

D2 第二十届D2技术大会
20TH D2 TECHNOLOGY CONFERENCE

AI·新

AI 驱动 创新

2026.03.24

阿里巴巴全球总部_访客中心
中国杭州 HANGZHOU CHIA

INTERACTION &
VISUALIZATION 互动与可视化

TERMINAL
TECHNOLOGY
终端技术

DOMAIN
EMPOWERMENT
领域赋能

INNOVATIVE
EXPERIENCE
创新体验

INTELLIGENT
PRODUCTION
智能生产

ENGINEERING
R&D
工程研发

LANGUAGE &
FRAMEWORK 语言与框架



从上下文工程到 Harness Engineering

构建高效的 Agent 执行环境：上下文工程 & 工具链设计

分享人：周晓 | Web Infra AI coding 负责人

About me



周晓 / zhoushaw



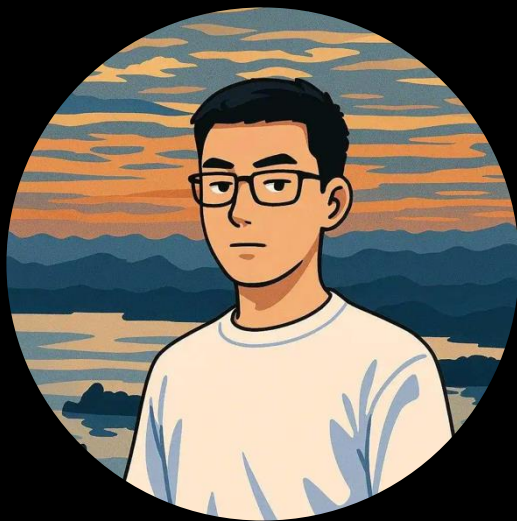
字节 Web Infra 成员



AI Coding / Harness Engineering



Midsceen.js / Module Federation 2.0



AI 时代的研发焦虑与反思

是什么让我们在代码生成的狂欢中感到不安？

Code Review 困境
面试体系失效
手写退化
职业护城河
算法霸权
终局
技术债黑盒
AI 幻觉
体验平庸化
架构腐化
前端出路在哪
角色重新洗牌
工程判断力
降维打击
八股文无用
行业颠覆

为什么 AI 出现后，研发普遍觉得**更累了**？

真正的瓶颈：编码提速并未打破系统性天花板，非编码工作量正在随着代码生成量的增加而指数级爆发。

软件开发工作时间分配



局部最优陷阱

AI 目前只在开发环境里极速写代码，但并没有帮你解决测试、验证和沟通等环节。价值上限被后续流程死死锁住。

痛点爆发：代码生成得越快越多，你用于手动测试、验证排障和 Code Review 的工作量正在**呈指数级上升**。



破局：Harness Engineering

必须从“单行代码补全”跃升至让 Agent 执行跨越整个生命周期的**长时间闭环任务 (Long-term Tasks)**。

核心路径：通过专属工具链集成，解决上下文爆炸问题，让 AI 接管测试与排障，彻底打破 70% 的非编码流程枷锁。

范式转移：什么是 Harness Engineering？



工程师角色转变

从“编写代码”转向“设计环境、明确意图”



Harness Engineering

构建 Agent 专属工具链，实现受控执行



Agent 可读性

北极星指标：代码库重塑为模型能“读懂”的环境

“不要只把大模型看作‘大脑’，必须为它打造专属的‘工作室’。”

PART 01

唯快不破 — 从 Transformer 原理到 Prompt Cache

Transformer 自回归、KV Cache、Prompt Cache、ReAct 循环、上下文布局

Claude Code / Codex 在遵循哪些“物理限制”？

你觉得产品“慢、笨、会幻觉”？其实每个行为都是一道工程选择题

SLOW / 慢

并非产品 Bug

自回归逐 Token 生成是物理铁律；上下文越长，Prefill 就越久，这是算力的基本开销。

DUMB / 笨

并非模型能力弱

上下文塞得越多，注意力被稀释得越严重。在万行代码中，关键逻辑往往被掩埋。

HALLUCINATION / 幻觉

并非故意欺骗

当上下文中缺少事实锚点，模型为维持概率连贯，只能进行“合理的猜测”。

产品设计并非在对抗限制，而是在做 **Harness Engineering**：顺应物理限制，在约束中求解最优体验。

1.2–1.3 KV Cache & Prompt Cache

解释了为什么 System Prompt 要保持极高稳定性——只有稳定，缓存才能命中，响应才能加速

1.4 ReAct 循环机制

解释了为什么 Agent 要“想→做→看”反复迭代——利用外部反馈来对冲模型的概率波动

上下文布局与压缩

解释了为什么会自动 Compact——在有限的注意力槽位内，塞入信噪比最高的信息

理解这些约束 = 理解 Harness Engineering 的底层逻辑

NEXT: 以 Transformer 第一个 Token 为起点，逐层拆解 →

1.1 为什么是一个 token 一个 token 吐？

自回归分解

LLM 本质是"下一个 token 预测器"

