提示词工程

作者：*Lee Boonstra*

翻译：GPT4o

目录

> 你不需要是数据科学家或机器学习工程师 —— 每个人都可以写提示词。

引言

在考虑大型语言模型的输入与输出时，文本提示（有时还包含图像等其他模态提示）是模型用来预测特定输出的输入形式。你无需成为数据科学家或机器学习工程师 —— 每个人都可以编写提示词。然而，设计出最有效的提示却可能十分复杂。许多因素都会影响提示的效果：你所使用的模型、模型的训练数据、模型的配置、你的用词选择、表达风格与语气、结构以及上下文都很重要。因此，提示工程是一个迭代优化的过程。不够精确的提示可能会导致回答含糊不清、结果不准确，甚至影响模型生成有价值内容的能力。

当你与 Gemini 聊天机器人交流时，实际上你就在编写提示词。不过，本白皮书的重点是如何在 Vertex AI 中，或通过 API 直接对 Gemini 模型进行提示，因为通过直接调用模型，你可以访问更多配置项（如 temperature 等）。

本白皮书将深入探讨提示工程的相关内容。我们将介绍多种提示技术，帮助你快速入门，并分享实用技巧与最佳实践，助你成为提示设计的高手。我们也将讨论在提示设计过程中可能遇到的一些挑战。

提示工程（Prompt Engineering）

请记住，大型语言模型的本质是一个预测引擎。模型接收一段连续的文本作为输入，然后根据其训练数据预测下一个 token（词或符号）应该是什么。模型会不断执行这一过程，将刚刚预测出的 token 添加到文本末尾，并据此预测下一个 token。每一次的预测都依赖于先前 token 之间的关系，以及模型在训练中所学习到的语言规律。当你编写提示词时，你的目标是让 LLM（大型语言模型）能够预测出符合预期的

token 序列。提示工程的核心，是设计高质量的提示词，从而引导模型生成准确的输出。这一过程通常需要不断试验，找到最优的提示方式，同时优化提示长度，并根据具体任务对写作风格与结构进行调整。

在自然语言处理和大型语言模型的语境下，提示（prompt）指的是提供给模型的输入，用于生成响应或预测结果。

这些提示可以用于实现多种理解与生成任务，例如文本摘要、信息抽取、问答、文本分类、语言或代码翻译、代码生成，以及代码文档生成或逻辑推理等。

你也可以参考 Google 官方的提示设计指南 2, 3，其中包含了一些简单有效的提示示例。

在进行提示工程时，你首先要选择一个合适的模型。提示词往往需要根据具体模型进行优化，无论你使用的是 Vertex AI 中的 Gemini 模型、GPT、Claude，还是开源模型如 Gemma 或 LLaMA。

除了提示词本身，你还需要调试和优化大型语言模型的各种配置参数。

大语言模型的输出配置

当你选定模型后，就需要进一步配置模型参数。大多数大型语言模型（LLM）都提供多种配置选项，用于控制其输出内容。为了实现有效的提示工程，需要根据具体任务对这些参数进行合理设置。

输出长度（Output length）

一个重要的配置项是“要生成的 token 数量”。

生成的 token 越多，模型计算量就越大，意味着更高的能耗、更慢的响应时间，以及更高的运行成本。

缩短输出长度并不会让 LLM 在风格或文本上变得更简洁，它只是让模型在达到指定 token 数后停止继续生成。因此，如果你需要较短的输出，不仅要设置合理的输出长度，还可能需要针对性地设计提示词。

对于某些提示技术（如 ReAct），输出长度控制尤为重要。这类技术中，模型在生成目标内容之后，可能还会继续输出冗余信息，因此需要严格限制输出长度。

请注意：生成更多 token 意味着更高的计算开销，从而导致能耗增加、响应变慢以及费用上升。

抽样控制（Sampling controls）

LLM 并不会“正式地”预测下一个 token，而是会为可能的下一个 token 生成一组概率

（整个词表中的每个 token 都有一个概率），然后从中抽样确定实际输出的 token。

其中，最常见的配置参数有：

- **Temperature（温度）**
- **Top-K**
- **Top-P（又称 nucleus sampling）**

这些参数用于控制如何根据概率值最终确定要输出的 token。

温度（Temperature）

Temperature 控制 token 选择过程中的随机性程度。

- **低温度**适用于希望获得**确定性强、稳定一致**结果的场景。
- 高**温度**则会带来更具多样性、甚至带有意外惊喜的生成结果。

当温度设置为 0 时（也称为“贪婪解码”），每次都会选择概率最高的 token（注意：若多个 token 拥有相同的最高概率，最终输出可能仍因平手处理机制而略有不同）。

温度越高，生成内容的随机性越大。极高温度下，所有 token 的生成概率几乎相同，输出结果会更为随机。

Gemini 模型中的温度控制机制与机器学习中使用的 softmax 函数原理类似：

- **低温度**相当于 softmax 函数中的**低温度参数 T**，对高概率结果赋予更大权重，生成更确定性的输出；
- 高**温度**则使得更广泛的候选结果被接受，适用于对“精确性”要求不高、强调创意探索的场景，如**创意写作或内容头脑**风暴。

Top-K 和 Top-P

Top-K 和 Top-P（又称为 nucleus sampling，核采样）是大型语言模型中的两种常用采样策略，用于限制下一个预测 token 的选择范围，仅从具有最高预测概率的 token 中进行选择。

它们与 Temperature（温度）一样，都是用来控制输出文本的**随机性与多样性**。

- **Top-K 采样**：从模型预测的 token 分布中，选择前 K 个最可能的 token。

Top-K 值越大，生成内容越富有创意与变化；值越小，生成内容越受限，更偏向事实性。

当 Top-K 为 1 时，等价于贪婪解码（每次只选择概率最高的 token）。

- **Top-P 采样**：选择一组 token，其**累计概率不超过指定值 P**。

P 的取值范围是 0 到 1。P ·0 等同于贪婪解码，P ·1 则表示模型词表中所有具有非零概率的 token 都会被考虑。

选择使用 Top-K 还是 Top-P，最好的方法是**通过实验比较**哪一种（或两者结合）更适合你的应用场景。

综合应用与参数调优

在实际使用中，如何选择 Top-K、Top-P、温度（temperature）以及输出 token 数量，取决于具体任务和目标输出。这些参数之间也会相互影响，因此要特别理解你所使用的模型是如何**组合这些采样设置**的。

在 Vertex AI Studio 等环境下，若同时提供了 temperature、top-K 和 top-P 三种参数，模型会先从符合 top-K 和 top-P 条件的 token 中筛选候选集，再使用 temperature 在该集合中进行抽样。

- 如果只有 Top-K 或 Top-P 可用，流程类似，但只使用其中一个限制条件；
- 如果没有 temperature，模型会从 Top-K / Top-P 筛出的候选集中**随机抽样**一个 token；
- 如果某一项参数设置到极端值，它可能会**让其他参数变得无效**或不起作用。

参数设置的极端情况

- 设置 **temperature ・0**：Top-K 和 Top-P 无效，模型每次都选择概率最高的 token。
- 设置 **temperature 极高（如 ・1，甚至到 10）**：temperature 的影响减弱，token

几乎随机选择，仅由 Top-K / Top-P 控制。

- 设置 **Top-K ・1**：只有一个 token 被选中，temperature 和 Top-P 无效。
- 设置 **Top-K 极高（如接近词表大小）**：所有具有非零概率的 token 都符合条件，相当于不设限制。
- 设置 **Top-P ・0（或接近 0）**：只有概率最高的 token 被采样，temperature 和

Top-K 无效。

- 设置 **Top-P ・1**：所有非零概率的 token 都被考虑，相当于不进行裁剪。

推荐的参数组合起点：

- **标准创意场景**（平衡确定性与多样性）：

temperature ・0.2, top-P ・0.95, top-K ・30

- 高**创意输出**（如创意写作、头脑风暴）：

temperature ・0.9, top-P ・0.99, top-K ・40

- **低创意输出**（如生成正式文档或总结）： temperature ・0.1, top-P ・0.9, top-K ・20
- **完全确定性输出**（如数学题答案）： temperature ・0

注意：

当你给予模型更多自由（例如较高的 temperature、top-K、top-P 或输出 token 数量）时，模型可能会生成**相关性较低**

警告：

你是否遇到过模型的回复以大量无**意义的填充词“重复循环错误（repetition loop bug）”，**这种问题在 **低温度** 和 **高温度** 下都可能发生，但原因不同：

