Latent Space - Context Engineering for Agents - Lance Martin, LangChain

Alessio：

大家好，欢迎来到最新一期播客。我是来自 Kernel labs 的 Alessio，此次与 SmoI AI 的创始人 Swyx 一同参与直播。

Swyx：

大家好！我们非常高兴地邀请到 Lance Martin —— 他来自 LangChain。

Lance：

能来到这里真的很棒。我是这个播客的长期听众，现在终于有机会参与进来，感觉太好啦。

Swyx：

是啊，你其实早就和我们有交集了 —— 你在其中一场 AI 峰会上做过分享，而且显然我们和 LangChain 的关系也很密切。

最近我注意到你还做了很多教程。我记得你做过类似 “R1D Researcher” 的

内容，那是个挺受欢迎的项目，还有 “Basic Ambient Agents”。但真正让我觉得是时候邀请 Lance Martin 来录播客了的，是你最近在 context engineering（上下文工程） 方面的研究 —— 这是个挺新的领域。你是怎么开始研究这个方向的?

Lance:

有意思的是，行业热词往往是在人们有共同经历时才会出现。我觉得今年年初很多人就开始构建智能体（agents）了，还把今年称作智能体之年（the Year of Agents）。但实际情况是，当你真的去搭建智能体时，会发现整个流程的循环太长了。表面上看步骤很简单，但要让它顺畅运行其实特别难。其中一个核心问题就是智能体的上下文管理。

后来 Karpathy 发了一条推文，正式确立了 “context engineering（上下文工程）” 这个术语，还给出了一个不错的定义：**上下文工程是指 “为大型语言模型的下一步操作提供‘恰好合适’的上下文的挑战”**，这个定义对智能体领域非常适用。我觉得这个定义之所以能引起很多人的共鸣，是因为我自己在过去一年研究智能体的过程中，也有过同样的体会 —— 我在一篇关于 “构建 Open Deep Research” 的文章里也提到过这一点。

所以，“上下文工程” 这个术语能流行起来，本质是因为它捕捉到了很多人共同的经历 —— 大家都在做智能体，也都遇到了类似的问题，所以这个术语一

出现就被广泛接受了。

Alessio：

那你怎么界定 prompt engineering（提示词工程） 和上下文工程的边界呢？比如在你看来，提示词优化算不算上下文工程？我觉得很多人都有困惑 ——是不是我们在用一个新术语替代旧术语？这两者到底是什么关系？

Lance：

我认为**提示词工程其实是上下文工程的一个子集。**关键区别在于，当我们从 “对话模型（chat models）和对话交互” 转向 “智能体” 时，会发生一个很大的转变。

比如 ChatGPT 这类对话模型，人类输入的信息是最主要的输入源，所以人们会花很多时间打磨传给模型的 “提示词”。但智能体的情况更复杂：**智能体的上下文不仅来自人类，还来自智能体运行过程中的工具调用** —— 这些工具调用会不断产生新的上下文。

这正是我和很多人都观察到的核心挑战：当你搭建一个智能体时，要管理的不只是 “系统指令” 和 “用户指令”，还要管理在大量工具调用过程中，每一步产生的所有上下文。其实已经有不少相关的优质文章了，比如 Menace 就写过

一篇很棒的文章，聊到了 “上下文污染” 问题 —— 他们提到，一个典型的 Menace 任务需要约 52 次工具调用。Anthropic 的多智能体研究也给出了类似案例：他们提到，一个典型的生产级智能体需要多达数百次工具调用。

我第一次搭建智能体时就遇到了这种情况，相信很多人也有同感：一开始你听说 “智能体就是工具调用的循环”，觉得很简单，于是就动手搭建了。我当时在做 Open Deep Research，这个智能体的每次研究类工具调用都特别消耗 token。结果突然发现，要是用简单的 “工具调用循环” 逻辑，**这个深度研究智能体每次运行要消耗 50 万个 token**，成本大概是 1 到 2 美元。

这种经历应该很多人都有过 —— 搭建智能体的难点在于，如果你只是简单地把每次工具调用的上下文都 “直接传入”，很快就会碰到 LLM 的上下文窗口限制，这是最明显的问题。但 **Chroma 的 Jeff 在最近的播客里也提到，当上下文变长时，还会出现各种奇怪且特殊的故障模式。他还写过一份关于 “上下文衰减（context decay）” 的报告**，里面提到了这些问题。

所以，如果你搭建的是一个 “简单智能体（naive agent）”，就会同时面临两个问题：工具调用会产生几十到几百条上下文，这些上下文不仅会导致 LLM 的性能随上下文长度增加而下降，还会直接触发上下文窗口限制。正是这个痛点，催生了 “上下文工程” 这个新方向 —— 我们需要专门设计传给智能体的上下文。我在博客里总结的那些方法（借鉴了 Anthropic、Menace 的研究和我自己的经验），就是为了解决这个问题。

Swyx:

我刚好准备把一些相关资料放到屏幕上 —— 我们团队喜欢用视觉辅助来讲解。我们之前写过一篇编号 55 的博文，标题是《Thinking of Tools》，里面提到 “工具的作用之一就是获取上下文”。我认为，智能体可以通过工具来获取它需要的上下文 —— 只要你明确告诉它要这么做。另外，我还记得你写过一篇关于这个的博客，是吗？

Lance:

说起来也挺有意思的，我其实本来也想提这个 —— 我确实有一篇相关博客，但这个领域发展太快了，博客发完之后，我又在一次线下交流会上更新了一些内容。所以现在看交流会的材料会更合适，它相当于博客内容的 “超集”，不过我确实也写了博客。**只是博客和交流会之间只隔了两周，内容就已经有变化了 —— 这领域就是这么快。**对，就是这份交流会材料，我们按顺序讲吧？这样更容易跟上，而且内容也比博客更全面。

Alessio:

你之前提到了 “五个类别”，能具体定义一下吗？比如 “卸载（offload）”，我大概知道是什么意思，但能不能再详细说说？

Lance:

好的，我们来逐一梳理。首先，我之前提到的 “简单智能体（naive agent）” 是这样的：智能体进行多次工具调用，每次调用的结果都会传回 LLM，而且所有上下文都会被 “直接传入”。这样一来，**上下文窗口会迅速膨胀，因为工具调用产生的反馈会不断积累在消息历史里**。

Menace 团队分享过一个很好的观点：上下文卸载（offload context）非常重要且有用。**不要把每次工具调用的完整上下文都直接传回去，而是可以把它们 “卸载” 到其他地方** —— 他们提到可以卸载到磁盘（disk），也就是把文件系统（file system）当作 “外部化内存（externalized memory）”。比如，工具调用产生的上下文可能很消耗 token，你**不用把完整内容传回模型，而是把它们写到磁盘里，只传回去一个 “摘要” 或者 “URL” 之类的标识** —— 智能体知道需要的时候可以随时调取这些内容，但不会把原始上下文都塞进模型。

这就是 “卸载（offload）” 的核心逻辑：存储介质可以是文件系统，也可以是其他形式，比如 LangGraph 里的 “状态（state）” 概念，或者智能体运行时的 “状态对象（runtime state object）”。关键在于，不要把工具调用的所有上下文都塞进智能体的消息历史，而是存在外部系统里，需要时再调取 —— 这样能大幅降低 token 成本。

Alessio：

那关于 “卸载”，我有个疑问：要在上下文里保留多少 “最小化的摘要、元数据”，才能让模型知道卸载的上下文里有什么？比如做深度研究时，你显然会把整页内容都卸载掉，但怎么生成一个有效的摘要或简介，来说明文件里有什么呢？

Lance：

这其实是个非常关键且有趣的问题。我用自己做 Open Deep Research 的经历举个例子吧 —— 这个深度研究智能体我已经开发了大概一年，根据 “Deep Research Bench” 的数据，它目前是性能最好的深度研究智能体（至少在这个基准测试里是这样），表现还不错。当然，它不如 “Organized Deep Research”，但作为开源项目，它的性能已经很强了。

我在生成摘要时，会**用精心设计的提示词—— 我会提示摘要模型生成详尽的要点列表，把文档里的核心信息都列出来。**这样智能体后续就能判断是否需要调取完整上下文。所以，做摘要时，既要通过提示词工程确保摘要能准确还原核心信息，又要压缩内容 —— 但必须保证 LLM 能通过这些要点知道 “完整上下文里有什么”，这一点非常重要。

Cognition 也写过一篇很棒的博客聊这个话题，他们提到 “摘要生成其实值得花很多时间打磨”，所以我不想把这个步骤说得太简单。不过至少在我的实践里，

用精心设计的提示词让模型准确捕捉核心信息这个方法很有效。比如他们在博客里提到，甚至可以用 “微调模型（fine-tuned model）” 来做摘要 —— 当时他们聊的是 “智能体边界（agent boundaries）” 和 “消息历史摘要”，但这个逻辑同样适用于 “工具调用产生的大 token 量上下文的摘要”：核心就是让模型知道上下文里有什么。我花了很多时间做提示词工程，就是为了让摘要既能 “高还原度（high recall）” 地覆盖文档核心，又能大幅压缩内容。

Swyx：

我觉得 “压缩” 也是昨天那场线下交流会的核心发现之一 —— 就是 Chroma 主办的那场 Meetup。当时大家都提到 “要频繁压缩上下文”，因为不想触发上下文窗口限制。

而且我觉得 “卸载” 的重要性已经很明确了，应该是必做的步骤。另外还有个很有意思的观点：有人把 “卸载” 和 “多智能体” 联系起来，说为什么需要多智能体？因为不同角色的智能体可以分别压缩和调取不同的内容，而单个智能体没必要掌握所有上下文。

Lance：

你说得太对了！其实我之前想重点聊的另一个核心主题，就是 “多智能体的上下文隔离” —— 这一点和 Cognition 的观点其实是相通的，很有意思。不过

Cognition 其实是 “反对多智能体” 的，他们有几个主要论点。

其中一个核心是 “**给子智能体传递足够的上下文很难**”。他们花了很多时间研究 “子智能体间的摘要 / 压缩” —— 甚至用了微调模型来确保 “所有相关信息都能传递到位”。你看这份材料下面，他们展示的其实是一个 “线性智能体”，但即便在这种 “智能体边界” 处，他们也强调 “要谨慎处理信息的压缩和传递”。

Alessio：

对我来说，最大的疑问其实在 “编码（coding）” 场景 —— 这是我最常用的场景，但我还没搞清楚 “要向子智能体展示多少实现过程的信息”。比如，有个子智能体负责写测试，另一个负责其他任务，我需要向它们解释代码库是怎么发展到现在这个状态的吗？还是不用？另外，如果子智能体需要修改代码来完成任务，它只需要把测试结果传回主智能体（main agent）吗？还是应该把修改逻辑也告诉主智能体？

我觉得 “深度研究” 这个场景很清晰，因为它处理的是 “原子化的内容片段”，但当子智能体之间存在 “状态依赖（state dependency）” 时 —— 比如编码场景 —— 我就完全搞不清楚了。

Lance:

你这个问题其实正好击中了 “上下文隔离” 这个类别的核心。Cognition 有个论点我觉得很合理：**他们反对用子智能体，因为每个子智能体都会自主做决策，而这些决策可能会冲突**。比如，子智能体 1 负责 A 任务，子智能体 2 负责 B 任务，它们的决策逻辑可能矛盾，最后整合结果时就会出问题。

你说的编码场景确实容易出现这种 “决策冲突” —— 我也遇到过。我比较认可的一个观点是：**只在 “任务能清晰并行” 的场景用多智能体。**Cognition 的 Walden Yan 也经常聊这个，他提到了 “反向书写任务（reverse write tasks）” 的概念 —— 比如，如果每个子智能体负责 “最终解决方案的一个组件”，这种情况就很难，因为它们需要频繁沟通（就像你说的）。目前智能体之间的通信还处于早期阶段，但 **“深度研究” 场景就不一样了 —— 智能体只需要 “读取信息”**，也就是收集上下文，等所有 “读取任务” 完成后，再统一进行 “书写”。我发现这种模式在深度研究场景里特别好用，Anthropic 的报告也提到了这一点：他们的深度研究智能体（Deep Researcher）就是用 “并行子智能体” 收集研究信息，最后用 “一次性书写” 完成报告。

