???

??????????

??????????

1.1 ?????? vs ??? vs LLM

生物脑 /人脑的核心组成要素: 逻辑能力、长期记忆、短期记忆 。

- 计算能力（ **ALU**）：计算机中是 ALU， LLM 对应为 LLM 本体（语言 ALU）。
- 寄存器 ：计算机中是寄存器， LLM 对应为 KV cache。
- 指令：计算机中是指令集， LLM 对应为模型权重。
- 内存（短期记忆） ：计算机中是 RAM， LLM 缺失（需补充）。
- 硬盘（长期记忆） ：计算机中是磁盘， LLM 缺失（需补充）。

几种记忆 /状态存储形态的对比:

- **KV cache**: 不断增长的、被动态调度执行（通过新 prompt）的计算器。
- **SSM / RNN** 状态
  ：固定大小、不断被更新的状态存储器。适合批量处理一段文本时只关心区域内语义、不需要全局语义的场景；也适合动态生成句子的 **meaning tree**，处理可被层级抽象、组合的表示意义。
- 外存：长期记忆的存储器，可被检索和更新。

1.2 ??"??"?"??"?????

1. 意识主要存在于短期记忆中 ，长期记忆也需要保持自洽性。
2. 短期临时记忆维护了当前意识 ，信息容量相对固定，不会太大。其本质是:
   - 语义压缩 + 上下文存储，上下文处理是一个 "信息压缩 "过程。
   - 记忆过程 = 不断把未知输入换成大脑内部已有知识的表达，持续匹配知识、确保自洽，
     不断根据输入修正大脑（上下文）当前的主观感觉、想法、语境 。
3. 外界输入的目的 ：让大脑内部根据已有的知识点组织出一个意识 /
   想法。对文本的理解和记录，本质上与 LLM 一致 ——文本是串流的，准确的编码过程与 LLM 一致。
4. 语言是大脑的工具，不是智能本身
   ：大脑不断重新组织语言，由中心意识判断是否符合中心思想再决定输出。
   最核心的是人类的意识（即临时的状态记忆） 。
   - 人类处理语言的过程 = 通过语言表达的信息不断修正短期记忆中的观点、感觉、意识。
   - 说话是这个过程的逆过程。
5. 自我 提供了人性本能的最高层级抽象。
6. **LLM** 只是个强大的计算器 ——利用这个计算器 + 记忆，就能实现强大的智能。

1.3 ???????

- 人脑的推理模型: https://agix.host/books/013d4/page/75fc0
- 基于 Transformer 的带额外记忆单元的技术 /模型:
  - Meta 的 **Free Transformer**（自由 Transformer ）：增加一部分算力和参数维护额外状态。
- 与普林斯顿的 **CoALA** 有相似之处，但本架构更强调基础设计抽象: 自洽 、 **LLM** 即计算器 。参考: https://zhuanlan.zhihu.com/p/1909206010096259088

?????????

2.1 ??????????????????

- 人设：长期稳定、缓慢改变的预设特征。
- 当前注意力表达 ：根据输入输出不断改变，容量有限。
- 固有特征状态 ：固有状态、情绪。
- 高度自洽 。

2.2 ????????

- 强大的分级信息检索 。
- 区分临时记忆与长期记忆。
- 自动遗忘、热度排序。

> 旁注：自动遗忘与热度排序作为未来演进方向列出。当前 *WeaveAgent* 工程设计未采纳这两项 ——短时记忆随 *Frame* 生命期自然消亡，长时记忆以关键词树组织、不使用热度概念（详见 `Agent/WeaveAgentDesign.md` *§13*）。

2.3 ???????????

- 自我、人性化 ：维护基本需求、价值观、好奇心、模仿学习。
- 总结抽象 、自动学习和迭代各种技能。
- 自洽。
- 调用和整理记忆 。

2.4 ??????????????????

- [illegible] [illegible]
- [illegible]
- [illegible]
- [illegible]
- [illegible] [illegible]
- [illegible] [illegible]
  - [illegible]
  - [illegible]
  - [illegible]
    [illegible]
- [illegible] "[illegible] "[illegible]

2.5 ???????????????

- [illegible]
- [illegible]
- [illegible]
- [illegible] [illegible]

?????????

1. [illegible]
   - [illegible] prompt[illegible]
   - [illegible]
2. [illegible]
   - [illegible]
   - [illegible]
   - [illegible]
   - [illegible] + LLM [illegible] + [illegible]
3. [illegible]
   - [illegible] + [illegible]
4. [illegible] LLM [illegible]
   - Frame / [illegible]
   - [illegible] "[illegible] xxx [illegible] xxx[illegible] xxx [illegible] xxx"[illegible]
   - [illegible]
5. [illegible]
   - weavemind[illegible]
   - [illegible]
   - [illegible]
6. [illegible] /[illegible] /[illegible]
   1. [illegible]
   2. [illegible]

3. 自洽的灵魂要求
4. 多层，灵活，健壮，自洽的记忆系统

?????????

4.1 ???????

- 人格与个性　　：价值观、灵魂。
- 核心动机　　：
  - 行动力　　：总是为了解决任务，想着下一步干嘛，　　"完成任务　　"是重要目标。
  - 自洽：保持信息合理性。
  - 人性、自我　　：维持人性。
  - 基本价值观、好奇心、模仿学习、　　总结抽象　　。
- 行为逻辑描述　　：
  - 方法学、技能、计划树、　　todolist。
  - 任务调度的方法学描述　　。
  - 理性的行为逻辑模式。
  - 感性的行为逻辑模式。

4.2 ??????????????

- 作用：意识节点，提供当前的背景信息，是　　"意识　"的存储实体。
- 标准操作接口　　：
  - 临时记忆的加载与遗忘。
  - 临时记忆的读取与分析。
  - 寻找/创建/修改/删除。
  - 通过　Frame 的包装接口访问。

4.3 ??????????????????

- 标准操作接口　　：
  - 寻找/创建/修改/删除。
  - 记忆之间有复杂的索引。
  - 基于 关键词　　的信息分类存储。
  - 当前项目相关知识的多层级动态组织　　。
  - 外挂知识操作接口（支持大量永久记忆）。

> 接口形态说明：长时记忆是结构化的知识图（关键词树　　+ *Info* 节点　+ 多对多关联），其物理接口以　`Search / CreateKeyword / CreateInfo / LinkInfo /`

> `MergeKeywords` 等专用操作呈现，与短时记忆的通用 *CRUD* 基类不共享继承关系。两者概念上都是 "寻找/创建/修改/删除" ，实现上是两套接口。详见 `Agent/WeaveAgentDesign.md` §8。

- 存储内容 ：
  - 知识、信息。
  - 各种 skills / tools 的使用方法与方法学:
    - 使用文本描述。
    - 编程执行任务。
    - GSD 方法学的自动设计与自动优化。
  - 外部工具、接口的使用方法（笔记本、计算器等）。

4.4 ?? / Frame ??????????????

> 注: Frame 是最通用的抽象 —— 表达大脑的一个思考任务、一个上下文片段、一个工作片段、一个 handle 、一个阶段性思考过程。

4.4.1 ??/Frame ????

- 任务的 ID、任务的存储空间。
- 任务的类型、性质: 探索型 / 固定成熟任务 / 思考研究。
- 任务过程的描述。
- 任务创建的 原因 。
- 预期达到的结果、 目标 。
- 依赖的任务、前置任务、 背景 。
- 下一步的计划、规划。

4.4.2 ??/Frame ???

- 实际 结果： 失败 / 成功。
- 执行的所有 过程 。
- 执行的历史信息和当前状态。
- 思维逻辑的记录。
- 执行结果的总结、学习、记录。

4.4.3 ???????????

- 通过分层的树状任务组织架构实现: 细节分层隔离 + 全局信息全览 。
- LLM 输出格式化的任务图（任务描述、功能、任务之间的关系），本地自动执行。
- SubFrame List nested to tree。
- 计划树 = 本地数据结构，用本地代码执行:

- 记录所有计划与执行情况。
- 每个节点是一个 skill，有 prompt、输入、输出、执行脚本。
- LLM 执行节点任务，有明确的输入、输出、调用要求。
- 根节点: "你是个有用的助手，等待接收具体的任务。"
- 节点执行失败可触发重新设计节点的拆分、提示词，成功后更新节点（提升效率、节省 token）。
- 技能库保存所有可用 skill，可输出 prompt 描述当前技能库情况。
- 生成/更新规划时，可基于整棵树生成整体状态快照。

4.4.4 ????

- 创建的原因、逻辑关系、需要达到的效果都要明确。
- 所有的行动都要有合理的理由、逻辑链条、结果。

4.4.5 ??????LLM ? Frame ??????