$$P(x_1, \dots, x_T) = \prod P(x_t | x_1, \dots, x_{t-1})$$

- ▶ 流式渲染暴露了模型自回归的物理现实
- ▶ 无 KV Cache 时，吐第 100 词 → 前 99 词全部重算
- ▶ Coding Agent 塞入几万字 → Prefill 阶段极其昂贵

PyTorch 朴素推理 (无 KV Cache)

```
for step in range(max_tokens):  
    # 朴素实现：每步全量前向  
    out = model(input_ids)  
    next = argmax(out.logits)  
    # 拼接 → 序列越来越长  
    input_ids = cat([input_ids, next])
```

结论：上下文越长 → Prefill 越重 → 首 Token 延迟越大 — 这就是 KV Cache 的动机

1.2 KV Cache : 为什么"追加"便宜？



推理两阶段：Prefill（全量算 KV）➤ Decode（复用 KV）

Attention 的物理本质：

Q：问题 | K：标签 | V：细节

标准 Attention 公式

$$\text{Attention}(Q,K,V) = \text{softmax}(QK^T/\sqrt{d}) V$$

长任务黄金定律：

✅ 追加 (Append) 极度便宜

老 KV 复用，新 token 只算一次

❌ 中间修改 极其昂贵

修改点之后的所有 Cache 全部作废重算

⚡ 核心结论：一切上下文工程，本质上都是为了“尽量不破坏前缀”

追加 = 复用 KV, $O(\text{新 token})$ | 修改 = 从修改点起重算 KV, $O(N)$

1.3 Prompt Cache: 多快省

缓存KV → 跳过Prefill → 更快+更省

Claude Sonnet 4.6定价实例

Base Input

\$3.00 / MTok

\$3.00 / MTok

Cache Write (5min)

\$3.75 / MTok (1.25x)

1.25x

Cache Read

\$0.30 / MTok (0.1x)

↓ 90%!

⚡ 更快



Req

→



Cache

→



→



TIFT: 大幅缩短

💰 更省

Cache Read 仅 0.1x 基础价格。

更多上下文 → 更多节省。



🔄 工作原理

Input

+



→



Write KV, 1.25x (5min)

Input

+



→



Read (0.1x), 节省90%

1.4 ReAct : Agent 的操作系统

不是"一问一答", 而是 Thought → Action → Observation 的循环

一个真实的 ReAct 过程

User: 给 user-service 加 /healthz 接口

Reason: 先找路由注册方式.....

Act: search / read_file

Observe: 返回文件内容

Reason: 原来是 FastAPI, 该这样写.....

Act: write_file + bash pytest

Thought (思考)

"需要查看 index.ts 的错误信息"
模型在上下文里"自言自语"

Action (行动)

read_file() / write_file() / bash()
调用工具, 影响物理世界

Observation (观察)

返回文件内容、终端输出、报错信息
结果追加到上下文 → 触发下一轮

Turn 1

Turn 5

Turn 20

上下文随迭代爆炸 → 第一原则 : 保护前缀稳定性 → 接下来 : 怎么做 ?



思考一下

你的 Agent 工具，平均每轮往上下文里塞多少 token？

如果超过 5,000 token，你的 Cache 命中率可能已经崩了

接下来：从原理走向实战 — 上下文工程的具体策略

PART 02

上下文工程实战

Prompt 布局、缓存优化与上下文膨胀的应对策略

一次 Agent 任务背后：Prompt 的五层结构

不是"把你的话丢给模型"，而是构造了一整坨 Prompt

1 System

模型服务商系统提示 + Agent 身份定义 + 工具能力声明

2 AGENTS.md

技术栈、代码风格、安全边界、团队工作流

3 项目快照

当前工作目录、选中文本、项目结构

4 会话历史

对话记录 + 工具调用与返回 + snapshot / 总结

5 工具定义

MCP 工具清单：名称 + 说明 + JSON Schema

⚡ **核心结论：**上下文不是"越多越好"，而是要"把最重要的东西放到最该放的位置"

