架构师

ARCHITECT

2025年第二季

特别专题

架构师视角的软件技术栈与语言新秩序

推荐文章

当代码遇上大模型：智能编程助手的架构设计与工程实践

InfoQ 极客传媒

CONTENTS / 目录

热门演讲实录 | 落地和进化

原创访谈 | Interview

工具图谱 | 大模型时代的编程工具

架构变革 | 大模型时代的软件技术栈

语言新秩序 | 大模型时代的编程语言

AI时代，编程语言选型更难也更重要：Go、Rust、Python、TypeScript谁该上场？

Python新版本去GIL刷屏，Karpathy点赞敢死队，Python之父：冷静，别神话并发

Cloudflare用Rust重写核心系统：CDN性能提升25%，响应时间缩短10毫秒

Cloudflare酿六年最惨宕机：一行Rust代码，全球一半流量瘫痪！ChatGPT、Claude集体失联

架构师 2025 年第二季

本期主编 Tina
流程编辑 丁晓昀
发行人 霍泰稳

反馈 feedback@geekbang.com
商务合作 hezuo@geekbang.org
内容合作 editors@geekbang.com

InfoQ 极客传媒

AiCon 北京站

全球人工智能开发与应用大会

2025年度AI热词图鉴
如何与AI顶流保持同频

12月AICon北京站
给你答案

AI安全
Agent（智能体）
多模态
AI民主化
具身智能
Vibe Coding
人工智能+
推理
Data
量子AI
基础超级模型
开源模型
AI 重塑软件研发
AI治理
企业级智能体
世界模型

12 月北京站
2025 年 12 月 19-20 日
北京石景山万达嘉华酒店

卷首语

过去几年，软件行业的叙事悄然改变。招聘市场从“前端遍地”变成“全栈为王”，AI正在吞噬样例代码、CRUD页面与重复性业务逻辑。那个毕业两三年、训练营转行就能拿到高薪的时代，确实一去不返。

但软件行业并不是唯一的“受害者”。老师在重新设计作业，分析师在适应AI总结报告，摄影师被智能相机和生成式图像重塑，连牙医、兽医诊所都在被资本流程化。几乎每一个行业，都在经历由技术、资本与叙事共同推动的剧烈重构。工程师只不过站在这场变革的第一排。因此，一个问题比以往任何时候都更重要：当写代码不再是核心壁垒，工程师的价值究竟是什么？

过去，熟练掌握框架和API、能快速出活的工程师可以稳定工作多年；但今天，这些能力正快速被AI和模板化方案覆盖。真正难以替代的，是能在混乱中抽象问题、在不确定中设计可演进系统、能把技术决策与业务目标对齐，并在系统失效时带着团队把它救回的人。

这不仅是架构师的挑战，也是所有工程师面临的新现实。

与此同时，AI重写了招聘方式与成长路径：简历可以自动生成，面试题可以提前演练。真正拉开差距的，是你如何使用工具解决真实问题：如何规划系统、如何用AI验证设计、如何做权衡，以及如何把自己的实践沉淀到GitHub、技术博客和社区里，让作品自己说话。

运气固然存在，但软件行业有一个从未改变的规律：越是持续把自己暴露在真实问题和真实用户面前的人，越显得“幸运”。他们靠的不仅是天赋，而是在一次次“做出

来、丢出去”的实践中，被反馈校正、被复杂性逼着成长。

这本电子书不会告诉你如何规避风险，也不会提供所谓的“安全路径”。它想讨论的是：在每个行业都在被重写的当下，工程师如何重新理解自己的价值，并在不确定中找到确定性。

如果你开始意识到：“写好代码”已经不够，“守住现状”反而更危险，那么这里，就是你重新定义自己角色的起点。真正值得思考的问题不是“工程师会不会被AI替代”，而是：**当软件行业的叙事改变，你准备靠什么能力继续写入下一章？**

当代码遇上大模型：智能编程助手的架构设计与工程实践

演讲嘉宾 段潇涵　编辑 Kitty

在人工智能快速发展的今天，大语言模型正在重塑软件开发的范式。从最初的代码补全工具，到基于Prompt的智能助手，再到具备自主规划与执行能力的Agent，我们正经历着开发模式的革命性转变。在InfoQ举办的QCon全球软件开发大会上，字节跳动豆包MarsCode/Trae IDE架构师段潇涵做了专题演讲“从指令到Agent：基于大语言模型构建智能编程助手”，他深入介绍了如何基于大语言模型构建新一代智能编程助手，分享了从概念到落地的完整实践经验。

内容亮点

- 在Agent技术架构方面，不同于传统的Prompt模板方案，将开发需求分解为具体

步骤，并在执行过程中进行自我反思和修正

- 在上下文感知方面，构建代码知识图谱，实时构建代码的结构和依赖关系等，解决了大语言模型处理大型代码库时的上下文限制问题

以下是演讲实录（经InfoQ进行不改变原意的编辑整理）。

在人工智能快速发展的今天，大语言模型正在重塑软件开发的范式。从最初的代码补全工具，到基于Prompt的智能助手，再到具备自主规划与执行能力的Agent，我们正经历着开发模式的革命性转变。本次演讲深入探讨如何基于大语言模型构建新一代智能编程助手，分享从概念到落地的完整实践经验。

演讲首先剖析当前研发效率的痛点，展示传统开发工具的局限性，以及大语言模型为我们带来的新机遇。接着，详细介绍了智能编程助手的技术架构，包括Agent的核心设计理念、工程能力构建方法、以及与IDE的深度集成实践。通过实际案例的演示，我们展示智能助手如何在代码理解、生成、优化等场景中提供有效支持。

作为一线实践者，我们也会分享在工程落地过程中积累的经验与教训，包括架构选型、prompt优化、系统集成等关键决策点的思考。最后，我们展望智能编程助手的发展方向，探讨如何打造真正的智能研发伙伴。

编程助手解决的痛点

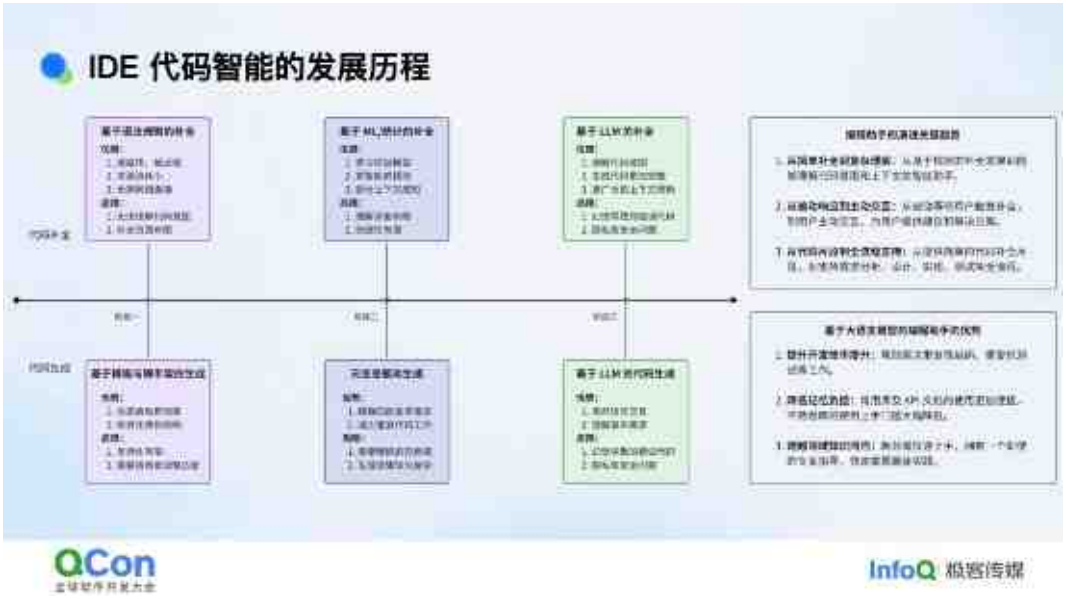
首先我们来回顾一下过往代码补全及代码生成的经验，以及在大语言模型时代，它能帮大家解决的一些痛点。我们可以看到代码智能编程助手的演进流程，大体可以分为两条线。

首先是代码补全这一条线。最早的时候，代码补全是基于语法规则的静态语法分析。现在，各家IDE（里肯定都有这样的功能。比如我们在写代码时，它会突然出现一个下拉框，列出当前类中可用的函数和变量，我们在编程中经常使用这种功能。第二个阶段是基于机器学习和统计数据分析的代码补全，它会更进一步，让我们感觉更有智能。比如曾经有一个插件叫TabNine，大家如果有用过应该有体会。它刚出来的时候非常惊艳。到了第三个阶段，也就是像Copilot这类工具出现的时候，大家会觉得大语言模型真的改变了这个时代。它能够理解代码意图，分析更多代码上下文，通过更广泛的上下文让代

码生成更符合逻辑、更加完整。它不再是一行代码片段的生成，而是会有更多上下文联动。比如现在大家常见的Tap键超级补全类的功能，它可以修改代码全文的引用。比如我改了一个函数的名字，所有调用的地方都可以通过一个Tab键全部修改掉。这个时代会让人感觉更惊艳。

再看代码生成这一条线，它也经历过几个时期。第一个时期是现在仍在使用的，比如在项目初始化时，基于某些模板或脚手架。它的优势在于各家公司可以自己定制流程，在特定场景下可以一键生成项目框架。但它的局限性在于灵活性非常受限，只能是硬编码的方式。第二个阶段是基于元信息生成代码的方案。简单解释一下什么是元信息生成：大家在前几年会经常看到类似于OpenAPI的平台。我们会声明好API的元信息，然后它可以自动帮你生成调用代码，甚至可以生成不同语言的调用代码。这在云厂商、PaaS厂商或SaaS厂商提供的SDK中比较常见。它的优势在于能够精确地支撑业务，比如云厂商提供SDK，过去多种语言的SDK都是完全手写编码，现在通过声明API，可以用各种语言的生成器，根据元信息生成多种语言的SDK，从而减少大量重复工作。不过它的劣势在于，更多依赖于元信息管理，比较复杂，而且生成逻辑需要根据不同的编程语言单独处理生成器。

到了大语言模型时代，代码生成变得更加自然。我们可以通过自然语言与大语言模型交互，要求它帮我们生成特定场景的代码。比如ChatGPT刚出来的时候，我自己感觉非常惊艳。我可以把需求以及可能已有的代码片段上下文贴到它的输入框里，它就能生成代码，而且代码质量相对来说是中等偏上。我们可以看到，代码智能的发展从最开始的简单补全，已经慢慢演化到现在可以与人协同交互，理解偏复杂场景，从代码片段补全到全流程支撑的状态。

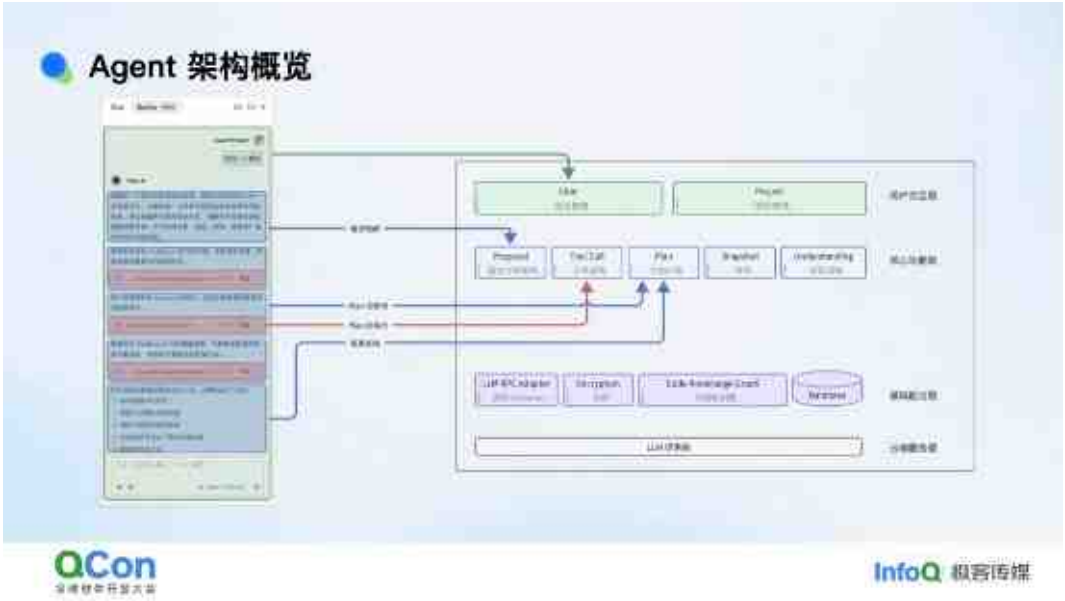


在这个时代，研发人员可以从以下几个方面受益：一是提升研发效率。研发效率的提升不一定体现在时间缩短，也可能是心态或工作量上的提升。大量的重复性编码，尤其是业务上可能存在的面条型代码，都可以由大语言模型来实现。二是降低记忆负担。我自己比较常用这个功能。在做开源项目时，会用到很多开源库，而开源库的API如果不熟悉，就需要反复查阅文档。有了大语言模型，它的记忆能力比人类更强，我们可以直接利用它来补全这些API，更加便捷。三是跨越知识鸿沟。大家可能会有这种感受，有了大语言模型，我们的能力不再局限于原来定位的工作。比如我是客户端开发人员，也可以用大语言模型帮我写一些服务端代码；我是服务端开发人员，想自己写前端控制页面，也不用再依赖学习前端完善的基础知识，或者与前端同事配合。我可以利用AI生成一个虽然可能只有五六十分，但也能用的代码。

Agent技术架构与核心能力

Agent架构概览

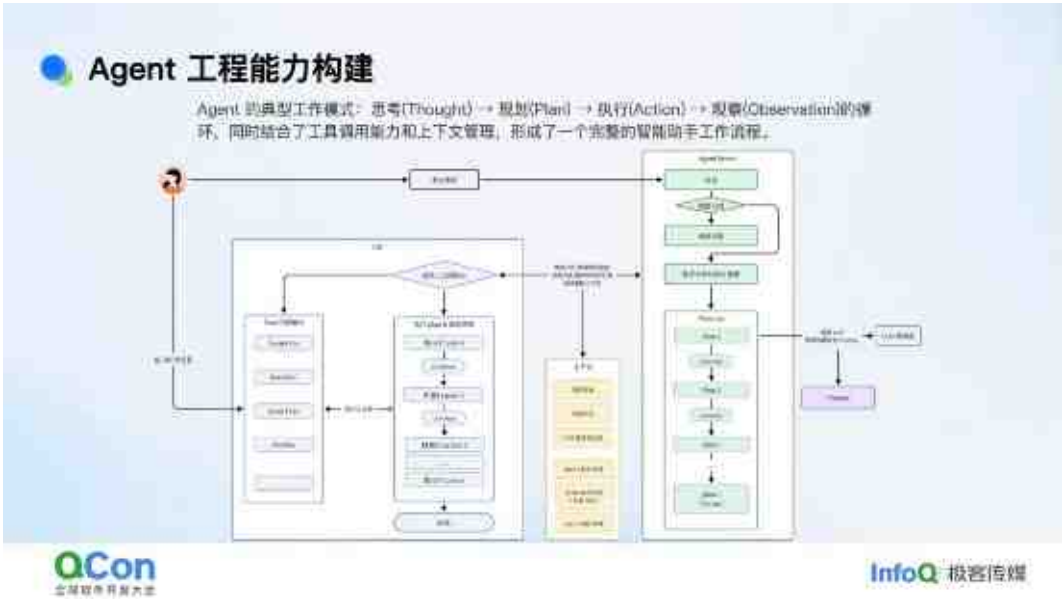
接下来我将重点讲解一个Agent的形态，它的技术架构大概是什么样的，以及我们通过什么样的工程手段可以把大语言模型结合落地到我们的IDE中。首先，我们可以通过一张比较直观的图来展示，我们的技术架构大体分为四层。



首先，最直观的是用户界面（UI）如何与用户衔接。第二层是Agent的核心功能，其中有两个模块，连线非常多，说明这是比较核心的部分。一个模块是计划的执行，另一个是工具的调用。通过蓝色和红色框区分，计划执行时，模型会输出规划，同时搭配IDE提供的工具执行操作，例如文件的修改、查询等。再往下一层，是我们为了构建Agent架构体系所需要的基础能力。其中一个关键点是代码知识图谱（Code Knowledge Graph）。大语言模型要理解项目信息，首先需要构建足够的信息，并以合适的结构和方式提供给模型。此外，技术层还需要关注大语言模型的适配器，因为目前模型厂商和种类众多，工程能力需要对接各家厂商，并协调模型之间的协作。加密和数据管理也是工程领域必不可少的环节。通过这样的架构设计，我们可以将大语言模型的能力与IDE结合起来，提升开发效率和智能化水平。

Agent工程能力构建

我们再来看一张更细致的图，了解如何构建Agent的工程能力。这张图从左到右展示了从用户需求输入到Agent主流程Server的过程。Agent主流程Server负责拆解用户的需求流程，并利用大语言模型将其拆分为逐步的规划。



与此同时，从左侧回看，IDE为大语言模型提供了工具集，这些工具集可以类比为大语言模型的“手和脚”，用于触发调用和变更代码文档，以及生成SVG素材等。从右侧往左侧看，从模型的视角出发，模型接收到的信息结构是什么，以及它能够做到什么。图中紫色框标注的是“prompt”，这与大家常用ChatGPT时输入的文本类似。这段文本进一步拆解后，可以看到黄色框标注的“上下文”，它包含许多组成部分。

Agent在IDE中的工程实现看起来非常流畅，主要是因为我们通过工程手段将这一整套环节串联起来，从而实现了从用户需求到模型执行的完整流程。

Prompt结构

我们继续深入分析下图右侧的Prompt结构。首先，从结构定义来看，Prompt包含基础的角色目标和工作方式，主要是引导模型在特定场景下以何种思路解决问题，或者在研发过程中使用何种技术栈来解决问题。接下来是工具的定义。IDE为大语言模型提供了便捷的工具，例如文本文件操作、终端（Terminal）联动、预览（Preview）、代码检查（Lint）以及错误分析等功能。最后是格式要求。在使用类似ChatGPT的聊天软件时，我们通常会按照这样的结构来定义完善的Prompt。

再看红色框标注的上下文信息部分，这是Agent与IDE结合使用大语言模型时，与直接使用聊天型大语言模型的最大差异所在。Agent的上下文信息联动性和完善性更强，比简单地将代码文件复制到输入框中，让模型生成后再复制回来的方式要完善得多。用户输入的部分就是用户的具体要求，这里不再赘述。

在Prompt设计方面，除了结构清晰之外，描述也必须清晰、精准且无歧义，不能有冲突，这样才能更好地引导模型按照预期效果执行。上下文信息要充分，但并非越多越好，关键是要精准。如果信息量过大，可能会产生噪音，影响模型的理解和生成效果。

此外，示例的作用非常重要。在与模型交互时，模型有时难以仅通过推理理解用户期望的细节，或者有些内容用自然语言难以描述清楚。就像人与人交流时，一个直观的演示（demo）往往比文字描述更有效。例如，通过视频展示Agent如何自动执行某项工作，比单纯用文字描述更容易让人理解。在Prompt设计时，还需要注意以下几点：

- **迭代优化的稳定性**：在Prompt演进过程中，要考虑到迭代优化可能带来的破坏性。可以采用A/B测试或通过大量评测集来验证效果，确保Prompt的演进是稳定的。
- **模型输出的可靠性**：在大语言模型时代，过去常说的“代码不会骗我”可能不再成立。因为模型可能会产生幻觉，导致输出错误内容。因此，我们不能像过去一样依赖单元测试，而是需要通过评测案例和实验来验证Prompt的效果。

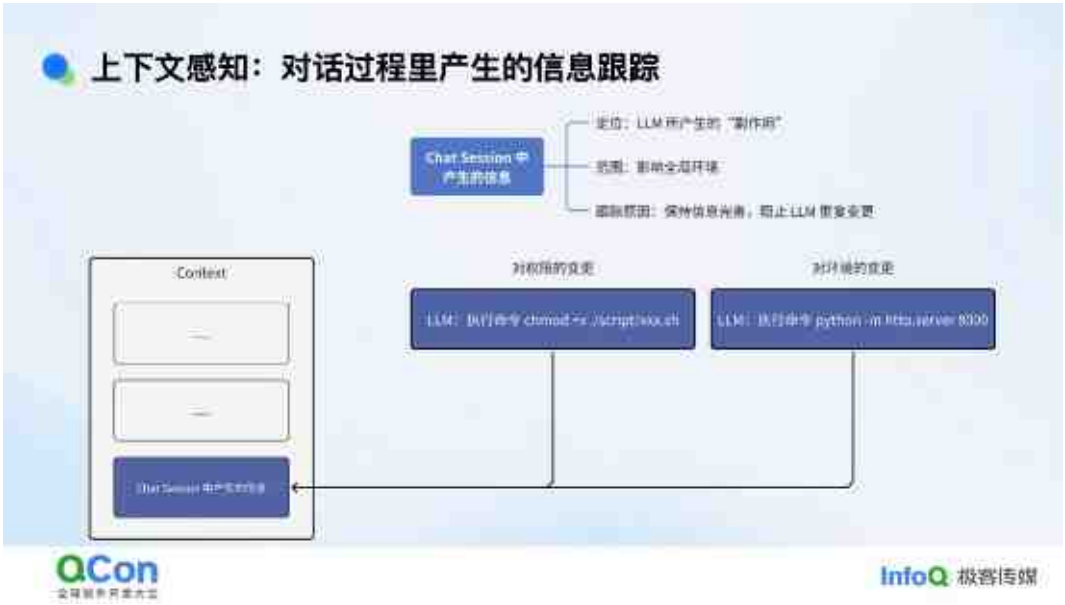
最后，成本问题也需要关注。Prompt的内容丰富且量大，而目前使用模型的成本仍然较高。因此，需要考虑如何通过优化等方式降低使用成本。

