ChatGPT

提示词优化工程化：可复现流程、结构化设计与评估体系

背景说明

大型语言模型（LLM）的能力表现很大程度取决于所给定的提示词（Prompt）。**Prompt 工程（Prompt Engineering）**指的就是通过精心设计和优化这些提示词，来引导模型产生更准确、高效、符合预期的输出 [1]。OpenAI 官方文档（2023）将 Prompt 工程定义为"为模型编写有效指令的过程，使其能够一致地生成满足要求的内容" [2]。换言之，Prompt 工程的本质是在 **人类与AI之间建立"语义契约"**，通过精确的指令设计来弥合自然语言需求与机器理解之间的鸿沟 [3]。在 GPT-4、Claude 等数千亿参数模型时代，一个优秀提示词设计往往可使模型性能提升数倍（提高300%～500%），已成为AI应用开发的关键竞争力 [3]。

值得注意的是，**Prompt 优化已经演变为一个工程问题**。它不再是凭运气的"玄学调参"，而是需要系统化的方法论和迭代流程，类似传统软件工程中的调试和优化。正如某份谷歌内部白皮书总结的那样：我们正在将"如何和AI对话最有效"这件事从玄学变成科学，从碰运气变成可复用的工程实践 [4]。这意味着，对提示词的优化需要**明确的目标、可重复的流程、量化的评估**，并结合软件工程思维进行版本管理和持续改进。下文将围绕 Prompt 工程的结构化设计原则、上下文与代理机制、评估体系、常见失败模式及改进、以及完整的闭环流程等方面，进行系统性的阐述。

提示词工程的结构化设计

要让一个提示词有效，首先需要关注其**结构化设计**。经验表明，清晰、有条理的提示结构可以显著提升模型理解意图的能力 [5] [6]。一个完整的 Prompt 通常包含以下模块 [7]：

- **角色定义**：给予模型一个身份或角色定位，比如"你是一位有10年经验的Python工程师"。这有助于模型以特定视角和口吻回答。
- **任务描述**：明确要模型完成的任务是什么，例如"请分析以下代码中的潜在 bug"。尽量使用清晰具体的语言描述目标。
- **相关上下文**：提供完成任务所需的背景信息或输入数据，例如提供要分析的代码片段或待总结的文本。上下文可以帮助模型更精准地理解问题 [8]。
- **输出格式要求**：清楚指定希望模型输出的形式和格式。如要求输出JSON、列表，或限定答案字数等 [9] [10]。这可以避免模型给出杂乱无章的答案，方便后续解析处理。
- **示例（Few-shot 示例）**（可选）：提供一到多个范例输入-输出对，以示范期望的回答风格或步骤 [11] [12]。Few-shot 提示利用**上下文学习**能力，让模型从给定示例中总结模式，从而在新输入上复制这种模式，提高准确度。实践中，零样本 (Zero-shot) 提示直接给指令，少样本 (Few-shot) 提示则加入示例，当任务复杂或输出格式特殊时，Few-shot 常能显著改善效果。

以上要素可以按照需要组合成模板。在实际设计提示词时，有几项**核心原则**需遵循：

- **目标导向，指令明确**：在提示的一开头就清晰阐明用户的意图和任务目标，避免含糊不清 [5]。例如"不仅告诉模型做什么，还要指出做到何种程度、有什么具体要求"。明确的指令能降低模型误解任务的概率。
- **信息充分，提供上下文**：确保模型完成任务所需的信息尽量包含在提示中 [13] [14]。这包括相关背景、前提条件等。例如对于问答任务，可以在提示中附加知识片段或上下文资料。没有提供的知识，模型可能会胡乱

猜测，从而产生幻觉（臆断内容）。通过检索增强（Retrieval-Augmented Generation, RAG）将外部知识注入提示，是提升专业领域问答准确性的有效手段 [15]。

