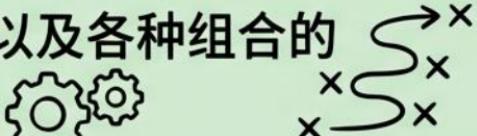
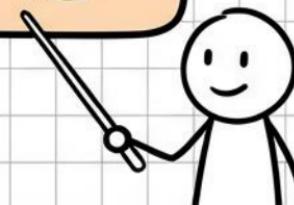


# 这次想聊什么（以及不聊什么）

- 我想回答： agentic coding tool 到底能帮我“写代码”，还是只会帮我“写自信”？ → 

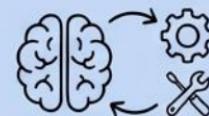
- ✓ 我会讲： Claude Code (CC) 与 Codex，以及各种组合的体验差异，以及我踩过的坑。 

- ✗ 我尽量不讲：我用agent做的科研(贻笑大方)，不同家model的token价格(都是蹭实验室的账号啦)。 



# 什么是 LLM agent

- LLM agent: 用 Large Language Model (LLM) 当“脑子”，能做 Reasoning & Planning，并且能调用 Tool 去执行动作。

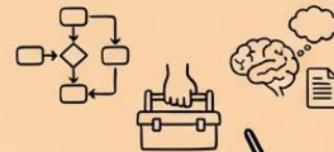


- ✓ 和 chatbot 的差别: chatbot 做一轮轮“问答”；agent 调用工具、维护状态以“推进任务”。

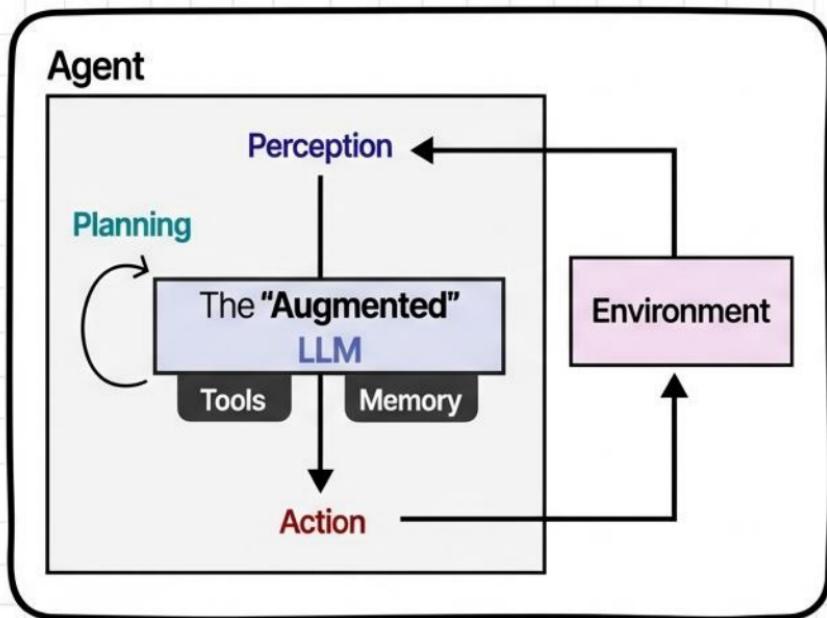


✓ 常见组成：

- Reasoning & Planning: 把目标拆成子任务，决定下一步做什么。
- Tool Use: 读文件、改文件、跑命令、查资料、访问外部系统。
- Memory: short-term memory (当前上下文)、long-term memory (文档)。



# 什么是 LLM agent (cont.)

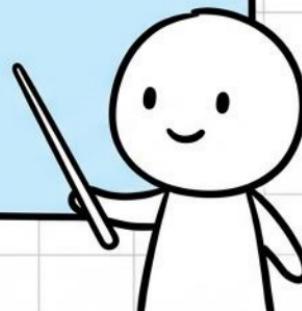


- The LLM serves as the central “brain” for perception, reasoning, and decision-making.
- Tools extend the LLM’s capabilities (e.g., searching, calculation, API calls).
- Memory provides short-term context and long-term knowledge retention (e.g., RAG, embeddings).
- The “Augmented” LLM continuously interacts with the Environment through an Action-Perception feedback loop.
- This architecture transforms a passive chatbot into an autonomous agent capable of executing multistep tasks.