上下文感知

在拆解了Prompt的整体结构之后，我们再深入分析红色框标注的上下文信息部分。我们将进一步拆解上下文信息，看看Agent是如何与大语言模型进行交互的。在上下文拆解中，有一个关键点是：在与大语言模型的对话过程中产生的信息需要被跟踪。用通俗的话来说，这就是模型产生的数据所造成的“副作用”，它会对全局产生影响。我们需要跟踪这些信息，以保持模型输出信息的连贯性和完整性，确保后续的交互能够持续地基于之前的信息进行。这样可以避免模型重复做相同的事情。

具体案例分析

- 代码权限变更
  - 场景描述：例如，对于代码权限的变更，这会对操作系统或文件系统产生全局影响。假设文件a的权限已经通过chmod命令被修改过一次。
  - 跟踪的重要性：我们可以通过历史信息获取这些变更记录，但历史信息会持续进行压缩和摘要处理，以避免超出上下文窗口（context window）的限制。因此，这些信息非常重要，需要进行跟踪，以确保不会丢失关键信息。
- 环境变更
  - 场景描述：另一种情况是对环境的变更。例如，启动了一个后台服务，比如一个HTTP服务器正在监听某个端口并为用户提供服务。
  - 跟踪的重要性：这种全局变更也需要被跟踪，因为它们会影响整个系统的运行状态。



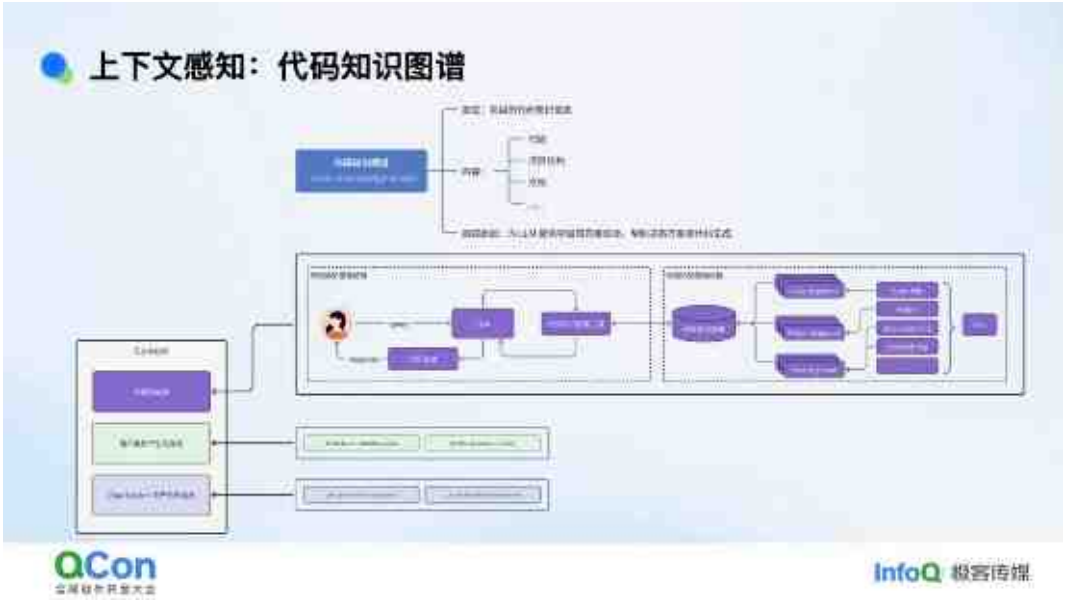
第二部分是关于人的操作信息及其产生的变更。因为Agent实际上是一种人与AI交互协作的方式，所以人类产生的信息对模型来说是非常重要的部分。

具体案例分析

- 倾向性引导
  - 场景描述：例如，使用某种实践方式（如V3实践）时，用户会在git commit中添加一些信息（message），表达自己对项目的期望。
  - 影响：这些信息反映了用户的倾向性，会对整个项目产生全局性的影响。通过这种方式，用户可以向AI模型传达自己对项目的方向、目标和期望，从而引导AI在后续的协作中更好地符合用户的意图。

第三部分是比较关键的内容，之前提到的 代码知识图谱（Code Knowledge Graph）。对于我们的代码仓库，无论是本地项目还是从Git拉取的项目，都可以统称为代码仓库。如果想让模型了解代码仓库的功能、所做的工作以及代码的组织结构，那么我们需要提前进行项目的索引构建和分析。

在下图的最右侧，可以看到代码知识图谱的构建过程。我们会分析仓库中的文件，这些文件不仅限于代码文件，例如README文件或者仓库中存放的架构描述文件等，这些文件也可以被索引和构建，因为它们所包含的知识对模型来说非常重要。同时，代码的目录结构本身也是一个关键信息。大致的分类包括代码类文件、文档类文件和README类文件。我们会根据一定的规则对这些文件进行切割，具体来说，可能会结合静态语法分析，例如按照函数维度、类（class）维度等将文件切割成片段，并将这些片段记录到索引库中。



从右侧向左侧看是消费阶段。当用户的需求输入时，大语言模型会判断这次需求是否与项目相关，或者与项目的哪些部分相关。它会提取相关的关键词或实体，检索我们已经构建好的代码知识图谱，并将检索到的信息补充到上下文中。

多轮对话与记忆管理

在分析完单轮交互的上下文信息后，我们已经清楚地了解到在单轮交互中，我们向语言模型传入了哪些信息。接下来，我们再看看在多轮交互中，这些信息是如何交互和处理的。在单轮交互中，我们已经积累了大量的上下文信息。随着交互轮次的增加，历史信息会不断累积，导致信息量线性增长。每一轮交互都会携带前面所有的历史信息，以确保信息的完整性。然而，当交互轮次达到两三轮或三四轮时，大量的历史信息可能会超出大语言模型的上下文窗口限制。那么，对于超出部分，我们应该如何处理呢？主要有以下三种处理方式。

- **粗暴截断**：第一种方式比较直接且在工程上实现较为简单。我们按照近期到远期的倒序方式，直接截断较远的历史信息，以确保不会超出上下文窗口限制，避免模型报错。这种方式虽然可以保证工程的正常运行，但有一个明显的劣势：会丢失大量的历史信息，导致模型在经过一段时间或若干轮交互后，出现类似“失忆”的状态。

- **压缩与摘要**，以保留历史信息，但可能会降低信息密度。为了尽可能避免信息密度的下降，我们可以使用大语言模型本身来进行拆解、分析和总结。这样可以更好地保持信息密度，但会带来一个问题：大语言模型的使用成本较高，尤其是在处理大量历史信息进行总结时，输入内容会更多，导致成本难以控制。虽然这种方式在效果上符合预期，但成本增加是一个需要考虑的问题。

- **工程折中方案**。我们会从大模型返回的信息以及用户输入的信息中，通过工程手段和规则策略，决定哪些信息的影响面足够小且有一定的权重，从而将其裁剪掉。例如，在执行流程中，模型会输出规划以及生成的代码。这些代码已经被应用到项目中，并重新被索引到代码知识图谱中。如果后续需要使用这些代码，我们可以通过这种方式进行补偿。因此，我们可以对产生的大量代码进行精简。这种方式虽然会丢失部分信息，但这些信息是我们经过决策后认为其丢弃的影响面足够小的，同时成本也能够得到有效控制。

Prompt Caching

关于如何降低成本，我们之前提到过大量的历史信息和Prompt文本信息。目前，大语言模型的各家厂商虽然在具体实现上存在一些细微差异，但大体逻辑是类似的，都提供了一种叫作Prompt Caching（提示词缓存）的功能。这个功能会对Prompt进行完整的前缀匹配。

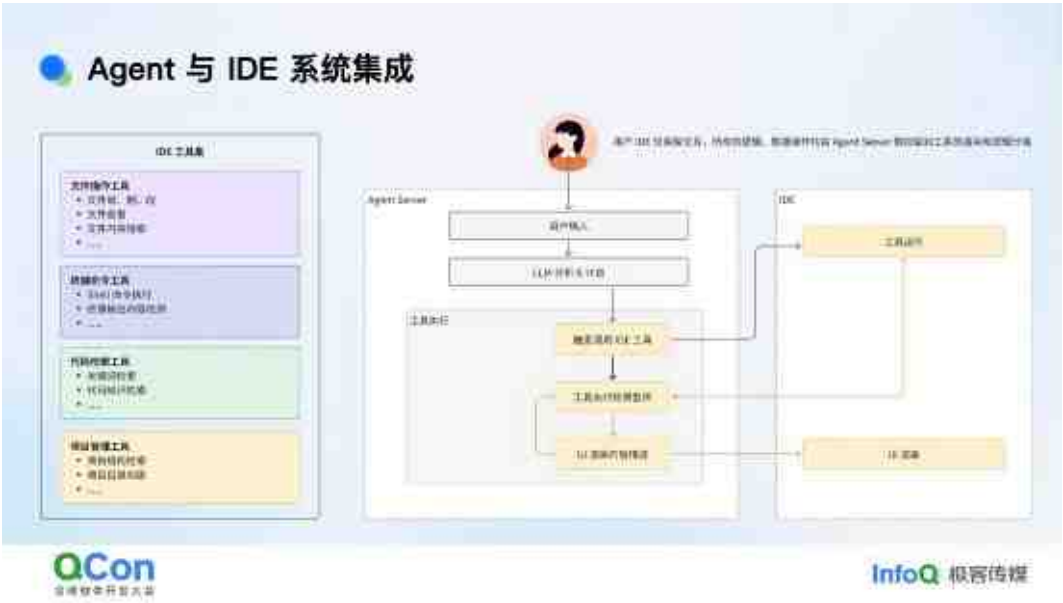
因此，在设计Prompt时，我们会尽可能将静态内容放在Prompt模板的前置部分。这样，如下图所示，在第二轮交互时，第一轮交互的内容可能会被缓存；到第三轮时，前两轮的内容也可能被缓存。不过，这张图展示的是一个极端理想的状态。我们知道，用户的输入以及当时的情况可能会导致数据不稳定，尤其是在真正超出上下文窗口时，工程上的一些裁剪操作可能会导致历史数据发生变化。

下图是一个极端理想的情况，即前置内容完全没有变更，且都是偏静态的内容，这样可以实现完全的缓存。这在设计时是需要大家仔细考虑的。目前，一般来说，启用缓存后，各家厂商的成本相比没有启用缓存时，通常可以降低到1/10的价格。这是一个我们可以操作且收益比较显著的优化点。

Agent与IDE系统集成

最后一部分我们来探讨一下IDE与Agent如何结合。IDE会提供一系列工具集，其中前三个工具尤为重要：首先是文档文件的编辑工具，包括增删改查和搜索功能；其次是终端的编辑工具，这些工具在我们人工开发过程中也经常使用；第三是检索类工具，包括关键词检索和正则表达式检索。最后，还有项目的一些元信息，例如项目所使用的技术栈，这些信息可以通过扫描获取。

请看下图右侧，工作流部分显示IDE和工具执行分为两大模块：IDE中的工具运行模块和UI渲染模块。通过这种方式，我们可以将数据和界面完全分离，所有的主逻辑都在Agent Server中控制，完全依赖数据驱动，因为与语言模型的交互主要就是数据交互。这种架构使得工具界面的渲染完全跟随数据结构，例如，如果某一天我们希望改变渲染卡片的颜色或布局，我们不需要修改Agent Server中的逻辑，因为界面渲染是基于数据结构动态生成的。



实践效果与案例分享

接下来我分享一个真实案例。我用自己写的一个小项目来说明，虽然它叫贪吃蛇，但可能和大家通常理解的版本不太一样。我们特意在这个项目中设计了一些测试场景，以验证场景的复杂度。

场景一：从0到1的创建

我们首先要求Agent帮助创建一个贪吃蛇游戏。这个要求非常简单，只是一条语句。Agent帮我们实现了游戏的基本逻辑，使用JavaScript编写了游戏代码，并通过HTML进行渲染。同时，Agent还启动了一个静态服务器来托管这个HTML文件，服务器是用Python实现的。

随后，我们进一步要求使用Vite+Vue技术栈来重构项目。Agent能够很好地完成这个任务。它首先分析了当前项目的目录结构，然后重新初始化项目，使用Vite的方式重新构建，并安装了必要的依赖。接着，Agent修改了代码，完成了项目的重构。

场景二：在已有项目中添加新功能

第二个典型场景是在已有项目中添加新功能。我们要求给贪吃蛇游戏增加一个爆炸粒子特效。对于前端开发人员来说，实现这样的特效可能需要花费一些时间，但Agent却能非常熟练地完成。它在短短两三分钟内修改了所有代码，并且完成了运行调试。

接着，我们又提出了一个更高级的需求：增加一个道具系统。道具系统涉及更多维度的碰撞检测和更复杂的倒计时逻辑。Agent也很好地实现了这些功能，展现了其强大的能力。

场景三：修复错误

第三个场景展示了Agent并非总是完美无缺。在实际案例中，Agent生成的代码确实出现了错误。我们把错误反馈给Agent，并要求它修复。右侧图中展示的是在终端中启动构建时的错误日志。我们将这些日志提供给Agent，它成功地修复了所有错误。

通过直观地观察Agent帮助我们创建的贪吃蛇游戏，我们发现道具系统和粒子碰撞检测都实现得相当不错。如果人工实现这样一个小游戏，可能需要花费半天甚至一天的时间，但Agent或结合大语言模型，可能只需要半小时到一小时，再加上一些细节调整。

以我自己为例，我完全不懂游戏开发，包括碰撞检测的具体实现。虽然贪吃蛇游戏本身很小，但其游戏逻辑、每一帧图像的绘制等，对我来说都是陌生的领域。然而，大语言模型帮助我们跨越了这些障碍，即使是不太了解相关领域的研发工程师，也能借助它开发出这样的小游戏。

经验总结

模型生成内容的不确定性

在工程过程中，模型生成内容的不确定性是一个关键问题，主要分为两种情况：

- **内容格式不符合要求**

有时模型输出的内容格式可能不符合我们的需求，导致工程与模型的衔接出现问题。例如，一些模型支持function call功能，直接以JSON或YAML格式输出。这种情况下，模型厂商通常会负责后处理。但如果模型不支持function call，我们就需要回到原始的Prompt引导方式，明确告诉模型应该遵循什么样的JSON schema或YAML schema来输出内容。然而，模型仍然可能输出错误的格式，尤其是JSON格式，容易出现括号、引号、冒号等格式异常。从工程角度来看，我们需要确保即使格式异常，也能正常执行工作流程，而不是直接中断Agent的工作流程。为此，我们可以实现一个类似于修复JSON的工具包，例如在GitHub上可以找到许多类似repair json或fix json的库。这些库相对较新，说明这是一个普遍存在的问题，大家都在努力解决。

- **内容输出的不稳定性**

即使模型输出的格式正确，其内容也可能存在不确定性。例如，要求模型实现一个贪吃蛇游戏，第一次可能采用单文件方式，第二次可能使用Vite+Vue，第三次可能生成Vite+React。由于Agent是为研发领域构建的，我们作为研发领域的工程师或专家，应该提供更明确的技术指引，例如在实现Web类游戏时，应该选择什么样的技术栈。这样可以确保模型的输出更加稳定，避免出现一会儿像专业工程师，一会儿又像入门小白的情况。

模型服务的稳定性和兼容问题

模型服务的稳定性也是一个重要问题。目前，模型供应商众多，模型种类繁多，我们在IDE中为用户提供了选择不同模型的能力。然而，不同模型之间的稳定性以及不同供应商之间的稳定性差异较大。在大语言模型时代，尤其是在资源紧张的情况下，模型的成功率远未达到过去所说的99.9%、99.99%的水平，能够达到99%已经相当不错。因此，我们需要关注如何实现灵活的负载均衡，并建立统一的模型接入层，以提高系统的健壮性和稳定性。只有这样，大语言模型在工程上的落地才是可行的，而不仅仅是一个玩具或演示项目。

Prompt调试

Prompt的设计和调试是一个需要特别关注的环节。由于模型输出存在不确定性，我们需要通过大量的数据积累或A/B实验来验证Prompt的效果。此外，不同模型的训练数据和特性不同，很难设计出一个适用于所有模型的通用Prompt。尤其是在为用户提供多种模型选择时，我们需要考虑如何让Prompt更好地贴合每个模型的特性及其训练数据，从而获得更好的效果。在实际应用中，可能只需要调整一两个词或稍微改变Prompt的组织结构，就能在不同模型上取得更好的效果。

未来展望

我大致从四个方向来思考未来的发展。

认知增强

目前的Agent仍处于早期形态，可以将其类比为一个实习生。我们发现这个“实习生”在某些场景下可能无法做到完全符合预期（比如达到80分），当它无法完成任务时，我们可能需要人工接手。但我们的期望是，Agent的认知和领域知识能够不断增强，甚至超越人类。这里的领域知识不仅仅是知识的积累，而是能够将这些知识组合起来，形成某一领域的最佳实践。当Agent能够做到这一点时，我们就会感受到它的成长，从一个初级开发者逐渐变成一个有1到3年经验的开发者，甚至最终成为一个资深开发者。

目前，多模态已经在一些场景中得到应用，例如Trae IDE中的Agent也支持多模态，但更多地是处理图片类内容，这对于前端或客户端开发者较为友好，比如可以通过设计稿截图来还原内容。然而，我所期望的多模态不仅仅是图片，还包括PPT、音频、视频，甚至是结构化的数据（尽管它们仍然是文本文件，但更偏向于结构化数据）。如果Agent能够更好地结合和理解这些不同类型的内容，其效果将会更好。此外，深度推理能力也是未来的一个发展方向，我们期望这些能力能够落地到Agent的工程实践中，并最终体现在产品形态上，为用户提供更好的服务。

工具整合与扩展

目前，IDE为Agent提供了一些默认工具。随着MCP协议的逐步推广，各家厂商也开始支持这一协议，这为工具的扩展提供了更多可能性。进一步来说，在人与AI的交互过程中，是否可以自定义一些工具，甚至与物理世界进行交互？这种物理世界的交互可能涉及物联网（IoT），甚至人形机器人等，这将是未来Agent需要探索的一个方向。

集体智能与多Agent协作

集体智能更多地体现在协作上，即多Agent的协作。在这种情况下，Agent的角色和领域将更加专业化。目前，我们期望一个Agent能够胜任所有领域的工作，例如在编程领域，既能做前端，又能做后端，还能做客户端和嵌入式开发，但现阶段这并不现实。

因此，未来可能更多地会按领域进行专业化分工，就像工程师有各自的专业领域一样。在这种情况下，我们需要更多地考虑多Agent协作的方式。最近，Google发布了Agent-to-Agent协议，这表明业界也在探索这一技术领域的演进。

自主性提升

这是一个更长远的方向，期望Agent能够自主规划以解决复杂问题。目前，一些产品（如Devin）已经聚焦于解决复杂任务和长耗时问题。此外，Agent如果具备了自主决策和解决问题的能力，是否也能自己构建缺失的工具？例如，IDE提供的工具可能不够用，MCP协议提供的工具也可能不足，Agent是否能够自己创造工具并持续自我完善？这可能标志着AI的自我滚动发展，人类参与的比例可能会进一步降低。

此外，还有一个深度研究的方向是，目前大语言模型对用户需求的响应主要基于其训练数据，或者我们提供的辅助工具（如互联网搜索补充的信息）。如果上下文中没有提供相应的指导，同时其训练数据中也没有相关内容，那么它如何解决这种未知问题，更好地应对复杂问题，这也是未来需要探索的方向。

嘉宾介绍

- **段潇涵**，字节跳动豆包MarsCode/Trae IDE架构师，致力于大语言模型在研发工程领域的落地实践，构建基于大语言模型的AI编码助手；同时致力于推进IDE云端化进程，与研发基础设施平台融合，为开发者提供一站式的智能研发体验。

打破AI辅助开发碎片化困境，阿里巴巴R2C Agent的AI编程实践

作者 付永生，阿里巴巴/高级技术专家
审核 罗燕珊

AI编码不是一个新话题，业界已有诸多实践，如外部的cursor、GitHub Copilot、Bolt.new、Cline，以及内部的Oneday、Weavefox等等一系列的平台或工具，都提供了AI辅助编码的能力，但这些平台工具与现有的生产链路如何结合还存在极大的挑战：一是没办法很好的和现有研发流程进行集成，二是协作过程碎片化，有一定门槛。

本文整理自阿里巴巴高级技术专家付永生6月份在AICon 2025北京站的分享**《R2C Agent打破AI辅助开发碎片化困境》**。本次演讲从阿里业务研发场景出发，并结合真实的业务应用，阐述R2C（requirement 2 code）Agent如何从“知识库+钉钉文档+设计稿”驱动整个研发链路，带来系统性的效率提升，并分享其中的最佳实践。

以下是演讲实录（经InfoQ进行不改变原意的编辑整理）。

随着人工智能技术的兴起，市面上涌现出许多面向编程的工具和平台，但这些工具的实际效果参差不齐，真假难辨。唯有亲身实践、深入其中，才能真正积累经验，获得切实的体验。在过去的一年多里，我们团队在面向客户需求的AI解决方案方面积累了丰富的经验。然而，针对编程这一我们自身工作的核心领域，却始终没有找到令人满意的现成方案。因此，我们决定自行探索，提出并实践了一套适合自身需求的解决方案，以更好地利用AI技术提升工作效率。

我希望与大家分享我们在这个过程中所积累的经验和教训。作为基础设施的使用者，我们将探讨如何更有效地使用这些工具，如何判断使用效果，以及在实践过程中遇到的挑战与反思。希望通过我们的分享，能够为大家提供一些有价值的参考和启发。