- **示例指导，少样本演示**：当单纯描述不容易让模型 grasp 需求时，提供**Few-shot 示例**是很好的做法 [12]。比如信息抽取任务中，给出几组输入文本和期望提取结果的示例，能大大提高模型按照指定格式提取的准确率 [16] [17]。需要注意示例的选择和顺序可能影响效果——通常应确保示例覆盖典型情况，最后一个示例最好与真实输入类似，从而防止顺序不当导致性能下降 [4]。
- **逐步推理，引入链式思维**：对于复杂推理任务，可以在提示中引导模型**逐步给出思考过程**，即所谓"链式思维 (Chain-of-Thought, CoT)"提示 [18]。例如在数学应用题提示中加入"请逐步推理并给出最终答案"，或提供示例展示思考步骤 [19] [20]。研究表明，让模型输出中间推理步骤能显著提升复杂推理题的准确率——谷歌的实验显示，在数学题上使用思维链提示将准确率从42%提升到了89% [21]。CoT 原则已成为提示词工程的重要策略之一，用来增强模型的逻辑性。
- **格式与风格控制**：通过在提示中明确要求输出格式（如表格、JSON、分点陈述）和风格语气（如严肃学术风 vs. 通俗说明），可以让结果更符合预期 [9] [22]。例如："输出格式应为包含关键字段的JSON"或"语气友好且专业"。模型往往能遵循这些约束，从而提高结果的可用性和一致性。
- **正面引导，避免消极提示**：提示中与其罗列**不希望**模型做什么，不如清晰描述**希望**模型怎么做 [23] [24]。例如，将"不要出现敏感信息"改为"如果用户询问敏感信息，请礼貌地解释无法提供"。正向指令能避免模型过度专注于禁忌，反而忽视了要完成的任务。

通过结构化设计和上述原则，一个 Prompt 模版可能如下：

> **系统角色**：你是一名知识渊博的历史学家和文学评论家。
> **任务**：请比较以下两段文献的写作风格异同，并提供你的分析见解。
> **上下文**：段落A："...(内容)..." 段落B："...(内容)..."
> **要求**：以条列清单形式输出三点比较，每点不超过50字。必要时可引用文中句子作为例证。
> **示例**：*(如果有示例输出，可在此提供格式演示)*

这样的提示包含了角色、任务、内容、格式要求，并辅以可能的示例，模型据此往往能产出有条理且符合要求的答复。

上下文工程与 Agent 拆解机制

随着需求复杂度提高，单轮的提示词有时难以完成任务。这时就需要借助**上下文工程**和**代理 (Agent) 拆解**的思想，将问题分解、充分利用多轮对话或工具，提高解决问题的能力。

上下文工程关注的是如何有效地提供和管理模型的上下文信息。LLM具有固定的上下文窗口长度（如GPT-4可达数千 tokens），合理利用这有限的上下文至关重要 [25]。上下文工程包括：控制提示长度以避免超过窗口、使用分隔符将不同部分信息明确分块 [5]、在多轮对话中适时地**重复关键指令**防止模型"遗忘"等等。此外，通过引入外部知识库，当模型自身知识不足时，可以**检索相关信息并附加在提示里** (即RAG技术) [15]。例如，提供最新的产品文档片段在上下文，使模型回答技术问题时有据可依。上下文工程的目标是**在不修改模型参数的情况下，最大化利用提示承载的信息量**，相当于用提示软性地"微调"模型行为。

而**Agent 拆解** 则是另一种工程思路，即将复杂任务拆分成多个可控的子步骤，由模型逐步完成，类似于让模型扮演问题求解的"智能体"。LangChain 等开源框架在2023年流行起来，正是提供了这样的能力：开发者可以定义一系列 Chain 或 Agent，让模型可以**先思考（Reason），再行动（Act）**，必要时调用工具或函数来获取外部信息，之后再据此回答。这一思路源自论文 *ReAct*（Reason + Act）提示范式 [26]，其流程通常是：模型读取到用户问题后，先产出一段思考内容（这部分在内部不直接呈现给用户），然后根据需要选择调用某个工具（比如搜索引擎、计算函