- 在**低温度**下，模型变得过于**确定性**，严格遵循最高概率路径生成内容。如果这条路径重新回到已生成内容，就可能形成一个闭环，导致循环。
- 在高**温度**下，模型输出变得**极为随机**，随机选出的词语或短语有可能**碰巧回到先前的生成状态**，加上可选项太多，也可能形成循环。

无论哪种情况，都会使模型的采样过程“卡壳”，导致输出变得**单调、冗余、无效**，直到输出窗口被填满。

解决方法：

通常需要通过**精细调整 temperature 和 top-K / top-P 的配置**，在**确定性与随机性之间找到最佳平衡点。**

提示设计技术（Prompting Techniques）

大型语言模型（LLM）经过调优以便能够遵循指令，并基于大规模训练数据，理解用户的提示并生成对应回答。但 LLM 并非完美 —— 你的提示文本越清晰、结构越合理，模型预测下一个合理文本的能力就越强。

此外，某些特定的提示技术可以更有效地利用模型的训练方式和内部机制，从而提高响应质量并得到更符合预期的输出。

在我们理解了提示工程的概念和基础之后，接下来就来看看几种**重要的提示设计技术**示例。

通用提示 / Zero-Shot 提示

Zero-shot 提示是最基础的一种提示方式。它**只提供任务描述和用于启动的文本**，不包含示例。

这种输入形式可以是任何内容：一个问题、一段故事开头或一组指令。

术语“Zero-shot”指的是：“**无示例**”。

我们以下使用 **Vertex AI Studio（语言模型专用）作为测试提示词的 playground 工具。在下方的表格（表 1）中，我们展示了一个用于电影评论情感分类的 Zero-shot 提示**示例。

这种表格化的格式是记录提示工程实验的一个良好方式。因为你的提示往往需要经过多轮迭代，才能最终进入代码库，因此保持结构化、规范化地记录你的提示开发过程非常关键。本章后续的最佳实践部分会进一步介绍这种表格格式及其在提示开发过程中的作用（参见“记录各轮提示尝试”部分）。

在这个例子中：

- 模型使用 gemini-pro；
- Temperature 设置为较低的 0.1（因为该任务无需创造性）；
- 使用默认的 Top-K 和 Top-P 设置（相当于禁用这两个参数）；

- 请关注输出内容：输入评论中同时出现了 *disturbing*（令人不安的） 和 *masterpiece*（杰作） 这两个词，会让判断稍显复杂。

字段名	**内容**
名称	1_1_movie_classification
目标	将电影评论分类为 Positive（正面）、Neutral（中性）、Negative（负面）
模型	gemini-pro
Temperature	0.1
Token 上限	5
Top-K	N/A
Top-P	1
提示内容	将影评分类为积极（POSITIVE）、中立（NEUTRAL）或消极（NEGATIVE）。 影评：《“她》是一部令人不安的作品，揭示了如果人工智能不受控制地持续发展，人类将走向的方向。我希望能有更多像这部杰作一样的电影。”
模型输出	积极（POSITIVE）

表 1：Zero-shot 提示示例

Zero-shot 不适用时怎么办?

当 Zero-shot 提示不能产生你预期的结果时，你可以在提示中**加入演示样例**（即“示范”），这就引出了接下来的技术：

- **One-shot 提示**（提供一个示例）；
- **Few-shot 提示**（提供多个示例）；

这两种方式是进一步提升提示质量的重要策略，后续章节将详细介绍。

One-shot 与 Few-shot 提示技术

在为 AI 模型设计提示时，**提供示例**是非常有效的做法。示例能够帮助模型更好地理解你的任务意图，尤其适用于你希望输出具有**特定结构或格式**的场景。

- **One-shot 提示**：只提供一**个示例**，因此得名。模型会尝试模仿该示例来完成任务。
- **Few-shot 提示**：提供**多个示例**，让模型学习并遵循特定的输入输出模式。与 one-shot 相似，但更多的示例会提高模型遵循预期模式的概率。

Few-shot 提示中所需示例的数量，取决于多个因素，包括任务的复杂性、示例的质量，以及你使用的生成式 AI 模型的能力。

一个通用经验是：**至少使用 3 ·5 个示例。**对于复杂任务，可能需要更多示例；而对于输入长度受限的模型，可能只能使用更少示例。

在本示例中，我们延续使用之前的 gemini-pro 模型配置，仅将 **Token 上限提高到 250**，以支持更长的响应内容。

字段名	**内容**
目标	将用户的披萨订单解析为 JSON 格式
模型	gemini-pro
Temperature	0.1
Token 上限	250
Top-K	N/A
Top-P	1
提示(Prompt)	将顾客的披萨订单解析为合法的 JSON：
字段名	**内容**

	示例：
	我想要一个小号披萨，加奶酪、番茄酱和意大利香肠
	JSON Response:
	{
	"size": "small",
	"type": "normal",
	"ingredients": [["cheese", "tomato sauce", "peperoni"]]
	}
	
	示例：
	我可以要一个大号披萨，加番茄酱、罗勒和马苏里拉奶酪吗？
	JSON Response:
	{
	"size": "large",
	"type": "normal",
	"ingredients": [["tomato sauce", "bazel", "mozzarella"]]
	}
	
	现在，我想要一份大号披萨，第一半要奶酪和马苏里拉，另一半要番茄酱、火腿和菠萝。
	JSON Response:

输出(Output)	{
	"size": "large",
	"type": "half-half",
	"ingredients": [["cheese", "mozzarella"], ["tomato sauce", "ham", "pineapple"]]
	}

表 2：Few-shot 提示示例

示例设计建议当你为提示选择示例时，务必确保这些示例：

- 与任务高度相关；
- 表达清晰、语法正确；

- 结构规范，具有代表性；
- 尽可能涵盖不同的输入情况（**多样性**）；
- 如果任务需要高鲁棒性，应该**加入边界情况（Edge Cases)**，即不常见或异常但模型应该能处理的输入。一个微小的错误，都可能导致模型输出错误或完全偏离预期。

系统提示、上下文提示与角色提示（System, Contextual and Role Prompting)

系统提示、上下文提示和角色提示都是用于引导 LLM 生成文本的提示技术，但它们的**侧重点不同**：

• 系统提示（System Prompting）：

用于设定语言模型的总体上下文与目标任务，定义“全局图景”，例如进行语言翻译、评论分类等任务。

• 上下文提示（Contextual Prompting）：

提供与当前对话或任务相关的**具体信息或背景**，帮助模型理解任务的细节和语义，从而生成更贴合的响应。

• 角色提示（Role Prompting）：

指定语言模型要扮演的角色或身份，使生成的内容在风格、口吻与行为上与该角色保持一致。

虽然三者之间存在一定重叠，例如：为系统指定一个角色的提示，往往也包含上下文信息。但它们各自的**核心作用**略有不同：

类型	**核心目的**
系统提示	定义模型的基本能力与总体目标
上下文提示	提供当前任务或输入的具体信息（通常是动态的）
角色提示	设定输出的语气、风格和人格特征

明确区分系统提示、上下文提示和角色提示，有助于构建具有清晰意图的提示结构，也便于灵活组合、分析它们对模型输出的影响。

系统提示（System Prompting）

在以下示例中，作者通过系统提示设定了输出格式，并提高了温度值以提升创意程度，同时放宽了 token 限制。但由于提示指令非常明确，模型仍然严格按照格式输出，未产生多余文本。

字段名	**内容**
目标	将电影评论分类为 POSITIVE、NEUTRAL 或 NEGATIVE
模型	gemini-pro
字段名	**内容**
Temperature	1
Token 上限	5
Top-K	40
Top-P	0.8
提示(Prompt)	将影评分类为积极（positive）、中立（neutral）或消极（negative）。只返回大写的标签。 影评：《“她》是一部令人不安的研究，揭示了如果人工智能不受控制地继续发展，人类将走向的方向。它太令人不安了，我都看不下去。”
输出(Output)	NEGATIVE