AI编程赛道火爆背后

我想分享一下我对当前AI编程领域持续火爆的理解。在我看来，这个领域之所以如此火热，本质上是因为有大量资本愿意投入其中。资本的大规模投入必然有其内在逻辑，这一点从公开报道中也能得到印证。我思考了很长时间，认为这一切的源头在于大模型技术的兴起。虽然大模型提出了一种全新的可能性，但长期以来，人们并未找到理想的解决方案。直到去年Claude 3.5的出现，才从底层能力上带来了突破性的进展，让人们看到了新的范式就在眼前。因此，我们可以看到包括Cursor在内的许多工具都在那个阶段获得了全新的能力，而这些能力的提升都源于大模型的进步。

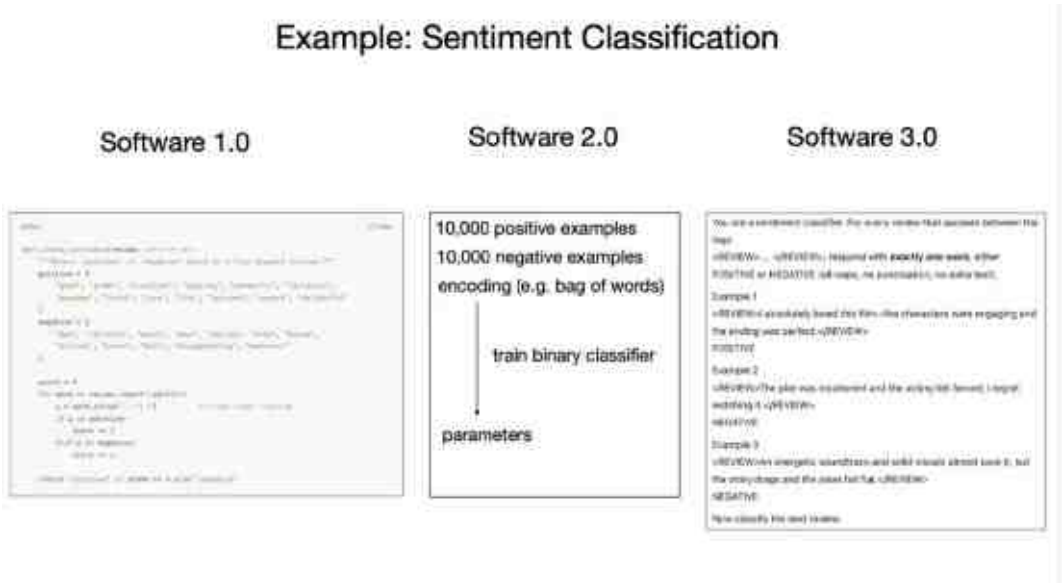
海外独角兽	最新融资	估值	年化收入（ARR）	成立时间
Anysphere	2025年6月官方宣布融资9亿美元，总融资额超过 10 亿美元。	估值约99亿美元	超过 5 亿美元	2022年
Windsurf	2024 年 8 月，完成 1.5 亿美元 C 轮融资，投后估值达 12.5 亿美元，总融资额 2亿美元。4 月中旬，OpenAI 被曝将以约 30 亿美元收购Windsurf，交易还未对外公告。	按收购估值约 30 亿美元	超过 1 亿美元	2021 年
Lovable	2025年6月，有消息称其计划募集至少1亿美元，谈判还处早期阶段。此前累计融资2250万美元。	本轮估值15亿美元或更高	超过5000万美元	2023年
Poolside	2024年10月完成5亿美元融资。总融资近6.3亿美元。	估值30亿美元	去年收入或不足1000万美元	2023年
Cognition	2025年3月完成数亿美元的A轮融资，累计融资超过3亿美元。	估值近40亿美元	不足 50 万美元	2023年
Magic	2024年宣布获得3.2亿美元融资。累计融资 4.65 亿美元。	估值未透露，预计超过此前传言的15亿美元	未透露	2022年
Replit	2025年4月传出Replit 正洽谈新一轮融资，预计将募集约 2 亿美元，融资谈判仍在进行中。此前Replit累计融资约4.4亿美元。	融资后估值有望至 30 亿美元	未透露	2016年

第一财经根据官方公告及公开信息整理

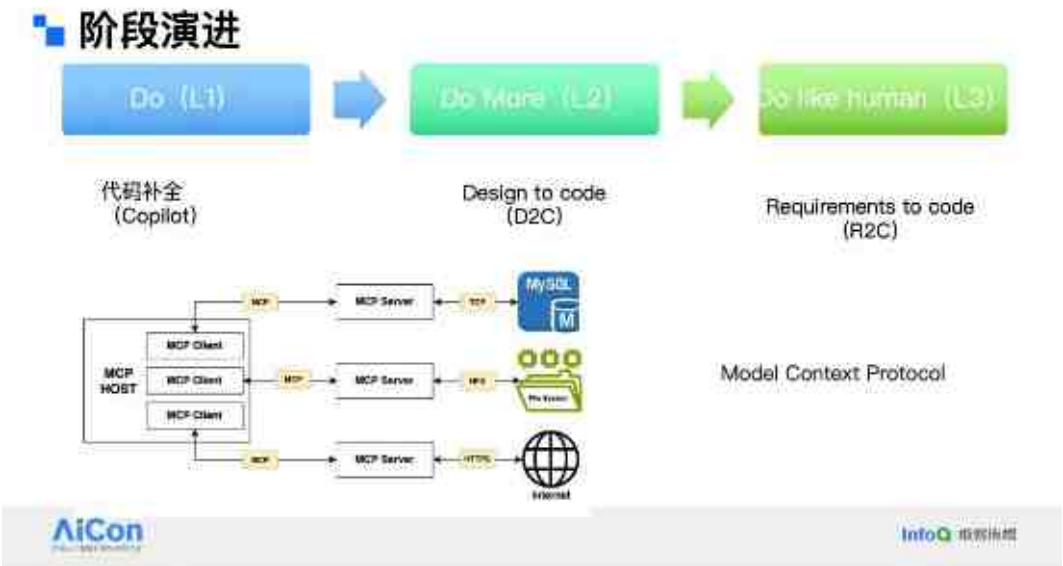
我们所做的事情应当避免那些容易被颠覆的方向。举个例子，如果你花费一年时间开发一个平台或产品，但突然出现一个突破性的大模型，那么你之前的很多努力可能会被颠覆。我并不是说这些努力没有价值，而是强调这个领域的发展速度非常快。因此，我们必须明确自身的边界和定位。在听取其他平台的分享时，我会带着这样的逻辑去理解：虽然你目前的进步可能是1%或5%，看起来也很显著，但你的定位是否准确？要知道，大模型的提升可不是1%或5%，而是可能达到500%。为什么演示时看似完美无缺，实际落地却表现不佳？因为在真实场景中，没有“重来一次”的机会，也没有人能替你兜底。

在我看来，编程场景天然适合AI的应用。全球至少有4000万的高频用户，他们每天依赖编程工作。如果你能覆盖其中10%的用户，那就是400万的真实活跃用户。他们每天通过token消费，这个市场规模是非常可观的。因此，这些判断构成了整个行业持续投入的最大驱动力。

最近，Andrej Karpathy分享了他对软件定义的看法，但我认为每个人的理解都不尽相同。我们可以听取他的观点，但不必完全受其影响。他的思考确实引发了行业对AI编程未来演进方向的深入思考，但他的判断是否准确，还需要持续观察。



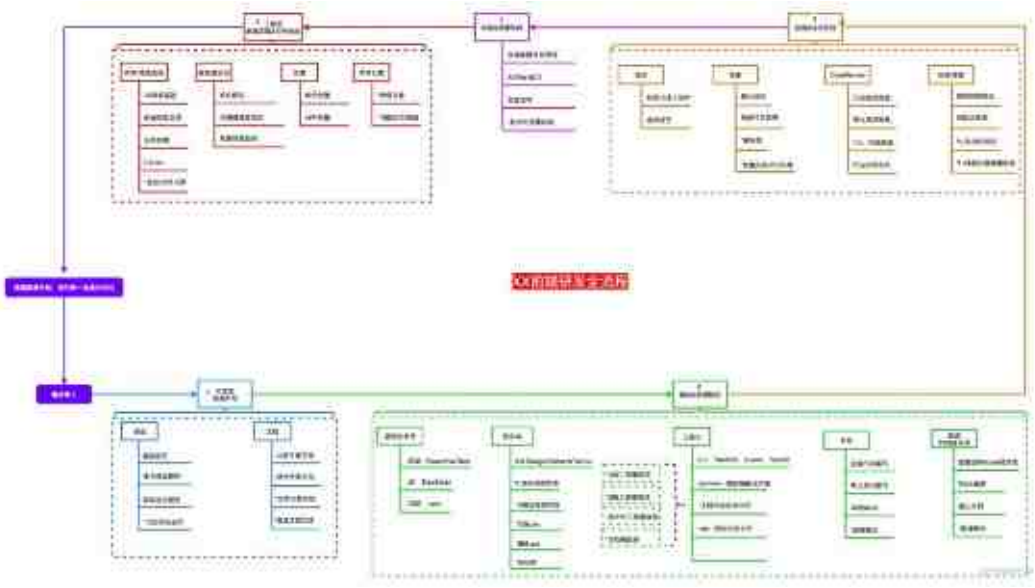
我们在过去一年多时间里，一直在思考和解决这个命题。这个过程可以大致分为几个阶段：最初，代码补全是最常用的功能；随后，出现了将设计稿直接转换为前端代码的功能；我们真正想要实现的，是第三阶段：无论需求是什么，也无论想法如何变化，我都希望大模型能够按照我的设想，将完整的功能开发出来，并推动项目进展到预期阶段。这也是我们探索的起点。



工程师AI编程“困境”

工程师的日常工作其实是高度碎片化的。作为真正使用AI工具进行开发的用户，我们一线工程师实际用于编码的时间，据我估算，可能只占全部工作时间的30%到40%。除了编写代码，我们还需要理解现有代码、编写单元测试、设计新功能等等，这些工作往往并不为外人所见。此外，一线开发工作并非单打独斗，任何一个项目通常都需要前端、后端、设计师、产品经理等多个角色共同参与。在这种情况下，如果只是单点地引入AI工具，往往难以取得理想的效果。

我们希望从端到端的角度来理解和解决整个研发流程的问题。**我们的目标是，无论使用哪个平台，都能够实现对整个研发过程的端到端支持**。当涉及多个角色时，必然会形成完整的流程。下图是一位资深前端开发者在GitHub上分享的工作流程，这个过程就像一条流水线，非常漫长。如果AI只参与其中某一个环节，即使在这个环节表现出色，一旦放到整个流程中，其带来的收益也可能被大大稀释。换句话说，它可能只解决了局部问题，而无法解决全局性问题。



当然，这是一个必经的过程。目前的开发平台和AI工具所处的阶段，很像十多年前互联网刚刚兴起时的基础设施不完善时期。在这个阶段，必然会催生出新的基础设施。然而，就目前而言，整体效果仍然参差不齐。因此，AI工具的用户或团队必须深入其中，

才能找到最适合自身团队的问题点和解决方案。

R2C解决方案新思路

我们提出的新思路是端到端的解决方案。无论是用户提出的需求，还是我们自己想要实现的功能，无论项目规模大小，我们都希望找到一种更高效的方式来完成。传统的辅助代码生成、一次性代码生成等工具仍然会在过程中发挥作用，但我们更关注的是如何构建一个完整的流程。

我们希望将任何需求或想法转化为一种描述性的表达，通过文档或其他载体清晰地呈现出来。我们的AI提案正是为了实现这种端到端的开发流程而设计的。在最终的开发阶段，AI将在代码审核和集成过程中发挥重要的协同作用。

我们不仅提出理念，更注重将其付诸实践。因此，我们更加关注如何真正理解并落地这种端到端的解决方案。我们希望它成为一个整体性的方案，无论使用何种开发工具，都能贯穿需求分析、设计、编码、测试等整个流程。AI能够渗透到哪个环节，最终取决于整个方案所能带来的实际效果。

在流程串联中，领域知识库的构建也至关重要。每个开发团队所面对的内容、需求和产品，都有其独特的领域积累。我们还需要实现自动化的流程衔接，并确保用户体验的一致性。事实上，一致性体验是我们最初也是最重要的追求。考虑到每位开发人员可能使用不同的IDE，在编译过程中可能需要复制代码，以及前端、后端、测试等环节所使用的平台各不相同，我们希望通过统一的解决方案，为所有参与者提供一致且流畅的开发体验。

框架、实施和进展

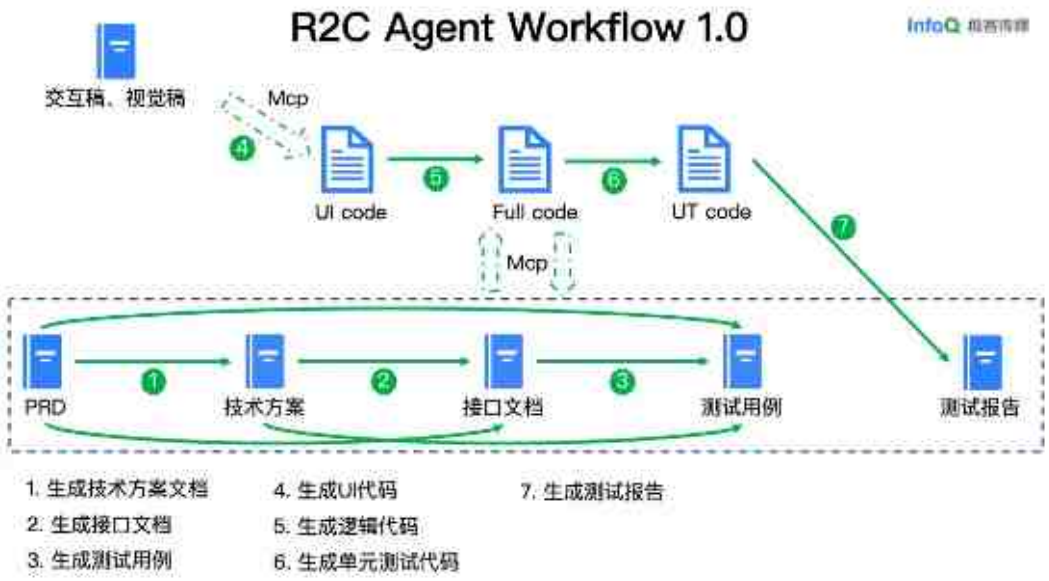
在实施过程中，我们逐渐形成了一套清晰的框架，也遇到了一些值得分享的问题。我们内部称之为“R2C Agent”，它代表了我们目前对这件事的整体理解。简单来说，这个框架的核心在于将各类输入统一到一个中间层，包括MRD、PRD、技术方案、接口测试用例、视觉稿、交互稿等。这些输入的形式，无论是标准化文档还是视觉理解，本质上都反映了我们团队或所在领域对项目的规范和理解。

我们将整个端到端的解决方案抽象为两个关键要素：提示词和上下文。中间层的构建，正是为了确保在这两个方面保持一致性和可理解性，从而让整个流程更加顺畅。我们主要关注四个方面：领域知识库、项目文档、设计稿，以及它们的结构化表达。这些文档可以是结构化的，也可以是非结构化的，但必须是AI易于理解的。

在最初“手搓”阶段，我们并没有完善的基础设施支持，因此我们选择在VSCode中开发了一个插件形式的Agent。我们的需求文档通常存储在钉钉文档、语雀或Word中，Agent通过类似RPA的方式调用浏览器读取这些文档内容，作为整个集成链路的起点。这种方式也解决了权限和内容可读性的问题，只要开发者有权限，Agent就能访问相应文档。未来，当基础设施更加完善后，这一过程可以进一步简化。

目前，我们的核心输入主要包括需求文档和设计稿。设计稿可以是设计师手工完成的，也可以由AI生成，只要具备足够的确定性即可。沿着这个思路，我们将设计稿转化为设计元素的语义表达，这种表达方式已经相对成熟，能够被大模型较好地理解。

我们在5月中旬上线了1.0版本，按照上述思路实现了从交互稿、视觉稿到技术方案、接口文档、测试用例的完整流程。前端代码的还原度已经非常高，不仅包括视觉还原，还包括基于接口和技术文档的集成工作。前端工程师的工作几乎只剩下“填空”，整体流程已经具备较高的实用性。



在实践过程中我们也经历了一些“翻车”时刻。最初我们尝试完全依赖AI生成技术文档，但效果并不理想。后来我们意识到，如果由工程师自己将需求转化为AI友好的技术文档，整体效率和准确率会显著提升。也就是说，无论是前端、后端还是测试工程师，只要你能清晰地描述项目需求，AI就能更准确地生成所需内容。我们验证发现，这种方式不仅适用于单个环节，也能显著提升整个流程的效果。

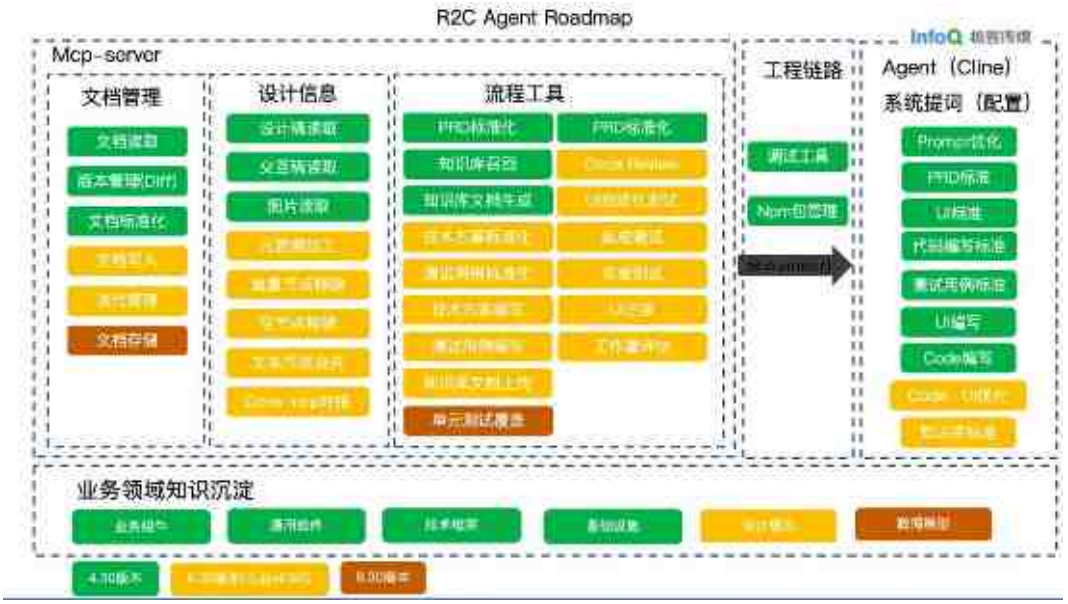
在实现层面，我们采用了MCP服务，并发布了自己的MCP服务以提供整体能力。通过定义workflow，我们将需求文档、接口文档、技术文档、业务知识库和视觉库等输入组织起来，形成可迭代的开发流程。这种方式既能利用现有工具的能力，也能随着大模型的进步持续优化。关键在于如何规范需求、接口和技术文档的描述。根据我们的经验，只要需求文档写得足够清晰，达到评审时可被团队理解的程度，大模型就能较好地处理。

在定制过程中，我们重点关注三个方面。首先是上下文窗口管理。我们不希望一次性将所有信息都输入给大模型，而是只提供项目所需的关键内容，以避免信息过载和生成无关内容。其次实现主任务、子任务独立运行。最后是主任务负责管理、子任务负责实施。我们采用类似多Agent的思路，但在实际项目中，一个workflow可能会涉及大量上下文，因此我们更倾向于将任务拆分为子任务，由一个主Agent统一协调执行。这样做的好处是，每个子任务都是确定性的，不需要Agent进行额外规划，从而避免失控。我们认为，Agent并非万能，对于确定性任务，明确指令比自由发挥更有效。

我们在内部多个项目中验证了这一方案的有效性。在前端开发中，代码采纳率和覆盖率已经达到较高水平，甚至包括常规业务逻辑的处理也表现出色。整体开发效率和成本都有显著改善。需求一旦明确，当天晚上就能看到初步效果，大大缩短了沟通周期。虽然这还不是完整的端到端流程，但已经展现出明显的效率提升。

关于后端代码生成，很多人持怀疑态度，认为AI难以胜任。但实际上，问题不在于AI的能力，而在于我们对其缺乏信心。我们验证发现，AI在这些场景中并不存在本质障碍。编程是一个确定性过程，AI生成的代码虽然不是唯一解，但可以从整体角度优化系统效果，往往比局部优化更具价值。以服务端开发为例，只要明确接口定义和领域模型，AI就能生成高质量代码。我们并不追求一步到位，而是希望AI生成的代码能大幅减少人工工作量，让工程师专注于“填空”和优化。

在工程实施方面，最关键的是文档管理。所有输入都依赖于文档，因此如何组织和管理文档，将直接影响整个链路的实现效果。每个团队都有自己的编程规范和需求文档格式，这些都需要文档化。如果你已经有一个运行多年的系统，也可以让AI理解现有代码结构，并生成相应的文档，反哺知识库。从设计信息和文档信息入手，是构建高效AI开发流程的关键起点。



AI DEV vs AI CODING

在完整的开发链路中，时间的分布其实是高度碎片化的，每个环节都紧密相连。AI并不会颠覆这一流程，因为软件开发本身的流程是合理的，AI的作用在于加快进度、提升效率，并改变某些环节的执行方式。真正的开发过程涉及多个阶段，包括系统集成和测试等。目前市面上所谓的AI编程工具或脚手架，只有与现有的工作流程结合，才能发挥最大效用，否则只会加剧碎片化。