数），获取结果再反馈进上下文，最后给出回答。通过这样的链式 Agent，模型可以在一个复杂任务中多次迭代提示-响应，从而**分而治之**完成原本单轮无法完成的任务。例如，在一个文本问答Agent中，模型可以被提示："如果需要，先Summarize文本再回答问题"，模型先输出总结，再根据总结回答用户问题，实现了两步推理。这种多步骤提示在LangChain中可以通过定义**PromptTemplate**和**AgentExecutor**等模块方便地实现，开发者只需定义每一步的Prompt内容和衔接逻辑。

OpenAI 提供的**函数调用(Function Calling)**接口（2023）则是另一种实现 Agent 的方式。开发者可以事先定义好模型可用的一组函数（工具），包括函数名称、用途描述、参数格式等，然后在提示中通过系统消息明确告知模型何时应调用哪个函数 [27]。模型在生成回答时，如果判断需要用工具，就会输出一个特殊格式要求调用函数以及参数 [28]。此时由外部应用执行该函数并将结果返回给模型，再让模型继续生成最终答复 [29]。这种机制保证了模型输出**结构化的函数调用信息**而非自由文本，因而**降低了解析复杂文本的负担** [30]。例如，以前为了让模型给出JSON格式，可能需要反复强调格式且仍有出错风险；但用函数调用，只要定义输出schema，模型会严格填充JSON对象。这不仅提高了格式一致性，也赋予模型调用外部API、查询数据库等能力，大幅扩展了AI助理的实用性。总结来说，Agent拆解和函数调用体现了 Prompt 工程的 **流程编排能力**：通过设计一系列互相衔接的提示，让模型像程序一样分步骤完成任务，实现比单次对话更复杂的行为。 [26] [27]

需要注意，在构建多步骤提示或Agent时，要警惕累积误差和上下文混淆。因此工程上常用的技巧包括：在系统提示中加入元指令，明确每一步模型该做什么、不该做什么；在每轮用户输入前后加入分隔符或标签区分各阶段；对模型的中间思考过程进行过滤避免直接显示给用户等。这些措施都属于 Prompt 工程的一部分，目的是**确保复杂提示流程依然可控、可解释**，让初学者也能调试每一步的行为。

提示词优化的评估体系

要将 Prompt 优化落到实处，必须建立**评估体系**来衡量提示词的质量和改进效果。与传统模型调优类似，我们需要从多维度制定指标，对模型在特定 Prompt 下的输出进行测评 [31]。面向提示词优化的评估通常包括以下几个方面：

- **格式一致性**：模型输出是否严格符合预期的格式要求，包括标点、段落结构、JSON键名等的一致性。如果我们要求输出Markdown列表，模型是否每次都输出正确的列表语法？格式一致性可以通过程序自动检查，例如对JSON输出进行反序列化验证。如果模型偶尔偏离格式，就需要改善提示中对格式的强调或示例示范。
- **输出稳定性**：同一提示多次生成结果的稳定程度。理想情况下，一个高质量Prompt在相同条件下应产生内容一致或风格近似的输出 [32]。若输出波动很大，可能提示存在歧义或模型对细节把握不牢。评价稳定性可以让模型在温度为0等设置下多次生成并计算相似度 [32]。例如要求至少达到85%以上的相似度，否则就需要通过减少随机性或增加限制条件来提高一致性。
- **内容质量**：模型输出在完成任务目标上的有效性和正确性。不同应用有不同侧重点：比如问答任务关注**准确率**和**完整性**，创意写作任务关注**连贯性**和**文采**，代码生成关注**功能正确性**等。内容质量往往需要人工评价或任务相关的自动指标来衡量。 [33] 中提到可采用人工评分（如1-5分）来评估任务完成度，设定达到某个均分（如≥4.2）为目标。也可结合一些代理指标，如摘要任务用ROUGE/L，翻译任务用BLEU等，但对于开放文本生成，这些指标局限性较大。目前业界也探索用LLM自身来充当评审（如GPT-4打分)的方式辅助评估。
- **拒答和安全性**：模型在不该回答时是否适当拒绝，在该回答时是否不会错误拒绝。这方面涉及**内容安全和合规**。一个好的Prompt应该在保证模型听从指令完成任务的同时，不引导模型产生有害内容。当用户请求越界内容时，模型能否按照预期给出拒绝或安全警告的回应？评估方法可以是构建一组敏感请求来测试模型的响应：要求该拒答时拒答率接近100%（不产出违规内容） [34]，同时对一些正常请求不出现过度谨慎的无端拒绝。提示词可以通过**加入明确的安全边界描述**来影响模型的行为。例如在系统消息中声明禁止输出何种