这是"复杂化的 TodoWrite"——LLM 在操作这棵树，可扩展节点、放大分支、合并节点、查看节点详细信息。

文本 + 逻辑决策 **= prompt** 生成。

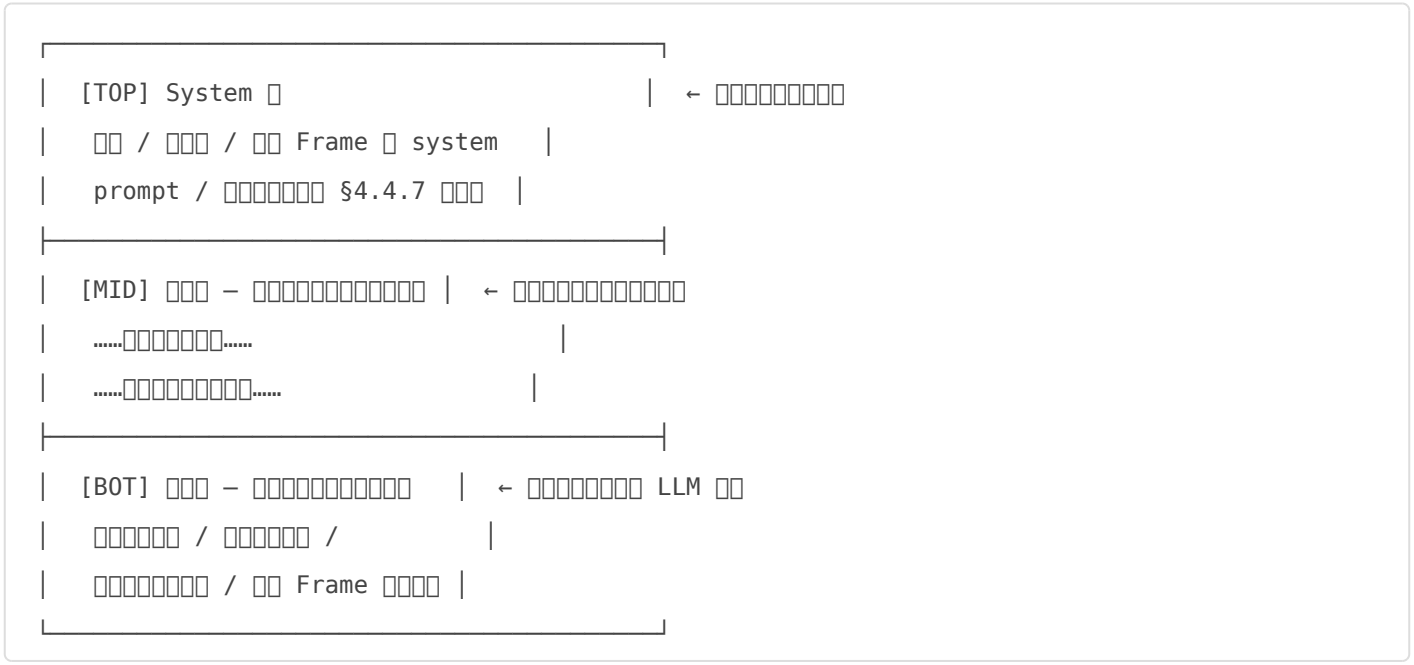
- 扩展：这个问题可以分为以下几个步骤。
- 总结：总结下 xxx 模块在 xxx 方面的设计规则。
- 尝试：我们先尝试以下一个方案。
- 回退：当前方向是错的，回退到前面的第 5 步。
- 收集：当前问题需要先获取 xxx 的信息。
- 抽象：对当前节点任务及子任务进行精炼，按模板生成 skill。
- 笔记：记录临时信息到笔记，并返回笔记段的名称和索引。
- 学习：把指定的信息或笔记写入永久记忆。
- 搜索记忆 / 记录临时记忆 / 规划下一步行动。
- 判断问题需要采用的方法。

4.4.6 Frame ???

- 子 Frame 的完成通知（经事件总线，非直接方法回调——子更新自身状态后由调度器通过 `SUBFRAME_DONE` 事件唤醒父）。
- 父 Frame 的输入激励。
- 检查当前 Frame 的状态、历史、完整思维逻辑。
- Frame 与 Session 的接口交互:
  - task 的格式化返回: 失败 / 成功 + 结果。
  - task 的创建: 收集信息、分析当前问题。
- 每个 Frame 对应一个独立的上下文。 **session** 上下文 与 **Frame** 状态 分离。

4.4.7 Frame ?????

- 工具（文件操作、记忆读写、子 Frame 管理等）统一注册在 Agent 全局的 `ToolRegistry`，Frame 不持有本地工具实例。
- 工具以 分层路径 组织（如 `memory/*`、`frame/*`、`file/*`、`skill/*`），支持简短的多层索引。



- 每个 Frame 只持有全局 Registry 的引用; `Frame.Next()` 在组装 prompt 时按当前状态 / 动作集按路径前缀拉取可见子集注入 System 段，达成 "分层加载 "的效果。

4.4.8 ???????????"?????????"?

> 设计原则: 让上下文压缩变成一次安全的 "中段截断 ", 而不是一次复杂的摘要工程。

本架构对每一次送给 LLM 的上下文采用 严格三段式布局 :

```
┌──────────────────────────────────────────┐
│ [TOP] System 段                          │ ← 长度可控、永远有用
│  人格 / 价值观 / 当前 Frame 的 system    │
│  prompt / 可见工具集（按 §4.4.7 注入）   │
├──────────────────────────────────────────┤
│ [MID] 历史段 — 思维过程、工具调用与返回  │ ← 长度可增长、可被安全裁剪
│  ……过去的思考轨迹……                     │
│  ……过去的工具调用结果……                 │
├──────────────────────────────────────────┤
│ [BOT] 有效段 — 当前推进所需的最少信息    │ ← 长度已知、远小于 LLM 上限
│  最近一段对话 / 最新工具返回 /           │
│  当前要回答的问题 / 当前 Frame 状态摘要  │
└──────────────────────────────────────────┘
```

核心不变量 :

> **`TOP + BOT`** 在任意时刻都是一个可独立执行的最小完整上下文。

- `TOP` 长度由人格 + system prompt + 可见工具集决定, 编排器静态可知 。
- `BOT` 长度由 Frame 类型固定上限（如最近 N 轮对话 + Frame 状态摘要）, 编排器静态可知 。
- `len(TOP) + len(BOT) << len(LLM_max_context)` ——预留出来的 "中段预算 "全部留给 `MID`。

压缩 = 直接删除 **MID** 中段 :

当 `len(TOP) + len(MID) + len(BOT) > 阈值`（例如 LLM 窗口的 80%）时:

1. 不做摘要、不做 **LLM** 调用、不做语义压缩 。
2. 直接从 `MID` 段从中向两端截断 （保留 MID 头若干 token 与 MID 尾若干 token , 删除中间最久远的过程性记录）。
3. 反复迭代直到总长度回到阈值以下。

为什么 "删中段 "不破坏信息完整性 :

关键信息类别	物理存放位置	受截断影响
人格 / 价值观 / 任务目标	`TOP`（ System 段）	□ □□□
当前可见工具与动作集	`TOP`（按 §4.4.7 注入）	□ □□□
当前要做的事 / 最新工具返回	`BOT`（有效段）	□ □□□
Frame 的结论、状态、思维链路总结	**Frame** 状态字段 （ §4.4.2，本地存储）	□ □□□□□ prompt 里）
过去的工具调用过程性记录	`MID`	□ □□□ ——但结论已沉淀到 **Frame** 状态 /笔记 /记忆，过程证据可丢

也就是说: `MID` 段是 "过程性证据 "，不是 "结论性事实 "。结论性事实由 §4.4.5 的 "笔记 / 学习 / 抽象 " 动作主动落入:

- **Frame** 状态字段 （短时，随 Frame 生命期）
- 临时记忆 （短时，随 Session 生命期）
- 长时记忆 / 关键词树 （永久）

`MID` 一旦被丢弃， agent 仍能从这三处 "主存 "中找回任何关键事实 ——**LLM** 上下文只是高速缓存，不是事实的唯一存储位置 。

与 **Anthropic Compaction** 的对比 :

维度	**Anthropic Claude Code**	本架构
触发	接近窗口上限	接近预设阈值（如 80%）
方式	LLM 摘要历史， 保留架构决策与未解 **bug**	直接删除中段 ，无 LLM 调用
成本	每次压缩需要一次 LLM call	零 LLM 调用， O(1) 截断
不变量保障	依赖摘要质量	依赖 Frame 状态 /记忆作为外部主存
失败模式	摘要丢关键信息	`MID` 段过大时 `BOT` 仍完整可用

工程上的额外约束 :

1. `BOT` 段长度必须严格上限 ：例如最近对话 ≤ 8K token，Frame 状态摘要 ≤ 2K token ，超出就主动滚出到 `MID` 或落盘。
2. 关键状态变更必须先写 **Frame** 状态再进入 `MID`：保证一旦 `MID` 被裁剪，状态没丢。
3. 截断只发生在 **Frame** 调度边界 ：不在工具执行中途截，避免破坏正在进行的 tool_call/tool_result 配对。
4. 截断后无需通知 **LLM**：模型从 `TOP + BOT` 即可继续推进；如果某次推理需要历史细节，应通过显式工具调用从记忆中取回，而不是依赖 `MID` 仍在。

> 设计哲学：把上下文压缩从一个"语义难题"降级为一个"长度算术问题"。代价是要求架构其他部分（Frame 状态、记忆系统）承担"事实主存"的责任——这与本架构 §4.5.2 "总结/抽象/记忆"和 §4.3 长时记忆的设计天然对齐。

4.5 ?? / ????????

创建当前状态的下一步行动，是整个系统的主循环。

4.5.1 ??????"????"???

- 有固定的子任务，比如"维持自我"。
- 维护人性、人格、自我的行动，维护基本价值观、好奇心。

4.5.2 ????????????

每个任务独立循环执行，只关心自己的上下文。

- 下一步行动的决策：
  - 主动维护自洽：通过任务上下文、子任务总结等进行自洽判断。
  - 所有行动都要有合理的理由、逻辑链条、结果：
    - 先仔细了解项目，清楚任务背景与逻辑关系后再行动。
    - **Frame** 的思维逻辑链条就是方法学的原料，可总结成方法学，按逻辑关系、分层、树状存储。例如：
      - "先尝试看看能不能快速解决问题"
      - "看起来比较复杂，那我来找找有什么现成的方法学、经验，或者做一个规划"
  - 自动总结/抽象/思考当前状态，规划下一步行动：
    - 根据当前任务需求和状态分析，提出下一步计划。
    - 方法学的选取、加载、知识获取：
      - 寻找／创建／修改／删除方法学。
      - 方法学作为所有工作经验的存储。
      - 特殊领域的 task 必须有方法学才能工作。
      - 方法学一定是包含多个步骤的、复杂、不确定流程的工作步骤。
      - 方法学也可以只是一句话，指导具体的工作方法。
- 总结/抽象/记忆：
  - 主动提取和组织框架性的知识，存储到基础知识框架。
  - 自动进化、自动学习：对任务的基础知识保持持续记忆。

4.5.3 ??????

- 创建、修改、查询、删除子任务。

4.5.4 ????

本能的任务/功能不断产生记忆，存入临时记忆，作为下一次行动的依据——临时记忆 ↔ 本能目标 形成循环。

??????

5.1 ?????????

- 临时记忆模块 ：寻找／创建／修改／删除；通过 Frame 包装接口调用。
- 永久记忆模块 ：
  - 寻找／创建／修改／删除。
  - 基于关键词的分类存储。
  - 当前项目相关知识的多层级动态组织。
- 公共存储区 ：当前基础信息，比如各个任务的状态、本能状态、精神状态。
- 分布式隔离 ：每个任务管理自己的存储，每个模块能一定程度独立工作。

5.2 Frame ????? §4.4?

- 表达大脑思考的一个任务、一个上下文片段、一个工作片段、一个 handle、一个阶段性思考过程。
- 提供下列能力:
  - SubFrame List nested to tree。
  - 自洽要求。
  - 规划下一步行动（方法判断、搜索记忆、记录临时记忆）。
  - Tool 管理（分层、分支组织）。
  - 抽象 Agent 的所有行为（历史、状态、思维逻辑、结果总结）。
  - Frame 接口（见 §4.4.6）。
  - Session 与 Frame 状态分离。

5.3 Prompt ??

- 基础 **prompt** 的维度 ：
  - 人格、人性、自我。
  - 自洽。
  - 行动规划（基于当前任务需求与状态，提出下一步计划）。
  - 学习、总结、抽象。
  - 记忆（什么东西放进记忆里）：
    - 方法学相关。
    - 技能（如 git 使用）。
    - 当前项目知识检索：文件结构、程序设计架构。
- **Prompt** 的生成策略 ：
  - 不同任务类型使用不同 system prompt，避免不必要的提示。
  - 不同任务类型对应不同处理方式与方法学。
  - **Prompt** 的多层级动态组织 。
- **Frame** 分类 对应不同 prompt 模板。

- **LLM** 输入输出的抽象 ：格式化。

5.4 ??? LLM ???????

