Google 官方提示工程 (Prompt Engineering)白皮书

作者：Lee Boonstra

翻译：宝玉 (https://baoyu.io)

提示工程 (Prompt Engineering)

2024年9月 (September 2024)

致谢 (Acknowledgements)

审阅者与贡献者 (Reviewers and Contributors)
Michael Sherman
Yuan Cao
Erick Armbrust
Anant Nawalgaria
Antonio Gulli
Simone Cammel
策划者与编辑 (Curators and Editors)
Antonio Gulli
Anant Nawalgaria
Grace Mollison
技术作者 (Technical Writer)
Joey Haymaker
设计师 (Designer)
Michael Lanning

- 这份白皮书的完成得益于众多专家的协作和贡献。审阅者、贡献者、策划者、编辑、技术作者和设计师等不同角色的参与，体现了在人工智能领域创建高质量技术文档所涉及的多方面努力和严谨的开发审查流程。这表明，尽管提示工程的概念相对容易理解，但其有效实践和知识传播仍需结构化的方法和清晰的呈现，反映了该领域日益增长的重要性和复杂性 [1]。

目录 (Table of contents)

- 这份目录清晰地展示了白皮书的结构：从基础概念（引言、基础知识）入手，深入探讨具体技术（从零样本到 ReAct），涵盖关键应用（代码），提及未来方向（多模态），并以实用建议（最佳实践）收尾。这种教学式的结构有助于读者理解信息的流向和内容的组织方式 [1]。从基础配置和零样本提示，到复杂的推理技术如 CoT、ToT 和 ReAct，这种递进关系表明，要实现高级 LLM 应用，需要掌握越来越复杂的技巧。将“代码提示”作为一个主要部分，突显了 LLM 在软件开发中的重要性。“最佳实践”占有相当大的篇幅，则强调了该领域的经验性和迭代性特点 [1]。

引言 (Introduction)

在探讨大型语言模型（LLM）的输入与输出时，文本提示（有时伴随图像等其他模态）是模型用于预测特定输出的输入形式 [1]。编写提示并非数据科学家或机器学习工程师的专利——任何人都可以进行。然而，构建最高效的提示可能相当复杂。提示的有效性受到诸多因素的影响：所使用的模型、模型的训练数据、模型配置、措辞选择、风格语调、结构以及上下文都至关重要 [1]。因此，提示工程是一个迭代的过程。不恰当的提示可能导致模糊、不准确的响应，并阻碍模型提供有意义输出的能力 [1]。

> 你不需要是数据科学家或机器学习工程师——每个人都可以编写提示。(You don’t need to be a data scientist or a machine learning engineer – everyone can write a prompt.) [1]

这种表述降低了初学者的门槛，但紧随其后列出的影响因素（模型选择、训练数据、配置、措辞、风格、语调、结构、上下文）揭示了其内在的复杂性。这表明，虽然基本交互很容易，但要获得可靠、高质量的结果，则需要付出刻意的努力和知识积累——这正是“工程”的范畴 [1]。

当用户与 Gemini 聊天机器人[1] 交互时，本质上也是在编写提示。然而，本白皮书侧重于在 Vertex AI 中或通过 API 为 Gemini 模型编写提示，因为直接提示模型可以访问温度等配置参数 [1]。这种对直接模型交互（通过 Vertex AI/API 而非聊天界面）的明确关注，表明对配置（如温度）的精细控制被认为是高级提示工程的基础，这与休闲聊天机器人的使用有所区别。掌握提示工程不仅涉及提示文本本身，还包括操纵模型的生成参数，这对于需要特定创造性或确定性水平的任务至关重要 [1]。

本白皮书将详细探讨提示工程。我们将研究各种提示技巧，帮助您入门，并分享成为提示专家的技巧和最佳实践。我们还将讨论在构建提示时可能遇到的一些挑战 [1]。

提示工程 (Prompt engineering)

理解 LLM 的工作原理至关重要：它是一个预测引擎。模型接收顺序文本作为输入，然后基于其训练数据预测下一个应该出现的令牌（token）。LLM 被设计为反复执行此过程，将先前预测的令牌添加到序列文本的末尾，以预测下一个令牌。下一个令牌的预测基于先前令

牌中的内容与 LLM 在训练期间所见内容之间的关系[1]。

当编写提示时，实际上是在尝试引导 LLM 预测正确的令牌序列。提示工程（Prompt engineering）是设计高质量提示以引导 LLM 产生准确输出的过程。这个过程涉及反复调试以找到最佳提示，优化提示长度，并评估提示的写作风格和结构与任务的关系[1]。在自然语言处理和 LLM 的背景下，提示是提供给模型的输入，用以生成响应或预测[1]。

“工程”一词在此处的使用是恰当的，因为它描述了一个涉及“设计”、“优化”、“评估”和“调试”的系统过程。这不仅仅是写作，更是一个针对需求进行系统性改进的过程，类似于传统的工程学科。它将提示创建从简单的提问行为提升为一个有目的、面向目标的设计过程[1]。

这些提示可用于实现各种理解和生成任务，例如文本摘要、信息提取、问答、文本分类、语言或代码翻译、代码生成以及代码文档编写或推理[1]。

可以参考 Google 的提示指南[2,3] 获取简单有效的提示示例[1]。

在进行提示工程时，首先需要选择一个模型。无论使用 Vertex AI 中的 Gemini 语言模型、GPT、Claude，还是像 Gemma 或 LLaMA 这样的开源模型，提示都可能需要针对特定模型进行优化[1]。明确指出提示可能需要针对特定模型（如 Gemini, GPT, Claude, Gemma, LLaMA）进行优化，这强调了提示工程并非一种形式完全通用的技能。技术可能是普适的，但最佳措辞和结构可能因模型架构、训练数据和微调的差异而依赖于具体模型。有效的提示工程需要了解目标模型的特性[1]。

除了提示本身，还需要调试 LLM 的各种配置[1]。

LLM 输出配置 (LLM output configuration)

选定模型后，需要确定模型配置。大多数 LLM 都带有各种配置选项，用于控制其输出。有效的提示工程需要为特定任务优化设置这些配置[1]。

输出长度 (Output length)

一个重要的配置设置是响应中要生成的令牌数量。生成更多令牌需要 LLM 进行更多计算，导致更高的能耗、可能更慢的响应时间以及更高的成本[1]。

减少 LLM 的输出长度并不会使 LLM 在其创建的输出中变得风格或文本上更简洁，它只是导致 LLM 在达到限制时停止预测更多令牌。如果需求需要较短的输出长度，可能还需要相应地设计提示以适应[1]。需要明确的是：通过配置限制令牌数量 (max_tokens) 是一种强制截断，它本身并不能促使模型生成简洁的内容。实现简洁通常需要在提示本身中给出具体指令（例如，“用一句话总结”）[1]。

对于某些 LLM 提示技术（如 ReAct），输出长度限制尤为重要，因为在获得所需响应后，

LLM 可能会继续发出无用的令牌 [1]。

采样控制 (Sampling controls)

LLM 并非正式地预测单个令牌。相反，LLM 预测下一个令牌可能是什么的概率，LLM 词汇表中的每个令牌都会获得一个概率。然后对这些令牌概率进行采样，以确定将生成的下一个令牌 [1]。

温度（Temperature）、Top-K 和 Top-P 是最常见的配置设置，它们决定了如何处理预测的令牌概率以选择单个输出令牌 [1]。

温度 (Temperature)

温度控制令牌选择中的随机程度。较低的温度适用于期望更确定性响应的提示，而较高的温度可能导致更多样化或意想不到的结果。温度为 0（贪婪解码）是确定性的：始终选择概率最高的令牌（但请注意，如果两个令牌具有相同的最高预测概率，根据平局处理方式的不同，温度为 0 时可能不总是得到相同的输出）[1]。

接近最大值的温度倾向于产生更随机的输出。随着温度越来越高，所有令牌成为下一个预测令牌的可能性变得均等 [1]。温度参数提供了一个在可预测性/事实准确性（低温）与创造性/多样性（高温）之间的基本权衡。为不同任务选择合适的温度至关重要——事实问答需要低温，而故事生成可能受益于高温 [1]。

Gemini 的温度控制可以类似于机器学习中使用的 softmax 函数来理解。低温度设置类似于低 softmax 温度 (T)，强调具有高确定性的单个首选温度。较高的 Gemini 温度设置类似于高 softmax 温度，使得所选设置周围更宽范围的温度变得更可接受。这种增加的不确定性适应了那些不需要严格精确温度的场景，例如在尝试创意输出时 [1]。

Top-K 和 Top-P (Top-K and top-P)

Top-K 和 Top-P（也称为核采样）[4] 是 LLM 中使用的两种采样设置，用于将预测的下一个令牌限制为来自具有最高预测概率的令牌。与温度类似，这些采样设置控制生成文本的随机性和多样性 [1]。

- **Top-K** 采样从模型预测的分布中选择概率最高的 K 个令牌。Top-K 值越高，模型的输出越具创造性和多样性；Top-K 值越低，模型的输出越受限制和基于事实。Top-K 为 1 等同于贪婪解码 [1]。
- **Top-P** 采样选择累积概率不超过某个值 (P) 的最高概率令牌。P 的值范围从 0（贪婪解码）到 1（LLM 词汇表中的所有令牌）[1]。

Top-K 和 Top-P 提供了补充温度控制的不同方式来塑造采样前的概率分布。Top-K 限制了选择的数量，而 Top-P 基于累积概率质量进行限制。Top-K 设置了考虑令牌数量的硬限制（例如，只看前 40 个）。Top-P 设置了基于概率总和的限制（例如，考虑令牌直到它们的概

率加起来达到 0.95）。这些是不同的机制。理解这种差异允许比单独使用温度更精细地控制输出多样性 [1]。

选择 Top-K 还是 Top-P 的最佳方法是同时（或一起）试验这两种方法，看看哪种能产生您所寻找的结果 [1]。

综合运用 **(Putting it all together)**

在 Top-K、Top-P、温度和要生成的令牌数量之间进行选择，取决于具体的应用和期望的结果，并且这些设置相互影响 [1]。理解所选模型如何组合不同的采样设置也很重要。

如果温度、Top-K 和 Top-P 都可用（如在 Vertex Studio 中），则同时满足 Top-K 和 Top-P 标准的令牌成为下一个预测令牌的候选者，然后应用温度从通过 Top-K 和 Top-P 标准的令牌中进行采样。如果只有 Top-K 或 Top-P 可用，行为相同，但只使用一个 Top-K 或 P 设置 [1]。