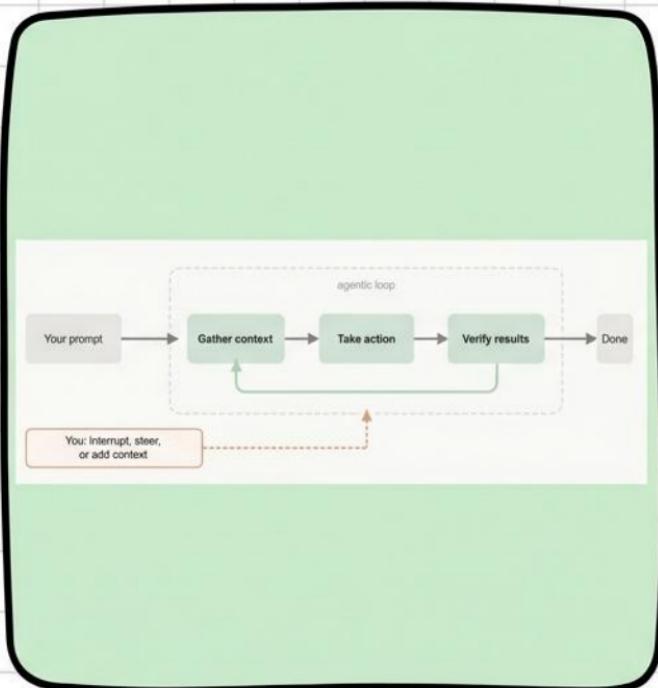


# 那什么是 agentic coding tool

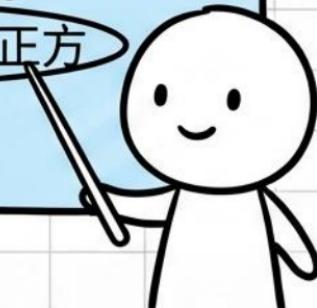
- agentic coding tool: 把 LLM agent 放进 coding workflow, 让它能“看 repo、改 repo、跑 repo”。
- 典型代表：
  - Claude Code (CC) : Anthropic家。
  - Codex: OpenAI家。



# Claude Code (CC) : agentic loop 是怎么跑起来的



- CC 的典型循环: gather context → take action → verify results。
- 这个 loop 会“自适应”: 问答可能只做 context; 修 bug 会反复 action/verify; 重构会疯狂 verify。
- 你也在 loop 里: 你可以随时 interrupt、补充上下文、修正方向。



# CC 的关键：Tools（不然就真只能扯皮了）

- 没有 Tool：模型只能输出 text。
- 有了 Tool：模型能 act
- **File operations**：  
read/edit/create/rename。
- **Search**：按 pattern 找文件、regex 搜内容。

- **Execution**：run shell、run tests、use git。
- **Web**：查 docs、搜错误。
- **Code intelligence**：LSP 支持的跳转/引用/诊断。

