

Agent 的两朵乌云：实时与环境交互，从经验中学习

Bojie Li

Co-Founder & Chief Scientist, Pine AI

2025 年 10 月

Press Space for next page →

物理学的大厦即将建成，只有两朵乌云……

演讲大纲

🌩️ 第一朵乌云：实时交互核心挑战

- 语音交互的高延迟（数十秒）
 - GUI 操作比人类慢 3-5 倍
 - 传统 ReAct 循环的串行瓶颈
- ### 技术突破
- **SEAL 架构 (Streaming, Event-driven Agent Loop)**

- 感知层：流式处理语音信号
- 思考层：观察、思考、行动异步进行的 Interactive ReAct
- 执行层：反馈闭环的 VLA / TTS

🌩️ 第二朵乌云：从经验中学习核心挑战

- 每次任务都从零开始
- 无法积累领域知识
- 缺乏任务熟练度提升

Agent 从经验中学习的三大范式

1. 后训练 (**Post-training**):
 - SFT 记忆 vs RL 泛化
2. 上下文学习 (**In-context Learning**):
 - Context ≠ 记忆
3. 外部化学习 (**Externalized Learning**):
 - 知识库：持久化经验存储
 - 工具生成：Agent 自我进化

Agent 的两朵乌云：实时交互、从经验中学习

OpenAI 研究科学家姚顺雨指出的第一个问题：Agent 办事过程中缺少与真人交互

Shunyu Yao
姚顺雨



The Second Half

tldr: We're at AI's halftime.

Inertia is natural, but here is the problem. AI has beat world champions at chess and Go, surpassed most humans on SAT and bar exams, and reached gold medal level on IOI and IMO. But the world hasn't changed much, at least judged by economics and GDP.

I call this the **utility problem**, and deem it the most important problem for AI.

Perhaps we will solve the utility problem pretty soon, perhaps not. Either way, the root cause of this problem might be deceptively simple: **our evaluation setups are different from real-world setups in many basic ways**. To name two examples:

- **Evaluation “should” run automatically**, so typically an agent receives a task input, do things autonomously, then receive a task reward. But in reality, an agent has to engage with a human throughout the task — you don't just text customer service a super long message, wait for 10 minutes, then expect a detailed response to settle everything. By questioning this setup, new benchmarks are invented to either engage real humans (e.g. [Chatbot Arena](#)) or user simulation (e.g. [tau-bench](#)) in the loop.

Agent 的两朵乌云：实时交互、从经验中学习

OpenAI 研究科学家姚顺雨指出的第二个问题：缺少从经验中学习的机制

- **Evaluation “should” run i.i.d.** If you have a test set with 500 tasks, you run each task independently, average the task metrics, and get an overall metric. But in reality, you solve tasks sequentially rather than in parallel. A Google SWE solves google3 issues increasingly better as she gets more familiar with the repo, but a SWE agent solves many issues in the same repo without gaining such familiarity. We obviously need long-term memory methods (and [there are](#)), but academia does not have the proper benchmarks to justify the need, or even the proper courage to question i.i.d. assumption that has been the foundation of machine learning.

第一部分：Agent 实时与环境交互

交互的对象：

- 人类：通过实时语音进行对话与协作。
- 互联网：操作电脑，浏览网页，使用软件。
- 物理世界：控制机器人，与真实环境互动。

语音 Agent 的实时交互挑战

根本矛盾：串行处理 vs 实时需求

- 必须等待：先听完才能想，想完才能说
- 阻塞等待：每个环节都成为瓶颈
 - 用户说话结束（VAD）→语音识别（ASR）→完整句子
 - 完整句子→带思考的LLM→思考后的完整输出
 - 完整输出→拆分成句子→语音合成（TTS）→语音回答
- 累积延迟：总延迟远超人类容忍度

⚡ 快慢思考的两难困境

- 深度思考：推理质量高，但CoT需要10+秒
 - 用户：“帮我订一个机票”
 - Agent：“（思考10秒）... 用户已失去耐心”
- 快速响应：延迟<1秒，但容易出错
 - 不假思索地同意不合适的方案
- 关键问题：无法在听的同时进行预判和思考

技术瓶颈：每一步都在等待

感知阶段

- 语音：等待整句结束才转文字，延迟高；送入语音识别模型的语音碎片化，识别准确率低
- 视觉：2K tokens 截图的 prefill 延迟高

思考阶段 (5-20 秒)

- 必须拿到完整输入才开始思考
- 无法预判用户意图
- test-time scaling 让延迟雪上加霜

执行阶段

- 思考结束才开始行动（说话、操作鼠标键盘）
- GUI 操作每步都要重新截图、思考

我们的架构创新：SEAL (Streaming, Event-driven Agent Loop)

核心思想：将所有交互抽象为异步事件流，实现低延迟、可中断的实时互动

1. 感知层 (Perception)

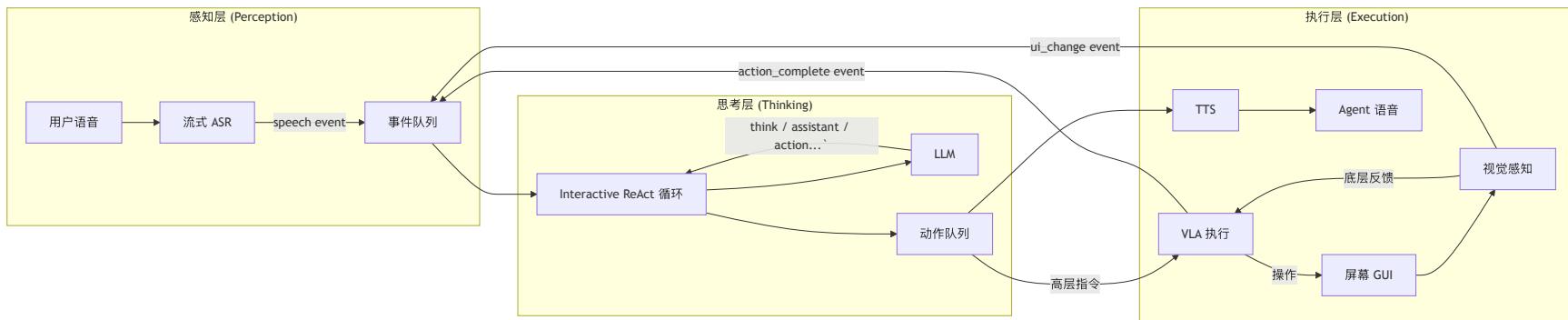
将连续的现实世界信号 (语音、GUI视
频) 转换成离散的事件流。

2. 思考层 (Thinking)

异步处理事件，边听边想，边想边说，
生成交错的思考和动作序列。

3. 执行层 (Execution)

将离散的动作指令转换回连续的现实世
界信号 (TTS语音, 鼠标移动)。



第一层：感知层 (Perception Layer)

将连续的现实世界信号，转换成离散的事件

输入 (Input)