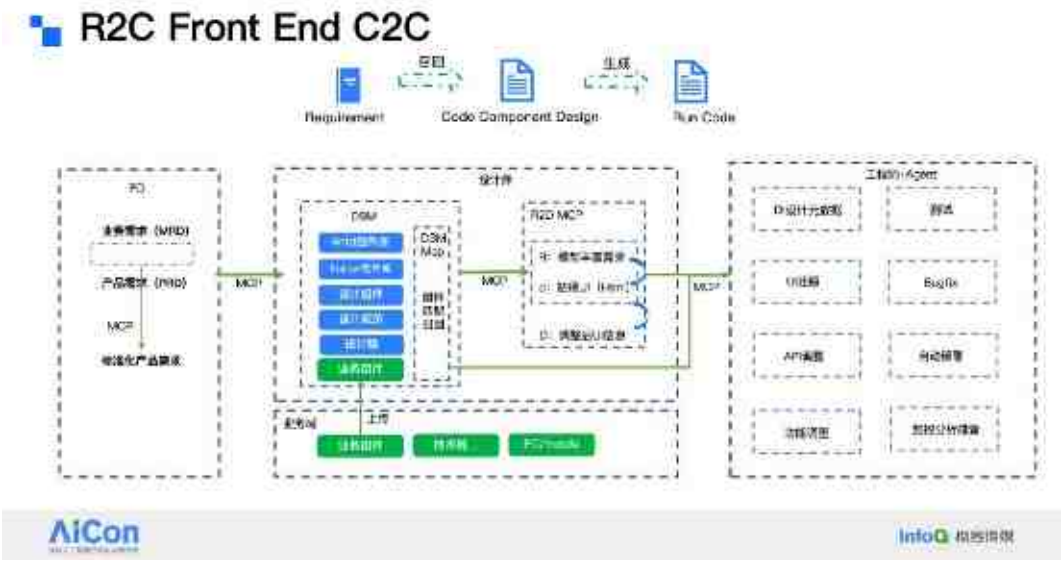
碎片化带来的唯一好处，是可能让团队中的一部分人脱颖而出，使用AI更熟练的人表现会更出色。但这种个体的优秀并不会带来团队整体质的飞跃，团队层面的提升需要更系统的理解和方法。

系统工程的核心始终是追求效率和质量，沿着这个方向推进是没有问题的。关键在于人的协作如何发挥作用。最值得分享的是，AI在理解和响应人类输入方面表现出色。我们围绕文档和流程所做的工作，正是基于这一点。无论处于哪个环节，只要你的工作对AI友好，就能在端到端的集成方案中发挥作用。人在整个过程中扮演的角色，是将现实世界的理解转化为AI能够更好理解的形式。

未来展望

下图展示的是我们前端开发中的一个关键环节，称为C2C（Component to Code，组件到代码）。在前端开发中，我们会积累大量组件和能力，并将需求拆解为多个模块。最核心的优势在于，凡是曾经写过的代码和组件，都可以被高效、准确地复用。从我们目前的前端实践来看，大约60%到70%的开发工作已经能够得到高质量的保障。

虽然行业内已有不少类似的做法，但据我观察，目前还没有哪个平台能够真正做到“只需提出需求，其余全部自动化完成”。要想真正享受到AI带来的成果，无论是个人还是团队，都必须亲自下场实践，亲手去用、去感受。这也是我们持续推动最佳实践的原因——只有在实际使用过程中，才能不断加深理解，持续优化流程。



我们始终在推进的目标，是在端到端的交付流程中不断提升AI的渗透率。我们设定了50%的目标，但很快就会突破这个数字。目前，在R2C的各个环节中，无论是前端、后端还是测试，AI生成内容的采纳率已经超过50%。我们的思路是将开发工作逐步转化为“填空题”——AI完成大部分内容，开发者只需补充关键部分。一旦进入这种迭代模式，整个流程会不断收敛，效率也会持续提升。

R2C Agent：生产交付

重构开发体验：CodeFuse智能代码助手的设计与实践

演讲嘉宾 牛俊龙 编辑 Kitty

在当今的研发领域，人工智能正迅速地在代码补全、对话系统、文本到代码转换等多个应用场景中重新定义我们的工作方式。在InfoQ举办的QCon全球软件开发大会（北京站）上，蚂蚁集团技术专家牛俊龙分享了“智能代码助手CodeFuse的架构设计与实践”，他介绍了本地核心服务的设计策略，通过实际案例展示如何结合本地与远程数据资源，进一步优化智能代码助手的AI能力。此外，他还分享了产品在落地过程中的宝贵经验和教训，并展望CodeFuse的未来发展路径。

内容亮点

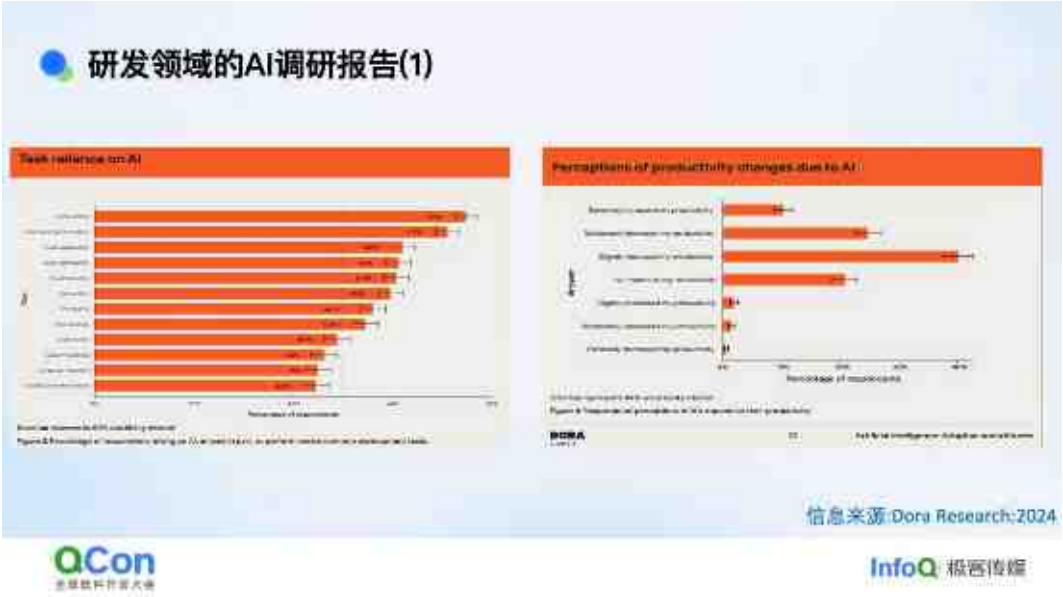
- 解密CodeFuse智能代码助手的设计理念与构建策略
- 详细剖析如何在多元化业务场景中实现高效数据处理的技术方案

以下是演讲实录（经InfoQ进行不改变原意的编辑整理）。

AI在研发领域带来的变化

近期，我关注了国内外一些数据报告，其中两份报告引起了我的注意。第一份是《2024年中国AI大模型产业发展与应用研究报告》。该报告数据丰富，从中可以提炼出两个关键信息。首先，从模型数量来看，近一两年发展迅猛，从2023年到2024年，模型数量增长了约200%。其次，从价格方面来看，百万token的价格从2023年的300元降低到2024年的1.5元，价格更加趋于合理化，使得更多人能够使用。报告还提到，从发布和备案的大模型中，垂直领域的大模型按行业分类，排名前五的是互联网、金融、医疗、教育和政务，其中互联网行业位居第一。这一数据表明，在互联网行业更适合大模型的落地应用。

另一份国外报告的数据来源于Dora Research 2024。Dora是一个在研发效能领域具有影响力的研究团队。该报告提到，他们调研了39,000多名不同级别的开发者，基于AI的任务进行了排序，其中排名第一的是代码编写，其次是理解文档、代码解释等。在生产力方面，75%的受访者表示AI能够提高他们的生产力，超过三分之一的人表示生产力提高了50%以上。从这些数据可以看出，AI已不再是一个高不可攀的高科技产品，而是离我们越来越近，甚至在研发领域已成为不可或缺的工具。



接下来，我想介绍一下CodeFuse的落地现状。从产品矩阵来看，我们为开发者提供了CodeFuse智能助手，这是一款常见的IDE插件。此外，我们还为研发团队提供了CodeFuse研发助手的构建平台，并且由于其部署在内部，我们还为其他工具平台提供了CodeFuse API，以支持其他平台的AI化和AI能力。因为主要面向开发者，CodeFuse在开发的各个阶段都能提供多种能力，例如仓库问答、代码补全等。

近期，CodeFuse也经历了一些新的变化。我整理了其中最主要的三个变化。首先是实时代码修改和预测功能，补全能力是抓住用户核心需求的关键。如果补全能力不佳，就难以吸引用户。实时代码修改和预测功能实际上是对现有补全能力的升级，它不仅支持在光标之后新增代码，还能支持光标前后一段代码的新增、修改和删除。

第二个新能力是AI Partner，它被定位为智能编程搭档，能够根据需求描述完成编码任务，支持生成和修改多个文件。与补全功能不同，补全是在后台默默辅助用户，而AI Partner需要用户主动与其对话。例如，用户在编写代码时，可以通过右侧的自然语言描述，让CodeFuse插件生成相关代码。

最后一个新能力是Text to Code。这一能力是基于用户在开发过程中提出的需求，能够一键改写指定代码，支持重构、简化和注释等功能。它与AI Partner也不太一样，因为AI Partner的焦点集中在编辑器右侧的对话框，可能会干扰用户的编程思路。为了解决这

一问题，我们提供了Text to Code功能，用户在编写代码时无需转移视线，只需选中需要修改的代码，并在上方添加自然语言描述，即可根据需求完成代码的生成或修改。

CodeFuse插件技术架构

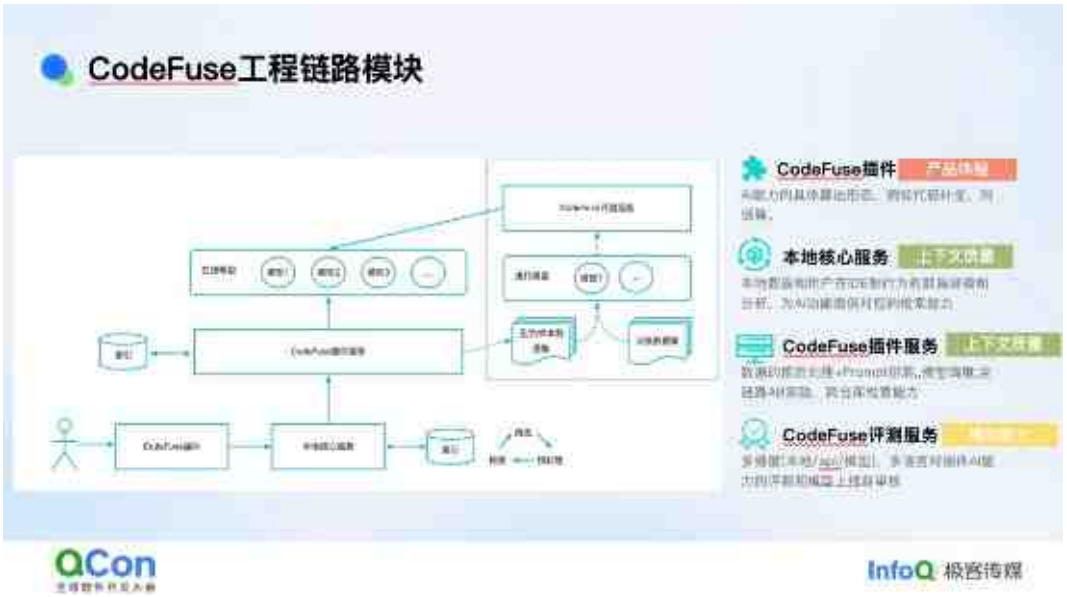
在深入介绍CodeFuse技术架构之前，我想先与大家探讨一下我们在设计和开发过程中所秉持的理念。因为正是这些理念，为我们后续的系统设计和开发提供了明确的方向和目标。

首先，我们思考的是CodeFuse插件的定位。它是一款AI产品，而AI产品本质上首先是产品，因此产品体验至关重要。如果界面设计不佳，或者功能设计违背用户习惯，那么这款产品肯定无法吸引用户。而作为一款AI产品，其核心竞争力在于AI能力。我们坚信，AI能力越强大，就越能吸引用户。那么，如何提升AI能力呢？我们从两个方面入手：一是模型能力，二是上下文质量。

模型能力是大家最为熟悉的，它决定了AI插件的智能水平和功能的强大程度。然而，仅有强大的模型是不够的。如果输入的上下文质量不高，生成的结果也难以令人满意。因此，上下文质量同样是影响AI能力的关键因素。高质量的上下文能够帮助模型更好地理解用户需求，从而生成更符合预期的结果。

基于以上思考，我们在设计CodeFuse插件时，将其整体划分为四个模块。第一个模块是CodeFuse插件本身。这是用户最直接接触的部分，具备AI补全、对话等功能。第二个模块是本地核心服务，它是一个可执行文件。在Mac或Linux系统上，它是一个二进制文件；在Windows系统上，则是一个可执行的.exe文件。这个服务的主要职责是提升上下文质量。它会分析本地仓库的数据以及用户在IDE中的行为习惯，为AI能力提供检索能力。这些检索到的数据会被整合到当前AI请求的上下文中。为了实现这一功能，背后有一个索引库，它会实时更新，以确保数据的准确性和时效性。

当一个AI请求经过插件和本地核心服务后，并不会直接发送到模型，而是会发送到远程的服务端。服务端会对接收到的AI请求进行数据的前后处理，包括数据的拼装和模型的调度。同时，服务端还会进行各种A/B实验。服务端背后也有一个索引，但与本地索引不同的是，本地索引专注于当前仓库的数据，而远程索引则用于跨仓库检索。例如，当用户新建一个应用并提出需求时，如果本地仓库中没有相关数据可供检索，服务端会查找与当前仓库主题相关的其他应用中的数据，作为参考并整合到上下文中。



在模型进行推理的同时，插件服务端会对每一个AI请求进行大量埋点。这些埋点数据会在后台进行离线计算，分析出正样本和负样本。这些样本会与内部数据结合，用于模型的迭代和训练，以推动模型的进一步发展。新训练出的模型并不会直接上线，而是会经过CodeFuse的评测服务。评测服务的主要目的是评估模型能力，为当前的AI功能打

分。例如，对于补全功能，评测服务会使用从线上梳理出的评测集进行评估，只有当分数超过60分时，模型才被允许上线。低于60分的模型则需要继续训练。

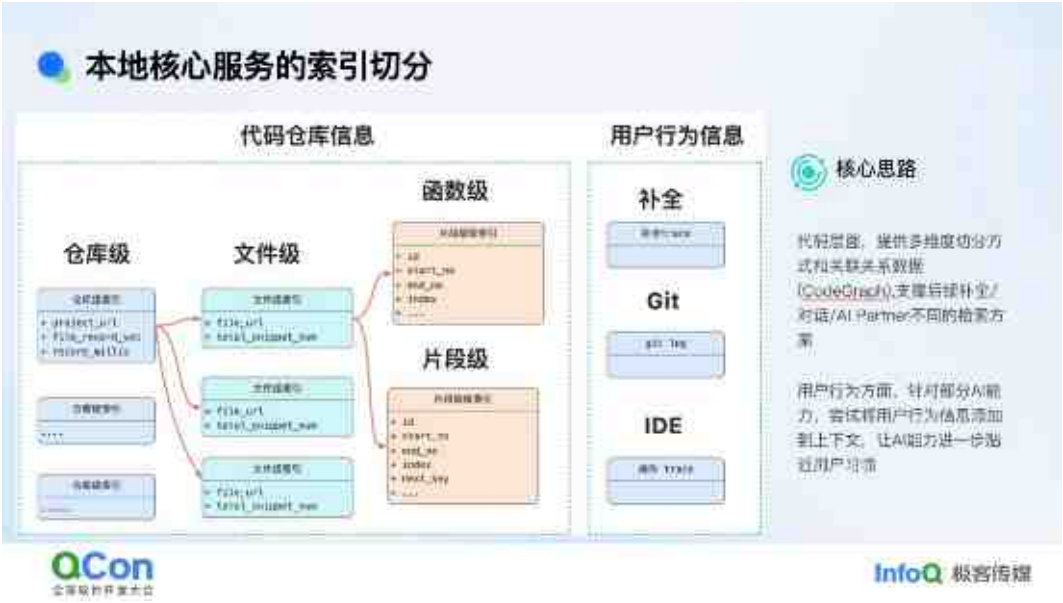
上线后的模型会通过服务端的各种A/B实验进行逐步推广。最初，可能会分配5%或10%的流量进行测试，每天都会关注模型的效果。如果效果良好，流量会逐步增加；反之，效果不佳的模型则会被下线。通过这种方式，我们确保了CodeFuse插件在提供强大AI能力的同时，能够持续优化和提升用户体验。

本地核心服务模块详解

补全过程详解

重点介绍一下本地核心服务的设计与实现，这部分的核心目标是提升上下文的质量。要实现这一目标，我们需要从三个关键维度进行考量：数据来源、检索策略以及提示工程。数据来源决定了我们能够为AI请求提供哪些补充信息，这些信息可能来自代码仓库、知识库或用户行为记录。检索策略则涉及如何高效地从海量数据中提取相关信息，我们采用了多种检索方式，包括BM25、向量检索和图检索，同时支持单轮或多轮检索。而提示工程则关乎如何将检索到的数据进行有效拼装，以何种方式呈现给模型，这直接关系到最终的用户体验和效果。这三个维度缺一不可，每一部分都对上下文质量有着显著影响。

本地核心服务的构建基于索引机制。由于服务运行在本地，它主要处理本地仓库的数据，而不涉及跨仓库的内容。具体来说，本地核心服务会遍历整个代码仓库，并将代码拆解为四类信息。首先是仓库级信息，例如仓库的Git地址、README文件内容以及仓库的Wiki概要等。其次，对于仓库中的每个文件，我们会记录其功能描述和注释信息。这些信息构成了文件级索引。然而，仅凭文件级索引是不够的，因为在对话或代码补全等场景中，我们通常需要检索代码块级别的内容。因此，我们进一步对文件进行了切分，目前采用两个维度：函数级别和片段级别（trunk level）。函数级别切分能够提供更精准的代码结构信息，适用于大多数编程语言；而片段级别切分则主要用于应对一些非主流语言或脚本语言，这些语言可能无法进行函数级别的解析。此外，片段级别索引还可以作为兜底方案，确保即使在函数级别索引无法满足需求时，仍能提供一定的检索能力。例如，对于SQL脚本或执行脚本，片段级别索引能够发挥重要作用。



除了代码相关数据，本地核心服务还会记录用户的行为数据，例如代码补全的历史记录、Git日志以及IDE操作记录等。这些行为数据有助于将用户的行为习惯融入上下文，从而使AI能力更加贴近用户的实际需求。

在代码补全场景中，上下文内容主要由相关性、相似性和历史行为三部分构成。相关性是指在补全时提供与当前代码片段直接相关的结构信息。例如，当用户输入“userService”时，如果能明确告知模型对应的“UserService.java”的具体结构，如其中包含“addUser”方法，那么模型就能更准确地进行补全，而不是随意猜测出“saveUser”或“insertUser”等可能并不准确的内容。相似性的作用则是学习用户之前编写代码的逻辑。例如，如果用户在仓库中多次编写了“new User()”后紧接着调用“user.setId”方法，那么在新的代码文件中再次编写“new User()”时，模型会根据之前的行为模式推测用户下一步可能需要调用“user.setId()”方法。历史行为则是基于用户在IDE中的操作习惯，例如某些用户可能习惯先编写方法名，然后通过快捷键补全前面的代码。这些行为记录会被CodeFuse捕捉到，并在后续的补全中自动应用，从而提高补全的效率和准确性。

在实现这些功能的过程中，我们面临的一个重要挑战是如何在短时间内完成大量的计算任务，以确保补全的响应时间足够短。一般来说，一次常规的补全操作整体耗时控制在500毫秒以内完成，超过这个时间，用户就会感受到明显的延迟，从而降低补全的

可用性。目前，我们已经将大部分补全操作的响应时间控制在200到400毫秒之间。整个补全链路中，模型推理通常会占用较多时间，一般需要上百毫秒。为了优化这一过程，我们采用了动态延迟策略。当用户输入字符时，并非每次输入都立即触发补全请求，而是设置了一个延迟时间。这个延迟时间是动态调整的，根据不同用户对时间的感知能力进行优化。例如，有些用户可能在100毫秒的延迟下有较高的采纳率，而有些用户则在30毫秒的延迟下表现更好。通过动态调整延迟时间，我们能够在保证补全效果的同时，尽量减少用户的等待时间。

除了动态延迟策略，我们还对数据检索部分进行了优化。在第一次尝试时，我们将目标设定为50毫秒以内完成检索任务，但实际上我们已经能够将这一时间进一步缩短。在补全的上下文中，相关性检索并非简单地提取导入语句，而是根据光标位置找到与当前代码片段最相关的代码文件。例如，如果光标前有一个“user.setId”语句，我们需要找到“user”所代表的对象，如“User.java”，并提取“User.java”的结构信息。同时，我们还需要对光标前后一定范围内的代码片段与全仓库的代码片段进行相似度对比。然而，这一过程的耗时可能会非常惊人。以一个包含约2,000个Java文件的仓库为例，按照我们之前提到的索引切分方式，可以切分出约5～6万个代码片段。即使是这样规模的仓库，一次相似度计算的耗时也可能超过100毫秒，更不用说那些规模更大的仓库了。这曾是我们面临的一个重大难题。

为了解决这一问题，我们从两个维度进行了优化：数量和编码。从数量上，我们结合了用户编写代码的习惯，通过定制规则筛选出与当前代码片段最相似的文件。例如，在编写“UserService.java”时，我们会查找与“UserService.java”路径相似或import语句相似的代码文件，并对其进行排序，取前50个文件。然后，从这50个文件中提取对应的索引片段，并将这些片段放入一个固定长度的集合中，例如5,000个片段。在每次补全时，我们只需将光标前后提取的代码片段与这5,000个片段进行相似度对比，而不是与整个仓库的所有片段对比。这样可以显著减少数据量，从而提高检索速度。