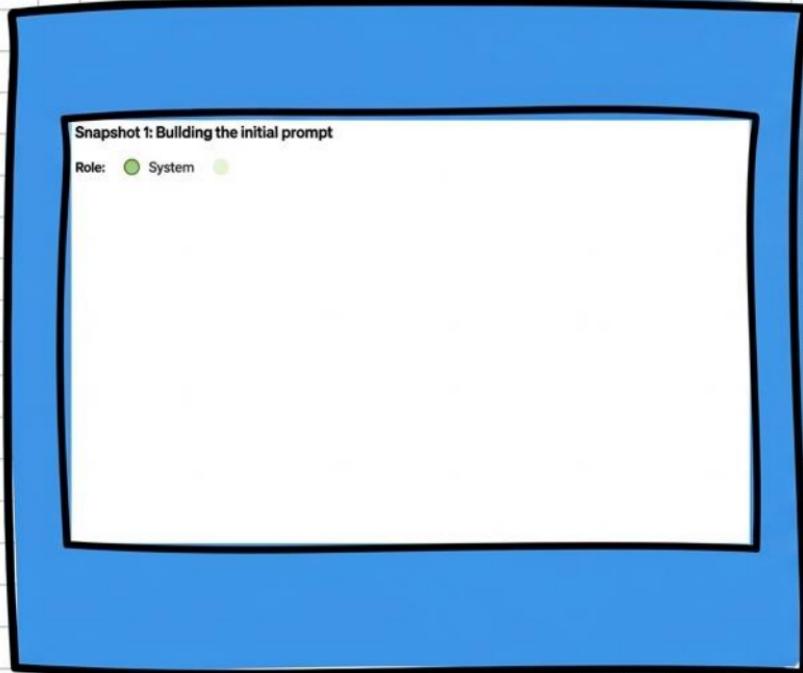


# Codex：绝对被低估的王者

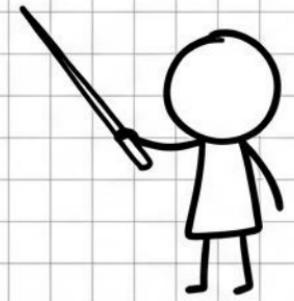
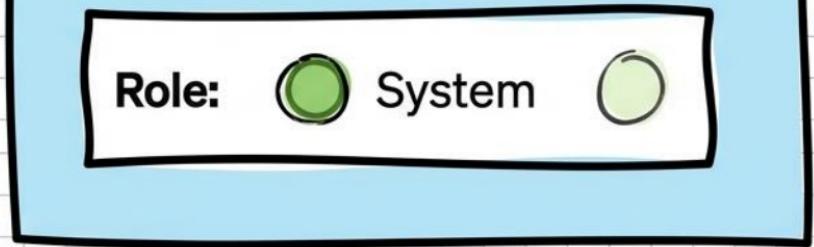
- Codex 的“感觉”更像工程流水线：你给一个任务，它会构造 prompt（结构化），然后在一个 turn 内做多次 tool calls。再做多个turn，每个turn是一次对话。
- 他甚至是开源的：<https://github.com/openai/codex>，甚至是 Rust写的！
- 有时间一定学习下（一定一定，不会咕的）！



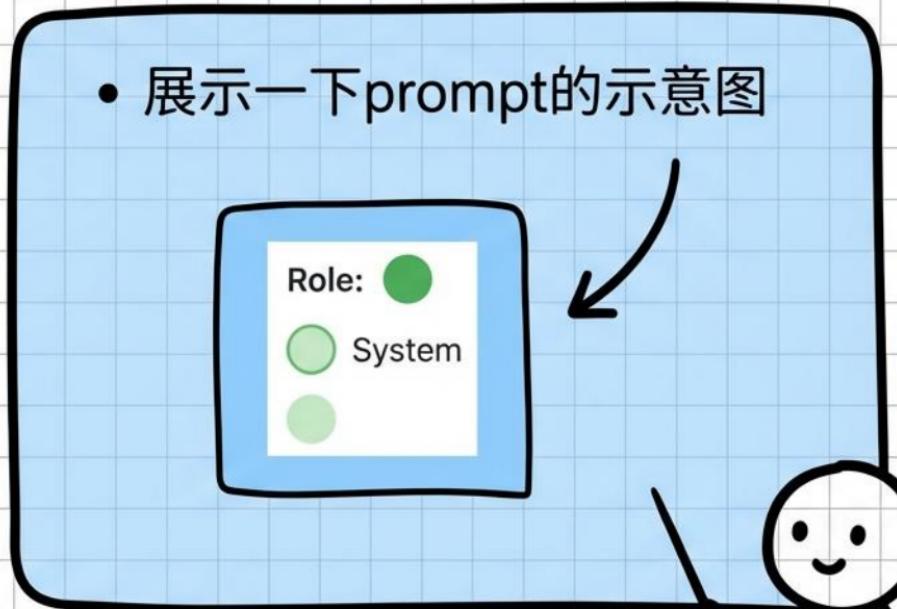
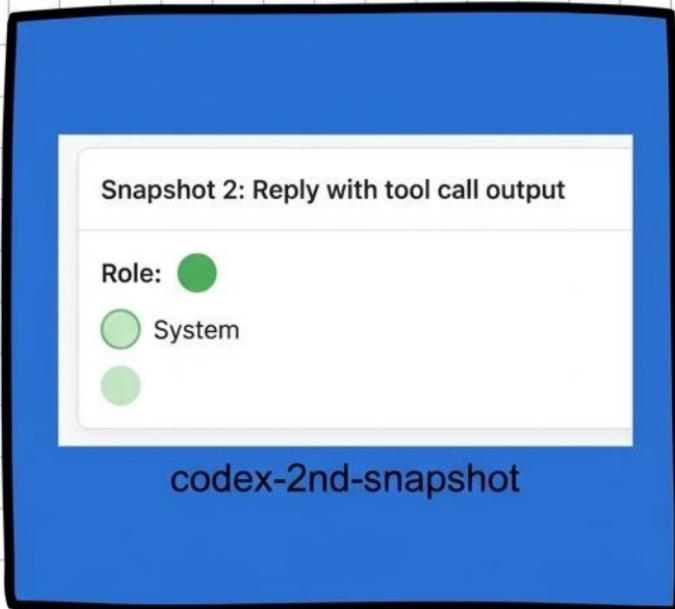
# Codex：绝对被低估的王者 (cont.)