所有这些一起拼成 Prompt → 位置和占比决定了模型的注意力分配

记忆的艺术： Compaction + AGENTS.md

上下文不是越多越好 — 要主动压缩、主动归档、主动锚定

主动 Compaction 流程



好处:

- ✓ 版本管理 & 可 review
- ✓ 对话不堆积
- ✓ 统一“状态锚点”

AGENTS.md — 项目宪法

≡ A

- 自动放置 Prompt 最前面
- 不会被压缩/总结
- 所有 Agent 默认遵守

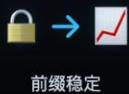
✓ 建议写入内容

技术栈/框架
代码风格约定
安全边界

✗ 不要写入内容

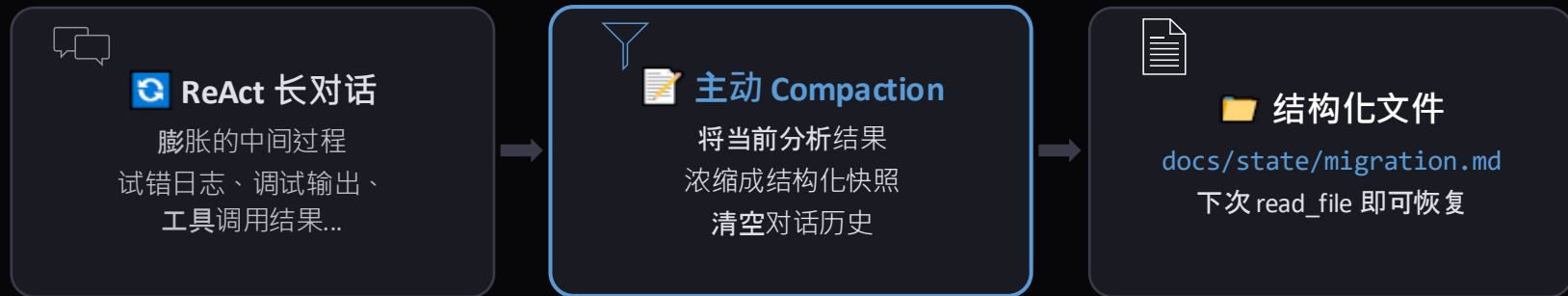
一次性需求
经常变动的信息
临时重构目标

⚡ 核心策略实现



抵抗膨胀：将上下文外包给"知识记录系统"

不在代码仓库里的隐形知识，对 Agent 来说就是不存在的



⚠ 隐形知识的陷阱

业务决策如果只在聊天记录里，对 Agent 就是不存在的。

正确姿势：执行完一个阶段 → 将探讨结论写成 Markdown 快照 → 清空对话历史 → 下次 `read_file` 恢复

⚡ **核心结论**：记忆外包给文件系统 — 但上下文治理只是一半，工具设计同样关键 →

到这里，你需要带走的 3 条核心结论

1 自回归 → 上下文越长，重复计算越多

每生成一个 token 都要重算所有历史 — 这是模型的物理现实，无法绕过

2 KV Cache → 追加便宜，修改昂贵

保护前缀稳定性 = 复用缓存 = 省钱 + 提速，中间插入 = 缓存作废

3 上下文是最贵的资源 — 治理它，而不是堆砌它

三段式布局 + 主动 Compaction + 记忆外包给文件系统

 **一句话：**所有后续策略的根基 — 保护前缀稳定性

PART 03

工具设计与 Agent 可读性

Agent 可读性、好工具标准、MCP / Skill / Sub Agent、闭环校验

核心范式转移：提升 Agent 可读性

从"对人友好"转向"对 Agent 可读" — 环境降维成结构化语义

过去的 Infra 陷阱

酷炫 GUI / 海量终端日志

直接把给人类看的界面、十几页火焰图扔给 Agent

→ Token 瞬间爆炸 · 上下文被垃圾填满

面向鼠标点击的交互

Agent 无法操作 GUI 按钮和拖拽界面

→ 人机交互范式完全不适用

视觉冗余图表

图形化仪表盘对 Agent 完全不可读

→ 缺乏语义 · 无法提取结构化信息

范式转移：一切为了 Agent 可读

结构化语义 (JSON / API)