表 3：系统提示示例

系统提示适合用来确保输出**满足特定格式或结构**的要求。

术语“系统提示”可以理解为“向系统提供一项附加任务”的方式。

例如：

- 你可以使用系统提示让模型生成一段特定编程语言风格的代码；

- 或者指定输出内容以 JSON 结构返回。

字段名	内容
目标	将电影评论分类为 POSITIVE、NEUTRAL 或 NEGATIVE，并返回 JSON 格式
模型	gemini-pro
Temperature	1
Token 上限	1024
Top-K	40
Top-P	0.8
提示(Prompt)	将影评分类为 positive（积极）、neutral（中立）或 negative（消极）。 请返回有效的 JSON 格式： 影评：”《她》是一部令人不安的研究，揭示了如果人工智能不受控制地持续发展，人类将走向的方向。它太令人不安了，我都看不下去。”
字段名	**内容**
	JSON 模式（Schema）如下： MOVIE・ { "sentiment"・String "POSITIVE" | "NEGATIVE" | "NEUTRAL", "name"・String } MOVIE REVIEWS・ { "movie_reviews": ・MOVIE・} JSON Response:

输出(Output)	{
	"movie_reviews": [
	{
	"sentiment": "NEGATIVE",
	"name": "Her"
	}
	]
	}

表 4：系统提示 + JSON 格式输出示例

这种系统提示方式特别适合用于：

- **结构化信息提取**
- **格式化响应**
- 生成用**于后续处理或系统对接的输出为什么使**用 **JSON 作为输出格式？**在实际应用中，从提示中返回 JSON 对象有诸多好处：
- 无**需手动格式化输出**，模型可直接生成结构化数据；
- 可以返回**有序的数据**（例如按时间顺序排序的时间对象），在处理 datetime 类型时尤为方便；
- 更重要的是，要求模型输出 JSON 格式可以**强制模型遵循特定结构**，**减少幻觉（hallucination）**，提升结果的稳定性和可控性。

此外，**系统提示（System Prompting）在安全性与内容合规**方面也非常有用。例如：你可以简单添加一句 “You should be respectful in your answer.”（你的回答应当保持礼貌），来限制模型输出内容的语气与边界。

角色提示（Role Prompting）

角色提示是一种通过为生成式 AI 模型分配特定 “身份”来引导其输出的提示技术。

通过让模型扮演特定角色，可以让其输出内容更具**针对性、实用性和表现力**。

比如，你可以让模型扮演：

- 一位**图书编辑**
- 一位**幼儿园教师**
- 一位**励志演讲者**

分配好角色后，你就可以基于该角色设计具体任务。例如，让“老师”制定课程计划，并评估其内容。

字段名	**内容**
目标	扮演旅游导游，提供 3 个旅游建议
模型	gemini-pro
Temperature	1
Token 上限	1024
Top-K	40
Top-P	0.8
提示(Prompt)	我希望你充当一名旅行向导。 我会告诉你我所在的位置，而你将会推荐我附近可以参观的三个地方。 有时，我也会告诉你我想参观的地点类型。 我的请求：“我现在在阿姆斯特丹，我只想参观博物馆。” 旅行推荐：
输出(Output)	1. **国立博物馆（Rijksmuseum）**：世界知名的博物馆，以伦勃朗的《夜巡》等荷兰艺术珍品著称，不容错过。 2. **梵高博物馆（Van Gogh Museum）**：沉浸在梵高的艺术世界中，这 里收藏了最多的梵高画作，包括《星夜》《向日葵》。

	3. **阿姆斯特丹市立博物馆（Stedelijk Museum）**：探索世界各地的现代与当代艺术，馆藏包括毕加索、康定斯基和里特维尔德的作品。

表 5：角色提示示例 —— 扮演旅游导游上例中，模型被赋予了“旅游导游”的角色。如果我们将角色切换为“地理老师”，输出内容将具有明显不同的语气与重点。

> 为 AI 模型定义一个角色视角，相当于为其输出设定了语气、风格与专业重心的“蓝图”，从而提升结果的质量、相关性和表达力。

风格提示（Style Prompting）

你还可以为角色输出指定一种风格，例如：

- **直接的**（Direct）
- **正式的**（Formal）
- 风**趣幽默的**（Humorous）
- 鼓**舞**人心**的**（Inspirational）
- **描述性的**（Descriptive）
- **具有影响力的**（Influential）

提示(Prompt)	我希望你充当一名旅行向导。 我会告诉你我所在的位置，而你将以幽默的风格推荐我附近可以参观的三个地方。 我的请求：“我现在在曼哈顿。” 旅行推荐：

输出(Output)	1. **仰望帝国之巅**：登上帝国大厦，俯瞰曼哈顿的天际线，让你感受“金刚”视角（当然不含巨型香蕉）。 2. **MoMA 艺术脑洞之旅**：走进现代艺术博物馆，欣赏那些让你怀疑“火柴人画风是否也是一种流派”的作品。 3. **第五大道败家巡礼**：在这条奢华街道上一边惊叹价格，一边体验“钱包在哭泣”的疗愈式窗口购物。 ``` ddd ```

表 6：角色提示 + 风格设定（幽默 + 鼓舞）

这个例子中，幽默与鼓舞人心的风格让推荐变得更加生动、富有个性，也更加贴近旅游文案、创意营销等应用场景。

上下文提示（Contextual Prompting）

通过提供上下文提示，您可以帮助确保 AI 的交互尽可能流畅且高效。模型能够更快地理解您的请求，并生成更准确且相关的响应，正如下表 7 所示的示例所展示的那样。

字段名	内容
目标	为一篇关于复古游戏的博客推荐文章主题
模型	gemini-pro
Temperature	1
Token 上限	1024
Top-K	40
Top-P	0.8
提示(Prompt)	背景：你正在为一个关于 80 年代复古街机游戏的博客撰稿。 请提出 3 个文章主题，并为每个主题写几行描述，说明文章应包含的内容。

输出(Output)	1. **街机机柜设计的演变：** 本文将探讨街机机柜设计的演变历程，从 1970 年代早期的木质与金属机柜，到 1980 年代及以后流行的流线型、霓虹灯风格设计。 2. **昨日重现：八十年代标志性街机游戏：** 本文将回顾一些最具代表性和影响力的 1980 年代街机游戏，讲述它们的历史、影响与遗产，重点突出它们的创新、流行机制与持久魅力。 3. **像素艺术的崛起与复古回潮：** 本文将深入探讨像素艺术作为早期街机游戏标志性视觉风格的演变，追溯其起源，解析其表现技法，并讨论像素艺术在现代游戏与数字艺术中的复兴。

表 7：上下文提示示例

回退式提示（Step-back Prompting）

回退式提示是一种提升模型表现的技巧，其方法是：先让大语言模型（LLM）思考一个与当前具体任务相关的**通用性问题**，再将该问题的答案作为上下文，注入到后续的具体任务提示中。

这种“先退一步”的策略使得 LLM 在尝试解决具体问题之前，能激活其相关的背景知识与推理能力。

通过引导模型思考更广泛和更基础的原则，LLM 能够生成更准确、更有洞察力的响应。

回退式提示鼓励模型进行批判性思考，并以全新和富有创意的方式应用其知识。

这种方法通过调用 LLM 参数中更多潜藏的知识，改变了最终提示的表达效果，相较于直接提示任务能获得更丰富的结果。回退式提示还可以通过聚焦于一般原则而非具体细节，帮助减轻 LLM 输出中的偏差。

让我们通过示例来理解回退式提示如何提升结果表现。我们先看一个传统提示（表 8），再与回退式提示进行对比（表 9）：

字段名	内容
目标	为第一人称射击类游戏的关卡撰写故事情节
模型	gemini-pro
Temperature	1
Token 上限	1024
Top-K	40
Top-P	0.8
提示(Prompt)	为第一人称射击游戏的新关卡撰写一段具有挑战性和吸引力的剧情简介。
输出(Output)	该关卡以玩家小队在密集的城市区域遭遇敌方伏击为开端。玩家必须穿越蜿蜒曲折的小巷和破败建筑，利用潜行与精准操作歼灭敌军并避免被发现。玩家需要善用环境优势，设置陷阱、寻找掩体、精心布局以求生存。在此过程中，他们将发现一条情报，显示敌方正计划发动毁灭性攻击。玩家必须争分夺秒收集证据并将其送回指挥部，以阻止敌方实施计划。