1. 记忆与存储 ：知识、历史、状态的记录与检索。
2. **Frame / Action** 管理：任务规划工具。
3. 逻辑思维管理 ：思考历史与逻辑关系的记录与检索。

????????????

6.1 LLM ??????????????????

因为 LLM 注意力长度有限，实现高深度、大范围自洽的途径：

- 外挂知识 ：大量可高效检索的信息。
- 独立的推理过程 ：实现高效的动态决策。
- "意识 "存储：背景、当前状态信息、思维链路 ——需要高效表示、表达逻辑链路。

6.2 ???????????

- 能理解非常大的软件工程，理解大量文件的工程设计思想。
- 复杂问题的开发、分析：解决空间较大的问题。
  - 例： CIM 算力芯片在 AI 算法上的应用。

6.3 ????????

- 通过本地可理解的思维步骤动态组织上下文和 **prompt**， **LLM** 不需要每次都看到全局 。
- 思维树 ：表示思维的逻辑图， LLM 通过定义好的动作集合操作这棵树。
- 上下文调度管理器 ：
  - 多个上下文同时协作：
    - 短上下文，分步回答。
    - 并行多个上下文，支持嵌套。
    - sub-agent 执行特定子任务。
  - 多 prefill 少 decode（直接回答问题）。
  - 本地小模型？（待探索）
- 是否需要本地逻辑推理？
  - LLM 输出作为逻辑推理的结果，本地直接处理，不一定走 tools/skills 调用。
  - 固定功能：记忆、数据库。

6.4 ??????????????????

- 创造和更新 skills。
- 制造和整理 tools、脚本。
- 学习的方法学 ：
  - 软件工程开发流程：产品、规划、验收、测试、发布 ……
  - 复杂任务可经过非常多次尝试后解决，并永久学习。

6.5 ??

（贯穿全文，是核心准则。）

6.6 ???

- 利用 "自我 "这一抽象，提供终极的目标指引。

????-Harness12????

??????????????× Harness 12 ?? ????

> 分析对象：`Agent/design/一种理想的智能体编排架构.md` 对照标准：`Agent/design/Agent-Harness解剖.md` 中归纳的 12 个生产级 Harness 组件 目的：把 "理想架构 "的设计意图，逐项映射到 Anthropic/OpenAI/LangChain 共识的工程组件上，找出 已覆盖 、等价但术语不同 ，缺失 三类位置。

???????

#	**Harness** 组件	覆盖度	对应文档位置（章节）	一句话判断
1	Orchestration Loop	□ ◫	§4.5 本能 /调度器、§4.5.4 循环闭环	"本能 "= dumb loop，主循环显式建模
2	Tools	□ ◫	§4.4.7 Frame 可见的工具、 §5.4	全局 `ToolRegistry` + 分层路径，比 PDF 更严格
3	Memory	□ ◫ + 增强	§2.2，§4.2，§4.3，§5.1	短 / 长时分离，关键词树组织，比 Claude Code 更结构化
4	Context Management	□ ◫ + 增强	§3.4，§4.4，§4.4.7，**§4.4.8**，§6.1，§6.3	三段式布局 + "删中段 "压缩，把语义难题降为长度算术问题
5	Prompt Construction	□ ◫	§5.3，§3.1，§4.5.1	强调多层级动态组织 + 任务类型差异化模板
6	Output Parsing	⚠ 部分	§4.4.3，§5.3	偏向 "LLM 输出格式化任务图本地执行 "，缺乏 native tool calling 的明确选型

#	**Harness** 组件	覆盖度	对应文档位置（章节）	一句话判断
7	State Management	□ ◫	§4.4.2、§4.4.6、§5.1 公共存储区	Frame 状态 + 事件总线（`SUBFRAME_DONE`），未提 git checkpoint
8	Error Handling	⚠ 部分	§4.4.3、§4.4.5（回退动作）	有"节点失败重设计"+"回退"，但缺四类错误的分级处理
9	Guardrails & Safety	⚠ 维度不同	§4.1、§6.6、§3.1（人格）	用"人格 + 自洽 + 价值观"做软约束；缺工程级 tripwire / 权限闸
10	Verification Loops	⚠ 部分	§4.4.4、§4.5.2（主动维护自洽）	"自洽判断"本质是 LLM-as-judge；缺 rules-based / visual 校验
11	Subagent Orchestration	□ ◫	§4.4.3、§4.4.6、§6.3 sub-agent	SubFrame nested-to-tree；缺 Fork/Worktree 的物理隔离讨论
12	Long-Horizon Execution	□ ◫ + 增强	§4.5.4、§6.4 自动更新与学习	"本能 + 永久记忆 + 学习闭环"= 比 Ralph Loop 更高阶的连续性

结论速览：12 项中 **7** 项完整覆盖、**3** 项部分覆盖、**2** 项采取了不同维度的解决思路（Output Parsing、Guardrails & Safety）。

????????

---

1. Orchestration Loop??????— ? ??

Harness 标准：实现 TAO（Thought-Action-Observation）循环 ——assemble prompt → call LLM → parse → execute tools → loop。Anthropic 提倡 "dumb loop"，把所有智能留给模型。

理想架构对应：

- **§4.5** 本能 / 调度器（主循环）：**"创建当前状态的下一步行动，是整个系统的主循环"**——直接对应 dumb loop。
- **§4.5.4** 循环闭环：**"本能任务不断产生记忆，存入临时记忆，作为下一次行动的依据**——临时记忆 ↔ 本能目标 形成循环"。

- **§2.4** 本能： "先思考，再行动 / 自动触发记忆检索 / 自动反思总结 "——把 TAO 三段内化为本能行为。

□□ □□ 完整，且做了 抽象升级 ——把循环本身命名为 "本能 "，赋予人格化语义。比 Anthropic 的 dumb loop 更厚一些 （包含主动反思 + 自动记忆检索），与 "harness 应越薄越好 "的原则有轻微张力。

---

2. Tools????— ? ??

Harness 标准：工具以 Schema（name/description/parameters）注入； Tool 层负责注册、校验、入参提取、沙箱执行、结果格式化；典型如 Claude Code 的 6 大类工具。

理想架构对应 ：

- **§4.4.7 Frame** 可见的工具 ：
  - 全局 `ToolRegistry`， Frame 不持有本地实例
  - 分层路径组织： `memory/*`、`frame/*`、`file/*`、`skill/*`
  - `Frame.Next()` 在装配 prompt 时按当前状态 /动作集 拉取可见子集注入 **System** 段
- **§5.4** 提供给 **LLM** 的基础设施 ：记忆与存储、 Frame/Action 管理、逻辑思维管理三类。

□□ □□ 完整， 比 **PDF** 更严格 ——

- "全局 Registry + 分层路径 + 按状态拉取可见子集 " 直接命中 **Harness** 决策表的第 **6** 项（ **Tool Scoping**）： Vercel 删 80% 工具反而更好的经验，这里通过架构层面就避免了 " 工具一次性塞满上下文 "。
- 与 Claude Code 的 "渐进披露 / Skills"机制等价。

---

3. Memory????— ? ?? + ??

Harness 标准：短期（ session 内对话） + 长期（跨 session 持久化）； Anthropic 三层结构：轻量索引（~150 字符常驻） → 主题文件 → 原始 transcript（搜索访问）； 关键原则： **agent** 把自己的记忆当 **hint**，行动前必须验证真实状态 。

理想架构对应 ：

- **§2.2** 记忆： 明确分 临时记忆 与 长期记忆 ；强调 "强大的分级信息检索 "。
- **§4.2** 临时记忆 ：意识节点，提供当前背景；通过 Frame 包装接口访问。
- **§4.3** 长时记忆 ：基于 关键词 分类存储； 多层级动态组织 ；外挂知识接口。
- **§5.1** 存储与记忆基础设施 ：分布式隔离 ——每个任务管理自己的存储。

□□ □□ 完整，且 结构化程度高于 **Claude Code**：

- Claude Code 用 `CLAUDE.md` + `MEMORY.md` 平铺文件
- 理想架构用 关键词树 **+ Info** 节点 **+** 多对多关联 （见 §4.3 旁注与 `WeaveAgentDesign.md` §8）

- 这与 MindStore 现有实现（ Qdrant + 关系边）天然对齐

潜在差距 ： Anthropic 的 "agent 把记忆当 hint，行动前验证真实状态 "原则，文档没有显式表达；建议在 §4.3 加上一条 "读出的长时记忆需经过当前 Frame 的自洽校验 "。

---

4. Context Management???????— ? ?? + ??

Harness 标准： 应对 Context Rot；四种生产策略： Compaction / Observation Masking / Just-in-time Retrieval / Sub-agent Delegation。Anthropic 目标： "找到最小高信号 token 集合 "。

理想架构对应 ：

Harness 策略	理想架构对应
Compaction	**§4.4.8** 三段式布局 + "删中段 "压缩（无需 LLM 调用， O(1) 截断）
Observation Masking	§4.4.7 工具按状态拉取子集（隐式屏蔽）
Just-in-time Retrieval	§4.4.7 分层加载 + §6.3 "本地可理解的思维步骤动态组织上下文 "
Sub-agent Delegation	§4.4.3 SubFrame nested-to-tree + §6.3 sub-agent
额外： Frame 隔离	§4.4.6 "每个 Frame 对应一个独立的上下文， session 上下文与 Frame 状态分离 "