内容，或采用Anthropic Claude类似的"宪法原则"指导，让模型自我审查不良输出。在评估中，一旦发现模型要么不该拒绝却拒绝了，要么该拒绝却给出了内容，都意味着 Prompt 需改进（可能要更清晰地加入或调整限制指令）。

- **效率和成本**（附加）：从工程视角，还可以关注提示词对调用成本和延迟的影响，比如每次输出的token长度、响应时间等[35]。一个提示如果过长过复杂，可能导致响应变慢、费用增加。因此评估时可以记录平均响应时间、tokens消耗，与行业基准对比[35]。在满足质量的前提下，越精简高效的Prompt越有优势。

综上，建立提示词优化评估体系需要综合考虑**格式、稳定性、质量、安全、效率**等维度。实际项目中，可以针对这些维度制定自动化的测试用例或指标。比如：准备一批标准输入，观察模型输出格式是否完全符合预期（格式检查）；重复试验多次看输出内容差异（稳定性）；让领域专家对答案打分（质量）；对敏感问句检查违规率（安全）等等。有了量化评估，才能客观比较不同Prompt方案的优劣，为后续优化指明方向。

常见失败模式与修复方案

Prompt 工程在实践中常会遇到各种"踩坑"或失败模式。了解这些模式并掌握相应的改进策略，是初学者提升提示词设计水平的必经之路。下面总结业界一些典型的失败案例及修复方案：

- **输出冗余或无关信息**：模型给出的答案夹带了不需要的内容，或啰嗦重复，不够简洁。这往往是因为提示对输出范围约束不明，模型自由发挥过度。**解决方案**：在提示中增加**长度限制**或明确要求简明扼要；使用标记提示输出的开始和结束，如添加特殊分隔符"### 答案开始 ###"，避免模型跑题。还可加入负面指令，例如"避免包含营销语言"来防止偏离正题。通过迭代调整，使模型只专注于所需的信息点。

- **格式错误或不一致**：模型输出未遵循所要求的格式，比如缺失字段、标点错误或多了一些解释性文字。这会给后续解析带来困难。**解决方案**：首先在 Prompt 中**严格声明格式**，最好提供**示例**展示正确格式[36][17]。例如告诉模型："输出JSON格式，包含以下键……，不要额外添加说明"。如果仍有偏差，可以尝试使用 OpenAI 函数调用或工具，使模型产出机器可读的结构化结果[30]。同时，调低随机性（如设置 temperature=0）也有助于减少不同格式之间的漂移。格式问题通常通过**加强提示中的约束**即可解决。

- **模型产生幻觉（不准确信息）**：模型在缺乏知识时编造了事实，给出错误答案。这是开放式生成中常见的问题。**解决方案**：提供**更多上下文依据**，不要让模型无中生有。如果能预见模型缺少某方面知识，提前在提示里附加相关资料（哪怕简短说明）。或者引导模型在回答不确定时直接说"不确定"而非乱猜（这可通过在系统信息中明示："如果没有把握宁可表明未知，也不要编造答案"）。另一种策略是在提示中加入**检索步骤**：如先让模型生成需要查询的关键词，然后由系统检索资料再供模型回答。这实际上将问题转化为Agent多步模式，避免模型基于训练记忆乱猜。总之，减少幻觉需要**减少模型不确定区域**：要么补足信息，要么明确限制模型不要妄自发挥。

