页 1：标题页—AI 编程分享：我之 AI 观

页面内容摘要

```
# AI 编程分享
## 我之 AI 观
 快速带过

副标题：从 Copilot 到 Claude Code 的踩坑之旅
时长标注：~2 小时
```

6 问分析

1. 如何讲

- **时长**：~30 秒
- **节奏**：无 v-click，直接展示
- **过渡语**：
  - 开场：「大家好，今天来分享一下我对 AI 编程的一些认知和踩坑经历。」
  - 结束：「标题里的'我之 AI 观'，其实源于我这两年的切身体验——接下来我会先讲一下，为什么要做这个分享。」
- **核心记忆点**：这是一场基于**个人踩坑经历**的分享，不是官方培训

2. 为何要懂

- **痛点**：听众可能以为这是一场"官方推广"或"技术培训"，心态被动
- **价值**：明确这是**同事视角的经验分享**，可以更轻松、互动地参与
- **与听众的关联**：设定预期——这是可以随时打断提问的讨论，不是单向灌输
- **重要程度**： 了解即可（标题页本身不传递核心知识）

3. 演示策略

- **需要演示**：否
- **备注**：标题页无需演示，保持简洁开场

4. 可能问题

同事视角：

问题	准备的回答
这个分享有录屏吗?	有，会后可以分享视频和 slides
多长时间?	预计 2 小时，中间会有休息

老板视角：

问题	准备的回答
为什么是你来分享?	我比较早开始用这些工具，踩了一些坑，想把经验同步给大家

5. 取舍逻辑

没讲的内容	取舍理由
个人背景介绍	同事都认识，不需要自我介绍
分享的完整目录	下一页”今天的旅程”会详细展开

如果被问到怎么答：「具体内容马上会讲到。」

6. 观点/事实区分

内容	类型	来源/依据	不确定性
“从 Copilot 到 Claude Code”	事实	个人使用经历	低
“我之 AI 观”	个人观点	强调是个人视角	低

讲解时注意：开场就明确”个人观点”，降低听众预期，避免被当作权威定论

页 2：为什么做这个分享?

页面内容摘要

```
# 为什么做这个分享?
 演示

内容:
1. 公司要引入 Agentic Coding 了
2. [v-click] 但很多人的认知还停在「把代码贴进 DeepSeek 对话框问一下」
3. [v-click] 比喻：「已经是保时捷车主了，可是科目一的题库还有两千题没刷」

脚注:
- Anthropic "Introduction to agentic coding"
- Stack Overflow 2025 报告解读

演示脚本：用 Claude Code + 巫女子 skill 讲笑话
```

6 问分析

1. 如何讲

- **时长**：~3 分钟（含演示）
- **节奏**：3 步 v-click 渐进揭示
  - Click 1：「公司要引入 Agentic Coding」—抛出背景
  - Click 2：「很多人认知还停在…」—点出痛点
  - Click 3：保时捷比喻—用幽默化解紧张，引发共鸣
- **过渡语**：
  - 开场：「为什么今天要做这个分享呢?」
  - 结束：「所以今天的目标，就是帮大家把这两千题刷一刷——接下来看看我们的旅程。」
- **核心记忆点**：**工具升级了，但认知还没跟上**

2. 为何要懂

- **痛点**：听众可能不知道自己”不知道什么”，对 AI 工具的认知停留在表面
- **价值**：明确分享的价值——**填补认知差距**，不是推销工具
- **与听众的关联**：直接点名”公司要引入”，与每个人的工作相关
- **重要程度**： 必须懂（这是整场分享的 motivation）

3. 演示策略

- **需要演示**：是（页面标注 演示）
- **演示内容**：用 Claude Code 桌面版 + 巫女子 skill 讲笑话
- **演示步骤**：
  1. 打开 VSCode 中的 Claude Code —「让 AI 来讲个笑话」
  2. 输入 prompt —「帮我用巫女子的口吻说一个笑话」
  3. 展示生成过程—「看，AI 不只是回答问题，还能用特定风格输出」
  4. 收尾—「这就是 Agentic Coding 的一个小例子——AI 可以理解上下文、调用技能、产出个性化内容」
- **时长**：~1 分钟
- **风险与备用**：提前测试一遍，确保 skill 正常工作

4. 可能问题

同事视角：

问题	准备的回答
什么是 Agentic Coding?	简单说就是 AI 从”回答问题”升级为”自主执行任务”，后面会详细讲
DeepSeek 和 Claude Code 有什么区别?	DeepSeek 是对话式，Claude Code 是 Agent 式，能直接操作文件、执行命令

老板视角：

问题	准备的回答
为什么是这个时候引入，不是更早或更晚?	更早：工具不够成熟，Context Window 太小，Agent 能力弱；更晚：竞争对手已经在用了，我们会落后
为什么是你来讲?	我从 Copilot 时代就开始用，踩过坑也积累了经验，想把这些同步给大家，帮大家少走弯路
为什么现在是好时候?	工具刚好到了 Agent 阶段（能执行而不只是建议），而且公司决定引入，正好需要扫盲
提升了多少工作效率?	因人而异、因任务而异。我个人体感：重复性任务（写 CRUD、配置文件）可能提升 50–80%；复杂逻辑设计提升有限，更多是”换一种协作方式”。后面踩坑部分会详细讲

5. 取舍逻辑

没讲的内容	取舍理由
Agentic Coding 的详细定义	后面 Part 2 会系统讲解
Stack Overflow 报告的完整数据	脚注提供链接，感兴趣的自己看
各种 AI 工具的对比	开场不适合深入对比，后面会涉及

如果被问到怎么答：「这些后面都会讲到，先留个悬念。」

6. 观点/事实区分

内容	类型	来源/依据	不确定性
公司要引入 Agentic Coding	事实	公司决策	低
很多人认知停在对话框阶段	个人观点	观察同事使用情况	中（可能有人已经很熟练）
保时捷比喻	个人观点	自创比喻	低（就是个比喻）
Stack Overflow 数据	事实	报告原文	低

讲解时注意：「很多人认知停在…」这句话要注意措辞，避免让听众觉得被”教训”

页 3：今天的旅程

页面内容摘要

```
# 今天的旅程
 快速带过 | 双栏布局

左栏：
- Part 2：AI 编程简史与核心概念（~60 min）
- Part 3：踩坑与反思（~40 min）
- 声明：个人视角、目标扫盲、欢迎打断
- [v-click] 预言：短期高估、长期低估

右栏：
- [v-click] 信源等级说明（L1–L4）
-  必读标记说明
- 范围声明：只聊 AI 辅助编程
```

6 问分析

1. 如何讲

- **时长**：~2 分钟
- **节奏**：
  - 直接展示议程：「今天分两大部分…」
  - 念完声明（30 秒）
  - Click 1：揭示”预言”，重点强调 Builder 心态
  - Click 2：快速带过右栏信源等级，点一下范围声明
- **过渡语**：
  - 开场：「先看一下今天的议程」
  - 结束：「好，那我们正式开始——Part 2，AI 编程简史。」
- **核心记忆点**：**短期高估、长期低估**——要用 Builder 心态

2. 为何要懂

- **痛点**：听众不知道分享的结构和预期时长，可能走神
- **价值**：设定预期，让听众知道”我们要去哪里”
- **与听众的关联**：声明”欢迎打断”降低距离感
- **重要程度**：　　了解即可（议程页本身不传递核心知识）

3. 演示策略

- **需要演示**：否
- **备注**：纯信息展示页，无需演示

4. 可能问题

同事视角：

问题	准备的回答
100 分钟会不会太长?	中间会休息，而且内容密度高，走马观花反而浪费大家时间
L1–L4 信源等级是什么?	简单说就是：官方 > 媒体 > 博主 > 社交讨论，让大家知道哪些信息更可靠
能讲讲嵌入式/前端/XX 领域怎么用吗?	我入职时间不长，对各业务线了解有限，今天讲的是通用概念，具体应用需要大家结合自己的场景探索

问题	准备的回答
有没有更新的工具/方案?	AI 领域更新很快，今天聚焦已经验证过的、相对稳定的内容，更前沿的等成熟了再分享

老板视角：

问题	准备的回答
为什么不涉及 AI 绘画?	今天聚焦编程领域，绘画/视频是另一个专题，可以以后再分享

5. 取舍逻辑

没讲的内容	取舍理由
Part 1 去哪了?	Opening 就是 Part 1，但没有单独编号，避免混淆
各工位的具体建议（嵌入式、前端、应用端）	我入职一年多，对公司业务图景所知不完整，只能讲**通用且重要**的知识点
更新的不确定方案（如最新的 AI 工具）	扫盲为主，不确定的内容不适合在这里讲
每页的详细时间分配	过于细节，听众不需要知道

如果被问”这个在我们项目能用吗”:「具体场景需要你们自己探索，今天只讲通用原理，帮大家建立认知框架。」

6. 观点/事实区分

内容	类型	来源/依据	不确定性
时间分配 60+40 min	个人估算	排练经验	中（实际可能有出入）
“短期高估、长期低估”	共识观点	技术采用的普遍规律	低
“今天的 AI 是未来十年最差的 AI”	个人观点	行业趋势判断	中（AI 发展有不确定性）

讲解时注意:“最差的 AI”这个说法要解释清楚——是指能力会持续提升，不是说现在的 AI 很差

页 5：GPT-3 发布（ 重点）

页面内容摘要

```
# 2020.5: GPT-3 发布 — 1750 亿参数，证明「规模」带来质变
 重点

左栏概念卡片：
- Token（词元）：计费单位，100-300 tokens ≈ 一个函数
- Next-Token Prediction：本质是「预测下一个词」→ 会胡说八道
- Context Window（上下文窗口）：GPT-3 仅 2K

右栏 [v-click]：
- 文本补全示例（不是对话）
- GPT-3 的局限： 无状态、无记忆、无网络、无工具、纯文字、2K

脚注：OpenAI 论文
```

6 问分析

1. 如何讲

- **时长**：~4 分钟
- **节奏**：
  - 先讲 Mermaid 图：「GPT-3 是一个文本补全模型——给它输入，它预测下一个词」
  - 逐个解释三个概念卡片（Token、Next-Token、Context Window）
  - Click：展示文本补全示例 + 六个局限
- **过渡语**：
  - 开场：「AI 编程的历史，我们从 2020 年的 GPT-3 讲起。」
  - 结束：「这些局限，后面的技术都是在一个一个解决它们——先来深入理解 Token 和 Context Window。」
- **核心记忆点**：**GPT-3 是”文本补全”，不是”对话”，它有很多局限**

