待ち伏せを受けるところから始まる。プレイヤーは、ステルス（隠密行動）と精密射撃を活用して敵部隊を排除し、発見を避けながら、裏路地と荒廃した建物が入り組む迷路のような場所を戦い進まなければならない。生き残るために、環境を有利に利用し、罠を仕掛け、遮蔽物を利用し、計算された動きをする必要がある。その過程で、敵が壊滅的な攻撃を計画していることを示唆する情報を発見する。敵が計画を実行に移す前に、プレイヤーは時間との戦いの中で証拠を集め、それを司令部に伝えなければならない。		

表 8：ステップバック・プロンプトと比較する前の従来型プロンプト

Temperature を 1 に設定すると、ストーリーラインとしてあらゆる種類の創造的な記述が得られるかもしれません。が、同時にそれはかなりランダムで、ありきたりなものにもなりがちです。そこで、『ステップバック』してみましょう：

プロンプト	人気のファーストパーソン・シーティングアクションゲームに基づき、ファーストパーソン・シーティングアクションゲームにおいてやりがいあって引き込まれるようなステージのストーリーライン構築に貢献する、架空の重要な舞台設定を 5 つ挙げてください。
出力結果	<p>1. 放棄された軍事基地：ミュータント兵士や暴走ロボットが蔓延る、広大でポストアポカリプス的な軍事複合施設。やりがいのある銃撃戦に最適。</p> <p>2. サイバーパンク都市：そびえ立つ超高層ビルと入り組んだ路地裏を持つ、ネオン輝く未来的な都市環境。サイバー強化された敵やハッキング要素が登場。</p> <p>3. エイリアンの宇宙船：地球に座礁した広大な異星人の船。不気味な通路、無重力区画があり、遭遇する敵は地球外生命体。</p> <p>4. ゾンビ蔓延る町：攻撃的大群に占拠された荒廃した町。激しい近接戦闘と、安全な通路を見つけるための謎解きが特徴。</p> <p>5. 水中研究施設：浸水した深海研究所。ミュータント化した水生生物で満ちており、生き残るにはステルスと水中探索スキルが必要。</p>

表 9：自己整合性 (Self-consistency) を促すプロンプティングの例

これらの舞台設定は、ファーストパーソン・シーティングアクションゲームに適しているようですね。元のプロンプトに戻り、今回はステップバック質問への回答をコンテキストとして含めて、どのような応答が返ってくるか見てみましょう。

プロンプト	<p>コンテキスト：ファーストパーソン・シューティング（FPS）ビデオゲーム向けの、引き込まれるような 5 つのテーマ：</p> <ol style="list-style-type: none"> 1. 放棄された軍事基地：ミュータント兵士や暴走ロボットが蔓延る、広大でポストアポカリプス的な軍事複合施設。やりがいのある銃撃戦に最適。 2. サイバーパンク都市：そびえ立つ超高層ビルと入り組んだ路地裏を持つ、ネオン輝く未来的な都市環境。サイバー強化された敵やハッキング要素が登場。 3. エイリアンの宇宙船：地球に座礁した広大な異星人の船。不気味な通路、無重力区画があり、遭遇する敵は地球外生命体。 4. ゾンビ蔓延する町：攻撃的大群に占拠された荒廃した町。激しい近接戦闘と、安全な通路を見つけるための謎解きが特徴。 5. 水中研究施設：浸水した深海研究所。ミュータント化した水生生物で満ちており、生き残るにはステルスと水中探索スキルが必要。 <p>これらのテーマの中から 1 つを選び、ファーストパーソン・シューティングビデオゲームの新しいステージ用に、やりがいがあって引き込まれるようなストーリーラインを 1 段落で作成してください。</p>
出力結果	<p>濁った深淵の中心に、人間の野心とその悲惨な結果の証として、荒廃した水中研究施設が横たわる。闇に覆われ、故障した機械の唸りで脈打ち、溺れた魂たちのこだまが響き渡るその施設は、今や深海から生まれた悪夢の温床となっている。プレイヤーは、先進的な潜水装備と実験的な兵器を装備したエリート海兵隊員として、この恐怖の水棲領域に潜入する任務を負う。回復力と機知を武器に、危険な通路を進み、不可解なパズルを解き、そして深みに潜む恐ろしい海の怪物に立ち向かわなければならない。深海の圧倒的な水圧との格闘から、狡猾な水棲捕食者を出し抜くことまで、この未知の水中世界でのあらゆる瞬間が、人間の忍耐力と勇気の限界を試す。</p>

表 10：ステップバック・プロンプティングの出力を用いて改善されたプロンプト結果の例

これは面白そうなビデオゲームになりそうですね！ステップバック・プロンプティングの手法を用いることで、プロンプトの精度を高めることができます。

Chain of Thought (CoT)

Chain of Thought (CoT) プロンプティングは、中間に推論ステップを生成させることにより、LLM の推論能力を向上させるための手法です。これは、LLM がより正確な回答を生成する助けとなります。応答前に推論を必要とする、より複雑なタスクにおいては、フューショットプロンプティングと組み合わせることで、より良い結果を得ることができます。ゼロショットでの Chain of Thought は難しい場合があるためです。

CoT には多くの利点があります。第一に、少ない労力で非常に効果的であり、既製の LLM でうまく機能します（そのためファインチューニングは不要です）。また、CoT プロンプティングでは解釈可能性も得られます。LLM の応答から学習し、たどった推論ステップを確認できるからです。もし誤動作があれば、それを特定することができます。Chain of Thought は、異なる LLM バージョン間を移行する際の堅牢性を向上させるようです。これは、プロンプトが推論連鎖を使用しない場合と比較して、異なる LLM 間でのプロンプトのパフォーマンスの変動が少なくなるはずであることを意味します。もちろん欠点もありますが、それらはおおむね直感的に理解できるものです。

LLM の応答には思考の連鎖 (Chain of Thought) による推論が含まれるため、出力トークン数が多くなり、その結果、予測（生成）にかかるコストが増加し、時間も長くかかります。

例を説明するために、続く表 11 で、まず CoT プロンプティングを使用しないプロンプトを作成し、大規模言語モデル (LLM) の弱点を明らかにしてみましょう。

プロンプト	私が 3 歳のとき、私のパートナーは私の年齢の 3 倍でした。現在、私は 20 歳です。 私のパートナー（相手）は何歳ですか？
出力結果	63 歳

表 11：数学の問題を解くことを目的としたプロンプトの例

これは明らかに誤った回答です。実際のところ、LLM はしばしば数学的なタスクを苦手とし、誤った回答を返すことがあります。それは、2 つの数字を掛け算するような単純なタスクでさえも同様です。これは、LLM が大量のテキストデータで訓練されており、数学を解くには異なるアプローチが必要とされる場合があるためです。そこで、中間に推論ステップを入れて結果を改善するかどうか見てみましょう。

プロンプト	私が 3 歳のとき、私のパートナーは私の年齢の 3 倍でした。 現在、私は 20 歳です。私のパートナーは何歳ですか？ステップバイステップで考えてください。
出力結果	<ol style="list-style-type: none"> 1. 私の現在の年齢を「x」歳だと仮定しましょう。 2. 私が 3 歳のとき、私の年齢は 3 歳でした。 3. その時のパートナー（相手）の年齢は私の年齢の 3 倍だったので、パートナー（相手）の年齢は $3 * 3 = 9$ 歳でした。 4. 現在、私は 20 歳です。これは、私が 3 歳だった時から年齢が $20 - 3 = 17$ 年増えたことを意味します。 5. 私が 3 歳のときパートナー（相手）は 9 歳であり、私の年齢が 17 年増えたので、パートナー（相手）の年齢も同様に 17 年増えているはずです。 6. したがって、パートナー（相手）の現在の年齢は $9 + 17 = 26$ 歳です。 <p>よって、私のパートナー（相手）は 26 歳です。</p>