精确、唯一的术语定义

Agent 直接解析，零歧义

面向自然语言指令

用文字描述操作意图 · 取代鼠标点击

Agent 原生理解，自主执行

高信噪比的诊断摘要

极致精简 · 层级分明的上下文导航

秒级反馈 + 100% 稳定结构化输出

Harness Engineering 的核心：给模型好的上下文、好的工具、可读的环境 — 这就是“为 Agent 打造专属工作室”的全部

好工具的标准

Agent 的工具质量直接决定 Harness 工程的成败



贴近物理上限

秒级冷启动与响应，防止 Agent 注意力涣散。



结构化输出

禁止 PDF/网页截图。必须输出 JSON/Markdown。



痛觉反馈

错误信息是闭环的起点，需精确到行号与类型。

⚡ 核心原则：工具不是给人用的 UI — 是给 Agent 用的 API。快、准、结构化，缺一不可。

大脑分工：MCP、Skill 与 Sub Agent

当任务足够复杂，光靠原子工具是不够的 — 我们需要给大脑分工



MCP

原子动作 · 常驻上下文

一次性挂在上下文里，提供基础的"原子动作"

GitHub · Slack · Database · K8s · Internal APIs



Skill

可见的 SOP · 按需激活

把成熟打法（CI Debug、审查清单）固化成可执行文件夹，在主会话里按步骤跑

skill.md + run.sh + examples/



Sub Agent

黑盒外包 · 并行大脑

完全独立的会话，上下文隔离 — 适合超长任务拆解

例：Explore 模式 — 起 3 个并行子 Agent 探索不同微服务，最后汇总 Summary

⚡ **核心思路**：用架构隔离，对抗长任务的熵增 — 原子动作 → 固化打法 → 并行外包

闭环校验与编辑：机器人的 LSP

自动化的 "Lint → Fix → Verify" 闭环

1. 静态分析

AST 检测
提前发现冲突



2. 增量重构

Agent 自动应用
最小差异



3. 动态验证

运行测试用例
真实反馈

⚠ 为什么需要闭环？

Agent 每一次编辑都可能引入新错误。没有 Lint → Fix → Test 的自动闭环，Agent 就会在错误上堆叠错误，直到上下文爆炸。

⚡ **核心结论：** LSP 不是可选项 — 它是 Agent 编码质量的最低保障线。

PART 04

多智能体协作与受控执行

Agents Team、人机协作、Spec/Plan — 从单兵到团队作战

Spec Kit 与 Plan 模式：谋定而后动

先写规格 · 再写代码 — 减少无目的的试错循环

Spec Kit (规格说明书)

定义"做成什么样"：

- 输入/输出 Schema
- 副作用边界 (Side Effects)
- 验证通过标准 (Success Criteria)

Plan 模式 (执行计划)

定义"怎么做"：

- 多步任务拆解
- 关键检查点 (Checkpoints)
- 失败回退策略 (Rollback)

 **核心结论**：Spec 答 What, Plan 答 How — 不确定时在第 1 轮就向人求助，而非猜错后浪费 20 轮

人机协作：Agent 何时该自主，何时该求助

Harness 不是让 Agent 完全取代人 — 而是让人机各司其职



受控闭环



Agent 自主执行

- Spec 明确 + 有闭环验证
- 工具能力覆盖完整
- 失败可自动回退

→ 放手让 Agent 跑



向人求助 (AskUser)