2. 为何要懂

- **痛点**：很多人以为 AI 是”理解”人类语言，其实只是”预测下一个词”
- **价值**：理解 Next-Token Prediction 是理解 AI 幻觉、局限的基础
- **与听众的关联**：知道为什么 AI 会”胡说八道”，就不会盲目信任
- **重要程度**： 必须懂（这是后续所有内容的基础）

3. 演示策略

- **需要演示**：否（页面有示例，不需要现场演示）
- **备用**：如果有人问”能演示一下吗”，可以在 Claude Code 里输入半句话让它补全

4. 可能问题

同事视角：

问题	准备的回答
什么是 Token?	简单理解：1 个英文单词 ≈ 1 token，1 个中文字 ≈ 2 token。API 按 token 计费
为什么说它会”胡说八道”?	因为它只是预测”最可能的下一个词”，不是真的”理解”——后面讲幻觉会详细解释
2K 是什么意思?	2000 个 token，大约 1500 个汉字或 1 页半文档

老板视角：

问题	准备的回答
2020 年的东西为什么要讲?	理解起点才能理解进化，GPT-3 的局限正是后续技术要解决的问题
为什么从 GPT-3 讲起，不从更早的开始?	从 word2vec、BERT 讲起一下午都推进不了进度。GPT-3 是”麻雀虽小五脏俱全”——Token、Context Window、Next-Token Prediction 这些核心概念它都有，是理解后面所有 LLM 玩法的基础

5. 取舍逻辑

没讲的内容	取舍理由
word2vec、BERT 等早期模型	一下午都推进不了进度，GPT-3 已经”五脏俱全”
GPT-1、GPT-2	GPT-3 是里程碑，之前的对理解当前工具帮助有限
训练细节（预训练、微调）	太深入，扫盲不需要

如果被问到怎么答:「GPT-3 虽然是 2020 年的, 但它包含了我们需要理解的所有核心概念——Token、Context Window、Next-Token Prediction。后面的模型都是在这个基础上演进的。」

6. 观点/事实区分

内容	类型	来源/依据	不确定性
1750 亿参数	事实	OpenAI 论文	低
“规模带来质变”	共识观点	行业共识	低
Next-Token Prediction 导致幻觉	事实	LLM 工作原理	低
Token 计费估算	事实	API 定价	低

讲解时注意：无特别争议

页 6：Token 与 Context Window（ 重点）

页面内容摘要

```
# Token 与 Context Window — LLM 的两个核心概念
 重点

左栏 - Token:
- 刻度尺: 200(函数) → 1K(页) → 5K(文件) → 200K(书)
- 中文 ≈ 1.4 tokens/字, 英文 ≈ 1.2 tokens/词
- 价格表: GPT-5.2 $1.75/$14, Claude Opus 4.5 $5/$25, DeepSeek V3.2 $0.28/$0.42

右栏 [v-click] - Context Window:
- GPT-3 的 2048 tokens 可视化
- 对比条: GPT-3 (2K) vs Claude 4 (200K)

[v-click] 两张图片 (详细可视化)

脚注: OpenAI Tokenizer, Jay Alammar 可视化
```

6 问分析

1. 如何讲

- **时长**：~4 分钟
- **节奏**：
  - 先讲左栏 Token：「Token 是计费单位，看这个刻度尺...」
  - 讲价格表：「不同模型价格差异很大，DeepSeek 比 GPT-5.2 便宜 6-33 倍」
  - Click 1：揭示 Context Window，讲 GPT-3 的 2K 限制
  - Click 2-3：展示可视化图片，加深理解
- **过渡语**：
  - 开场：「上一页提到 Token 和 Context Window，这页展开讲一下。」
  - 结束：「从 2K 到 200K，Context Window 增长了 100 倍——这是后面讲 Agent 的基础。接下来看 Copilot 是怎么用这些能力的。」
- **核心记忆点**：**Token 是计费单位，Context Window 决定 AI 能"看"多少**

2. 为何要懂

- **痛点**：不理解 Token 就不知道成本从哪来，不理解 Context Window 就不知道为什么 AI 会"忘事"
- **价值**：理解这两个概念是评估 AI 工具能力和成本的基础
- **与听众的关联**：知道为什么有时候 AI 回答变差了（Context 溢出了）
- **重要程度**： 必须懂

3. 演示策略

- **需要演示**：否（页面已有足够可视化）
- **备用**：如果有人好奇，可以打开 OpenAI Tokenizer 网页演示

4. 可能问题

同事视角：

问题	准备的回答
DeepSeek 这么便宜，为什么不都用它?	便宜但能力有差异，复杂任务 Claude/GPT 更强。不同场景选不同模型

问题	准备的回答
200K 是多少?	约 15 万字，一本中等长度的书
超过 Context Window 会怎样?	早期内容会被”遗忘”或截断，AI 看不到了
AI 公司怎么赚钱? 成本在哪?	成本分两部分：**训练成本**（一次性，如 GPT-4 据说花了上亿美元）和**推理成本**（持续的，每次调用都要算力，就是 Token 费）。目前主流 AI 公司都在亏损阶段

老板视角：

问题	准备的回答
我们用哪个模型？成本多少?	看具体场景。DeepSeek 适合简单任务，复杂任务用 Claude/GPT。成本取决于使用量
我们用 AI 的成本结构是什么?	我们付的是推理成本（按 Token 计费）。训练成本是 OpenAI/Anthropic 承担的

5. 取舍逻辑

没讲的内容	取舍理由
Tokenizer 算法细节（BPE 等）	太深入，知道”大概 1–2 token/字”就够了
各模型详细对比	时间限制，给出代表性的就行
Context Window 扩展技术	后面会讲到（RAG、滑动窗口等）
训练成本 vs 推理成本详细展开	时间限制，口头补充即可

如果被问到怎么答：「Tokenizer 的具体算法可以看脚注链接，今天重点是理解概念。」

6. 观点/事实区分

内容	类型	来源/依据	不确定性
Token 价格	事实	官方定价（可能过时）	中（价格常变动）
1.4 tokens/中文字	事实	Tokenizer 测试	低
Context Window 对比	事实	官方文档	低
AI 公司都在亏损	共识观点	行业报道	中

讲解时注意：价格可能已经变了，强调”这是截止到我准备材料时的价格”

页 7：GitHub Copilot（ 快速带过）

页面内容摘要

```
# 2021.6: GitHub Copilot
GPT-3 微调 → Codex · 159GB 代码 · 5400 万仓库

左栏 [v-click 动画]:
- 空白 → 写注释 → 幽灵代码 → Tab 接受
- 示例: movingAverage 函数

右栏:
- HumanEval: GPT-3 0% → Codex 28.8%
- 改变: 不用离开 IDE, 代码「就地生成」
- 受限: 无状态/记忆/网络、只能补全、无法跨文件
- 数据: +55.8% 效率 | ~40% 漏洞 | ⚖ 版权

脚注: 3 个研究引用
```

6 问分析（精简版）

1. 如何讲

- **时长**：~1.5 分钟
- **节奏**：快速演示 v-click 动画，点一下优缺点，不深入展开
- **过渡语**：
  - 开场：「理解了 Token 和 Context Window，来看第一个生产级 AI 编程工具」
  - 结束：「解决了不离开 IDE 的问题，但仍是无状态——接下来看 ChatGPT」
- **核心记忆点**：**Copilot 是第一个「IDE 内就地生成」的工具，但仍是无状态补全**

2. 为何要懂

- **重要程度**： 懂了更好（后续演进的参照点）
- **价值**：理解”第一代”局限，才知道为什么需要后续演进

3. 演示策略

- **需要演示**：否（页面本身就是动画演示）

4. 可能问题

问题	准备的回答
40% 漏洞是真的吗？	是 2021 年研究，现在有改善但仍需审查
我们能用吗？	企业版 $19/月，注意版权和代码泄露风险

5. 取舍逻辑

快速带过，不深入技术细节。重点是展示”第一代补全工具”的能力边界。

6. 观点/事实区分

页面数据均有研究来源支持，可信度高。注意 55.8% 是特定任务的结果。

页 8：ChatGPT（ 快速带过）

页面内容摘要

```
# 2022.11: ChatGPT
 快速带过
5 天 100 万 · 2 个月 1 亿 · 史上最快

左栏：RLHF
- GPT-3 vs ChatGPT 对比（哥伦布例子）
- 关键转变：预测「下一个词」→ 预测「人类喜欢的回答」
- 1.3B + RLHF > 175B 无 RLHF

右栏：幻觉 Hallucination
- 定义 + 为什么不可避免
- 真实案例：律师提交 6 个不存在判例被罚 $5,000
```

6 问分析

1. 如何讲

- **时长**：~2 分钟
- **节奏**：快速 v-click，重点讲「RLHF 让小模型胜过大模型」和「幻觉不可避免」
- **过渡语**：
  - 开场：「Copilot 只能补全，那能不能对话？2022 年 ChatGPT 来了」
  - 结束：「知道了幻觉的存在，接下来看 GPT-4 带来的能力跃升」
- **核心记忆点**：**RLHF 让小模型胜过大模型，但也让幻觉更「自信」**

2. 为何要懂

- **重要程度**： 懂了更好
- **价值**：理解”为什么 AI 会自信地说错”——律师案例是最好的警示
- **补充说明**：幻觉从 GPT-3 就有（Next-Token Prediction 天然会编），RLHF 让它更明显

3. 演示策略

- **需要演示**：否

4. 可能问题

问题	准备的回答
xxxB 参数是什么意思?	模型的「记忆容量」，175B=1750 亿个可调数字。但不是越大越好——1.3B+RLHF 胜过 175B 无 RLHF，训练方法比堆参数更重要
现在最大模型多少?	2025 年开源旗舰约 1T（万亿），如 Kimi K2、Qwen3-Max。GPT-5/Claude 不公开参数量。且现在用 MoE 架构，参数量不再是唯一指标
幻觉能解决吗?	目前只能抑制无法消除，所以要人工审查 + RAG 等技术减少
1.3B 比 175B 强是真的?	是在”对话质量”维度，人类更喜欢 1.3B+RLHF 的回答
它有记忆吗?	LLM 本身是无状态的。每次请求都把整个对话历史发过去，模型重读所有内容再回复。ChatGPT/Claude 的「记忆」功能是应用层包装（服务器存储 + 注入），不是模型本身的能力

5. 取舍逻辑

快速带过，不深入 RLHF 技术细节。重点是理解「训练方法 > 堆参数」和「幻觉不可避免」两个核心观点。

6. 观点/事实区分

内容	类型	来源
1.3B+RLHF > 175B	事实	OpenAI InstructGPT 论文
幻觉不可避免	事实	OpenAI 官方技术文章
律师案例	事实	公开法庭记录