- **领域专业性不足**：模型输出内容泛泛，未能达到领域专家要求。这可能因为提示过于通用，模型没有切换到特定领域的上下文。**解决方案**：在提示中**注入领域知识**或术语解释；设定模型为相关领域的专家角色（例如："你是一位拥有司法执照的律师…"）；必要时结合**领域数据微调**，但对于Prompt工程来说，更直接的是让Prompt里体现专业风格和知识引用。通过这些调整，可使输出更加专业精准。

- **Few-shot 示例失效**：有时明明提供了示例，但模型仍未按照示例模式输出。这可能因为示例选择不当或顺序不佳。比如示例不够典型，或者最后一个示例与真实任务差异太大导致模型类推失败[4]。**解决方案**：精选能够代表主要任务的示例，数量不宜过多以免超出上下文容量。调整示例顺序，一般将最接近用户问题形

式的示例放最后，确保模型重视该模式 [4] 。另外，检查示例本身是否存在歧义或错误输出，如果有应修正或更换。通过多次试验找到**有效示例集**，Few-shot 的优势才能发挥出来。

- **过度拒答或不当拒答**：有时模型对正常请求也产出了拒绝回复，显得过于谨慎；反之在需要拒绝的场合却漏拒绝。这通常与提示中的安全指令平衡有关。**解决方案**：如果过度拒答，可能是系统提示中过滤规则太严或措辞导致模型过度敏感。可以适当**放宽或具体化**限制说明，例如从“一律不可提供XXX”改为“在涉及违法时不可提供XXX”。如果模型漏拒绝，则需要**加强提醒**，比如在开头明确列出哪些类别内容绝对不允许，并要求模型发现此类请求立即拒答且不给具体信息。还可采用Anthropic提出的“宪法提示”理念：给予模型一系列原则让其自我检查输出是否符合伦理要求，然后在Prompt中引导模型执行这一检查步骤（相当于在回答前加一道“想一想，这是否违反原则”的链）。通过调整系统/辅助提示中的指导，能校准模型对拒答的拿捏，实现既不滥拒也不漏拒。

- **参数设置不当**：Prompt 工程不仅包含提示词文本，也涉及一些生成参数的设置。不恰当的参数可能导致失败模式。例如温度 (temperature) 过高会让输出每次变化很大、偶尔跑题；反之温度为0虽然稳定但可能陷入刻板重复。又如max_tokens过少会截断有用信息。**解决方案**：根据任务特性调整参数。**确定性要求高的任务**（如数学计算、格式填表）应使用低温度甚至0，以保证稳定和准确 [4] ；**创意类任务**可以提高温度增加多样性但仍需观察输出质量。逐步尝试不同参数组合并评估输出，找到最佳平衡。必要时可以采用**自一致性 (Self-Consistency)**方法：对同一问题采样生成多份答案，再通过多数投票或评分选最佳答案，降低单次随机性的影响。参数调优和Prompt文字优化常常需要配合进行，以达到最优效果。

总的来说，Prompt 失败的模式五花八门，但解决思路都是围绕**“哪里出问题就有针对性地在Prompt上做文章”**。这体现了Prompt优化的工程化思维：不断发现问题、分析原因、调整设计、再次验证。初学者在实践中应善于**记录每次提示的效果**, 将不理想的案例分类归因，然后有针对性地套用上述方案修正。例如发现输出有无关段落，就考虑是不是缺少长度约束；看到模型乱翻译专有名词，就考虑加入术语解释或要求保留英文等。通过持续的trial-and-error和经验积累，提示词的质量会逐步提升。