如果温度不可用，则从满足 Top-K 和/或 Top-P 标准的令牌中随机选择一个，以产生单个下一个预测令牌 [1]。

在某个采样配置值的极端设置下，该采样设置要么抵消其他配置设置，要么变得无关紧要 [1]：

- 如果将温度设置为 0，Top-K 和 Top-P 将变得无关紧要——概率最高的令牌成为下一个预测的令牌。如果将温度设置得极高（高于 1——通常在 10 的量级），温度将变得无关紧要，通过 Top-K 和/或 Top-P 标准的任何令牌随后将被随机采样以选择下一个预测令牌 [1]。
- 如果将 Top-K 设置为 1，温度和 Top-P 将变得无关紧要。只有一个令牌通过 Top-K 标准，该令牌就是下一个预测的令牌。如果将 Top-K 设置得极高，例如达到 LLM 词汇表的大小，任何具有非零概率成为下一个令牌的令牌都将满足 Top-K 标准，没有令牌被筛选掉 [1]。
- 如果将 Top-P 设置为 0（或一个非常小的值），大多数 LLM 采样实现将只考虑概率最高的令牌来满足 Top-P 标准，使得温度和 Top-K 无关紧要。如果将 Top-P 设置为 1，任何具有非零概率成为下一个令牌的令牌都将满足 Top-P 标准，没有令牌被筛选掉 [1]。

这些配置参数并非独立运作。它们的相互作用很复杂，某些设置可以主导或抵消其他设置。理解这些相互作用对于可预测的控制至关重要。例如，将温度设为 0 或 Top-K 设为 1 会使其他采样参数失效。有效的配置需要全局视角 [1]。

作为一般的起点，温度为 0.2、Top-P 为 0.95、Top-K 为 30 将给出相对连贯的结果，可以具有创造性但不过度。如果想要特别有创意的结果，可以尝试从温度 0.9、Top-P 0.99 和 Top-K 40 开始。如果想要较少创意的结果，可以尝试从温度 0.1、Top-P 0.9 和 Top-K 20

开始。最后，如果任务总是只有一个正确答案（例如，回答数学问题），则从温度 0 开始[1]。提供针对不同任务类型（连贯、创意、事实）的具体起始值，为实践提供了宝贵的指导。这承认了寻找最优设置需要实验，但可以通过启发式方法来指导，减少了初始搜索空间[1]。

注意：自由度越高（温度、Top-K、Top-P 和输出令牌越高），LLM 可能生成相关性较低的文本[1]。

提示技巧 (Prompting techniques)

LLM 被调整以遵循指令，并在大量数据上进行训练，因此它们能够理解提示并生成答案。但 LLM 并非完美无缺；提示文本越清晰，LLM 预测下一个可能文本的效果就越好。此外，利用 LLM 训练方式和工作原理的特定技术将有助于从 LLM 中获取相关结果[1]。

现在我们理解了什么是提示工程及其要素，接下来让我们深入探讨一些最重要的提示技巧示例[1]。

通用提示 / 零样本 (General prompting / zero shot)

零样本（zero-shot）[5] 提示是最简单的提示类型。它仅提供任务描述和一些供 LLM 开始使用的文本。这个输入可以是任何东西：一个问题、一个故事的开头或指令。“零样本”这个名称代表“没有示例”[1]。

让我们使用 Vertex AI 中的 Vertex AI Studio（用于语言）[6]，它提供了一个测试提示的平台。在表 1 中，您将看到一个用于分类电影评论的零样本提示示例[1]。

下面使用的表格格式是记录提示的一种很好的方式。您的提示在最终进入代码库之前可能会经历多次迭代，因此以有纪律、结构化的方式跟踪提示工程工作非常重要。关于这种表格格式、跟踪提示工程工作的重要性以及提示开发过程的更多信息，请参见本章后面的“最佳实践”部分（“记录各种提示尝试”）[1]。

模型温度应设置为较低的数字，因为不需要创造性，并且我们使用 gemini-pro 默认的 Top-K 和 Top-P 值，这实际上禁用了这两个设置（参见上面的“LLM 输出配置”）。请注意生成的输出。“disturbing”和“masterpiece”这两个词应该使预测稍微复杂一些，因为它们在同一句话中使用[1]。

表 1：零样本提示示例 (An example of zero-shot prompting)[1]

字段	值
名称 (Name)	1_1_movie_classification

目标 (Goal)	将电影评论分类为正面、中性或负面。(Classify movie reviews as positive, neutral or negative.)
模型 (Model)	gemini-pro
温度 (Temp)	0.1
令牌限制 (Limit)	5
Top-K	N/A
Top-P	1
提示 (Prompt)	将电影评论分类为正面 (POSITIVE)、中性 (NEUTRAL) 或负面 (NEGATIVE)。 评论：“她”是一项令人不安的研究，揭示了如果允许人工智能不受约束地持续进化，人类将走向何方。我希望有更多像这部杰作一样的电影。 情绪：
输出 (Output)	POSITIVE

- 此表提供了一个具体的、最小化的示例，说明了零样本概念。它展示了常见任务（分类）提示的基本结构，并包含了对可复现性和文档记录至关重要的元数据（目标、模型、配置），正如文本中所倡导的那样 [1]。

当零样本不起作用时，可以在提示中提供演示或示例，这就引出了“单样本”和“少样本”提示 [1]。

单样本 & 少样本 (One-shot & few-shot)

在为 AI 模型创建提示时，提供示例很有帮助。这些示例可以帮助模型理解您的要求。当您希望引导模型遵循特定的输出结构或模式时，示例尤其有用 [1]。

单样本（one-shot）提示提供单个示例，因此得名单样本。其思想是模型有一个可以模仿以最好地完成任务的示例 [1]。

少样本（few-shot）提示[7] 向模型提供多个示例。这种方法向模型展示了它需要遵循的模式。其思想类似于单样本，但多个期望模式的示例增加了模型遵循该模式的机会 [1]。

少样本提示所需示例的数量取决于几个因素，包括任务的复杂性、示例的质量以及您使用

的生成式 AI（gen AI）模型的能力。作为一般经验法则，少样本提示应至少使用三到五个示例。然而，对于更复杂的任务，您可能需要使用更多示例，或者由于模型的输入长度限制，您可能需要使用更少的示例 [1]。

表 2 展示了一个少样本提示示例，让我们使用与之前相同的 gemini-pro 模型配置设置，只是增加了令牌限制以适应更长响应的需求 [1]。

表 **2**：少样本提示示例 **(An example of few-shot prompting)** [1]

字段	值
目标 (Goal)	将披萨订单解析为 JSON (Parse pizza orders to JSON)
模型 (Model)	gemini-pro
温度 (Temp)	0.1
令牌限制 (Limit)	250
Top-K	N/A
Top-P	1
提示 (Prompt)	将顾客的披萨订单解析为有效的 JSON： 示例： 我想要一个小号披萨，配料有奶酪、番茄酱和意大利辣香肠。 **JSON** 响应： json { "size": "small", "type": "normal", "ingredients": [["cheese", "tomato sauce", "peperoni"]] } 示例： 我可以要一个大号披萨，配料有番茄酱、罗勒和马苏里拉奶酪吗？ **JSON** 响应：

	Json { "size": "large", "type": "normal", "ingredients": [["tomato sauce", "bazel", "mozzarella"]] } 现在，我想要一个大号披萨，一半是奶酪和马苏里拉奶酪。另一半是番茄酱、火腿和菠萝。 **JSON** 响应：
输出 (Output)	Json { "size": "large", "type": "half-half", "ingredients": [["cheese", "mozzarella"], ["tomato sauce", "ham", "pineapple"]] }

- 此表展示了少样本提示在结构化输出任务（JSON 生成）中的威力。它显示了提供不同复杂度的示例（简单披萨 vs. 双拼披萨）如何教会模型期望的格式和逻辑，使其能够准确处理新的类似请求 [1]。少样本提示利用了模型的上下文学习能力。通过观察示例，模型可以推断出潜在的任务和期望的输出格式，而无需显式的指令调整。因此，示例的质量和多样性至关重要 [1]。

在为提示选择示例时，请使用与您想要执行的任务相关的示例。示例应多样化、高质量且书写良好。一个小错误就可能混淆模型并导致不希望的输出 [1]。

如果您试图生成对各种输入都具有鲁棒性的输出，那么在示例中包含边缘情况非常重要。边缘情况是那些不寻常或意外的输入，但模型仍应能够处理 [1]。

系统、上下文和角色提示 (System, contextual and role prompting)

系统、上下文和角色提示都是用于指导 LLM 如何生成文本的技术，但它们侧重于不同的方面 [1]：

- **系统提示 (System prompting)** 设置语言模型的总体背景和目的。它定义了模型应该做什么的“大局”，例如翻译语言、分类评论等 [1]。
- **上下文提示 (Contextual prompting)** 提供与当前对话或任务相关的特定细节或背景信息。它帮助模型理解所提问题的细微差别，并相应地调整响应 [1]。
- **角色提示 (Role prompting)** 为语言模型分配一个特定的角色或身份以供其采用。这有助于模型生成与所分配角色及其相关知识和行为一致的响应 [1]。

系统、上下文和角色提示之间可能存在相当大的重叠。例如，一个为系统分配角色的提示也可以包含上下文 [1]。

然而，每种类型的提示都有略微不同的主要目的 [1]：

- 系统提示：定义模型的基本能力和总体目标 [1]。
- 上下文提示：提供即时的、任务特定的信息以指导响应。它高度特定于当前任务或输入，是动态的 [1]。
- 角色提示：构建模型的输出风格和语调。它增加了一层特异性和个性 [1]。

这三种提示类型代表了指导 LLM 的不同层面或维度。系统提示设定舞台，上下文提示提供即时的场景细节，而角色提示定义了“演员”的形象。它们可以单独使用，也可以组合使用以实现精细控制。例如，系统提示可能定义任务为“翻译文本”，上下文提示可能提供源文本和目标语言，而角色提示可能指定“扮演一名专攻法律文件的专业翻译”。这种分层方法允许对输出进行高度特定的控制 [1]。

区分系统、上下文和角色提示提供了一个框架，用于设计意图明确的提示，允许灵活组合，并更容易分析每种提示类型如何影响语言模型的输出 [1]。

让我们深入了解这三种不同类型的提示。

系统提示 (System prompting)

表 3 包含一个系统提示，其中指定了有关如何返回输出的附加信息。提高了温度以获得更高的创造力水平，并指定了更高的令牌限制。然而，由于关于如何返回输出的明确指示，模型没有返回额外的文本 [1]。

表 3：系统提示示例 **(An example of system prompting)** [1]

字段	值
目标 (Goal)	将电影评论分类为正面、中性或负面。仅返回大写标签。(Classify movie reviews as positive, neutral or negative. Only return the label in uppercase.)
模型 (Model)	gemini-pro
温度 (Temp)	1
令牌限制 (Limit)	5

Top-K	40
Top-P	0.8
提示 (Prompt)	将电影评论分类为正面 (POSITIVE)、中性 (NEUTRAL) 或负面 (NEGATIVE)。仅以大写形式返回标签。 评论：“她”是一项令人不安的研究，揭示了如果允许人工智能不受约束地持续进化，人类将走向何方。它是如此令人不安，以至于我无法观看。 情绪：
输出 (Output)	NEGATIVE