所以这里的关键在于：你要把 “上下文隔离” 应用到什么类型的问题上，结果会完全不同。编码场景可能难得多 —— 尤其是如果每个子智能体都要创建 “系统的一个组件”，它们的决策很可能隐含冲突，最后整合整个系统时就会出现很多问题。但研究场景只是 “收集上下文”，只需要 “提交步骤”，最后一次性书写 —— 这种模式就很顺畅。

这也是 Cognition 和 Anthropic 观点的核心分歧：Cognition 说 “不要用多智能体”，Anthropic 说 “多智能体很好用” —— 其实取决于你要用多智能体解决什么问题。这一点非常微妙且重要：“多智能体的应用场景” 和 “使用方式” 会极大影响效果。**我个人倾向于把多智能体用在‘易并行、只读（read-only）’的场景，**比如深度研究的上下文收集，最后再做一次性书写（比如写报告）。而编码智能体的情况更复杂 —— 有意思的是，Claude Code（Anthropic 的代码智能体）现在已经支持子智能体了，说明 Anthropic 认为这种模式是可行的，至少是可以尝试的。但我还是比较认同 Cognition 的观点：如果子智能体的任务需要高度协同，编码场景会非常棘手。

Swyx：

你这个对比解释得很清楚，我没什么要补充的。不过很有意思的是，他们需要针对不同场景设计不同的智能体架构 —— 不知道这种 “场景化架构” 是暂时的，还是会像 “the Bitter Lesson”（Richard S. Sutton 的文章）里说的那样，最后会趋同？对了，我们或许应该聊聊你提到的系统其他部分，里面有很多有趣的技术点。

Lance：

那我们聊聊 “传统检索” 吧。RAG 其实已经存在很多年了，甚至在这次 “大

语言模型（LLM）浪潮” 之前就有了。我发现一个很有意思的点：不同的代码智能体（code agents）在检索上的思路差异很大 —— 比如 Windsurf 团队从引擎设计（engine perspective）出发，分享了他们在 Windsurf 中如何做检索：他们会按精心设计的语义边界拆分代码块，把这些代码块嵌入向量（embedding），然后用经典的语义相似性向量搜索做检索；但他们还会结合“grep”，甚至构建 “知识图谱”，最后再对这些检索结果做 “排序整合” —— 这是一种典型的 “复杂多步骤 RAG 流程”。

但 Anthropic 的 Boris 却采取了完全不同的思路 —— 他多次提到，Claude Code 不做任何索引，只靠 “生成式检索”，用简单的工具调用（比如 grep）在文件里 “摸索搜索（poke around files）”，完全不用索引 —— 但效果显然非常好。

所以你看，不同代码智能体在 “RAG 和检索” 上的思路差异很大，这也成了一个有趣的新兴趋势：什么时候需要 “更复杂的索引”？什么时候用 “简单的生成式搜索” 就够了？

Swyx：

是啊，我们最近播客里有个爆火的片段，有人提到，可以 “完全不做代码索引，只靠生成式搜索”—— 这可能就是 “80/20 原则” 的体现：用 20% 的简单方法解决 80% 的问题，如果你需要更精准的结果，再加点复杂方法，但可能大

多数场景下根本不需要。

Lance：

对，我昨天还看到了帖子，说他们 “只做抓取（scrap），不做索引”。所以在上下文工程的检索领域，其实有很多有趣的权衡：你是要做经典的向量检索或语义检索，还是用传统的生成式搜索（仅靠基础文件工具）？

我其实自己做过一次基准测试，我当时研究的是，如何在所有语言文档中做检索。我设计了 20 个和 Line Graph 相关的编码问题，让不同的代码智能体通过检索文档来生成 Line Graph 代码。我测试了 Claude Code 和 Cursor，用了三种不同的文档检索方法：

第一种是 经典向量存储检索：我把所有文档（约 300 万 token）都导入向量库（vector store），做标准检索；
第二种是 文本文件 + 简单文件加载工具：更接近 “生成式搜索” —— 就是一个文本文件，里面包含所有文档的 URL 和简单描述，让代码智能体通过工具调用调取它感兴趣的特定文档；
第三种是 上下文填充（context stuffing）：把所有 300 万 token 的文档直接喂给代码智能体。

结果很有意思（当然这只是我的特定测试场景）：我发现 “带清晰描述的文本

文件” 效果特别好 —— 它其实就是个 Markdown 文件，列了所有文档的 URL 和内容简介，再给代码智能体配个调取文件的简单工具。实际效果是：代码智能体看到问题后，会说“我需要调取这个文档看看”“再调取那个文档看看”，然后就能生成正确代码。我现在一直用这种方法，个人完全不用向量检索或索引，就靠 “文本 + 简单搜索工具”，搭配 Claude Code—— 不过这是几个月前的测试了，领域发展太快，当时 Claude Code 在我的测试场景里表现比 Cursor 好，而且成本低很多（我是今年 4 月做的测试），所以从那以后我就一直用 Claude Code 了。

这也印证了 Boris 的观点：**给 LLM 配备 “基础文件工具访问权限”，再用文本帮它 “了解每个文件的内容”，效果会非常好 —— 而且比搭建索引更简单、更易维护**。这也是我的亲身经历。

Swyx：

我特别喜欢文本这种形式，而且经常用 —— 其实 Cognition 的 “Deep Wiki” 就是类似的东西。我给自己做了个 Chrome 插件：不管是哪个代码仓库（包括你的），我一点就能打开 Wiki—— 它本质上就是文本，但可读性更好。

Lance：

对，这是个很好的例子！我觉得这种方法很值得推广：拿一个代码仓库，把它整

理成 “易读的文本格式”。我还发现，“用 LLM 生成文档描述” 能大幅提升效果 —— 我 GitHub 上有个小项目，能自动遍历文档，把每个页面传给 GPT（或其他 LLM），让它生成高质量摘要，再整合成文本文件。效果特别好，文本 里的摘要都是 LLM 生成的…… 我找找这个项目在哪，别吐槽我的仓库乱啊，这是个新项目，不是那个老的…… 我仓库太多了，往上翻翻……