提示词工程闭环流程

要实现 Prompt 优化的工程闭环，需要制定一套循序渐进、可重复执行的流程。一般而言，一个**提示词工程闭环**包括如下阶段（图示见下方） [37] ：

提示词工程闭环流程示意：从明确任务到设计提示、测试评估、反复迭代，直至部署上线，形成一个持续优化的循环。

1. **需求分析与任务定义**：首先明确需要解决的问题和目标输出。此阶段相当于制定“需求文档”，包括确定模型扮演的角色、任务边界、成品输出应达到的标准等。只有目标清晰，才能设计出有针对性的提示词。例如，要构建一个客服机器人，则需要罗列出它能回答的主题范围、语气风格、禁止事项等作为任务定义的一部分。

2. **初始Prompt设计**：根据任务需求起草一个初版的提示词。套用前文提到的结构化模板，填入角色、任务说明、上下文占位符、输出格式要求，必要时猜测性地添加few-shot示例。这个阶段产出的Prompt可以看作**baseline**版本，用于后续对比改进。尽量依据已有最佳实践来撰写，以减少明显的问题。

3. **测试与输出收集**：将初始Prompt投入实际测试。可以使用一组代表性的输入（测试用例）来调用模型生成输出 [38] 。注意收集多样化的场景来全面考察，比如不同难度的问题、不同长度的上下文等。测试既可以人

工交互进行，也可以编写脚本批量调用模型API获取结果。此阶段的目标是获得模型在现有Prompt下的**性能基线**，以及暴露存在的问题。

4. **结果评估与诊断**：对收集的模型输出按照前述评估体系进行分析。评估哪些指标未达预期，例如：格式是否正确？回答是否完整？有无出现违规内容？以及用户体验是否友好？针对发现的缺陷，诊断其可能的成因——这一步需要结合对Prompt内容和模型行为的理解。比如如果多次测试发现模型回答偏离主题，可能诊断为提示中的任务描述不够具体；如果输出格式时对时错，可能诊断为示例不足或模型未理解格式要求。**找准症结**为下一步改进提供了方向。

5. **Prompt优化迭代**：依据诊断结果，对提示词做有针对性的修改 [38] 。这可能是微调语言措辞，使指令更清晰；增加或替换few-shot示例；加入新的约束条件；调整系统角色描述；甚至细微到改变示例顺序、标点符号等。一项改动完成后，再次进行第3步测试，看问题是否缓解、指标是否提升。通常需要多轮迭代才能将Prompt调优到理想状态——这和软件调试有相似之处，每次只改一两样，验证效果，逐步逼近目标。整个过程中保持良好的版本管理很重要，可以采用注释或文档记录每次Prompt改动及其效果，形成**Prompt变更日志**。

6. **上线部署与监控**：当Prompt在测试中达到预期表现后，就可以在实际应用中部署。例如将Prompt集成到产品的对话流程中供真实用户使用。部署后并不是万事大吉，还需要设立**监控机制**：跟踪提示词的使用频率、模型响应日志、用户反馈等 [39] 。一旦出现新问题（例如某些用户提问触发了Prompt的盲区），就再次进入诊断-优化流程，不断改进。企业级应用中，有团队会建立**提示词库**，对每个Prompt进行版本控制、权限管理和生命周期管理 [40] [39] 。这样当Prompt更新时，旧版本可以追溯，新版本经过审批后替换，全程有记录可审计。这些做法都使Prompt优化成为一套**可复现、可协作的工程流程**，而非个人摸索的黑盒。

7. **持续反馈与迭代**：实际应用中的反馈是最宝贵的改进资源。可以定期收集用户对于模型回答的满意度、或者统计模型失败的案例类型，将这些反馈纳入下一轮Prompt优化。在一些先进实践中，甚至会将收集到的高质量问答对用来微调模型本身，使其对特定Prompt响应更佳 [39] 。当然，在模型架构更新、业务需求变化时，也需要同步调整Prompt。这种**反馈驱动的持续迭代**确保Prompt工程不会停滞，而是随着环境演进而不断优化，形成真正的闭环。正如有金融公司报告的那样，引入"提示-反馈-迭代"机制后，提示词复用率提升了60%，开发迭代周期缩短了近一半 [39] ，体现了Prompt工程闭环的价值。