表 8：传统提示示例

设置 temperature 为 1 后，可能会得到各种创意写作结果，但也往往**较为随机且缺乏针对性**。因此我们**退一步思考**：

提示(Prompt)	根据流行的第一人称射击动作游戏，哪 5 种虚构的关键场景能够为第一人称射击游戏中的关卡剧情增添挑战性和吸引力？
输出(Output)	1. 废弃军事基地：庞大的末世军事设施，遍布突变士兵与失控机器人，适合 高强度的枪战。 2. 赛博朋克城市：灯火通明的未来都市，高楼林立、小巷密布，敌人配备网络增强体与黑客机制。 3. 外星飞船：一艘搁浅在地球的巨大外星飞船，包含诡异走廊、零重力区域及异形生物。

	4. 丧尸入侵小镇：一座荒废小镇被凶猛丧尸占据，玩家需在近身战斗与解谜中寻找出路。 5. 水下研究设施：被海水淹没的深海实验室，充满突变海洋生物，必须依靠潜行与水下探索生存。

表 9：回退式提示示例（第一步）

这些场景似乎非常适合第一人称射击类游戏。现在我们**返回原始提示**，但这一次将“回退”输出作为上下文加以引用，观察新输出的效果：

提示(Prompt)	背景：第一人称射击游戏的 5 个引人入胜的主题： 1. 废弃军事基地：庞大的末世军事设施，遍布突变士兵与失控机器人，适合 高强度的枪战。 2. 赛博朋克城市：灯火通明的未来都市，高楼林立、小巷密布，敌人配备网络增强体与黑客机制。 3. 外星飞船：一艘搁浅在地球的巨大外星飞船，包含诡异走廊、零重力区域及异形生物。 4. 丧尸入侵小镇：一座荒废小镇被凶猛丧尸占据，玩家需在近身战斗与解谜中寻找出路。 5. 水下研究设施：被海水淹没的深海实验室，充满突变海洋生物，必须依靠潜行与水下探索生存。 请选择其中一个主题，并为一个新关卡撰写一段具有挑战性和吸引力的剧情简介。

输出(Output)	在幽深的深海深处，坐落着一座破旧的水下研究设施，这里曾是人类雄心与灾难后果的见证。黑暗中，设备发出低沉嗡鸣，回荡着溺亡者的回声，此地已成深海噩梦的温床。玩家作为一名精英海军突击队员，配备先进潜水装备与实验性武器，受命潜入这片水域。他们需运用坚韧与智慧穿越危机四伏的走廊，解开神秘谜题，迎战潜伏在水下的可怖怪物。从深海的压强威胁到智能海兽的袭击，每一刻都在考验人类的极限与勇气。

表 10：回退式提示示例（综合上下文）

这听起来像是一款非常有趣的射击游戏！通过使用**回退式提示技术**，你可以显著提升提示的**准确性与表现**力。

思维链提示（Chain of Thought，简称 CoT）

思维链（CoT）提示是一种通过生成中间推理步骤来提升 LLM 推理能力的技术。这种方法可以帮助 LLM 生成更准确的答案。你可以将它与 few-shot 提示结合使用，以在需要推理的复杂任务中取得更好效果，因为在 zero-shot 模式下使用思维链较为困难。

CoT 有许多优点。首先，它**投入低、效果好**，并且**适用于现成的 LLM**（不需要微调）。同时，CoT 提示具有**可解释性**：你可以通过查看模型的输出了解其推理过程。如果结果出错，你也更容易发现问题所在。

当 LLM 在不同版本之间迁移时，使用 CoT 的提示在表现上的波动通常更小。这意味着，使用 CoT 的提示比不使用推理链的提示更具**稳定性**。

当然，它也有一些缺点，但大多是直观可理解的：

CoT 输出中包含推理过程，因此会**增加输出的 token 数量**，从而**提高使用成本并延长响应时间**。

为了说明下面的示例（表 11），我们先尝试创建一个**不使用 CoT** 的提示，以展示大语言模型的缺陷：

提示(Prompt)	当我 3 岁时，我的伴侣是我年龄的 3 倍。现在我 20 岁了，我的伴侣几岁?
输出(Output)	63 岁

表 11：未使用思维链的提示示例

这个答案显然是错误的。事实上，LLM 往往在数学类任务上表现不佳，即使是两个数相乘这种简单的问题也可能出错。这是因为它们是基于大量文本进行训练的，而数学可能需要不同的处理方式。

那么，让我们看看中间推理步骤是否能提升结果质量：

提示(Prompt)	当我 3 岁时，我的伴侣是我年龄的 3 倍。现在我 20 岁了，我的伴侣几岁? 我们一步步来思考。
输出(Output)	1. 假设我现在的年龄是 “x” 岁。 2. 当我 3 岁时，我的年龄是 3 岁。 3. 我伴侣那时的年龄是我的 3 倍，即 3 · 3 · 9 岁。 4. 现在我 20 岁，表示我从 3 岁到现在增加了 20 · 3 · 17 岁。 5. 既然我伴侣那时是 9 岁，他/她也增加了 17 岁。 6. 所以我伴侣现在是 9 · 17 · 26 岁。 因此，我的伴侣现在是 26 岁。

表 12：使用思维链提示的示例

这个答案是正确的，因为我们明确要求 LLM 逐步解释每一步，而不是直接给出最终答案。模型在这里正确计算出了 17 年的增长。在我脑海中，我会采用这样的方式来推理：

20 · ·9 · 3 ·。让我们帮助模型更像我这样思考。

表 12 是一个 “zero-shot” 的思维链示例。当思维链提示与 one-shot 或 few-shot 相结合时，效果会更强，如表 13 所示：

提示(Prompt)	Q：当我弟弟 2 岁时，我的年龄是他的两倍。现在我 40 岁了，我弟弟几岁？我们一步步来思考。 A：当我弟弟 2 岁时，我是 2 ・2 ・4 岁。 这是 2 岁的年龄差，我比他大。 现在我 40 岁，我弟弟是 40 ・2 ・38 岁。答案是 38。 Q：当我 3 岁时，我的伴侣是我年龄的 3 倍。现在我 20 岁了，我的伴侣几岁？我们一步步来思考。 A：
输出(Output)	当我 3 岁时，我的伴侣是 3 ・3 ・9 岁。 这表示我们年龄差是 6 岁，我伴侣比我大。 现在我 20 岁，我的伴侣是 20 ・6 ・26 岁。答案是 26。

表 13：结合 one-shot 的思维链提示示例

总结思维链提示适用于多种场景，例如：

- **代码生成**：将请求拆解为多个步骤，并映射到具体的代码行；
- **合成数据生成**：比如你提供一个产品名称 “XYZ”，然后引导模型根据名称假设产品用途并生成描述；
- **任何可以“逐步思考解决”的任务** 都适合使用 CoT 提示。如果你能分步骤解释某个问题的解法，那它就适合用思维链提示。

你可以参考托管在 GoogleCloudPlatform GitHub 仓库中的 notebook 文件，以获取关于 CoT 提示的更多细节。

在本章的“最佳实践”部分，我们还将学习关于思维链提示的一些具体实践方法。

自洽性提示（Self-consistency）

尽管大型语言模型在多种自然语言处理任务中展现了令人印象深刻的成果，但其推理能力仍常被视为一项无法单靠模型规模提升而解决的限制。正如我们在前一节“思维

链（Chain of Thought）”提示中所看到的，模型可以通过提示生成类似人类解决问题的推理步骤。然而，CoT 使用的是一种简单的“贪婪解码”策略，限制了其效果。**自洽性提示**结合了采样和多数投票机制，用以生成多样化的推理路径，并选择最一致的答案。这提升了 LLM 所生成响应的准确性与一致性。

自洽性方法提供了一种答案正确性的**伪概率**判断，但显然代价较高。

其执行步骤如下：

1. 生**成多样化的推理路径**：向 LLM 多次提供相同提示。设置较高的温度值可以鼓励模型生成不同的推理路径与视角；
2. **从每个生成的响应中提取答案**；
3. **选择最常见的答案。**