- 意图模糊 / 多条路径
- 环境信息不足
- 涉及不可逆操作

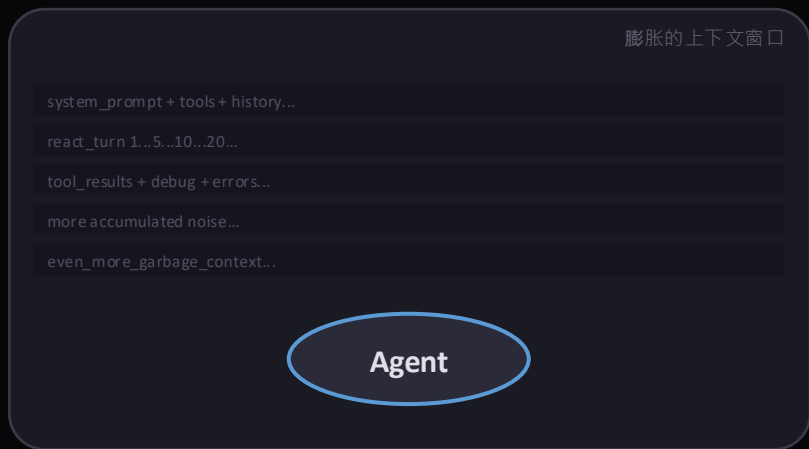
→ 第 1 轮就问 · 不猜

⚡ Harness 的边界：好的工具链让 Agent 知道自己能做什么、不能做什么 — 这本身就是 Harness

突破智能上限：Agents Team 与结构化通信

单 Agent 天花板 = 模型基础智商 → 用多智能体协作来突破

单 Agent



上下文膨胀 → 注意力稀释
智商被锁死在模型基础水平

Agents Team



仅通过结构化摘要通信

物理隔离噪音 · 独立上下文
交叉验证 · 突破智能上限

核心理念：用"结构化通信"代替"无脑堆砌" — Agent 在独立沙盒工作，只交换摘要



回想一下

上次让 Agent 跑一个超过 20 轮的任务,
它在第几轮开始"变笨"?

接下来看看我们的工程化解法 — 如何让 Agent 稳定跑完全程

PART 05

释放人类注意力 & 构建 AI native 工具

AI 降维 · 全链路流水线 标准化基建

基于 Agent 的 CLI 自迭代端到端闭环

```
1. Select
2. cli-test
3. aic-test-docs-generator
4. qtilepiatiliab-action
5.
6. [1] Nav [Select-Use Cancel]
```



Step 1: 需求输入 (Input)

自然语言描述交互修改需求



Step 2: AI 自动编码 (Code)

Agent理解需求，自动定位并修改代码逻辑 (UI层与 Middleware层)



Step 3: 自动化沙盒测试 (Test)

核心动作: AI 调用内置
``cli-test`` skill

技术细节: 基于 ``tmux`` +
Aiden CLI 操作手册, 进行行实的终端 UI 交互自测
自我验证通过, 确保功能满足需求



Step 4: 交付验收报告 (Output)

AI 自动沉淀飞书交付文档

- 变更范围说明
- TUI 截屏前后对比
- 自动化测试执行方案



从 Review 代码到 Review 交互和测试

> SKILL.md	.aiden/skills/cli-test/SKILL.md	+1 -1	Viewed	...
> interaction-guide.md	.aiden/skills/cli-test/interaction-guide.md	+37 -19	Viewed	...
> feat-slash-skill-merge.md	.changeset/feat-slash-skill-merge.md	+5 -8	Viewed	...
> plugin_command.test.ts	next/cli/e2e/plugin/plugin_command.test.ts	+1 -1	Viewed	...
> skill_dollar.test.ts	next/cli/e2e/skill/skill_dollar.test.ts	+8 -468	Viewed	...
> custom_slash.test.ts	next/cli/e2e/slash/custom_slash.test.ts	+5 -5	Viewed	...
> normal_slash.test.ts	next/cli/e2e/slash/normal_slash.test.ts	+3 -2	Viewed	...
> index.ts	next/cli/src/ui/commands/index.ts	+2 -8	Viewed	...
> skills.tsx	next/cli/src/ui/commands/skills.tsx	+11 -8	Viewed	...
> SkillsPanel.tsx	next/cli/src/ui/components/SkillsPanel.tsx	+119 -8	Viewed	...
> SkillSelectionDropdown.tsx	next/cli/src/ui/components/SmartInput/SkillSelectionDropdown.tsx	+8 -189	Viewed	...
> SlashCommandDropdown.tsx	next/cli/src/ui/components/SmartInput/SlashCommandDropdown.tsx	+84 -42	Viewed	...
> index.tsx	next/cli/src/ui/components/SmartInput/index.tsx	+6 -15	Viewed	...



MR: 合并 Skill 与 Command 到统一的 / 下拉菜单

修改范围

核心代码变更 (next/cli/src)

E2E 测试变更 (next/cli/e2e)

文档变更

E2E 测试策略

截屏对比

改前: 独立的 \$ 触发 skill 下拉

改后: 统一的 / 下拉 (builtin + command + skill 混排)

改后: 过滤搜索 skill

改后: 选中 skill → 输入框显示 /skillName

改后: /skills 内置命令

改后: \$ 不再触发任何下拉

Skill 完整执行流程

Step 1: 输入 /echo, 下拉列表显示 echo-test skill

Step 2: 按 Enter 选中, 输入框显示 /echo-test (斜杠前缀, 下拉消失)