□□ □□ 完整覆盖， 且在 **Compaction** 策略上有原创性增强 ——

- §3.4 "降低对 LLM 上下文的需求 "是这一组件的 显式设计目标
- §6.1 "LLM 上下文有限，难以实现大范围高深度自洽 " = Context Rot 的中文表述
- "Frame 隔离 + 分层加载 "是文档独有的 架构级解法 ，对应 PDF 决策表第 3 项的 "sub-agent delegation + structured note-taking"组合
- **§4.4.8** 三段式布局 是本架构对 Anthropic Compaction 的降维替代方案 ：
  - 核心不变量 ： `TOP（System 段）+ BOT（有效段）`
    在任意时刻是可独立执行的最小完整上下文，长度静态可知且 `len(TOP)+len(BOT) << len(LLM_max_context)`
  - 压缩 = 直接删除 **MID** 中段： 不调 LLM、不做摘要、 O(1) 截断
  - 不破坏完整性的根据 ： MID 段本质是 "过程性证据 "，结论性事实由 §4.4.5 的 "笔记 /学习 /抽象 "动作主动落入 Frame 状态 / 临时记忆 / 长时记忆 ——**LLM**
    上下文是高速缓存，不是事实主存

维度	**Anthropic Claude Code**	本架构 **§4.4.8**
触发	接近窗口上限	接近预设阈值（如 80%）
方式	LLM 摘要历史	直接删 MID 中段
成本	每次压缩 1 次 LLM call	0 次 LLM call
不变量保障	依赖摘要质量	依赖 Frame 状态 /记忆作主存

维度	Anthropic Claude Code	本架构 §4.4.8
失败模式	摘要丢关键信息	MID 过大时 BOT 仍完整

评价：把上下文压缩从一个 "语义难题 "降级为 "长度算术问题 "，代价是要求 Frame 状态系统与记忆系统承担 " 事实主存 "职责 ——这与本架构 §4.5.2 "总结 /抽象 /记忆 "和 §4.3 长时记忆的设计天然对齐，并不增加额外负担。 原 **P2** 级 **"Compaction** 触发条件缺失 "已闭环。

---

5. Prompt Construction?Prompt ???— ? ??

Harness 标准：层级化拼装 ——System Prompt → Tool Definitions → Memory Files → Conversation History → Current User Message； OpenAI Codex 用严格优先级栈。

理想架构对应 ：

- **§5.3 Prompt** 工程：
  - 基础 prompt 维度: 人格、自洽、行动规划、学习总结、记忆
  - **Prompt** 的多层级动态组织
  - 不同任务类型用不同 system prompt
  - **Frame** 分类 对应不同 prompt 模板
- **§3.1** 自我意识 ： "精心设计的 prompt"——把人格从大模型内部移出来到编排器
- **§4.5.1** 根任务 ：维持 "自我意识 "的入口 ——对应 system prompt 最高层

□□ □□ 完整，且 多了 "人格层 "维度 ——把人格、价值观显式作为 prompt 的最高层结构，Anthropic/OpenAI 通常只暗示在 system prompt 里。

---

6. Output Parsing??????— ?? ??

Harness 标准：现代系统依赖 原生 **tool calling**——模型返回结构化 `tool_calls` 对象，无需解析自由文本； OpenAI/LangChain 用 Pydantic 做 schema 约束。

理想架构对应 ：

- **§4.4.3 SubFrame nested to tree**： "LLM 输出 格式化 的任务图（任务描述、功能、任务之间的关系），本地自动执行 "——是格式化输出，但偏向 自定义 **DSL**。
- **§5.3 LLM** 输入输出的抽象 ： "格式化 "———笔带过，没具体方案。

评估： ⚠ 概念到位，工程选型缺失 ——

- 文档承认 "格式化输出 "是必要的，但 没有明确选型 ：用原生 tool_calls？ JSON Schema？自定义 DSL？
- "LLM 输出格式化的任务图 "听起来像自定义结构化输出，会 损失原生 **tool calling** 的训练增强收益

建议：对齐 Harness 共识 ——Frame 的动作集（ §4.4.5 扩展 /总结 /尝试 /回退 /收集 /抽象 /笔记 /学习）应该实现为原生 **tool** 列表，而不是文本指令。这样能直接受益于模型厂商对 tool calling 能力的训练优化。

---

7. State Management?????— ? ??

Harness 标准： LangGraph 用 typed dict 流过图节点 + 超步 checkpoint; OpenAI 四种互斥策略; Claude Code 用 git commit 当 checkpoint + progress 文件当结构化草稿本。

理想架构对应 ：

- **§4.4.2 Frame** 的状态 ：实际结果（失败 /成功）、执行的所有过程、历史信息和当前状态、思维逻辑的记录 、执行结果的总结。
- **§4.4.6 Frame** 接口：子 Frame 完成通知 经事件总线 （ `SUBFRAME_DONE` ） ——这是显式的状态变更协议。
- **§5.1** 公共存储区 ：当前基础信息 ——各任务的状态、本能状态、精神状态。

□□ □□ 完整 ——

- Frame 的 7 个状态字段（结果 /过程 /历史 /思维 /总结） 比 **LangGraph** 的 **typed dict** 更结构化
- 事件总线驱动的状态推进，避开了直接方法回调，有更好的解耦

潜在差距 ：缺 **checkpoint /** 时间旅行调试 ——LangGraph 在 super-step 边界打 checkpoint 支持回滚，文档里只有 §4.4.5 的 "回退 "动作但没说怎么持久化中间状态。可考虑借鉴 Claude Code 的 "git commit 当 checkpoint"模式，把每次 Frame 状态变更落到本地 git。

---

8. Error Handling?????— ?? ??

Harness 标准： 10 步 × 99% = 90.4% 端到端的复利效应; LangGraph 四类错误: Transient（ retry）、LLM-recoverable（回传 ToolMessage）、 User-fixable（ interrupt）、 Unexpected（ bubble up）; Stripe 重试上限 2 次。

理想架构对应 ：

- **§4.4.3**: "节点执行失败可触发重新设计节点的拆分、提示词，成功后更新节点（提升效率、节省 token） "——属于 LLM-recoverable 类。
- **§4.4.5**: 动作集中包含 回退： "当前方向是错的，回退到前面的第 5 步 "——这是模型层的自纠。

评估： ⚠ 思想到位，工程层缺失 ——

- 文档 只覆盖了 "模型可自纠 "那一类
- 缺: 瞬时错误的 retry/backoff、需要人介入的 interrupt 机制、意外错误的 bubble up
- 缺: 重试上限 （ Stripe 经验: 2 次）
- 缺: 错误复利的数学认识 ——长任务里 99% 单步成功率不够

建议：在 §4.5.2 补一节 "错误处理协议"，按 LangGraph 四分类显式约定 Frame 层与 Tool 层的错误传递方式。

---

9. Guardrails & Safety??????— ?? ????

Harness 标准：OpenAI 三层（input/output/tool guardrails）+ tripwire；Anthropic 把权限执行从模型推理中架构性分离——模型决定"想做什么"，工具系统决定"允许做什么"；Claude Code 分别上闸 ~40 个能力。

理想架构对应：

- **§4.1** 智能主体：人格与个性、价值观、灵魂——软性护栏。
- **§3.1** 自我意识：把人格从大模型内部移出来到编排器。
- **§4.4.4** 自洽要求："所有行动都要有合理的理由、逻辑链条、结果"——自我合理性检查。
- **§6.6** 价值观：利用"自我"提供终极目标指引。

评估：⚠ 维度不同，互补而非重合——

- 理想架构走的是"内化的人格 + 价值观自洽"路线，类似阿西莫夫机器人三定律
- Harness 标准走的是"外化的工程闸门"路线（input/output/tool guardrails + tripwire）
- 两者并不冲突：人格是"想不想做"，guardrail 是"能不能做"。两者都需要

建议：在 §5 工程实现里显式补一节"权限与护栏"——

- Tool 调用前的权限检查（继承 Anthropic 的 ~40 个能力闸门思路）
- 高风险操作（写入永久记忆、删除、外部 API 调用）需要显式确认或 tripwire
- 与"价值观/自洽"形成双层防护

---

10. Verification Loops??????— ?? ??

Harness 标准：三种 verification——Rules-based（测试/linter/类型检查）、Visual（Playwright 截图）、LLM-as-judge（subagent 评估）。Boris Cherny："给模型验证自己工作的方式，质量提升 2-3×"。

理想架构对应：

- **§4.4.4** 自洽要求：所有行动都要有合理理由、逻辑链条、结果——**LLM-as-judge** 的雏形。
- **§4.5.2** 下一步行动的决策："主动维护自洽：通过任务上下文、子任务总结等进行自洽判断"。
- **§3.2** 大范围、高深度的信息自洽："主动的检查和主动的行动逻辑判断"。

评估：⚠ 只覆盖了 **LLM-as-judge** 一种——

- "自洽判断"本质就是用 LLM 评估自己输出的合理性

- 缺 **rules-based**：单元测试、linter、类型检查这类确定性 **ground truth**
- 缺 **visual**：UI / 图片输出的截图比对

建议：把 §4.4.4 自洽要求扩成"三层校验"——

1. 规则校验：本地代码可判定的（schema 校验、断言、类型检查）
2. 执行校验：跑测试 / 编译 / 静态分析
3. 自洽校验：LLM 判官（已有）

参考 Martin Fowler/Thoughtworks 的 **guides**（**feedforward**）**vs sensors**（**feedback**）框架。

---

11. Subagent Orchestration?? agent ???— ? ??

Harness 标准：Claude Code 三种执行模型——Fork（字节级副本）、Teammate（独立终端 + 文件邮箱）、Worktree（独立 git 分支）；OpenAI 用 agents-as-tools + handoffs；LangGraph 用嵌套 state graph。

理想架构对应：

- **§4.4.3** 组织形式：分层树状结构：`"SubFrame List nested to tree" = LangGraph 嵌套 state graph 的同构表达。`
- **§4.4.6 Frame** 接口：子 Frame 完成通知经事件总线（`SUBFRAME_DONE`）——比 LangGraph 的 reducer 更解耦。
- **§6.3** 上下文调度管理器：sub-agent 执行特定子任务、并行多个上下文支持嵌套。

□□ □□ 完整 ——

- "嵌套 Frame + 事件总线"= LangGraph 的"嵌套状态图"
- **Frame ID +** 独立存储空间 **+ Session/Frame** 状态分离（§4.4.6）= Anthropic 的 Fork 思想

潜在差距：未讨论物理隔离层级（同进程/跨进程/跨机器）——Claude Code 的 Worktree 模式是文件系统级隔离，理想架构如果要支持长程并行子任务，需要决定隔离粒度。

---

12. Long-Horizon Execution????? / Ralph Loop ?— ? ?? + ??

Harness 标准：Anthropic 两阶段 Ralph Loop——Initializer Agent 搭环境（init script + progress 文件 + feature 列表 + 初始 commit），Coding Agent 每个会话读 git log + progress 文件自我定位，挑高优 feature → 干活 → commit → 写摘要。文件系统提供跨上下文窗口的连续性。

理想架构对应：

- **§4.5.4** 循环闭环：本能任务不断产生记忆，存入临时记忆；临时记忆 ↔ 本能目标 形成循环 = Ralph Loop 的本质。
- **§3.3** 自动进化、自动学习：“灵活的记忆接口 + 预设的学习本能任务”。
- **§6.4** 自动更新与学习：
  - 创造和更新 skills
  - 制造和整理 tools、脚本
  - 学习的方法学：软件工程开发流程
  - 复杂任务可经过非常多次尝试后解决，并永久学习
- **§4.5.2** 总结/抽象/记忆：主动提取和组织框架性的知识，存储到基础知识框架。

□□ □□ 完整 + 比 **Ralph Loop** 更高阶 ——

- Ralph Loop 是“用文件系统续命”
- 理想架构是“用 长时记忆 + 自动方法学进化 续命”——把跨会话连续性从 文件层 升到了 知识层
- **§4.4.5** 学习动作（“把指定的信息或笔记写入永久记忆”）= Ralph Loop 的“写摘要”，但语义化更强

潜在差距：Ralph Loop 用 **git commit** 作为天然 **checkpoint**，理想架构没有显式接入 git——可作为工程上的最小补充：每次 Frame 关键状态变更落一次 commit。

???????