连续信号：语音流、GUI 视频流

输出 (Output)

离散事件：`speech_start` , `interrupt` , `laugh` , `speech_fragment` , `ui_change` 等

流式语音感知模型：替代 VAD + ASR

传统 VAD + ASR 级联架构的问题

VAD + ASR 的局限性：

- 延迟累积：VAD 检测延迟 + ASR 处理延迟
- 信息丢失：VAD 仅输出二值信号，丢失声学细节
- 错误传播：VAD 模型很小，误判情况多，导致 ASR 漏识别或误触发
- 上下文缺失：ASR 分段识别语音，破坏语义连续性，导致语音识别准确率低

实际场景中的问题

- 打断不够智能：不管遇到什么声音，哪怕是对方附和一句“嗯”，都打断 AI 的说话，导致交互不自然
- 语音识别缺少上下文，准确率低：对邮箱地址、品牌名、人名、领域专有名词等需要上下文的场景，错误率尤其高
- 情感信息丢失：语音中的叹气、笑声等非语言信息
- 背景信息丢失：环境是否嘈杂、是否有背景音乐等无法被感知

流式语音感知模型：替代 VAD + ASR

基于开源自回归 LLM 的流式语音感知模型

- 与 whisper 等传统 ASR 模型不同，使用自回归架构，降低语音识别延迟
 - 流式处理输入的语音 tokens
 - 流式输出文本和声学事件
- 基于开源 LLM 后训练
 - 保留对话上下文，支持 in-context learning，从而对用户个人信息、领域专业名词等的识别准确率显著提升
 - 具备世界知识和 common sense，对品牌名、金额等的识别率显著提升

输出信息丰富，不仅包括文本，还包括声学事件

Text Tokens: 实时转写的文本片段

Special Tokens (声学事件) :

- <speak_start> 用户开始说话
- <speak_end> 用户结束发言
- <interrupt> 检测到打断意图
- <emotion:happy> 情感标记
- <laugh> <sigh> 副语言信息
- <music> 环境声音信息

第二层：思考层 (Thinking Layer)

基于事件驱动循环，实现可中断、异步的边听边想、边想边说

输入 (Input)

离散事件流 (from Event Queue)

输出 (Output)

交错的思考与动作指令 (to Action Queue)

核心创新：交互式 ReAct (Interactive ReAct)

核心思想：打破僵化循环，实现观察、思考、行动的灵活交织

传统 ReAct: 僵化的 OTA 循环



Interactive ReAct: 灵活交织的 OTA



局限性:

- 固定循环: 必须完成整个 OTA 循环
- 思考易被打断: 新输入导致思考丢失
- 响应延迟高: 深入思考时陷入沉默

- O = 观察 (Observation)
- T = 思考 (Thinking)
- A = 行动 (Action)

优势:

- 边听边想: 新观察可随时插入
- 边想边说: 快速思考后立即响应
- 统一上下文: 保持思考连贯性

SEAL 思考层：可中断的 Interactive ReAct 循环

关键洞察：LLM 思考速度远超语音 I/O，充分利用"空隙时间"

🧠 LLM 处理速度

- 输入处理: **500+ tokens/秒**
- 思考/输出: **100+ tokens/秒**

🎙 语音 I/O 速度

- 语音输入/输出: 仅 **5 tokens/秒**
- 速度差: **20-100倍**

在观察（语音输入）和行动（语音输出）之外的"空隙时间"里，我们有充足的时间进行深度思考！只是僵化的观察-思考-行动循环没有充分利用这些空隙时间！

⚡ 快思考 → 🐢 慢思考 → 🐌 持续思考

1. 快速响应 (**0.5秒**): 50 tokens 快速思考 → 立即给出初步回应 (5秒内)
2. 深入分析 (**5秒后**): 500 tokens 慢速思考 → 生成更完善的回答
3. 持续思考 (**需要时**): 如果 500 tokens 仍不够, 继续思考 5 秒 → 继续生成回答, 直到思考和回答完毕。如果需要多轮思考, 结果就是不断输出当前这轮思考的总结, 像人一样 "边想边说"。

Interactive ReAct: 边听边想 (Think while Listening)

优雅地处理对话中的打断

传统 **ReAct**: 用户一打断, 之前的思考就全部作废, 必须从头再来。

Interactive ReAct: 保留被打断的思考过程, 追加新的用户输入后, 让模型在之前的基础上继续思考。

<user>我想把我的套餐从目前的 109 美元改成你们新推出的... </user>

<think>用户想要更换套餐, 目前是 109 美元的套餐, 让我查询一下新套餐的信息...

需要了解: 1) 用户当前套餐详情 2) 新套餐价格...<interrupted/></think>

<user> (打断) 对了, 新套餐是每月 79 美元的那个吗? </user>

<think> (继续之前的思考) 用户确认了新套餐价格是 79 美元,

这样从 109 美元降到 79 美元, 每月能节省 30 美元。

我还需要确认: 1) 套餐内容差异 2) 是否有合约限制...</think>

<assistant>是的, 79 美元的套餐。让我确认一下, 您当前的 109 美元套餐包含...</assistant>

优势: 思考过程连贯, 能根据最新信息快速调整策略。

Interactive ReAct: 边想边说 (Speak while Thinking)

用"垫话"争取深度思考时间，降低首字延迟

场景: 用户提出一个复杂问题, Agent 需要时间思考。

传统 ReAct:

```
<user>你确认订购这个套餐吗? </user>
* (长达10秒的沉默...) *
<assistant>经过考虑, 我确认订购。</assistant>
```

Interactive ReAct:

```
<user>你确认订购这个套餐吗? </user>
<think> (快速思考, <0.5s) </think>
<assistant>让我确认一下, 这个套餐是每月 79 美元吗? </assistant> (初步回应)
<think> (深入思考) </think>
<assistant>是的, 这个套餐很划算, 我确认订购! </assistant> (最终回答)
```

优势: 极大改善了交互的流畅度, 避免了尴尬的长时间等待。

Interactive ReAct: 详细序列示例

深入思考的完整对话序列

<user>你确认订购这个套餐吗? </user>

<think>这是一个严肃问题，需要先垫一句话，然后再深入思考</think>

<assistant>让我确认一下，这个套餐是每月 79 美元吗? </assistant>

<think>让我深入思考一下，用户原来的套餐是 109 美元，现在降到了 79 美元，
把用户的 Wi-Fi 速度从 800 Mbps 降到了 400 Mbps，但是用户说可以降速度<interrupted></think>