参考来源

- TechTarget: Best LLMs 2026
- Shakudo: Top 9 LLMs 2025

页 9：GPT-4（ 快速带过）

页面内容摘要

```
# 2023.3: GPT-4 — 能力跃升
 快速带过

左栏：
- 多模态（能看图）
- Function Call（Agent 基石，后面详讲）
- Context 8K → 32K
- 幻觉↓40%、推理↑、20× 价格
- Technical Report 首次不公开参数

右栏：
- 考试成绩对比：律师资格从后 10% → 前 10%
- 不是渐进改进，是「质变」
```

6 问分析

1. 如何讲

- **时长**：~1.5 分钟
- **节奏**：快速过三个新能力，重点看考试成绩对比表
- **过渡语**：
  - 开场：「ChatGPT 让对话成为可能，GPT-4 带来了质变」
  - 结束：「Function Call 是 Agent 的基石，下一页详细讲」
- **核心记忆点**：**从「后 10%」到「前 10%」= 质变，但代价是 20 倍价格 + 不再开源**

2. 为何要懂

- **重要程度**： 懂了更好
- **价值**：理解 GPT-4 是”质变”级别的跃升，为后续 Agent 概念做铺垫

3. 演示策略

- **需要演示**：否

4. 可能问题

同事视角：

问题	准备的回答
20 倍价格值吗?	看任务。简单任务（翻译、格式化）用 3.5/4o-mini 够了；复杂推理（多步逻辑、代码 debug）用 4/4o。现在还有 Claude Sonnet 等性价比选择
多模态能干嘛?	截图 → 代码、UI 稿 → 实现、报错截图 → 诊断。实际工作中，截图比复制粘贴错误信息更快更准
Function Call 是什么?	让模型”说”要调用什么函数、传什么参数，由程序执行后把结果返回。是 Agent 能”动手”的基础，下一页详细讲
Context 32K 够用吗?	当时够看一个中等文件。但现在 Claude 已经 200K，Gemini 有 1M+，32K 已经是”起步价”
幻觉降 40% 是怎么测的?	OpenAI 内部 TruthfulQA 等基准测试。但 40% 是相对降低，不是绝对消除——仍需人工审查

老板视角：

问题	准备的回答
现在价格还是 20 倍吗?	不是。2025 年 GPT-4 级别的模型价格已降到当初的 1/10 以下，且有 Claude Sonnet、Gemini Flash 等更便宜的替代品
“不再 Open”有什么影响?	意味着无法审计模型、无法本地部署、受 OpenAI 政策变化影响。这也是为什么很多企业转向开源模型（Llama、Qwen）或 Claude（有更透明的安全政策）
律师资格前 10% 是真的能当律师了?	只是考试能力，不是执业能力。AI 不能理解案件背景、客户需求、职业伦理。就像学生能考高分不代表能当好律师
能用来替代什么工作?	辅助而非替代。适合：文档起草、代码生成、数据分析初稿。不适合：最终决策、客户沟通、创意方向

5. 取舍逻辑

快速带过技术细节，重点突出「质变」和「代价」。Function Call 留到下一页详讲。

6. 观点/事实区分

内容	类型	来源
考试成绩对比	事实	OpenAI Technical Report
幻觉降低 40%	事实	OpenAI 内部测试
20× 价格	事实	当时 API 定价
“质变”	共识观点	行业普遍认同

页 10：Function Call（ 重点）

页面内容摘要

```
# Function Call
 重点
2023.6 · 从「回答」到「执行」

左栏：5 步流程
1. 开发者定义可用函数
2. 用户提问
3. 【关键】模型输出结构化 JSON — 决定调用什么，但不执行
4. 系统执行函数，返回结果
5. 模型整合结果，生成回复

右栏：Mermaid 时序图 + 核心提示
 没有 Function Call，就没有 Agent
```

6 问分析

1. 如何讲

- **时长**：~4 分钟
- **节奏**：先讲 5 步流程，重点强调第 3 步「决定但不执行」，再看时序图，最后现场演示
- **过渡语**：
  - 开场：「前面都是 LLM 本身的能力——能理解、能回答。但 GPT-4 还带来了另一个关键能力：Function Call。从这里开始，AI 能『做事』了」
  - 结束：「有了 Function Call，AI 从『回答问题』进化到『执行任务』。但如何保证它知道『该查什么』？下一页讲 RAG」
- **核心记忆点**：**模型只『决定』，不『执行』——这是安全边界，也是 Agent 的基础**

2. 为何要懂

- **重要程度**： 必须懂
- **价值**：这是理解 Agent 的关键。不懂 Function Call，后面的 Agent、MCP、Subagent 都会一头雾水
- **痛点**：很多人以为 AI「能做事」是魔法，其实是一套结构化的协议

3. 演示策略

- **需要演示**：是（强烈推荐）
- **演示指令**：「帮我分析一下当前仓库的 markdown 文件有多少文字?」
- **演示工具**：Claude Code
- **演示要点**：
  1. Claude 先调用 Glob 找到所有.md 文件
  2. 再调用 Read 或 Bash(wc) 统计每个文件
  3. 最后整合结果，给出汇总
- **亮点**：观众能清楚看到「决策 → 执行 → 整合」的完整循环
- **时长**：约 30 秒

4. 可能问题

同事视角：

问题	准备的回答
Function Call 和 API 有什么区别?	API 是程序员写代码调用；Function Call 是让 AI 自己决定调用什么 API。你告诉 AI「有这些工具」，它自己选择用哪个
为什么模型不直接执行?	安全考虑。如果模型能直接执行 `rm -rf /`，后果不堪设想。决策和执行分离，程序可以审核和拦截危险操作
这和 Copilot 补全有什么区别?	Copilot 只能预测代码，Function Call 能调用真实 API（查天气、操作文件、发邮件）。一个是「写」，一个是「做」
调用失败怎么办?	返回错误信息给模型，模型可以换个工具或换个参数重试。好的 Agent 会有重试和降级策略
JSON 输出格式是固定的吗?	是的，OpenAI/Anthropic 都定义了标准格式。开发者用 JSON Schema 描述函数签名，模型按格式输出

老板视角：

问题	准备的回答
这能接入我们的系统吗?	可以。只需定义 JSON Schema 描述你的接口，模型就能调用。Claude 和 GPT 都支持，主流框架（LangChain 等）都有封装
安全吗? 会不会做错事?	设计上是安全的——模型只输出「我想调用 X」，真正执行由你的代码控制。可以加权限检查、人工确认等
能做什么具体业务?	查库存、查订单、查设备状态、生成报表、调用内部 API。任何能写成 API 的操作，理论上都能让 AI 调用
和 RPA 有什么区别?	RPA 是「录制回放」，写死流程；Function Call 是「智能决策」，根据需求动态选择工具。更灵活，能处理变化

5. 取舍逻辑

没讲的内容	取舍理由
JSON Schema 具体语法	太技术细节，开发时查文档即可
Streaming 响应处理	实现细节，扫盲不需要
Parallel Function Calling	进阶功能，基础概念先过
各厂商实现差异	OpenAI/Anthropic 大同小异，不必展开

如果被问到怎么答：「技术实现上各家略有差异，但核心思想一样——模型输出 JSON 决策，程序负责执行。具体语法开发时查文档就好。」

6. 观点/事实区分

内容	类型	来源
2023.6 发布	事实	OpenAI 官方文档
「没有 Function Call 就没有 Agent」	共识观点	行业普遍认同
决策与执行分离	事实	API 设计规范
5 步流程描述	事实	OpenAI/Anthropic 文档

演示备忘

正式演示：

```
帮我分析一下当前仓库的 markdown 文件有多少文字？
```

备用演示（如果时间更紧）：

```
这个目录下有多少个 TypeScript 文件？
```

演示时口述：

> 「注意看，我只是问了一个问题，但 Claude 自己决定了要先找文件、再统计、最后汇总。这就是 Function Call——模型决定『调用什么工具』，程序负责『真正执行』。」

页 11：RAG 检索增强生成（ 快速带过）

页面内容摘要

```
# RAG 检索增强生成
 快速带过
2020 · 与其让模型记住所有知识，不如在需要时「去查」

Mermaid 流程图：
- 索引阶段：文档库 → Embedding Model → 向量数据库
- 查询阶段：query → Embedding → 向量搜索 → Top-K 相关上下文 → LLM → answer

解决的问题： 知识截止 减少幻觉
典型应用：Cursor @codebase、NotebookLM
```

6 问分析

1. 如何讲

- **时长**：~1.5 分钟
- **节奏**：快速过流程图，重点讲「解决的问题」和「典型应用」
- **过渡语**：
  - 开场：「Function Call 让 AI 能调用工具，但如何让它知道『该查什么』？RAG 就是答案」
  - 结束：「RAG 解决了知识问题，接下来看如何让 AI『想得更清楚』——思维链」
- **核心记忆点**：**与其让模型记住所有知识，不如在需要时「去查」**

2. 为何要懂

- **重要程度**： 懂了更好
- **价值**：理解 Cursor @codebase 为什么能理解整个项目的原理

3. 演示策略

- **需要演示**：否（流程图已足够清晰）

4. 可能问题

问题	准备的回答
向量数据库是什么？	存储「语义」而不是「关键词」的数据库。「猫」和「喵星人」在向量空间里很近，能被一起检索出来
和传统搜索有什么区别？	传统搜索是关键词匹配；RAG 是语义匹配。搜「如何修复登录问题」能找到标题是「认证失败解决方案」的文档
Cursor @codebase 就是 RAG？	是的。它把你的代码库索引成向量，提问时先检索相关代码，再喂给 LLM
能完全消除幻觉吗？	不能，只能减少。如果检索到的内容本身有错，或者检索不全，仍可能幻觉
为什么 Claude Code 不用 RAG？	因为 Context 够大了（200K+），可以按需读取文件。RAG 是「先索引再检索」，Claude Code 是「理解任务后按需读取」——更灵活、更实时。但超大代码库仍需要 RAG

5. 取舍逻辑

快速带过，不深入 Embedding、分块策略、重排序等技术细节。重点是理解「查了再答」的核心思想。

技术演进对比（如果被问到）：

因素	RAG（2020）	现代 Agent（2025）
Context 大小	2K–8K，必须精选	200K+，能装下大量代码
检索方式	预先索引 → 向量搜索	按需使用工具（Glob/Grep/Read）
实时性	索引可能过时	每次都读最新文件
灵活性	依赖索引质量	根据任务动态决定读什么

6. 观点/事实区分

内容	类型	来源
2020 年提出	事实	Meta AI RAG 论文
解决知识截止和幻觉	共识观点	行业普遍认同
Cursor 使用 RAG	事实	Cursor 官方文档
现代 Agent 更多用按需读取	共识观点	工具实践