3.1 ????

维度	评分	说明
概念完备性	**A**	12 项中 10 项有显式或等价表达
工程具体度	**B**	大量“目标级”描述，落地细节相对薄弱
创新点	**A**	“本能 + 人格 + Frame + 关键词树记忆”是有特色的组合
与 Harness 共识对齐度	**B+**	主要差异在 Output Parsing、Guardrails、Verification 三项的工程化

3.2 ???????????PDF ??????

1. 人格作为编排器一等公民（§3.1、§4.1、§6.6）——把价值观/自洽显式提到 prompt 最高层
2. **Frame** 作为统一抽象——同时是 task / context / 思维节点 / handle，比 LangGraph 的“node”语义更通用
3. 关键词树长时记忆——比 Claude Code 的平铺 `MEMORY.md` 结构化得多
4. 本能/调度器把“先思考再行动”内化为系统行为——而不是依赖模型每次自己决定

3.3 ??????????

优先级	缺口	对应 **Harness** 组件	最小补丁
P0	错误的分级处理	#8 Error Handling	§4.5.2 增加四类错误协议（瞬时 /可纠 / 可修 /意外） + retry 上限
P0	工程级护栏	#9 Guardrails	§5 增加 "权限与 Tripwire" 小节，与人格自洽形成双层防护
P1	校验回路三分类	#10 Verification	§4.4.4 自洽要求扩成 "规则 + 执行 + 自洽 "三层
P1	Output 选型	#6 Output Parsing	明确 Frame 动作集用原生 tool calling 实现
P2	Checkpoint 机制	#7 State / #12 Long-Horizon	借鉴 Claude Code，把 Frame 关键状态变更落 git commit
~~P2~~	~~Compaction 触发条件——~~	~~#4 Context Mgmt~~	□ □□□ — §4.4.8 三段式布局 + 删 MID 中段策略

3.4 ?????????

已对齐	待对齐
关键词树长时记忆 ↔ MindStore（ Qdrant + 关系边）	git checkpoint → 接入 Auto/ 流水线
Frame 状态 ↔ WeaveAgent Chronicle（ "原因 + 结果 "）	工程级 guardrail → WeaveAgent ToolRegistry 加权限闸
Sub-agent ↔ WeaveAgent SubFrame	错误四分类 → 当前 Chronicle 未区分
渐进披露工具 ↔ §4.4.7 分层路径 ToolRegistry	三层 verification → 接入 Qwen3-Embedding 训练数据校验

???????

> 理想架构在 "概念完整性 "和 "哲学高度 "上超过了 **Harness 12** 组件共识；但在 " 工程化细节 "——尤其是错误处理、权限护栏、校验三分类 ——还需要把 **Anthropic / OpenAI / LangChain** 已经踩过的坑显式吸收进来。
>
> 下一步最经济的演进路径 ：保留全部独有抽象（人格 / 本能 / Frame / 关键词树），把 P0 的两个缺口（错误协议 + 工程级护栏）作为下一版 `WeaveAgentDesign.md` 的增量补丁。

?????????

??????????????

[illegible] [illegible] [illegible]

????????

[illegible]
[illegible] [illegible]

[illegible] "[illegible]" [illegible]

- [illegible] / [illegible] [illegible] "[illegible]" [illegible]
- [illegible] / [illegible] [illegible] "[illegible]" [illegible] "[illegible]" [illegible]
- [illegible] [illegible] "[illegible] / [illegible] / [illegible] / [illegible] / [illegible] ……" [illegible]

[illegible]

> **[illegible] + [illegible] + [illegible] + [illegible] [illegible]**
>
> [illegible] "[illegible]"
> [illegible]

????????????????

[illegible]

1. ?? / ?????????

>

> "A 是 B 的上位概念" / "B 是从 A 派生出来的"。

- 有方向：`父 → 子`。
- 整体构成一张有向无环图（**DAG**）——一个概念可以挂在多个父节点下（"鲸鱼"既是"哺乳动物"也是"海洋生物"），但不允许成环。
- 这层结构是人类心智的脚手架：从顶层往下走，能定位到任何具体概念。

2. ??????????

> "A 和 B 相关，但说不清谁包含谁"。

- 无方向：`A ↔ B`。
- 表达"邻近性"——譬如"狗"和"狗粮"、"加密"和"哈希"，它们既不互相包含，也不互相派生，但提到一个就该想到另一个。

????

- 同一对概念可以同时挂这两种关系，互不影响。"哺乳动物 → 狗"是包含；"狗 ↔ 猫"是关联——这两件事天然分层。
- 没有第三种、第四种边。"相似""引用""反义"这些都被刻意排除：要么用关联表达，要么干脆不表达。语义类型越少，写入门槛越低，结构越稳定。
- 导航与展开走两套规则：从上往下找概念时，只沿包含边下钻；找到概念后展示"周边"时，沿关联边横向铺开。两层互不干扰。

??????????????????

绝大多数检索系统都在试图回答"有多像"。这套设计反其道而行：

> 匹配只有两种状态：命中 / 不命中。不打分、不排序、不做模糊比较。

具体到查询时：

- 用户给出一个查询词。
- 做最基础的归一化（大小写统一、标点空白剥离、全角半角拉平）。
- 在倒排索引里查：有就是有，没有就是没有。

这看起来过于简陋，但它换来三件事：

1. 结果可预期 ——同一个查询永远返回同一组节点，不会因为模型版本、嵌入空间漂移而变。
2. 维护成本极低 ——没有相似度阈值、没有 top-K、没有 "调参"。
3. 真正的 "找不到" 信号 ——
   倒排零命中是一个清晰的状态，可以触发完全不同的处理流程（见下一节）。

精确匹配的代价是 召回率低 ——用户写 "小猫"，库里只有 "猫"，倒排不会命中。这正是下一节要解决的问题。

??????????????????"??"

倒排零命中不是终点，而是另一条路径的起点：

> 沿着包含树（ **DAG**），让 **LLM**
> 一步步往下走，直到找到目标、否定所有候选、或者证明库里确实没有。

????

想象一个图书管理员：

- 读者问 "我想找一本关于深度伪造的书"。
- 管理员先看自己的目录卡片（倒排） ——没有 "深度伪造" 这张卡。
- 但他知道目录的顶层分类（ DAG 根节点），于是从 "科技" 一路往下走： "科技" → 人工智能 → 计算机视觉 → ....."，每到一层就判断 "是不是该往这下面找"。
- 走到某一层，他发现下面有 "图像合成 / GAN / 视频生成" ——他识别出 "GAN" 和读者要的东西最接近，匹配。

LLM 在这套系统里扮演的就是这位管理员： 它不评估相似度，它只做 "该往哪里走" 的导航选择。

????

1. ???????????

下钻不是从根节点盲目展开，而是先构造一组 种子节点 作为起点，混合多种信号：

- 直接的语言信号 ：查询词的子串出现在哪个节点的名字里？哪怕没精确命中，也是强信号。
- 最近访问偏置 ：用户刚才碰过哪些节点？大概率仍在同一片话题区域。
- 重要性偏置 ：度数高的节点（连接多）通常是话题枢纽，从它们出发更可能快速逼近目标。
- 结构兜底 ：如果上面三路都没有信号，就把所有 DAG 根节点作为起点 ——保证 fuzzy 查询永远能从 "图顶层" 启动。

这套加权的目的不是排出 "最像 "的节点，而是给智能体一个 信息丰富的起跑线
：既不空手出发，也不被冗余候选淹没。

2. LLM ??????

每一轮，智能体只看到极少量候选（默认不超过 10 个），每条候选只包含两件事：

- 节点的名字；
- 它距离起点走过几跳。

不暴露内部 **ID**、不暴露描述、不暴露信息。 原因有三：

- 节省 token；
- 防止 LLM 把 ID 字符串本身当成判断依据（幻觉的常见来源）；
- 强迫模型只用 "语义 "做决策，而不是 "字符相似 "。

3. ??????????

最关键的设计承诺： 没有节点会因为单轮容量限制被默默丢弃 。

- 探索过程维护一条全局的 BFS 队列；
- 每轮只从队首取少量候选喂给 LLM；
- 没看完的留在队列里，下一轮继续；
- 直到 LLM 明确判定 "这个就是 " / "都不是、给个建议名字 " / "队列被完整走完 "。

这样即使 LLM 一开始走偏了，只要兄弟分支还在队列里，最终仍能被探索到。 召回率由结构而非模型保证 。

4. ????????

- **match**（找到了）：单目标 → 命中；多目标 → 歧义返回。
- **jump**（往下走）：把指定节点的子节点提到队首，但 不丢弃兄弟分支 。
- **missing**（不在这条路上）：可以软提示某个分支不要继续走，也可以彻底放弃并建议一个新名字。
- **ambiguous**（候选都对，但分不出主次）：返回一组候选给上层处理。

四种动作覆盖了 "前进 / 后退 / 否决 / 终止 "的全部决策面，没有第五种。

????????????"??"