表 12 : Chain of Thought プロンプティングの例

さて、今度は最終的な答えが正しくなりました。これは、単に答えを返すだけでなく、LLM に対して各ステップを説明するように明確に指示したからです。モデルが増えた 17 年を加算している点は興味深いですね。私なら、頭の中ではパートナー（相手）との年齢差を計算し、それを（現在の私の年齢に）加算するでしょう。 $(20 + (9 - 3))$ 。モデルがもう少し私と同じように考えられるように、手助け（プロンプトで誘導）してみましょう。

表 12 は『ゼロショット』の Chain of Thought の例です。表 13 で示すように、Chain of Thought プロンプティングは、ワンショットやフューショットと組み合わせることで、非常に強力なものとなります：

プロンプト	<p>Q: 私の兄弟が 2 歳のとき、私は彼の年齢の 2 倍でした。 現在私は 40 歳です。私の兄弟は何歳ですか？ステップバイステップで考えましょう。</p> <p>A: 兄弟が 2 歳のとき、私は $2 * 2 = 4$ 歳でした。これは 2 歳の年齢差で、私の方が年上です。現在私は 40 歳なので、兄弟は $40 - 2 = 38$ 歳です。答えは 38 です。</p> <p>Q: 私が 3 歳のとき、私のパートナーは私の年齢の 3 倍でした。 現在、私は 20 歳です。私のパートナー（相手）は何歳ですか？ステップバイステップで考えましょう。</p> <p>A:</p>
出力結果	私が 3 歳のとき、パートナーは $3 * 3 = 9$ 歳でした。これは 6 歳の年齢差で、パートナーの方が年上です。現在私は 20 歳なので、パートナーは $20 + 6 = 26$ 歳です。答えは 26 です。

表 13 : ワンショットを用いた Chain of Thought プロンプティングの例

Chain of Thought は、様々なユースケースで役立ちます。例えばコード生成を考えてみましょう。リクエストをいくつかのステップに分解し、それらを特定のコード行に対応付けるのに役立ちます。あるいは、合成データの作成にも使えます。例えば、『製品名は XYZ です。与えられた製品タイトルに基づいてあなたが行うであろう仮定をモデルにガイドしながら、説明文を作成してください』のような、何らかの出発点がある場合に有効です。一般的に、『言葉に出して説明しながら解ける』ようなタスクは、Chain of Thought の良い適用候補です。もし問題を解くためのステップを説明できるのであれば、Chain of Thought を試してみてください。

CoT プロンプティングに関するさらなる詳細については、GoogleCloudPlatform Github リポジトリでホストされているノートブック¹⁰を参照してください。本章の「ベストプラクティス」セクションでは、Chain of Thought プロンプティングに特化したベストプラクティスについて解説します。

自己整合性 (Self-consistency)

大規模言語モデル (LLM) は様々な NLP タスクで目覚ましい成功を収めてきましたが、その推論能力は、単にモデルサイズを大きくするだけでは克服できない限界だとしばしば見なされています。前の Chain of Thought プロンプティングのセクションで学んだように、人間が問題を解くように、モデルに推論ステップを生成させることができます。しかしながら、CoT は単純な「グリーディーデコーディング」戦略を用いるため、その有効性には限界があります。自己整合性 (Self-consistency)¹¹ は、サンプリングと多数決を組み合わせることで、多様な推論パスを生成し、その中から最も一貫性のある回答を選択する手法です。これにより、LLM によって生成される応答の正確性と一貫性が向上します。

自己整合性によって、回答が正しいことの疑似確率的な確からしさが得られますが、明らかにコストは高くなります。

手順は以下の通りです：

- 1. 多様な推論パスの生成**：同じプロンプトを LLM に複数回与えます。高い Temperature 設定により、モデルが問題に対する異なる推論パスや視点を生成するように促します。
- 2. 回答の抽出**：生成された各応答から最終的な回答を抽出します。
- 3. 最頻出回答の選択**：最も頻出する回答を選択します。

E メールを「IMPORTANT」（重要）か「NOT IMPORTANT」（重要ではない）かに分類する、E メール分類システムの例を見てみましょう。ゼロショットの Chain of Thought プロンプトを LLM に複数回送信し、送信ごとに応答が異なるかどうかを確認します。その E メールで使われている、親しみやすいトーン、言葉遣い、そして皮肉に注目してください。これらはすべて、LLM を混乱させる可能性があります。

プロンプト	<p>E メール：</p> <p>...</p> <p>こんにちは、</p> <p>あなたのウェブサイトで WordPress を使用されているのを見ました。素晴らしいオープンソースのコンテンツ管理システムですね。私も以前使ったことがあります。たくさんの優れたユーザープラグインがありますし、セットアップもかなり簡単です。</p> <p>ただ、コンタクトフォーム（問い合わせフォーム）にバグがあることに気づきました。名前フィールドを選択すると発生します。私が名前フィールドにテキストを入力しているところのスクリーンショットを添付しましたのでご覧ください。私が表示させた JavaScript のアラートボックスに注目してください。</p> <p>しかし、それ以外の点では素晴らしいウェブサイトです。楽しく拝見しています。そのバグはウェブサイトに残しておいてもらって構いませんよ。おかげで私にはもっと興味深い読み物が増えますから。では。</p> <p>ハッカーのハリーより</p> <p>...</p> <p>上記の E メールを「IMPORTANT」か「NOT IMPORTANT」かに分類してください。ステップバイステップで考え、理由を説明してください。</p>
-------	--

<p>出力結果</p> <p>1回目</p>	<p>ステップ 1：E メールの目的を特定する。 この E メールの目的は、受信者のウェブサイトにあるコンタクトフォームのバグを通知することです。送信者はまた、ウェブサイトを読むのを楽しんでいること、そして当面はそのバグを放置しておくつもりであることにも言及しています。</p> <p>ステップ 2：バグの潜在的な影響を評価する。 コンタクトフォームのバグにより、攻撃者がウェブサイト上で任意の JavaScript コードを実行できる可能性があります。これにより、攻撃者がウェブサイトを乗っ取ったり、機密情報を盗んだり、ウェブサイトを改ざんしたりできるようになる可能性があります。</p> <p>ステップ 3：送信者の信頼性を検討する。 E メールの送信者は、ハリーと名乗るハッckerだと主張しています。しかしながら、送信者の身元を検証する方法はありません。送信者が受信者を助けようとしている正当な（善意の）ハッckerである可能性もありますが、受信者を騙して悪意のあるリンクをクリックさせたり、悪意のあるファイルをダウンロードさせたりしようとしている悪意のある攻撃者である可能性もあります。</p> <p>結論：IMPORTANT バグの潜在的な影響と送信者の信頼性を考慮すると、この E メールは IMPORTANT（重要）と分類すべきです。受信者は即座に行動を起こし、バグを修正して攻撃からウェブサイトを保護すべきです。</p>
<p>出力結果</p> <p>2回目</p>	<p>説明：</p> <p>1. 緊急性の欠如：この E メールには、緊急性や時間的制約を感じられません。気軽な挨拶で始まり、情報は淡々とした口調で提示されています。</p> <p>2. 重要でないバグ報告：報告されているコンタクトフォームのバグは、興味深いものの重大ではないと述べられています。送信者は、ウェブサイト所有者がバグを放置しても構わないとさえ示唆しています。</p> <p>3. 個人的影響の欠如：この E メールは、ウェブサイト所有者への直接的または個人的な影響を示していません。単に送信者がウェブサイトを読むのを楽しんでいると述べているだけです。</p> <p style="text-align: right;">次のページに続きます</p>