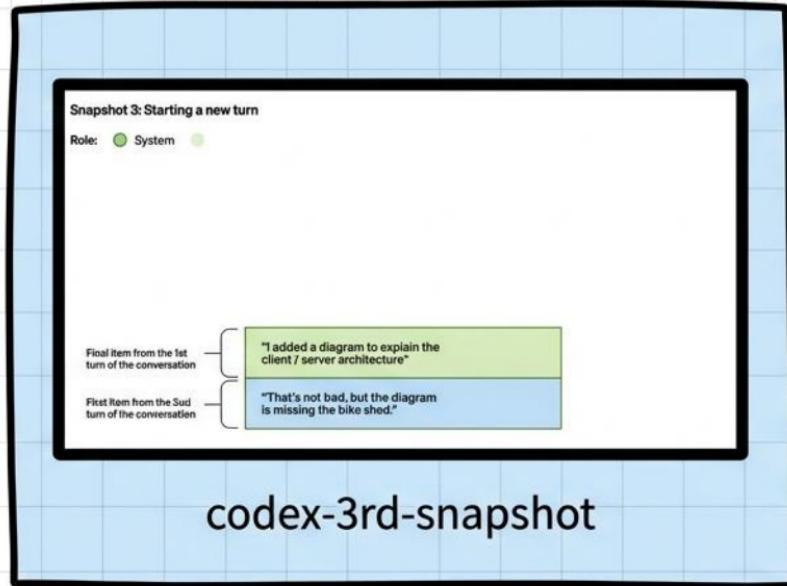
- 展示一下prompt的示意图



# Codex：绝对被低估的王者 (cont.)



# Codex: 绝对被低估的王者 (cont.)

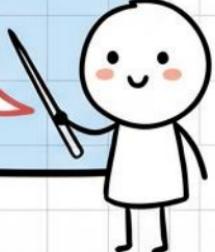


- 展示一下prompt的示意图



# 体验对比

- Copilot: tab completion 很舒服，但只适合“你已经知道要写什么”。 ✓
- Cursor: human review 密集型，把人类当CI，适合高手。 🧑
- Claude Code: 沉浸式 Vibe coding，容易停不下来；但实现上可能出现“完成了！”然后代码里全是 placeholders 的鸟语花香。 💭
- Codex: 更愿意干活、也更爱自证；但不那么听话，语言系统很晦涩，节奏偏慢，逼着人人频繁context switch (耍手机, 划掉)。 ❌
- Claude Code 更像“只会沟通扣666的队友”，Codex 更像“能carry但脾气怪的高手”。



# 体验对比：Claude Code vs Codex（谁更强？先问你想不想被哄）

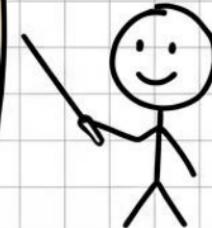
- 想要通过和agent聊天消磨时间，那就选CC.



- 想要托管但又得花大心思去理解设计，那就选Codex.



- 可以移步：《为什么Codex似乎更强，Claude Code却更流行？》微信公众号，十分形象公允，比如：Claude Code的使用体验是真正的Vibe，停不下来。Codex的使用体验像是和谢耳朵打交道，得停下来思考，不够Vibe。



# 方法论：真的能Vibe coding?

- 我个人很难在真实科研/项目里完全 Vibe coding：至少要掌握 feature / architecture / example，否则很容易失控：无法提出正确的指导意见。甚至最终验收的版本还是建议要做到百分百掌握代码。



- 在项目/科研上,情商/对话可能真的不重要,谁真的希望一轮又一轮地跟agent对话呢,一个字都不想多说的啊!