从编码角度，我们对代码片段进行了进一步优化。由于相似度计算本质上是字符串对比，其速度和资源消耗相对较高，我们将代码片段文本编码为数字数组，通过对比数字数组来加速相似度计算。

在本地核心服务中，我们充分利用了用户端的CPU资源。在补全的瞬间，我们会将

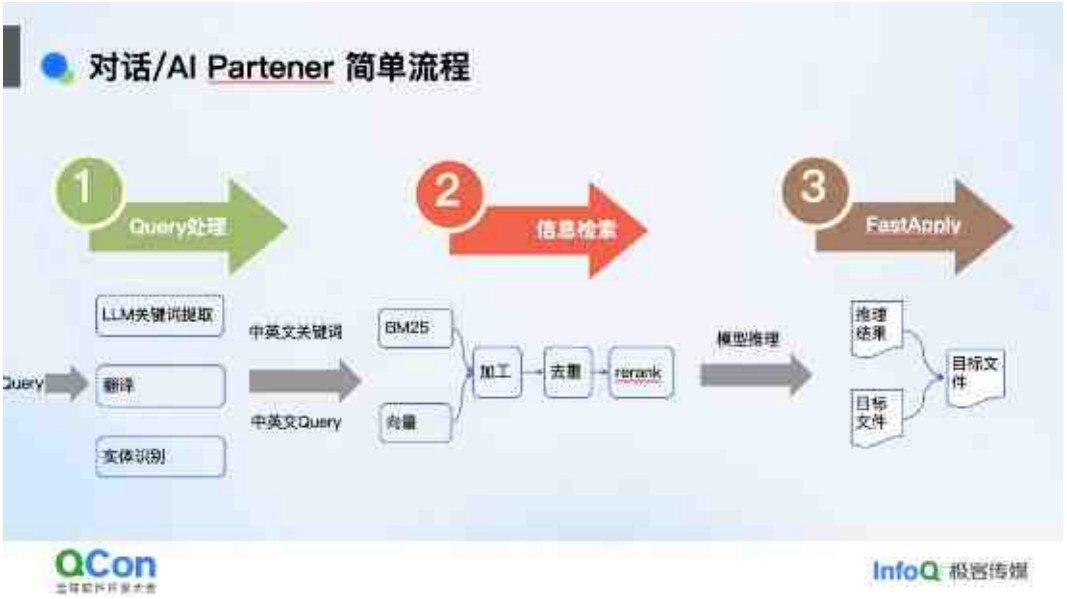
5,000个代码片段的相似度对比任务分配到不同的CPU核心上进行并行计算，然后将结果存储在二叉堆中，从中提取前几名的数据用于后续处理。在这个过程中，一旦某个CPU完成当前所有相似度对比任务队列，它会尝试从其他CPU任务队列中获取新的任务，从而确保CPU资源得到充分利用。尽管在用户本地端上计算相似度时对CPU资源的消耗非常敏感，但我们在实际运行中发现，整个相似度计算过程的耗时非常短，通常在20毫秒以内。在这个时间内，用户端上几乎无法察觉到任何延迟。落地实际效果，CPU消耗通常小于3%。

对话流程详解

之前提到的AI Partner，其实可以简单理解为一个智能编程助手。举个例子，假设我在代码中实现了一个快速排序算法，然后我让AI Partner帮我写一个调用这个快速排序的函数。AI Partner能很快的帮我写出来。当然，这只是个简单的例子，实际的推理过程会更复杂一些，推理完成后会生成一个可直接运行的代码片段。

CodeFuse的对话和AI Partner的实现流程可以分为三个主要步骤：查询处理、信息检索和快速应用。首先，当用户提出一个需求，比如“创建一个用户接口”，查询处理阶段会提取关键词，如“用户”“创建”等，并识别其中的实体，例如“UserService”。经过这一阶段，用户的原始需求会被转化为两份数据：一份是中英文关键词集合，另一

份是中英文查询集合。关键词会通过BM25算法进行检索，而查询集合则通过向量检索进行处理。检索完成后，会对数据进行去重、重新排序等加工操作，然后将这些数据输入模型进行推理。推理完成后，模型会生成一个相关的函数，比如“createUser”函数，而不是生成整个文件。



接下来会遇到一个问题：这个函数应该插入到当前仓库的哪个位置？为此，我们训练了一个专门的模型，用于判断生成的函数在目标文件中的合适位置。通过这一系列流程，可以完成一些简单的需求。然而，大家也能感受到，对于复杂的业务需求，这种流程还远远不够。例如，如果让用户直接将生成的代码应用于线上环境，大家肯定不敢这么做。那么，问题出在哪里？是模型的问题，还是检索环节出了问题？我们深入思考后发现，人解决这类问题的方式或许可以给我们一些启示。

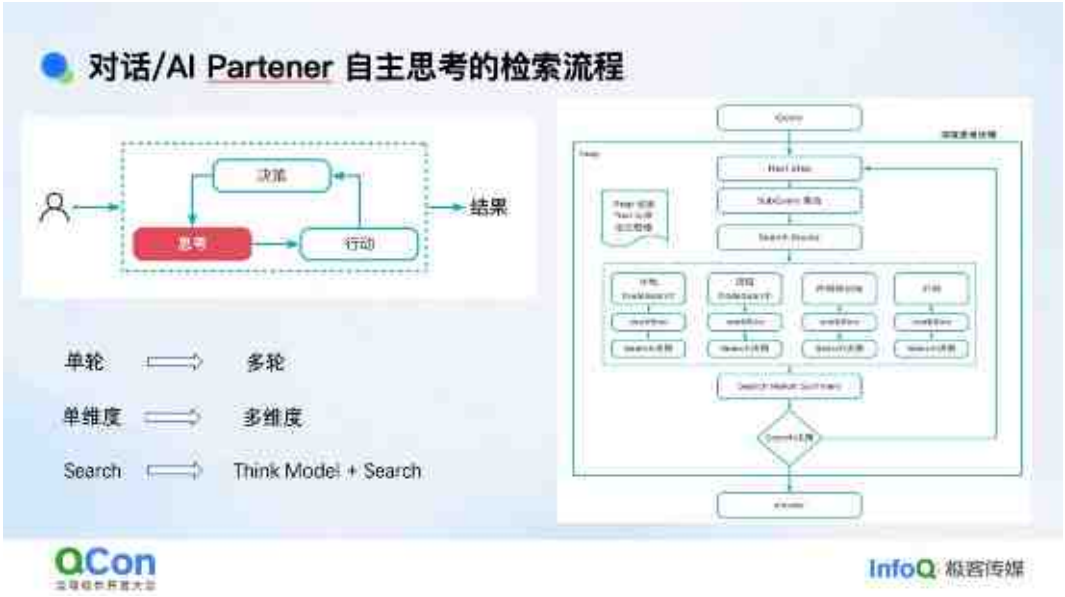
假设你让一个默认开发者编写一个创建用户接口，他写出来的代码能直接用在你的业务中吗？答案是不能。即使加上一些背景信息，比如告诉他有一个“UserService”，让他基于这个服务编写接口，结果依然不能直接使用。其实，这种表现和模型的现状很相似。既然人也很难解决这个问题，那为什么还要让模型去直接解决呢？我们进一步思考，现在开发者是如何编写出可用代码的呢？

想象一下，当你招聘一个人到公司，让他编写一个创建用户接口，他一开始肯定写

不出来，因为他对业务背景和应用基础信息一无所知。如果告诉他这个接口用于一个内部系统，每天只有十几个人使用，他可能会直接实现一个简单的数据库CRUD操作。但如果告诉他这个接口将用于一个面向消费者的电商平台，他肯定不会只做简单的CRUD，而是会考虑是否需要限流、权限校验等其他功能。这说明，人在编写代码时，会基于多维度的信息进行判断，比如业务背景、应用信息等。此外，人还会进行多步骤的规划，而不是拿到需求就直接开始写代码。他会先了解各种信息，然后打开仓库，找到相关的接口和领域对象，思考如何复用现有代码，哪些地方需要更新，最后找到入口函数开始编写代码。这个过程与我们之前提到的流程有明显的区别。

我们发现，人解决问题的方式有两个关键特点：一是多维度的信息收集，二是多步骤的规划能力。基于这些特点，我们尝试设计一个具有自主思考能力的检索系统，以提升AI代码生成的能力。

从宏观角度看，当用户提出一个需求时，我们首先让系统进行“思考”，将任务拆解为三个步骤：思考、行动和决策。思考阶段进一步细化，让模型先判断第一步做什么，比如先了解应用的背景信息，然后基于这个信息拆解出多个子查询，如读取应用的README或远程检索仓库的Wiki等。这些子查询会根据不同的检索条件走到不同的检索流程，然后获取解决子查询问题所需要的数据。



获取数据后，系统会进行决策，判断这些数据是否能满足子查询的需求。如果不能满足，继续检索；如果能满足，将所有子查询的结果汇总，进行简单总结，然后询问模型是否需要下一步操作。如果有下一步，整个流程会重复；如果没有，则结束。相比之前的流程，这个新流程有三个显著特点：一是检索方式从单轮变为多轮；二是从单维度变为多维度；三是将原本单纯的检索流程转变为一个模型与检索相结合的流程。这个流程参考了人类解决问题的方式，让AI Partner具备了自主思考的检索能力。

经验与总结

全链路的A/B实验

在CodeFuse的落地过程中，我们始终致力于提升AI效果，这些努力主要通过技术手段来实现。然而，技术手段的优化只是第一步，更重要的是如何判断这些优化是否真正有效。在我们看来，A/B实验是验证效果的关键环节。我们开展的A/B实验不仅局限于模型层面，而是贯穿于整个产品链路，包括插件侧和服务端等多个环节。

以插件侧为例，我们曾对动态延迟时间进行优化。动态延迟时间的调整是通过一个算法小模型实现的，而我们开发了多种算法小模型，并通过A/B实验来比较它们的效果，最终选择表现最佳的模型。在服务端，我们也进行了大量A/B实验，涉及prompt的拼装、任务阶段的组装等多个方面。本地核心服务部分的实验更为复杂，例如代码切分方式和检索策略等。我们曾针对相似度检索进行实验，通常情况下，我们会以光标前后的内容为基础进行检索。然而，通过A/B实验，我们发现对于某些编程语言，仅使用光标之前的内容进行检索，然后将检索结果下移一定行数（例如10行或20行），这种简单的策略调整竟然使采纳率提高了2%。这充分说明了A/B实验在优化检索策略方面的价值。

业务定制化方案

在与业务方合作的过程中，我们也遇到了一些挑战。一方面，同一种能力在不同业务背景下表现存在差异。以代码补全为例，虽然我们内部测试的采纳率可能达到“80%”，但在与业务方沟通时，有些团队反馈他们的实际采纳率可能只有“70%”或“60%”，而有些团队甚至能达到“90%”（这里所提及的数字均为模拟案例，不代表真实业务数据）。这种差异的原因在于不同业务场景（如终端开发、数据研发或业务代码编写）的逻辑可能截然不同。另一方面，有些团队有定制化的对话诉求，例如他们需要检索自己业务平台上的数据，而我们的通用检索方案可能无法完全覆盖这些需求。

为了解决这些问题，我们采取了以下措施。首先，针对定制化的检索需求，我们开放了本地核心服务的定制能力。本地核心服务是一个用Rust编写的可执行文件，其代码结构分为四层：基座层负责将代码打包成可执行文件；SPI接口层定义了通用接口，例如代码补全和对话检索的接口；数据层提供数据索引和检索能力；中间层则是CodeFuse默认实现的一套检索和索引构建能力。当业务方有定制需求时，我们可以将这些定制能力开放给他们。例如，终端团队如果希望提高Android或iOS语言的采纳率，他们可以基于自己的业务需求进行检索定制。这种定制化的检索方案能够显著提升上下文质量，从而更好地满足业务需求。

其次，针对用户有定制化检索诉求的情况，我们支持用户创建自定义的助手。用户可以选择合适的模型、系统提示词以及工具（这些工具背后会处理文档切分和检索等任务）。用户还可以上传PDF文档或其他文档链接。完成自定义助手的创建后，用户可以在平台上设置权限并发布。这样，内部用户就可以通过@提及的方式，使用这些定制化的检索和对话策略。通过这种方式，我们不仅满足了用户的个性化需求，还进一步提升了CodeFuse在不同业务场景中的适用性和灵活性。

未来展望

下图左侧是OpenAI提出的AI发展方向，我认为其观点确实具有启发性。OpenAI的理念更多是基于通用场景展开的，而我们团队在思考如何将这些理念具体应用到研发领域，尤其是研发助手或IDE插件方面时，提出了一个分阶段的发展思路。

在我看来，研发领域的AI应用可以分为三个阶段：AI辅助、AI协同和AI驱动。在AI辅助阶段，我们看到的主要是像代码补全这样的功能。这种能力为开发者提供了便利，但仍然只是辅助性质的，开发者是主导，AI只是提供一些可能的代码片段或建议。

接下来是AI协同阶段，这正是我们目前CodeFuse所处的阶段。在这个阶段，AI不再仅仅是被动地提供帮助，而是像一个“小机器人”一样，在背后默默地支持开发者。当开发者有简单的任务时，可以直接分配给AI，让它协助完成。例如，AI Partner就是这种能力的体现，它可以根据开发者的需求生成代码片段，甚至完成一些简单的编码任务。

最后是AI驱动阶段，这是我们的长远目标。在这个阶段，AI将具备一定的自主思考能力，能够像一个独立的开发者一样工作。例如，当提出一个需求时，AI不仅能够生成代码，还能自己进行测试，并在测试完成后自动提交代码。这种能力将极大地提高开发效率，甚至可能改变软件开发的模式。

目前，CodeFuse正在努力完善AI协同阶段的能力，我们希望通过不断优化和提升这一阶段的功能，为未来迈向AI驱动阶段奠定坚实的基础。

嘉宾介绍

- **牛俊龙**，蚂蚁集团技术专家，多年互联网系统研发经验，参与过多个亿级流量的重点系统设计和研发。目前就职于蚂蚁集团CIO技术部，致力于研发效能领域的AI落地方向，从0到1设计CodeFuse IDE插件的整体系统架构，将AI能力运用到补全，对话，TextToCode等多个功能，为研发人员提供更智能，更便捷的研发体验。

游戏研发中的AI转型：网易多Agent系统与知识工程实践

作者 林香鑫，网易游戏高级技术经理
审校 罗燕珊

近一两年大量AI编码工具如雨后春笋般诞生，但在游戏项目工程级别的代码编写参与度仍不高，主要受限于LLM上下文长度、游戏业务的复杂度与灵活性。

本文整理自网易游戏高级技术经理林香鑫6月份在AICon 2025北京站的分享《大模型在游戏研发中的落地实践》。团队通过代码知识谱图构建、多agent RAG召回、MCP等技术方式，打造了一个可完成复杂游戏编码任务的超级助手，在代码搜索、知识问答、功能迭代、新功能编写等业务场景均有不错的落地效果，已在公司内部广泛应用，同时推动了内部AI生成代码覆盖率大幅提升。

以下是演讲实录（经InfoQ进行不改变原意的编辑整理）。

在游戏研发管线中，我们正在开展一系列与人工智能赋能相关的中台化工作，其中也包括与AI编程工具相关的内容。今天，我主要想和大家分享一下大模型在游戏研发中的落地实践，重点在于我们在这一过程中的思考与感受。

理想与现实的差距

从早期的ChatGPT 3.5到GitHub Copilot，再到Devin、Claude Code等，我们可以看到随着大模型技术的不断发展，技术落地工作逐渐成熟，并催生了多种不同的产品形态。这引发了一个令人焦虑的问题：随着这些看似无所不能的工具的出现，程序员是否真的处于失业的边缘？对于从事AI编程的从业者来说，这种焦虑感尤为强烈，我们不禁思考，这些工具是否真的有如此强大的影响力？

在这样的浪潮下，我们开始思考在游戏研发过程中，这些工具是否真的适用。为此，我们开展了一次内部的大规模调研。调研结果显示，与大家的普遍认知不同，游戏研发人员花费最多时间的环节并非代码编写，而是代码理解。这一现象与游戏研发体系和模式有着天然的契合点。无论是策划、美术还是开发人员，虽然他们提交到代码仓库的内容不同，但整个游戏系统的运作需要多方协同。以一款知名游戏为例，其代码仓库行数达到了千万级别，这无疑带来了巨大的理解成本。此外，当出现问题需要调试时，这一环节也占据了相当多的时间，而代码编写反而排在后面。这正是游戏领域在研发效能方面面临的困境。

进一步分析，我们发现导致这一困境的原因主要有三个方面。首先，约30%的人认为游戏研发过程中缺乏清晰、及时的技术文档，这与游戏的发布周期密切相关。例如，一个资料片可能需要在两个月内上线，根本没有时间进行详细的技术设计，通常是基于上一次资料片的经验快速复用并上线，这种心态导致了技术债务的不断积累，占比约28%。其次，游戏研发管线比传统的Web开发复杂得多，测试、编译、调试的速度缓慢，一次性审查大量代码耗时过长，这也是一个突出问题。

针对这些问题，我们借助大模型和Copilot的初期优势，在2024年初构建了一套以IDE加Web双端形式存在的系统。除了常见的代码聊天、代码补全等基础功能外，这套系统更多地围绕团队如何开展研发工作而设计。团队可以定义自己的规则、知识库，并在系统上运行代码审查流程，甚至可以随时针对一段代码发起讨论。

但这些并非重点，重点在于我们利用AI赋能的机会，将所有面向程序的工作逐步集中到一个入口，即大家常用的IDE。我们不希望研发人员每天在不同的地方切换工作，而是希望他们能够在IDE中完成所有工作。同时，我们通过Web后端形成与IDE相互支撑的形态，IDE的使用情况和数据会回流到后端，而后端配置管理的团队数据也可以在IDE中使用，从而将整个游戏研发过程集中到一个入口。

在过去的一年中，我们重点解决了三个方面的工作。首先是缩短代码理解周期。代码理解不仅涉及代码逻辑，还可能包括游戏玩法、角色技能等开发人员需要理解的内容。其次是代码补全与生成，我将重点分享我们内部是如何落地代码生成的。最后是经典的代码质量问题，即AI代码审查。虽然这一领域备受关注，但实际操作起来并不容易，今天我也会分享一些相关经验和心得。

游戏研发知识工程

在游戏研发过程中，如何更好地理解代码及其背景知识，是我们一直在探索的问题。在这个过程中，我们逐步构建了一套游戏研发知识工程体系。一个游戏团队通常涉及策划、研发、美术等多个角色，大家每天需要协同交互的问题非常多。例如，一个游戏角色的技能是什么？像最近大火的某款竞技类游戏，已经有 几十个角色，每个角色可能有好几个技能，普通人很难全部记住，只能通过询问来获取信息。再比如，去年的中秋活动是如何实现的？如果我要设计今年的中秋活动，自然希望借鉴去年的经验。还有如何让UI界面背景实现毛玻璃效果，这可能涉及组件配置属性等问题。此外，开发人员和QA经常会因为“我改了这个东西会不会影响其他东西”而产生分歧，而这些问题的答案都藏在代码仓库里。

在这种背景下，我们遇到的首要问题是大家都不喜欢写文档。程序员最讨厌的事情

之一就是写文档和注释，这导致知识很难沉淀下来。即使有了文档，很多时候也用不上，因为直接询问同事往往能更快得到回应。

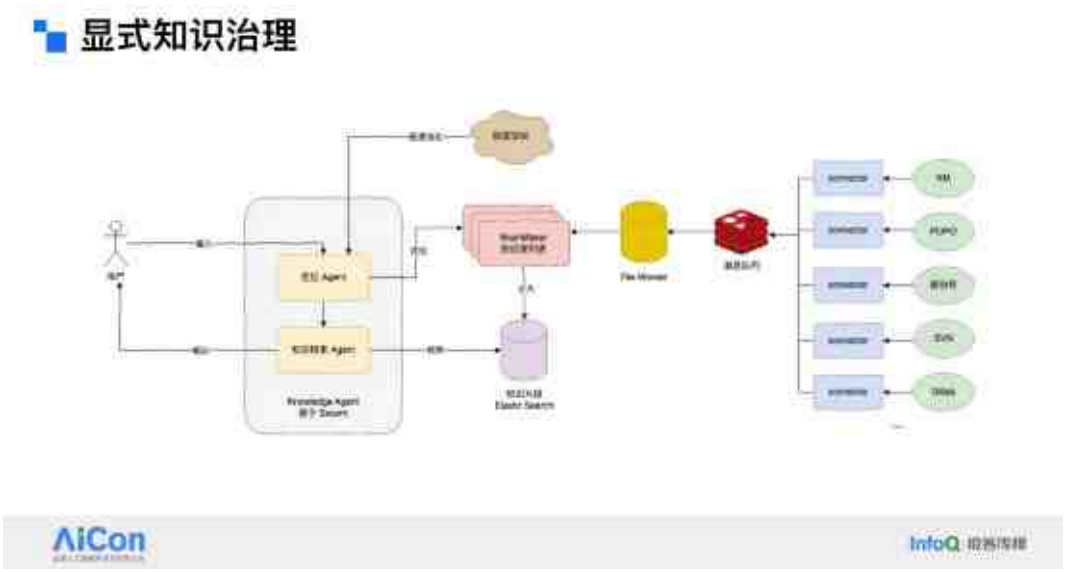
然而，在大模型技术爆发的当下，这一问题有了新的解决方案。大家应该都听说过智能问答系统，那么如何利用RAG或Agent技术来解决这个问题呢？我们推进了这一项目，因为如果能让人们快速获取知识，又不希望在这个过程中互相损耗效率，那么引入一个持续工作的第三方系统就显得尤为重要。