通过以上流程，Prompt优化从灵感尝试上升为**系统工程实践**：每一步都有据可依、有迹可循，团队协作下也能重复执行。这对于刚入门的大模型应用开发者而言，将零散的技巧融会贯通，搭建起完善的思维框架，是迈向Prompt高手的必经之路。

总结

Prompt 工程化代表了一种利用LLM的全新范式转变。从本质上看，它是**连接人类需求与AI能力的关键桥梁** [41] 。通过结构化的提示设计和严格的优化评估，我们可以在无需改动模型参数的情况下，大幅提升模型输出质量，实现从"通用生成"到"精准控制"的跨越 [42] 。本报告讨论了为何Prompt优化是工程问题，以及如何围绕模板设计、上下文利用、评估指标和失败模式来系统性地改进提示词。对于研一新生或企业初学者来说，这些经验意味着：与其把大模型当作神秘黑箱，不如以工程师的视角去调试它、打磨它。每一次Prompt的改进，都是在教会模型更好地理解我们的意图。

展望未来，Prompt工程将继续演进并融合新的技术手段。一方面，**自动化提示生成和优化**正在兴起，比如用强化学习算法自动搜索最优提示（如AutoPrompt），或将提示向量化为可训练参数（Prompt Tuning）等 [43] 。这些技术

有望减少人工试错成本。另一方面，随着多模态大模型的发展，提示工程将扩展到**文本-图像-音频跨模态**的指令设计[44]，这对提示的结构和上下文提出更高要求。此外，在提示工程中引入**伦理与治理框架**也变得必要，例如自动检测提示潜在偏见、记录提示版本以确保可追溯等[45]。所有这些进展都指向同一事实：Prompt工程正日趋成熟，成为AI应用开发不可或缺的模块。对初学者而言，把握好提示词优化的可复现流程、结构化方法和评估体系，将为日后深入AI领域打下坚实基础。正如业内共识，**掌握Prompt工程，就掌握了释放大模型潜能的钥匙**。让我们以工程师的严谨和创新精神，不断探索这一领域，在实践中迭代成长。

.references

---

[1] [8] [11] [19] [20] 一文彻底搞懂大模型 - Prompt Engineering（提示工程）-CSDN博客
https://blog.csdn.net/xx_nm98/article/details/141269197

[2] Prompt engineering | OpenAI API
https://platform.openai.com/docs/guides/prompt-engineering

[3] [7] [9] [10] [15] [18] [21] [25] [31] [32] [33] [34] [35] [39] [40] [41] [42] [43] [44] [45] AI大模型Prompt工程全解析：定义、格式与优化实践
https://cloud.baidu.com/article/3715184

[4] [26] 谷歌AI操控手册！Prompt Engineering核心玩法 - 53AI-AI知识库|大模型知识库|大模型训练|智能体开发
https://www.53ai.com/news/tishicikuangjia/2025070185932.html

[5] [6] [12] [16] [17] [22] [23] [24] [36] Best practices for prompt engineering with the OpenAI API | OpenAI Help Center
https://help.openai.com/en/articles/6654000-best-practices-for-prompt-engineering-with-the-openai-api

[13] [14] [37] [38] Overview of prompting strategies | Generative AI on Vertex AI | Google Cloud Documentation
https://docs.cloud.google.com/vertex-ai/generative-ai/docs/learn/prompts/prompt-design-strategies

[27] Function calling | OpenAI API
https://platform.openai.com/docs/guides/function-calling

[28] [29] [30] Create an Agent with OpenAI Function Calling Capabilities | by Tianyi Li | TDS Archive | Medium
https://medium.com/data-science/create-an-agent-with-openai-function-calling-capabilities-ad52122c3d12