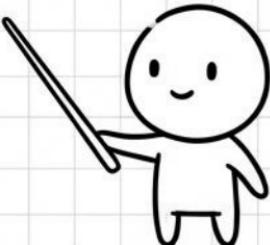


# 实战 1: EggMind

- 背景: Egglog是支持EqSAT optimization的语言+engine.



- 目标: 对任意给定domain 做 scalable superoptimization (例子: Isaria, ASPLOS 2025 distinguished paper)。
- 技术: LLM



# 实战 1: EggMind (cont.)

- .gitmodules
- CLAUDE.md
- README.md # 包含了背景和非常粗犷的技术(毕竟完全没想好怎么做)
- docs
  - egglog.pdf
  - guided-eqsat.pdf
  - guided-tensor-lifting.pdf
  - isaria.pdf
  - llm-eqsat.pdf
- egglog
- 接着, 我开始让CC做不设计具体创新点的纯框架设施搭建。
- 我给 CC 的 prompt 风格 (示例) :
  - "Use Python for infra, but interact with egglog in Rust; propose architecture."
  - "Use Pixi for environment setup."
  - "Use multi-process architecture and message passing."
  - "Write docs and tests for each feature."



## 第一个commit:

misc: initialize project with README.md, egglog as submodule, and reference papers.



# 实战 1: EggMind (cont.)

- 得到一个又一个的feat commit

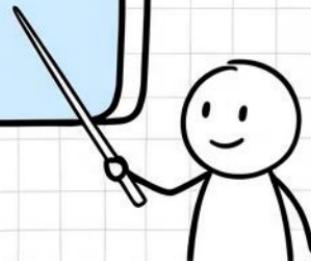


feat: complete core infrastructure with Rust bindings and multiprocess communication  
refactor: migrate to class-based Functor system with FixedFunctor  
feat: Enhance LLM client with YAML configuration support and error handling  
feat: add performance metrics tracking and evolved function logging  
feat: add in-memory snapshot/rollback system and agent forking  
feat: add customizable metrics system and Makefile for cached builds  
docs: add comprehensive Meta-Control system design  
feat: implement Meta-Control system for agent orchestration  
feat: add LocalMetaCtrl demo and reorganize documentation  
feat: add interactive debug visualizer with process tree and message flow  
... more



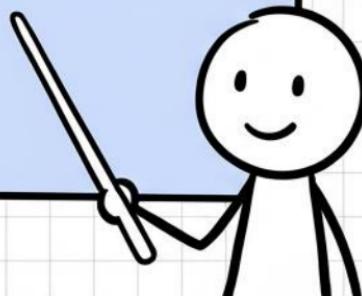
# 实战 1: EggMind (cont.)

结果：节奏确实快，但我被迫高频对话验证；如果不盯紧，repo 容易出现金玉其外败絮其中的屎山，逐渐体现为怎么对话也无法通过的Bug。我每天要花 10h以上和CC对话，他也从不告诉我他在写代码糊弄我逗我玩。



# 实战 1.5：写论文真能Vibe

- paper writing 反而更适合 Vibe：因为产物是 text，都是一眼看过去就有数的东西，不是深藏在代码里的情绪炸弹。
- 但即使如此：我最后拿到一个 30+ pages manuscript，内容是一团糟；
- 不过它确实是个不错的 starting point，结构格式基本内容是有的。



# 方法与教训：怎么避免被 agent “糊弄”

- 纯对话式迭代：很费时间；你每隔 1 分钟要看一次 response，再花 2 分钟写 comments；如果卡住就得看代码，发现全是假的。
- 文档总结：能部分解决遗忘；但往往嘱托和要求只存在 context 中，一旦 compact，就容易丢失，然后错误会“复发”，这非常气人。
- 一个不太好笑但真实的结论：agent 让生活很充实，因为你一直在 review 和反复纠正。



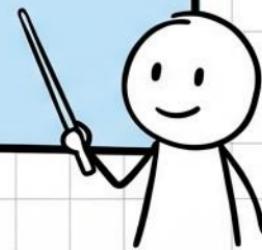
# 解决方法：把计划写进文件（顺便逼自己验收）

- **planning-with-files**: 用 persistent markdown files 做 planning / progress tracking / knowledge storage。
- **ralph-loop**: 迭代执行 loop (实现→验证→记录) , 强调调 completion promise。
- 我喜欢它们的原因：把“设计/计划/验收”从 chat 里拉回到 repo 里，减少被“上下文遗忘”支配的恐惧，也减少被agent绑架聊天。