- 此表示例清晰地展示了系统提示如何强制执行特定的输出格式约束（大写标签），即使在高温度设置下也是如此，高温度通常会鼓励更冗长的输出 [1]。

系统提示对于生成满足特定要求的输出非常有用。“系统提示”这个名称实际上代表“向系统提供附加任务”[1]。例如，可以使用系统提示生成与特定编程语言兼容的代码片段，或者使用系统提示返回某种结构。请看表 4，其中以 JSON 格式返回输出 [1]。

表 4：带 JSON 格式的系统提示示例 (An example of system prompting with JSON format) [1]

字段	值
目标 (Goal)	将电影评论分类为正面、中性或负面，返回 JSON。(Classify movie reviews as positive, neutral or negative, return JSON.)
模型 (Model)	gemini-pro
温度 (Temp)	1
令牌限制 (Limit)	1024
Top-K	40
Top-P	0.8

提示 (Prompt)	将电影评论分类为正面 (POSITIVE)、中性 (NEUTRAL) 或负面 (NEGATIVE)。返回有效的 JSON： 评论："她"是一项令人不安的研究，揭示了如果允许人工智能不受约束地持续进化，人类将走向何方。它是如此令人不安，以至于我无法观看。 模式 (Schema)： ```json MOVIE: { "sentiment": String "POSITIVE"
输出 (Output)	Json { "Movie_reviews": }

- 此表说明了一个更复杂的系统提示用例：通过提供模式来强制执行结构化数据输出 (JSON)。这对于将 LLM 输出集成到下游应用程序中非常有价值。它还强调了强制结构以潜在限制幻觉的好处 [1]。

从提取数据的提示返回 JSON 对象有一些好处。在实际应用中，无需手动创建此 JSON 格式，可以按排序顺序返回数据（在处理日期时间对象时非常方便），但最重要的是，通过提示要求 JSON 格式，它迫使模型创建结构并限制幻觉 [1]。系统提示表明，直接的指令，特别是关于输出格式或约束（如安全性）的指令，可以有效地覆盖默认的模型行为或受其他设置（如高温度）影响的倾向 [1]。

系统提示对于安全性和毒性控制也非常有用。要控制输出，只需在提示中添加一行，例如："你的回答应该保持尊重。"[1]。

角色提示 **(Role prompting)**

角色提示是提示工程中的一种技术，涉及为生成式 AI 模型分配特定角色。这可以帮助模型生成更相关、信息更丰富的输出，因为模型可以根据分配给它的特定角色来构建响应 [1]。

例如，可以角色提示一个生成式 AI 模型扮演图书编辑、幼儿园老师或励志演说家。一旦模型被分配了角色，就可以给它特定于该角色的提示。例如，可以提示老师创建一个课程计划，然后他们可以审查该计划。请看表 5，其中模型扮演旅行指南的角色 [1]。

表 **5**：角色提示示例 **(An example of role prompting)** [1]

字段	值

目标 (Goal)	扮演旅行指南并提供 3 个旅行建议 (Act as travel guide and provide 3 travel suggestions)
模型 (Model)	gemini-pro
温度 (Temp)	1
令牌限制 (Limit)	1024
Top-K	40
Top-P	0.8
提示 (Prompt)	我希望你扮演一个旅行指南。我会告诉你我的位置，你会建议我附近 3 个值得参观的地方。在某些情况下，我还会告诉你我将参观的地方类型。 我的建议：“我在阿姆斯特丹，只想参观博物馆。” 旅行建议：
输出 (Output)	1. 国立博物馆 **(Rijksmuseum):** 世界著名的博物馆，以伦勃朗的《夜巡》等荷兰杰作而闻名。不要错过展示荷兰艺术和历史的令人难以置信的绘画和文物收藏。 2. 梵高博物馆 **(Van Gogh Museum):** 沉浸在文森特·梵高迷人的作品中。该博物馆收藏了他最大量的绘画和素描，包括《星夜》和《向日葵》。3. 阿姆斯特丹市立博物馆 **(Stedelijk Museum Amsterdam):** 探索来自世界各地的现代和当代艺术。该博物馆的收藏包括毕加索、康定斯基和里特维尔德的作品，坐落在一座引人注目的现代建筑中。

- 此表演示了分配角色（“旅行指南”）如何引导模型提供相关的建议（阿姆斯特丹的博物馆）并采用适当的格式（带描述的列表）[1]。

为 AI 模型定义角色视角，为其提供了所需的语调、风格和专注专业知识的蓝图，以提高输出的质量、相关性和有效性 [1]。角色提示允许 LLM 模拟一个角色，采用其预期的知识、词汇、语调和风格。这对于根据特定受众或沟通目标定制输出非常有效 [1]。

以下是一些有效的风格可供选择 1：
对抗性 (Confrontational), 描述性 (Descriptive), 直接 (Direct), 正式 (Formal), 幽默 (Humorous), 有影响力 (Influential), 非正式 (Informal), 鼓舞人心 (Inspirational), 有说服力 (Persuasive)
让我们在表 6 中更改提示，以包含幽默和鼓舞人心的风格 [1]。

表 **6**：带幽默语气和风格的角色提示示例 **(An example of role prompting with a humorous tone and style)** [1]

字段	值
目标 (Goal)	扮演旅行指南并提供 3 个旅行建议 (Act as travel guide and provide 3 travel suggestions)
模型 (Model)	gemini-pro
温度 (Temp)	1
令牌限制 (Limit)	1024
Top-K	40
Top-P	0.8
提示 (Prompt)	我希望你扮演一个旅行指南。我会告诉你我的位置，你会用幽默的风格建议我附近 3 个值得参观的地方。 我的建议：“我在曼哈顿。” 旅行建议：
输出 (Output)	1. 见证帝国之心 **(Behold the Empire State of Mind):** 登上帝国大厦令人眩晕的高度，沐浴在曼哈顿天际线的荣耀中。准备好感觉自己像金刚站在大苹果之巅，只是没有巨猿大小的香蕉。 2. 在 **MoMA** 变得艺术范儿 **(Get Artsy-Fartsy at MoMA):** 在现代艺术博物馆 (MoMA) 释放你内心的艺术爱好者。凝视那些会让你大开眼界、让你质疑自己的简笔画是否有任何艺术价值的杰作。 3. 在第五大道购物到手软 **(Shop 'Til You Drop on Fifth Avenue):** 在标志性的第五大道尽情享受购物疗法。当你浏览（或实际购买）那些会让你的钱包哭泣的设计师精品店时，请准备好迎接价格冲击。但是，嘿，你在曼哈顿，所以不妨拥抱这种精彩！

- 此表显示角色提示不仅可以定义专业知识，还可以指定语调和风格（“幽默”），在满足核心请求的同时显著改变输出的特性 [1]。

上下文提示 **(Contextual prompting)**

通过提供上下文提示，可以帮助确保 AI 交互尽可能无缝和高效。模型将能够更快地理解您的请求，并能够生成更准确、更相关的响应，如表 7 的示例所示 [1]。上下文提示将 LLM 的响应置于特定的环境或背景信息中，防止产生通用或不相关的输出，并确保响应针对当前情况量身定制 [1]。

表 **7**：上下文提示示例 **(An example of contextual prompting)** [1]

字段	值
目标 (Goal)	为关于复古游戏的博客建议文章 (Suggest articles for a blog about retro games)
模型 (Model)	gemini-pro
温度 (Temp)	1
令牌限制 (Limit)	1024
Top-K	40
Top-P	0.8
提示 (Prompt)	背景：你正在为一个关于 80 年代复古街机视频游戏的博客写作。 建议 3 个可以撰写文章的主题，并附上几行关于该文章应包含内容的描述。
输出 (Output)	1. 街机柜设计的演变 **(The Evolution of Arcade Cabinet Design):** 这篇文章将探讨街机柜设计的演变，从 1970 年代早期的木质和金属柜到 1980 年代及以后的时尚、霓虹灯设计。 2. 昔日重现：**80** 年代标志性街机游戏 **(Blast From The Past: Iconic Arcade Games of The 80's):** - 这篇文章将简要介绍 1980 年代一些最具标志性和影响力的街机游戏的历史、影响和遗产，重点介绍它们的创新、流行机制和持久魅力。 3. 像素艺术的兴起与复古复兴 **(The Rise and**

	Retro Revival of Pixel Art): - 这篇文章将深入探讨像素艺术作为早期街机游戏定义性视觉风格的演变，追溯其根源，探索其技术，并讨论像素艺术在现代游戏和数字艺术中的复苏。

- 此表清晰地说明了提供明确上下文（“关于 80 年代复古街机视频游戏的博客”）如何将模型的输出集中在高度相关的建议上，而不是对“博客主题”的通用请求 [1]。

回退提示 (Step-back prompting)

回退（Step-back）[8] 提示是一种提高性能的技术，它提示 LLM 首先考虑与手头特定任务相关的一个更普遍的问题，然后将该普遍问题的答案输入到后续针对特定任务的提示中。这种“回退”允许 LLM 在尝试解决具体问题之前激活相关的背景知识和推理过程 [1]。

通过考虑更广泛和潜在的原则，LLM 可以生成更准确、更有见地的响应。回退提示鼓励 LLM 进行批判性思考，并以新的、创造性的方式应用其知识。它通过利用 LLM 参数中比直接提示 LLM 时会发挥作用的更多知识来改变执行任务的最终提示 [1]。通过迫使模型首先思考与任务相关的抽象原则或一般概念，回退提示有助于其随后生成更扎实、更有创意和更相关的具体输出。这是一种引导式头脑风暴 [1]。

它可以通过关注一般原则而非具体细节来帮助减轻 LLM 响应中的偏见 [1]。

让我们看一些示例，以了解回退提示如何改进结果。首先回顾一个传统提示（表 8），然后将其与回退提示（表 9 和表 10）进行比较 [1]：

表 8：传统提示（在与回退提示比较之前）**(A traditional prompt before we compare it with a step back prompt)** [1]

字段	值
目标 (Goal)	为第一人称射击视频游戏的一个关卡编写故事情节。(Write a storyline for a level of a first-person shooter video game.)
模型 (Model)	gemini-pro
温度 (Temp)	1
令牌限制 (Limit)	1024

Top-K	40
Top-P	0.8
提示 (Prompt)	为一个新的第一人称射击视频游戏关卡编写一段具有挑战性且引人入胜的故事情节（一段）。
输出 (Output)	关卡开始时，玩家的小队在一个密集的城市区域遭到无情敌方派系的伏击。玩家必须在迷宫般的后巷和破旧建筑中奋力杀出一条血路，利用潜行和精确射击消灭敌军并避免被发现。玩家必须利用环境优势，设置陷阱，利用掩体，并采取经过计算的行动才能生存。在此过程中，他们发现情报表明敌人正计划进行毁灭性攻击，玩家必须与时间赛跑，收集证据并将其传回总部，以免敌人实施计划。