Step 3: 追加参数 hello world, 输入框显示 /echo-test hello world

Step 4: 按 Enter 提交, / 自动转换为 \$, SDK 正常执行 skill

端到端闭环交付, Aiden CLI

需求: 将 CLI 的 skill 列表和 command 列表合并为统一的 / 触发下拉菜单 (对齐 Claude Code 交互)。

改前: 独立的 \$ 触发 skill 下拉

代码块

```
1
2 > $
3
4
5 $ Select Skill (search...)
6 > analyze
7   cli-test
8   e2e-test-debugger
9   tmates-docs-generator
10  gitlab:gitlab-action
11
12 ↑↓ Navigate • Enter Select • Esc Cancel
13
```

/ 下拉只有 builtin 和 markdown command, skill 需要用 \$ 单独触发。



改后: 统一的 / 下拉 (builtin + command + skill 混排)

代码块

```
1
2 > /
3
4
5 Type to filter • Tab to complete
6 /agents
7   Manage subagents - create, view, delete [builtin]
8 /analyze
9   Analyze a LangSmith trace by id... [command:project]
10 /analyze
11   Fetch LangSmith traces and threads... [skill]
12 /clear
13   Clear agent context, session history, and screen content [builtin]
14 /cli-release-note
15   获取版本发布日志并更新 CHANGELOG.md [command:project]
16 /cli-test
17   Test the CLI by launching it in tmux... [skill]
18 /compact
19   Manually trigger context compression to save tokens [builtin]
20
21 Showing 1-7 of 27 commands
22
23 ↑↓ Navigate • Enter Confirm • Esc Cancel
24
```

Midscene.js: 为 Agent 打造 AI Native 的工具

当 Agent 需要操控网页时，有两条路径

Agent 自己控制网页

获取网页 DOM 树

`document.querySelector(...)`

解析结构 / 截图识别

视觉模型 + 坐标映射

定位目标元素

CSS 选择器 / XPath

模拟人类操作

`click(x,y)` / `type(text)`

Agent 在做人类才需要做的事

上下文爆炸 · 脆弱 · 无法自愈

Agent + Midscene.js

Agent 用自然语言描述意图

"点击登录按钮" "验证购物车有3件商品"



Midscene 原理解理解并执行

自主定位 · 自动适配 · 自愈重试

AI Native = 为模型设计的工具

不获取 DOM · 不截图 · 不模拟人类操作
自然语言本身就是 Agent 的操控接口

好的 Harness = 为 Agent 量身打造 Native 工具

从“模拟人类”到“AI Native”：重塑 Agent 开发全链路闭环

核心价值总结：释放人力注意力，构建高效工具



1. 破局：将人从“AI 调试器”中解放

(UNBURDEN HUMAN ATTENTION)



痛点：人机纠缠

传统 Agent 需要人类密切关注过程，观察 TUI/UI 截图，进行 DOM 定位纠错，注意力被繁琐细节锚定。



解法：自我验证闭环

引入基于 `tmux` 的自动化沙盒环境与内置 `cli-test` 技能，Agent 在内部自动完成测试。



价值：结果导向交付

人类仅需审计最终生成的 飞书验收报告（含对比截图和测试报告），无需人工监控。



2. 进化：构建 AI Native 的生产力工具

(BUILD AI-NATIVE HARNESS)



理念转变

不再强迫 AI 模拟人类脆弱的操作（获取 DOM、计算坐标、模拟点击）。



Midscene.js 核心能力

- 自然语言接口：将复杂 Web 操作抽象为“点击登录按钮”等简洁指令。
- 原生理接与执行：工具层实现自主定位、自动适配与自愈重试，模型与环境无缝握手。



效率跃迁

消除上下文爆炸，提升任务鲁棒性和成功率。

总结与展望：AI 时代的研发新范式

CONCLUSION & OUTLOOK: THE NEW R&D PARADIGM IN THE AI ERA



1. 从编码者到 BUILDER

(FROM CODER TO BUILDER)

- 核心角色转变
- Harness Engineering:** 为 AI 提供工具、上下文、反馈。



2. 驾驭模型物理限制

(MASTERING MODEL CONSTRAINTS)

- 利用 Transformer 自回归特性和 Prompt Cache 问题。



3. 连接物理世界

(CONNECTING TO THE REAL WORLD)

- 模型落地需要精准工具与上下文。



4. 时间是稀缺资源

(TIME IS SCARCE)

- 人类时间是最稀缺的资源。
- 让 AI 处理繁琐任务，释放人类创造力。



Q & A

欢迎提问与交流环节

(Open for Questions & Discussion)