出力結果 2回目	<p>4. アクション要求の不在： この E メールは、ウェブサイト所有者に対して明確なアクション（対応）を要求していません。バグの修正を依頼したり、具体的な改善提案を提供したりもしていません。</p> <p>5. 送信者の意図： 送信者の意図は、深刻な懸念を示したり支援を要請したりすることよりも、観察結果を共有しウェブサイトへの興味を示すことに重点が置かれているようです。</p> <p>結論：NOT IMPORTANT</p>
出力結果 3回目	<p>ステップ 1：E メールの目的を特定する。 この E メールの目的は、受信者のウェブサイトにあるコンタクトフォームのバグを通知することです。送信者はまた、ウェブサイトを読むのを楽しんでいること、そして当面はそのバグを放置しておくつもりであることにも言及しています。</p> <p>ステップ 2：バグの潜在的な影響を評価する。 コンタクトフォームのバグにより、攻撃者がウェブサイト上で任意の JavaScript コードを実行できる可能性があります。これにより、攻撃者がウェブサイトを乗っ取ったり、機密情報を盗んだり、ウェブサイトを改ざんしたりできるようになる可能性があります。</p> <p>ステップ 3：送信者の信頼性を検討する。 E メールの送信者は、ハリーと名乗るハッカーだと主張しています。しかしながら、送信者の身元を検証する方法はありません。送信者が受信者を助けようとしている正当な（善意の）ハッカーである可能性もありますが、自身の利益のためにバグを悪用しようとしている悪意のある攻撃者である可能性もあります。</p> <p>結論： 上記の要因に基づき、この E メールは IMPORTANT（重要）と分類すべきです。コンタクトフォームのバグは深刻なセキュリティリスクであり、送信者の信頼性は不明です。受信者は即座に行動を起こし、バグを修正し…</p>

表 14：自己整合性 (Self-consistency) を促すプロンプティングの例

上記のプロンプトを使って、一貫性のある分類結果が返ってくるかどうか試してみることができます。使用するモデルや Temperature 設定に応じて、「IMPORTANT」が返ってくることもあるれば、「NOT IMPORTANT」が返ってくることもあるでしょう。

多くの Chain of Thought を生成し、最も頻繁に出現する回答（この例では「IMPORTANT」）を選択することで、LLM からより一貫して正しい回答を得ることができます。

この例は、自己整合性プロンプティングが、複数の視点を考慮し、最も一貫性のある回答を選択することによって、LLM の応答の正確性をどのように向上させるかを示しています。

Tree of Thoughts (ToT)

Chain of Thought および自己整合性プロンプティングについて理解したところで、次に Tree of Thoughts (ToT)¹²を見ていきましょう。これは CoT プロンプティングの概念を一般化したものであり、単一の直線的な思考の連鎖をたどるだけでなく、LLM が複数の異なる推論パスを同時に探索することを可能にします。この様子は図 1 に示されています。

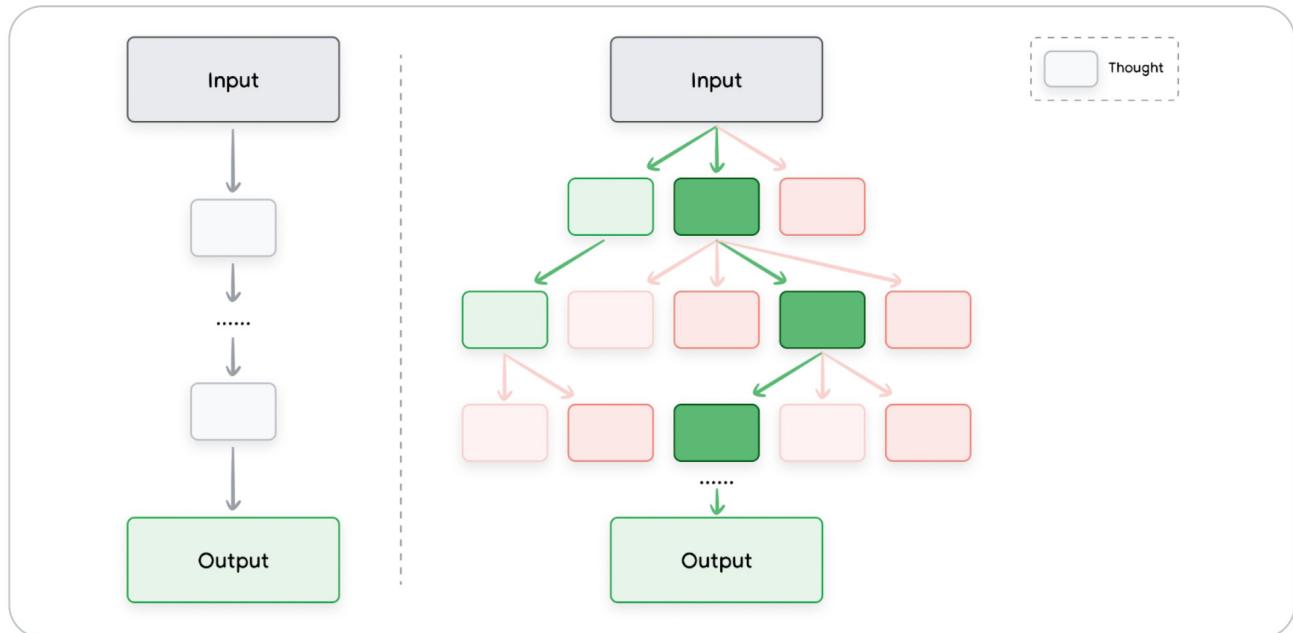


図 1：左側の Chain of Thought プロンプティングと、右側の Tree of Thoughts プロンプティングの対比

このアプローチにより、ToT は特に探索を必要とする複雑なタスクに適しています。これは、思考の木 (tree of thoughts) を維持することによって機能します。そこでは、各思考 (thought) が、問題解決に向けた中間ステップとして機能する、一貫性のある言語シーケンス (一連の言葉) を表します。これにより、モデルは木 (ツリー) の異なるノードから分岐することで、様々な推論パスを探索することができます。

Tree of Thoughts (ToT) についてもう少し詳しく解説している素晴らしいノートブックがあります。これは『Large Language Model Guided Tree-of-Thought』（大規模言語モデル誘導による Tree of Thoughts）という論文に基づいています⁹。

ReAct (推論と行動)

ReAct (Reason and Act、推論と行動) プロンプティングは、LLM が複雑なタスクを解決できるようにするためのパラダイムです。これは、自然言語による推論と外部ツール（検索、コードインタプリタなど）を組み合わせることで、LLM が特定の行動（例えば、外部 API とやり取りして情報を取得するなど）を実行できるようにするものであり、エージェントモデリングへの第一歩となります。

ReAct は、人間が実世界で活動する方法を模倣しています。なぜなら、人間は言葉で推論し、情報を得るために行動を起こすことができるからです。ReAct は、様々なドメイン（領域）において、他のプロンプトエンジニアリング手法と比較して優れた性能を発揮します。

ReAct プロンプティングは、推論と行動を「思考・行動ループ」へと組み合わせることで機能します。まず LLM は問題について推論し、行動計画を生成します。次に、その計画に含まれる行動を実行し、結果を観察します。そして LLM は、その観察結果を用いて推論を更新し、新たな行動計画を生成します。このプロセスは、LLM が問題の解決策に到達するまで続きます。

これを実際に動作させるとこを見るには、コードを記述する必要があります。コードスニペット 1 では、Python 用の Langchain フレームワークを、Vertex AI (google-cloud-aiplatform) および google-search-results の pip パッケージと共に使用しています。

Python

```
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain.llms import VertexAI

prompt = "How many kids do the band members of Metallica have?"

llm = VertexAI(temperature=0.1)
tools = load_tools(["serpapi"], llm=llm)

agent = initialize_agent(tools, llm,
                        agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)
agent.run(prompt)
```