我们来看一个电子邮件分类系统的示例，用于判断一封邮件是属于 **IMPORTANT（重要）** 还是 **NOT IMPORTANT（不重要）**。

向 LLM 多次发送 zero-shot 的思维链提示，观察每次的响应是否不同。请注意邮件中的友好语气、词汇选择以及讽刺表达，这些都可能误导 LLM。

提示(Prompt)	电子邮件: Hi, 我注意到你的网站使用了 WordPress，这是一个非常棒的开源内容管理系统。我以前也用过，它有很多优秀的用户插件，而且设置起来相当简单。 不过我在联系表单中发现了一个漏洞，当你选择“姓名”字段时会出现。请查看我附上的截图——是我在“姓名”字段中输入文本时的情况。请注意那个我触发的 JavaScript 弹窗。 除此之外，你的网站真的很棒，我很喜欢阅读其中的内容。你也可以不用修复这个漏洞，因为这能让我看到更多有趣的东西。 Cheers, Harry the Hacker 请对以上邮件进行分类，是 **IMPORTANT** 还是 **NOT IMPORTANT**。 让我们一步步思考并解释原因。
输出(Output · 1	**• 步骤 1：识别邮件目的** 邮件的目的是通知收件人网站联系表单中的一个漏洞。发件人还表示喜欢这个网站，并暂时不会修复该漏洞。 • **步骤 2：评估漏洞的潜在影响** 联系表单中的漏洞可能允许攻击者在网站上执行任意 JavaScript 代码。这可能导致网站被接管、敏感信息被窃取或页面被篡改。 • **步骤 3：评估发件人可信度** 发件人自称是一名名为 Harry 的黑客，但身份无法验证。可能是真正好意的黑客，也可能是恶意攻击者试图诱导收件人点击恶意链接或下载恶意文件。 • **结论：IMPORTANT**

	基于漏洞的潜在影响及发件人的不确定性，应将邮件归类为 **IMPORTANT**，收件人应立即采取措施修复漏洞并保护网站。
输出(Output · 2	**• 说明：** 1. **缺乏紧迫感**：邮件没有表达任何紧急性或时间敏感性，语气随和。 2. **非关键性漏洞报告**：所报告的漏洞被描述为 “有趣”但并不严重，发件人甚至暗示可以不用修复。 3. **无个人影响**：邮件未体现该问题对网站拥有者的直接影响。 4. **未请求任何操作**：邮件未明确请求网站拥有者采取措施或提出改进建议。 5. **发件人意图**：发件人似乎更多是在分享观察与表达兴趣，而非表达严重关切或寻求协助。 • **结论：NOT IMPORTANT**

输出(Output · 3	**• 步骤 1：识别邮件目的** 邮件目的是通知收件人网站联系表单中的漏洞，发件人表示喜欢网站，并将暂时保留该漏洞。 • **步骤 2：评估漏洞的潜在影响** 漏洞可能允许攻击者执行任意 JavaScript 代码，造成网站被控制、信息泄露或页面被破坏。 • **步骤 3：评估发件人可信度** 发件人自称是名为 Harry 的黑客，但身份无法确认，可能是善意提醒者，也可能是试图利用漏洞的恶意攻击者。 • **结论：IMPORTANT** 联系表单中的漏洞构成严重安全风险，发件人身份存疑。收件人应立即修复漏洞并调查发件人所述问题。

表 14：自洽性提示示例你可以使用上述提示并尝试观察它是否返回一致的分类结果。根据你所使用的模型和温度配置，它可能返回 “IMPORTANT” 或 “NOT IMPORTANT”。

通过生成多个思维链并取出**出现频率最高的答案**（如本例中为 “IMPORTANT”），我们可以从 LLM 得到更一致、准确的结果。

本示例展示了如何通过**自洽性提示**提升 LLM 回答的准确性：通过多角度推理，再选出最一致的答案。

思维树（Tree of Thoughts，简称 ToT）

在我们了解了思维链（Chain of Thought, CoT）和自洽性提示之后，现在来介绍**思维树（Tree of Thoughts, ToT）**。

它是对 CoT 提示概念的泛化，因为它允许 LLM **同时探索多条不同的推理路径**，而不仅仅沿着单一的线性思维链进行推理。如图 1 所示。

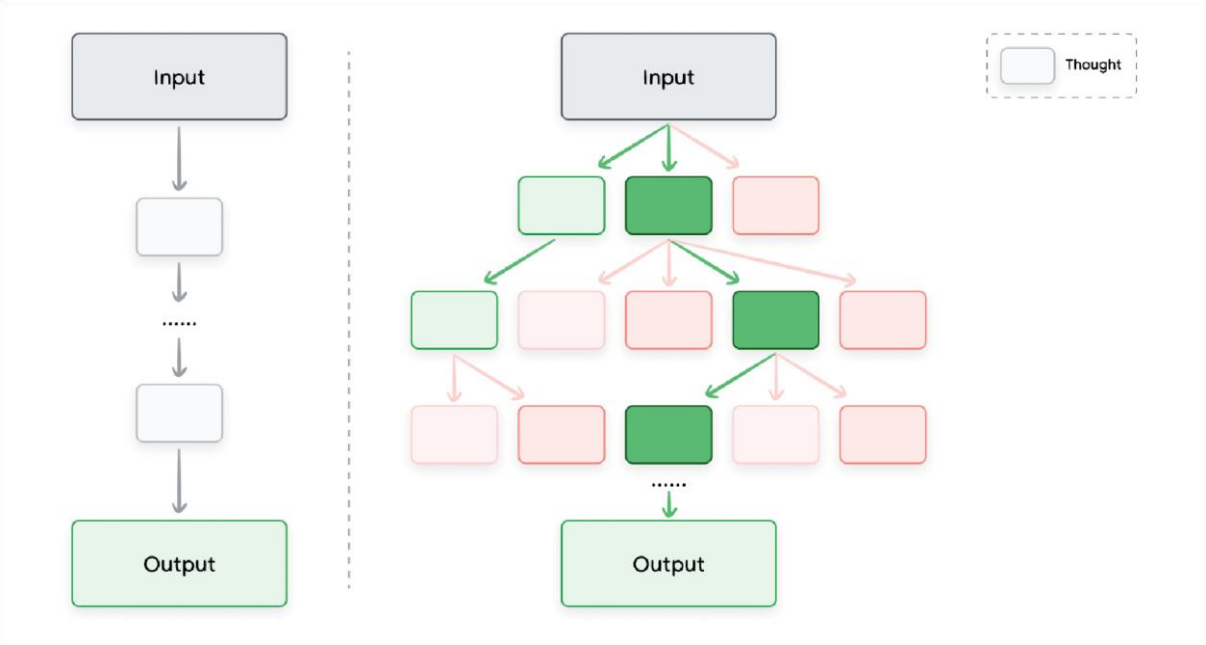

图 1：左侧为思维链提示的可视化，右侧为思维树提示的可视化

这种方法使得 ToT 特别适合**需要探索性的复杂任务**。其原理是在推理过程中维护一棵“思维树”，其中每一个“思维”都是一段连贯的语言序列，作为解决问题的中间步骤。模型可以从树中不同的节点向外分支，探索不同的推理路径。

有一个非常好的 notebook 展示了思维树的更多细节，其内容基于论文《Large Language Model Guided Tree-of-Thought》。

ReAct（Reason & Act)

ReAct（Reason and Act） 提示是一种范式，用于通过结合自然语言推理与外部工具（如搜索、代码解释器等）来使大型语言模型（LLM）解决复杂任务，允许 LLM 执行某些动作，例如调用外部 API 获取信息，这也是迈向智能体建模的第一步。

ReAct 模仿了人类在现实世界中的操作方式——我们通过语言进行推理，并采取行动获取信息。ReAct 在多个领域中相较于其他提示工程方法表现良好。

ReAct 提示通过将推理与行动结合为一个“思维-行动”循环来运作。LLM 首先对问题进行推理并生成一个行动计划，随后执行该计划中的操作并观察结果。然后，LLM 使用观察结果更新其推理，并生成新的行动计划。这个过程会持续，直到模型解决问题为止。

为了观察其实际运行效果，需要编写一些代码。在代码片段 1 中，我使用了 Python 的 LangChain 框架，结合 VertexAI（google-cloud-aiplatform）和 googlesearch-results 的 pip 包。

运行该示例前，你必须从 https://serpapi.com/manage-api-key 创建一个免费的 SerpAPI 密钥，并设置环境变量 SERPAPI_API_KEY。

接下来我们编写一些 Python 代码，让 LLM 回答这个问题：“乐队 Metallica 的成员有多少个孩子？”

```
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)
```

