**什么是Context Engineering**

Context Engineering正迅速成为人工智能工程师的一项关键技能。它不再仅仅关乎巧妙的提示，而是关于对上下文进行系统性的编排。

图片包含 游戏机, 电子, 光盘

AI 生成的内容可能不正确。

问题：大多数AI agent失败，并非因为模型不佳，而是因为它们缺乏取得成功所需的正确上下文。

Context Engineering涉及创建动态系统，该系统能够提供：

* 正确的信息
* 正确的工具
* 以正确的格式

这可以确保LLM能够有效地完成任务。

**为什么传统的提示工程不够**

早期，我们专注于使用 “magic words” 来诱导出更好的回应。但随着人工智能应用变得日益复杂，完整且结构化的上下文远比巧妙的措辞重要得多。

**上下文工程系统的4个关键组成部分**

* 动态信息流：上下文来自多个来源：用户、先前的交互、外部数据、工具调用。你的系统需要智能地将所有这些信息整合在一起。
* 智能工具访问：如果你的人工智能需要外部信息或操作，为其提供合适的工具。对输出进行格式化，使其尽可能易于理解。
* 内存管理：
  + 短期：总结长对话
  + 长期：跨会话记住用户偏好

* 格式优化：一条简短、描述性的错误消息每次都胜过一个庞大的JSON数据块。

总结：上下文工程正成为新的核心技能，因为它解决了真正的瓶颈问题：不是模型能力，而是信息架构。随着模型不断改进，上下文质量成为了限制因素。

**Context Engineering for AI Agents: Lessons from Building Manus**

* [https://manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus](https://link.zhihu.com/?target=https%3A//manus.im/blog/Context-Engineering-for-AI-Agents-Lessons-from-Building-Manus)

在Manus项目开始阶段，面对关键决策：

1. train an end-to-end agentic model using open-source foundations
2. build an agent on top of the **in-context learning** abilities of frontier models

最终Manus选择全面押注 **context engineering**

context engineering是一门实验科学，我们将这种manual process of architecture searching, prompt fiddling, and empirical guesswork（手动架构搜索、提示调整和经验猜测的过程）称为 **"**[**Stochastic Graduate Descent**](https://zhida.zhihu.com/search?content_id=260871316&content_type=Article&match_order=1&q=Stochastic+Graduate+Descent&zhida_source=entity)**"**

**Design Around the KV-Cache**

[KV-cache](https://zhida.zhihu.com/search?content_id=260871316&content_type=Article&match_order=1&q=KV-cache&zhida_source=entity) hit rate 是生产阶段AI agent最重要的单一指标。

**How typical agent operates:**

* 当收到用户输入后，agent通过 a chain of tool uses来完成任务。
* 在每次迭代中，模型根据current context从预定义的动作空间中选择一个 **action**。然后在 **environment** 中执行该action，以产生 **observation**。**action**和**observation**被附加到context中，形成下一次迭代的输入。
* 这个循环持续进行，直到任务完成。
* 随着每一步的推进，context不断增长，而输出——usually a structured function call——保持相对简短。这使得 agents 相比chatbots的 prefilling and decoding比例高度倾斜（agent会随着任务的解决过程不断增加context，导致prefilling远大于decoding）。例如在Manus中，平均输入与输出的token比例约为100:1。
* 幸运的是，具有相同前缀的上下文可以利用**KV-cache**，无论你是使用自托管模型还是调用推理API，这大大减少了time-to-first-token(TTFT)和inference cost。
  + 我们说的不是小幅度的节省：例如使用Claude Sonnet时，缓存的输入token成本为0.30美元/百万token，而未缓存的成本为3美元/百万token——相差10倍。
  + [https://medium.com/@joaolages/kv-caching-explained-276520203249](https://link.zhihu.com/?target=https%3A//medium.com/%40joaolages/kv-caching-explained-276520203249)

图示

AI 生成的内容可能不正确。

**How improving KV-cache hit rate**

1. Keep your prompt prefix stable. 由于LLM的自回归特性，即使是单个标记的差异也会使该标记之后的缓存失效。一个常见的错误是在系统提示的开头包含时间戳——尤其是精确到秒的时间戳。虽然这让模型能告诉你当前时间，但也会降低你的缓存命中率。
2. Make your context append-only.避免修改之前的actions or observations。确保你的序列化是确定的。许多编程语言和库在序列化JSON对象时不保证键顺序的稳定性，这可能会悄无声息地破坏缓存。
3. Mark cache breakpoints explicitly when needed.某些模型提供商或推理框架不支持自动增量前缀缓存，而是需要在上下文中手动插入缓存断点。在分配这些断点时，要考虑潜在的缓存过期问题，并至少确保断点包含系统提示的结尾。