页 12：思维链 Chain of Thought

页面内容摘要

```
# 思维链 Chain of Thought
2022 Google · 让模型「一步步想」而不是「直接答」

左栏：
- 示例：Roger 5个球 + 2罐×3个 → 5+(2×3)=11
- 为什么有效：
  - 注意力聚焦：一次只关注问题的一部分
  - 外部工作记忆：中间步骤 = 额外计算资源
  - 可检查纠错：每步都是检查点

右栏：
- AIME 数学竞赛：GPT-4o 12% → o1 83%（精英级）
- DeepSeek R1：1/27 OpenAI 价格，MIT 开源
```

6 问分析

1. 如何讲

- **时长**：~2 分钟
- **节奏**：先过示例，再讲「为什么有效」的三点，最后看 AIME 对比
- **过渡语**：
  - 开场：「RAG 解决了『知道什么』，思维链解决『如何想』」
  - 结束：「有了这些能力——Function Call、RAG、思维链——我们就可以构建 Agent 了」
- **核心记忆点**：**「一步步想」比「直接答」准确得多——o1 从 12% 到 83%**

2. 为何要懂

- **重要程度**：　　懂了更好
- **价值**：理解为什么 o1/R1 这类「推理模型」能解决复杂问题
- **与听众关联**：写 prompt 时加「let's think step by step」就是在用 CoT

3. 演示策略

- **需要演示**：否（页面数据已足够说明效果）

4. 可能问题

问题	准备的回答
CoT 和普通 prompt 有什么区别?	普通 prompt 直接要答案；CoT 要求模型先写推理过程再给答案。相当于让学生「写计算过程」而不只是「写答案」
我怎么用 CoT?	最简单的方法：在 prompt 末尾加「Let's think step by step」。或者给一个带推理过程的示例
o1 和普通 GPT-4 有什么区别?	o1 把 CoT「训进去」了，不需要你提示它就会自动思考。但更贵、更慢
DeepSeek R1 能替代 o1 吗?	数学推理能力相当，但其他方面（代码、创意）可能不如。最大优势是开源 + 便宜
思维链是应用层还是模型层?	两者都有。Prompt 加「step by step」是应用层；o1/R1 把它训进模型是模型层
Thinking 和非 Thinking 是两种模型吗?	看情况。Claude Sonnet + Extended Thinking 是同一模型的不同模式；GPT-4o 和 o1 是独立训练的不同模型。核心区别：Thinking 先推理再答，更准但更慢更贵

5. 取舍逻辑

快速带过技术原理，重点用 AIME 成绩说明效果。不深入 o1 的训练方法。

Thinking vs 非 Thinking 对比（如果被问到）：

类型	例子	特点
同模型不同模式	Claude Sonnet + Extended Thinking	开关切换，开启后先输出思考过程
独立训练模型	GPT-4o vs o1	o1 专为推理训练，架构可能不同
开源推理模型	DeepSeek R1	专门训练，思考过程可见

使用建议：

- 简单任务（翻译、格式化）→ 非 Thinking，省钱省时间
- 复杂任务（多步推理、代码调试）→ Thinking，准确率高

6. 观点/事实区分

内容	类型	来源
2022 年 Google 提出	事实	Jason Wei 论文
o1 AIME 83%	事实	OpenAI 官方
R1 价格 1/27	事实	DeepSeek 定价
~100B 参数才有效	事实	涌现能力研究

页 13：Agent 从「回答」到「执行」（ 重点 演示）

页面内容摘要

```
# Agent：从「回答」到「执行」
 重点 演示

左栏：核心区别
- Traditional LLM：问题 → 回答（单次交互 · 无记忆 · 只生成文字）
- Agent：目标 → 循环执行 → 完成（自主规划 · 调用工具 · 自我纠错）

右栏：Deep Research 示例
- 用户：帮我研究 MCP 协议的安全风险
- Agent 循环：思考 → 搜索 → 阅读 → 分析 → 生成报告
- 本质：Thought → Action → Observation 循环
```

6 问分析

1. 如何讲

- **时长**：~5 分钟
- **节奏**：先对比 LLM vs Agent 核心区别，再逐步展示 Deep Research 示例，最后现场演示
- **过渡语**：
  - 开场：「有了 Function Call、RAG、思维链这些能力，我们终于可以构建 Agent 了」
  - 结束：「刚才是通用 Agent 平台，接下来看专门为程序员设计的 AI 编程工具」
- **核心记忆点**：**LLM 是「问答」，Agent 是「目标驱动的循环执行」——Thought → Action → Observation**

2. 为何要懂

- **重要程度**： 必须懂
- **价值**：这是理解所有 AI 工具的关键。Cursor、Claude Code、GitHub Copilot Workspace 都是 Agent
- **与听众关联**：理解 Agent 才能理解为什么这些工具能「自己干活」

3. 演示策略

- **需要演示**：是（强烈推荐）
- **演示工具**：ChatGPT Deep Research 或 Claude Research
- **演示 Prompt**：

```
请帮我调研 2025 年主流大语言模型的对比分析，重点关注：

1. **国际模型**：
   - GPT-5.1（OpenAI）
   - Claude 4.5 Sonnet / Opus 4.5（Anthropic）
   - Gemini 3 Pro（Google）

2. **国内模型**：
   - DeepSeek V3.2
   - 通义千问 Qwen3 / Qwen2.5-Max
   - Kimi（月之暗面）
   - 文心一言 ERNIE-5.0（百度）
   - 豆包（字节跳动）

对比维度：
- 参数规模与架构（Dense vs MoE）
- 上下文长度
- 多模态能力
- API 定价（每百万 token）
- 开源程度
- 各自擅长的场景

请给出清晰的对比表格，并总结「什么场景用什么模型」的选型建议。
```

- **演示要点**：
  - 让听众看到 Agent 自动搜索、阅读多个网页、整合信息的过程
  - 强调「不是一次搜索，而是多轮调研」
  - 最终输出带引用的结构化报告
- **时长**：启动后让它跑 1–2 分钟展示过程即可

4. 可能问题

问题	准备的回答
Agent 和 ChatGPT 有什么区别?	ChatGPT 是一问一答；Agent 接收目标后自主规划、调用工具、循环执行直到完成。ChatGPT Plus 的 Deep Research 功能就是一个 Agent
Agent 会不会失控?	好问题。现在的 Agent 都有人类确认机制——危险操作要求确认，超过一定步数会暂停。但这确实是 AI 安全研究的重点
这和 RPA 有什么区别?	RPA 是写死的流程脚本；Agent 是理解目标后动态规划。RPA 遇到变化会报错，Agent 可以自己调整策略
Deep Research 要多少钱?	GPT-4 Plus 订阅（$20/月）包含一定额度。Claude Pro 的 Research 功能类似。企业级按 token 计费
我们能自己做 Agent 吗?	能。用 LangChain、Claude Agent SDK 等框架，定义工具和目标，就能构建。但调稳定需要经验

5. 取舍逻辑

没讲的内容	取舍理由
ReAct、AutoGPT 等具体框架	太技术，扫盲不需要
Agent 的 prompt 工程	进阶话题
多 Agent 协作	后面 Subagent 页会讲

6. 观点/事实区分

内容	类型	来源
Thought → Action → Observation 循环	事实	ReAct 论文
Deep Research 2025.2 发布	事实	OpenAI 官方
5–30 分钟自主研究	事实	OpenAI 产品描述

演示备忘

2025 年模型格局速览（演示时可口述）：

维度	代表模型
推理最强	Gemini 3 Pro（1501 Elo，首破 1500）
编程最强	Claude 4.5 Sonnet（30+ 小时自主编程）
性价比之王	DeepSeek V3.2（$0.27/M tokens，MIT 开源）
生态最全	通义千问 Qwen3（119 语言，Apache 2.0）
长文本专精	Kimi（20 万字输入）

演示时口述：

> 「注意看，我只是给了一个目标，Agent 自己决定要搜索什么、阅读哪些网页、如何整合信息。这就是 Agent 和普通 LLM 的区别——不是一问一答，而是自主规划、循环执行。」

参考来源

- Shakudo: Top 9 LLMs 2025
- 2025 LLM Review
- 中国大模型排名
- 国内主流 AI 大模型对比

页 14：AI 编程工具：三代范式（ 演示）

页面内容摘要

```
# AI 编程工具：三代范式
 演示

三代工具对比：
- 第一代 - 补全：Copilot (2021) - Tab 接受，行内预测
- 第二代 - 对话：Cursor (2024) - Chat + Diff，多文件编辑
- 第三代 - 自主：Claude Code (2025) - 终端 Agent，自主执行

对比表：
| 维度 | 补全 | 对话 | 自主 |
|------|------|------|------|
| 交互 | Tab 接受建议 | 审核 Diff 变更 | 确认目标即可 |
| 范围 | 行/函数级补全 | 跨文件重构 | 项目 + 工具链 |
| 角色 | 你写，它补 | 它写，你审 | 你定目标，它执行 |
| 上下文 | 当前文件片段 | 项目级索引 | 200K tokens |

范式演进：人类从「执行者」→「决策者」
三者正在合流：Copilot 加了 Agent，Cursor 加了 Background Agent
```

6 问分析

1. 如何讲

- **时长**：~3 分钟
- **节奏**：快速介绍三代工具，重点看对比表，强调范式演进
- **过渡语**：
  - 开场：「了解了 Agent 概念，现在看程序员最关心的——AI 编程工具怎么演进的」
  - 结束：「三代工具各有定位，现在深入看第一代 Copilot」
- **核心记忆点**：**人类从「执行者」变成「决策者」——你定目标，它执行**

2. 为何要懂

- **重要程度**： 懂了更好
- **价值**：帮听众建立 AI 编程工具的全景图，理解为什么需要不同工具
- **与听众关联**：三代工具他们日常都可能用到

3. 演示策略

- **需要演示**：是（点开官网看官方演示）
- **演示方式**：
  1. **Copilot** → github.com/features/copilot – 官网有补全动画
  2. **Cursor** → cursor.com – 首页有 Chat + Diff 演示视频
  3. **Claude Code** → claude.ai/download – 有终端演示动画
- **时长**：每个 15–20 秒，快速切换展示差异
- **口述**：「注意看交互方式的变化——Tab 接受 → 审核 Diff → 确认目标执行」

4. 可能问题

问题	准备的回答
我该用哪个?	看任务。日常写代码用 Copilot（已集成 IDE）；重构或功能开发用 Cursor；大型任务或脚本自动化用 Claude Code。可以叠加使用
三者正在合流是什么意思?	Copilot 现在有 Copilot Workspace（Agent 模式），Cursor 有 Background Agent。边界在模糊，但核心定位还是不同