找到目标节点不是终点，用户往往还想知道 这个概念附近还有什么 。

设计上严格区分两个动作：

动作	沿什么走	给用户的语义
下钻（找到目标之前）	仅包含边（树结构）	"我在分类体系中往下找 "

动作	沿什么走	给用户的语义
展开（找到目标之后）	关联边 + 包含的子边（混合）	"目标节点的周边有什么"

为什么命中后也要走包含的子边？因为"狗"的"周边"理应包括"金毛 / 拉布拉多"等子类——它们和"狗"的关联性甚至比关联边更强。但祖先方向不进入展开结果——用户已经"在狗这里"，不必再被拉回"哺乳动物 → 动物"。要看祖先，得显式调用祖先子图的接口。

??????????

知识不仅有"概念"，还有"事实 / 资料 / 引用"。这套设计把它们彻底分开：

- 关键词节点：是概念骨架，名字唯一、有别名、有简短描述。
- 信息单元：是真正的内容（一段文字、一份链接、一段笔记）。
- 挂载关系：一条信息可以挂在多个关键词上；一个关键词可以收集多条信息。语义固定为"信息是关键词的来源 / 佐证 / 实例"，不再细分。

好处：

- 同一段资料不会被复制——它只存一份，多个相关关键词共享。
- 删除概念时，资料本身可以保留（因为别的概念可能还在引用它），只拆掉本节点和资料的链接。
- "什么是 X"和"X 这里有哪些资料"是同一次查询的两个层次：先命中节点，再列出该节点挂载的所有信息。

??????????????

写入只允许两种动作：

- 新增一行（创建）；
- 新增一行，标记为已删除（删除）。

永远不修改已有行。要"改"一个节点，就追加一条版本号 +1 的新行。

启动时扫一遍所有行，按 ID 取最新版本，丢弃删除标记的，重建所有内存索引。这意味着：

- 崩溃恢复几乎免费——文件本身就是事实来源，最多丢失最后一条未刷盘的记录。
- 变更日志天然存在——每一次写都留痕，将来要做撤销 / 审计 / 合并是直接的。
- 没有"数据库"——一组追加文件就是全部存储。没有 schema 迁移、没有索引重建脚本。

代价是文件会无限增长，需要定期压实（重写一次只保留最新版本）。但在概念量级是数千~数万的场景下，这点开销完全可以接受。

??LLM ???????????

理解这套设计的关键，是看清 LLM 做什么 和 不做什么 。

LLM ???

1. 导航判断 ——在概念骨架上选择往哪走、停在哪。
2. 缺失建议 ——当库里确实没有目标时，给出一个建议名字，让用户决定要不要新建。
3. 关联挖掘 （可选） ——给定一个节点，让模型推荐它可能 "相关但还没连 "的其他节点。

LLM ????

1. 不算相似度 ——相似度由倒排和子串匹配处理。
2. 不打分排序 ——没有 top-K 的概念。
3. 不感知 **ID**——只看名字和跳数。
4. 不参与存储 / 写入 ——LLM 的所有输出都要经过结构校验（动作合法、目标节点存在、非自环、不与现有边冲突）才会落盘；校验失败静默跳过，不污染图。

这条边界划得很清楚： **LLM** 是导航员，不是裁判，更不是仓库管理员 。

??????????

维度	向量库 / 嵌入检索	传统知识图谱	本设计
召回方式	相似度排序	关系类型查询	精确命中 + 智能体下钻
关系建模	隐式（向量空间）	多种边类型 + 本体	仅两类边，最小本体
写入门槛	低（嵌入即可）	高（要选关系类型）	中（要起名、挂位置）
结果稳定性	随模型 / 数据漂移	稳定	完全稳定
"找不到 "的语义	模糊（永远能返回 top-K）	清晰	清晰，且能触发下钻流程
维护成本	中（嵌入要重算）	高（本体要维护）	低（追加文件）
适用规模	百万级以上	任意	千~十万级概念

这套设计的 舒适区 是：中等规模、关系语义不需要细分、希望 **LLM** 增强但不希望被 **LLM** 主导的知识组织场景。它不试图取代向量库或图数据库 ——它给 "个人 /团队的概念档案 "提供了一个介于 wiki 和图谱之间的中间形态。

???????

> 用最小的本体（两类边） + 最严格的匹配（二态精确） + 最克制的 **LLM** 用法（只做导航） +
> 最朴素的存储（仅追加），换取一个能让人和模型协同维护、长期稳定演化的概念骨架。

Agent Harness ??????????? 12 ???

> 来源: Akshay Pachaar,《 The Anatomy of an Agent Harness》(2026-04-06)
> 推文: https://x.com/akshay_pachaar/status/2041146899319971922 本仓库归档: `Agent/design/harness.pdf`

????? Agent Harness

你已经搭过 chatbot, 也许还接了几个工具的 ReAct 循环 ——demo 跑得很好。可一旦要上生产，问题立刻浮现：模型忘掉三步前做过的事、工具调用静默失败、上下文窗口被垃圾塞满。

> 问题不在模型，而在模型周围的一切。

LangChain 在 TerminalBench 2.0 上做过一个 "反事实 "实验: 模型不变（同权重），只改外围基础设施，排名从 30 名开外飙到第 5。另一项研究让 LLM 自己优化基础设施，pass rate 达到 76.4%，超过人手设计的系统。

这套基础设施有了正式名字: **Agent Harness**（智能体外壳）。

1.1 ??

> **Harness** 是包裹 **LLM** 的完整软件基础设施：编排循环、工具、记忆、上下文管理、状态持久化、错误处理、护栏。

- Anthropic 的 Claude Code 文档: SDK 就是 "the agent harness that powers Claude Code"
- OpenAI 的 Codex 团队明确把 "agent" 与 "harness" 当同义词使用
- LangChain 的 Vivek Trivedy 给出了 经典口径 :

>

> **"If you're not the model, you're the harness."** （不是模型的部分，就都是 harness。）

1.2 ?????"Agent" vs "Harness"

- **Agent**: 用户感知到的 "涌现行为" ——目标导向、会用工具、会自我纠错的实体
- **Harness**: 产生这种行为的 机器

> 当有人说 "我做了个 agent"，本质上是: 做了个 **harness**，把它指向某个模型 。

1.3 ????·??????Beren Millidge, 2023?

裸 LLM = 一颗没有 RAM、没有硬盘、没有 I/O 的 CPU:

计算机	**LLM Agent**
RAM（快但小）	Context window
磁盘（大但慢）	外部数据库
设备驱动	Tool 集成
操作系统	**Harness**

??????

围绕模型的三层同心圆:

1. **Prompt Engineering**: 雕琢喂给模型的指令
2. **Context Engineering**: 管理模型在何时看到什么
3. **Harness Engineering**: 包含上面两者，再加上 完整的应用基础设施 ——
   工具编排、状态持久化、错误恢复、校验回路、安全约束、生命周期管理

> Harness 不是 prompt 的包装。它是让自主 agent 行为成为可能的 完整系统 。

????? Harness ? 12 ???

> 综合 Anthropic、OpenAI、LangChain 与广泛实践者社区的工程经验。

---

1. Orchestration Loop??????

整个心脏。实现 **TAO** 循环（**Thought-Action-Observation**），也叫 ReAct loop:

```
组装 prompt → 调 LLM → 解析输出 → 执行 tool calls → 喂回结果 → 循环至完成
```

机制上往往 只是一个 **while** 循环。复杂度全在循环管理的东西，而不在循环本身。

> Anthropic 把自家 runtime 称为 **"dumb loop"**：循环越笨越好，所有智能都留在模型里 ，harness 只管轮次。

---

2. Tools????

工具是 agent 的 "手 "。以 Schema 形式声明（name、description、parameter types），注入到 LLM 上下文中告诉模型有哪些能力可用。 Tool 层负责:

- 注册（registration）
- Schema 校验（validation）
- 入参提取（argument extraction）
- 沙箱执行（sandboxed execution）
- 结果捕获（result capture）
- 把结果格式化回模型可读的 observation

典型实现 :

- **Claude Code**：6 大类工具 ——文件操作、搜索、执行、Web 访问、代码智能、subagent 派生
- **OpenAI Agents SDK**：function tools（`@function_tool`） + hosted tools（WebSearch / CodeInterpreter / FileSearch） + MCP server tools

---

3. Memory????

记忆按时间尺度分两层:

- **Short-term memory**：单会话内的对话历史

- **Long-term memory**: 跨会话持久化
  - Anthropic: `CLAUDE.md` 项目文件 + 自动生成的 `MEMORY.md`
  - LangGraph: namespace 组织的 JSON Stores
  - OpenAI: SQLite/Redis 支持的 Sessions

Claude Code 的三层结构：

1. 轻量索引（每条 ~150 字符，始终常驻）
2. 主题详情文件（按需取入）
3. 原始 transcript（只能搜索访问）

> 关键原则：**agent** 把自己的记忆只当作"提示"，行动前必须先去验证真实状态。

---

4. Context Management???????

很多 agent 在这里静默失败。根本问题是 **Context Rot**：

- 关键内容落在窗口中段时，模型性能下降 **30%+**（Chroma 研究，与 Stanford "Lost in the Middle" 互证）
- 百万 **token** 窗口也照样会随上下文增长发生指令遵循退化

生产策略：

策略	做法
Compaction	临近上限时摘要历史（Claude Code 保留架构决策与未解 bug，丢弃冗余工具输出）
Observation Masking	屏蔽老的工具输出但保留工具调用本身（JetBrains Junie）
Just-in-time Retrieval	只持有轻量标识符，按需加载（Claude Code 用 grep / glob / head / tail，而不是直接读全文件）
Sub-agent Delegation	子 agent 充分探索，但只返回 1k–2k token 的浓缩摘要

> Anthropic context engineering 指南的目标：找到使期望结果概率最大化的"最小高信号 **token** 集合"。

---

5. Prompt Construction?Prompt ???

每一步模型实际看到的东西。层级化拼装：

```
System Prompt → Tool Definitions → Memory Files → Conversation History → Current User Message
```

OpenAI Codex 采用严格优先级栈：

- Server-controlled system message（最高）
- Tool definitions
- Developer instructions
- User instructions（级联的 AGENTS.md，**32 KiB** 上限）
- Conversation history

---

6. Output Parsing??????

现代 harness 普遍依赖原生 **tool calling**：模型直接返回结构化的 `tool_calls` 对象，不再需要解析自由文本。

```
有 tool calls？ → 执行 + 循环
没 tool calls？ → 这就是最终回答
```