Alessio：

你项目确实多，你起名字还是很厉害的，尤其是在 AI 领域的几个命名都很准。

Lance：

应该是这个 —— 这个仓库没什么人关注，但我自己用得很爽。它逻辑很简单：你指定一个文档路径，它会自动遍历所有页面，把每个页面传给 LLM，让 LLM 写摘要，最后整合成文本文件。我把这个文本喂给 Claude Code 后发现，Claude Code 特别擅长根据摘要判断该调取哪个页面—— 比如根据问题，它能精准定位需要从哪个 URL 调取文档。我现在一直用这个工具，比如生成新文档的文本，或者处理 Line Graph 和其他常用库的文档。唯一要注意的是：文本里的描述质量很重要，因为 LLM 要靠这些描述判断该读什么。这就是个简单又实用的小工具。

Alessio：

我们之前有个客户提到个 “项目文档专用 MCP”，用的人特别多。你试过吗？

或者还见过类似的工具？就是能自动处理这些流程的。

Lance：

有意思的是，我们自己有个 MCP 服务器—— 它的作用就是给 Claude Code 这类智能体提供文本文件和简单的文件搜索工具。

Claude 现在有内置的 “调取工具（fetch tools）”，但我当时做这个服务器的时候还没有，所以它其实就是个简单的 MCP 服务器，把文本文件暴露给 Claude Code 这类智能体，名字叫“MCP Doc”—— 特别简单的小工具，我一直用，特别好用。你只要把它和你所有的文本文件关联起来就行。

Alessio：

但 MCP Doc 的服务器是能搜索文档的—— 这就形成了循环。

不过我想问：“是不是一个项目应该配一个服务器？” 还是说 “最后会出现一个‘元服务器（meta server）’”？而且我觉得，当你从单纯的工具调用和服务器转向采样（sampling）、提示词和资源管理时，其实可以把很多 “提取工作（extraction）” 放在服务器里做 —— 这又回到了你之前说的 “上下文工程”：或许可以把复杂工作放在服务器端，只把最终需要的信息传入上下文。但现在这方面好像还处于早期阶段。

Lance:

你这个点特别有意思 —— 我和不少人聊过这个。我发现**把提示词存在 MCP 服务器里其实很重要，尤其是 “要告诉 LLM / 代码智能体 ‘怎么用这个服务器’”**。所以我会给不同项目配不同的服务器，每个服务器里放项目专属的提示词，有时候还会放项目专属资源。我其实不介意按项目分服务器—— 每个服务器里放该项目所需的上下文和提示词，这样更精准。

Swyx:

是啊，很多人可能没注意到 MCP 协议的一些功能 —— 其实它里面是支持 提示词（prompts）的，这可能是最早的功能之一，但被严重低估了。很多人觉得 MCP 只是 “交互入口”，但其实它还有很多功能，比如 “采样（sampling）”。

Lance:

对，“采样” 也被低估了！

而且 “提示词” 其实特别重要 —— 比如我们那个 Line Graph 文档的 MCP Doc 服务器，我发现给它加提示词会更好用。一开始我把提示词写在 README 里，比如怎么提示服务器调取文档，但后来发现提示词应该直接存在服务器里—— 这样就能把 LLM / 代码智能体使用服务器所需的提示词和服务器本身绑定。

我之前发现，很多人用我们的 MCP Doc 服务器时觉得不好用，其实是提示词没写好——但这不该是用户的问题，提示词应该存在服务器里，让代码智能体能直接获取。所以**检索其实是个大主题，它早于下文工程这个新术语，但检索显然是上下文工程的重要子集**。

Swyx：

在结束检索这个话题前，我还想问问有没有其他趋势——我最近在关注 ColBERT 和“延迟交互”的概念，不知道你们有没有研究过？它有点像“全检索（full retrieval）”和“全预索引（full pre-indexing）”之间的中间状态，我称之为“两阶段索引（two-phase indexing）”。你有什么看法吗？

Lance：

我个人没怎么深入研究 ColBERT，只简单试过几次，所以没太多发言权，抱歉。

Alessio：

没关系，那我们继续往下聊吧。

Lance：

那我们简单聊聊 “上下文精简（reducing context）” 。其实每个人都有过类似经历：用 Claude Code 时，会碰到上下文窗口用了 95%的提示，Claude Code 会自动触发压缩—— 这就是需要上下文精简的典型场景，很直观。

但有个有趣的观点：除了接近上下文窗口时精简，其实在工具调用边界（tool call boundaries）做压缩或剪枝也很合理。我在 Open Deep Research 里就用了这个方法，Hugging Face 上有个很有意思的 Open Deep Research 实现 —— 它不是编码智能体，但借鉴了编码智能体的逻辑：它不用 “工具调用”，而是把 “相邻的代码块” 传到编码环境里运行，然后他们提出 “要对代码块做摘要 / 压缩，只把有限的上下文传回 LLM，而把 ‘token 消耗大的原始工具调用结果’ 存在环境里”—— 这是另一个例子。Anthropic 的 “多智能体研究智能体” 也会对研究发现做摘要。

所以剪枝其实很常见，也很直观。但 Menace 团队提出了一个有意思的 “反观点”：他们警告剪枝有风险，尤其是不可逆的剪枝——Cognition 也提到了这一点，他们说**摘要生成必须非常谨慎，甚至需要用微调模型来确保效果**。这也是为什么 Menace 认为一定要做上下文卸载（context offloading）：**工具调用后，先把结果卸载到磁盘（这样原始数据还在），然后再做剪枝 / 摘要** —— 就像你之前问的要传什么信息回 LLM，但原始上下文还在，不会因为有损压缩或 “有损摘要丢失信息。所以摘要 / 剪枝时要警惕信息丢失，这是个重要的注意事项。