问题	准备的回答
200K tokens 是多少代码?	约 15–20 万行代码。一个中型项目的全部源码都能装进去
Claude Code 免费吗?	CLI 本身免费，但调用 Claude API 收费。Pro 订阅 $20/月 有额度，企业按 token 计费

5. 取舍逻辑

快速概览，不深入任何一个工具。后续页面会分别详讲 Copilot、Cursor、Claude Code。

6. 观点/事实区分

内容	类型	来源
三代范式划分	共识观点	行业普遍认同
Copilot 2021、Cursor 2024、Claude Code 2025	事实	官方发布时间
200K tokens 上下文	事实	Anthropic Claude 文档
Cursor 综合领先，Claude Code 自主执行最强	事实	Render 2025 测评

演示备忘

演示顺序:

1. 点开 Copilot 官网 → 展示补全动画 → 「Tab 接受，你写它补」
2. 点开 Cursor 官网 → 展示 Chat + Diff 视频 → 「它写你审」
3. 点开 Claude Code 官网 → 展示终端演示 → 「你定目标，它执行」

口述总结:

> 「三代工具代表三种交互范式。你的角色在变——从执行者变成决策者。但这三者正在融合，Copilot 加了 Agent，Cursor 加了 Background Agent。」

页 15：MCP（Model Context Protocol）（ 重点 演示）

页面内容摘要

```
# MCP 2024.11
 重点  演示

副标题：Model Context Protocol — AI 世界的 USB-C

左栏：实战时序图
- 用户 → Claude → MCP → GitHub 的查询 PR 流程
- 核心价值：AI 不再局限于训练数据，可以获取实时信息并执行实际操作

右栏：
- AI 像新员工：聪明但缺信息和工具 → MCP = 给它接工具箱
- N×M 问题：以前 N 应用 × M 工具 = N×M 适配器 → 现在写一次 Server，所有 Host 都能用
- 三层架构：Host（Claude Desktop、Cursor）/ Client（连接器）/ Server（Playwright、GitHub）
- 三大 Primitives：Tools（模型调用）/ Resources（应用读取）/ Prompts（用户模板）
- 时间线：2024.11 Anthropic → 2025.3 OpenAI → 2025.12 Linux 基金会
- ⚠ 注意：MCP Server 装太多会大量消耗 Context
```

6 问分析

1. 如何讲

- **时长**：~5 分钟
- **节奏**：先用 USB-C 类比建立直觉，再讲 N×M 问题，看时序图理解流程，最后现场演示
- **过渡语**：
  - 开场：「刚才讲了三代 AI 编程工具，现在看一个关键的基础设施——MCP。它让 AI 能够连接各种外部工具和数据源」
  - 结束：「MCP 解决了工具连接问题。但 Agent 运行时间长了，Context 会满。下一页讲两种 Context 管理策略——Subagent 和 Skill」
- **核心记忆点**：**MCP = AI 世界的 USB-C——写一次 Server，所有 Host 都能用**

2. 为何要懂

- **重要程度**： 必须懂
- **价值**：这是 2024–2025 年最重要的 AI 基础设施标准。Anthropic 提出，OpenAI、Google 先后采用，已进入 Linux 基金会
- **与听众关联**：
  - 如果团队要让 AI 连接内部系统（数据库、API、文档），MCP 是标准方式
  - **IoT 场景**：公司的灯控制服务也能做成 MCP Server，让 LLM 直接控制灯——这正是听众团队可以做的事

3. 演示策略

- **需要演示**：是（强烈推荐）
- **演示工具**：Claude Code + Playwright MCP
- **演示指令**：

```
用 Playwright 打开 anthropic.com，点击导航进入 Claude 产品页面，然后截图
```

- **演示要点**：
  1. 观众能看到浏览器自动打开
  2. 页面自动导航到 Claude 产品页
  3. 截图保存作为证明
- **时长**：~30–45 秒
- **演示时口述**：> 「我让 Claude 通过 MCP 控制浏览器——打开 Anthropic 官网，导航到产品页面，然后截图。整个过程 Claude 自己决定怎么操作。这就是 MCP 的价值——统一的协议让 AI 能调用各种工具。」

4. 可能问题

问题	准备的回答
MCP 和 Function Call 什么区别?	Function Call 是「模型调用函数的协议」；MCP 是「让不同 AI 应用共享工具的标准」。MCP 建立在 Function Call 之上，解决的是「写一次，到处用」的问题
我们怎么用 MCP?	两种方式：① 用现成的 Server（GitHub、数据库、Playwright 等）；② 自己写 Server 封装内部系统。Claude Code 和 Cursor 都支持配置 MCP Server
安全吗? MCP Server 能访问什么?	Server 只能做你授权它做的事。启动时需要配置权限。但要注意：不要装来路不明的 Server，它理论上可以访问你授权的所有资源
有哪些现成的 Server?	awesome-mcp-servers 有精选列表。推荐：Context7（文档查询）、Playwright（浏览器自动化）、GitHub（代码仓库）、PostgreSQL（数据库）
为什么说装太多会消耗 Context?	每个 MCP Server 的工具描述会注入到 System Prompt。10 个 Server 各 5 个工具，就是 50 个工具描述，可能占用数千 tokens
Resources 和 Prompts 是什么?	Tools 让模型调用函数；Resources 让应用读取数据（如文件内容）；Prompts 是用户可触发的模板。实践中 Tools 用得最多

5. 取舍逻辑

没讲的内容	取舍理由
MCP SDK 具体语法	开发时查文档即可
JSON-RPC 传输层细节	太底层，扫盲不需要
Resources/Prompts 深入	实践中 Tools 最常用
MCP 安全模型细节	进阶话题

6. 观点/事实区分

内容	类型	来源
2024.11 Anthropic 发布	事实	Anthropic 官方公告
2025.3 OpenAI 采用	事实	OpenAI 官方公告
2025.12 进入 Linux 基金会	事实	Linux Foundation 公告
USB-C 类比	共识观点	社区普遍采用
N×M → N+M 问题简化	共识观点	协议设计文档

演示备忘

正式演示：

```
用 Playwright 打开 anthropic.com，点击导航进入 Claude 产品页面，然后截图
```

备用演示（如果 Playwright 有问题）：

```
用 Playwright 打开 cursor.com 首页，截个图
```

演示时口述：

> 「我让 Claude 通过 MCP 控制浏览器——打开官网，导航到产品页面，然后截图。整个过程 Claude 自己决定怎么操作。这就是 MCP 的价值——统一的协议让 AI 能调用各种工具。」

演示亮点：

- 多步骤操作（不是简单的单次请求）
- 真实浏览器控制（观众能看到页面变化）
- 截图作为证明（可视化结果）

个人案例分享（口述素材）

案例 1：多平台工作整合

> 「我自己的例子——公司用云效做协作开发，我个人写 Home Assistant 贡献代码是在 GitHub。通过 MCP，我很自然地就能让 LLM 帮我看今天有什么任务、今天有什么日程。现在发布版本、打 tag 这些操作，我都让它来帮我做了。」

要点：MCP 让 AI 能同时连接多个平台（云效 + GitHub + 日历），统一在一个对话中完成。

案例 2：IoT 灯控制

> 「再想远一点——我们公司的灯控制服务，完全可以做成一个 MCP Server。然后你就能对 AI 说『把会议室的灯调暗一点』，它就帮你做了。这不是科幻，技术上现在就能实现。」

要点：

- 直接关联听众业务（IoT 团队）
- 展示 MCP 不只是「开发工具」，而是「万物互联的协议」
- 暗示听众团队可以尝试的方向

案例 3：AI 日常管家

> 「其实还能更进一步——我的邮箱服务、Apple 日历也能接进来。问 AI『今天有什么事』，它能同时查邮箱通知、日历日程，综合告诉我。安排会议、回复邮件，都可以让它来做。它就像一个统一入口的日常管家。」

要点：

- 从开发工具延伸到生活助手
- **统一入口**：不用在各个 App 之间切换
- 展示 MCP 的无限可能性

页 16：Subagent vs Skill —Context 管理的两种策略（ 重点 演示）

页面内容摘要

```
# Subagent vs Skill — Context 管理的两种策略
□ 重点□ 演示

左栏：Subagent = 分离出去，独立执行
- 主 Agent 200K context
- Explore / Plan 子进程：独立 context，Haiku 驱动
- 只返回摘要
-□ 优势：不污染主 context · 可并行 10 个 · 失败隔离
- 内置：Explore · Plan · Code Review · Test Runner

右栏：Skill = 注入进来，按需加载
- 同一 Context，渐进式加载
- 1□ 元数据 ~100 tokens
- 2□ 匹配后加载指令 <5K tokens
- 3□ 需要时加载资源/模板
-□ 优势：继承上下文 · 无启动开销 · 可组合
- 内置：/commit · /review-pr · /init
```

6 问分析

1. 如何讲

- **时长**：~4 分钟
- **节奏**：先画对比图，再逐个讲解，最后现场演示两种策略
- **过渡语**：
  - 开场：「MCP 解决了工具连接问题。但 Agent 运行时间长了，Context 会满。怎么办？两种策略」
  - 结束：「这两种策略帮我们管理 Context。但它的生命周期是怎样的？下一页看完整流程」
- **核心记忆点**：**Subagent = 隔离执行（防污染）；Skill = 按需注入（省空间）**

2. 为何要懂

- **重要程度**： 必须懂
- **价值**：理解 Claude Code 的核心架构设计，知道何时用 Subagent、何时用 Skill
- **与听众关联**：日常使用 Claude Code 时，Explore、Plan 都是 Subagent；/commit、git-workflow 是 Skill

3. 演示策略

- **需要演示**：是（强烈推荐，演示两种策略的对比）

演示 1：Subagent（Explore）

```
帮我探索这个项目的架构
```

- 让听众看到 Explore 子进程被触发
- 强调「它有自己的 Context，用完就释放」
- 结果返回时只是摘要，不会污染主 Context

演示 2：Skill（git-workflow）

```
帮我提交代码
```

或直接输入 `/git-workflow`

- 让听众看到指令被注入到当前 Context
- 强调「没有启动新进程，就在主 Context 里执行」
- 它能继承之前的对话上下文
- **时长**：各 30 秒，共 ~1 分钟

- **演示时口述**：> 「刚才 Explore 是 Subagent——分离出去执行，用完释放。现在看 Skill——我说『帮我提交代码』，它会注入 git-workflow 的指令到当前 Context，然后按步骤执行。注意它没有启动新进程，就在主 Context 里完成。」