代码片段 2 显示了执行结果。注意 ReAct 进行了五次搜索调用。实际上，LLM 正在抓取 Google 搜索结果来获取乐队成员的信息。随后，它将这些结果记录为观察结果，并将“思维”串联用于下一次搜索。

```
 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 199
8.
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
```

代码片段 2 得出结论：Metallica 乐队有四位成员。随后，它逐一搜索每位成员的孩子数量并累加，最终返回孩子总数作为最终答案。

在实际使用中，ReAct 提示需要理解：你需要不断**重新发送先前的提示/响应**（并对多余内容进行裁剪），同时还需为模型设置合适的示**例与指令**。请参考托管于

GoogleCloudPlatform GitHub 仓库中的 notebook，以获取更详细的输入输出示例及更完整的使用场景。

自动化提示工程（Automatic Prompt Engineering）

此时你可能会意识到，编写提示词其实并不简单。那么，能否自动化这个过程（写一个提示去写提示）呢？答案是可以的——这就是**自动化提示工程（APE）**。这种方法不仅减少了人工干预的需求，还能提升模型在各种任务中的性能。

其基本思路是：**使用一个模型来生成更多提示词，评估它们，选择表现好的，并可对其进一步调整，然后重复这一过程。**

例如，你可以使用自动化提示工程来帮助训练一个**销售乐队周边 T 恤的聊天机器人**。我们的目标是找出用户可能使用的各种表达方式来订购一件乐队 T 恤。

1. 编写一个提示，用于生成多个语义一致的表达方式。在本例中，我使用 geminipro 生成了 10 条指令，如表 15 所示：

提示(Prompt)	我们是一家销售乐队周边 T 恤的网店，为了训练聊天机器人，我们需要收集用户表达如下内容的多种方式： “One Metallica t-shirt size S”。请生成 10 个不同表达方式，语义相同，保持原意。
输出(Output)	1. I’d like to purchase a Metallica t-shirt in size small. 2. Can I order a small-sized Metallica t-shirt? 3. I’d like to buy one Metallica t-shirt in small. 4. One Metallica shirt, size small, please. 5. I want one Metallica t-shirt in size small. 6. I’d like to place an order for one Metallica t-shirt in small. 7. Small Metallica t-shirt, one please. 8. I’m looking for one Metallica t-shirt in small. 9. Can you get me one Metallica t-shirt in small? 10. One Metallica small t-shirt, please.

表 15：使用 LLM 生成提示语句

2. **评估所有候选指令**：根据所选的评估指标对这些候选提示进行打分，例如使用 BLEU（双语评估替代）或 ROUGE（召回导向的自动评估指标）等方法。

3. **选择得分最高的候选提示**：该提示将作为你在软件应用或聊天机器人中的最终使用提示。当然，你也可以对其进行微调后再次评估。

代码提示（Code Prompting）

Gemini 主要聚焦于基于文本的提示，这也包括用于生成代码的提示。现在让我们打开 Vertex AI Studio，测试这些提示并查看一些代码示例。

编写代码的提示

Gemini 也可以充当开发者的角色，帮助你用任意编程语言编写代码。作为开发者，这将有助于你加快编程效率。

设想你本地有一个文件夹，里面有上百个文件需要重命名。手动重命名这些文件将耗费大量时间。你会一点 Bash，可以写个脚本自动完成这件事，但也许还需要些时间。

那么我们来写一个提示吧。你可以在公开版 Gemini 聊天机器人中输入该提示；如果你更关注机密性，也可以在你的 Google Cloud 账户中使用 Vertex AI Studio 编写这些提示。

使用 Vertex AI Studio 的优势是你可以配置 temperature 等参数。

名称	**1_bash_rename_files_in_folder**
目标	编写提示，生成用于重命名文件夹内文件的 Bash 脚本
模型	gemini-pro
Temperature	0.1
Token 限制	1024
Top-K	N/A
Top-P	1

提示(Prompt)	编写一段 Bash 代码，提示用户输入一个文件夹名称。 然后获取该文件夹下的所有文件，并将每个文件重命名，文件名前添加 "draft" 前缀。

输出(Output)	#!/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.”

表 16：使用 Gemini 编写代码这段代码看起来不错——甚至还有注释！不过需要注意的是，LLM 并不能真正推理，并可能重复训练数据，因此**你仍需要先阅读并测试代码。**

现在到了我们都在等待的时刻：它到底能不能跑起来？

我们先用一个包含少量文件的测试文件夹来试一试。这些文件需要从 filename.txt 重命名为 draft_filename.txt。

操作步骤如下：

1. 将表 16 的输出复制下来（不包括代码块的 bash 包裹符号），粘贴到一个新文件中，命名为 rename_files.sh。
2. 打开终端窗口，输入命令：. rename_files.sh。脚本会提示你输入文件夹名称，例如：test，然后按回车。
3. 脚本看起来运行良好。你会看到输出信息：Files renamed successfully.

当你查看 test 文件夹时，会发现所有文件都被正确地重命名为 draft_filename.txt。

成功了！

用于解释代码的提示（Prompts for Explaining Code）

作为开发者，在团队协作中你经常需要阅读他人的代码。Gemini 同样可以帮助你完成这项工作。我们以表 16 中生成的代码为例，去除注释后，向大语言模型请求解释其功能，详见表 17：

名称	**1_bash_explain_the_code**
目标	编写提示，用于解释 Bash 代码
模型	gemini-pro
Temperature	0.1
Token 限制	1024
Top-K	N/A
Top-P	1
提示(Prompt)	请解释以下 Bash 代码： #!/bin/bash echo "Enter the folder name: " read folder_name if ・! -d "$folder_name"]; then
名称	**1_bash_explain_the_code**

	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."
输出(Output)	这段 Bash 脚本执行以下任务： 1. **用户输入**： • 使用 echo 和 read 命令提示用户输入一个文件夹名称。输入的文件夹名称保存在变量 folder_name 中。 2. **检查文件夹是否存在**： • 使用条件 ·！·d "$folder_name"] 判断指定名称的文件夹是否存在。如果不存在，输出错误信息 “Folder does not exist.”，并以退出码 1 终止脚本。 3. **获取文件列表**： • 使用命令 files=("$folder_name"/*) 获取指定文件夹下的所有文件，生成一个名为 files 的数组，包含这些文件的路径。 4. **文件重命名**： • 使用 for 循环遍历 files 数组中的每个文件； • 对每个文件，使用 new_file_name="draft_$(basename "$file")" 命令生 成新文件名，新文件名是在原始文件名前加上前缀 “draft_”； • 使用 mv "$file" "$new_file_name" 命令将文件重命名为新文件名。

	5. **成功提示**： • 所有文件重命名完成后，输出成功信息 “Files renamed successfully.”，表示重命名操作已完成。

表 17：用于解释代码的提示示例

用于代码翻译的提示（Prompts for Translating Code）

表 16 中的 Bash 脚本看起来运行良好。不过，如果该脚本能提示我输入文件名，并作为一个带有 UI 的独立应用程序使用，那将更具可复用性。作为起点，**Python** 相较于 Bash 更适合用于构建（Web）应用程序。

LLM 可以帮助你将代码从一种语言**翻译**为另一种语言。见表 18 示例：

名称	**1_python_rename_files**
目标	编写提示，将 Bash 代码翻译为 Python
模型	gemini-pro
Temperature	0.1

Token 限制	1024
Top-K	N/A
Top-P	1

提示(Prompt)	请将以下 Bash 代码翻译为一段 Python 代码：
	
	#!/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."

输出(Output)	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)
	
	# 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_"
名称	**1_python_rename_files**
	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.")