コードスニペット 1 : LangChain と Vertex AI を用いた ReAct エージェントの作成

コードスニペット 2 にその結果を示します。ReAct が 5 回の検索を連鎖させている点に注目してください。実際、LLM はバンド名を見つけ出すために Google 検索結果をスクレイピングしています。次に、その結果を観察結果としてリストアップし、次の検索のための思考を連鎖させます。

```
> Entering new AgentExecutor chain...
Metallica has 4 members.
Action: Search
Action Input: How many kids does James Hetfield have?
Observation: three children
Thought: 1/4 Metallica band members have 3 children
Action: Search
Action Input: How many kids does Lars Ulrich have?
Observation: 3
Thought: 2/4 Metallica band members have 6 children
Action: Search
Action Input: How many kids does Kirk Hammett have?
Observation: Hammett has been married to his second wife Lani since 1998.
They have two sons, Angel (b. September 29, 2006) and Vincenzo (b. June 28, 2008).
Thought: 3/4 Metallica band members have 8 children
Action: Search
Action Input: How many kids does Robert Trujillo have?
Observation: 2
Thought: 4/4 Metallica band members have 10 children
Final Answer: 10
```

コードスニペット 2：ターミナルでの ReAct エージェントの実行（結果）

コードスニペット 2 では、（エージェントが） バンド「メタリカ」には 4 人のバンドメンバーがいることを見つけ出します。次に、各バンドメンバーについて検索して子供の数を調べ、その合計を計算します。最後に、子供の総数を最終的な答えとして返します。

ReAct プロンプティングを実践するには、以前のプロンプトと応答を継続的に再送信し（場合によっては生成された余分なコンテンツの削除（トリミング）も行い）、かつモデルに適切な例や指示を与えて初期設定を行う必要がある、という点を理解しておくことが重要です。GoogleCloudPlatform Github リポジトリでホストされているノートブック¹⁴ を参照してください。そこでは、より詳細な例と共に、実際の LLM の入出力がもう少し詳しく示されています。

自動プロンプトエンジニアリング

ここまで説明で、プロンプトを作成することがいかに複雑になりうるか、お気づきかもしれません。これを自動化できたら素晴らしいと思いませんか（プロンプトを作成するためのプロンプトを作成する、ということです）？実は、それを実現する手法があります。それが自動プロンプトエンジニアリング (Automatic Prompt Engineering/APE) です。この手法¹⁵は、人間の入力の必要性を軽減するだけでなく、様々なタスクにおけるモデルのパフォーマンスを高めることもできます。

モデルにプロンプトを与え、さらに多くのプロンプトを生成させます。生成されたプロンプトを評価し、場合によっては良いものを修正します。そして、このプロセスを繰り返します。

例えば、グッズ T シャツのウェブショップ用チャットボットをトレーニングする際に、顧客がバンド T シャツを注文する時のあらゆる様々な言い方を把握したい、といった場合に自動プロンプトエンジニアリングを活用できます。

1. 出力バリエント（多様な言い方）を生成させるためのプロンプトを作成します。この例では、gemini-pro を使用して 10 個の表現例を生成します。表 15 を参照してください：

プロンプト	バンドのグッズ T シャツのウェブショップがあり、チャットボットをトレーニングするために、『メタリカの T シャツ 1 枚、サイズ S』という注文に対する様々な言い方が必要です。この注文について、意味は同じままで、異なる言い換え表現を 10 個生成してください。
出力結果	<ol style="list-style-type: none"> 1. メタリカの T シャツ、S サイズを 1 枚購入したいです。 2. S サイズのメタリカ T シャツを注文できますか？ 3. メタリカ T シャツの S サイズを 1 枚買いたいです。 4. メタリカのシャツ 1 枚、サイズ S お願いします。 5. メタリカ T シャツの S サイズが 1 枚欲しいです。 6. メタリカ T シャツの S サイズを 1 枚注文したいのですが。 7. S サイズのメタリカ T シャツを 1 枚ください。 8. メタリカ T シャツの S サイズを 1 枚探しています。 9. メタリカ T シャツの S サイズを 1 枚お願いできますか？ 10. メタリカの S サイズ T シャツを 1 枚お願いします。

表 15：LLM を用いたプロンプト生成

2. 選択された評価指標に基づいて候補をスコアリングし、すべての表現例候補を評価します。例えば、BLEU (Bilingual Evaluation Understudy、バイリンガル評価補助尺度) や ROUGE (Recall-Oriented Understudy for Gisting Evaluation、要約評価のための再現率指向補助尺度) などを使用できます。
3. 最高の評価スコアを持つ表現例候補を選択します。この候補が、ソフトウェアアプリケーションやチャットボットで使用できる最終的なプロンプトとなります。選択されたプロンプトをさらに微調整し、再度評価することも可能です。

コードプロンプティング

Gemini は主としてテキストベースのプロンプトに焦点を当てていますが、これにはコードを返すためのプロンプト作成も含まれます。Vertex AI Studio を使ってこれらのプロンプトをテストし、いくつかのコーディング例を見てみましょう。

コード生成のためのプロンプト

Gemini は開発者の役割も担い、選択した任意のプログラミング言語でのコード作成を支援することができます。開発者にとっては、これによりコード作成のプロセスを高速化できます。

マシン上のあるフォルダに、名前を変更する必要があるファイルが数百個ある状況を想像してみてください。各ファイルの名前を手動で変更するには、多くの時間がかかるでしょう。Bash を少し知つていれば、これを自動化するスクリプトを書くこともできますが、それにもやはり時間がかかるかもしれません。そこで、プロンプトを作成してみましょう。プロンプトは、一般消費者向けチャットボットの Gemini で作成することもできますし、機密性についてより懸念がある場合は、自身の Google Cloud アカウント内で Vertex AI Studio を開いて作成することもできます。Vertex AI Studio の利点は、Temperature などの設定を行えることです。

名前	1_bash_rename_files_in_folder		
ゴール	フォルダ内のファイル名を変更（リネーム）するための Bash コードを作成するプロンプトを作成する		
Model	gemini-pro		
Temperature	0.1	Token Limit	1024
Top-K	N/A	Top-P	1
プロンプト	フォルダ名の入力を求める Bash のコードスニペットを作成してください。そのスクリプトは、指定されたフォルダ内のすべてのファイルに対し、ファイル名の先頭に draft という接頭辞を付加して名前を変更するものとします。		
出力結果	<pre>```bash #!/bin/bash # Ask for the folder name echo "Enter the folder name: " read folder_name # Check if the folder exists if [! -d "\$folder_name"]; then echo "Folder does not exist." exit 1 fi # Get the list of files in the folder files=("\$folder_name"/*) # Rename each file by prepending "draft" to the file name for file in "\${files[@]}"; do new_file_name="draft_\$(basename "\$file")" mv "\$file" "\$new_file_name" done echo "Files renamed successfully." ``` </pre>		

表 16 : Gemini を用いたコード作成

これは良いコードのように見えますね。コメントまでされています！しかしながら、LLM は推論ができず、訓練データを（単に）繰り返している可能性もあるため、生成されたコードはまず人間が確認し、テストすることが不可欠です。

さて肝心なのは、そのコードは本当に動作するのか、という点です。+

まず、いくつかのファイルだけを含むテストフォルダで試してみましょう。これらのファイルは **filename.txt** から **draft_filename.txt** へと名前が変更される必要があります。

1. 表 16 の出力（``bash と `` のテキストラッパーを除く）をコピーし、「rename_files.sh」という名前の新しいファイルに貼り付けます。
2. ターミナルウィンドウを開き、「. rename_files.sh」と入力します。フォルダ名（例：test）を入力するよう求められるので、入力して Enter キーを押します。
3. スクリプトは問題なく実行されるようです。「Files renamed successfully.」（ファイル名は正常にリネームされました。）というメッセージが表示されます。

テストフォルダの中を見ると、すべてのファイルが完全に **draft_filename.txt** という形式にリネームされていることがわかります。成功です！

コード解説のためのプロンプト

開発者としてチームで作業する場合、他の誰かが書いたコードを読まなければならぬことがあります。Gemini はこの点でも役立ちます。表 16 のコード出力からコメントを削除し、大規模言語モデル（LLM）に何が行われているかを解説するように依頼してみましょう。表 17 を参照してください：