<user>是的，每月 79 美元</user>

<think>客服确认了这些信息，但我还需要更多思考，因此我还需要再垫一句话</think>

<assistant>让我想一想</assistant>

<think>继续思考，客服提供的套餐包含.....综合看起来挺不错的，
但为了保险起见，还需要再确认一下套餐信息</think>

<assistant>这个套餐挺好的，但我还想再确认下，除了 Wi-Fi 速度，
是不是其他所有服务都保持不变? </assistant>

<user>是的，其他所有服务都保持不变</user>

<think>看起来不错，我应该继续完成预订</think>

<assistant>太好了，我现在就订! </assistant>

Interactive ReAct: 抢话处理示例

实时对话中的中断与恢复

```
<user>这里是 Alaska Airlines, 有什么我可以帮你的吗? </user>
<think>终于过了 IVR, 接到真人了, 我需要简要描述一下用户的需求</think>
<assistant>我要订一张从 San Francisco 到 Seattle 的机票, 价格<interrupted></assistant>
<user>您是打算哪天出发? </user>
<think>客服打断了我的话, 我需要描述用户的出发日期, 在 context 中查到是 9 月 1 日</think>
<assistant>9 月 1 日出发, 单程</assistant>
<user>让我来查一下, 请稍等</user>
<think>我现在需要等待完成操作</think>
<assistant><wait></assistant>
<user>[SYSTEM HINT] 对方有 10 秒钟没有人声了</user>
<think>客服正在查询航班, 可以问一下进展</think>
<assistant>你查到了吗? </assistant>
<user>目前查到了 10 趟航班, 你是要什么时间出发的? </user>
<think>这是一个简单问题, 用户需要尽量便宜, 在价格差不多的情况下要上午出发的</think>
<assistant>要价格尽量<interrupted></assistant>
<user>要经济舱还是公务舱? </user>
<think>客服打断了我的话, 我需要描述我的价格需求</think>
<assistant>经济舱, 要价格尽量便宜的, 在价格差不多的情况下要上午出发的</assistant>
```

为 Interactive ReAct 训练 Adaptive Thinking 模型

训练方法

标签设计:

- 多轮交替的 `<think>` 和 `<assistant>` 标签
- 支持中断和续思考

数据来源:

- 使用 Claude 4 Sonnet 生成训练数据
- 避免使用 OpenAI/Gemini 模型（思考过程经过总结）

数据质量控制:

- 多样化的思考场景
- 平衡简单和复杂问题
- 包含中断和恢复案例

优化目标

核心能力:

- 快速判断是否需要深度思考
- 自然的垫话和时间管理
- 避免简单问题的幻觉

性能指标:

- 首字延迟 < 500ms
- 思考质量不降低
- 用户体验显著提升

Interactive ReAct: 数据工程要点

思考长度控制

1. 首轮思考: 严格控制在 50 tokens 内 (~0.5秒)
2. 简单问题优化: 最小化思考时间, 单轮即可完成
3. 思考长度作为 **RL** 奖励函数的一部分

场景分类处理

- 简单问候: 直接回答, 无需深度思考
- 复杂决策: 生成合适垫话 + 深度思考
- 多轮对话: 保持上下文连贯性

中断处理能力

1. 思考中断恢复:
 - 利用已有思考内容
 - 快速生成合适回复
 - 继续未完成的推理
2. 说话中断处理:
 - 记录中断位置
 - 理解对方插话内容
 - 自然衔接后续对话

第三层：执行层 (Execution Layer)

将离散的动作指令，转换成连续的现实世界信号

输入 (Input)

离散动作指令： speak(...), click(...)

输出 (Output)

连续信号 (语音波形, 鼠标轨迹)

GUI 操作的"最后一公里"难题

Agent 中的莫拉维克悖论

模型“知道”做什么

- 强大的世界知识: 多模态模型能准确理解屏幕内容, 并清晰地描述出操作目标。
- 例如: "点击那个蓝色的'提交'按钮。"

模型“做不到”

- 笨拙的动作: 难以准确输出点击坐标或执行精确动作。
- **Action Space 冲突:** 预训练数据主要是文本, 缺少将高级指令映射到精确物理坐标的数据。

为什么 LLM 点不准坐标?

坐标输出的困难

- 对 LLM 不友好: 要求 LLM 直接输出 (x, y) 坐标, 就像让人类凭空说出屏幕上一个点的精确坐标一样, 非常困难且不直观。

"画框辅助"方案的局限

- 方案: 为页面上所有可点击元素编号, 让 LLM 输出编号。
- 问题: 对于复杂界面, 框会有数百个之多 (例如 Gmail 邮箱界面, 每封邮件、每个分类都是一个可点击区域), 屏幕会变得混乱不堪, 无法使用。一些软件甚至根本无法画框 (例如文档编辑器、表格编辑器、绘图软件、视频剪辑软件)。

解决方案：训练端到端的 VLA 模型

从"知道"到"做到"，打通感知与行动

核心思想：借鉴 Robotics 领域的 VLA 模型，通过 RL 对模型进行后训练，让模型直接输出动作。

方案一：主模型直接输出鼠标点击坐标

- 任务：给定屏幕截图和完整的 Agent 上下文，LLM 直接输出点击坐标，如 `click(x, y)`。
- 训练：利用 LLM 代码能力生成海量测试页面进行 RL。
- 优点：单一模型端到端完成点击动作，操作延迟较低。
- 缺点：没有鼠标移动轨迹，与爬虫脚本一样，容易被当成机器人；模型对屏幕分辨率敏感；需要 RL 训练规模较大的主模型，成本较高。

方案二：训练一个单独的 VLA 模型，模仿人类移动鼠标的模式

- 模仿人类操作：采用“移动-微调-点击”的闭环反馈模式
- 架构：主模型输出的动作中，用自然语言描述想要点击的位置或要进行的操作，VLA 模型生成具体的鼠标移动或点击动作，并根据鼠标指针与元素的相对位置，快速调整下一步的动作。
- 要求：需要一个低延迟 (~100ms) 的 VLA 模型，能够根据“点击第三行第二列的商品”“点击搜索按钮”等自然语言指令，快速调整鼠标指针位置，直至完成操作。

语音合成如何通过图灵测试

核心洞察：人类的“不完美”源于思考速度限制

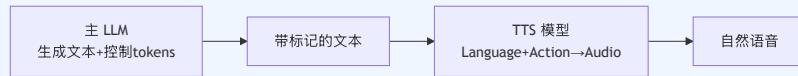
🧠 人类语音的自然规律

- 思考速度 < 说话速度
- 当不知道下句说什么时 → 自然产生：
 - 停顿（思考时间）
 - 填充词（“嗯”、“呃”、“那个”）
 - 重复（“这个这个...”）

🤖 AI 的“太完美”问题

- 思考速度 >> 说话速度
- 生成的永远是干净的整句 → 导致：
 - 过于流畅（无停顿）
 - 零填充词
 - 立即暴露机器身份

解决方案：主 LLM 生成认知停顿，再由 TTS 生成语音



- 主 LLM：决定在哪里停顿
 - 句子转折：插入 [THINKING]
 - 搜索词汇：插入 [SEARCHING]
 - 不确定时：插入 [UNCERTAIN]
- TTS 模型：多模态生成器
 - 输入：文本 + 控制动作
 - 输出：对应的音频