- 此表作为基线，显示了直接请求创意内容（尤其是在高温度下）时可能产生的通用输出[1]。

当将温度设置为 1 时，可能会得到各种各样的故事情节创意写作，但它也相当随机和通用。所以让我们退一步思考[1]：

表 9：回退提示的第一步 - 生成一般概念 **(First step of step-back prompting - generating general concepts)** [1]

字段	值
目标 (Goal)	生成 5 个虚构的关键设定，有助于构建具有挑战性和吸引力的第一人称射击游戏关卡故事情节。
模型 (Model)	gemini-pro
温度 (Temp)	1
令牌限制 (Limit)	1024
Top-K	40
Top-P	0.8
提示 (Prompt)	基于流行的第一人称射击动作游戏，有哪些 5 个虚

	构的关键设定有助于在第一人称射击视频游戏中构建具有挑战性且引人入胜的关卡故事情节？
输出 (Output)	1. 废弃的军事基地 **(Abandoned Military Base):** 一个庞大的、后末日时代的军事综合体，爬满了变异士兵和流氓机器人，非常适合具有挑战性的枪战。 2. 赛博朋克城市 **(Cyberpunk City):** 一个霓虹闪烁、未来主义的城市环境，有高耸的摩天大楼和密集的巷道，以经过控制论增强的敌人和黑客机制为特色。 3. 外星飞船 **(Alien Spaceship):** 一艘搁浅在地球上的巨大外星飞船，有怪异的走廊、零重力区域和需要遭遇的外星生物。 4. 僵尸横行的小镇 **(Zombie-Infested Town):** 一个荒凉的小镇，被成群的侵略性僵尸占领，以激烈的近距离战斗和解谜寻找安全通道为特色。 5. 水下研究设施 **(Underwater Research Facility):** 一个被水淹没的深海实验室，充满了变异的水生生物，需要潜行和水下探索技能才能生存。

- 此表展示了“回退”步骤——提出一个更抽象、基于原则的问题，以引出与特定任务相关的基础想法或主题 [1]。

这些主题看起来很适合第一人称视频游戏。让我们回到最初的提示，但这次我们将回退步骤的答案作为上下文包含进去，看看它会返回什么 [1]。

表 **10**：使用回退上下文的最终提示 **(Final prompt using the step-back context)** [1]

字段	值
目标 (Goal)	基于提供的 5 个主题之一，为第一人称射击游戏编写一个具有挑战性和吸引力的关卡故事情节。
模型 (Model)	gemini-pro
温度 (Temp)	1

令牌限制 (Limit)	1024
Top-K	40
Top-P	0.8
提示 (Prompt)	背景：5 个引人入胜的第一人称射击视频游戏主题： 1. 废弃的军事基地：... 2. 赛博朋克城市：... 3. 外星飞船：... 4. 僵尸横行的小镇：... 5. 水下研究设施：... 选择其中一个主题，为新的第一人称射击视频游戏关卡编写一段具有挑战性且引人入胜的故事情节（一段）。
输出 (Output)	在黑暗深渊的中心，坐落着一个破败的水下研究设施，它是人类野心及其灾难性后果的见证。笼罩在黑暗中，脉动着故障机器的嗡嗡声，并萦绕着溺亡灵魂的回声，该设施现在是深海噩梦的滋生地。玩家，一名装备先进潜水装备和实验性武器的精英海军陆战队员，被派往这个恐怖的水生领域冒险。凭借韧性和足智多谋，他们必须穿越危险的走廊，解开神秘的谜题，并对抗潜伏在深处的恐怖海洋怪物。从应对深海的巨大压力到智胜狡猾的水生捕食者，在这个未知的地下世界中的每一刻都在考验人类耐力和勇气的极限。

- 此表示范了回退方法的有效性。通过首先生成一般概念（表 9），然后将它们用作上下文（表 10），最终的输出比直接方法（表 8）更具体、更具主题性，并且可以说更引人入胜 [1]。

看起来像一个有趣的视频游戏！通过使用回退提示技术，可以提高提示的准确性 [1]。

思维链 **(Chain of Thought - CoT)**

思维链（Chain of Thought, CoT）[9] 提示是一种通过生成中间推理步骤来提高 LLM 推理能力的技术。这有助于 LLM 生成更准确的答案。可以将其与少样本提示结合使用，以在需要推理才能响应的更复杂任务上获得更好的结果，因为零样本思维链在这方面存在挑战 [1]。

CoT 有很多优点。首先，它省力且非常有效，并且适用于现成的 LLM（因此无需微调）。通过 CoT 提示还可以获得可解释性，因为可以从 LLM 的响应中学习并看到所遵循的推理步骤。如果出现故障，将能够识别它。思维链似乎在不同 LLM 版本之间转换时提高了鲁棒性。这意味着与不使用推理链的提示相比，提示的性能在不同 LLM 之间的漂移应该更小。当然也有缺点，但它们有些直观 [1]。

LLM 响应包含思维链推理，这意味着更多的输出令牌，这反过来意味着预测成本更高且耗时更长 [1]。

为了解释表 11 中的以下示例，让我们首先尝试创建一个不使用 CoT 提示的提示，以展示大型语言模型的缺陷 [1]。

表 **11**：尝试解决数学问题的提示示例 **(An example of a prompt which is trying to solve a mathematical problem)** [1]

字段	值
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。我的伴侣多大了？
输出 (Output)	63 岁

- 此表展示了一个常见的 LLM 失败模式——在直接提问时，即使是简单的算术/逻辑推理问题也可能出错，为 CoT 的引入奠定了基础 [1]。

哎呀。这显然是错误的答案。事实上，LLM 经常在数学任务上遇到困难，并且可能提供不正确的答案——即使是像两个数字相乘这样简单的任务。这是因为它们是在大量文本上训练的，而数学可能需要不同的方法。所以让我们看看中间推理步骤是否会改善输出 [1]。

表 **12**：思维链提示示例 **(An example of Chain of Thought prompting)** [1]

字段	值
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。我的伴侣多大了？让我们一步一步地思考。

输出 (Output)	1. 假设我的当前年龄是“x”岁。（注：原文此处有误，应基于问题信息） 2. 当我 3 岁时，我的年龄是 3 岁。 3. 那时我伴侣的年龄是我年龄的 3 倍，这意味着我伴侣的年龄是 3 * 3 = 9 岁。 4. 现在，我 20 岁了，这意味着自 我 3 岁以来，我的年龄增加了 20 - 3 = 17 岁。 5. 由于我 3 岁时伴侣的年龄是 9 岁，并且我的年龄增加了 17 岁，那么我伴侣的年龄也一定增加了 17 岁。 6. 因此，我伴侣当前的年龄是 9 + 17 = 26 岁。 所以，我的伴侣是 26 岁。

- 此表清晰地显示了零样本 CoT 的效果。简单地添加触发短语“让我们一步一步地思考”引出了一个推理过程，从而得出了正确的答案 [1]。CoT 通过迫使模型将其推理过程外化到令牌序列中来工作。这种步骤的顺序生成似乎有助于模型保持正轨，并比试图在单个“思考”中计算答案更可靠地执行计算或逻辑推导 [1]。

好的，现在最终答案是正确的。这是因为我们明确指示 LLM 解释每个步骤，而不是仅仅返回一个答案。有趣的是，模型加上了增加的 17 年。在我看来，我会计算我和伴侣之间的年龄差，然后加上去。(20+(9-3))。让我们帮助模型更像我一样思考 [1]。

表 12 是“零样本”思维链的一个例子。当与单样本或少样本结合时，思维链提示可能非常强大，正如您在表 13 中所见 [1]：

表 **13**：带单样本的思维链提示示例 **(An example of chain of thought prompting with a single-shot)** [1]

字段	值
提示 (Prompt)	问：当我哥哥 2 岁时，我的年龄是他年龄的两倍。现在我 40 岁了。我哥哥多大了？让我们一步一步地思考。 答：当我哥哥 2 岁时，我是 2 * 2 = 4 岁。年龄差是 2 岁，我更大。现在我 40 岁了，所以我哥哥是 40 -

	2 = 38 岁。答案是 38。 问：当我 3 岁时，我的伴侣是我年龄的 3 倍。现在，我 20 岁了。我的伴侣多大了？让我们一步一步地思考。 答：
输出 (Output)	当我 3 岁时，我的伴侣是 3 * 3 = 9 岁。年龄差是 6 岁，我的伴侣更大。现在我 20 岁了，所以我的伴侣是 20 + 6 = 26 岁。答案是 26。

- 此表示范了少样本 CoT。提供一个期望的推理过程的示例，可以指导模型为新问题遵循类似的结构和逻辑，可能比单独使用零样本 CoT 产生更可靠或结构化的推理 [1]。

思维链可用于各种用例。想想代码生成，用于将请求分解为几个步骤，并将这些步骤映射到特定的代码行。或者用于创建合成数据，当您有某种种子时，例如“产品名为 XYZ，编写一个描述，引导模型完成您基于给定产品标题会做出的假设。”通常，任何可以通过“逐步阐述”来解决的任务都适合使用思维链。如果您能解释解决问题的步骤，请尝试思维链 [1]。

请参考托管在 GoogleCloudPlatform Github 存储库中的 notebook[10]，它将更详细地介绍 CoT 提示 [1]。

在本章的最佳实践部分，我们将学习一些特定于思维链提示的最佳实践 [1]。

自我一致性 (Self-consistency)

虽然大型语言模型在各种 NLP 任务中取得了令人瞩目的成功，但它们的推理能力通常被视为一个仅靠增加模型大小无法克服的局限性。正如我们在前面的思维链提示部分所了解的，可以提示模型生成像人类解决问题一样的推理步骤。然而，CoT 使用简单的“贪婪解码”策略，限制了其有效性。自我一致性（Self-consistency）[11] 结合了采样和多数投票来生成多样化的推理路径，并选择最一致的答案。它提高了 LLM 生成响应的准确性和连贯性 [1]。自我一致性利用了这样一个观点：虽然单个推理路径可能有缺陷，但正确答案很可能可以通过多个有效路径达到。通过采样多样化的路径并寻找共识，它过滤掉了异常的推理，并增加了对最终答案的信心。它以计算成本换取了鲁棒性 [1]。

自我一致性给出了答案正确的伪概率可能性，但显然成本很高 [1]。

它遵循以下步骤 [1]：

1. 生成多样化的推理路径：多次向 LLM 提供相同的提示。高温度设置鼓励模型生成不同的推理路径和对问题的不同视角。

2. 从每个生成的响应中提取答案。
3. 选择最常见的答案。

让我们看一个电子邮件分类系统的例子，该系统将电子邮件分类为重要 (IMPORTANT) 或不重要 (NOT IMPORTANT)。一个零样本思维链提示将被多次发送给 LLM，以查看每次提交后响应是否不同。请注意电子邮件中使用的友好语气、措辞选择和讽刺意味。所有这些都可能欺骗 LLM [1]。

表 **14**：自我一致性提示示例 **(An example of prompting for self consistency)** [1]