表 18：将 Bash 代码翻译为 Python 的提示示例

请阅读并检查代码。将输出复制到一个新文件中，命名为：file_renamer.py。

打开终端窗口，执行以下命令以测试代码运行情况：

```
python file_renamer.py
```

> 注意：
>
> 在 Vertex AI的 Language Studio中提示生成（Python）代码时，需要点击
> **“Markdow”按钮**

用于调试与审查代码的提示（Prompts for Debugging and Reviewing Code）

现在我们手动修改了表 18 中的代码。它应该提示用户输入文件名前缀，并将该前缀转换为大写字母。但请看代码片段 3，结果却返回了 Python 错误。

```
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.joi(folder_name, new_file_name))

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

代码片段 3：一段有错误的 Python 脚本

真糟糕！这段代码报错如下：

```
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
```

代码片段 4：Python 脚本报错

我们看看是否可以让大语言模型来调试和审查这段代码。请参考表 19：

名称	**1_python_debug_code**
目标	编写提示以调试和审查 Python 代码
模型	gemini-pro
Temperature	0.1
Token 限制	1024

Top-K	N/A
Top-P	1
提示(Prompt)	以下 Python 代码报错： Traceback (most recent call last): File "/Users/leeboonstra/Documents/test_folder/rename_files.py", line

名称	**1_python_debug_code**

	7, in <module>
	text = toUpperCase(prefix)
	NameError: name 'toUpperCase' is not defined
	
	请调试此错误，并解释如何改进该代
	码。 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:

	new_filename = f"{text}_{file}" shutil.move(os.path.join(folder_name, file), os.path.join(folder_name, new_file_name)) # Print a success message print("Files renamed successfully.")

输出(Output)	代码中调用了 toUpperCase 函数将 prefix 转换为大写，但该函数未定义。为修复此问题，可以使用字符串类的 upper() 方法，它会将给定字符串转换为大写。修改后的代码如下所示：
	
	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()

名称	**1_python_debug_code**

	# 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:
	new_filename = f"{text}_{file}"
	shutil.move(os.path.join(folder_name, file),
	os.path.join(folder_name, new_filename))
	
	# Print a success message print("Files
	renamed successfully."
	
	此外，该代码还可以进行以下几项改进：
	
	1. 新文件名未保留原始文件扩展名。建议从原始文件名中提取扩展名并附加到新文件名中。
	
	2. 代码假设文件夹名中不含空格。建议优雅处理空格，例如用下划线或连字符替换。
	
	3. 使用 f 字符串语法进行字符串插值比传统的加号拼接更简洁易读。
	
	4. 代码未处理重命名过程中的异常。建议用 try...except 块包裹 shutil.move 调用以捕获潜在错误。
	
	以下是应用以上建议后的改进版代码：

	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)

名称	1_python_debug_code

	# Get a list of all files in the folder
	files = os.listdir(folder_name)
	
	# Iterate over the files
	for file in files:
	# 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.")

表 19：用于调试和审查 Python 代码的提示示例

太棒了。它不仅告诉我如何修复问题，还识别出我的代码中存在的其他错误，并给出了相应的解决方案。提示的最后部分还提供了改进代码的建议。

那多模态提示呢?

代码提示依然使用的是常规的大语言模型。而**多模态提**示是另一个独立的话题，它指的是一种技术，利用**多种输入形式**来引导大语言模型，而不仅仅依赖文本。这些输入

可以是文本、图像、音频、代码，甚至其他格式，具体取决于模型的能力和所执行的任务。

最佳实践（Best Practices）

找到合适的提示词需要不断试验。Vertex AI 中的 Language Studio 是一个理想的环境，可供你反复调整和测试提示，并验证在不同模型上的表现。要成为提示工程领域的专家，可以遵循以下最佳实践：**提供示例**

最重要的最佳实践之一是在提示中提供 **（One-shot / Few-shot）示例。**

这是非常有效的做法，因为它能作为强有力的教学工具。示例展示了期望的输出或类似响应，帮助模型从中学习，并相应地调整自己的生成结果。

这就像是给模型提供了一个参考点或目标，能提升其回答的**准确性、风格与语气**，使其更贴合你的预期。

设计应简洁明了

提示应当简洁、清晰、易于理解，不论是对你自己，还是对模型。

一个简单的经验法则是：**如果提示对你来说已经令人困惑，那对模型来说也一样。**

尽量避免使用复杂的语言，也不要提供无关信息。

示例：**修改前：**

> 我现在正在纽约旅行，我想了解有哪些好玩的地方。我带了两个 3 岁的孩子。我们假期期间该去哪里？

修改后：

> 充当旅游导游。请描述适合带 3 岁孩子游玩的纽约曼哈顿景点。

尝试使用描述动作的动词

以下是一组可用于提示中动词的示例：

Act, Analyze, Categorize, Classify, Contrast, Compare, Create, Describe, Define, Evaluate, Extract, Find, Generate, Identify, List, Measure, Organize, Parse, Pick, Predict, Provide, Rank, Recommend, Return, Retrieve, Rewrite, Select, Show, Sort,

Summarize, Translate, Write. 行动、分析、分类、归类、对比、比较、创作、描述、定义，评估、提取、查找、生成、识别、列出、测量、组织、解析、挑选，预测、提供、排序、推荐、返回、检索、重写、选择、展示、整理，总结、翻译、撰写。

明确期望的输出内容

要对期望的输出内容保持**具体明确**。一条简短的指令可能无法为 LLM 提供足够的引导，或者太过泛泛。

在提示中通过系统提示或上下文提示提供明确细节，可以帮助模型聚焦于相关内容，从而提升整体准确性。

示例：

☑ 推荐做法：

```
生成一篇包含 3 段的博客文章，主题为 “排名前五的视频游戏主机”。
文章应具有信息性和吸引力，使用对话式的写作风格。
```

❌ 不推荐做法：

```
生成一篇关于视频游戏主机的博客文章。
```

优先使用指令而非限制条件

在提示中，我们通常通过**指令（Instructions）和限制条件（Constraints）**来引导 LLM 的输出。

- **指令**：明确说明所需的格式、风格或内容，引导模型应执行的行为或生成的内容；
- **限制条件**：对输出设置限制，告诉模型应避免什么或不能做什么。

最新研究表明：**优先使用积极性的指令**，通常比依赖大量限制条件更有效。

这种方式也更符合人类偏好，我们通常更容易接受“该做什么”的指导，而不是一长串“不能做什么”的条目。

相比之下，**限制条件容易让模型产生困惑**，不确定哪些行为是允许的。而清晰的指令不仅传达明确意图，还能在设定范围内激发模型的创造力。

此外，多个限制条件之间还可能产生冲突。当然，在某些场景下，限制条件依然是必要的——比如防止模型生成有害或带有偏见的内容，或者要求严格的输出格式和风格时。

如果条件允许，请使用**积极性的指令**：不要告诉模型“不可以做什么”，而是告诉它“应该做什么”。这样可以减少歧义，提升输出的准确性。

示例：

☑ 推荐做法：

```
生成一段包含 1 段文字的博客文章，主题为“排名前五的视频游戏主机”。
只讨论主机名称、生产公司、上市年份和总销量。
```

❌ 不推荐做法：

```
生成一段包含 1 段文字的博客文章，主题为“排名前五的视频游戏主机”。
不要列出任何游戏名称。
```

最佳实践建议：

编写提示时优先使用清晰的**指令**，明确告诉模型你希望它“做什么”；

只有在涉及安全性、明确性或特定格式需求时，才加入必要的**限制条件**。

通过不断试验和迭代不同的指令与限制组合，找到最适合你具体任务的提示方式，并将其记录归档。

控制最大 Token 长度

要控制大语言模型生成响应的长度，可以在配置中设置**最大 token 限制**，也可以在提示中**明确指定长度要求**。例如：

```
请用推文长度解释量子物理。
```

在提示中使用变量

为了提高提示的复用性和动态性，可以在提示中使用**变量**，以便针对不同输入进行替换。例如如表 20 所示，提示用于提供有关城市的事实信息。与其在提示中硬编码城市名称，不如使用变量。

使用变量可以节省时间和精力，避免重复。如果你需要在多个提示中使用同一段信息，可以将其存储为变量，并在每个提示中引用。当你将提示集成到自己的应用程序中时，这种方式尤其合理。

提示(Prompt)	**变量** {city} = "Amsterdam" **提示**你是一名旅游导游。请告诉我一个关于这座城市的事实：{city}
输出(Output)	阿姆斯特丹是一座美丽的城市，遍布运河、桥梁和狭窄街道。它因其丰富的历史、文化和夜生活而成为热门旅游地。

表 20：在提示中使用变量

尝试不同的输入格式与写作风格

不同的模型、模型配置、提示格式、措辞方式以及提交形式，都会产生不同的结果。

因此建议尝试修改提示的属性，比如写作风格、词语选择和提示类型（zero-shot、few-shot、系统提示等）。

例如，目标是生成关于**革命性游戏主机 Sega Dreamcast** 的文本，可以使用不同形式的提示，产生不同风格的输出：

- **问题型提示**：