主 LLM 生成控制标记 → TTS 渲染

架构分工：LLM 决策，TTS 执行

主 LLM 生成的控制标记

```
# LLM 根据上下文插入认知停顿  
[THINKING] "嗯..." "让我想想"  
[SEARCHING] "那个..." "怎么说呢"  
  
# LLM 决定情绪和语速  
[EMO:happy] "太好了！"  
[SPEED:0.8x] "让我慢慢解释"  
  
# LLM 选择说话风格  
[STYLE:casual] "哈喽，咋样啊"  
[STYLE:formal] "您好，请问需要什么"  
  
# LLM 输出特殊声音  
[BREATH]  
[SIGH]  
[CLEAR_THROAT]  
[LAUGH:small]
```

实际生成流程

1. 主 LLM 输出（模拟认知负载）

```
[STYLE:casual] [THINKING]  
"我觉得吧" [SEARCHING] "这个"  
"方案挺不错的" [CLEAR_THROAT] "你觉得呢？"
```

2. TTS 模型处理

- 输入：Language（文本）+ Action（控制标记）
- 处理：多模态生成（Language + Action → Audio）
- 输出：自然语音

SEAL 架构小结：与环境实时交互的 Agent

一个统一的事件驱动循环，将感知、思考、执行解耦，实现真正的实时与并行。

感知层 (Perception)

输入: 连续信号 (语音, GUI)

输出: 离散事件流

解决: 传统语音感知的延迟、打断不自然与声学信息丢失问题。

思考层 (Thinking)

输入: 离散事件流

输出: 交错的思考/动作指令

解决: 传统 ReAct 的串行瓶颈，实现可中断、异步的边听边想、边想边说。

执行层 (Execution)

输入: 离散动作指令

输出: 连续信号 + 反馈事件

解决: Agent 动作笨拙、缺乏反馈的“最后一公里”难题，形成行动闭环。

未来展望：端到端模型

当前 SEAL 架构：

- 感知层 LLM: 音频 → 文本 + 声学事件
- 思考层 LLM: 文本 + 声学事件 → 思考 + 动作
- 执行层 LLM: 动作 → 音频

未来端到端架构（无损信息传递，真正的实时思考）：

- **Audio Encoder:** 音频 → Audio Tokens
- 统一 LLM: 感知 + 思考 + 执行（难点：多模态任务不降低智商）
- **Audio Decoder:** Audio Tokens → 音频

AI Agent 与世界交互的三个层次

从语音到视觉再到物理世界

实时语音电话

- 输入模态: 语音
- 输出模态: 语音
- 数据密度: 15-50 tokens/秒
- 延迟要求: <500ms
- 主要挑战: 快慢思考平衡

图形界面操作

- 输入模态: 视觉
- 输出模态: 鼠标点击、键盘按下等动作序列
- 数据密度: ~2K tokens/截图
- 延迟要求: <1秒
- 主要挑战: 精确动作执行

物理世界交互

- 输入模态: 视觉+语音+触觉
- 输入模态: 语音+各关节动作序列
- 数据密度: ~20K tokens/秒
- 延迟要求: <100ms
- 主要挑战: 实时感知与控制

技术挑战逐层增长，但解决方案可以逐层复用和迁移

第二部分：Agent 从经验中学习

为什么 Agent 要从经验中学习：从"聪明"到"熟练"

SOTA 模型 ≈ 顶尖毕业生

- 知识渊博：掌握海量通用知识。
- 缺乏经验：在专业任务（例如记账报税）上表现可能不如经验丰富的专业人士。

Pine AI 的业务挑战

- 验证信息：Pine AI 帮用户打电话办事，第一次打客服电话前不知道客服要验证信用卡后四位，第二次打同一个公司的客服能不能先找用户要信用卡信息？
- 办事流程：第一次办理某服务退订的时候，电话里客服说要填表单，那么第二次办理这种服务退订的时候，能不能直接填表单，而不要再去打电话？
- 办事规则：享受某个优惠的条件是什么？（例如退伍军人、超过 2 年的忠实客户）
- 价格预估：某地某运营商 3 Gbps 家庭宽带 \$60 一个月，是高了还是低了，有没有谈判空间？

核心问题：许多业务流程是动态变化的、非公开的。仅靠提升基座模型的通用能力，无法解决这类"经验"问题。

从经验中学习：机器学习的根本问题

机器学习的本质就是从经验中学习

范式一：后训练

核心挑战：

- LLM 不是真正的世界模型
- RL 样本效率低 ($O(1)$ bits/episode)

解决方案：

- SFT：固化格式与协议
- RL：学习可迁移策略
- 反馈引导 + 世界模型提升效率

范式二：上下文学习

核心洞察：

- Context 本质是检索，非总结
- 记忆差是 Feature，非 Bug

最佳实践：

- 显式总结 > 隐式学习
- 选择性遗忘，降低噪声
- 跨模态压缩（如 OCR）

范式三：外部化学习

核心：主动提炼，而非被动堆积

- 知识库：统计聚合、规则归纳、层次化组织
- 工具生成：重复流程封装为代码
- 利用额外算力进行深度总结
- 从信息检索到知识发现
- 延续 Scaling Law 到外部生态

方法一：后训练 (Post-training) - RL 让 Agent 变"熟练"

核心问题：RL 样本效率低下

强化学习之父 Richard Sutton 指出：**LLM** 只是模仿人说什么，而不是理解世界如何运转

LLM 不是真正的世界模型 !

RL 样本效率低下的根源 

预训练学到的是描述性知识：

- 学习"人类如何描述一个过程"
- 学习"人类如何谈论一个概念"
- 缺乏交互性知识："执行动作后会发生什么"

Model-free RL 的困境：

- 只从 reward 学习，不从 observation 学习
- 每个 episode 仅提供 $O(1)$ bits 信息
- 环境的丰富反馈被浪费

举例：

-  模型能描述"如何打电话给客服"
-  不知道"客服说缺少 SSN 时应该怎么办"

实际案例：

第一次打电话：

客服："我需要您的信用卡后四位"
→ 任务失败， $reward = 0$

传统 RL：需要数百次尝试才能学会
人类：第一次就记住了

SFT 记忆, RL 泛化

监督微调 (SFT) 的特点

优势:

- 样本效率极高 (数千条即可)
- 快速固化格式与协议
- 训练稳定、收敛快

局限性:

- 记忆训练分布的表面模式
- 分布外场景性能断崖式下降
- 难以学习可迁移的抽象策略

强化学习 (RL) 的特点

优势:

- 学习可迁移的策略表征
- 在分布外场景保持稳健
- 能发现超越示范的新策略

局限性:

- 样本效率低 (需数百倍数据)
- 训练成本高、时间长
- 需要可验证的奖励信号

💡 工程实践: 先形后神

1. **SFT** 阶段: 建立格式稳定性, 确保输出可解析
2. **RL** 阶段: 在稳定基础上突破泛化边界
3. 关键平衡: SFT 训练到"格式稳定、能力初具"即可, 过度训练容易导致模式崩溃