# planning-with-files

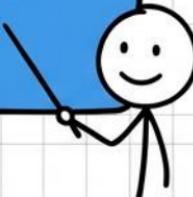
- 口号: Work like Manus — the AI agent company Meta acquired for \$2 billion.
- 蹰麻了。
- <https://github.com/OthmanAdi/planning-with-files>
- 解决了对话式迭代的问题, 先通过快速对话敲定设计细节后, 使用planning-with-files展开多任务的实施计划(一定要有验证!) , 然后就可以让coding agent自己实现, 我们去做项目级并发(project-level parallelism, 笑)。



# planning-with-files (cont.)

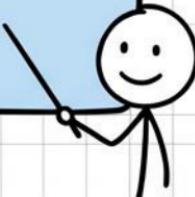
- 关键: 维护三个文件.
- 吐槽: 其实我觉得只有`task\_plan.md`有用.
- `planning-with-files`是一个Skill, 等下, Skill?

File	Purpose	When to Update
`task_plan.md`	Phases, progress, decisions	After each phase
`findings.md`	Research, discoveries	After ANY discovery
`progress.md`	Session log, test results	Throughout session



# Agent Skill是啥？

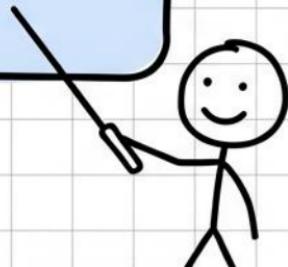
- <https://agentskills.io/home>
- Agent Skills 一组instructions, scripts, resources, 帮助 agent掌握特定任务所需的知识或技巧。
- Skills能够被agent在匹配的场景中按需加载, 向agent的 context中加入prompt. 适合场景?
- 新能力:
- 新能力: 如正确使用某哥工具 (e.g. 创建新Skill, 创建PPT)。
- 重复工作流: 把多步操作创建为被验证正确的可复用Skill.
- 以及更多!



# Agent Skill长什么样？

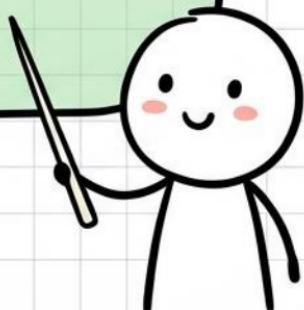
```
my-skill/
├── SKILL.md      # Required: instructions + metadata
├── scripts/       # Optional: executable code
├── references/    # Optional: documentation
└── assets/         # Optional: templates, resources
```

Youwei Xiao



# Agent Skill是啥? (cont.)

- 宝藏小站 [skillsmp.com](http://skillsmp.com)
- 如何创建Skills?
- 直接让agent自己创建, 比如在Codex里输"\$skill-creator, create a skill for xxx".



# 比较Skill和...?

- **Hooks:** 某个事件发生时执行的固定脚本或者Prompt.
- 如PostToolUse做个什么检查之类的。
- **Subagents:** own context window\*\* with a custom system prompt, specific tool access, and independent permissions.
- 我很不喜欢也基本不用subagents，因为独立的context意味着前文对话信息的丧失！
- 比如这里的CC的Plan agent，我使用的效果总是很差。

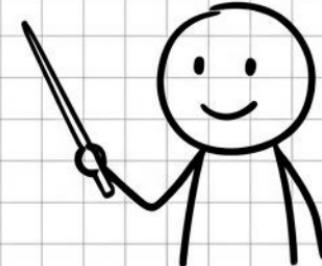
- **Plugin:** 组装一些skills, hooks, agents，比如/ralph-loop:start/stop，这里ralph-loop就是一个插件，包含一些skills (start, stop).
- **MCP:** MCP servers give Claude Code access to external tools, databases, and APIs.  
常用的比如github，会在CC/Codex里内置。
- 其他的场景基本都可以被Skill覆盖。