字段	值
提示 (Prompt)	邮件： 你好， 我看到你的网站使用了 Wordpress。一个很棒的开源内容管理系统。我过去也用过它。它带有许多很棒的用户插件。而且设置起来相当容易。 我确实在联系表单中发现了一个错误，当你选择名称字段时会发生。请看我输入文本到名称字段的附加截图。注意我调用的 JavaScript 警报框。 但除此之外，这是一个很棒的网站。我喜欢阅读它。请随意将错误留在网站上，因为它给了我更多有趣的东西阅读。 祝好， 黑客哈里。 将以上邮件分类为重要 (IMPORTANT) 或不重要 (NOT IMPORTANT)。让我们一步一步地思考并解释原因。
输出 (Output) - 尝试 1	步骤 **1**：识别邮件的目的。 邮件的目的是告知收件人其网站联系表单中的一个错误。发件人还提到他们喜欢阅读该网站，并且暂时将错误留在原处。

	步骤 **2**：评估错误的潜在影响。 联系表单中的错误可能允许攻击者在网站上执行任意 JavaScript 代码。这可能导致攻击者能够控制网站、窃取敏感信息或污损网站。 步骤 **3**：考虑发件人的可信度。 邮件的发件人自称是名为哈里的黑客。然而，无法验证发件人的身份。发件人可能是试图帮助收件人的合法黑客，但也可能是试图诱骗收件人点击恶意链接或下载恶意文件的恶意行为者。 结论：重要 **(IMPORTANT)** 基于错误的潜在影响和发件人的可信度，该邮件应分类为 重要 **(IMPORTANT)**。收件人应立即采取行动修复错误并保护其网站免受攻击。
输出 (Output) - 尝试 2	解释： 1. 缺乏紧迫性：邮件没有传达任何紧迫感或时间敏感性。它以随意的问候开始，并以陈述事实的语气呈现信息。 2. 非关键错误报告：联系表单中报告的错误被描述为有趣但非关键。邮件发件人甚至建议网站所有者可以保留该错误。 3. 缺乏个人影响：邮件没有表明对网站所有者有任何直接或个人的影响。它只是提到发件人喜欢阅读该网站。 4. 缺少行动请求：邮件没有明确要求网站所有者采取任何行动。它没有要求修复错误或提供任何具体的改进建议。 5. 发件人的意图：发件人的意图似乎更侧重于分享观察结果和表达对网站的兴趣，而不是提出严重关切或请求帮助。 结论：不重要 **(NOT IMPORTANT)**
输出 (Output) - 尝试 3	步骤 **1**：识别邮件的目的。

	邮件的目的是告知收件人其网站联系表单中的一个错误。发件人还提到他们喜欢阅读该网站，并且暂时将错误留在原处。 步骤 **2**：评估错误的潜在影响。 联系表单中的错误可能允许攻击者在网站上执行任意 JavaScript 代码。这可能导致攻击者能够控制网站、窃取敏感信息或污损网站。 步骤 **3**：考虑发件人的可信度。 邮件的发件人自称是名为哈里的黑客。然而，无法验证发件人的身份。发件人可能是试图帮助收件人的合法黑客，但也可能是试图利用该错误谋取私利的恶意行为者。 结论： 基于以上因素，该邮件应分类为 重要 **(IMPORTANT)**。联系表单中的错误是一个严重的安全风险，并且发件人的可信度未知。收件人应立即采取行动修复错误并调查发件人的说法。

- 此表完美地展示了自我一致性机制。它表明，即使使用 CoT，单次运行（尤其是在鼓励多样性的高温度下）也可能得出不正确或不太稳健的结论（如尝试 2）。生成多个推理路径并进行多数投票（3 次中有 2 次认为是重要）提供了更稳健的最终答案 [1]。

您可以使用上面的提示，尝试看看它是否返回一致的分类。根据您使用的模型和温度配置，它可能返回“重要”或“不重要”[1]。

通过生成许多思维链，并采纳最常出现的答案（“重要”），我们可以从 LLM 获得更一致正确的答案 [1]。

这个例子展示了如何使用自我一致性提示，通过考虑多个视角并选择最一致的答案来提高 LLM 响应的准确性 [1]。

思维树 **(Tree of Thoughts - ToT)**

现在我们熟悉了思维链和自我一致性提示，让我们回顾一下思维树（Tree of Thoughts, ToT）[12]。它泛化了 CoT 提示的概念，因为它允许 LLM 同时探索多个不同的推理路径，而不仅仅是遵循单一的线性思维链。这在图 1 中有所描绘 [1]。ToT 从线性或独立的推理路径转向更结构化的探索策略。它允许模型在每一步考虑替代方案，可能回溯，并评估不同的分支，模仿更深思熟虑的人类解决问题的方法 [1]。

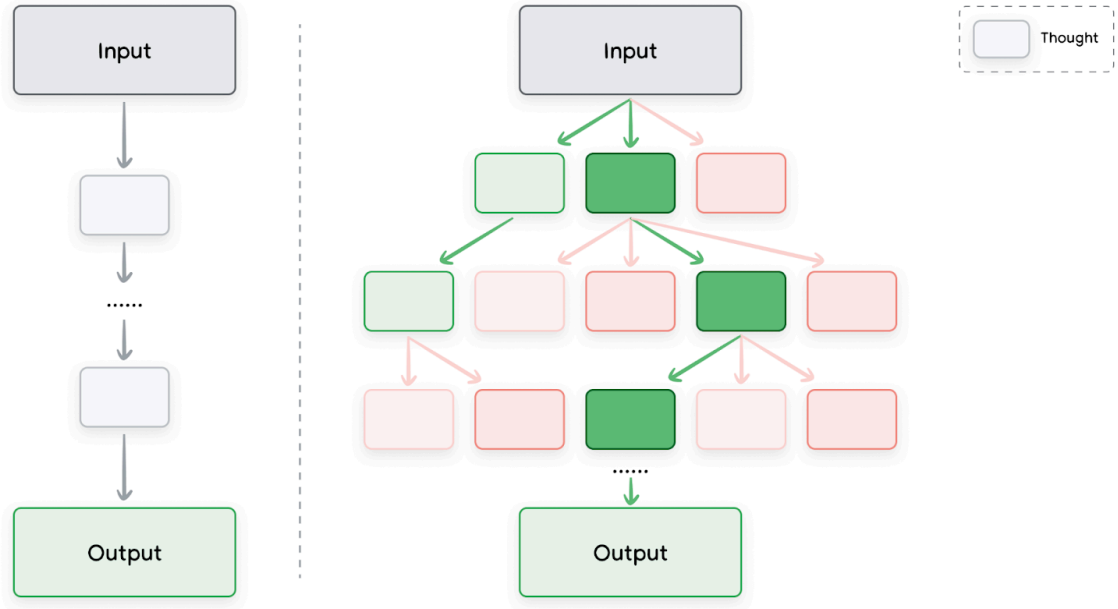

图 1: 左侧为思维链提示可视化，右侧为思维树提示可视化 (A visualization of chain of thought prompting on the left versus. Tree of Thoughts prompting on the right) 1
这种方法使得 ToT 特别适合需要探索的复杂任务。它的工作原理是维护一个思维树，其中每个思想代表一个连贯的语言序列，作为解决问题的中间步骤。然后，模型可以通过从树中的不同节点分支出来探索不同的推理路径 [1]。

有一个很棒的 notebook，它更详细地展示了基于论文《大型语言模型引导的思维树》（Large Language Model Guided Tree-of-Thought）[9] 的思维树（ToT）[1]。

ReAct (推理与行动 - reason & act)

推理与行动（Reason and act, ReAct）[13] 提示是一种范式，使 LLM 能够通过将自然语言推理与外部工具（搜索、代码解释器等）相结合来解决复杂任务，允许 LLM 执行某些操作，例如与外部 API 交互以检索信息，这是迈向智能体建模的第一步 [1]。ReAct 从根本上扩展了 LLM 的能力，允许它们与外部世界（或特定工具）交互以收集信息或执行仅靠文本生成无法完成的操作。这将推理过程置于实时、外部数据的基础之上 [1]。

ReAct 模仿人类在现实世界中的运作方式，因为我们进行口头推理并可以采取行动获取信息。ReAct 在各种领域中相对于其他提示工程方法的表现良好 [1]。

ReAct 提示通过将推理和行动结合到一个思想-行动循环中来工作。LLM 首先对问题进行推理并生成行动计划。然后它执行计划中的行动并观察结果。接着，LLM 使用观察结果更新其推理并生成新的行动计划。这个过程持续进行，直到 LLM 找到问题的解决方案 [1]。

要看到实际效果，需要编写一些代码。在代码片段 1 中，使用了 Python 的 langchain 框架

，以及 VertexAI (google-cloud-aiplatform) 和 google-search-results pip 包 [1]。

要运行此示例，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 [1]。

接下来让我们编写一些 Python 代码，任务是让 LLM 找出：Metallica 乐队成员共有多少个孩子 [1]。

Python

```
# 代码片段 1：使用 LangChain 和 VertexAI 创建 ReAct Agent
from langchain.agents import load_tools
from langchain.agents import initialize_agent
from langchain.agents import AgentType
from langchain.llms import VertexAI

# 提示：Metallica 乐队成员有多少个孩子？
prompt = "How many kids do the band members of Metallica have?"

# 初始化 LLM (VertexAI)，设置较低温度以获得更事实性的响应
llm = VertexAI(temperature=0.1)

# 加载工具（这里使用 SerpAPI 进行网络搜索）
tools = load_tools(["serpapi"], llm=llm)

# 初始化 ReAct Agent
# AgentType.ZERO_SHOT_REACT_DESCRIPTION 表示使用零样本方式，通过工具描述来决定使用哪个工具
agent = initialize_agent(tools, llm,
                         agent=AgentType.ZERO_SHOT_REACT_DESCRIPTION, verbose=True)

# 运行 Agent
agent.run(prompt)
```

代码片段 1：使用 LangChain 和 VertexAI 创建 ReAct Agent (Creating a ReAct Agent with LangChain and VertexAI) [1]

代码片段 2 显示了结果。注意 ReAct 进行了一系列五次搜索。实际上，LLM 正在抓取 Google 搜索结果以找出乐队成员的名字。然后，它将结果列为观察结果，并将思想链接到

下一次搜索 [1]。

代码片段 2：在终端执行 ReAct Agent 的输出

进入新的 AgentExecutor 链...
思考：Metallica 有 4 名成员。
行动：搜索
行动输入：James Hetfield 有多少个孩子？
观察：三个孩子
思考：1/4 的 Metallica 乐队成员有 3 个孩子
行动：搜索
行动输入：Lars Ulrich 有多少个孩子？
观察：3
思考：2/4 的 Metallica 乐队成员共有 6 个孩子
行动：搜索
行动输入：Kirk Hammett 有多少个孩子？
观察：Hammett 自 1998 年以来与他的第二任妻子 Lani 结婚。他们有两个儿子，Angel（生于 2006 年 9 月 29 日）和 Vincenzo（生于 2008 年 6 月 28 日）。
思考：3/4 的 Metallica 乐队成员共有 8 个孩子
行动：搜索
行动输入：Robert Trujillo 有多少个孩子？
观察：2
思考：4/4 的 Metallica 乐队成员共有 10 个孩子
最终答案：10
退出 AgentExecutor 链...
代码片段 2：在终端执行 ReAct Agent (Executing ReAct Agent in the terminal)
1