Swyx：

其实这个问题大家有分歧 —— 我要提一下 “错误路径（wrong paths）”：Menace 说不要删错误记录，这样智能体能从错误中学习；但另一些人说一旦出错，智能体可能会一直沿着错误路径走，所以必须‘回溯（unwind）’或‘剪枝错误记录’，明确告诉它‘别再这么做了，试试别的’。你有什么看法吗？我昨天就遇到有人反驳 Menace 的观点。

Lance：

这个话题确实很有意思。Drew Bernick 写过一篇很棒的博客，聊上下文故障模式（context failure modes）。

Swyx：

他写过好几篇这类博客。

Lance：

其中一篇是 “上下文污染（context poisoning）”。他提到，Gemini 在技术报告里也提过这个问题：**模型可能会产生幻觉，而这个幻觉会留在智能体的消息历史里，像污染一样引导智能体偏离正确方向**。他还引用了 Gemini 技术报告里的一个具体例子：Gemini 在玩《精灵宝可梦》时出现幻觉，进而影响后续决

策 —— 这是要警惕上下文里的错误，避免污染的观点。

另一种观点就是你说的 “保留错误记录，让智能体学习修正” —— 比如智能体调用工具出错了，把错误留在上下文里，它下次就能根据错误调整。这两种观点其实有张力。

我注意到 Claude Code 会保留错误记录—— 比如我用它时，会看到错误日志被打印出来，它会根据这些日志修正。我自己的实践也发现，对于工具调用错误，保留错误记录其实很有用，我个人不会刻意剪枝。而且从技术角度看，“判断什么时候该从消息历史里剪枝错误” 其实很麻烦，会增加智能体框架（agent scaffolding/harness）的逻辑复杂度 —— 我不太喜欢这种选择性剪枝消息历史的做法，会增加维护成本。

Swyx：

这其实就是 “精准度与召回率” 的权衡，只不过在智能体工作流的上下文管理里重新体现了。

Lance：

完全正确。

Swyx：

说到 Drew Bernick，他确实是个很厉害的作者，还创造了 “上下文工程法则（context engineering law）” 之类的概念。你对他的观点有什么特别认同或反对的吗？

Lance：

我给你们看个有意思的 —— 打开他的博客，我在交流会上也提到过这个。他引用了 Stewart Brand 的一句话，还挺搞笑的：“如果你想知道未来在哪里孕育，就看人们在发明新语言、律师在聚集的地方（If you want to know where the future is being made, look forward to language being invented, and lawyers are congregated）”。这句话其实是在解释为什么热词会出现—— 他也是第一个让我意识到上下文工程这种术语能流行，是因为它捕捉了很多人的共同经历的人。

你往下翻会发现，他其实写了一整篇关于如何创造热词的博客，但核心观点是：**成功的热词（buzzwords）都是捕捉了很多人共同经历的词—— 它们不是凭空出现的。比如 “上下文工程” 能火，是因为很多人都在做智能体，也都遇到了上下文管理的痛点**，看到这个术语就会觉得 “对，这就是我遇到的问题”。

上下文工程其实是很多人都在做的事，只是大家没找到一个统一的词，直到这个

术语出现，大家才产生共鸣：“对，这就是我在做的事”。所以这其实是个很有意思的 “小插曲”。

Swyx：

我完全同意 —— 我当初创造 “AI 工程师（AI Engineer）” 这个词，也是因为类似的原因：当时企业想招 “懂 AI 的工程师”，但又不想局限于传统机器学习（ML）工程师；而工程师也想加入 “尊重 AI 工作价值” 的公司，同时摆脱 “传统 ML 工程师” 的包袱 —— 很多 AI 工程师甚至不用 PyTorch，只要会写提示词和做常规软件工程就行。在 “前沿模型大多来自封闭实验室（closed labs）” 的当下，我觉得这个定位是合理的。

Lance：

有个反例能印证这个观点：如果有人创造的 “语言” 不能引起共鸣，不能捕捉共同经历，就会很快消失。也就是说，热词其实是和生态同频的 —— 它们能流行，是因为应该捕捉共同经历；如果有人强行创造一个没有共鸣的术语，根本火不起来。

Swyx：

你有过这种 “创造术语却没人用” 的经历吗？

Alessio：

我最不擅长起名字了，但 Lance 你很厉害，你在 AI 领域的几个命名都很准。

Swyx：

没错！好啦，我们还是回到 “上下文工程” 吧 —— 抱歉刚才有点跑题了。

Lance：

没事，跑题的内容也很有价值，覆盖了很多核心主题。我们可以再简单聊一个点，然后聊聊 “the Bitter Lesson” 之类的话题收尾。回到那份表格（注：指之前提到的交流会材料表格），我想特别提一下 Menace 团队，他们有个观点很有意思。

我们之前聊了 “卸载（offloading）” “上下文精简（reproducing context）” “检索（retrieval）” “上下文隔离（context isolation）” —— 这些都是很常用的核心方法。

我想重点说 Menace 的 “缓存（caching）” 观点：很多人第一次搭建智能体时，都会被智能体循环运行时，每次都要把之前的工具调用结果传一遍这件事震惊 —— 每次循环都要重复传递这些 token，成本很高。Menace 提出缓存之前的消息历史（caching prior message history）是个好办法，我虽然没亲自试过，但逻辑上很合理：缓存能大幅降低延迟和成本。

Swyx：

但大部分 API 不是会自动缓存吗？比如用 OpenAI 的 API，应该会自动命中缓存吧？

Lance：

我其实不确定 —— 比如你搭建智能体时，每次都会把消息历史传回去，而 API 本身通常是无状态的。

Swyx：

不同服务商的 API 在这方面差异很大，但尤其是“响应式 API（response API）”—— 如果你的状态从不修改，那对用户和你来说都很方便：如果你认为不该压缩对话历史，那没问题；但如果你需要修改状态，就麻烦了。不过像 OpenAI 的 API，现在已经不用加特殊请求头（header）了，缓存会自动生效。

Lance：

这点很重要 —— 我以前用 Anthropic 的 API 时，会明确加缓存请求头（caching header），但现在可能他们已经默认开启缓存了，这当然很好。不过 Menace 提的缓存观点还是值得关注。