# 实战 2：PTO-WSP (装备成型)

- 目标：在 Ascend 平台上做算子编程相关探索，从“one kernel”到“multi-kernel runtime”。
- 我先给CC提供了：
  - pto-isa的仓库
  - 汪超师兄提供的需求文档 <- 这个很重要
- 因为我个人觉得USL(user-scheduling language)是编程runtime的正确打开方式, 所以把Halide, TVM, 以及更多论文的pdf直接提供给CC。

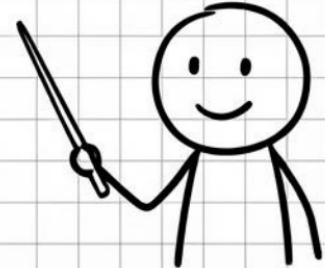
- 同时，我觉得flashinfer的JIT思路是正确的, 所以相关的代码和论文也都给CC.
- 我还希望实现megakernel!也都给!
- 毕竟是面向Ascend平台, 我把CANN的一些文档也喂给CC。



# 实战 2: PTO-WSP (准备妥当)

- 分析参考论文与仓库
- 01\_flashinfer.md

- 02\_gpu\_patterns.md
- 03\_pl\_design.md
- ...
- 16\_dato.md



# 实战 2: PTO-WSP (一败涂地)

现在,我只要求CC帮我生成满足需求文档的设计,不Coding (反正CC也挺难写对的).  
然后我浪费了接近一周的时间得到了6个我不咋满意的版本:

- v1: raw task graph, 无动态循环支持
- ✓ v2: task gen (thread 0) + task graph execution (thread 1-N).
- 其实这个和目前pto-runtime的设想就很相似了,但是并不满足我的预期.
- ✗ 仍然是动态任务生成,而非JIT,对dispatch的编程弱.



# 实战 2: PTO-WSP (一败涂地, cont.)

现在, 我只要求CC帮我生成满足需求文档的设计, 不Coding (反正CC也挺难写对的) . 然后我浪费了接近一周的时间得到了6个我不咋满意的版本:

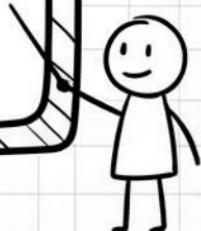
- v3-v4: Plan-Descriptor-Execute.
- 我要求对JIT, 给的例子是flashinfer, 导致过拟合.
- 任务图支持直接没了.

- v5: task graph + dispatch-issue.
  - 通过区分dispatch-issue把JIT和task graph结合.
- ✗** 但完全没提供对dispatch-issue的编程能力.
- ✗** CC还把之前的USL的东西全扔了, 退回task graph了.



## 实战 2: PTO-WSP (一败涂地, cont.)

- v6: 仍然是task graph...
  - + 加了一套基于event handler callback的编程接口 ÷ 用于dispatch-issue编程。
- 太难受.太恶心! 



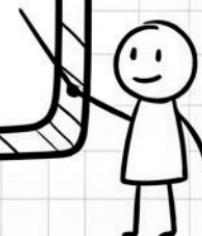
## 实战 2: PTO-WSP (拯救)

然后我被迫给了一套详细的直接指明方案的comments

### -v7: Workload-Schedule.

比较符合我的预期了, 结合JIT+task graph的可编程runtime, workload自带依赖分析. 把event-driven模型换成了CSP模型, 这是个人审美原因.

- v8: 有机合并CSP进入workload, 删减特性.
- 得到比较满意的版本.
- + 使用ralph-loop + planning-with-files完成prototyping代码, 跑通CPU Sim.
- v9: 这时看到了廖博的版本, 强调python binding. 向其靠拢.
- 得到python+CppIR+codegen的版本.



# 实战 2: PTO-WSP (Codex时代! )

在v9版本这个版本开始使用Codex.

- 初用起来体验并不好,无法像CC一样ralph-loop+planning-with-files快速出代码,因为codex没有hook机制,ralph-loop是残废的.
- 并且Python-CppIR的bridge并不trivial.
- 但在codex比较严厉的执行检查下,feature的文档和实现基本对齐.