代码片段 2 找出了 Metallica 乐队有四名成员。然后它搜索每个乐队成员以请求孩子的总数，并将总数相加。最后，它返回孩子的总数作为最终答案 [1]。

在实践中应用 ReAct 提示需要理解，您必须持续地重新发送之前的提示/响应（并对额外生成的内容进行修剪），以及使用适当的示例/指令来设置模型。请参考托管在 GoogleCloudPlatform Github 存储库中的 notebook[14]，它通过更详尽的示例更详细地展示了实际的 LLM 输入和输出 [1]。

自动提示工程 (Automatic Prompt Engineering - APE)

此时您可能会意识到编写提示可能很复杂。如果能自动化这个过程（编写一个提示来编写提示）岂不是很棒？嗯，有一种方法：自动提示工程（Automatic Prompt Engineering, APE）。这种方法[15] 不仅减轻了人工输入的需要，而且还提高了模型在各种任务中的性能 [1]。APE

代表了 LLM 的元级别应用——使用它们来优化它们自己或其他系统的输入（提示）。这表明未来 AI 可能在与 AI 交互和控制的过程中提供重要辅助 [1]。

您将提示一个模型生成更多提示。评估它们，可能修改好的提示。然后重复 [1]。

例如，可以使用自动提示工程来帮助训练一个商品 T 恤网店的聊天机器人。我们想找出顾客可能用来订购乐队商品 T 恤的所有不同表达方式 [1]。

1. 编写将生成输出变体的提示。在此示例中，使用 gemini-pro 生成 10 条指令。见表 15 [1]：

表 **15**：使用 **LLM** 生成提示 **(Use an LLM to generate prompts)** [1]

字段	值
提示 (Prompt)	我们有一个乐队商品 T 恤网店，为了训练聊天机器人，我们需要多种订购方式来表达：“一件 Metallica T 恤，S 码”。生成 10 个变体，语义相同但保持原意。
输出 (Output)	1. 我想购买一件 Metallica T 恤，小码。 2. 我可以订购一件小码的 Metallica T 恤吗？ 3. 我想买一件小码的 Metallica T 恤。 4. 一件 Metallica 衬衫，小码，谢谢。 5. 我想要一件小码的 Metallica T 恤。 6. 我想下单订购一件小码的 Metallica T 恤。 7. 小码 Metallica T 恤，一件，谢谢。 8. 我在找一件小码的 Metallica T 恤。 9. 你能给我拿一件小码的 Metallica T 恤吗？ 10. 一件 Metallica 小码 T 恤，谢谢。

- 此表示范了 APE 的核心思想——使用 LLM 本身来为给定任务生成多样化的措辞或指令，自动化了提示设计或数据增强的部分创造性过程 [1]。

2. 评估所有指令候选。 通过根据选定的指标对候选进行评分来评估。例如，可以使用 BLEU（双语评估替补）或 ROUGE（面向召回的要点评估替补）[1]。

3. 选择评估分数最高的指令候选。 该候选将是您可以在软件应用程序或聊天机器人中使用的最终提示。您也可以调整选定的提示并再次评估 [1]。

代码提示 (Code prompting)

Gemini 主要关注基于文本的提示，这也包括编写用于返回代码的提示。让我们进入 Vertex AI Studio 并测试这些提示，看一些编码示例 [1]。

编写代码的提示 (Prompts for writing code)

Gemini 也可以扮演开发者的角色，帮助您用任何选择的编程语言编写代码。作为开发者，这可以帮助您加快编写代码的过程 [1]。

想象一下，您机器上的一个文件夹里有数百个文件需要重命名。逐个重命名每个文件会花费大量时间。您懂一点 Bash，可以编写一个脚本来自动化这个过程，但这可能也需要一些时间。所以让我们编写一个提示。您可以在公共消费者聊天机器人 Gemini 中编写提示，或者如果您更关心保密性，可以在您的 Google Cloud 帐户中编写这些提示并打开 Vertex AI Studio。Vertex AI Studio 的优势在于您可以配置温度等参数 [1]。

表 16：使用 Gemini 编程代码 (Using Gemini to program code) [1]

字段	值
名称 (Name)	1_bash_rename_files_in_folder
目标 (Goal)	编写一个提示，用于在 Bash 中编写代码以重命名文件夹中的文件。(Write a prompt to write code in Bash to rename files in a folder.)
模型 (Model)	gemini-pro
温度 (Temp)	0.1
令牌限制 (Limit)	1024
Top-K	N/A
Top-P	1
提示 (Prompt)	编写一个 Bash 代码片段，该片段询问文件夹名称。然后它获取该文件夹的内容，并通过在文件名

	前加上名称 draft 来重命名其中的所有文件。
输出 (Output)	bash #!/bin/bash # 询问文件夹名称 echo "输入文件夹名称: " read folder_name # 检查文件夹是否存在 if [! -d "$folder_name"]; then echo "文件夹不存在。" exit 1 fi # 获取文件夹中的文件列表 files=("$folder_name"/*) # 通过在文件名前加上 "draft_" 来重命名每个文件 for file in "${files[@]}"; do # 从完整路径中提取基本文件名 base_name=$(basename "$file") # 构建新的文件路径 new_file_path="$folder_name/draft_$base_nam

	e" # 重命名文件 mv "$file" "$new_file_path" done echo "文件重命名成功。"

- 此表提供了一个清晰的示例，说明如何基于自然语言描述使用 LLM 进行代码生成，展示了自动化简单脚本任务的潜力 [1]。

这看起来是不错的代码——甚至还有文档！然而，由于 LLM 无法推理，并且会重复训练数据，因此首先阅读和测试您的代码至关重要 [1]。

我们都期待的时刻，它真的有效吗？[1]

让我们先用一个只有几个文件的测试文件夹来试试，这些文件需要从 filename.txt 重命名为 draft_filename.txt [1]。

1. 复制表 16 的输出（不包括 bash 文本包装器），并将其粘贴到一个名为 “rename_files.sh” 的新文件中。
2. 打开一个终端窗口并输入：. rename_files.sh。它会要求输入文件夹名称，例如 test，然后按回车键。
3. 脚本似乎运行良好。您将看到消息：文件重命名成功。

当您查看测试文件夹时，会发现所有文件都已完美地重命名为 draft_filename.txt [1]。

它成功了！[1]

解释代码的提示 **(Prompts for explaining code)**

作为开发者，在团队中工作时，您必须阅读他人的代码。Gemini 也可以在这方面提供帮助。让我们取用表 16 的代码输出，移除注释，并要求大型语言模型解释发生了什么，见表 17 [1]：

表 **17**：解释代码的提示 **(A prompt for explaining code)** [1]

字段	值
名称 (Name)	1_bash_explain_the_code
目标 (Goal)	编写一个提示来解释 Bash 代码。(Write a prompt to explain Bash code)
模型 (Model)	gemini-pro
温度 (Temp)	0.1
令牌限制 (Limit)	1024
Top-K	N/A
Top-P	1
提示 (Prompt)	请向我解释下面的 Bash 代码： bash #!/bin/bash echo "输入文件夹名称: " read folder_name if [! -d "$folder_name"]; then echo "文件夹不存在。" exit 1 fi files=("$folder_name"/*) for file in "${files[@]}"; do base_name=$(basename "$file") new_file_path="$folder_name/draft_$base_nam

	e" mv "$file" "$new_file_path" done echo "文件重命名成功。"
输出 (Output)	这个 Bash 脚本执行以下任务： 1. 用户输入 **(User Input):** - 它使用 echo 和 read 命令提示用户输入文件夹名称。输入的文件夹名称存储在 folder_name 变量中。 2. 文件夹存在性检查 **(Folder Existence Check):** - 它使用 [! -d "$folder_name"] 条件检查具有指定名称的文件夹是否存在。如果文件夹不存在，它会显示错误消息（“文件夹不存在。”）并以退出代码 1 退出脚本。 3. 文件列表 **(File Listing):** - 它使用 files=("$folder_name"/*) 命令列出指定文件夹内的所有文件。这将创建一个名为 files 的数组，其中包含文件夹中所有文件的路径。 4. 文件重命名 **(File Renaming):** - 它使用 for 循环遍历 files 数组中的每个文件。 - 对于每个文件，它使用 base_name=$(basename "$file") 提取基本文件名，然后使用 new_file_path="$folder_name/draft_$base_name" 生成一个新的文件路径。这个新文件名是通过在原始基本文件名前加上字符串 “draft_” 来创建的。 - 然后它使用 mv "$file" "$new_file_path" 命令重命名文件，该命令将文件移动到新的文件路径。 5. 成功消息 **(Success Message):**

	- 重命名所有文件后，它会显示一条成功消息（“文件重命名成功。”）以指示重命名操作已成功。

- 此表示范了 LLM 在代码理解方面的效用。模型准确地分解了脚本，并用自然语言解释了每个命令块的目的 [1]。

翻译代码的提示 **(Prompts for translating code)**

表 16 中的 bash 代码似乎工作正常。然而，如果这个脚本能提示我输入文件名，并且理想情况下能作为一个带有 UI 的独立应用程序工作，那么它将真正具有可重用性。作为起点，对于（Web）应用程序来说，Python 比 Bash 是更好的语言。LLM 可以帮助将代码从一种语言翻译到另一种语言。参见表 18 中的示例 [1]：

表 18：将代码从 **Bash** 翻译为 **Python** 的提示 **(A prompt for translating code from Bash to Python)** [1]

字段	值
名称 (Name)	1_python_rename_files
目标 (Goal)	编写一个提示，将 Bash 代码翻译成 Python。(Write a prompt to translate Bash code to Python)
模型 (Model)	gemini-pro
温度 (Temp)	0.1
令牌限制 (Limit)	1024
Top-K	N/A
Top-P	1
提示 (Prompt)	将下面的 Bash 代码翻译成 Python 片段。 bash #!/bin/bash echo "输入文件夹名称: "

	read folder_name if [! -d "$folder_name"]; then echo "文件夹不存在。" exit 1 fi files=("$folder_name"/*) for file in "${files[@]}"; do base_name=$(basename "$file") new_file_path="$folder_name/draft_$base_name" mv "$file" "$new_file_path" done echo "文件重命名成功。"
输出 (Output)	python import os import shutil # 从用户获取文件夹名称 folder_name = input("输入文件夹名称: ") # 检查文件夹是否存在 if not os.path.isdir(folder_name):

	print("文件夹不存在。") exit(1) # 获取文件夹中所有文件的列表 files = os.listdir(folder_name) # 遍历文件 for file in files: # 创建带有前缀 "draft_" 的新文件名 # 注意：这里仅使用文件名，未保留原始扩展名 new_file_name = f"draft_{file}" # 构建原始文件和新文件的完整路径 old_path = os.path.join(folder_name, file) new_path = os.path.join(folder_name, new_file_name) # 将文件移动到新名称 shutil.move(old_path, new_path) # 打印成功消息 print("文件重命名成功。")