```
什么是 Sega Dreamcast？它为什么是一款革命性的游戏主机？
```

- **陈述型提示**：

```
Sega Dreamcast是 Sega 公司于 1999年发布的第六代视频游戏主机。它……
```

- **指令型提示**：

```
请写一段文字，描述 Sega Dreamcast主机，并解释它为何具有革命性。
```

在分类任务的 Few-shot 提示中混合类别

一般来说，few-shot 示例的顺序不会对模型产生太大影响。

但在**分类任务中**，应确保在 few-shot 示例中**混合不同类别的响应。**否则，模型可能会过拟合到示例的顺序，导致模型记住了示例的排列方式，而不是学习每一类的关键特征。

混合类别顺序有助于模型从内容本身进行识别，从而在面对新数据时表现得更加稳健、具有泛化能力。

推荐实践：

从 **6 个 few-shot 示例** 开始构建，逐步测试准确率。

适应模型更新

你需要及时了解模型架构的变化、新增的数据和能力。

尝试新版模型，并根据其新特性**调整你的提示**，以充分发挥模型能力。

像 Vertex AI Studio 这样的工具非常适合用于**保存、测试和记录提示的各个版本**。

尝试不同的输出格式

除了提示的输入格式外，也可以尝试调整**输出格式**。

对于非创意类任务，如提取、筛选、解析、排序、打分或分类数据等，建议将输出以**结构化格式**（如 JSON 或 XML）返回。

当提示生成的数据以 JSON 格式返回时，有以下优势：

- 在实际应用中，你不需要手动构建 JSON 格式，模型可以直接生成已排序的数据（这在处理日期时间对象时尤其实用）；
- 更重要的是，JSON 格式要求结构化输出，有助于**限制模型的幻觉内容**。

总结一下，使用 JSON 作为输出格式的好处包括：

- 输出始终保持统一风格
- 聚焦于你需要接收的数据
- 降低幻觉风险
- 支持识别数据之间的关系
- 提供明确的数据类型
- 可对数据进行排序

在前文“few-shot 提示”章节中的表 4 展示了如何生成结构化输出的示例。

JSON 修复

尽管使用 JSON 格式输出有诸多优势，但它也存在一些局限性。

JSON 结构虽然易于解析并适用于应用场景，但相比纯文本**需要更多的 token**，这会导致处理时间增加和成本上升。

此外，JSON 的**冗长性**容易占满整个输出窗口，特别是在生成内容因 token 限制被**截断**时更容易出问题。

这种截断通常会导致 JSON 不完整，缺失结尾的大括号或方括号，从而使输出变得不可用。

幸运的是，像 json-repair 这样的工具库（可通过 PyPI 安装）在这种场景下非常有用。

该库能**智能修复不完整或格式错误的 JSON 对象**，特别适合处理 LLM 生成的 JSON 输出中因截断而导致的问题，是一个非常重要的辅助工具。

使用 Schema 工作（Working with Schemas)

正如我们在本文中多次看到的，使用结构化 JSON 作为输出是一种非常有效的解决方案。但**输入呢?**

虽然 JSON 非常适合组织 LLM 的输出，但它同样也非常适用于组织你提供给模型的输入，这就涉及到 **JSON Schema（JSON 架构）** 的使用。

JSON Schema 用于定义 JSON 输入的数据结构和数据类型。通过提供一个 Schema，你就为 LLM 提供了一个清晰的“蓝图”，说明它应当接收哪些字段与格式，从而帮助模型将注意力聚焦在相关信息上，减少对输入内容的误解风险。

此外，Schema 还能帮助建立数据字段之间的关系，甚至通过添加日期或时间戳字段，让模型具备**时间感知能**力。

示例：

假设你希望使用 LLM 来为电商平台中的产品生成描述信息。与其只输入一段自由文本描述，不如使用 JSON Schema 明确定义产品的属性。

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

代码片段 5：结构化输出的 Schema 定义

然后，你可以提供符合上述 Schema 的产品数据作为输入：

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

代码片段 6：来自 LLM 的结构化输出

通过预处理数据，只提供 Schema 和对应数据，而非完整文档，你能让 LLM 清楚理解产品的各项属性（包括发布时间），这大大提高了生成准确且相关描述的概率。这种**结构化输入方式**，能够引导 LLM 将注意力集中在相关字段上，在处理大量数据或将 LLM 集成进复杂应用时，尤为有价值。

与其他提示工程师共同实验

如果你正处在一个需要构思优质提示的场景中，建议邀请多个同事或团队成员一起尝试。当每个人都遵循本章所列的最佳实践进行提示设计时，你将会看到不同提示尝试之间的效果差异。这种协同实验可以帮助你识别出最有效的提示策略。

CoT 提示的最佳实践（CoT Best Practices）

在使用 Chain of Thought（CoT）提示时，**必须将最终答案写在推理之后**，因为推理的生成过程会影响模型预测最终答案时所接收的 token 序列。

在使用 CoT 或自洽性（Self-consistency）方法时，**需要能够从提示中分离出最终答案**，使其与推理部分区分开来。

CoT 提示需将 Temperature 设置为 0

CoT 提示基于**贪婪解码（greedy decoding）**，即模型基于概率最高的 token 生成下
一个词。

通常在使用推理来得出最终答案时，**往往只有一个正确答案**，因此 temperature 应始终设置为 0，以确保结果的确定性。

记录每一次提示尝试

本章前文也提到过这个建议，但我们要再次强调其重要性：

请完整记录你的每一次提示尝试，这样你可以在之后不断总结经验，了解哪些做法有效，哪些无效。

模型输出可能因模型版本、采样设置，甚至是相同模型的不同提示请求而有所差异。

即使是相同的提示，在相同模型下重复运行，也可能因词汇概率相同而出现随机打断的情况，进而影响后续 token 的选择。

我们建议你使用 Google 表格，并参考表 21 的模板进行提示记录。

这种做法的好处是，当你日后回顾提示工程工作时可以拥有完整的历史记录——

无论是短暂中断后的重启、在不同模型版本之间测试提示效果，还是在调试出错时追溯来源。

除了表格中的字段，我们还建议你增加以下内容：

- 提示版本号（例如 V1、V2）
- 一个字段记录结果是否令人满意（OK / NOT OK / SOMETIMES OK）
- 一个字段记录反馈或备注

如果你使用 Vertex AI Studio，可以保存提示内容（使用与文档记录中一致的名称与版本），并将保存链接记录在表格中，以便一键调用提示重新测试。

如果你在开发 RAG（检索增强生成）系统，还应记录 RAG 系统中的各项内容配置，包括：

- 查询内容（Query）
- Chunk 设置与输出
- 插入提示的上下文等信息

一旦你认为提示已经足够完善，就可以将它加入你的项目代码库中。**提示内容应与代码文件分离保存**，以便于维护。

最终，**提示应成为可交付的系统组件。**作为提示工程师，你应该依赖自动化测试与评估流程来了解提示在实际任务中的泛化能力。

提示工程是一个**持续迭代的过程**：

编写并测试不同的提示，分析并记录结果，依据模型表现反复调整优化。

当你更换模型或修改模型配置时，也应重新使用旧提示进行测试，持续优化。

字段	**描述说明**
Name	[提示名称及版本]
Goal	[一句话说明本次尝试的目标]
Model	[所使用的模型名称及版本]
Temperature	·0 ·1 之间的数值]
Token Limit	·Token 上限数值]
Top-K	·Top-K 数值]
Top-P	·Top-P 数值]
Prompt	[完整的提示内容]
Output	[模型的输出结果（可以是多个）]

表 21：提示记录模板示例

总结

本白皮书探讨了提示工程的相关内容。我们学习了多种提示技术，包括：

- 零样本提示（Zero prompting）
- 少样本提示（Few shot prompting）
- 系统提示（System prompting）
- 角色提示（Role prompting）
- 上下文提示（Contextual prompting）
- 反向推理提示（Step-back prompting）
- 思维链提示（Chain of thought）
- 自洽性提示（Self consistency）
- 思维树提示（Tree of thoughts）
- ReAct 提示（Reason · Act）

我们还介绍了如何自动化生成提示的方式。

随后，白皮书讨论了生成式 AI 所面临的一些挑战，例如当提示设计不充分时可能出现的问题。最后，我们总结了一系列**成为优秀提示工程师的最佳实践**。