结构化输出：OpenAI 与 LangChain 均通过 Pydantic 模型做 schema 约束。

老式的 `RetryWithErrorOutputParser`（把原 prompt + 失败补全 + 解析错误一并喂回模型）作为兜底仍然可用。

---

7. State Management??????

四种代表性流派：

框架	做法
LangGraph	类型化字典在图节点间流动，reducer 合并更新；super-step 边界打 checkpoint，可断点恢复、时间旅行调试
OpenAI	四种互斥策略：应用内存 / SDK Sessions / 服务端 Conversations API / 轻量的 `previous_response_id` 链式拼接
Claude Code	用 **git commit** 当 **checkpoint**，**progress** 文件当结构化草稿本

---

8. Error Handling??????

为什么这件事至关重要 :

> 10 步流程，每步成功率 99% → 端到端只剩 **~90.4%**。错误会快速复利 。

LangGraph 的四类错误 :

类型	处理
Transient (瞬时)	retry + backoff
LLM-recoverable (模型可自纠)	错误以 `ToolMessage` 形式回传，让模型自我调整
User-fixable (需要人介入)	中断（ interrupt) 等待人类输入
Unexpected (意外)	向上冒泡，便于调试

- Anthropic: 在 tool handler 内部捕获失败，作为 error result 返回， 保持循环不中断
- Stripe 生产 harness: 最多重试 **2** 次

---

9. Guardrails and Safety???????

OpenAI SDK 的三层 :

- **Input guardrails**: 在第一个 agent 上运行
- **Output guardrails**: 在最终输出上运行
- **Tool guardrails**: 在每次工具调用上运行
- **Tripwire** 机制: 触发即立刻停机

Anthropic 的核心理念 : 把权限执行从模型推理中架构性分离 。

- 模型决定 想做什么
- 工具系统决定 允许做什么

Claude Code 把约 **40** 个独立工具能力 分别上闸，分三阶段:

1. 项目加载时建立信任（ trust establishment)
2. 每次调用前检查权限（ permission check)
3. 高风险操作要求显式用户确认

---

10. Verification Loops??????

这是把玩具 **demo** 与生产 **agent** 区分开的关键。

Anthropic 推荐三种:

- **Rules-based feedback**: 测试、 Linter、类型检查
- **Visual feedback**: UI 任务用 Playwright 截图比对
- **LLM-as-judge**: 另一个 subagent 来评估输出

> Claude Code 作者 **Boris Cherny**: 给模型一个能验证自己工作的方式，质量提升 **2** 到 **3** 倍。

---

11. Subagent Orchestration?? agent ???

Claude Code 的三种执行模型 :

模式	含义
Fork	字节级完整副本 (复制父上下文)
Teammate	独立终端面板 + 文件邮箱通信
Worktree	自己的 git worktree, 每个 agent 独立分支

- **OpenAI SDK**: agents-as-tools (专家处理有界子任务) + handoffs (专家完全接管)
- **LangGraph**: subagent 实现为嵌套 state graph

---

12. Long-Horizon Execution????? / Ralph Loop ?

> 原文标题说 12 个组件、正文显式编号到 11; 这里把 "跨上下文窗口的长程执行 "列为第 12 项 ——它在 PDF 中以独立模式 **"Ralph Loop"** 详细展开。

问题： 单次会话可能跑 1–2 轮就结束，但 复杂重构任务可能跨多个上下文窗口 —— 简单的循环根本撑不到完成。

Anthropic 的两阶段 **Ralph Loop**:

1. **Initializer Agent**: 搭好环境
   - init 脚本
   - progress 文件
   - feature 列表

- 初始 git commit

2. **Coding Agent**（之后每个会话循环）：
   - 读 git log + progress 文件 自我定位
   - 挑优先级最高的未完成 feature
   - 干活 → commit → 写摘要

> 文件系统在跨上下文窗口的场景里提供连续性。

终止条件 是分层的：模型给出无 tool call 的回答 / 达到最大轮次 / token 预算耗尽 / guardrail tripwire 触发 / 用户中断 / 安全拒绝。

??????? 7 ???

把 12 个组件串起来跑一圈：

步骤	内容
1. Prompt Assembly	装配： system prompt + tool schemas + memory files + conversation history + 当前消息。 重要内容放在 **prompt** 的开头和结尾 （ Lost in the Middle）
2. LLM Inference	送进模型 API，生成 text / tool call requests / 两者都有
3. Output Classification	文本无 tool call → 结束；有 tool call → 执行； handoff → 切换当前 agent 重启
4. Tool Execution	校验入参 → 检查权限 → 沙箱执行 → 捕获结果。 只读并发，写操作串行
5. Result Packaging	工具结果格式化为 LLM 可读消息； 错误也以 **error result** 形式回传，模型可自纠
6. Context Update	追加到对话历史；接近窗口上限就触发 compaction
7. Loop	回到第 1 步直到终止

> 简单问题 1–2 轮即可；复杂重构可能跨数十次 tool call 与多轮上下文。

?????????

框架	关键抽象	特点

Anthropic Claude Agent SDK	单一 `query()` 函数，返回流式异步迭代器	"dumb loop"；Gather-Act-Verify 循环（gather context → take action → verify results → repeat）
OpenAI Agents SDK	`Runner` 类（async/sync/streamed）	**Code-first**：用原生 Python 写工作流，不用 graph DSL
OpenAI Codex Harness	三层架构	Codex Core（agent + runtime） + App Server（双向 JSON-RPC） + 客户端（CLI / VS Code / Web）。**"Codex 模型在 Codex 表面感觉更好"，因为共享同一 harness**
LangGraph	显式 state graph：`llm_call` 与 `tool_node` + 条件边	从已废弃的 `AgentExecutor` 进化而来；原生支持多 agent
LangChain Deep Agents	明确使用 "agent harness" 一词	内置工具 + 规划（`write_todos`） + 文件系统 + subagent + 持久记忆
CrewAI	Agent / Task / Crew + Flows	角色制（role/goal/backstory/tools）；Flows 提供 "intelligence where it matters"的确定性骨架
AutoGen → Microsoft Agent Framework	Core / AgentChat / Extensions 三层	五种编排模式：sequential / concurrent (fan-out/fan-in) / group chat / handoff / **magentic**（manager agent 维护动态任务账本协调专家）

?????? Harness ? 7 ?????

#	决策	关键 **trade-off**
1	单 **agent vs** 多 **agent**	Anthropic & OpenAI 都建议 先把单 **agent** 做到极致 。多 agent 增加额外的路由 LLM 调用、 handoff 时的上下文丢失。 只在工具数量超过约 **10** 个重叠工具，或任务领域明显分离时才拆分
2	**ReAct vs Plan-and-Execute**	ReAct 每步都交错推理与动作（灵活但每步成本高）； Plan-and-Execute 把规划与执行分离。 **LLMCompiler** 报告比顺序 **ReAct** 快 **3.6×**
3	上下文窗口管理策略	五种： time-based clearing / summarization / observation masking / structured note-taking / sub-agent delegation。**ACON** 研究： **26–54% token** 削减 **+ 95%+** 准确率保留（关键： reasoning trace 优先于 raw tool output）

#	决策	关键 **trade-off**
4	校验回路设计	Computational（测试、 linter） = 确定性 ground truth； Inferential（ LLM-as-judge） = 抓语义问题但增加延迟。 Martin Fowler / Thoughtworks 把它们框成 **guides**（ feedforward ，行动前引导） vs **sensors**（ feedback ，行动后观察）
5	权限与安全架构	Permissive （快但险，自动批准多数动作） vs Restrictive （安全但慢，每个动作都需批准）。 取决于部署场景
6	工具范围策略	更多工具往往意味着更差的性能 。 Vercel 从 v0 删掉 80% 工具，结果反而更好； Claude Code 用懒加载实现 95% 上下文削减。 原则：只暴露当前步骤所需的最小工具集
7	**Harness** 厚度	多少逻辑放在 harness 、多少放在模型里。 **Anthropic** 押注薄 **harness** 与模型进化 ；图框架押注显式控制。 **Anthropic** 会随新模型版本，定期从 **Claude Code** 的 **harness** 里删掉规划步骤 （因为模型自己内化了）

??????????

7.1 ???????????????

- 建筑脚手架 = 临时基础设施 ，让工人能去到原本到不了的位置
- 它 不做施工本身 ；但 没有它，工人上不到顶层
- 建筑完工时脚手架就被拆除

> 核心洞见：模型变强 → **harness** 复杂度应该下降。

Manus 6 个月内重写 5 次，每次重写都在删复杂度 ：

- 复杂的工具定义 → 通用 shell 执行
- "Management agents" → 简单的结构化 handoff

7.2 ??????

模型现在是 带着特定 **harness** 一起做 **post-train** 的:

- Claude Code 的模型学会了用它训练时配套的那套 harness
- 改工具实现可能因为这种紧耦合而降低性能

7.3 "?????"??

> 如果换用更强的模型时，性能能 **scale up** 而不需要追加 **harness** 复杂度，这个设计就是好的。

??Harness ???

两个用相同模型的产品，可以仅因 harness 设计的差异而 性能天差地别 。TerminalBench 的证据是清晰的: 只改 **harness**， **agent** 名次能跨越 **20+** 位 。

Harness 不是已解决问题，也不是商品化层。 这里才是真正的硬工程:

- 把上下文当稀缺资源管理
- 设计能在错误复利之前抓住失败的校验回路
- 构建能提供连续性又不幻觉的记忆系统
- 在 "多搭脚手架 "和 "留给模型自己 "之间做架构性押注

> 趋势是 **harness** 在变薄 ——但它不会消失 。
> 即便最强的模型，也需要东西来管它的上下文窗口、执行它的工具、持久化它的状态、校验它的工作。

> 下一次你的 **agent** 出问题时，别怪模型，看 **harness**。

???????AllData????

对照仓库里现有 Agent 设计文档（`一种理想的智能体编排架构.md`、`基于关键词的知识树系统设计方案.md`）以及 WeaveAgent，建议的对齐方向：

Harness 组件	与本仓库的映射
Memory 三层结构	与 MindStore 的多层级摘要（pyramid + MarkdownLogic）高度同构，可以把"轻量索引/详情/原文"模式直接落到 MindStore 上
Context Management — Compaction & Tool Output Offloading	直接对应 `project_chronicle_grain.md`：WeaveAgent Chronicle 只记"原因 + 结果"，不复述工具返回——这就是 PDF 里说的 **tool output offloading**
Verification Loops（rules / visual / LLM-as-judge）	后续 Agent 输出质量评估的标准范式，Qwen3-Embedding 训练数据增强环节也可以接 LLM-as-judge
Long-Horizon Execution / Ralph Loop	Auto/ 流水线（PDF→MD→Logic→Embedding→MindStore）天然是长程任务，可以引入 progress 文件 + git checkpoint 的 Ralph Loop 模式
Harness 厚度原则	随着 Qwen3-Embedding checkpoint 持续训练增强，本地 Agent 的 harness 应有意识地变薄——删功能比加功能更重要
工具范围 **— "**少即是多**"**	Vercel 砍 80% 工具反而更好的经验，对 Auto/ 中各 `gen_*.py` 也成立：合并语义重复步骤优于堆功能

Agentic Engineering ?????