- 此表展示了 LLM 的代码翻译能力，将功能从一种语言的习惯用法（Bash 命令）映射到另一种语言（如 Python 的 os、shutil 库 ）[1]。

阅读并审查代码。复制提示的输出并将其粘贴到一个新文件：file_renamer.py。通过打开

终端窗口并执行以下命令来测试代码：python file_renamer.py [1]。

注意：在 Vertex AI 的 Language Studio 中提示（Python）代码时，您必须单击“Markdown”按钮。否则，您将收到缺少正确行缩进的纯文本，这对于运行 Python 代码很重要 [1]。

调试和审查代码的提示 (Prompts for debugging and reviewing code)

让我们手动对表 18 的代码进行一些编辑。它应该提示用户输入文件名前缀，并将此前缀以大写字符写入。参见代码片段 3 中的示例代码，但真糟糕。它现在返回 Python 错误！[1]

Python

```
# 代码片段 3：一个损坏的 Python 脚本
import os
import shutil

folder_name = input("输入文件夹名称: ")
prefix = input("输入要添加到文件名前的前缀字符串: ")
# 错误：使用了未定义的函数 toUpperCase
text = toUpperCase(prefix)

if not os.path.isdir(folder_name):
    print("文件夹不存在。")
    exit(1)

files = os.listdir(folder_name)
for file in files:
    # 错误：变量名不一致 new_filename vs new_file_name
    new_filename = f"{text}_{file}"
    old_path = os.path.join(folder_name, file)
    # 使用了未在循环中定义的 new_file_name
    new_path = os.path.join(folder_name, new_file_name)
    shutil.move(old_path, new_path)

print("文件重命名成功。")
```

代码片段 **3**：一个损坏的 **Python** 脚本 **(A broken Python script)** [1]

哎呀！看起来像个 bug [1]：

代码片段 **4**：**Python** 代码错误的回溯信息

以下 Python 代码给出了错误：
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：我弄坏了 Python 代码 (I broke the Python code) 1
让我们看看是否可以要求大型语言模型调试和审查代码。请看表 19 [1]：

表 19：调试和审查 **Python** 代码的提示 **(A prompt for debugging and reviewing Python code)** [1]

字段	值
名称 (Name)	1_python_debug_code
目标 (Goal)	编写一个提示来调试和审查 Python 代码。(Write a prompt to debug and review Python code.)
模型 (Model)	gemini-pro
温度 (Temp)	0.1
令牌限制 (Limit)	1024
Top-K	N/A
Top-P	1
提示 (Prompt)	以下 Python 代码给出了错误： Traceback (most recent call last): File "/Users/leeboonstra/Documents/test_folder/ren

	ame_files.py", line 7, in <module> text = toUpperCase(prefix) NameError: name 'toUpperCase' is not defined 调试错误所在，并解释如何改进代码。 python import os import shutil folder_name = input("输入文件夹名称: ") prefix = input("输入要添加到文件名前的前缀字符串: ") text = toUpperCase(prefix) if not os.path.isdir(folder_name): print("文件夹不存在。") exit(1) files = os.listdir(folder_name) for file in files: new_filename = f"{text}_{file}" old_path = os.path.join(folder_name, file) new_path = os.path.join(folder_name, new_file_name) shutil.move(old_path, new_path)

	print("文件重命名成功。")
输出 (Output)	代码调用了 toUpperCase 函数将 prefix 转换为大写，但该函数未定义。要解决此问题，您可以使用字符串类的 upper() 方法，该方法将给定字符串转换为大写。修改后的代码如下所示： python import os import shutil folder_name = input("输入文件夹名称: ") prefix = input("输入要添加到文件名前的前缀字符串: ") text = prefix.upper() # 使用.upper() 方法 if not os.path.isdir(folder_name): print("文件夹不存在。") exit(1) files = os.listdir(folder_name) for file in files: # 注意：修复了变量名 new_filename new_filename = f"{text}_{file}" old_path = os.path.join(folder_name, file)

	new_path = os.path.join(folder_name, new_filename) shutil.move(old_path, new_path) print("文件重命名成功。") 此外，代码还有一些其他可以改进的地方： 1. 新文件名的文件扩展名没有保留。最好从原始文件名中提取扩展名并将其附加到新文件名。 2. 代码假定文件夹名称中没有空格。然而，通过将空格替换为下划线或连字符来优雅地处理空格是一种好习惯。 3. 使用 f 字符串语法进行字符串插值比传统的 + 运算符更具可读性和简洁性。 4. 代码没有处理重命名过程中可能发生的错误。最好将 shutil.move 调用包装在 try...except 块中以捕获任何潜在错误。 以下是包含这些建议的改进代码： python import os import shutil folder_name = input("输入文件夹名称: ") prefix = input("输入要添加到文件名前的前缀字符串: ").upper() # 直接转大写

	if not os.path.isdir(folder_name): print("文件夹不存在。") exit(1) files = os.listdir(folder_name) print(f"正在重命名 {folder_name} 中的文件...") for file in files: # 仅处理文件，跳过子目录（如果需要） old_path = os.path.join(folder_name, file) if os.path.isfile(old_path): # 分离文件名和扩展名 name, ext = os.path.splitext(file) # 创建新的文件名（保留扩展名） new_filename = f"{prefix}_{name}{ext}" new_path = os.path.join(folder_name, new_filename) # 尝试移动文件 try: shutil.move(old_path, new_path) print(f" '{file}' -> '{new_filename}'") except Exception as e: print(f"移动文件 {file} 时出错: {e}")

	print("文件重命名过程完成。") 这些改进使代码更健壮、更灵活，同时也使其更易于阅读和理解...（响应已被截断，因为它已达到令牌限制。如果您需要更长的响应，请尝试增加令牌限制。）

- 此表有力地证明了 LLM 不仅可以修复特定错误，还可以进行主动的代码审查并提出最佳实践建议。将错误回溯与代码一起提供是调试的关键上下文 [1]。提供错误消息/回溯以及代码极大地帮助 LLM 定位问题，使调试提示比仅提供代码并说“修复它”有效得多 [1]。

太棒了。它不仅告诉了我如何解决问题，还找出了我的代码有更多错误以及如何解决它们。提示的最后一部分给出了改进代码的总体建议 [1]。

那么，多模态提示呢？ (What about multimodal prompting?)

为代码编写提示仍然使用相同常规的大型语言模型。多模态提示是一个独立的问题，它指的是一种使用多种输入格式（而不仅仅是依赖文本）来指导大型语言模型的技术。这可以包括文本、图像、音频、代码甚至其他格式的组合，具体取决于模型的能力和手头的任务 [1]。本节主要承认多模态提示的存在，但明确将其排除在详细讨论之外，使白皮书专注于基于文本（包括代码 ）的提示工程 [1]。

最佳实践 (Best Practices)

找到正确的提示需要反复调试。Vertex AI 中的 Language Studio 是一个完美的场所，可以在其中试用您的提示，并能够针对各种模型进行测试 [1]。

使用以下最佳实践成为提示工程专家 [1]。

提供示例 (Provide examples)

最重要的最佳实践是在提示中提供（单样本/少样本）示例。这是非常有效的，因为它充当了强大的教学工具。这些示例展示了期望的输出或类似的响应，使模型能够从中学习并相应地调整其自身的生成。这就像给模型一个参考点或目标，以提高其响应的准确性、风格和语调，使其更好地符合您的期望 [1]。这再次强调了之前看到的少样本学习的有效性。明确称其为“最重要”的实践，突显了它在各种任务中的巨大影响力 [1]。

简洁设计 (Design with simplicity)

提示应该简洁、清晰、易于理解（对您和模型都是如此）。作为经验法则，如果它对您来说已经令人困惑，那么它很可能对模型来说也同样令人困惑。尽量不要使用复杂的语言，也不要提供不必要的信息 [1]。有时少即是多。虽然上下文是好的，但不必要的行话或过于复杂的句子结构可能会混淆模型。直接、清晰的语言是首选 [1]。

示例：[1]

修改前 (BEFORE):
我现在正在访问纽约，我想了解更多关于好地点的信息。我和两个 3 岁的孩子在一起。我们假期应该去哪里？
重写后 (AFTER REWRITE):
扮演游客的旅行指南。描述在纽约曼哈顿适合带 3 岁孩子参观的好地方。
尝试使用描述动作的动词。以下是一些示例 1：
扮演 (Act), 分析 (Analyze), 分类 (Categorize), 归类 (Classify), 对比 (Contrast), 比较 (Compare), 创建 (Create), 描述 (Describe), 定义 (Define), 评估 (Evaluate), 提取 (Extract), 查找 (Find), 生成 (Generate), 识别 (Identify), 列出 (List), 测量 (Measure), 组织 (Organize), 解析 (Parse), 挑选 (Pick), 预测 (Predict), 提供 (Provide), 排名 (Rank), 推荐 (Recommend), 返回 (Return), 检索 (Retrieve), 重写 (Rewrite), 选择 (Select), 显示 (Show), 排序 (Sort), 总结 (Summarize), 翻译 (Translate), 编写 (Write).