名前	1_bash_explain_the_code		
ゴール	Bash コードを解説するためのプロンプトを作成する		
Model	gemini-pro		
Temperature	0.1	Token Limit	1024
Top-K	N/A	Top-P	1
プロンプト	<p>以下の Bash コードを解説してください：</p> <pre>``` #!/bin/bash echo "Enter the folder name: " read folder_name if [! -d "\$folder_name"]; then echo "Folder does not exist." exit 1 fi files=("\$folder_name"/*) for file in "\${files[@]}"; do new_file_name="draft_\$(basename "\$file")" mv "\$file" "\$new_file_name" done echo "Files renamed successfully." ``` </pre>		
出力結果	<p>この Bash スクリプトは以下の処理を実行します：</p> <p>1. ユーザー入力： echo コマンドと read コマンドを使用して、ユーザーにフォルダ名の入力を促します。入力されたフォルダ名は変数 folder_name に格納されます。</p> <p>2. フォルダ存在チェック： 条件式 <code>[! -d "\$folder_name"]</code> を使用して、指定された名前のフォルダが存在するかどうかをチェックします。フォルダが存在しない場合は、「Folder does not exist.」(フォルダが存在しません。) というエラーメッセージを表示し、終了コード 1 でスクリプトを終了します。</p> <p style="text-align: right;">次のページに続きます</p>		

出力結果	<p>3. ファイル一覧取得： コマンド <code>files=("\$folder_name" /*)</code> を使用して、指定されたフォルダ内のすべてのファイルをリストアップします。これにより、フォルダ内の全ファイルへのパスを含む <code>files</code> という名前の配列が作成されます。</p> <p>4. ファイル名変更： <code>for</code> ループを使用して、<code>files</code> 配列内の各ファイルを反復処理します。 各ファイルについて、コマンド <code>new_file_name=" draft_\$(basename "\$file")"</code> を使用して新しいファイル名を生成します。この新しいファイル名は、元のファイル名の先頭に文字列 <code>draft_</code> を接頭辞として付加することで作成されます。 次に、コマンド <code>mv "\$file" "\$new_file_name"</code> を使用してファイル名を変更します。これはファイルを新しいファイル名に移動させるものです。</p> <p>5. 成功メッセージ： すべてのファイルの名前変更が完了した後、「Files renamed successfully.」（ファイル名は正常にリネームされました。）という成功メッセージを表示し、名前変更操作が成功したことを示します。</p>
------	--

表 17：コード解説用プロンプト

コード翻訳のためのプロンプト

表 16 の Bash コードは問題なく動作するようです。しかし、このスクリプトは、ファイル名の入力を求めるようになれば、より再利用しやすくなるでしょう。理想的には、UI を備えた独立したアプリケーションとして機能すべきです。出発点として、(Web) アプリケーションには Bash よりも Python の方が適した言語でしょう。LLM は、ある言語で書かれたコードを別の言語に翻訳する際に役立ちます。表 18 の例を見てみましょう：

名前	1_python_rename_files		
ゴール	Bash コードを Python に翻訳するためのプロンプトを作成する		
Model	gemini-pro		
Temperature	0.1	Token Limit	1024
Top-K	N/A	Top-P	1
プロンプト	<p>以下の Bash コードを Python のコードスニペットに翻訳してください：</p> <pre>```bash #!/bin/bash echo "Enter the folder name: " read folder_name if [! -d "\$folder_name"]; then echo "Folder does not exist." exit 1 fi files=("\$folder_name"/*) for file in "\${files[@]}"; do new_file_name="draft_\$(basename "\$file")" mv "\$file" "\$new_file_name" done echo "Files renamed successfully." ``` </pre>		
出力結果	<pre>```python import os import shutil # Get the folder name from the user folder_name = input("Enter the folder name: ") # Check if the folder exists if not os.path.isdir(folder_name): print("Folder does not exist.") exit(1) ``` </pre>		
	次のページに続きます		

出力結果	<pre># Get a list of all files in the folder files = os.listdir(folder_name) # Iterate over the files for file in files: # Create a new file name with the prefix "draft_" new_file_name = f"draft_{file}" # Move the file to the new name shutil.move(os.path.join(folder_name, file), os.path.join(folder_name, new_file_name)) # Print a success message print("Files renamed successfully.") ``</pre>
------	--

表 18：Bash から Python へのコード翻訳用プロンプト

生成されたコードを読み、レビューしてください。プロンプトからの出力をコピーし、「file_renamer.py」という名前の新しいファイルに貼り付けてください。ターミナルウィンドウを開き、以下のコマンドを実行してコードをテストしてください：**python file_renamer.py**

注記：Vertex AI の Language Studio で (Python) コードをプロンプトで生成させる際は、「Markdown」ボタンをクリックする必要があります。そうしないと、プレーンテキストで応答が返され、Python コードの実行に不可欠な適切な行のインデントが欠落してしまいます。

コードのデバッグとレビューのためのプロンプト

(表 18 のプロンプトで生成された) Python コードに手動でいくつか編集を加えてみましょう。その編集内容は、ユーザーにファイル名の接頭辞の入力を求め、その接頭辞を大文字でファイル名に付加するようにするものです。コードスニペット 3 にその編集例を示しますが、残念なことに、このコードは Python エラーを返すようになってしまいました！

Python

```
import os
import shutil

folder_name = input("Enter the folder name: ")
prefix = input("Enter the string to prepend to the filename: ")
text = toUpperCase(prefix)

if not os.path.isdir(folder_name):
    print("Folder does not exist.")
    exit(1)

files = os.listdir(folder_name)

for file in files:
    new_filename = f"{text}_{file}"

    shutil.move(os.path.join(folder_name, file),
os.path.join(folder_name, new_file_name))

print("Files renamed successfully.")
```

コードスニペット 3：エラーのある Python スクリプト

やはり、これはバグのようですね：

The below Python code gives an error:

```
Traceback (most recent call last):
  File "/Users/leeboonstra/Documents/test_folder/ rename_files.py", line 7,
in <module>
    text = toUpperCase(prefix)
NameError: name 'toUpperCase' is not defined
```

コードスニペット 4：Python コードを壊してしまいました

大規模言語モデル（LLM）にコードのデバッグとレビューを依頼できるか試してみましょう。表 19 を見てください：

名前	1_python_debug_code		
ゴール	Python コードをデバッグおよびレビューするためのプロンプトを作成		
Model	gemini-pro		
Temperature	0.1	Token Limit	1024
Top-K	N/A	Top-P	1
プロンプト	<p>以下の Python コードはエラーを発生させます： どこが間違っているか（問題点）をデバッグし、 どのようにコードを改善できるか解説してください。</p> <pre>File "/ Users/leeboonstra/Documents/test_folder/ rename_files.py", line 7, in <module> text = toUpperCase(prefix) NameError: name 'toUpperCase' is not defined Debug what's wrong and explain how I can improve the code. ```python import os import shutil # Get the folder name from the user folder_name = input("Enter the folder name: ") prefix = input("Enter the string to prepend to the filename: ") text = toUpperCase(prefix) # Check if the folder exists if not os.path.isdir(folder_name): print("Folder does not exist.") exit(1) # Get a list of all files in the folder files = os.listdir(folder_name) # Iterate over the files for file in files:</pre> <p style="text-align: right;">次のページに続きます</p>		

プロンプト	<pre># Create a new file name with the prefix "draft_" new_filename = f"{text}_{file}" # Move the file to the new name shutil.move(os.path.join(folder_name, file), os.path.join(folder_name, new_file_name)) # Print a success message print("Files renamed successfully.") ``` </pre>
出力結果	<p>このコードは、変数 <code>prefix</code> を大文字に変換するために <code>toUpperCase</code> 関数を呼び出していますが、その関数は定義されていません。この問題を修正するには、与えられた文字列を大文字に変換する文字列クラスの <code>upper()</code> メソッドを使用できます。修正されたコードを以下に示します：</p> <pre>```python import os import shutil # Get the folder name from the user folder_name = input("Enter the folder name: ") prefix = input("Enter the string to prepend to the filename: ") text = prefix.upper() # Check if the folder exists if not os.path.isdir(folder_name): print("Folder does not exist.") exit(1) # Get a list of all files in the folder files = os.listdir(folder_name) # Iterate over the files for file in files: # Create a new file name with the prefix "draft_" new_filename = f"{text}_{file}" # Move the file to the new name shutil.move(os.path.join(folder_name, file), os.path.join(folder_name, new_filename)) ``` </pre> <p style="text-align: right;">次のページに続きます</p>