Swyx：

是啊，现在要跟上 API 的更新太难了 —— 你得关注所有人的推特，读所有文档，才能不落后。

Lance：

不过 API 默认支持缓存确实是好事。我之前用 Anthropic 的显式缓存请求头，但现在如果默认缓存，就省了很多事。但有个重要且容易被忽略的点：**缓存解决不了 “长上下文问题” —— 它能解决延迟和成本问题，但如果你的上下文有 10 万个 token，不管有没有缓存，LLM 还是要处理这么长的上下文。**我之前在 Anthropic 的交流会上问过这个问题，他们提到 “上下文衰减（context decay）的问题，不管有没有缓存都会存在” —— 所以缓存不能解决 “长上下文导致的性能下降”，只能优化成本和延迟。

Swyx：

是啊，我还在想还有什么能缓存 —— 这其实涉及 “厂商锁定”：理想情况下，你希望能在多个服务商之间切换，但缓存本身是个难题。所以如果你能运行自己的开源模型，就能完全控制缓存，否则只能用服务商提供的 “半成品方案”。你说得对。

Swyx：

好啦，关于 “上下文工程” 的核心内容我们差不多都聊到了。你从昨天的交流会里还有什么其他感悟，或者有什么问题想聊吗？

Alessio：

没有了，我昨天最大的感悟是 “压缩质量”—— 有张图表显示，用“自动压缩功能”，结果和 “不压缩” 差不多，而且 “之前指令的质量” 也会受影响。Jeff（Chroma 团队成员）做的图表显示，“人工精选压缩（curated compaction）”的效果是 “自动压缩” 的两倍，但问题是 “怎么做到人工精选压缩？” 我觉得我们可以写一篇后续博客聊聊这个 —— 这对我来说很有意思，尤其是编码智能体的压缩。

感谢你聊了这么久！比如 “深度研究” 场景，拿到报告就完事了，但 “编码”场景需要持续构建—— 我发现，即使是做 “代码修改” 这类任务，保留 “之前的历史” 对模型也有帮助，知道为什么做这些决策的模型表现更好。但如何用更省 token 的方式提取这些决策信息，目前还不清楚。我没有答案，但希望听众里有人能研究这个方向。

Lance：

你这个点说得特别好 —— 这其实和 Cognition 的 Walden Ganz 的观点一致：“摘要 / 压缩步骤绝非小事，必须谨慎对待”。有时甚至用 “微调模型” 做编码场景的摘要，可见他们在这个步骤上花了很多功夫。Menace 也提到，**每次剪枝 / 压缩 / 摘要时，都要警惕信息丢失，所以他们会先把原始数据卸载到文件系统，确保能找回，再做精简**。所以 智能体搭建中，压缩是有风险的，这一点需要重点提醒。

Swyx：

我还想到一个点：之前大家都很关注 “记忆（memory）”，现在又聊 “上下文工程” —— 我觉得这两者有点像换了个说法。你觉得 “记忆” 和 “上下文工程” 有本质区别吗？比如你们最近重新推出的工具，本质上也是一种上下文工程吧？有没有哲学层面的质的差异？

Lance：

这个问题很好，我其实会从两个维度看：“写入记忆（writing memories）” 和 “读取记忆（reading memories）”，以及 “两者的自动化程度（degree of automation）”。

最简单的例子是 Claude Code—— 我很喜欢它的设计：“读取记忆” 时，它每次启动都会自动加载你的 GitHub 仓库；“写入记忆” 时，需要用户手动指

定我要把这个保存到记忆里，然后它会写入 GitHub 的某个文件。从读写自动化程度来看，它差不多是 “0/0” —— 非常简单，完全符合 Boris 所说的 “超级简单理念，我其实很喜欢这种设计。

另一个极端是 “全自动记忆” —— 比如有些智能体会在后台自动判断 “什么时候写入记忆” “什么时候读取记忆”。我记得有个很棒的演讲，虽然主题不是记忆，但提到了一个记忆检索失败的案例：用户想要生成某个场景的图片，结果模型自动读取了用户所在位置（Half Moon Bay，半月湾），并把位置信息放进了图片里 —— 用户其实并不想要这个，这就是记忆检索失控的例子。OpenAI 的 ChatGPT 在记忆功能上也花了很多时间，但效果仍不理想，可见全自动记忆有多难。

我的观点是：**“写入记忆” 的难点在于判断什么时候该写，而 “读取记忆” 在大规模场景下其实就是“检索（retrieval）”** —— 比如 “大规模记忆检索” 本质上就是 “检索”。所以我会把 “记忆” 的读取部分看作检索的一种特定场景。

Swyx：

对，它是 “特定上下文下的检索” —— 比如 “检索过去的对话”，这和 “检索知识库” “检索公开网页” 不同。顺便说一句，你网站上好像有句话就是这个意思，之前有人指出来过。

Lance:

完全正确！这里有个微妙的点：我不知道 OpenAI 的 “记忆工具” 背后具体是怎么实现的，但很可能是索引用户的过去对话，用向量搜索或其他方法做检索 —— 所以从这个角度看，**复杂的记忆系统其实就是复杂的 RAG 系统**，和我们之前聊的 Windsurf 的多步骤 RAG 流程类似。所以我会把 “记忆（至少是读取部分）” 看作 “检索的一种”。Claude Code 的方法其实很简单检索就是 ‘每次启动时自动加载仓库’，简单但效果很好。

Swyx:

没错！我还想强调 “记忆的语义分类”：情景记忆（episodic）、语义记忆（semantic）、程序记忆（procedural）和背景记忆（background memory） —— 我们之前做过一期关于 “睡眠时计算（sleep time compute）” 的播客，如果你在做 “长期运行的环境智能体（ambient agents）”，就会遇到这类 “记忆相关的上下文工程”，而这在传统的上下文工程讨论中还没有涉及。

Swyx:

而且我觉得，经典的上下文工程讨论里，还没有涵盖这些 “记忆分类” 的内容。

Lance:

你说得对。我之前开过一门 “搭建环境智能体” 的课程，还做了一个邮件管理系统—— 我发现 “记忆” 和 “人机协同（human-in-the-loop）” 特别搭。比如我的邮件助手，它本质是个管理邮件的智能体，每次发送邮件前，我都可以暂停它，手动修正—— 比如调整邮件语气，或者直接修改工具调用的参数。每次修正后，这些用户反馈都可以被写入记忆。

Swyx:

我就是这么做的！

Lance:

其实 “记忆 + 人机协同” 是个非常好的组合 —— 当你用人机协同修正智能体时，这些修正可以被捕捉到记忆里。我那个邮件助手就是这么设计的：它会用 LLM 分析我做的修改，对比之前的指令，然后自动更新记忆里的指令。这种窄范围的记忆使用—— 比如只捕捉用户偏好 —— 在环境智能体搭建中非常简单有效，我很喜欢这种方式。

Swyx: 36:45

你在 GitHub 上能找到那门课程，而且你们已经就这些主题做过很多次分享了。

Lance：

没错。但我还是想强调："记忆该什么时候用" 其实很容易混淆，而 "人机协同场景" 是记忆的一个非常清晰的应用场景 —— 因为 "人机协同的修正" 是更新智能体记忆、捕捉用户偏好的绝佳时机，能让智能体逐渐 "变聪明"。比如我的邮件助手就是这样，有从业者也说过他用类似系统管理所有邮件 —— 他作为 CEO 邮件很多，我邮件少，但这个系统对我也很有用。所以 "记忆 + 人机协同" 是个很棒的组合。

Swyx：

是啊，我之前也试过用邮件系统，但我还是离不开 Superhuman。好吧，关于上下文工程的内容我们差不多都覆盖到了，很棒。最后能不能聊点 "the Bitter Lesson" 相关的内容收尾？

Lance：

这个话题很有意思，我也想听听你们的看法。之前 OpenAI 的 Yang Wang Cheng 做过一个很棒的演讲，聊 "AI 研究中的苦涩教训 (the Bitter Lesson in AI Research)"。核心观点是，相同成本下，计算能力每 5 年翻 10 倍—— 机

器学习的历史已经证明，抓住这种‘计算缩放（scaling）’的趋势，是最重要的事。尤其是归纳偏置少、更通用、依赖更多数据和计算的算法，往往比手工调特征、内置归纳偏置的算法表现更好。**简单说就是：让机器用更多数据和计算自己学习‘如何思考’，比教机器我们如何思考更好——这就是苦涩教训的核心。**

他还有个微妙的论点：在任何时间点，当你做研究时，为了在当前计算能力下达到目标性能，通常需要添加一些结构（structure）；但随着时间推移，这些结构会成为进一步发展的瓶颈。他的幻灯片里也展示了这一点：在低计算量区间添加更多结构（比如更多建模假设、更多归纳偏置）比少结构更好；但随着计算量增加，少结构、更通用的算法会胜出。

他的建议是：**当前计算能力下，添加必要的结构让系统工作，但要记得以后去掉这些结构**——很多人往往会忘记后续去掉结构。我把这个观点和上下文工程联系起来：你看这张图表，这是我过去一年搭建 Open Deep Research 的历程：

一开始，我用的是高度结构化的研究流程，甚至不用工具调用——因为当时大家都觉得工具调用不可靠。我把研究该如何进行的假设都嵌进了系统：分解问题为多个部分，并行处理，最后整合成报告。一开始，这个流程比当时的智能体更可靠，但随着模型能力快速提升，这个结构很快就成了瓶颈——我没法利用 MCP 的普及，也没法利用工具调用越来越可靠的趋势。

后来我转向了智能体架构，去掉了很多结构，允许工具调用，让智能体自己决定

研究路径—— 这正好印证了 Yang Wang Cheng 在斯坦福演讲里的观点：**要不断重新评估‘基于当前模型能力，我的假设是否还成立’**。我之前还犯过一个错：让每个子智能体写报告的一部分”—— 这又回到了我们之前聊的子智能体隔离问题：子智能体之间沟通不畅，写出来的报告片段很零散，这正是 Alessio 提到的 “多智能体的核心挑战”。所以我后来去掉了 “子智能体独立写作”，改成 “最后一次性书写报告”—— 这就是现在的 Open Deep Research 版本，性能很好，而且是开源的。

我们甚至用 GPT-5 做了一些测试，结果很强 —— 模型一直在进步，而我们的开源系统能 跟上模型进步的浪潮。我自己的经历其实就是 “the Bitter Lesson” 的亲身实践：2024 年初，我搭建的结构化系统在当时很可靠，但随着模型能力提升，它成了瓶颈，我不得不两次重构系统，推翻之前的假设，才能利用模型的新能力。

所以我想强调：在指数级提升的模型之上搭建应用，其实很难。Anthropic 的 Boris 在聊 Claude Code 时，提到了“基于苦涩教训（the Bitter Lesson）的设计”—— 他说 Claude Code 之所以简单且通用，就是因为这个原因：他们希望给用户无束缚的模型访问权限，而不是用大量脚手架限制用户。

他的核心观点是 “苦涩教训的推论：模型周围的 ‘通用型组件（general things）’ 往往会胜出”。所以搭建应用时，要添加当前必需的结构让系统工作，但密切关注模型进步，及时去掉瓶颈结构—— 这是我的核心收获。Yang Wang Cheng

的演讲真的值得所有人听，里面很多观点都适用于工程实践。

Alessio：

我觉得这和 “传统企业采用 AI 的困境” 很像：传统企业会 “把 AI 嵌入现有工作流”，因为他们已经有成熟的流程和结构，AI 能让流程更好用；但 “AI 原生产品（AI-native products）” 会等模型能力足够强后再入场，不用 “去掉旧结构”—— 比如 Cursor 和 Windsurf 比 VS Code 更适合 AI 编码，因为它们不用 “改造旧流程”；Cognition 也是如此，从一开始就没把 “智能体” 当作 “现有工具的补充”，而是直接做 “智能体原生设计”。

现在市场上有个趋势：前 2.5 年，大家纠结是把 AI 嵌入现有工作流，还是重构工作流—— 当时模型能力不够，重构工作流效果不好；但现在模型能力已经过了那个临界点。现在应该从少结构化开始搭建。