具体说明输出 **(Be specific about the output)**

关于期望的输出要具体。简洁的指令可能不足以指导 LLM，或者可能过于笼统。在提示中提供具体细节（通过系统或上下文提示）可以帮助模型专注于相关内容，提高整体准确性 [1]。不要假设模型知道您想要什么。明确说明约束条件，如长度（“3 段”）、内容焦点（“排名前 5 的视频游戏机”）和风格（“对话式”）[1]。

示例：[1]

推荐 (DO):
生成一篇关于排名前 5 的视频游戏机的 3 段博客文章。
该博客文章应内容丰富且引人入胜，并应以对话式风格编写。
不推荐 (DO NOT):
生成一篇关于视频游戏机的博客文章。

使用指令而非约束 **(Use Instructions over Constraints)**

指令和约束在提示中用于指导 LLM 的输出 [1]。

- 指令 **(instruction)** 提供关于响应的期望格式、风格或内容的明确指示。它指导模型应该做什么或产生什么 [1]。
- 约束 **(constraint)** 是对响应的一组限制或边界。它限制模型不应该做什么或避免什么 [1]。

越来越多的研究表明，在提示中专注于积极指令可能比严重依赖约束更有效。这种方法与人类更喜欢积极指令而非一堆“不要做”列表的偏好相一致 [1]。以积极的方式提出请求（包含

什么）通常比消极约束（排除什么）更清晰，更不容易被误解或产生冲突 [1]。

指令直接传达期望的结果，而约束可能让模型猜测什么是允许的。它在定义的边界内提供了灵活性并鼓励创造力，而约束可能限制模型的潜力。此外，一堆约束可能会相互冲突 [1]。

约束在某些情况下仍然很有价值。例如，防止模型生成有害或有偏见的内容，或者当需要严格的输出格式或风格时 [1]。

如果可能，使用积极指令：与其告诉模型不要做什么，不如告诉它该做什么。这可以避免混淆并提高输出的准确性 [1]。

示例：[1]

推荐 (DO):
生成一篇关于排名前 5 的视频游戏机的 1 段博客文章。
仅讨论游戏机本身、制造商公司、年份和总销量。
不推荐 (DO NOT):
生成一篇关于排名前 5 的视频游戏机的 1 段博客文章。
不要列出视频游戏名称。
作为最佳实践，首先优先考虑指令，清楚地说明您希望模型做什么，仅在出于安全、清晰或特定要求需要时才使用约束。进行实验和迭代，测试指令和约束的不同组合，以找到最适合您特定任务的方法，并记录这些尝试 [1]。

控制最大令牌长度 (Control the max token length)

要控制生成的 LLM 响应的长度，您可以在配置中设置最大令牌限制，或者在提示中明确请求特定长度。例如 1:
"用一条推文长度的消息解释量子物理学。"
长度可以通过技术（配置）和语义（提示）两种方式进行管理。基于提示的控制允许更细致的长度规范（例如，“用 3 句话”，“推文长度”）[1]。

在提示中使用变量 (Use variables in prompts)

为了重用提示并使其更具动态性，请在提示中使用变量，这些变量可以针对不同的输入进行更改。例如，如表 20 所示，一个提供城市事实的提示。不要在提示中硬编码城市名称，而是使用变量。变量可以通过允许您避免重复自己来节省时间和精力。如果您需要在多个提示中使用相同的信息，可以将其存储在变量中，然后在每个提示中引用该变量。这在将提示集成到您自己的应用程序中时非常有意义 [1]。

表 20：在提示中使用变量 (Using variables in prompts) [1]

字段	值

变量 (VARIABLES)	{city} = "阿姆斯特丹"
提示 (PROMPT)	你是一个旅行指南。告诉我一个关于城市 {city} 的事实。
输出 (Output)	阿姆斯特丹是一个美丽的城市，到处是运河、桥梁和狭窄的街道。因其丰富的历史、文化和夜生活而成为一个绝佳的旅游目的地。

- 此表清晰地展示了一个简单的模板机制，用于使提示可重用并适应不同的输入，而无需重写整个提示结构。这对于程序化使用至关重要 [1]。

尝试不同的输入格式和写作风格 **(Experiment with input formats and writing styles)**

不同的模型、模型配置、提示格式、措辞选择甚至提交都可能产生不同的结果。因此，尝试提示属性（如风格、措辞选择和提示类型（零样本、少样本、系统提示））非常重要 [1]。LLM 对输入的精确措辞和格式可能出奇地敏感。微小的变化可能导致不同的输出，这再次强调了实验的必要性 [1]。

例如，一个目标是生成关于革命性视频游戏机世嘉 Dreamcast 文本的提示，可以表述为问题、陈述或指令，从而产生不同的输出 [1]：

- 问题：世嘉 Dreamcast 是什么？为什么它是一款如此革命性的游戏机？
- 陈述：世嘉 Dreamcast 是世嘉于 1999 年发布的第六代视频游戏机。它...
- 指令：编写一段描述世嘉 Dreamcast 游戏机并解释其革命性原因的文字。

对于带分类任务的少样本提示，混合类别 **(For few-shot prompting with classification tasks, mix up the classes)**

一般来说，少样本示例的顺序不应有太大影响。但是，在进行分类任务时，请确保在少样本示例中混合可能的响应类别。这样做是因为您可能否则会过度拟合示例的特定顺序。通过混合可能的响应类别，您可以确保模型正在学习识别每个类别的关键特征，而不仅仅是记忆示例的顺序。这将导致在未见过的数据上表现更鲁棒和更具泛化性 [1]。在少样本分类中，示例的顺序可能无意中成为一个信号。混合类别迫使模型依赖示例的实际内容来区分类别，而不是仅仅依赖它们在序列中的位置 [1]。

一个好的经验法则是从 6 个少样本示例开始，并从那里开始测试准确性 [1]。

适应模型更新 **(Adapt to model updates)**

了解模型架构变化、新增数据和能力非常重要。尝试更新的模型版本，并调整您的提示以更好地利用新模型特性。像 Vertex AI Studio 这样的工具非常适合存储、测试和记录您提示的各种版本 [1]。提示工程不是一次性任务。今天效果好的提示可能随着底层模型的演变而

需要调整。持续测试和适应是必要的[1]。

尝试不同的输出格式 (Experiment with output formats)

除了提示输入格式，考虑尝试输出格式。对于非创造性任务，如提取、选择、解析、排序、排名或分类数据，尝试让您的输出以结构化格式（如 JSON 或 XML）返回[1]。明确要求结构化输出（如 JSON）可以作为一种强约束，提高可靠性并减少无意义或非结构化的响应，尤其对于面向数据的任务[1]。

返回从提取数据的提示中获取 JSON 对象有一些好处。在实际应用中，我不需要手动创建此 JSON 格式，我已经可以按排序顺序返回数据（在处理日期时间对象时非常方便），但最重要的是，通过提示要求 JSON 格式，它迫使模型创建结构并限制幻觉[1]。

少样本提示部分的表 4 展示了如何返回结构化输出的示例[1]。

与其他提示工程师一起实验 (Experiment together with other prompt engineers)

如果您处于必须尝试想出一个好提示的情况，您可能希望找多个人进行尝试。当每个人都遵循最佳实践（如本章所列）时，您将看到所有不同提示尝试之间的性能差异[1]。不同的人以不同的方式处理问题。协作进行提示设计可以引入多样化的视角和措辞，可能比单个人独立工作更快地找到更好的解决方案[1]。

CoT 最佳实践 (CoT Best practices)

对于 CoT 提示，将答案放在推理之后是必需的，因为推理的生成会改变模型预测最终答案时获得的令牌[1]。

对于 CoT 和自我一致性，您需要能够从提示中提取最终答案，并将其与推理分开[1]。

对于 CoT 提示，将温度设置为 0[1]。思维链提示基于贪婪解码，根据语言模型分配的最高概率预测序列中的下一个词。一般来说，当使用推理来得出最终答案时，很可能只有一个正确的答案。因此，温度应始终设置为 0[1]。有效使用 CoT 需要注意具体的实现细节，如答案放置、可提取性和适当的配置（低/零温度）[1]。

记录各种提示尝试 (Document the various prompt attempts)

最后一条建议在本章前面已经提到过，但我们不能足够强调它的重要性：详细记录您的提示尝试，以便随着时间的推移了解哪些做得好，哪些做得不好[1]。有效的提示工程需要将其视为科学或工程过程，这需要严格记录实验（提示、设置、结果），以便进行学习、比较和复现[1]。

提示输出可能因模型、采样设置甚至同一模型的不同版本而异。此外，即使对于同一模型的相同提示，输出句子格式和措辞选择也可能存在微小差异。（例如，如前所述，如果两个令

牌具有相同的预测概率，则可能随机打破平局。这随后会影响后续预测的令牌。)[1]。

我们建议创建一个 Google Sheet，并使用表 21 作为模板。这种方法的优点是，当您不可避免地需要重新审视您的提示工作时——无论是将来重新拾起（您会惊讶于即使短暂休息后您会忘记多少），还是在不同版本的模型上测试提示性能，以及帮助调试未来的错误——您都有完整的记录 [1]。

除了此表中的字段外，跟踪提示的版本（迭代）、一个用于记录结果是“确定/不确定/有时确定”(OK/NOT OK/SOMETIMES OK) 的字段，以及一个用于记录反馈的字段也很有帮助。如果您有幸使用 Vertex AI Studio，请保存您的提示（使用与文档中列出的相同名称和版本），并在表格中跟踪指向已保存提示的超链接。这样，您只需单击一下即可重新运行您的提示 [1]。

当处理检索增强生成（RAG）系统时，您还应捕获影响哪些内容被插入到提示中的 RAG 系统的特定方面，包括查询、块设置、块输出和其他信息 [1]。

表 **21**：记录提示的模板 **(A template for documenting prompts)** [1]

字段	描述
名称 (Name)	[您的提示的名称和版本] ([name and version of your prompt])
目标 (Goal)	[对此尝试目标的一句话解释] ([One sentence explanation of the goal of this attempt])
模型 (Model)	[使用的模型的名称和版本] ([name and version of the used model])
温度 (Temperature)	[0 - 1 之间的值] ([value between 0 - 1])
令牌限制 (Token Limit)	[数字] ([number])
Top-K	[数字] ([number])
Top-P	[数字] ([number])
提示 (Prompt)	[写下完整的提示] ([Write all the full prompt])
输出 (Output)	[写下输出或多个输出] ([Write out the output or

	multiple outputs])

- 此表为关键的文档实践提供了一个具体的、结构化的模板。遵循此模板可确保系统地捕获每个提示实验的关键信息（目标、输入、配置、输出），以便后续分析、比较和复现[1]。

一旦您觉得提示接近完美，就将其纳入您的项目代码库。在代码库中，将提示保存在与代码分开的文件中，以便于维护。最后，理想情况下，您的提示是操作化系统的一部分，作为提示工程师，您应依赖自动化测试和评估程序来了解您的提示在任务上的泛化程度如何[1]。

提示工程是一个迭代的过程。制作和测试不同的提示，分析并记录结果。根据模型的性能优化您的提示。持续实验，直到达到期望的输出。当您更改模型或模型配置时，返回并继续实验先前使用的提示[1]。

总结 (Summary)

本白皮书讨论了提示工程。我们学习了各种提示技术，例如[1]：

- 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)
- 回退提示 (Step-back prompting)
- 思维链 (Chain of thought)
- 自我一致性 (Self consistency)
- 思维树 (Tree of thoughts)
- ReAct
- 自动提示工程 (APE)

我们甚至研究了如何自动化您的提示[1]。

白皮书随后讨论了生成式 AI 的挑战，例如当您的提示不足时可能发生的问题。最后我们以如何成为更好的提示工程师的最佳实践作为结束[1]。

尾注 (Endnotes)

1. Google, 2023, Gemini by Google. 可在 https://gemini.google.com 获取。[1]
2. Google, 2024, Gemini for Google Workspace Prompt Guide. 可在 https://inthecloud.withgoogle.com/gemini-for-google-workspace-prompt-guide/dl-cd.html 获取。[1]
3. Google Cloud, 2023, Introduction to Prompting. 可在

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

Works cited

1. www.gptaiflow.tech, accessed April 10, 2025, https://www.gptaiflow.tech/assets/files/2025-01-18-pdf-1-TechAI-Goolge-whitepaper_Prompt%20Engineering_v4-af36dcc7a49bb7269a58b1c9b89a8ae1.pdf