4. 可能问题

问题	准备的回答
Subagent 和 Skill 怎么选?	看任务规模。探索整个代码库、做 Code Review → Subagent（需要独立空间）；创建 PR、生成 Commit → Skill（步骤明确，按需注入）
Subagent 用的是什么模型?	默认 Haiku（快且便宜），复杂任务可以指定 Sonnet。主 Agent 通常用 Opus/Sonnet
为什么不都用 Subagent?	Subagent 有启动开销，而且失去主 Context 的上下文。简单任务用 Skill 更高效
Skill 会不会把 Context 撑爆?	渐进式加载设计——先加载元数据（~100 tokens），匹配后加载指令（<5K tokens），需要时才加载资源
我能自己写 Skill 吗?	能。Skill 本质是 Markdown 格式的 Prompt + 触发规则。放在.claude/skills/ 目录即可
git-workflow 做了什么?	遵循 conventional commits 规范，自动分析变更、生成 commit message、创建 PR。我自己日常用它来发布版本

5. 取舍逻辑

没讲的内容	取舍理由
Skill 的 YAML 配置语法	开发时查文档
Subagent 的内部通信机制	太底层
多 Subagent 协调	进阶话题

6. 观点/事实区分

内容	类型	来源
Subagent 用 Haiku 驱动	事实	Claude Code 架构
可并行 10 个 Subagent	事实	Claude Code 文档
Skill 渐进式加载 ~100 → <5K tokens	事实	Claude Code 设计
内置 Subagent：Explore/Plan/Code Review	事实	Claude Code 功能

演示备忘

演示顺序：

1. 先演示 Subagent：「帮我探索这个项目的架构」
   - 指出 Explore 进程启动
   - 等待返回摘要结果
2. 再演示 Skill：「帮我提交代码」或 `/git-workflow`
   - 指出指令注入到当前 Context

- 展示它继承了之前的对话

对比口述：

> 「注意两者的区别——Subagent 是『派出去干活，带结果回来』；Skill 是『把专家请进来，在这里干活』。前者隔离但失去上下文，后者共享但占用空间。根据任务选择。」

个人案例：

> 「我自己用 git-workflow 来发布 Home Assistant 的代码。说『帮我提交代码』，它就按 conventional commits 规范生成 commit message，然后创建 PR。整个过程在一个 Context 里完成。」

页 17：Context Window 的生命周期（ 重点）

页面内容摘要

```
# Context Window 的生命周期
 重点

左栏：Context 膨胀动画
- 固定层：System 5K + Tools 3K + MCP 2K
- 动态增长：用户请求 → 读文件 +3K → AI 回复 +2K → 编辑测试 +8K → 安全检查 +25K → 继续迭代 +120K
- 95% 时触发 Auto Compact

右栏：Compact 后效果
- 从 95% → 18%
- 摘要替代详细历史（~8K）
- 可用空间 ~180K

最佳实践：在逻辑断点手动 /compact，而非等待 95%

脚注：
- LLM 是无状态的，每次请求都带上完整 context
- Prompt Caching：厂商缓存重复前缀降成本（Anthropic 10%、OpenAI 50%、Google 25%）
```

6 问分析

1. 如何讲

- **时长**：~3 分钟
- **节奏**：跟着 v-click 动画逐步讲解，让听众「看到」context 膨胀过程
- **过渡语**：
  - 开场：「刚才讲了 Subagent 和 Skill 两种策略。但 Context 终究会满——现在看它的完整生命周期」
  - 结束：「理解了 Context 生命周期，接下来看如何主动管理它——Context Engineering」
- **核心记忆点**：**95% 自动压缩——但在逻辑断点手动 /compact 更好**

2. 为何要懂

- **重要程度**： 必须懂
- **价值**：这是日常使用 Claude Code 最容易遇到的问题——对话久了变慢、回答质量下降
- **与听众关联**：
  - 理解为什么长对话后 AI 表现变差（Context 接近极限）
  - 知道何时该 /compact，而不是被动等待

真实案例（HN 讨论）：

> 有开发者抱怨 LLM 回答质量差。一问才知道——他们在同一个对话里问食谱、聊个人问题、然后写代码。整个 context 都混在一起，AI 当然懵了。

教训：不同话题开不同对话，或者在切换任务时 /compact。这就是为什么理解 context 机制很重要。

3. 演示策略

- **需要演示**：是（简短演示）
- **演示方式**：在 Claude Code 中输入 `/context`，展示当前 context 使用情况
- **演示要点**：
  - 让听众看到实际的 context 使用百分比
  - 展示 System、Tools、Conversation 各占多少
  - 如果当前对话已经有一定长度，效果更好
- **时长**：~15 秒
- **补充口述**：「我一般在完成一个功能后 /compact，相当于『存档』」

4. 可能问题

问题	准备的回答
Compact 会丢失信息吗?	会压缩详细过程，但保留关键决策、代码变更、进行中任务。重要结论不会丢
什么时候该手动 /compact?	在逻辑断点——完成一个功能、解决一个 bug、准备切换任务时。相当于「保存进度」
LLM 无状态是什么意思?	模型没有「记忆」。每次请求都把整个对话历史发过去，模型重新读一遍再回答。这就是为什么 context 会膨胀
Prompt Caching 省多少钱?	各厂商不同。Anthropic 缓存命中只收 10%，OpenAI 50%，Google 25%。长对话越多越省
95% 为什么是临界点?	研究表明 LLM 在接近 context 限制时表现恶化。剩余空间是「工作记忆」，太少了推理质量下降

5. 取舍逻辑

没讲的内容	取舍理由
Compact 算法细节	太底层，用户不需要知道
不同模型的 context 限制对比	变化快，查最新文档
Context 管理的高级技巧	下一页 Context Engineering 会讲

6. 观点/事实区分

内容	类型	来源
95% 时自动 compact	事实	Claude Code 行为
LLM 无状态，每次带完整 context	事实	LLM 架构原理
Prompt Caching 价格（10%/50%/25%）	事实	各厂商官方文档
在逻辑断点手动 compact 更好	最佳实践	Claude Code 使用经验

演示备忘

演示命令:

```
/context
```

演示时口述:

> 「我输入 /context 看看当前的使用情况。你们看——System 占多少、Tools 占多少、对话内容占多少。现在用了 X%，还有很多空间。但如果继续聊下去，就会越来越满。」

口述重点:

> 「每次交互都在增加 context。到 95% 时自动压缩。但我建议不要等到 95%——在完成一个功能后主动 /compact，相当于『存档』。」

个人经验分享:

> 「我自己的习惯：完成一个功能就 /compact 一次。这样既保留了关键信息，又腾出空间给下一个任务。比被动等 95% 更可控。」

页 18：Context Engineering —现代 AI 编程的核心技能（ 重点）

页面内容摘要

```
# Context Engineering — 现代 AI 编程的核心技能
 重点

左栏：比喻引入
- 想象你请了一位行业顾问，TA 能力很强，但只能待一天，而且对你公司一无所知
- 映射：
  - 只能待一天 → Context Window 有限
  - 不知道你公司 → 每次对话都是无状态
  - 你的任务 → 给 TA 最相关的资料
- ⚠ Context Rot：给错资料反而害 TA — 有效 context < 256K
  - 「大多数 Agent 失败是 Context 失败」— Anthropic

右栏：四大解决策略
1.  Write — 给顾问一个笔记本（Scratchpads、长期记忆、Todo 列表）
2.  Select — 只拿最相关的文档（CLAUDE.md、RAG、@codebase）
3.  Compress — 100 页压成 3 页摘要（Auto-compact、/compact）
4.  Isolate — 让 TA 带助手分头调研（Subagent、并行执行）

脚注：
- Anthropic "Effective context engineering" — 「找到最小的高信号 Token 集合」
- Karpathy on X — 「Context Engineering 是填充 context window 的精妙艺术与科学」
- ESR "How To Ask Questions The Smart Way" — 前 AI 时代的经典：向社区提问要给足 context
```

6 问分析

1. 如何讲

- **时长**：~4 分钟
- **节奏**：先用「行业顾问」比喻建立共鸣 → 逐步展开四大策略 → 强调「不是塞越多越好」
- **过渡语**：
  - 开场：「理解了 Context 生命周期，现在看如何主动管理它——这叫 Context Engineering」
  - 结束：「这四大策略是理论框架。下一页看具体怎么写 Prompt」
- **核心记忆点**：**Context Engineering = 给 AI 塞对的东西，而不是塞越多越好**

2. 为何要懂

- **重要程度**： 必须懂
- **价值**：这是 2025 年 AI 编程最热门的概念之一。Karpathy 和 Anthropic 都在强调
- **与听众关联**：
  - 理解为什么「说清楚需求」比「多说几遍」更有效
  - 四大策略是日常使用 Claude Code 的指导框架
  - **IoT 场景**：如果要让 AI 控制设备，Select 策略决定给它哪些设备状态、哪些操作权限

3. 演示策略

- **需要演示**：否（概念框架页）
- **替代策略**：口述时回顾之前的演示
  - 「刚才演示的 Explore、/compact，就是 Isolate 和 Compress 策略」
  - 「后面 CLAUDE.md 那页会展示 Select 策略」

4. 可能问题

问题	准备的回答
Context Rot 是什么?	Context 太多反而降低质量。研究表明有效 context < 256K，塞太多让 AI 迷失方向
「顾问」比喻太抽象?	想想你自己——如果有人给你一大堆不相关的文档，你也会懵。AI 也一样

问题	准备的回答
四个策略有优先级吗?	Select（选对的）最重要。然后是 Write（让它记录）和 Compress（压缩）。Isolate 是进阶技巧
CLAUDE.md 是什么?	项目级别的配置文件，告诉 AI「这个项目用什么技术栈、有什么约定」。后面那页会讲
为什么说「不是塞越多越好」?	Anthropic 的研究：context 越多，准确率反而可能下降。要找「最小的高信号 Token 集合」

5. 取舍逻辑

没讲的内容	取舍理由
每个策略的详细实现	下一页「实操」会讲具体例子
RAG 的技术细节	前面 RAG 那页已讲过原理
Subagent 详细机制	前面那页已讲过

6. 观点/事实区分

内容	类型	来源
Context Engineering 是核心技能	共识观点	Anthropic、Karpathy 等多方强调
有效 context < 256K	事实	Anthropic 工程博客
「大多数 Agent 失败是 Context 失败」	事实	Anthropic 官方声明
四大策略框架	整理归纳	根据最佳实践整理

演示备忘

本页不需要单独演示，但口述时串联之前的演示：

> 「刚才演示的 Explore 是 Isolate 策略——分离出去执行。/compact 是 Compress 策略——压缩历史。后面 CLAUDE.md 会展示 Select 策略——告诉 AI 项目的关键信息。」