提升样本效率的前沿探索（一）：反馈引导采样

批次内的样本复用 - 让 RL 从失败中快速学习

传统 GRPO/DAPO 的问题

每个训练批次：

- 为同一 prompt 生成 N 个独立 rollout
- 每个 rollout 独立采样，不考虑其他结果
- 即使前几次暴露了明显问题，后续仍重复错误
- 最终每个 rollout 只贡献一个二元信号 (0/1)

信息浪费：

Rollout 1: 客服说需要 SSN → 失败
Rollout 2: 客服说需要 SSN → 失败
Rollout 3: 客服说需要 SSN → 失败
... (完全没有利用环境反馈)

反馈引导采样的改进

序贯依赖生成：

1. 第 1 次 rollout：从原始 prompt 生成
2. 第 2 次 rollout：原始 prompt + 第 1 次的反馈
3. 第 N 次 rollout：累积所有之前的反馈

效果对比：

Rollout 1: 客服说需要 SSN → 失败

Rollout 2: [知道需要SSN] → 准备好 → 成功

Rollout 3: [知道需要SSN] → 准备好 → 成功

... (批次内快速适应)

样本效率提升：单个训练批次产生更多高质量样本

提升样本效率的前沿探索（二）：双 LoRA 世界模型

从环境观测中提取额外学习信号

核心思想

问题：

- 传统 RL 每个 episode 仅从 reward 学习 ($O(1)$ bits)
- 环境返回的观测包含数十到数百个 token
- 这些丰富的信息完全被浪费

解决方案：

- 为基础模型添加两个 LoRA adapter：
 - **Policy LoRA**: 从 reward 学习策略
 - **World Model LoRA**: 预测环境观测

训练目标：

世界模型如何提升样本效率

1. 提供额外学习信号：

- 预测客服会说什么
- 从每个 episode 提取 $O(\text{观测长度})$ bits

2. 共享表征带来隐式知识迁移：

- World Model 学习环境规律
- Policy 利用增强的表征更快学习

3. 选择性预测：

- 只预测信息量高的观测（客服回复）
- 忽略确定性的工具返回

```
L_total = L_policy + λ * L_world_model
```

方法二：上下文学习 (In-context Learning)

Long Context 的误解：Context ≠ 记忆

很多人认为，有了 long context，可以把所有历史信息放进去让模型自动推理。这是对 **context** 能力的严重误解。

Context 的本质：检索而非总结

Attention 机制的真实能力：

- 基于相似度匹配的键值查询
- 把相关信息放到里面
- 类似于 RAG，而非推理引擎
- 擅长"找到相关信息"，不擅长"统计聚合"

实际案例（三次打电话限制）：

- System Prompt 规定：打同一个商家的电话不能超过 3 次
- Agent 执行轨迹包含多次打 Xfinity 电话的记录
- 模型每次需要决策时，都需要扫描全部轨迹，容易数错。经常陷入循环，又打第 4 次
- 即使较大的模型能够数对，每次都要消耗额外的 reasoning tokens 来遍历上下文中的通话记录

System Hint: 将隐式状态显式化

强制引导注意力，从"扫描推理"到"直接使用"

解决方案: System Hint

在上下文末尾添加聚合信息:

```
<system_hint>
Tool call summary:
- 'phone_call' 已调用 3 次
  - Xfinity: 3 次 (已达上限)
```

```
Constraint check:
- 不能再打 Xfinity 电话
</system_hint>
```

效果:

- ✓ 复杂度从 O(轨迹长度) 降为 O(1)
- ✓ 模型直接使用聚合信息，无需推理

三类 System Hint

- 任务规划: TODO list

```
<system_hint>
TODO List:
- [✓] Call Xfinity customer service
- [ ] Call Xfinity retention department
</system_hint>
```

- 侧信道信息: 时间戳、地理位置

```
[User Message] [2025-06-25 11:00:20] Hello?
```

- 环境状态: 工具调用次数、当前目录

```
<system_hint>
Current directory: /home/ubuntu
Operating System: Ubuntu 24.04
</system_hint>
```

Karpathy 的洞察：记忆差是 Feature 不是 Bug

为什么 Agent 不应该记住所有细节？

传统观点：记忆越多越好

Karpathy 的反直觉洞察

误区：

- 希望模型记住所有历史交互的细节
- 追求无限长的上下文窗口
- 认为“遗忘”是缺陷

核心观点：选择性遗忘是智能的标志

1. 人类的记忆机制：

- 自动过滤无关细节
- 强化重要经验
- 构建抽象概念

2. Agent 应该模仿人类：

- 主动丢弃低价值信息
- 压缩重复模式为规则
- 外部化精确细节（知识库）

问题：

- 模型容易记住细节，而不是提取知识

理论洞察：无需训练的学习

基于论文 "Learning without training: The implicit dynamics of in-context learning" ([arxiv:2507.16003](https://arxiv.org/abs/2507.16003))

核心发现: Transformer 块能够通过上下文实现对 MLP 权重的隐式更新

隐式权重更新机制

关键洞察:

- Self-Attention 层与 MLP 层的堆叠产生特殊效应
- 上下文信息被转换为对 MLP 权重的低秩更新
- 每个 attention head 贡献一个 rank-1 更新

```
# 简化的数学表达  
MLP_effective = MLP_original + Σ(rank_1_updates)  
# 其中 rank_1_updates 来自 attention 输出
```

意义: 解释了 LLM 如何在不训练的情况下通过 prompt 学习新模式

In-Context Learning 的本质

特性	传统训练	上下文学习
权重更新	显式 (gradient)	隐式 (attention)
持久性	永久	仅当前序列
计算方式	反向传播	前向传播
适应速度	慢 (需要多轮)	快 (单次推理)
■ LLM 在推理时实际上在进行 "软训练"		
■ 解释了为何 few-shot learning 如此有效		
■ 为设计更好的 prompt 提供理论基础		

ICL vs Fine-tuning: 隐式模式学习的反直觉发现

基于论文 "Deeper Insights Without Updates: The Power of In-Context Learning Over Fine-Tuning"
(arxiv:2410.04691)

核心发现: 对于包含隐式模式的任务, ICL 能够比 Fine-tuning 更好地捕捉和利用这些模式

🔍 实验设计

隐式模式任务类型:

- 数学计算: 表达式求值、布尔函数
- 文本推理: 关系推理 (城市连接判断)
- 代码理解: 输出预测

隐式模式示例:

```
# 表达式中包含归零项  

$$(6 - 1) + (6 - 6) * (-10 + 1 + 2 + 13) = 5$$
  
#           ↑ 这部分为0, 可作为捷径
```

📊 关键实验结果

方法	准确率提升	样本效率
ICL	显著提升	仅需几个示例
Fine-tuning	有限改进	需要数千倍样本

性能对比:

- ICL 能快速识别深层模式
- Fine-tuning 即使用更多数据也难以学习隐式模式
- ICL 在 OOD (分布外) 数据上表现更稳健

Circuit Shift 理论：为什么 ICL 更擅长模式识别？

从可解释性角度解释 ICL 与 Fine-tuning 的差异

电路转移（Circuit Shift）机制

什么是 Circuit？

- 模型中负责特定行为的子图
- 由特定的注意力头和 MLP 层组成
- 代表模型解决问题的“思维路径”

关键发现：ICL 导致更大规模的电路转移，意味着 ICL 更彻底地改变了模型的问题解决方法

激活补丁（Activation Patching）分析：

- 通过操纵特定组件的激活值
- 量化不同组件对任务的贡献
- 发现 ICL 激活了不同的电路模式

模式捕捉能力：

- **ICL**：“能够快速捕捉深层模式，并显著提升准确率”
- **微调（Fine-tuning）**：“即便使用成千上万倍的训练样本，提升也十分有限”

机制解释：

- ICL 不是简单的模式匹配，而是激活了模型中不同的计算电路
- Fine-tuning 虽然更新参数，但在隐式模式任务上的电路转移较小
- 这解释了为何 ICL 在无需参数更新的情况下能取得更好效果

方法三：外部化学习 (Externalized Learning)

RAG 不能直接把原始案例/对话扔进知识库

案例一：黑猫白猫的计数问题

场景：知识库有 100 个案例，包括 90 只黑猫的独立描述、10 只白猫的独立描述

问题：“黑猫和白猫的数量比例？”

RAG 检索的困境：

- 每个案例语义相似度不同
- Top-k=20 只能检索到部分案例
- 基于不完整样本推理 → 错误结论

正确做法：

预先提炼：“共 100 只猫：90 只黑猫(90%)，10 只白猫(10%)”
→ 一次检索即可获得准确统计

案例二：Xfinity 优惠规则推理错误

场景：知识库有 3 个孤立案例：退伍军人 John 获得优惠，医生 Sarah 获得优惠，教师 Mike 不能享受优惠

问题：“我是护士，能享受优惠吗？”

RAG 检索的困境：

- “护士”与“医生”语义相近 → 检索到案例 B
- 未召回案例 A、C → 推断错误

正确做法：

预先提炼规则：“Xfinity 优惠仅适用于退伍军人和医生，其他职业不符合条件”
→ 一次检索即可获得完整规则

外部化学习：主动知识提炼 vs 被动信息堆积

超越参数化知识的根本局限

3.1 知识库 (Knowledge Base)

核心思想: 利用额外算力主动提炼知识

- 不是简单地存原始案例/对话
- 投入算力进行知识总结、压缩、结构化
- 从“100 个案例”提炼为“统计摘要”
- 从“3 个案例”提炼为“明确规则”
- 实现高效、可靠、无幻觉的知识检索

3.2 工具生成 (Tool Generation)

核心思想: 将重复流程封装成代码

- 识别重复性操作序列
- 封装为可精确执行的工具（代码）
- Agent 通过创造工其实现自我进化
- 从最小预定义到最大自我进化
- 形成可复用、可组合的能力单元

3.1 外部知识库：利用额外算力主动提炼知识

关键原则：压缩即理解，提炼即学习

为什么需要主动提炼？

知识提炼的价值（打电话次数统计）：

原始轨迹：

10:00 打 Xfinity 电话（讨论账单）
10:30 打 Xfinity 电话（转接部门）
11:00 打 Xfinity 电话（谈判降价）

无提炼：模型需扫描轨迹统计次数

→ 成本： $O(\text{轨迹长度})$ 的推理 token
→ 容易数错，陷入循环

提炼后：System Hint "已打 Xfinity 电话 3 次，已达上限"

→ 成本： $O(1)$ 的推理 token
→ 立即识别限制，避免重复

核心洞察：投入额外算力（LLM 调用）进行总结

知识提炼的三个层次

1. 统计聚合：

- 从 100 个独立案例 → "90% 黑猫，10% 白猫"
- 降低信息密度，提高检索效率

2. 规则提炼：

- 从 3 个孤立案例 → "仅退伍军人和医生可享优惠"
- 从案例到抽象规则的飞跃

3. 层次化组织（RAPTOR/GraphRAG）：

- **RAPTOR**: 树状层次摘要
- **GraphRAG**: 实体关系网络

知识提炼实战：从原始经验到结构化知识

知识表达的多种形态

概要性总结：

"办理 A 公司 B 业务退订流程：

1. 验证身份（订单号 + 注册邮箱 + 信用卡后四位）
2. 填写退订表单（需提供退订原因）
3. 等待 3-5 个工作日处理"

结构化提炼 (Advanced JSON Cards) :

```
{  
  "category": "business_process",  
  "company": "A公司",  
  "service": "B业务退订",  
  "auth_required": ["order_number", "email", "last_4_cc"]  
  "success_rate": 0.85,  
  "avg_duration": "3-5 days",  
  "backstory": "基于 23 次成功案例总结"  
}
```

知识检索策略

Agentic 语义搜索：

- Agent 自主构建查询关键词
- 向量数据库 + BM25 混合检索
- 神经重排序提升精度

上下文感知检索：

- 为每个知识片段添加上下文前缀
- "本条来自 A 公司退订流程总结"
- 同时增强稀疏和稠密检索效果

Agentic 文件系统操作：

- `read_file` : 读取总结文档
- `grep` : 精确搜索关键词
- `find` : 按文件名模糊匹配

Contextual Retrieval：提升 RAG 的检索精度

基于 Anthropic 的 Contextual Retrieval

核心问题：传统 RAG 在切分文档时会丢失上下文，导致检索精度下降

Contextual Embeddings & Contextual BM25

- 在 embedding 和创建 BM25 索引前，为每个 chunk 添加特定的解释性上下文

示例转换

```
# 原始 chunk (缺乏上下文)
original = """The company's revenue grew by 3%
over the previous quarter."""

# 上下文化后的 chunk
contextual = """This chunk is from an SEC filing
on ACME corp's Q2 2023; previous
quarter revenue was $314M.
The company's revenue grew by 3%..."""

# 将 original 替换为 contextual
original = contextual
```

实现方式

- 使用 Claude 自动生成每个 chunk 的上下文 (50-100 tokens)

效果

检索失败率降低 (top-20 chunks) :

- 仅 **Contextual Embeddings**: ↓35%
- Contextual Embeddings + BM25**: ↓49%
- + **Reranking**: ↓67%

关键优势:

- 保留文档上下文：每个 chunk 都知道自己来自哪里
- 提高精确匹配：BM25 处理专有名词、错误代码等
- 语义理解增强：embedding 捕捉更准确的含义