在大模型时代，知识工程指的是对企业内部数据进行系统化的治理、挖掘、应用管理与迭代，构建一个能够支撑AI Agent运行的知识体系。这听起来概念简单，但涉及的内容却非常广泛。比如结构化知识，包括游戏中的技能、角色、系统、玩法等，这些都可以定义为一系列结构化数据。同时，也涉及大量非结构化文本，比如角色背后的设计与解释，这些可能以文档形式存在，而在游戏行业，很多策划喜欢用Excel管理内容，这些都是非结构化数据。此外，还有领域知识、团队积累的案例，以及潜移默化运行的内部业务模式等，这些对企业来说都是知识，因为它们是你每天可能接触或需要应用的内容，如果你要完成某个实现，就需要了解这些知识。

基于前面提到的统一入口，我们将大家理解的通用知识，比如从GPT或Claude等模型获取的知识，升级为具有内部知识的聊天模式。对于游戏来说，除了业务逻辑和知识，

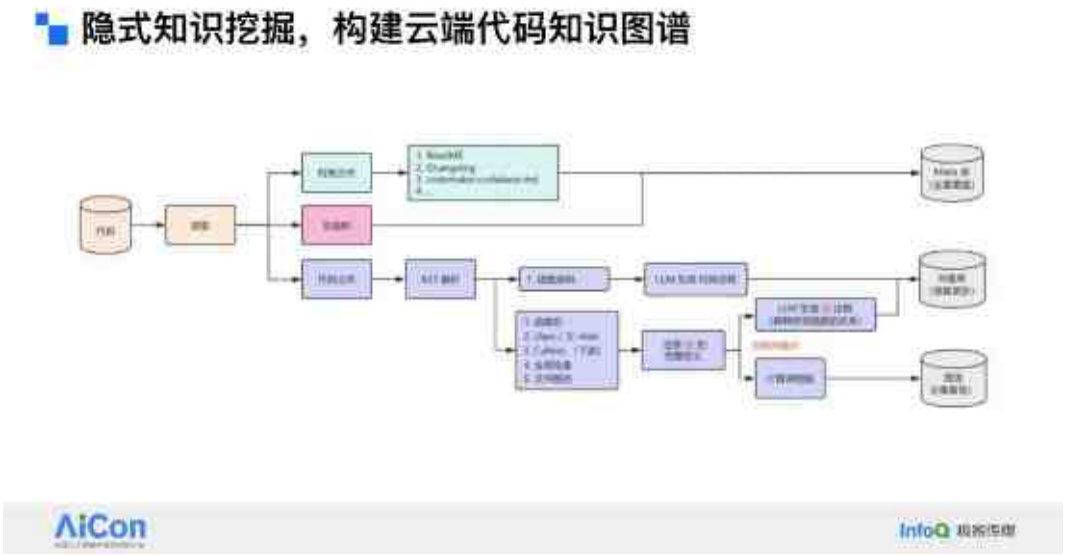
还有一个非常重要的点，那就是游戏引擎。我们内部基本都使用自建的游戏引擎，因此我们将这些知识关联在一起，一旦需要使用，就能通过这种方式快速获取。比如用到了引擎的哪些API、哪些SDK，如果在实现过程中能够快速获取这些信息，那将是一种更高效的方式。

基于前面的讨论，我们可以将知识分为两类。一类是显性知识，即那些即使不问AI，也可以在文档中找到的知识。在这个过程中，我们主要做的是已有知识的治理。对于内部的知识管理系统、工单系统、代码仓库（如SVN、GitLab等），我们都实现了数据的归集，并将其处理进一个统一的知识库，或者说是团队的知识大脑。然后通过agentic RAG等方式，实现了对知识的切割、解读和检索。我们最深入的工作是定义AI Agent，希望通过AI Agent的路由，以最高效的方式找到相关数据，这是显性知识治理的过程。



另一部分非常重要的是隐性知识。如果一个开发人员要实现某个功能，他可能每天都在与代码仓库打交道。在这个过程中，我们进行了隐性知识的挖掘。其实一切都在代码里，我们的目标是构建一套代码知识图谱。这里用到的技术其实很传统，只是结合大模型的发展，能够进一步在业务价值上激活。首先，我们对代码目录结构进行了解析，因为代码目录结构对于理解一个工程来说非常重要。其次，我们基于抽象语法树（AST）和调用流，完成了对整个仓库中所有函数甚至所有类的语义化解读，将原本枯燥难懂的代码变成语义上更容易理解的文本。接着，我们挖掘了代码中的调用关系和调用链，形

成了一张地图，最终构建了一张完整的代码地图。虽然可视化可以做得非常炫酷，但重点是我们对代码仓库进行了非常精细的解构，让你可以通过聊天的方式获取其中的信息，这是非常重要的一个点。

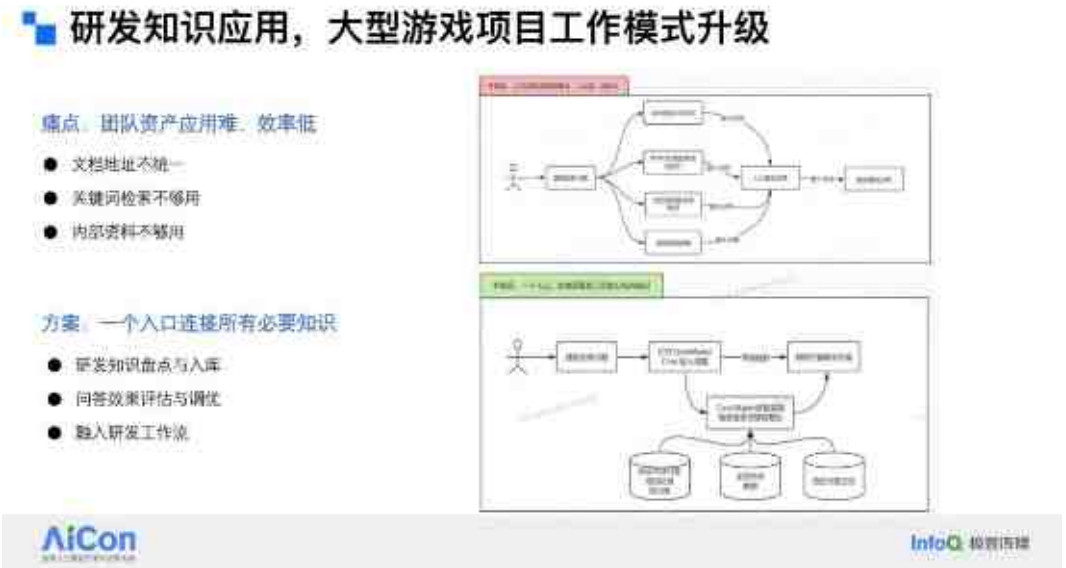


在整个过程中，我们还会剔除一些特殊的文件，因为这些文件可能会对原本的知识产生干扰。此外，除了完成目录树本身的抽取，我们还做了一件事，就是基于以往的所有工单，工单里会有与提交记录的关联，我们将这一层信息打通，这意味着一个业务模块与代码文件，或者某段解读完的代码逻辑是有关系的，从而形成了一个具有业务背景知识的代码知识解读结果。剩下的工作是基于传统的AST技术或线上的动态调用链关系组织出的一张图谱。

Codemaker最新推出的全量仓库代码可视化功能模块，对逻辑完整的代码仓库进行局部代码解析后生成的内容，我们称为一份代码地图。它包含代码文件、函数之间的调用关系，以及文件、类、函数的语义化解读，对函数复杂度、代码规模的洞察。数据支持简洁模式和代码模式，支持与地图交互展开查看更多上下游信息，支持人工标注对应模块。

经过这个过程，代码仓库对我们的意义已经从单纯的代码存储，变成了一个可以贡献知识的资源。这意味着在一个100人的团队中，一个人能够了解所有人写的代码，只要愿意与它交流。

我们也将这一模式应用到了很多大型游戏中。例如之前在海外非常火的一款知名超英题材多人在线游戏，整个游戏涉及几百人的共同研发。在这个过程中，我们直接基于他们的场景解决了这些问题。痛点非常明确：资产非常多，但很多都没有被充分利用，而大家每天都在互相询问各种问题，比如“这是怎么回事”“那个是怎么实现的”“数值是多少”等。我们统计发现，大家每天可能需要花费20%的时间去获取这些信息，这对效率的阻碍是非常大的。最后，我们通过前面的各种方式实现了一个入口，能够连接到所有必要的支持，形成了这样的模式。



我们还对一些日常问题进行了分类，这些问题远不止开发层面上的，还包括开发技术、公共引擎、游戏设计工具、环境配置，甚至涉及一些项目管理的流程。这是我们建立的模式。最后我想强调一点，如果仅仅完成了第一轮治理，只是把现有数据激活，这是不够的。在实践中我们发现，当大家开始使用这套系统时，会产生使用过程中的数据，这些数据可以进一步回流进来，补充整个团队的知识库。这一点非常重要，我们称之为迭代知识。补充这一环节后，整个知识体系就形成了一个正向循环，知识才能真正“活”起来。以前最大的问题是知识是“死”的，除非你去用它。现在大家已经养成了习惯，要找相关信息就来这里，这样就会产生过程数据，我们可以进一步利用这些数据。

Multi-agent赋能代码编写

在构建了坚实的知识底座之后，很多事情变得容易了许多。因为有了这样一个良好的知识基础，人们能够接触到这些知识，自然而然地，我们就会想到：AI是否能够接触到这些知识？如果它能够接触并理解，那么它是否能够帮助我们编写代码？这一阶段，我们正是在解决这个问题。

结合了知识工程之后，我们的系统能够理解整个游戏仓库的整体架构、具体的技术栈，以及如何运行，还能快速查找代码。这与大家熟悉的Cursor或Codebase有些类似，但正如之前提到的，游戏代码仓库的规模非常庞大，如果仅依赖本地模式，是无法满足需求的。因此，我们在前面的基础上构建了一套云端模式，只有这样，才能面向整个团队开展工作。

我们从传统的聊天模式升级到了一个由AI驱动的模式，我们称之为“仓库智聊”。这个名字的寓意很简单：我们的所有工作都可以面向仓库展开，或者说，我们的所有工作都可以基于仓库进行。大概就是这样的意思。

然而，我们很快发现这还不够。因为如果整个团队要基于仓库的代码实现展开工作，还需要很多其他的东西。为了让Agent能够基于对整个团队的业务背景或技术栈的理解，生成符合我们需求的代码，我们做了一件事：从团队的角度构建一个研发空间。如果你

作为一个新人加入一个团队，你首先需要了解什么？如果你什么都不了解就开始写代码，那么写出来的东西一定会被人嫌弃。你可能需要经过1到3个月的培训，才能逐渐掌握团队那些隐藏在水面下的知识。我们的研发空间就是想把团队那些隐藏的知识显性化，定义出来。

在这个过程中，我们发现，除了要让仓库能够被挖掘出来，以及梳理原有的项目过程文档之外，还需要关注很多细节。比如，我的游戏正在使用哪个版本的SDK？我正在使用哪个版本的引擎代码？里面是否还用到了其他通用的技术库？这些内容其实都隐藏在我们日常的运作过程中，但从未被真正梳理出来。我们正在做的，就是把这些内容梳理清楚。

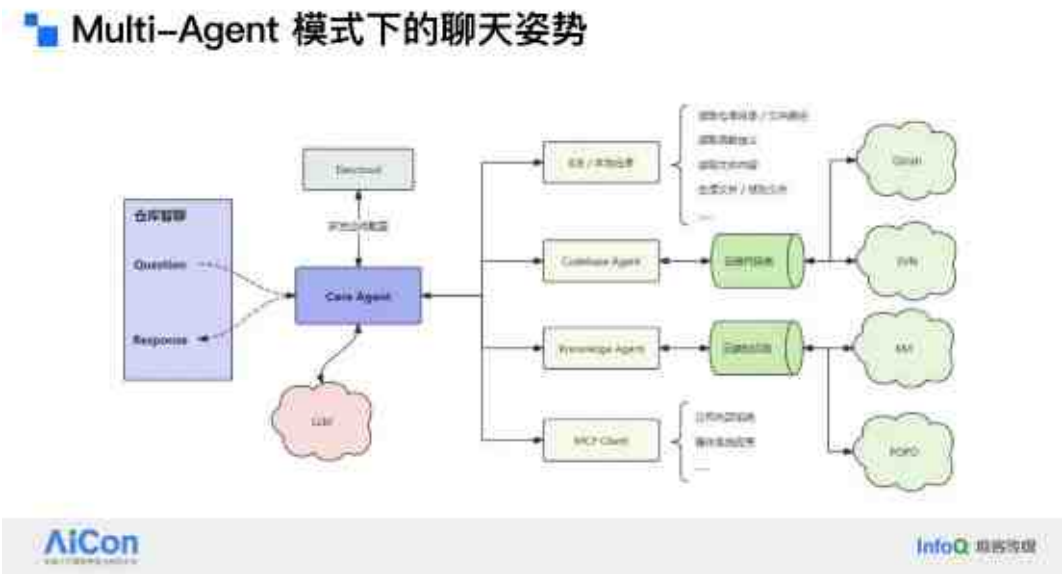
还有，这个团队的编码风格是怎样的？曾经有一个团队领导使用我们的功能时，一开始抱怨说，虽然生成的代码看起来都没错，但不符合我们团队的风格。这确实是一个问题。一旦这种风格被定义出来，比如我们规定Python应该使用哪个版本，或者遵循哪些代码规则，慢慢地，你就会发现，它越来越像这个团队的一个成员在工作。我们就是基于这样的前提构建了研发空间。

这里有一个案例。在梳理完前面提到的这些内容之后，整套架构其实非常清晰。这与大家在网上看到的通用Agent架构没有太大区别。真正核心的区别在于，我们给Agent提供了什么样的上下文。现在有人说，未来工程师可能是面向上下文编程，我对此深有感触。Agent架构最后的差异不会太大，真正重要的是你给Agent提供了什么样的信息。这正是我们在构建团队空间研发空间时所关注的内容。

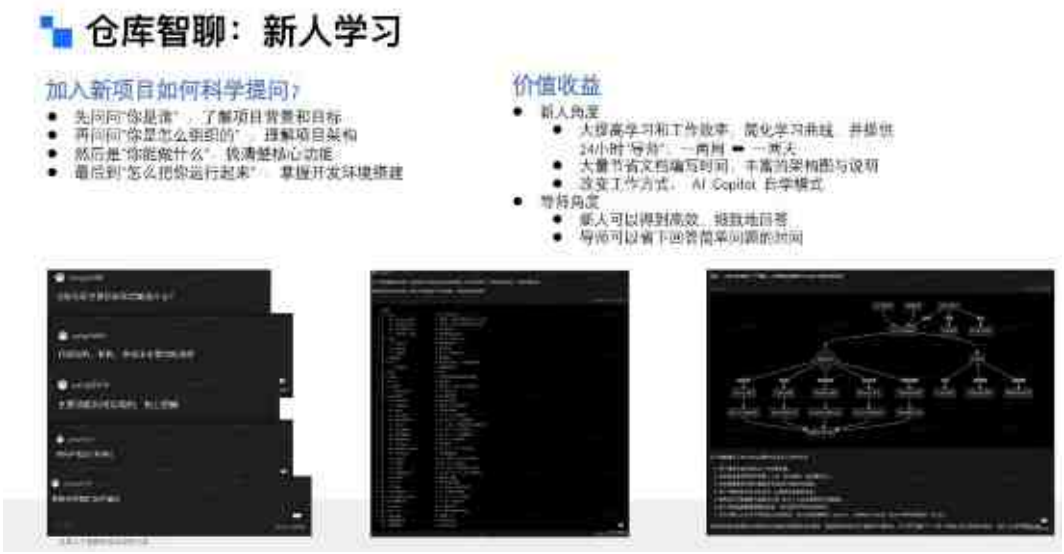
在这种模式下，我们定义了一个核心Agent（Core Agent）。这个Agent与其他架构中的Agent类似，能够进行规划和拆解，但最重要的是，我们基于内部工具或内部数据，为它提供了什么样的人员去协调。这一点非常重要。如果你协调的是一个不了解这个项目的人，他可能会给出错误的信息，自然也就很难生成正确的结果。经过我们的治理，很多倾向于项目风格和项目偏好的Agent逐渐被定义出来并产生。经过核心Agent的协调，并且通过系统提示（system prompt）等方式让核心Agent表现得更好，整套体系就会逐渐成熟起来。这就是我们对结合研发空间的多Agent系统的思考方式。

目前，虽然大家看到的形态是一样的，但背后的逻辑是不同的。我们提出一个需求，比如增加一个功能或修改一个逻辑，最后它会帮你完成生成。我想强调的是，真正困难

的是背后的这部分内容。如果你直接把一个开源的Agent应用到你的团队，它真的能正常工作吗？我对此表示怀疑。因为最终我们发现，核心价值在于背后的这些细节。



在完成这些工作之后，我们在内部落地的情况也值得分享。首先，我们将整套系统应用到新人学习阶段，形成了一个非常明确的指引。在这种工作模式下，新人应该如何循序渐进地利用这个工具来完成理解？我们有一个实践案例：我给了一个新人4万行代码，原本估计他需要两周时间来熟悉，但我告诉他使用这个工具，最后他告诉我只需要一到两天。因为4万行代码，说实话，如果你一行一行地去读，或者在没有任何引导的情况下熟悉它，那是一个非常枯燥的过程。但在前期加入团队学习时，你其实不需要这么细致地去了解所有内容，更多的是有人帮你梳理这些内容，让你能够理解整体框架，以及未来你要修改的内容和位置，这些都会有一个清晰的指导。这也间接释放了导师的负担。以前，大家都要一对一地指导新人，现在可以让AI来指导新人。



从项目协同的角度来看，虽然我们的系统最初是围绕代码生成设计的，但经过大家的探索，它现在可以用在很多方面。除了常规的生成变量名、业务代码逻辑、现有代码重构以及生成测试用例之外，它还可以用在很多创造性场景中。之前，一个内部用户偷偷告诉我，他用这个工具来写专利。这其实是一个挖掘已有内容并加以利用的过程。在实际应用中，这个工具在游戏团队中的使用率已经达到30%。当然，这是三个月前的数据。我最近看到的数据显示，这个工具一个月能够贡献大约500万行代码，分布在几十个游戏项目中。500万行代码是一个非常庞大的数量。

AI Review助力研发质量

在刚才提到的30%的代码由AI生成的情况下，剩下的70%仍然需要人工编写。如果手工编写代码，大家自然会非常关注研发质量。在我看来，游戏开发是一个比较特殊的领域，很多时候工期非常紧张，很多时候我们只能完成核心逻辑的代码审查。然而，一些运营事故往往是由一些非常低级的错误引起的。比如，我今天要设计一个活动，每人赠送十个仙玉，但如果手抖敲成了20个，甚至100个，从代码逻辑层面是很难发现这种错误的。因为你写了一个100，系统怎么知道这是错误呢？在没有上下文的情况下，这种问题很难被发现。因此，基于这样的特殊背景，我们内部形成了一套基于传统静态代码分析技术与AI代码审查相结合的方法，对整个从研发到测试再到发布的流程进行必要的代码质量把控。

代码审查确实是一个非常耗精力的过程，而且可能会打断你的工作。它的模式其实很简单，无非是提交合并请求，然后大家去查看。但正是因为这种简单且频繁的事情，大家很难坚持下去。所以，我们的目标是将代码审查过程智能化，争取在研发早期就发现潜在的错误。

我们之所以觉得AI适合用于代码审查，尤其是在游戏场景下，是因为我们发现很多错误其实是由一些比较低级的问题引起的，而AI恰好比较擅长处理这些问题。如果让高级专家专门去审查这些低级错误，他们可能会觉得没有成就感。此外，AI可以持续工作，我们内部已经构建了一套方式，白天由人编写代码，晚上由AI审查代码。这样基本不会占用你的时间，第二天你只需查看结果即可。

当然，我们也要有一个基本的心理预期，即AI能发现一个错误就是赚到了，而不是期望它能找出所有错误。基于这样的前提，我们开展了这项工作。

整个过程中，我们其实也遇到了不少困难。因为实际操作下来会发现有很多问题。如果作为一个通用的大模型，它存在一些典型问题，比如幻觉问题，这里就不多说了。另外，它会非常纠结细节，尤其是那些非常细的所谓规范性问题，其实大家通常并不在意。还有就是它的幂等性很难保持，今天说这里是问题，明天可能又说这里是另一个问题。当你重复提交同一行代码时，可能会得到不同的反馈。最重要的是它不懂业务。就像刚才说的，如果我要实现赠送10个仙玉，但你写成了100个，它在没有业务背景的情况下也发现不了。所以，我们的整体目标是从原来误报多、不准确的状态，逐步演进到

少而精的状态。

整个技术升级过程其实也经历了几个阶段。最初，大家做AI审查时，明确要做的是搞prompt工程，让它去发现问题。但结果并不理想，业务团队会抱怨说“搞了1000个问题给我，我该怎么去修复和确认呢？”因为可能里面只有10个是有效的。第二轮，我们基于双引擎，把静态分析和大模型结合起来，先由静态代码分析过一轮，再由大模型做补充。结果发现，这样发现的问题更多了。

后来，我们引入了Multi-Agent协同架构，因为业务团队可能会说“我不在意这个问题，帮我抑制掉”。于是，我们做了像filter、AI这样的东西，也尝试对问题进行分类分级，因为每个业务关注的点不一样。我们不断引入更多单项能力的AI来对问题进行进一步筛选和定义。最后，我们进一步解决它不懂业务的问题，引入了前面提到的知识工程，逐步将其应用到AI审查过程中。