口述重点：

> 「记住这个比喻——AI 就像一个能力很强但对你公司一无所知的顾问。你的任务不是给它所有文档，而是给它最相关的资料。这就是 Context Engineering 的核心。」

Karpathy 金句（可选引用）：

> 「Context Engineering 是填充 context window 的精妙艺术与科学」

ESR「提问的智慧」类比（可选引用）：

> 「前 AI 时代有一个经典——ESR 的『How To Ask Questions The Smart Way』。那时候在社区提问如果没给足 context，会被 AT 回这条置顶链接羞耻一下。向 LLM 协作其实同理——没给足精准的 context，就是在犯一样的问题。」

Meta 用法（可选延伸）：

> 「不过这个年代有个好处——你可以让 LLM 按照『提问的智慧』里的原则，帮你检查自己的提问是否及格、是否需要补充更多信息。用 AI 来帮你更好地和 AI 协作。」

页 19：Context Engineering 实操—具体胜过模糊（ 重点）

页面内容摘要

```
# Context Engineering 实操 — 具体胜过模糊
 重点

左栏： 模糊 Prompt
-  帮我写一个时钟
-  好的，用 React...
-  不要 React，用原生 JS
-  好的，加模拟表盘...
-  不要模拟的，要数字时钟
-  3 轮对话，还没开始写代码...

右栏： 具体 Prompt
- 用原生 HTML/CSS/JS 写数字时钟
  · 24小时制，每秒更新
  · 深色背景 #1a1a2e，白色等宽字体
  · 单个 index.html，CSS/JS 内联
-  write_file → index.html
-  1 轮对话，直接完成

底部提示：
-  不确定要写什么？→ 问 AI：「先别回答，为了更高质量的答案，你还需要什么信息？」
- 核心原则：Context 是有限资源 · 工具集要精简 · 一次说清楚

脚注：Anthropic "Effective Context Engineering" — 找到最小的高信号 Token 集合
```

6 问分析

1. 如何讲

- **时长**：~3 分钟
- **节奏**：先让听众看左边的「反面教材」→ 再展示右边的正确做法 → 强调「一次说清楚」
- **过渡语**：
  - 开场：「刚才讲了 Context Engineering 的理论框架。现在看一个具体例子——写时钟」
  - 结束：「这就是 Select 策略的实践——给 AI 最精准的信息。下一页看另一个 Select 策略——AGENTS.md」
- **核心记忆点**：**3 轮对话 vs 1 轮完成——差别就在于「一次说清楚」**

2. 为何要懂

- **重要程度**： 必须懂
- **价值**：这是日常使用 AI 最常见的低效模式——模糊提问导致来回修正
- **与听众关联**：
  - 每个人都有过「说了好几遍 AI 还是不懂」的经历
  - 这个例子让听众立刻知道「具体」应该具体到什么程度
  - **黄金问句**：「先别回答，你还需要什么信息?」——可以现场就用

3. 演示策略

- **需要演示**：可选（如果时间充裕）
- **演示方式**：现场演示「具体 Prompt」的例子
- **演示指令**：

```
用原生 HTML/CSS/JS 写数字时钟
· 24小时制，每秒更新
· 深色背景 #1a1a2e，白色等宽字体
· 单个 index.html，CSS/JS 内联
```

- **演示要点**：
  - 让听众看到「一次提问，直接出结果」
  - 强调不需要来回修正
- **时长**：~30 秒
- **备注**：如果跳过演示，口述时强调对比效果即可

4. 可能问题

问题	准备的回答
具体到什么程度才够?	问自己：「如果我把这个需求给一个新人，他能不能不问问题就开始做?」如果不能，说明还不够具体
每次都要写这么详细吗?	看任务复杂度。简单任务可以模糊一点；复杂任务越具体越好。宁可一开始多写几行，也比来回 3 轮省时间
不知道要什么怎么办?	用黄金问句：「先别回答，为了更高质量的答案，你还需要什么信息?」让 AI 帮你想需要什么
这和传统需求文档有什么区别?	类似，但更口语化。关键是「约束条件」——技术栈、样式、文件结构。传统需求文档往往漏了这些

5. 取舍逻辑

没讲的内容	取舍理由
更多 Prompt 示例	一个例子足以说明原则
Prompt Engineering 技巧大全	进阶话题，查文档
多轮对话的场景	本页强调「一次说清楚」，多轮场景另讲

6. 观点/事实区分

内容	类型	来源
「找到最小的高信号 Token 集合」	事实	Anthropic 官方博客
具体 > 模糊	共识观点	普遍最佳实践
黄金问句有效	最佳实践	社区经验

演示备忘

演示指令（可选）：

```
用原生 HTML/CSS/JS 写数字时钟
· 24小时制，每秒更新
· 深色背景 #1a1a2e，白色等宽字体
· 单个 index.html，CSS/JS 内联
```

口述重点：

> 「左边是我们常见的模式——说一句、改一句，来回 3 轮还没开始写代码。右边一次说清楚——技术栈、样式、文件结构，直接完成。差别就在于『一次说清楚』。」

黄金问句强调：

> 「如果你不知道要说什么，有一个万能问句：『先别回答，为了更高质量的答案，你还需要什么信息?』让 AI 帮你想需要什么。我自己经常用这招。」

页 20：AGENTS.md：项目记忆（ 重点）

页面内容摘要

```
# AGENTS.md：项目记忆
 重点

左栏：有 vs 没有的对比
- 没有：每次都要解释项目是什么、用什么语言、怎么跑测试 → 重新解释
- 有：直接开始，风格一致

WHAT/WHY/HOW 框架：
- WHAT：项目是什么、技术栈、目录结构
- WHY：项目目标、各模块作用
- HOW：开发流程、测试、提交规范
- 指令容量有限：LLM ~150 条 / Claude Code 自带 ~50 条

右栏：示例（60 行以内）
- Commands: npm run build/test
- Code Style: ES modules, 2-space indentation
- Workflow: Typecheck after changes

关键原则：
- 别当 Linter — 交给 ESLint/Ruff
- 手动编写 — 别用 /init

本质：System Prompt 的一部分 — 每次对话自动注入，是 Context Engineering 的落地

脚注：
- HumanLayer "Writing a Good CLAUDE.md" — WHAT/WHY/HOW 框架
- Anthropic "Claude Code Best Practices" — 官方推荐
```

6 问分析

1. 如何讲

- **时长**：~3 分钟
- **节奏**：先对比「有 vs 没有」的差异 → 介绍 WHAT/WHY/HOW 框架 → 展示示例 → 强调「手动编写」
- **过渡语**：
  - 开场：「刚才讲了具体 Prompt 的重要性。但每次都写那么详细？有没有『一劳永逸』的方法？有——AGENTS.md」
  - 结束：「AGENTS.md 是项目级别的 Context。下一页看更高级的工作流——OpenSpec」
- **核心记忆点**：**AGENTS.md = 项目记忆，每次对话自动注入**

2. 为何要懂

- **重要程度**： 必须懂
- **价值**：这是 Claude Code 最重要的配置文件，决定了 AI 是否「懂」你的项目
- **与听众关联**：
  - 不用每次都解释「这是 Python 项目、用 pytest 测试、遵循 Conventional Commits」
  - 60 行以内就能显著提升 AI 的表现
  - **IoT 场景**：可以写「这是 Home Assistant 集成项目，用 Python，测试命令是 pytest」

3. 演示策略

- **需要演示**：可选（展示自己的 CLAUDE.md）
- **演示方式**：打开当前项目的 CLAUDE.md，展示结构
- **演示要点**：
  - 让听众看到真实的 CLAUDE.md 长什么样
  - 强调「60 行以内」的简洁性
- **时长**：~20 秒
- **备注**：如果跳过，口述时强调「我自己的项目都有这个文件」

4. 可能问题

问题	准备的回答
AGENTS.md 和 CLAUDE.md 有什么区别?	一样的东西。CLAUDE.md 是 Claude Code 专用；AGENTS.md 是通用名称，其他工具也能用
为什么不用 /init 自动生成?	自动生成的往往太泛，没有你项目的特色。手动写才能精准描述你的约定
指令太多会怎样?	研究表明 LLM 可靠遵循 ~150 条指令。Claude Code 自带 ~50 条，你还有 ~100 条的空间。超过了会被忽略
放在哪里?	项目根目录。Claude Code 启动时自动读取
多个项目怎么办?	每个项目各自一个 CLAUDE.md。Claude Code 只读当前目录的

5. 取舍逻辑

没讲的内容	取舍理由
.claude/settings.json	进阶配置，查文档
多层级 CLAUDE.md	进阶话题
与.cursorrules 的对比	离题

6. 观点/事实区分

内容	类型	来源
LLM 可靠遵循 ~150 条指令	事实	HumanLayer 博客引用的研究
WHAT/WHY/HOW 框架	最佳实践	HumanLayer 博客
建议 60 行以内	最佳实践	Anthropic 官方 + HumanLayer
别当 Linter	最佳实践	社区共识

演示备忘

演示方式（可选）：

```
cat CLAUDE.md
```

口述重点：

> 「AGENTS.md 就是项目记忆——技术栈、测试命令、提交规范。Claude Code 每次启动都会读取。写一次，后面的对话都自动继承。」

强调手动编写：

> 「有人会问能不能自动生成。我的建议是手动写。自动生成的太泛，没有你项目的特色。60 行以内，写一次，受益终身。」

页 21：从提案到 Skill：OpenSpec 工作流（ 重点 演示）

页面内容摘要

```
# 从提案到 Skill：OpenSpec 工作流
□ 重点□ 演示

Spec-Driven Development — 意图先行，代码随后

流程图（四阶段）：
1.□ Proposal — proposal.md (Why & What), design.md (技术决策), tasks.md (实施清单)
   触发：/openspec:proposal
2.□ Review & Apply — refine specs & tasks, feedback loop, 直到达成一致
   触发：/openspec:apply
3.□ Archive — 合并到 specs/, 移入 archive/, 成为「正式文档」
   触发：/openspec:archive
4.□ Skill 升华 — 提炼通用模式，模型自动调用，跨项目复用
   结构：SKILL.md + 脚本

三列说明：
- 为什么用 Spec-Driven？意图先行、可审计、可迭代、减少幻觉
- 目录结构：openspec/ → changes/, specs/, archive/
- 何时提炼成 Skill？信号：同样的 prompt 打了 3+ 次

脚注：
- Thoughtworks "Spec-Driven Development" — 2025 年 AI 辅助工程新实践
- Anthropic "Agent Skills" — 重复 prompt 就该提炼成 Skill
- OpenSpec — Spec-Driven Development 工作流工具
```