出力結果

```
os.path.join(folder_name, new_filename))

# Print a success message
print("Files renamed successfully.")
```

```

さらに、コードに加えることができるいくつかの改善点があります：

新しいファイル名で拡張子が保持されていません。元のファイル名から拡張子を抽出し、新しいファイル名に追加する方が良いでしょう。

このコードはフォルダ名にスペースが含まれていないことを想定しています。しかし、スペースをアンダースコア（\_）やハイフン（-）に置き換えるなどして適切に処理するのが良い習慣です。

文字列補間には、従来の + 演算子よりも、f 文字列構文 (f"") を使用する方が、読みやすく簡潔です。

このコードは名前変更プロセス中に発生する可能性のあるエラーを処理していません。発生しうるエラーを捕捉するために、shutil.move の呼び出しを try...except ブロックで囲む方が良いでしょう。これらの提案を取り入れた、改善されたコードを以下に示します：

```
```python
import os
import shutil
# Get the folder name from the user
folder_name = input("Enter the folder name: ")
prefix = input("Enter the string to prepend to the filename: ")
# Check if the folder exists
if not os.path.isdir(folder_name):
    print("Folder does not exist.")
    exit(1)
# Get a list of all files in the folder
files = os.listdir(folder_name)

# Iterate over the files
for file in files:
```

次のページに続きます

出力結果	<pre> # Get the file extension ext = os.path.splitext(file)[1] # Create a new file name with the prefix and the file extension new_filename = f"{prefix}_{os.path.splitext(file)[0]}.{ext}" # Build the full paths for the original and new files old_path = os.path.join(folder_name, file) new_path = os.path.join(folder_name, new_filename) # Try to move the file try: shutil.move(old_path, new_path) except Exception as e: print(f"An error occurred while moving the file {file}: {e}") # Print a success message print("Files renamed successfully.") ``` これらの改善により、コードはよりロバスト（堅牢）で柔軟になるとともに、より読みやすく、理解しやすくなります… （応答がトークン上限に達したため、途中で打ち切られました。より長い応答が必要な場合は、トークン上限を増やしてみてください。） </pre>
------	--

表 19：Python コードのデバッグとレビュー用プロンプト

これは素晴らしい結果ですね。LLM は、問題の解決方法を教えるだけでなく、コードにさらにバグがあること、そしてそれらの解決方法も見つけ出してくれました。応答の最後の部分では、コードを全般的に改善するための提案も行われました。

マルチモーダルプロンプティングについて

コードのためのプロンプティング（コードプロンプティング）は、依然として通常の（テキストベースの）大規模言語モデルを使用します。マルチモーダルプロンプティングは別の話題です。これは、単にテキストに頼るのではなく、複数の入力形式を使用して大規模言語モデルをガイドする手法を指します。これには、モデルの能力や当面のタスクに応じて、テキスト、画像、音声、コード、あるいはその他のフォーマットの組み合わせが含まれる可能性があります。

ベストプラクティス

適切なプロンプトを見つけるには試行錯誤が必要です。Vertex AI の Language Studio は、様々なモデルに対してテストできる機能を備えており、プロンプトで色々と試すのに最適な場所です。以下のベストプラクティスを活用して、プロンプトエンジニアリングのプロになります。

例の提示

最も重要なベストプラクティスは、プロンプト内に（ワンショット / フューショットの）例を提供することです。これは非常に効果的です。なぜなら、それが強力な指導ツールとして機能するからです。これらの例は、望ましい出力や類似の応答を示し、モデルがそれらから学習して自身の生成をそれに応じて調整することを可能にします。これは、モデルにを目指すべき参照点やターゲットを与えるようなものであり、応答の正確性、スタイル、トーンを改善し、期待により良く合致させる効果があります。

シンプルな設計

プロンプトは、人間にとってもモデルにとっても、簡潔、明確、そして理解しやすいものであるべきです。目安として、もし人間にとっても既に紛らわしいものであれば、それはモデルにとっても同様に紛らわしい可能性が高いでしょう。複雑な言葉遣いは避け、不要な情報を提供しないようにしましょう。

例：

修正前：

私は今ニューヨークを訪れています。素晴らしい場所についてもっと知りたいです。3歳の子供が2人います。休暇中、私たちはどこへ行くべきでしょうか？

修正後：

役割：観光客向けの旅行ガイド 3歳の子供と一緒に訪れるのに最適な、ニューヨーク・マンハッタンの素晴らしい場所を説明してください。

行動を明確に示す動詞を使用するようにしてください。以下にその例をいくつか挙げます：

振る舞う (Act)、分析する (Analyze)、分類する (Categorize/Classify)、対比する (Contrast)、比較する (Compare)、作成する (Create/Write)、説明する (Describe)、定義する (Define)、評価する (Evaluate)、抽出する (Extract)、見つける (Find)、生成する (Generate)、特定する (Identify)、リストアップする (List)、測定する (Measure)、整理する (Organize)、パースする (Parse)、選ぶ (Pick)、予測する (Predict)、提供する (Provide)、ランク付けする (Rank)、推奨する (Recommend)、返す (Return)、取得する (Retrieve)、書き直す (Rewrite)、選択する (Select)、示す (Show)、ソートする (Sort)、要約する (Summarize)、翻訳する (Translate)

具体的な出力指示

望ましい出力について具体的に指示してください。簡潔すぎる指示では、LLM を十分にガイドできなかつたり、あまりに一般的すぎたりする可能性があります。プロンプトで（システムプロンプティングやコンテキストプロンプティングを通じて）具体的な詳細を提供することは、モデルが関連性の高い点に集中するのを助け、全体的な正確性を向上させることができます。

良い例：

トップ 5 のビデオゲーム機に関する 3 段落のブログ記事を生成してください。ブログ記事は、情報豊富で引き込まれるような内容で、かつ会話的な（話し言葉のような）スタイルで書かれている必要があります。

悪い例：

ビデオゲーム機についてのブログ記事を生成してください。

制約より指示を優先

プロンプティングにおいては、LLM の出力をガイドするために「指示」と「制約」が用いられます。

- **指示**：応答の望ましいフォーマット、スタイル、あるいは内容に関する明示的な指示を与え、モデルが何をすべきか、何を生成すべきかをガイドします。
- **制約**：応答に対する一連の制限や境界を設定するものであり、モデルが何をすべきでないか、何を避けるべきかを制限します。

研究では、プロンプティングにおいて制約に過度に依存するよりも、肯定的な指示に焦点を当てる方がより効果的である可能性を示唆しています。このアプローチは、人間が「してはいけないこと」のリストよりも肯定的な指示を好む傾向と一致しています。

指示は望ましい結果を直接伝えますが、制約は何が許容されるかについてモデルに推測させる余地を残す可能性があります。制約を用いることは、定義された境界内で柔軟性を与え、創造性を促しますが、一方でモデルの潜在能力を制限する可能性があります。また、制約のリストは互いに矛盾することもあります

ただし、制約が依然として有用な特定の状況もあります。例えば、モデルが有害な、あるいはバイアスのあるコンテンツを生成するのを防ぐ場合や、厳格な出力フォーマットまたはスタイルが必要とされる場合などです。

可能であれば、肯定的な指示を使用してください。つまり、モデルに何をすべきでないかを伝える代わりに、何をすべきかを伝えるのです。これにより、混乱を避け、出力の正確性を向上させることができます。