最佳实践组合：Contextual Embeddings + Contextual BM25 + Reranking + Top-20 chunks

基于 LLM 的行业知识总结：把算力变成知识库

传统知识库构建的问题

- 知识碎片化: 大量知识和行业经验未经总结
- 查询低效: 信息散落在网络各处
- 成本高昂: 行业知识总结成本极高
- 知识库难建: 公司内部知识库建立困难

基于 LLM 构建大规模知识库

- 通用知识 → 基础模型
 - 模型训练本身就是知识总结过程
 - Transformer 不擅长记忆大量事实细节，大量事实细节会导致参数量爆炸
- 事实性信息、行业经验 → LLM 自动化总结
 - 个人信息、行业数据等具体事实
 - 验证信息、办事流程、办事规则、价格预估等行业经验
 - 对海量原始数据进行整理，形成结构化知识库



核心洞察: LLM 可以将算力转化为可扩展的知识库

学习方法对比：外挂知识库 vs. 内置注意力

外挂知识库 (RAG)

- 类比: 让不支持思考的 LLM 通过 "Think step by step" 的 Prompt 来引导思考。
- 优点: 无需重新训练, 即插即用。可以用额外的算力做多方面总结, 并且总结过程中可以结合行业 know-how, 较为灵活。
- 缺点: 效果依赖于检索的 precision 和 recall。很多时候, 多个片段之间语义相关的内容难以提取出来。此外, RAG 提取的片段缺失上下文, 影响理解。

内置注意力 (Long Context)

- 类比: 原生支持长上下文和内置思考能力的模型。
- 优点: 端到端优化, 效果上限更高。
- 缺点: 目前成本较高。总结过程一般由模型本身决定, 不太容易融入行业 know-how, 也不方便利用额外算力。

Fine-tuning vs RAG：知识注入方法的实证对比

基于论文 "Fine-Tuning or Retrieval? Comparing Knowledge Injection in LLMs" (EMNLP 2024)

💡 论文核心洞察：RAG 不仅更有效，还避免了微调可能带来的知识遗忘和幻觉问题

论文的对比实验

- 比较无监督微调 (Unsupervised Fine-tuning) 与 RAG
- 评估已有知识 (Existing Knowledge) 和全新知识 (New Knowledge) 的处理

关键发现

1. RAG 整体表现更优

- 在处理已有知识和全新知识时均优于微调
- 更可靠的事实信息检索

2. 微调的局限性

- LLMs 通过无监督微调学习新事实性信息存在困难
- 需要同一事实的多种变体才能有效学习

3. 改进策略

- 在训练中暴露同一事实的多种表述形式
- 结合使用而非单一依赖某种方法

RAG 的优势场景

- 需要准确的事实性知识注入
- 处理新领域或新知识
- 避免模型参数污染

微调的挑战

- 单纯的无监督微调难以有效注入新事实
- 需要数据增强：同一知识的多种表述
- 可能影响模型原有能力

推荐策略

- 优先考虑 RAG 用于知识注入
- 如需微调，确保数据多样性
- 考虑混合方案以取长补短

3.2 工具生成 (Tool Generation) - 让 Agent 自我进化

核心思想：从最小预定义到最大自我进化 (Alita: ([arxiv:2505.20286](#)))

最小预定义原则

- 极简架构: 仅配备单一核心组件 (Web代理)
- 避免过度设计: 不预设复杂工具和工作流程
- 通用性优先: 减少领域特定的硬编码

效果: GAIA 基准测试

- Pass@1: **75.15%**
- Pass@3: **87.27%**
- 超越许多复杂设计的 Agent 系统

最大自我进化机制

核心能力:

1. 自主创建工具: 根据任务需求生成新工具
2. 能力完善: 迭代改进已有工具的性能
3. 经验重用: 将成功的模式固化为可复用组件

Model Context Protocols (MCPs):

```
# Agent 自动生成任务相关的协议
mcp = agent.generate_protocol(task)
# 执行并优化
result = agent.execute_with_mcp(mcp)
# 存储成功的模式
agent.store_successful_pattern(mcp)
```

工具规模化挑战：MCP-Zero 主动工具发现

基于论文 MCP-Zero: Active Tool Discovery for Autonomous LLM Agents

工具数量爆炸的问题

规模挑战：MCP 生态 308 个服务器，2,797 个工具

传统方法困境：

- 全量注入：GitHub MCP 26 个工具需 4,600 tokens
 - 完整工具集需要 **248k tokens** → Context 爆炸
- 静态检索：基于初始查询选择，无法预判任务演化
 - "调试文件"需要文件系统+代码分析+命令执行

根本问题：Agent 沦为被动选择器，而非主动发现者

MCP-Zero：从被动到主动

核心理念：让 Agent 主动识别能力缺口，按需请求工具

三大机制：

1. 主动工具请求：Agent 生成结构化需求

```
server: github # 平台域  
tool: search_repos # 操作类型
```

2. 分层语义路由：先筛选服务器，再匹配工具

3. 迭代能力扩展：执行中动态发现并构建工具链

实测效果：

- APIBank 测试：**98% token** 节省
- 从 3000 个工具中准确选择，保持高准确率
- 工具生态增长时性能稳定

Agent 自动生成工具案例 1: Computer Use 智能 RPA

Computer Use 的困境

- 速度慢: 每一步都需要和 LLM 交互, 延迟高。
- 成本高: 每次操作都是一次昂贵的 LLM 调用。

传统 RPA 的启发

- 速度快: 运行预先编写好的脚本, 速度远超人类。
- 问题: 无法处理动态界面, 缺乏理解和判断能力。

新思路: 让 LLM 自动将操作过程总结成一个"智能 RPA"工具。

智能 RPA：技术实现细节

根据操作序列生成 RPA 代码的核心挑战

三大技术难题 解决方案

1. 坐标漂移问题

- 页面元素位置动态变化
- 固定坐标点击经常失败

1. 智能元素定位

- 使用元素 ID/XPath 替代坐标
- Playwright 自动适配界面变化

2. 动态内容判断

- 某些步骤需要理解内容
- 例：选择最便宜的机票

2. LLM 分析优化

- 识别固定流程 vs 动态判断
- 提取可自动化的操作序列

3. 响应时间不确定

- 网页加载时间不固定
- 异常情况无法处理

3. 事件驱动执行

- 监听浏览器事件完成状态
- 智能等待元素可点击
- 超时自动切换 LLM 接管

智能 RPA 的效果

用例1：查天气 (9 步)

- 传统方式: 47 秒, 9 LLM 调用
- 加速后: 10 秒, 2 LLM 调用 + 1 RPA

用例2：官网订机票 (240 步)

- 传统方式: 19 分钟, 240 LLM 调用
- 加速后: 4 分钟, 25 LLM 调用 + 4 RPA

核心优势: 零人工开发成本, 全自动生成, 且能在网站改版时自动生成新工具以适应变化。

Agent 自动生成工具案例 2: Agent 日志解析和可视化

挑战: Agent 执行流程日志

- 需求: 可视化 Agent 的执行流程 (trajectory) , 以便观察、调试
- 格式多样性: Agent 的工具、sub-agent 种类多样, 每种工具和 sub-agent 都有不同的参数格式和工具返回的结果格式, 且在不断改变
- 日志量大: 整个 agent 执行流程常常有几百步, 需要压缩显示更直观。但又要能展开每次 LLM 调用和每次工具调用的完整细节, 以便调试。
- 维护成本高: 手写解析代码需要不断更新, 且难以预测所有可能的日志格式。