Lance：

你这个例子太贴切了！再看这张图表，还有个有趣的点：“早期模型阶段，结构化方法效果更好”。Anthropic 的创始人 Jared Kaplan 几周前在创业学校的演讲里也提到：**有时候，搭建‘目前还不完美的产品’是个好策略，因为模型会指数级进步，最终会‘解锁’产品的价值**——Cursor 就是这样：一开始它并非完美，但随着 Claude 3.5 的发布，它突然就 “爆发了”，正好赶上模型能力

追上产品需求的节点。

但在早期模型阶段，你很容易被结构化方法效果更好的表象欺骗，以为这个结构就是对的，直到模型能力追上，才发现结构成了瓶颈。这就是 “结构化方法” 和 “少结构方法” 的张力：结构化方法会在早期领先，但少结构方法会在模型能力提升后反超。

Swyx：

你这张图表和 Windsurf 的图表特别像 —— 我得调出来给大家看，因为我参与了这张图表的撰写。是不是很像？先是一个平台期（ceiling），然后突然爆发（boom），增长放缓 —— 这其实是 “苦涩教训” 在企业场景里的体现。完全一样！

Swyx：

对我来说，图表的线条固然重要，但要点（bullet points）才是核心 —— 理解了这些要点，就能从别人的错误中学习。很多人都在这些要点上花了功夫，对吧？好啦，最后有个犀利问题：你觉得 Line Graph 和 “苦涩教训” 的理念契合吗？显然你们团队是了解 “苦涩教训” 的，所以这应该不是意外，但我觉得让结构容易拆解特别重要 —— 如果你相信 “苦涩教训”，就会这么做。

Lance：

这个问题特别关键，我在博客结尾也聊到了这一点。这里有个微妙的区别：当人们谈论 “框架（frameworks）” 时，其实指的是两种不同的东西，很多人反对的是智能体抽象（agent abstractions），而非底层编排框架（low-level orchestration frameworks）。

比如 Anthropic 做过一个很棒的演讲，他们内部搭建了一个叫 Roast 的编排框架 —— 本质上和 Line Graph 一样，就是提供 “可组合的底层构建块”，没有 “预设状态判断（no judges state）”，你可以用这些构建块搭智能体、搭工作流。我不反感这种框架 —— 底层构建块很容易拆解和重构。比如我用 Line Graph 搭 Open Deep Research 时，先搭了工作流，后来拆了重构成智能体，这些底层构建块（节点、边、状态）很灵活。

但我理解大家对框架的顾虑 —— 很多框架会提供 “from framework import agent” 这种 “智能体抽象”，这才是麻烦的地方：你不知道这个抽象背后藏了什么逻辑。很多人反对框架，本质上是反对不透明的抽象，我完全认同这一点。我自己也不喜欢智能体抽象，因为我们还处于智能体的早期时代 —— 抽象会掩盖细节，如果你用抽象搭 Open Deep Research，当模型能力提升时，你根本不知道该怎么拆解重构。

所以我的立场是：警惕抽象，但不排斥底层编排框架 —— 只要框架提供的是可

自由组合的节点（nodes），而不是“黑箱抽象”，就很有价值。我用 Line Graph，是因为它能提供 “ checkpoint（检查点）” “状态管理” 这些底层功能，很实用。很多客户喜欢 Line Graph，也不是因为它的 “智能体抽象”，而是因为它的底层灵活性。所以关于框架的争议，其实应该聚焦在 “抽象是否透明” —— 很多人反对的是 “不知道底层逻辑的抽象”。

我们和企业客户聊 Line Graph 时的常见场景：企业内部想搭智能体 / 工作流，一开始大家都自己造轮子（roll their own），但后来发现代码难管理、难协同、难评审—— 这时候就需要标准库 / 框架，提供可组合的底层组件，这正是 Roast 和 Line Graph 的定位。这也是很多人喜欢 Line Graph 的原因。

我还想提一下 Engineer 的 John Welsh 关于 MCP 的演讲 —— 那是个被严重低估的演讲，我当时极力推荐，但没多少人听。如果你听到这里，强烈建议去听 John Welsh 的演讲，真的很棒。

Swyx：

对，特别好！他在演讲里明确提到了 “为什么企业需要 LangGraph 这类框架” —— 他说，2024 年年中 Anthropic 的工具调用能力变好后，大家都开始做集成，结果一片混乱，于是 MCP 就诞生了：为工具访问制定标准协议，让大家都能采用，降低协同成本和认知负荷。这其实就是大型组织需要标准化工具（无论是框架还是协议）的务实原因 —— 不是为了用框架而用框架，而是为了解

决实际协同问题。

Swyx：

同意！而且框架应该是 “入门跳板”，这正是它们的核心价值。好啦，非常感谢你抽出这么多时间分享！最后有没有什么 “硬广” 要打？比如你的项目或课程。

Lance：

当然有！如果听到这里，非常感谢大家的收听。我们有好几门课程：我教过 “环境智能体搭建” 和 “Open Deep Research 搭建”。我其实是受 Carpal 的启发 —— 他很久以前发过一条推文，聊 “搭建入门路径（building on ramps）”：他有个 “micro rap repo”（疑似项目名称），一开始没多少人关注，但他做了个 YouTube 视频当 “入门教程”，之后仓库关注度暴涨。我很喜欢这种 “1+2 组合”：先做一个产品（比如 Open Deep Research），再做一个 “入门教程”，让大家能自己动手搭建。

所以我开了一门 “Open Deep Research 搭建课程”，完全免费，里面有很多笔记本（notebooks），一步步教大家怎么搭建 —— 你能看到这个智能体的性能有多好，我们很快还会发布更好的测试结果。如果你需要 “开源的深度研究智能体”，可以去看看，搭建过程很有趣。我在 “苦涩教训” 的博客里也聊

到了这个智能体的搭建历程，感兴趣的可以去读。

Alessio：

太棒了，Lance，感谢你的参与！

Lance：

谢谢大家，聊得很开心，能来这里特别棒！