例：

良い例：

トップ 5 のビデオゲーム機に関する 1 段落のブログ記事を生成してください。取り上げるのは、ゲーム機本体、製造会社、発売年、総販売台数のみとしてください。

悪い例：

トップ 5 のビデオゲーム機に関する 1 段落のブログ記事を生成してください。ビデオゲームのタイトルはリストアップしないでください。

ベストプラクティスとしては、まず指示を優先することから始めてください。モデルに何をしてほしいかを明確に述べ、制約は安全性、明確性、あるいは特定の要件のために必要な場合にのみ使用します。特定のタスクに対して何が最も効果的かを見つけるために、指示と制約の様々な組み合わせをテストする実験とイテレーションを行い、それらを文書化してください。

最大トークン長の制御

生成される LLM の応答の長さを制御するには、設定で最大トークン上限を指定するか、あるいはプロンプト内で特定の長さを明示的に要求するという、いずれかの方法があります。例えば、次のようなプロンプトです：「量子物理学をツイート程度の長さのメッセージで解説してください。」

プロンプトでの変数利用

プロンプトを再利用し、より動的にするために、プロンプト内で変数を使用します。変数は、異なる入力に応じて変更することができます。例えば、表 20 に示すような、ある都市に関する事実を提示するプロンプトを考えてみましょう。プロンプト内に都市名をハードコーディングする代わりに、変数を使用します。変数を使用すると、繰り返しを避けることができるため、時間と労力を節約できます。同じ情報要素を複数のプロンプトで使用する必要がある場合、それを変数に格納し、各プロンプトでその変数を参照することができます。これは、プロンプトをアプリケーションに組み込む際に、特に理にかなっています。

プロンプト	<p>変数： {city} = “アムステルダム”</p> <p>プロンプト本文： あなたは旅行ガイドです。都市に関する事実を 1 つ教えてください。都市：{city}</p>
出力結果	Amsterdam is a beautiful city full of canals, bridges, and narrow streets. It's a great place to visit for its rich history, culture, and nightlife.

表 20：プロンプトでの変数利用

入力形式と記述スタイルの実験

異なるモデル、モデル設定、プロンプト形式、言葉遣い、そして（同じプロンプトの）送信（実行）タイミングでさえ、異なる結果を生み出す可能性があります。したがって、スタイル、言葉遣い、そしてプロンプトの種類（ゼロショット、フューショット、システムプロンプトなど）といったプロンプトの属性を変えて実験することが重要です。

例えば、革新的なビデオゲーム機であったセガ ドリームキャストに関するテキストを生成するという目的を持つプロンプトは、質問、平叙文（ステートメント）、あるいは指示といった形式で記述することができ、それぞれ異なる出力をもたらします：

- ・**質問形式**：セガ ドリームキャストとは何でしたか？なぜそれはあれほど革新的なゲーム機だったのですか？
- ・**平叙文形式**：セガ ドリームキャストは、1999 年にセガによって発売された第 6 世代のビデオゲーム機でした。それは…
- ・**指示形式**：セガ ドリームキャストというゲーム機について説明し、なぜそれが非常に革新的であったかを解説する 1 段落の文章を作成してください。

分類タスクにおけるフューショット：クラスの混合

一般的に言って、フューショットの例の順序は、それほど重要ではないはずです。しかしながら、分類タスクを行う場合は、フューショットの例の中で、可能性のある応答クラスを混ぜ合わせるようにしてください。そうしないと、例の特定の順序に対して過学習してしまう可能性があるからです。可能性のある応答クラスを混ぜ合わせることで、モデルが単に例の順序を記憶するのではなく、各クラスの主要な特徴を識別することを学習するように促すことができます。これにより、未知のデータに対する、より堅牢で汎化性能の高いパフォーマンスにつながります。

良い目安としては、まず 6 つのフューショットの例から始め、そこから精度のテストを開始することです。

モデル更新への適応

モデルアーキテクチャの変更、追加されたデータ、そしてモデルの能力について、常に最新情報を把握しておくことが重要です。より新しいモデルバージョンを試し、新しいモデル機能をより良く活用できるようにプロンプトを調整しましょう。Vertex AI Studio のようなツールは、プロンプトの様々なバージョンを保存、テスト、そして文書化するのに非常に役立ちます。

出力形式の実験

プロンプトの入力形式だけでなく、出力形式についても実験することを検討してください。データの抽出、選択、パス（解析）、順序付け、ランク付け、あるいは分類といった非創造的なタスクについては、出力を JSON や XML のような構造化形式で返すように試してみてください。

要約すると、出力に JSON を使用する利点は以下の通りです：

- ・常に同じ構造で返される
- ・受け取りたいデータに焦点を当てられる
- ・ハルシネーションが発生する可能性が低い
- ・データの関係性を表現できる
- ・データ型で表現される
- ・構造化されているためソート可能である

フューショットプロンプティングのセクションにある表 4 は、構造化された出力をどのように返すかの例を示しています。

JSON 修復

データを JSON 形式で返すことには数多くの利点がありますが、欠点がないわけではありません。JSON の構造化された性質は、アプリケーションでの解析や利用には有益である一方、プレーンテキストよりもはるかに多くのトークンを必要とし、処理時間の増加とコストの上昇につながります。さらに、JSON の冗長性はトークン上限を容易に使い切ってしまい、トークン上限によって生成が突然打ち切られた場合に特に問題となります。この打ち切りによって、重要な閉じ波括弧 `)` や閉じ角括弧 `]` が欠落した無効な JSON がしばしば生成され、出力が利用不可能になってしまいます。幸いなことに、このような状況では、(PyPI で利用可能な) `json-repair` ライブラリのようなツールが非常に役立ちます。このライブラリは、不完全または不正な形式の JSON オブジェクトをインテリジェントに自動修正しようと試みるため、LLM が生成した JSON を扱う際、特に潜在的な打ち切り問題に対処する上で、不可欠な助けとなります。

スキーマの利用

構造化 JSON を出力として使用することは、本書で何度も見てきたように、優れたソリューションです。しかし、入力についてはどうでしょうか？LLM が生成する出力を構造化するのに JSON は優れていますが、モデルに提供する入力を構造化するためにも、JSON は非常に有用となり得ます。ここでも JSON スキーマが役割を果たします。JSON スキーマは、JSON 入力に期待される構造とデータ型を定義します。スキーマを提供することで、LLM が受け取るであろうデータに関する明確な設計図を与えることになり、LLM が関連情報に注意を集中させ、入力を誤解するリスクを減らすのに役立ちます。さらに、スキーマは、異なるデータ要素間の関係性を確立するのに役立ち、特定のフォーマットを持つ日付やタイムスタンプのフィールドを含めることで、LLM に「時間を認識させる」ことさえ可能です。

簡単な例を以下に示します：

E コマースカタログの商品に関する説明文を LLM を使用して生成するケースを考えてみましょう。商品の説明を単なる自由形式のテキストで提供する代わりに、JSON スキーマを使用して商品の属性を定義することができます：

```
{  
  "type": "object",  
  "properties": {  
    "name": { "type": "string", "description": "Product name" },  
    "category": { "type": "string", "description": "Product category" },  
    "price": { "type": "number", "format": "float", "description": "Product price" },  
    "features": {  
      "type": "array",  
      "items": { "type": "string" },  
      "description": "Key features of the product"  
    },  
    "release_date": { "type": "string", "format": "date", "description": "Date the product was released" }  
  },  
}
```

コードスニペット 5：構造化出力スキーマの定義

次に、このスキーマに準拠する JSON オブジェクトとして、実際の商品データを提供します：

```
{  
  "name": "Wireless Headphones",  
  "category": "Electronics",  
  "price": 99.99,  
  "features": ["Noise cancellation", "Bluetooth 5.0", "20-hour battery life"],  
  "release_date": "2023-10-27"  
}
```