Agent 自我进化的解决方案

核心机制: 前端遇到解析失败时自动向 Agent 报告

- 智能识别: Agent 分析失败日志的模式
- 自动生成解析代码: 基于样本生成新的前端解析代码
- 自动测试解析代码: 利用虚拟浏览器执行环境, 检查新的解析代码是否能正确解析新的日志类型 (是否有报错), 并用 Vision LLM 检查可视化效果是否符合预期
- 热更新部署: 解析代码自动更新

Agent 自动生成工具案例 3：生产系统问题自动诊断

生产系统调试的挑战

- 问题定位困难：从海量 Agent 执行流程日志中找出根因
- 重现成本高：生产问题难以在测试环境重现
- 已修复的问题容易再次出现：构建 regression test case（回归测试用例）的成本高

Agent 自动化诊断流程

工作机制：从生产环境的 Agent 执行流程日志中自动 Triage 问题，并生成问题报告和测试用例

1. Agent 执行流程日志分析：

- 结合系统架构文档和 PRD（产品需求文档），自动分析 Agent 的执行流程是否符合预期

2. 测试用例生成：

- 自动生成 Regression Test Case
- 确保问题修复后不再复现

3. 工作项创建：

- 自动创建 Scrum Work Item
- 包含问题描述、影响范围、建议修复方案

小结：从后训练到上下文学习再到外部化学习

范式一：后训练 (Post-training)

核心发现: SFT 记忆、RL 泛化

- **SFT**: 固化格式与协议，样本效率高
- **RL**: 学习可迁移策略，分布外鲁棒
- 挑战: RL 样本效率低 ($O(1)$ bits/episode)

范式二：上下文学习 (In-context Learning)

核心洞察: Context \neq 记忆

- 本质: 检索而非总结，类似 RAG
- 关键: 显式总结 > 隐式学习

范式三：外部化学习 (Externalized Learning)

3.1 知识库 (Knowledge Base)

优点: 解决幻觉问题，知识更新快，可用额外算力深入总结
方法: Contextual Retrieval, RAPTOR 树形总结

3.2 工具生成 (Tool Generation)

优点: 流程代码化，高效、可靠、可组合
理念: 最小预定义 + 最大自我进化 (Alita)

Scaling Law: 从预训练到 RL

"We want AI agents that can discover like we can, not which contain what we have discovered. Building in our discoveries only makes it harder to see how the discovering process can be done." — Rich Sutton, [The Bitter Lesson](#)

第一阶段: 基于预测下一 token 的预训练 (Pre-training)

- 核心定律: 模型的性能随着计算资源、模型参数和训练数据量的增加而可预测地提升。
- 实现方式: 在海量的互联网文本上进行无监督的预训练, 学习通用的世界知识。
- 本质: 将知识内化到模型的参数中, 形成一个庞大的、静态的知识库。
- 成就: 实现了通用的语言理解和推理能力。

第二阶段: 强化学习: 通过与世界交互进行后训练 (RL-based post-training)

- Agent 让模型不再局限于被动学习, 而是能够主动探索世界。
- 交互的对象:
 - 人类: 通过实时语音进行对话与协作。
 - 互联网: 操作电脑, 浏览网页, 使用软件。
 - 物理世界: 控制机器人, 与真实环境互动。
- 学习方式: 通过强化学习, 从交互的成功与失败中进行发现和学习, 这是 Scaling Law 的第二曲线。
- 目标: 从一个知识的容器变为一个发现的引擎。

Scaling Law 遇到的瓶颈: Transformer 的精确记忆局限

- Transformer 架构不擅长精确、可靠地记忆和更新这些高度动态、具体的细节信息。
- 强行记忆容易导致幻觉和信息混淆, 且成本不经济 (训练推理时需遍历所有知识中较大比例的部分), 成为 Scaling Law 的巨大障碍。

未来之路：外部化学习延续 Scaling Law

*"The two methods that seem to scale arbitrarily... are **search** and **learning**." — Rich Sutton, [The Bitter Lesson](#)*

突破瓶颈：外部化学习 LLM 驱动的主动知识提炼

- 统计聚合能力：
 - 从 100 个独立案例提炼出"90% vs 10%"的统计摘要
- 规则归纳能力：
 - 从 3 个孤立案例抽象出"仅 X 和 Y 符合条件"的明确规则
- 结构化总结能力：
 - SOTA LLM 将非结构化经验总结为结构化知识
 - LLM 时代前需要昂贵领域专家，现在 $LLM \approx$ 无限多领域专家
 - Summarization 是 LLM 时代后仍活跃的传统 NLP 课题

代码作为通用知识表示

- LLM 将经验或流程精确描述为代码
- 代码 = 通用的结构化数据表示：精确、可验证、可组合
- 从模糊的"经验"到可执行的"工具"

延续 Scaling Law：两种通用方法的胜利

Search (搜索)：

- 对应外部知识库和工具库的检索
- 从原始信息到提炼知识的精准查找

Learning (学习)：

- 对应 LLM 的总结和代码生成能力
- 将交互经验主动提炼为知识和工具

Pine AI

我们正在寻找能够构建 SOTA autonomous AI Agent 的全栈工程师。

我们秉持的理念：

每个人对公司估值的贡献都要在千万美元以上

加入 Pine AI 的要求

1. 熟练使用 AI 编程

- 80%+ 代码通过人机协作完成
- 代码面试：AI 辅助下 2 小时完成功能开发
- 内部系统全部基于 AI 构建

2. 热爱动手解决问题

- "Talk is cheap, show me the code"
- 成为架构师和产品经理的结合体
- 直接指挥 AI，减少信息损耗

3. 扎实的软件工程能力

- 完善的文档和测试
- 让 AI 能理解和维护代码
- 高质量的工程实践

4. 理解 LLM 原理

- 了解基础原理和能力边界
- 驾驭 LLM 的正确方法
- 提供合适的上下文和工具

5. 解决世界级难题的信心

- 追求 SOTA 水平
- 与初创公司共同成长
- 不断超越现有水平

我们的使命

通过构建能与世界实时交互、从经验中学习的 Agent，真正为用户解决烦恼，办成事。

Pine AI - Building Agents That Get Things Done

```
mail -s "Join Pine AI" -A /path/to/your_resume.pdf boj@19pine.ai
```

Agent 的两朵乌云：实时与环境交互，从经验中学习

"The biggest lesson that can be read from 70 years of AI research is that general methods that leverage computation are ultimately the most effective, and by a large margin."

— Rich Sutton, [The Bitter Lesson](#)