此外，如果使用像 vLLM 这样的框架自托管模型，请确保prefix/prompt caching is enabled，并且如果在distributed workers之间使用session IDs 等技术需要确保route requests的一致性。

**Mask, Don't Remove**

随着agent能力的增强，其action space自然变得更加复杂，简单说就是tools数量的爆炸增长。 最近流行的MCP会使得这个问题火上浇油。如果你允许用户自定义工具，更会加重这个问题。

我们的建议是：unless absolutely necessary, **avoid dynamically adding or removing tools mid-iteration(迭代过程中)**.

1. 在大多数LLM中，tool definitions在序列化后位于上下文的前部，通常在system prompt之前或之后。因此（tool）任何更改都会使后续所有actions and observations的KV-cache失效。
2. 当先前的actions and observations仍然引用当前上下文中不再定义的工具时，模型会感到困惑。如果没有constrained decoding（约束解码），这通常会导致schema violations or hallucinated actions（模式违规或幻觉动作）

为了解决这个问题同时improving action selection，Manus的方案：使用context-aware 的状态机来管理工具可用性。它不是remove tools，而是在解码过程中**mask the token logits**，以基于current context阻止（或强制）选择某些动作。

图形用户界面

AI 生成的内容可能不正确。

在实践中，大多数模型提供商和推理框架支持某种形式的响应预填充，这允许你在不修改工具定义的情况下约束动作空间。

Function calling通常有三种模式

* Auto – The model may choose to call a function or not. Implemented by prefilling only the reply prefix: <|im\_start|>assistant
* Required – The model must call a function, but the choice is unconstrained. Implemented by prefilling up to tool call token: <|im\_start|>assistant<tool\_call>
* Specified – The model must call a function from a specific subset. Implemented by prefilling up to the beginning of the function name: <|im\_start|>assistant<tool\_call>{"name": “browser\_

通过这种方式，我们通过直接mask token logits来约束动作选择。例如，当用户提供新输入时，Manus必须立即回复而不是执行动作。我们还有意设计了具有一致前缀的动作名称——例如，所有与浏览器相关的工具都以browser\_开头，命令行工具以shell\_开头。这使我们能够轻松确保agent在给定状态下只从特定工具组中进行选择而 without using stateful logits processors.

**Use the File System as Context**

现在LLM一般支持128K token或更大的context windows，但在构建agent时，通常不够，且带来一些痛点；

* Observations can be huge尤其是当agent与网页或PDF等非结构化数据交互时。很容易超出上下文限制。
* Model performance tends to degrade超过一定的上下文长度后，即使技术上支持该窗口大小，模型效果也会下降
* Long inputs are expensive即使使用prefix caching。

为了解决这个问题，很多agent系统使用了**context truncation（截断） or compression（压缩）**，但过度的压缩不可避免地导致信息丢失。这个问题是根本性的：agent本质上必须根据所有先前状态预测下一个动作——而你无法可靠地预测哪个观察结果可能在十步之后变得至关重要。从逻辑角度看，任何不可逆的压缩都带有风险。

Manus的方案是将**file system as the ultimate context**，这样的好处是大小不受限制，天然持久化，并且agent可以直接操作。模型学会按需写入和读取文件——不仅将文件系统用作存储，还用作结构化的外部记忆。

图示

AI 生成的内容可能不正确。

我们的压缩策略始终设计为可恢复的。例如，只要保留URL，网页内容就可以从上下文中移除；如果沙盒中仍然保留文档路径，则可以省略文档内容。这使得Manus能够缩短上下文长度，而不会永久丢失信息。

**Manipulate Attention Through Recitation 通过复述操控注意力**

在处理复杂任务时，manus倾向于创建一个todo.md文件——并在任务进行过程中逐步更新它，勾选已完成的项目。

这不仅仅是可爱的行为——这是一种**操控注意力的刻意机制**。

图示

AI 生成的内容可能不正确。

通过不断重写待办事项列表，Manus将其目标复述到上下文的末尾。这将全局计划推入模型的近期注意力范围内，避免了"丢失在中间"的问题，并减少了目标不一致。实际上，它使用自然语言来使自己的注意力偏向任务目标——而不需要特殊的架构变更。