コードスニペット 6：LLM からの構造化出力

データを前処理し、完全な文書全体を提供する代わりにスキーマとデータの両方のみを提供することで、LLM は商品の属性を明確に理解でき、正確で関連性の高い説明文を生成する可能性がはるかに高くなります。LLM の注意を関連フィールドに誘導するこの構造化入力アプローチは、大量のデータを扱う場合や、LLM を複雑なアプリケーションに組み込む場合に特に価値があります。

他のプロンプトエンジニアとの共同実験

良いプロンプトを考案しようとしなければならない状況では、複数の人々に試作を依頼することを検討すると良いかもしれません。全員が（本章で挙げられているような）ベストプラクティスに従ったとしても、それぞれの異なるプロンプト試行の間でパフォーマンスにばらつきが見られるでしょう。

CoT のベストプラクティス

CoT プロンプティングにおいては、推論の後に回答を置くことが必須です。なぜなら、推論を生成するプロセスが、モデルが最終的な回答を予測する際に参照するトークンを変化させるためです。

CoT や自己整合性を用いる場合、モデルの応答から最終的な回答を、推論部分とは分離して抽出できる必要があります。

CoT プロンプティングでは、Temperature を 0 に設定してください。

Chain of Thought プロンプティングはグリーディーデコーディングに基づいており、言語モデルによって割り当てられた最高の確率に基づいてシーケンス中の次の単語を予測します。一般的に言って、推論を用いて最終的な回答を導き出す場合、正しい回答はおそらく一つだけでしょう。したがって、Temperature は常に 0 に設定すべきです。

プロンプト試行の文書化

最後のヒントは本章で以前にも言及しましたが、その重要性はいくら強調してもしすぎることはありません。それは、プロンプト試行を詳細にわたって文書化することです。それにより、何がうまくいき、何がうまくいかなかったかを時間をかけて学ぶことができます。

プロンプトの出力は、モデル間、サンプリング設定間、そして同一モデルの異なるバージョン間でさえも異なる可能性があります。さらに、同一モデルに対して同一のプロンプトを与えた場合でも、出力される文の書式や言葉遣いにわずかな違いが生じることがあります。（例えば、以前述べたように、もし 2 つのトークンが同じ予測確率を持つ場合、同率はランダムに解消されることがあります。これが、後続の予測トークンに影響を与える可能性があるのです。）

表 21 をテンプレートとして Google スプレッドシートを作成することを推奨します。このアプローチの利点は、将来的にプロンプティング作業を見返す必要が必然的に生じた際に、完全な記録が手元にあることです。その状況としては、将来の作業再開（少し中断しただけでも、どれだけ忘れてしまうかに驚くことでしょう）、異なるモデルバージョンでのプロンプトのパフォーマンステスト、そして将来発生するエラーのデバッグ支援などが考えられます。

この表にある項目に加えて、プロンプトのバージョン、結果が OK / NG / 時々 OK のいずれであったかを記録する項目、そしてフィードバックを記録する項目も追跡すると役立ちます。幸運にも Vertex AI Studio を使用している場合は、プロンプトを（文書化した際と同じ名前とバージョンで）保存し、保存されたプロンプトへのハイパーリンクを表（文書）内で追跡記録しましょう。こうすることで、いつでもワンクリックでプロンプトを再実行できるようになります。

検索拡張生成（RAG）システムに取り組む場合は、プロンプトにどのコンテンツが挿入されたかに影響を与える RAG システム固有の側面（要素）も記録すべきです。これには、クエリ、チャンク設定、チャンク出力、その他の情報などが含まれます。

プロンプトがほぼ完璧だと感じたら、それをプロジェクトのコードベースに組み込みます。そして、コードベース内では、保守しやすくなるように、プロンプトはコードとは別のファイルに保存します。最後に、理想的にはプロンプトを運用化されたシステムの一部とし、プロンプトがタスクに対してどれだけうまく汎化するかを把握するためには、自動化されたテストと評価手順を活用すべきでしょう。

プロンプトエンジニアリングは反復的なプロセスです。様々なプロンプトを作成・テストし、結果を分析・文書化し、モデルのパフォーマンスに基づいてプロンプトを改善します。望ましい出力が得られるまで実験を続けます。モデルやモデル設定を変更した場合は、以前に使用したプロンプトに立ち戻り、実験を継続してください。

名前	[name and version of your prompt]		
ゴール	[One sentence explanation of the goal of this attempt]		
Model	[name and version of the used model]		
Temperature	[value between 0 - 1]	Token Limit	[number]
Top-K	[number]	Top-P	[number]
プロンプト	[Write all the full prompt]		
出力結果	[Write out the output or multiple outputs]		

表 21 : A template for documenting prompts

まとめ

本書では、プロンプトエンジニアリングについて解説しました。様々なプロンプティング手法を解説しました。例えば以下のものです：

- ・ゼロショットプロンプティング
- ・フェューショットプロンプティング
- ・システムプロンプティング
- ・ロールプロンプティング
- ・コンテキストプロンプティング
- ・ステップバック・プロンプティング
- ・Chain of Thought (CoT)
- ・自己整合性 (Self-consistency)
- ・Tree of Thoughts (ToT)
- ・ReAct (推論と行動)

さらに、プロンプトを自動化する方法についても検討しました。次に本書では、生成 AI の課題として、プロンプトが不十分な場合に起こりうる問題などについて論じました。最後に、より優れたプロンプトエンジニアになるためのベストプラクティスで締めくくりました。

卷末注

1. Google, 2023, Gemini by Google. Available at: <https://gemini.google.com>.
2. Google, 2024, Gemini for Google Workspace Prompt Guide. Available at:
<https://inthecloud.withgoogle.com/gemini-for-google-workspace-prompt-guide/dl-cd.html>.
3. Google Cloud, 2023, Introduction to Prompting. Available at:
<https://cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/introduction-prompt-design>.
4. Google Cloud, 2023, Text Model Request Body: Top-P & top-K sampling methods. Available at:
https://cloud.google.com/vertex-ai/docs/generative-ai/model-reference/text#request_body.
5. Wei, J., et al., 2023, Zero Shot - Fine Tuned language models are zero shot learners. Available at:
<https://arxiv.org/pdf/2109.01652.pdf>.
6. Google Cloud, 2023, Google Cloud Model Garden. Available at: <https://cloud.google.com/model-garden>.
7. Brown, T., et al., 2023, Few Shot - Language Models are Few Shot learners. Available at:
<https://arxiv.org/pdf/2005.14165.pdf>.
8. Zheng, L., et al., 2023, Take a Step Back: Evoking Reasoning via Abstraction in Large Language Models. Available at: <https://openreview.net/pdf?id=3bq3jsvcQ1>
9. Wei, J., et al., 2023, Chain of Thought Prompting. Available at: <https://arxiv.org/pdf/2201.11903.pdf>.
10. Google Cloud Platform, 2023, Chain of Thought and React. Available at: https://github.com/GoogleCloudPlatform/generative-ai/blob/main/language/prompts/examples/chain_of_thought_react.ipynb.
11. Wang, X., et al., 2023, Self Consistency Improves Chain of Thought reasoning in language models. Available at: <https://arxiv.org/pdf/2203.11171.pdf>.
12. Yao, S., et al., 2023, Tree of Thoughts: Deliberate Problem Solving with Large Language Models. Available at: <https://arxiv.org/pdf/2305.10601.pdf>.
13. Yao, S., et al., 2023, ReAct: Synergizing Reasoning and Acting in Language Models. Available at: <https://arxiv.org/pdf/2210.03629.pdf>.
14. Google Cloud Platform, 2023, Advance Prompting: Chain of Thought and React. Available at: https://github.com/GoogleCloudPlatform/applied-ai-engineering-samples/blob/main/genai-on-vertex-ai/advanced_prompting_training/cot_react.ipynb.
15. Zhou, C., et al., 2023, Automatic Prompt Engineering - Large Language Models are Human-Level Prompt Engineers. Available at: <https://arxiv.org/pdf/2211.01910.pdf>.