# 实战 2：PTO-WSP (Codex时代! )

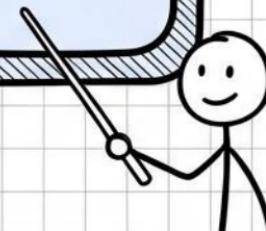
- Is the problems in docs/implementation\_review.md all resolved?
- Do the review again
- \$create-plan create plans to solve the remaining real gaps .
- docs/v9\_fix\_plan.md and docs/v9\_fix\_tracker.md shoudl be summarized into docs/task\_plan.md .
- And we should put the new plan in renewed docs/v9\_fix\_plan.md and docs/v9\_fix\_tracker.md for tracking
- Write the detailed, fully clarified plan to docs/v9\_fix\_plan.md (renew from empty) and the task tracking to docs/v9\_fix\_tracker.md (also renewed).
- \$ralph-loop start doing and complete tasks in docs/v9\_fix\_tracker.md .
- Review the implementation again per docs/implementation\_review.md , is the features in docs/features.md well implemented?
- Update the related documents to reflect the up-to-date status.



# 实战 2：PTO-WSP（做大做强）

- v10 (in plan): 汪超师兄的pto-runtime项目解耦了npu目标代码生成,计划将PTO-WSP与其集成.

- Prompt:
- We are working on a decoupled pto-runtime, cloned in @references/pto-runtime.
- For v10, we actually should target the pto-runtime.
- We should do detailed analysis on the pto-runtime.
- Indeed, the upcoming features of pto-runtime is previewed in docs/future/pto-runtime-task-buffer.md, we should take it into consideration.
- We should also think about how to integrate pto-wsp with pto-runtime (submodule?).
- Write down all note and update documents under @docs/future

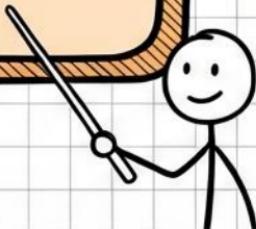


# 在PTO-WSP上，我所尝试的

- 在这个过程中我也尝试了Kimi Code + Codex，这里codex也是skill（怎么创建？让agent自己创建就好）。
- 在AGENTS.md里写：
- Kimi Code的界面和操作还是比较舒服的，甚至有GUI。

**## Important: Delegate to Codex (even for other agents)**

This repo is designed so **\*\*all concrete work is executed through Codex\*\*** (by the codex Skill).



# 在PTO-WSP上，我所尝试的 (cont.)

- Kimi Code Cli也没有Hook等机制，所以不是CC一样合格的监军。
- 所以最好还是让CC调Kimi API：

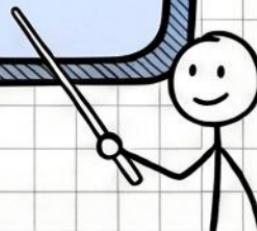
但是！

```
export ANTHROPIC_BASE_URL=https://api.kimi.com/coding/  
export ANTHROPIC_BASE_URL=https://api.kimi.com/coding/  
export ANTHROPIC_API_KEY=sk-kimi-xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx  
xxxxxxxxxxxxxxxxx # Fill in the API Key generated on the membership page  
claude
```



# 在PTO-WSP上，我所尝试的 (cont.)

- 然而我其实现在更习惯直接用Codex了.
- 虽然他语言晦涩, 但也基本习惯了, 而且codex自己的 planning模式也好用起来了.
- 而加一层Kimi Code, 他经常会自作主张, 不好好传递任务, 导致额度不够用, 任务完成的也不好.



# 想试但还没试: Superpowers / Parallel Vibe Coding / ClawdBot

- Superpowers: workflow 很强大，但我个人不喜欢功能过剩（可能会实测，但先按下不表）。
- Parallel Vibe Coding: 用 Git worktree 并发跑多个 agent session，但我目前是inter-project parallelism，还没开始发挥 intra-project parallelism。
- ClawdBot/MoltBot/OpenClaw: 名字天天变，先等等稳定（我已经用 ClawdBot 把组里 CC 搞封号了，这段不是笑话）。

# 使用Agentic Coding Tool的自省规则

- 牢记每一次对话后自己必须完全掌握的项。
- 先写 README.md, 让 agent 创建文件结构, 这一步还是挺爽的。
- 用 references (papers/repos) 提供知识。
- 该装 Skill 就装 Skill (可重复工作不要每次重讲) 。

- 讨论设计要非常仔细,失之毫厘谬之千里,等生出代码了就晚了!
- 使用plan和持久文档保存设计与要求。
- 迭代版本(叹气)。

# 感谢大家聆听！

欢迎一起交流呀！