**Keep the Wrong Stuff In**

代理会犯错。这不是bug——这是现实。语言模型会产生幻觉，环境会返回错误，外部工具会出现异常行为，意外的边缘情况随时都会出现。在多步骤任务中，失败不是例外；它是循环的一部分。

一个常见的冲动是隐藏这些错误：清理痕迹，重试操作，或重置模型的状态并将其留给神奇的"温度"。这感觉更安全，更受控制。但这是有代价的：擦除失败会移除证据。没有证据，模型就无法适应。

图示

AI 生成的内容可能不正确。

根据我们的经验，改善agent行为最有效的方法之一出奇地简单：**将错误的尝试保留在上下文中**。当模型看到一个失败的action——以及由此产生的观察结果或堆栈跟踪——它会隐式地更新其内部信念。这会改变其先验，降低重复相同错误的可能性。

事实上，我们认为 **error recovery**是true agentic behavior的最明显指标之一。然而，在大多数学术工作和公共基准测试中，这一点仍然代表性不足，它们通常关注理想条件下的任务成功。

**Don't Get Few-Shotted**

[Few-shot prompting](https://zhida.zhihu.com/search?content_id=260871316&content_type=Article&match_order=1&q=Few-shot+prompting&zhida_source=entity) 是提高LLM输出的常用技术。但在代理系统中，它可能会以微妙的方式适得其反。

语言模型是优秀的模仿者；它们模仿上下文中的行为模式。如果你的上下文充满了类似的过去行动-观察对，模型将倾向于遵循该模式，即使这不再是最优的。

这在涉及重复决策或行动的任务中可能很危险。例如，当使用Manus帮助审查20份简历时，代理通常会陷入一种节奏——仅仅因为这是它在上下文中看到的，就重复类似的行动。这导致偏离、过度泛化，或有时产生幻觉。

图示, 文本

AI 生成的内容可能不正确。

解决方法是增加多样性。Manus在行动和观察中引入少量的结构化变化——不同的序列化模板、替代性措辞、顺序或格式上的微小噪音。这种受控的随机性有助于打破模式并调整模型的注意力。

换句话说，不要让自己陷入few-shot的窠臼。你的上下文越单一，你的智能体就变得越脆弱。

**Conclusion**

上下文工程仍然是一门新兴的科学——但对于智能体系统来说，它已经是必不可少的。模型可能变得更强大、更快速、更经济，但再多的原始能力也无法替代对记忆、环境和反馈的需求。你如何塑造上下文最终决定了你的智能体的行为方式：它运行的速度、恢复的效果以及扩展的范围。

**关于Context Engineering其他的一些讨论**

1. 上下文越大，性能越差。[https://research.trychroma.com/context-rot](https://link.zhihu.com/?target=https%3A//research.trychroma.com/context-rot)
2. @karpathy 在五月对Context Engineering作出了非常精彩的总结。

文本

AI 生成的内容可能不正确。

相比于“提示工程”，“上下文工程”更值得点赞。

人们通常将提示与日常使用大语言模型（LLM）时给出的简短任务描述联系起来。然而，在每一个具备工业强度的大语言模型应用中，上下文工程是一门精妙的艺术与科学，它要为下一步在上下文窗口中填充恰到好处的信息。之所以说是科学，是因为要做好这一点，需要**涉及任务描述与解释、少量示例、检索增强生成（RAG）、相关（可能是多模态的）数据、工具、状态与历史记录、压缩等方面…… 信息太少或形式不对，大语言模型就没有最佳性能所需的合适上下文。信息过多或太过无关，大语言模型的成本可能会上升，性能可能会下降**。要把这做好绝非易事。而之所以说是艺术，是因为它围绕着人类精神层面的大语言模型心理有着引导性的直觉。

除了上下文工程本身，一个大语言模型应用还必须做到：

* 恰当地将问题分解为控制流
* 恰当地填充上下文窗口
* 向合适类型和能力的大语言模型发送调用请求
* 处理生成 - 验证的用户界面/用户体验流程
* 还有更多——防护措施、安全性、评估、并行处理、预取等等……

因此，上下文工程只是新兴的一层复杂且重要的软件中的一小部分，这层软件将单个大语言模型调用（以及更多内容）协调成完整的大语言模型应用。“ChatGPT 包装器”这个说法已经过时，而且大错特错。