在产品形态上，从早期依赖人工自主发起审查的过程，逐步演进到与我们的流水线集成，甚至与工单系统进行质量门禁。最终，所有信息都集中在一个入口，即IDE上。

在代码审查方面，我们支持本地代码审查和团队协作审查。本地代码审查无需接入支持，可以直接对待提交代码或划选文件代码范围一键发起审查，支持查看审查结果并处理问题。团队协作审查则需要接入如GitLab等代码仓库，在Web端或插件端创建审查请求。用户可以选中需要审查的Commit或MR，配置团队审查成员，勾选AI审查和AI解读，也可以添加工单信息辅助AI审查。

填写完成后提交创建，创建完成后，在审查详情页可以查看审查详情，查看AI反馈的问题并标记其有效性及标签，还可以对问题进行回复，触发与AI模型的进一步交互。点击代码DIFF右侧的加号，可以添加一条新的问题。上述操作也可以在平台侧进行。平台还提供面向管理员的仓库维度的审查数据统计。

这是我们目前演进的形态。在这个过程中，我们还发现了一个好处。以前，一些错误解决了就解决了，但经过这样一套中央管理后，我们会发现一个很大的保障：错误数据在哪里发生，以及如何修复的，我们逐步将其沉淀下来。这对于我们后续进一步提升能力，甚至如果真的要去训练一些模型，这些数据的价值非常大。另一方面，这些错误的经验可以作为整个团队编码的进一步输入，帮助抑制一些问题，这也是我们目前正在尝试的一个方向。

未来展望

未来还有许多工作要做，但我认为这一切都离不开一个核心要素——前面提到的研发空间。对于我们这些专注于内部落地工作的人来说，经过多年的摸爬滚打，我们逐渐意识到最关键的问题是：研发数据究竟在哪里？这个问题至关重要。因此，我们优先要做的，不仅仅是面向研发人员的知识治理，而是将整个体系演进为一个团队的大脑。

目前，像Cursor或其他工具更多地帮助个人维持记忆，但我们希望构建的是一个能够维持团队记忆的系统。无论谁加入团队，都能获取过去多年积累的相关经验。这正是我们想要实现的目标。策划文档、研发规范、工单处理经验，甚至我们在即时通讯（IM）中沟通得出的结论，这些内容构成了整个团队的记忆。虽然它们在日常工作中看似零散，但当我们把它们整合在一起时，却能发现不少惊喜。

由于我的工作涉及整个全流程环节，我一直在思考如何在游戏研发管线中让一套真正体系化的团队AI Agent运转起来。我们发现，策划与研发、策划与美术、研发与QA之间的交流，最终都会汇聚到一些共同的点上。那么，这些知识和Agent是否能够体系化地协同工作，就显得尤为重要。到目前为止，我们更多地是在单点上解决职能相关的问题，但未来，我们希望这套团队AI Agent能够像真正的团队协作模式一样运作起来，基于前面提到的支持Agent进行操作。目前，我们已经在进行一些尝试，并且积累了一些相关案例，期待在下次有机会与大家分享更多这方面的内容。

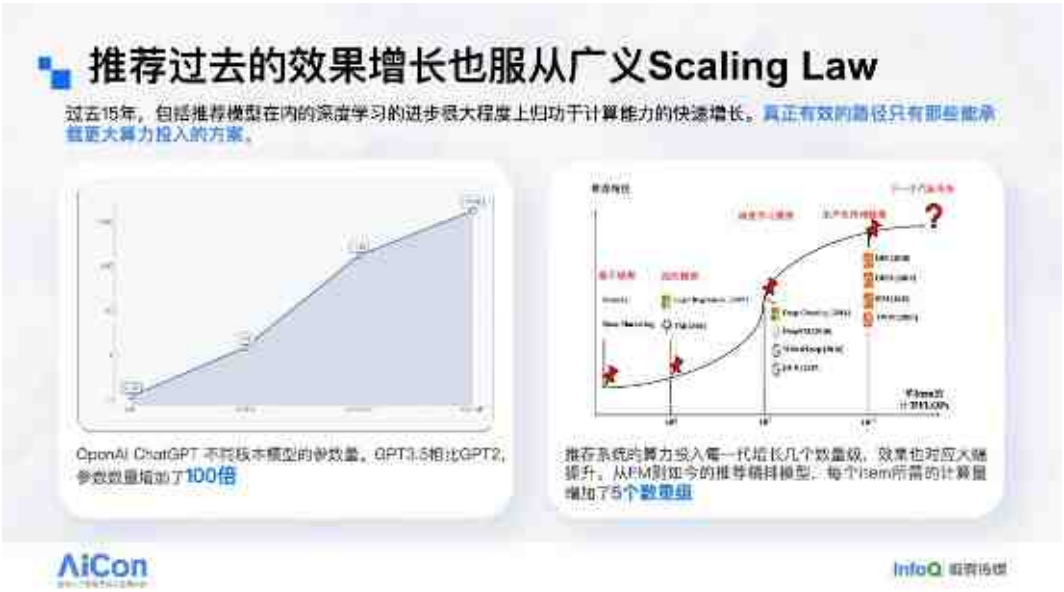
抛弃“级联”架构！快手OneRec用大模型重构推荐系统，服务成本降至1/10

作者 周国睿　编辑 李忠良、蔡芳芳

传统级联式推荐将召回、粗排、精排与重排割裂，反馈难以闭环，模块间不一致持续累积，令算力利用效率（MFU：总消耗算力/OPEX可购最大算力）长期偏低，成为技术与业务演进的共同瓶颈。快手以生成式端到端架构OneRec重构推荐链路，在核心场景实现“更大模型、却更低成本”的跃迁：服务成本降至原系统的1/10，并重新定义推荐系统的智能边界与效率标尺。快手科技副总裁、基础大模型及推荐模型负责人**周国睿**在AICon全球人工智能开发与应用大会上，系统解析了推荐系统的范式革新、OneRecV2的scaling定制优化，以及OneRec-Think的“生成—理解”统一进展，为AI原生时代的推荐提供可复制的方法论。

以下是演讲实录（经InfoQ进行不改变原意的编辑整理）。

传统推荐架构的规模化瓶颈与范式局限



传统推荐架构正成为技术与业务演进的瓶颈。从最早的规则引擎到线性模型，再到深度学习进入推荐、行为序列建模，每一次技术跃迁都伴随着可用算力的数量级提升。以FM到基于行为序列的精排模型为例，单个推荐item的计算量大约提升了5个数量级，同时带来显著效果增益。然而，近两年LLM突飞猛进，推荐领域却更多停留在精排等局部环节的微调上，整体创新进入瓶颈。如果我们仍将迭代重心局限于排序模块、围绕单点模型做细节优化，要获得数量级的提升将愈发困难。

能否直接借用LLM的技术栈与算力方案，重构一套端到端的推荐系统？

直接采用LLM架构来做推荐的想法，看似美好，但深入思考并不简单，根本原因在于LLM与推荐的在线服务场景差异巨大。推荐是典型的低延迟、高并发服务，需要实时响应海量请求；而在Chatbot等场景中，用户可以接受更长的等待时间，请求量也与推荐系统不在同一量级。

因此，当被询问“能不能用LLM来做推荐”时，团队往往会给出“想法很好但不现实”的反馈，常见理由有三点：其一，推荐模型本身已经很大。若将用于表示物料ID的

稀疏Embedding计算在内，推荐模型规模已达T级（万亿参数级）；其二，延迟要求极为严苛，难以留出足够的推理时间，推荐需要毫秒级响应；其三，计算成本高，在信息流等场景中，大模型推理成本可能难以被业务收益覆盖。

这些观点初看合理，但经不起推敲。首先，对“模型已足够大”的判断应聚焦“激活参数”。在在线推理中，推荐模型的激活参数远小于典型语言模型：以快手为例约250M；亦有同行baseline仅16M，近期已提升至1B，说明仍有充足的scale-up空间。若否认这一空间，推荐技术将难以演进——广义Scaling Law指出，在更低算力下难以获得更优效果。

其次，延迟无需过度焦虑，可通过离线、近线等机制加以解决。

更值得讨论的是成本：在推荐任务中引入大模型，在线推理成本能否由业务收益覆盖。与广义AI场景不同，推荐已是高度商业化、成熟的领域。只要引入大模型后效果提升、收益增加且可覆盖算力成本，就存在明确的商业驱动力——即便我们不做，也会有人去做。

实际上，仅从在线推理成本视角看，在多数场景下，使用激活参数约10B的模型做推荐，仍可维持很高的毛利。

以信息流场景进行测算为例。假设信息流业务CPM为20元，广告占比10%。则10,000次曝光的收入为20元。以DeepSeek模型为参照，20元可购买到“输入与输出各2M Token”的算力（以671B总参数、激活37B的大模型为例）。在该价格下，其理论毛利约84.5%左右。这意味着，当我们同样以“输入与输出各2M Token”的配置提供服务、并达到相当的性能优化时，10,000次曝光对应的20元收入下，毛利仍可超过80%。折算到单次曝光，可用的输入输出合计约为200 Token。

对于需快速响应的推荐服务而言，200 Token已能满足需求。上述仅为感性认识，下面给出更为严谨的测算。

仍以相同假设：千次广告曝光收入20元、广告占比10%，则10,000次曝光收入20元，单次曝光收入为0.002元。若希望保持80%的“计算毛利”（仅考虑计算成本），则可投入的计算成本为收入的20%，即单次曝光可投入0.0004元。

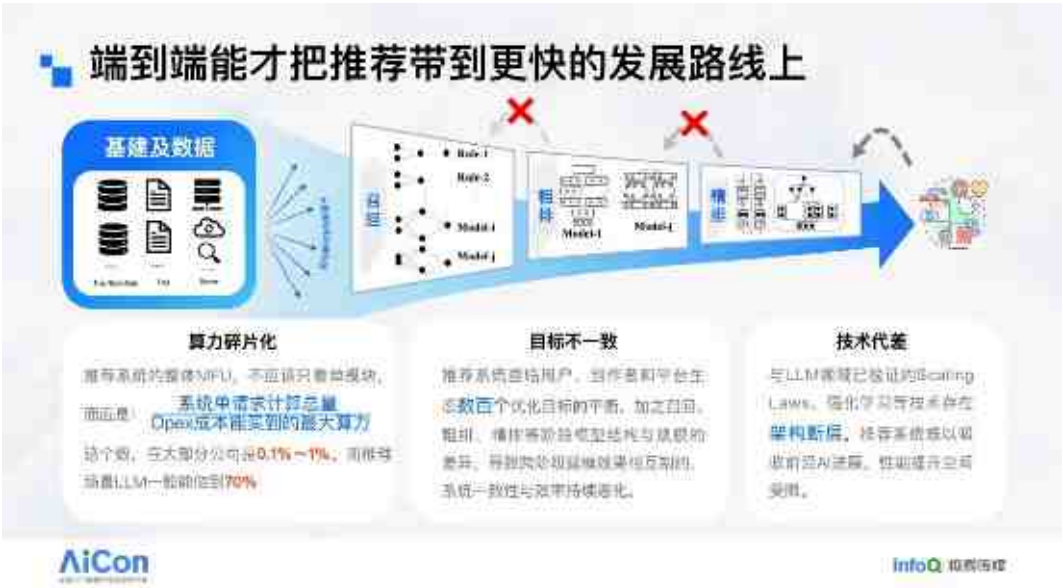
以H20为计算硬件，OPEX约为5.6元/小时（云服务商公开报价，若自建可更低）。在此价格下，单次曝光可投入的4/10,000元对应约15.2T FLOPs的计算能力。这里已计入算力利用率（MFU）40%。

若采用LLM模型完成推荐，其计算开销主要由两项决定：模型尺寸N与Token数D。设单次曝光服务消耗1,000 Token，此为合理假设。则单次曝光可用的计算量约为15.2T

FLOPs ÷ 1,000 ÷ ≈ 7.6B参数规模。

由此可见，只要业务满足“CPM 20元、广告占比10%”的量级，系统中使用约7.6B的模型进行在线服务在成本上是可行的。

上述结论与不少团队的直观感受存在差异，核心在于对MFU的假设不同。这里按40%计算，而传统级联式推荐系统（数据→召回→粗排→精排→重排）由于算力高度碎片化，MFU往往极低，可能仅为1%甚至更低。



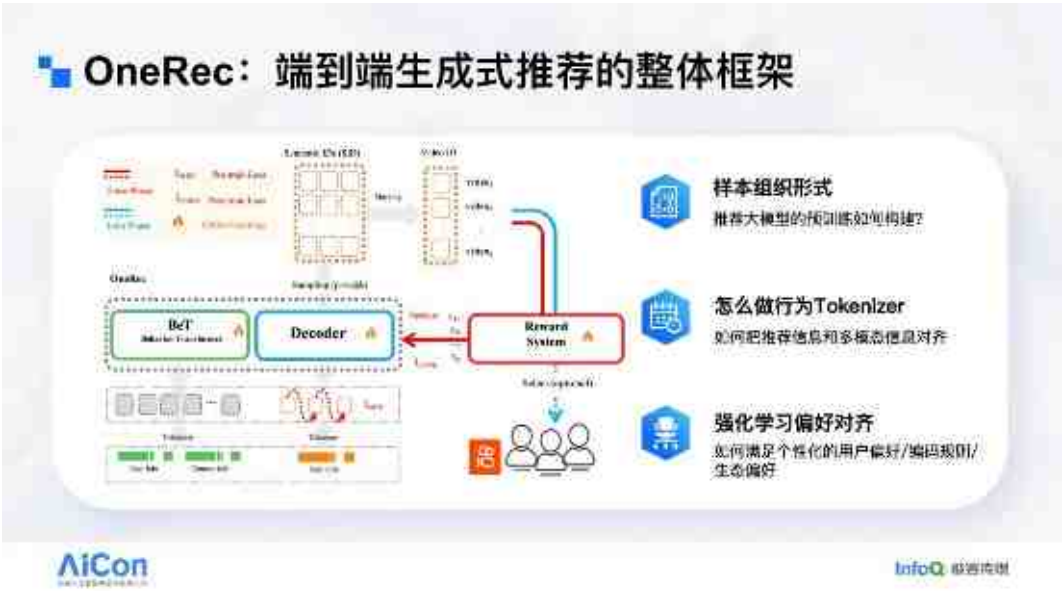
关于MFU的度量：在语言模型场景，可直接用“实际计算开销的FLOPs ÷ 硬件峰值FLOPs”计算；而在由多模块构成的推荐系统中，逐模块估算并不现实。

因此我们引入“广义MFU”概念：以系统在一天内产生的推理OPEX（人民币/美元）为基准，计算这些资金在市场上可购买的前沿GPU计算能力（T FLOPs）作为分母；以该系统当天实际发生的计算量（FLOPs）为分子；二者之比即为系统整体算力利用率。其含义是：增加计算量可以提升胜率，但无法突破模型的智能边界。

以LLM为例更易理解：其求解过程本质上是深度搜索，通过不断预测下一个Token，令已生成的Token成为后续的上下文，并激活模型内部知识，答案由此逐步收敛。典型的CoT现象表明：序列越长，在编码与数学任务上的可解性越高。基于这一认识，我们

将推荐系统设计为生成式范式。

OneRec生成式推荐框架



OneRec的总体架构如上图所示：首先，将原有稀疏ID通过多模态表征离散化为一串Token序列，记为语义ID（Semantic ID，SID）；其次，使用Behavior Transformer（类似Encoder）对用户行为序列及相关特征进行处理；随后以多层Decoder进行生成，得到Token，并反解回原始item ID或video ID，完成推荐。

借此，OneRec以单一模型覆盖召回、粗排、精排与重排，在全量候选空间中进行深层求解。实战中，强化学习带来显著增益，因此在训练与推理阶段均引入Reward System，以明确“优质推荐”的目标信号。相应地，内部协作也做了调整：模型团队聚焦Tokenizer、预训练方法与基础设施等底层技术；各业务算法团队负责界定“优质推荐”，并据此设计Reward System。当前两者耦合顺畅，迭代效率良好。

先说明Tokenizer的方案。OneRec将原始ID数据转化为类文本的Token体系。方法并不复杂：只要能为每个ID获得Embedding表征，即可采用离散化思路（类似VAE）将其离散为Token。关键在于表征质量：既要刻画内容本身（短视频“在说什么”），又要提炼与推荐最相关的信息（用户“真正关心什么”）。

以此例说明：多模态信息里会包含颜色等线索（如绿色葡萄）。但“绿色”未必影响用户决策，因此并非关键。若以该视频为查询，单纯从多模态内容相似度出发做检索，往往会返回其他“水果相关”的短视频——形似而神不似；用户看了“葡萄视频”，并不意味着想了解其他水果。

从协同信号视角看，例如使用线上“精排模型”的Embedding做检索，结果可能会倾向“挑选蔬菜”一类内容。这类相似性更多是“点击过A也可能点B”的共现表征，仍然偏表层。

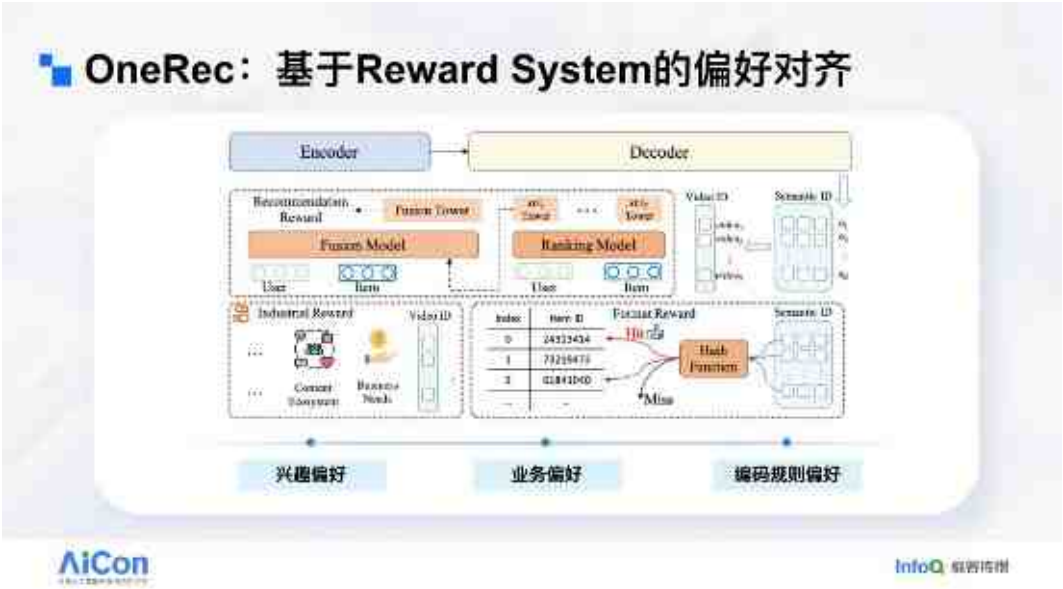
理想的做法是获得一个“既懂内容、又懂推荐”的Embedding。我们在线上使用的OneRec表征便能识别更深层语义：该视频的核心在于“农药污染/食品安全”，因此会检索到其他食品安全相关的短视频。

具体实现上，底层采用Vision-Language Model处理视频原始信息（抽帧图像与作者文本等）。由于输入为变长，上层先得到一段变长的Token序列（每个位置输出一个向量/Tensor），再通过Q-Former转换为定长向量，形成最终表征。该表征的关键：一是用VLM准确“读懂”视频；二是通过合理监督“教会”它对推荐有用。

我们引入两类Loss：其一是item-to-item，要求多模态表征空间中的相似关系与协同过滤视角一致（哪些更像、哪些不像），训练思路与Clip的对比学习相近，效率较高；其二是语义保持loss，在表征后接入一个Language Model，要求其根据该表征生成原始视频的细粒度描述，从而保留视频语义。

完成上述训练后，模型既能理解视频在“说什么”，又能抓住与推荐相关的关键信息。随后做离散化就相对简单。

这套Tokenizer的泛化性较强：即便线上系统数月不更新，也能编码最新上传的短视频，因为它仅依赖原始内容而非线上样本积累。同时，OneRec只要完成新视频ID的编码，就能够对这些新视频进行推荐。



接下来介绍强化学习。OneRec设计了三类奖励（Reward）：一类直接对应用户体验与兴趣偏好；一类体现业务偏好，即在保证用户体验的前提下满足生态约束（如创作者成长、营销等）；最后一类为编码规则约束。

先看兴趣偏好。传统系统通常针对点赞、关注、观看等大量目标分别建模，再通过手工或调参公式加权以选出最优结果。OneRec将其简化为统一的逻辑输出（Logic），对上述信号进行一致化监督，并将该得分反馈给OneRec，作为“用户喜欢什么”的直接指示。

其次是业务偏好。过去推荐系统常因产品/运营频繁介入，叠加众多业务目标而日益臃肿。传统流程需要先将“生态类”目标量化成可做A/B Test的指标，再让算法在各模块“加点东西”以优化该指标，长期导致系统冗余。端到端的OneRec则只需明确定义“业务—生态控制”的Reward：例如要控制“营销感”内容占比，只需界定何为“营销感”并设定相应Reward，将其纳入OneRec的奖励集合即可。系统能在天级完成响应，且所需样本量极少，约为1/1000。

最后是编码规则约束。系统最终生成一系列Semantic ID，并据此解码出待推荐的视频。类似LLM可能“说错话”，并非所有生成的Semantic ID Token都能映射到真实短视频。为此我们引入与LLM相似的“Format Reward”：只有能成功解码到真实视频的生成才计

为正确，避免无效计算，本质上是一种“指令对齐”。引入后，整体“合法率”显著提升。

目前，OneRec已在快手多项业务中规模化落地。以核心短视频推荐为例，在全量推送中仅以原线上系统推理成本的1/10，即在更低计算成本下使用更大的模型并取得更优效果。针对“是否仅替换并降级QPS不公平”的质疑，我们在全量QPS流量对比中同样获得更优结果，这表明快手的端到端推荐方案已在整体上超越传统级推荐，并在大量业务场景稳定运行。