?????????-Agent???

我们知道当前不管是 Code、Cowork 以及 比较火的 OpenClaw 等助手类的 Agent，不外乎两个特点

1. 所有的自洽和逻辑都只能维持在当前的上下文
    1. 上下文可以被动态的追加、压缩、整理、拼接
2. 外挂一些固定的处理接口: Task、SubAgent、Skills、Mcp等等
    1. 支持额外的记忆
    2. 支持和现有一些软件的控制、执行
    3. MCP 支持现有一些服务的的对接

?????

1. 上下文局限性，决定了不能进行 大范围 ，高深度 的思考和自洽
    1. 使用有限的上下文处理经过拆分的大范围的知识片段会非常的繁琐
    2. LLM本身存在的抽象深度问题，智商不够
    3. 虽然现在的上下文长度可以达到 1M，但是实际的理解深度还是非常欠缺
    4. 依赖把全部的信息塞到 LLM的上下文来维持自洽
2. 动态学习能力的欠缺
    1. 没有主动的学习和总结的能力
3. 效率低
    1. LLM需要深度思考
    2. 拆解大任务成很多的小任务
4. 智能体在遇到困难时陷入低效试错循环，而非通过反思实现突破
5. 利用 todo list工具，进行简单的线性任务规划
    1. 如果中间步骤执行异常，将难以处理
6. 黑盒式的编程，容易造成疯狂堆砌「屎山」
    1. 不能做到逐步的，有逻辑的，教程式的进行代码编写
    2. 不能做到和开发者匹配的开发节奏

????

1. 复杂非直接可读代码：流程图的 xml， PCB的 xml
2. 编程问题的测试和复现比较困难
    1. 需要结合系统服务，图形界面的应用程序
    2. 需要复杂的测试步骤，键盘输入测试输入法
    3. 需要其他的网络环境
    4. 随机问题
3. 单个文件超出上下文就很难处理
    1. 超大代码文件的整理
4. 大范围的批量编码会有问题

1. 把所有 “_”开头的命名都去掉 "_"，这个任务会导致大量的 token消耗，而且导致代码修改不全 / 命名冲突出错
5. 不自洽，不能理解用户问题的不完整信息，从而继续咨询补充信息
    1. 生成一个解析 json的图形化界面，只会根据已经有的一个 json里面的字段做支持，而不能根据 json的格式进行自动递归的解析所有可能的字段，不能从更高级的维度进行工作的设计
6. 虽然可以通过 CLAUDE.md来部分提示工程信息，但是复杂代码的逻辑关系，每次都要 LLM重新使用 tool探索，非常浪费，而且可能造成遗漏。这就是和人类工程师一个比较大的差异

????????

1. **LLM**比喻成一个计算器 ，那么 Agent编排的目标就是，使用计算器开发一个编译器
2. 信息分层处理，分层表达，以实现全局的自洽
3. **LLM**直接写代码执行任务，比通过文本传参数调用工具更高效
4. TodoWrite类语义是 LLM一种内置的自动编排技能
5. 能扩展上下文的理解广度和深度
6. 递归式 AI
7. 不仅仅用于软件编程、工程设计，也可以用于深度研究、长期科研

???

作为一个工具，虽然不是所有的场景需要人性化，比如编译器，秒表。但是很多工具的瓶颈就是缺少人性化。

比如

1. 个人助理 Agent在比价需要购买的物品过程中遇到的各种非正常情况的不正常处理
2. 我想去洗车，洗车店距离我家 50 米，我应该开车过去还是走过去?
3. AI Coding 工具，会反复询问执行权限，尽管同一个项目我已经默认同意了很多次，但是还是机械得执行 Prompt的准则，这里和完全开放权限不同，需要的是能自动判断我的预期的同意
4. 记忆的自动遗忘，人类助理，需要理解人类的记忆习惯
5. 记忆内容优先级排序
6. 比较关心的关键词提取
7. 任何响应，都是先思考，再行动
8. 自动触发检索记忆
9. 自动的反思总结

要让人类觉得 **AI**很聪明，很听得懂人话，这里面还缺了一个非常重要的就是机器的人性化

人性化能无缝的捕获人类指令中的一些隐含的意思，有了人性化之后就会显得很容易听懂，很能理解你的感受

，

在处理人类的需求的时候，比如说修改代码，作为人类首先会主动地去了解一下整个工程，而不是像现在的 AI 一样，直接上手就修改某个文件，这就是人性和自洽

人性化的可能途径和前提:

1. 支持 意识 和 自我 的信息处理，也就是情商，能理解和处理人类的一些 抽象的高层级的 复杂情感、人文信息
2. 具有存储和实现意识 /自我的实际载体

Harness

这套方法不是理论上的概念，而是一种 工程结构（ **Engineering Structure**）：让 Agent 不再 “ 一口气做完所有事 ”，而是像一个真正的软件工程师一样：

- 从一个明确定义的 *feature list* 开始
- 初始化项目骨架
- 每轮做一小步、只做一个 feature
- 每轮产生可追踪的 artifact 和 commit
- 保持项目状态的连续性
- 通过日志、文件结构、状态缓存持续 “记住 ”上下文

最关键的约束: Agent 永远在 “小步快走 ”，这样才能跨数十轮保持稳定

- 拥有持久化工作空间（ workspace）
- 拥有 feature list / task list
- 拥有日志、状态、内存
- 通过一轮轮 “patch + commit” 前进
- 由 harness 保证其 可控性、稳定性和可追踪性

Agent Harness 是包裹在 **LLM** 外层的一套编排系统。它的核心作用是将不确定的模型行为转化为稳定、高效的生产力。

- **Instructions** (指令系统)： 包含 System Prompt 和动态规则（ Rules）。
- **Tools** (工具集)： 赋予模型 “手脚 ”，如文件读写、终端执行、搜索能力。
- **User Messages** (交互流)： 用户的指令以及上下文的维护。
- 实现工具。 给 agent 一双手。文件读写、 Shell 执行、 API 调用、浏览器控制、数据库查询。每个工具都是 agent 在环境中可以采取的一个行动。设计它们时要原子化、可组合、描述清晰。
- 策划知识。 给 agent 领域专长。产品文档、架构决策记录、风格指南、合规要求。按需加载（ s05 ），不要前置塞入。 Agent 应该知道有什么可用，然后自己拉取所需。
- 管理上下文。 给 agent 干净的记忆。子 agent 隔离（ s04）防止噪声泄露。上下文压缩（ s06 ）防止历史淹没。任务系统（ s07）让目标持久化到单次对话之外。
- 控制权限。 给 agent 边界。沙箱化文件访问。对破坏性操作要求审批。在 agent 和外部系统之间实施信任边界。这是安全工程与 harness 工程的交汇点。
- 收集任务过程数据。 Agent 在你的 harness 中执行的每一条行动序列都是训练信号。真实部署中的感知 -推理 -行动轨迹是微调下一代 agent 模型的原材料。你的 harness 不仅服务于 agent -- 它还可以帮助进化 agent。

??

1. [illegible] OpenClaw[illegible] Claude Code
[illegible]
[illegible]
2. [illegible] harness[illegible] skills[illegible]
3. OpenClaw[illegible] Agent
[illegible] AGI
[illegible]
4. [illegible] [illegible] [illegible]

??Agent???

Agent?????

1. 这些本应被封装为「日常 AI工作流」的能力，却仍被塞进一个通用聊天框里手工完成。
2. 这正是留给 AI
   创业者的机会，我们不该让普通人用临时脚本搭建自己的「购房智能代理」，而应当创建一个个可复
   用、可协作、可沉淀的垂直 AI
   应用。这些应用能自动聚合多源文档、动态构建决策知识图谱、实时比对市场数据、生成合规话术建
   议等。这样的垂直 AI应用，
   以真实生活任务为中心，封装提示工程、记忆管理、多模态上下文维护
   ，从而构建辅助人类做判断的一体化智能工作台。
3. AI时代的 Facebook或 Google还尚未创立。当下竞争激烈的基础模型和 GPU
   属于基础设施范畴。真正的应用还没有出现
4. 理论有一家创业公司使用 Gemini 3来进行上下文设计，然后再把结果输入到 OpenAI
   模型中去执行，他们会根据新模型的发布情况不断进行替换，每个类别的智能体工作中表现最佳的模
   型可能都不同。而他们之所以能够这样做，是因为他们有独属于自己的模型评估体系。
   作为一家垂直领域的 **AI**
   智能体公司，他们的核心壁垒不是自己的模型，而是私有的评测数据集 。
5. 当前模型「能够做到的事情」，与人们「实际使用 AI
   的方式」（产生效果）之间，存在巨大的断层。因此，在 2026年， OpenAI
   将继续前沿研究，同时重点投入于应用层、系统层、人机协同，尤其强调医疗、商业和日常生活场景
   。

??????

1. MemRL 证明了，一个冻结的大脑，配合一个不断自我进化的记忆系统，就能实现持续的终身学习（
   Lifelong Learning）
2. 智能不是玄幻的，也可以用算法表达
   1. 人工开发的分类算法，被梯度下降取代，那么代表意识决策的源头，
      第二系统的自动化实现，也需要靠人工规则吗? agent就是在做这件事情
   2. 从 fast rcnn的动态选择到各种 LLM自然语言的逻辑处理
   3. 主体智能，自动 agent，自主意识动作，无需编写规则， 运行态的 强化学习
   4. 意识系统，更高级别的抽象， 层层递进才有实现的可能
3. 几乎所有注意力机制、本地记忆结构，乃至优化器本身，其实都可以视为联想记忆的特例

当前的方法

1. 上下文压缩
2. 集成 RAG
3. 递归语言模型 RLM ， token代理， Python REPL交互式编程环境
   https://mp.weixin.qq.com/s/Kg5oiN4LUWPDuW6ngTIP5A

1. [illegible] -[illegible]
2. [illegible] REPL[illegible]
3. [illegible]

Agent????????

1. [illegible]
2. [illegible] **Agent**[illegible]