6 问分析

1. 如何讲

- **时长**：~4 分钟
- **节奏**：跟着 v-click 动画逐步展开四个阶段 → 停在「Skill 升华」强调「自动调用」
- **过渡语**：
  - 开场：「刚才讲了 AGENTS.md——那是项目记忆。现在看一个更系统的工作流——OpenSpec。它把『意图』变成可追溯的规范」
  - 结束：「OpenSpec 是 Part 1 的收尾——从工具介绍、能力边界、到工作流。Part 2 我们看知识路线图」
- **核心记忆点**：**Skill = SOP——把踩过的坑固化成规范，让 LLM 每次都能正确执行**

2. 为何要懂

- **重要程度**： 必须懂
- **价值**：这是把「用 AI 编程」从随意对话升级为工程实践的关键方法论
- **与听众关联**：
  - 项目越大越需要可审计的决策记录
  - 「打了 3+ 次同样的 prompt」——每个人都有这个经历
  - **IoT 场景**：设备固件升级流程、测试规范，都可以用 Spec-Driven 管理

观点逆转——SOP 在 Vibe Engineering 时代的必然性：

顶级软件工程实践中的各种 SOP：「100% 测试覆盖率」「语义化类型名称」「代码风格统一」「MAX Linter」「静态类型检查」「PRD/设计文档/TDD」「持续集成/部署」。

> 以前总感觉这些对于小团队的需求有「大炮打小蚊子」的嫌疑。前司 BOSS 说这些不过是「自欺欺人的减慢速度的玩意」。但在**给 LLM 擦了一年的屁股后**，这个观点逆转了。

以前人很难遵守 SOP 的原因：

- Deadline 一紧，代码风格就先放一放
- Review 一忙，测试覆盖就睁一只眼闭一只眼
- 这个项目的实现方案很可能下个月就会弃用了，用半个月来探索「如何正确地搭建项目」不是浪费时间吗?
- 更别提各种 CICD 的 WorkFlow 校验和严格 TDD、PRD 了
- MAX Linter 更是让人痛苦得没脾气

但现在再不定好这些 SOP，你可能会得到：

- 每一轮提问都是全新的代码风格
- 充满 debug 遗留下来的 log 语句
- 每一轮都要不断强调的设计思路
- 一不小心写出来的 shit 被无限放大
- 实际上不能 work 的代码
- 以及完全没有必要的冗余流程

核心转变：

- 以前小团队靠「默契」「脑子里的规矩」就够了，但 LLM 不吃这套
- 如果你不把规范写下来，你就要无限重复
- SOP 变得必要——不是为了「流程正规化」，而是为了让 LLM 每次都能正确地工作
- 也为了在代码量暴增时减轻 review 负担（与 HA 的超繁琐 PR 流程达成和解）

Skill 本质上就是 SOP——把踩过的坑固化成规范，让下次不用再踩。只不过以前 SOP 是给人看的，现在是给 LLM 执行的。

Simon Willison 在 Vibe Engineering 中提到：

> 「**顶级工程实践在 LLM 时代会获得更大的回报** (LLMs actively reward existing top tier software engineering practices)」

Meta 演示价值：这份演讲本身就是用 OpenSpec 管理的——可以现场展示 `openspec/` 目录结构，说明「我说的我自己在用」

3. 演示策略

- **需要演示**：是（强烈推荐 Meta 演示）
- **演示方式**：展示当前项目的 `openspec/` 目录
- **演示指令**：

```
ls -la openspec/
ls -la openspec/changes/
```

- **演示要点**：
  - 展示 `changes/`、`specs/`、`archive/` 三个目录
  - 指出「这份演讲的 speaker notes 就是在 `changes/review-slide-delivery-notes/` 里管理的」
  - 强调「我说的我自己在用」的可信度
- **时长**：~30 秒
- **备用**：如果跳过演示，口述「这份演讲本身就是用 OpenSpec 管理的」

4. 可能问题

问题	准备的回答
这和 Git 分支有什么区别?	Git 管代码，OpenSpec 管意图。Proposal 记录的是「为什么要改」「要达成什么效果」，比 commit message 更完整
为什么要这么复杂?	小项目可以简化。但当项目变大、团队变多，可追溯的决策记录就变得很有价值。而且 LLM 时代这些规范反而更必要
Skill 和 AGENTS.md 有什么区别?	AGENTS.md 是静态配置——项目的约定。Skill 是可执行的——检测到特定任务会自动触发
Skill 怎么触发的?	模型根据任务描述匹配 Skill 的触发条件。比如你说「帮我提交」，它会匹配 git-workflow skill
我可以直接用 OpenSpec 吗?	可以。它是开源的 CLI 工具。但核心理念——Spec-Driven——即使不用工具也能实践

问题	准备的回答
以前觉得 SOP 太重怎么办?	观点要逆转。以前靠默契就够，现在 LLM 不吃这套。不写下来就要无限重复。顶级工程实践在 LLM 时代回报更大

5. 取舍逻辑

没讲的内容	取舍理由
proposal.md 的详细格式	进阶话题，看文档
Skill 的编写语法	进阶话题，看文档
与其他工作流工具对比	离题
Archive 的具体命令	次要细节

6. 观点/事实区分

内容	类型	来源
Spec-Driven Development 是 2025 新实践	事实	Thoughtworks 博客
「同样 prompt 打 3+ 次就提炼成 Skill」	最佳实践	Anthropic 官方建议
OpenSpec 四阶段流程	工具设计	OpenSpec 项目
Skill 自动匹配触发	事实	Claude Code 行为
「顶级工程实践在 LLM 时代回报更大」	观点	Simon Willison (Vibe Engineering)
Skill = SOP	洞察	个人总结

演示备忘

正式演示:

```
ls -la openspec/
ls -la openspec/changes/
```

演示时口述:

> 「这份演讲本身就是用 OpenSpec 管理的。你们看——`changes/` 目录下有正在进行的变更，`specs/` 是正式规范，`archive/` 是已完成的变更。每个变更都有 proposal.md 记录为什么要改、tasks.md 记录要做什么。」

Meta 亮点:

> 「我讲 OpenSpec，用的就是 OpenSpec。这份 speaker notes 的分析，现在就在 `changes/review-slide-delivery-notes/` 里。『说到做到』——这是最好的说服力。」

Skill = SOP 洞察（核心观点）:

> 「Skill 是什么？本质上就是传统的 SOP——把踩过的坑固化成规范。只不过以前 SOP 是给人看的，现在是给 LLM 执行的。而且 LLM 比人更老实——你写什么它就照做，不会『差不多得了』。」

观点逆转（打破旧见）:

> 「以前小团队总觉得各种规范流程太重——『大炮打蚊子』。但给 LLM 擦了一年屁股后，我发现这个观点要逆转了。LLM 不吃『默契』那一套。你不写下来，就要无限重复。正如 Simon Willison 说的：『顶级工程实践在 LLM 时代会获得更大的回报』。」

Skill 升华强调：

> 「最后一步是 Skill 升华。当你发现同样的 prompt 打了三次以上，就该提炼成 Skill。比如我的 `git-workflow` Skill——现在每次提交代码，模型自动按 conventional commits 格式来，不用我每次提醒。」

页 22：Part 2 知识路线图（ 快速带过）

页面内容摘要

```
# Part 2 知识路线图
 快速带过

时间轴：2020 (GPT-3) → 2022 (ChatGPT) → 2024 (Cursor) → 2025 (Agent)

四列内容：
1.  理解 AI — Token、Context Window、无状态、⚠ 幻觉
2. ⚡ AI 能力 — RLHF、多模态、Function Call、RAG、推理模型
3.  工具演进 — Copilot → ChatGPT → Cursor → Claude Code
4.  Agent 生态 — Agent、MCP、Subagent、Skill、Context Eng.

底部总结：
- 方法论演进：Prompt Eng. → Context Engineering
- 核心约束：Context 有限 · 边际递减 · Context Rot
- 一句话总结：LLM = 超强预测器 · Agent = LLM + 工具 + 循环

脚注：roadmap.sh "AI Agents" — AI Agent 学习路径参考
```

6 问分析

1. 如何讲

- **时长**：~1.5 分钟（快速带过）
- **节奏**：跟着 v-click 动画快速扫过四列 → 停在底部「方法论演进」强调 Context Engineering
- **过渡语**：
  - 开场：「这是 Part 2 的总结。把刚才讲的概念串起来看」
  - 结束：「这是知识框架。Part 3 我们看实战中的踩坑与反思」
- **核心记忆点**：**Prompt Engineering → Context Engineering；LLM = 超强预测器，Agent = LLM + 工具 + 循环**

2. 为何要懂

- **重要程度**： 快速带过（总结页）
- **价值**：帮助听众建立全局视野，把零散概念组织成体系
- **与听众关联**：
  - 如果前面有些概念没跟上，这里可以快速补课
  - 提供「查漏补缺」的机会
  - 可以作为后续自学的路线图

3. 演示策略

- **需要演示**：否（纯总结页）
- **替代策略**：快速扫过四列，不逐条解释
  - 「这四列是刚才讲过的内容——理解 AI、AI 能力、工具演进、Agent 生态」
  - 「重点看底部——方法论从 Prompt Engineering 演进到 Context Engineering」

4. 可能问题

问题	准备的回答
这些概念太多记不住怎么办?	不用全记住。关键是知道有这些概念，需要时能查。推荐 roadmap.sh 的 AI Agent 路线图
Context Rot 是什么?	Context 太多反而降低质量。研究表明有效 context < 256K，塞太多让 AI 迷失方向

问题	准备的回答
推理模型没讲?	时间有限跳过了。简单说就是「先思考再回答」——o1、R1 这类模型会花更多时间推理

5. 取舍逻辑

没讲的内容	取舍理由
每个概念的详细解释	前面已讲过，这里只是回顾
推理模型详细机制	时间有限，进阶话题
RLHF 详细过程	进阶话题，查资料

6. 观点/事实区分

内容	类型	来源
时间线 (2020→2022→2024→2025)	事实	各产品发布时间
Prompt Eng. → Context Eng. 演进	共识观点	Anthropic、Karpathy 等
LLM = 超强预测器	共识观点	技术原理
Agent = LLM + 工具 + 循环	共识观点	Agent 架构定义

演示备忘

口述重点（快速带过）：

> 「这是 Part 2 的知识地图。四列——理解 AI、AI 能力、工具演进、Agent 生态。刚才都讲过了。」

> 「重点看底部——方法论从 Prompt Engineering 演进到 Context Engineering。这是核心转变。」

> 「一句话总结：LLM 是超强预测器，Agent 是 LLM 加上工具和循环。记住这个就够了。」

如果有人问推理模型：

> 「时间有限跳过了。简单说就是 o1、R1 这类『先思考再回答』的模型。感兴趣可以查 roadmap.sh。」