在完成短视频场景后，我们迅速拓展至本地生活与电商等营收场景，呈现两点特征：其一，子场景上线后的提升幅度显著；其二，迁移效率高，从短视频到其他场景的迁移周期约一个半月。短视频落地实验中，我们设置了三组对照：基线组；在不降级缓存QPS条件下由旧系统全面承载的“计算膨胀”组；以及由OneRec承接的实验组。

结果显示，OneRec显著提升消费深度。快手为上下滑产品形态，在OneRec承载的流量部分，用户更愿意持续下滑，使我们能够插入更多视频与电商业化内容，从而获得更大的增量空间。

Lazy Decoder Only：推荐Scaling的定制

在第二版OneRec中，我们围绕Decoder-Only进行了定制化优化。早期版本采用Encoder – Decoder架构，但这一选择与不少大模型实践者的直觉并不一致：Encoder侧难以体现明显的Scaling Law效应。为便于理解这些优化，接下来先交代样本的组织方式以及最初采用该架构的动因。

许多人谈到“用大模型做推荐”时，第一反应是将用户历史行为序列拼成文档，将每位用户的历史视作一篇Doc，用“预测下一个行为”来完成推荐，这对应的是Decoder-Only架构。但在推荐场景中，这一做法不可行。

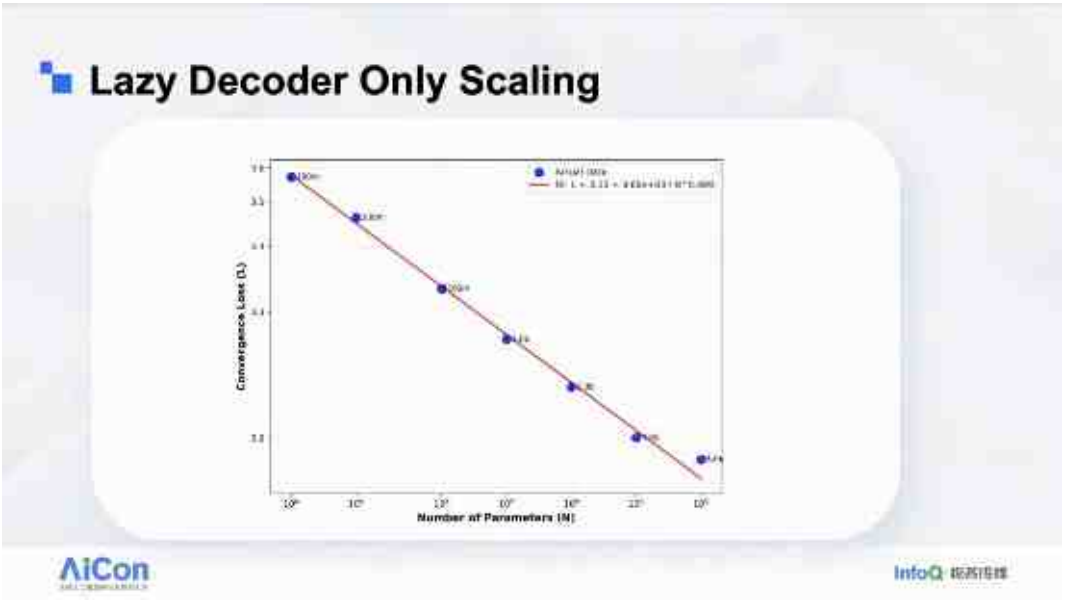
语言模型以Doc为单位训练，对全局时间排布并不敏感；而推荐必须严格遵守时间因果，这会造成严重的时间泄露。举例而言，用户U1的三次行为为A→B→C。按Next-Token Prediction训练完U1的Doc后，再去训练用户U2的Doc时，模型实际上已经“见过”发生在某时刻（如T4）之后的内容，再学习B→C的预测便会造成严重的时间泄露。这个问题我们仍在研究，目前尚无令人满意的通用解法；在时间泄露无法妥善解决之前，“用户Doc+下一行为预测”的范式难以成立。

因此，我们将用户行为序列拆解为曝光级样本：A预测B、B预测C，构造A、AB、ABC等样例。这样可确保训练严格服从数据的真实时间分布，这一点对推荐尤为关键。

代价是计算开销——语言中的单向Transformer可复用前缀计算，而我们对用户历史的处理会被重复计算，导致效率下降。

进一步分析表明，早期的Encoder – Decoder方案存在计算与监督的错配：训练资源主要消耗在对用户历史的编码上（约占近90%的计算），而该Encoder并无直接的Loss监督；主要监督信号集中在Decoder的两处——解码“用户更喜欢哪个候选”的目标，以及Token级的Next-Token Prediction。计算发生的位置与监督传递的位置错位，使得这部分参数与算力难以有效scale，投入更多算力也难以带来成比例的效果提升。

据此我们进行了优化：在当前版本中，几乎移除了用户历史编码阶段的大部分计算，仅保留一个轻量投影；将主要算力集中到后端的Decoder Transformer，使绝大多数计算用于“解码到底推荐哪个结果更好”。该优化带来约94%的训练资源节省；在等量资源下，过去可训练0.5B规模的模型，如今可训练到8B，模型规模显著放大。



如上图曲线显示，随着模型规模的扩大，我们观察到损失呈平滑且可预测的下降趋势，这与scaling law拟合结果一致。

OneRec-Think：全模态生成理解统一基座

尽管OneRec属于生成式大模型，但目前仍以行为数据为主进行训练。我们正尝试将

文本、图像与行为数据进行联合训练，目标是构建一个既能“说话”、又能“推理”的推荐模型。

为什么要这么做？语言模型具备强大的推理能力，随着CoT长度的增加，推理精度会持续提升。

其原理可以理解为：每当语言模型生成一个Token，便会激活相应知识并扩展上下文；生成的推理Token越多，被调动的知识与可用上下文就越丰富。在数学等任务中，这种“深度搜索”的过程尤为明显；相对而言，推荐并不具备这一特性。传统方法强调“宽度搜索”，对大量候选集独立打分，各item的评分彼此独立，难以通过追加计算让模型进一步变得更“聪明”。现有的OneRec也难以从根本上解决这一点。虽然采用生成式范式，但其生成对象是SID，难以直接进行CoT训练：一方面模型缺乏世界知识——未见过文本数据就很难理解“惊喜”“多样性”等概念；另一方面，基于人类难以直观解读的黑盒编码Token来设计CoT数据并不可行。

因此，我们转向将语言、视觉与行为Token进行联合训练，构建完全统一的基座，并在其上使用统一的语言模型完成解码与理解。

基于这一思路，我们将白色Token表示文本，橙色Token表示语义ID（可解码到具体短视频），绿色Token表示图像。整体结构与典型的Vision – Language模型相似：底层以

ViT等组件处理图像得到图像Token，SID则由自研Tokenizer产生，上层使用统一的LLM完成解码与理解。

这条路线可行性较高。不过，这并非“将多模态与行为数据简单混合再训练”即可奏效，关键在于构造足够多样的任务以支撑学习，类似多模态训练中同时提供图文对、Caption与VQA等任务的做法。对应地，我们构建了丰富的User QA与视频QA任务。

当前推荐效果好于预期，但尚未超越线上仅以行为数据训练的模型；不过，这一方向的可行性已得到验证。借助模型的文本生成与推理能力，还可以开展交互式推荐：用户提出需求后，模型据此推理并返回相应结果。

更有意思的是，当用户提示发生变化（如“心情不好，想看些不同内容”）时，模型的思考路径会发生明显转折：在理解既往偏好的基础上，依据新的提示生成与之匹配的推荐。目前仍有两点需要进一步验证：其一，实际推荐效果尚未全面超越线上体系，但已具备可比性；其二，当前成本偏高，不适合全量在线应用。可以确定的是，我们已实现“生成—理解”统一、覆盖多模态的模型原型。

需要强调的是，互联网中的大量行为数据并非文本或图像即可覆盖，例如购买具体商品、点击特定短视频、点选某个外卖等，这些行为均基于ID体系。由此，行为数据与语言、图像数据具有天然的融合空间，有望催生更智能的模型，并提供更优的个性化服务。

嘉宾介绍

- **周国睿**，快手科技副总裁，基础大模型及推荐模型负责人。深耕推荐系统/计算广告/大模型/计算引擎等领域，主导代表工作：DIN/DIEN/SIM/XDL，推动了深度学习在推荐系统的应用和序列建模发展。目前主要带领LLM/VLM等基础模型工作，开源工作有Keye多模态大模型等，以及下一代端到端推荐大模型OneRec研发。目前OneRec已经在快手短视频主场景、电商货架、本地生活等业务全量生产化，取得突出业务增益。

AI原生应用全栈可观测实践：以DeepSeek对话机器人为例

演讲嘉宾 夏明 编辑 Kitty

随着DeepSeek-V3 & R1火爆全球，基于大语言模型和AI生态技术栈构建的应用与业务场景与日俱增。AI原生应用架构从研发到生产落地，面临诸多新的挑战，包括模型选择、流程编排、评估分析等等。可观测技术可以帮助LLM应用开发及运维人员更好的优化模型性能、成本及效果。

在InfoQ举办的QCon全球软件开发大会（北京站）上，阿里云高级技术专家夏明做了专题演讲“AI原生应用全栈可观测实践：以DeepSeek对话机器人为例”，他以DeepSeek对话机器人为例，深入介绍AI原生应用架构的可观测需求、挑战与方案实践。比如DeepSeek为何频繁出现服务器繁忙？如何评估DeepSeek与其他模型的性能、成本与

效果差异？如何优化DeepSeek对话机器人的终端用户体验？等等。

以下是演讲实录（经InfoQ进行不改变原意的编辑整理）。

AI原生应用架构演进及痛点

AI领域的从业者对相关进展应该比较熟悉。例如，基础模型的快速发展，尤其是近两个月，Deepseek和阿里千问大模型等在国际上取得了领先的竞争力。在应用方面，目前比较热门的有Dify等应用编排和应用平台、LangChain编排框架以及MCP生态，它们都迅速融入了大模型生态。

从大模型的应用形态来看，主要有三种：一是相对简单的对话机器人，直接调用基础模型或加上RAG领域知识库，可快速提供对话服务，应用场景较多；二是领域化的编排，如Copilot智能助手；三是最近比较有代表性的多Agent协同，真正实现任务的规划、编排、生成、执行等一系列流程，更接近人类的预期。

在蓬勃发展的生态之下，也存在一些AI领域的核心痛点。首先是基础资源问题，例如GPU卡价格昂贵，如何更好地利用底层资源；其次是模型推理问题，模型推理需求可能是模型训练的百倍甚至千倍以上，推理速度慢，还存在安全、隐私、合规以及恶意攻击投毒等风险，需要进行防御和评估；第三是成本问题，尤其是MCP Server出现后，存在“token黑洞”现象，其内部实现不可见，恶意访问会导致反复调用大模型，快速消耗token额度。

阿里云AI全栈可观测架构

针对这些问题，提出了一个AI原生应用架构的方案。首先是用户终端，包括移动端和Web端，经过网关后，核心是应用模型层，例如Dify编排平台，上面可能运行多个语言模型应用。这些模型应用会调用向量数据库、模型服务调用缓存以及本地私有化或外部提供的MCP工具等。AI网关，可根据问题复杂度自动切换或路由模型，平衡服务性能和成本。再后面是模型服务层，可调用托管模型或自研自建模型，目前很多企业都在构建自己的模型服务。针对这套架构，总结出三个核心观测诉求：一是AI全栈统一监控，关注各层之间的动态，如模型性能、token成本消耗等；二是端到端模型调用全链路诊断，从终端用户发起问答对话到后端系统流转，找出瓶颈并调优，这给传统Tracing系统标准

带来新挑战；三是对模型生成结果的评估，这是提升生成质量的关键，需要探索自动化评估方法。

AI全栈统一监控分为几层：用户业务层关注用户体验，如终端性能卡顿，以及SSE流式问答响应等新挑战；模型应用层关注推理响应耗时，如首包响应时间和平均吞吐量等指标，以确定性能瓶颈；外部工具层涉及网关、缓存、对象存储等；模型服务层观测不同模型的效果和成本利用率；AI Infra层可在K8s上托管模型或直接调用GPU资源。

在一个典型的LLM聊天机器人的应用架构中，从用户请求到流转，还包括防御词检查、领域知识库外联等环节。传统链路观测视角难以满足算法从业者和模型平台开发者的诉求，因为他们更关注模型调用、embedding、retrieval等LLM层面的内容。针对不同角色和场景，需要定义新的指标，与社区共建，尝试定义新的GNI领域语义化能力。

模型生成结果的评估对现有研发运维体系是重大挑战。传统质量保证方法如黑白盒测试在语义结果上难以界定对错。针对新的语义响应，可采用评估模板，对用户输入输出进行评估，包括质量效果、安全性风险、用户意图提取、情绪等，这些都是新挑战，团队也在进行相关探索，如语义特征提取、评估自动化等。在阿里云视角下，从AI应用到大模型、AI PaaS、容器和智算基础设施，提出了一个整体解决方案。

大模型应用可观测技术剖析

在大模型应用的可观测性方面，我们需要关注一些与传统不同的指标。除了传统的黄金三指标（RED指标），大模型中还会出现一些新的指标，例如TTFT（首次首包传输时间）、TPOT（平均吞吐量）和Token per Second（每秒token数）等。这些指标从三个维度来观测模型的效率，每个指标都代表了模型在不同生成阶段可能存在的问题，比如在prefill阶段或decode阶段。此外，还需要关注token成本以及评估生成内容的毒性、幻觉等问题，这就需要我们定义新的指标来描述这些问题。

对于LLM应用的领域化Trace语义，大模型生态中的会话（session）概念变得更加重要，因为多轮对话的场景较为常见。例如，用户登录APP后可能会先问一个问题，然后不断追问，这就构成了一个会话，而每轮对话背后又会发生多次请求，即多条Trace。系统中不同Trace的流转被我们定义为LLM span chunk类型。例如，需要考虑如何编排整个流程，包括embedding、向量检索以及调用模型服务等环节。针对这些不同类型的Trace，

它们都有自己特有的字段语义。总的来说，我们不能简单地用传统微服务CPU架构的视角去套用到大模型GPU架构上，而应该以更开放的心态重新理解这一套新架构。

在具体实现方面，开源的迭代速度相对较慢，而阿里云通过自研探针进行高质量数据采集。阿里云的探针底座基于业界主流开源生态，并非另起炉灶。团队还计划将相关工作回馈社区，进行开源。以Python探针为例，它可以通过无侵入的方式进行埋点，无论是在Dify平台、自建模型服务的vLLM框架还是SGLang框架下，都能采集到对应的性能指标数据和链路信息，包括Llama index操作逻辑、prompt信息以及外部调用信息等。

阿里云自研探针与开源的OTel Python探针存在一定差异。例如，自研探针支持更多埋点框架，如最新的一些vLLM、SGLang以及正在做的MCP等，这些在开源领域是略微领先的。此外，阿里云探针会定义更丰富的埋点，因为开源探针为了兼容不同生态，可能会受到一定限制。例如，OpenAI有自己的tracing标准，与现有的OTel标准差别很大。阿里云希望尽可能兼容各种主流实现，虽然在标准实现上会有些差异，但在埋点上会丰富数据采集，最终归一为一套生产可用的实现。同时，阿里云探针还会针对多进程协程等细节进行优化，以提高稳定性和性能。在大模型数据处理链路方面，阿里云既支持自研探针，也兼容主流开源的数据集成方案。无论客户端类型如何，都可以实现统一上报采集和数据加工处理，最终提供一个稳定、高性能的服务。

在大模型领域，流式场景的LLM Span分段采集与合并是一个比较特殊的新问题。虽然流式场景本身并不新，比如Websocket也有流式传输，但在大模型领域，这个问题变得尤为重要，需要重新审视。针对流式场景，开源的OTel社区也在讨论相关问题。如果不将流式数据分段采集和上报，而是等到完整后再上报，会对探针客户端造成很大压力。因为一些极端的大模型调用上下文可能有几兆甚至几十兆，持续时间可能达数小时甚至超过一天，这与传统意义上的请求有很大差异。为了解决这个问题，可以将流式场景分chunk进行分段上报，但简单地分段存储会导致后续数据分析困难，比如算法人员难以找到分散的chunk信息进行模型上下文评估，尤其是进行批量回归时。因此，最终方案是与社区提案相近的分段数据上报后在服务端重新合并为一条记录。这其中涉及很多细节，如何时上报、如何缓存、高性能实现以及是否有限制（如截断）等，但整体思路是为了平衡客户端性能、实时性以及数据分析评估的易用性，采用分段采集和服务端合并，最终持久化为一条记录的方案。

在国内，Dify平台使用较为广泛，团队也在实践中使用Dify原生的可观测能力。但在使用过程中发现，Dify原生的可观测性以及探针存在一些问题，比如如何实现全多维度分析视角。Dify本身是一个大应用，上面会运行多个业务LLM应用，需要评估每个业务LLM的成本消耗和性能，这更符合生产级部门间的协同需求。而Dify只是AI全栈调用链路中的一环，它如何与外部依赖、模型服务层以及AI网关等上游层协同进行全量观测，是Dify框架本身无法完整回答的。通过整体方案可以解决这一问题。

在VLLM/SGLang推理性能可观测实战方面，比如在阿里云的PAI EAS上部署模型服务，或加速推理性能时，会遇到一些问题。例如，当发现Deepseek模型服务请求超时时，可以通过Request ID检索到相关联的Trace ID，通过端到端分析定位问题是否出在模型推理服务本身，还是在前置端侧或应用层。如果定位到模型推理问题，再观察相关指标，如首包延迟正常可排除prefill阶段问题，TPOT指标正常可排除Decode问题，最终发现是请求队列问题，通过调整队列大小解决了问题。这就是推理层面的一个实践案例。

基于LLM实现模型生成结果的自动化评估时，实践过评估的同学可能会面临一些问题。例如，现有方案中，用户输入的embedding过程和向量检索过程可能需要调用两次服务或两个组件来完成。即使通过评估语义检索查出了一些结果，但这些结果可能无法完全满足生产级的查询需求，还需要结合传统的关键词进行混合检索。针对这些问题，阿里云格文斯团队的实践是提供内置的评估服务。由于trace数据天然记录了整个模型调用的上下文过程，包括单次LLM请求的prompt和response，以及全链路每个阶段的完整上下文，因此可以基于这些数据快速提供开箱即用的内置评估模板。用户可以选择针对某一类用户或场景的模型调用进行质量检测、安全检测或意图提取等操作。然而，内置模板的上限相对较低，例如算法补全优化从百分之十几提升到40%后，若想继续提升就需要不断调优，丰富各环节的关键特征并进行微调，这就需要支持自定义扩展能力。从工程实践效率角度出发，还需要解决如何将embedding与retrieval过程结合，以及如何将语义检索与关键词、顺序扫描的混合检索结合等问题，团队提供了一些更好的工程化能力来简化开发流程。

MCP目前非常受欢迎，它主要解决了协议标准化的问题。之前function call虽然也能解决类似问题，但没有标准化，针对不同实现需要提供多种实现方式。如果未来OpenAI等遵循统一标准，那么只需要定义一套MCP Server即可。例如，阿里云会开放相应的MCP Server和公共工具，帮助用户简化构建智能运维、智能体的流程。用户可以直接集

成阿里云的智能诊断能力，而无需面对多种不同协议。MCP解决了n乘m的集成问题，但也引入了新问题。由于增加了Client和Server之间的交互，调用链路变得更加复杂。例如，Minus有上千个tool，在这种复杂场景下，调用链路的优化和定位变得非常困难。因此，针对MCP Server背后的观测以及client端的观测能力变得尤为重要，可观测性是解决这些问题的有效手段，团队在这方面也做了很多工作，并将陆续发布相关成果。

AI In可观测实战

阿里云可观测团队在AI应用方面的实战主要分为两个部分。第一部分是提供智能助手，例如Copilot智能助手，它适用于垂直领域，如分析复杂的大模型trace。传统领域也存在类似问题，很多客户不会用trace，这是一个行业难题。为了解决复杂度问题，团队提供了Copilot智能助手，用户点击“魔法棒”后，它能判断trace是否有问题，是错误问题还是性能问题，如果是性能问题，会指出是哪个组件导致的，背后原因是什么，以及如何优化。这种方式更易理解，有助于推动行业的广泛应用。类似地，在性能优化场景中，如CPU热点、内存OOM等问题，除了复杂的火焰图分析外，还需要结合死锁数据、资源管理配置、Pod规格等信息。针对这些问题，团队也通过Copilot的方式解决，现阶段倾向于用workflow方式提高确定性，规避模型幻觉问题。

第二部分是Problem Insights智能洞察，它解决的场景更复杂。当企业服务出现可用性风险（故障）时，团队希望通过自动发现故障、给出传播链事件流推理过程、根因分析，甚至结合企业运维的MCP工具实现故障自愈，构建智能运维体系。这个场景的复杂性使得编排难以应对，团队倾向于通过Agent的方式尝试回答，内部会涉及多种工具。

目前，Copilot已上线三类功能。第一类针对日志服务，偏SQL，可帮助用户进行自然语言转SQL或SQL分析，优化SQL语句。第二类是trace分析，当trace出现慢、错或异常时，它能指出入口服务报错的原因，如下游接口调用数据库SQL语法问题，并给出SQL优化建议。这背后涉及分析trace结构、识别特定领域问题、关联多模态的profiling日志和metrics等信息，目前通过workflow方式编排。第三类是profiling，实现了常态化持续性能剖析，可随时回溯对比发布前后的差分火焰图，定位性能问题代码。下一步，团队计划关联发布变更的Pod镜像版本，甚至探索Git提交的commit信息及责任人。

Problem Insights智能洞察主要面向故障应急场景，目标是实现真正的智能洞察。它

可以智能检测系统核心问题，也可关联告警事件触发洞察。例如，当检测到应用接口性能退化时，它会展示推理过程，先进行根因定界，判断是服务自身问题、下游问题还是基础环境问题；然后进一步分析是资源问题还是代码问题，以及对上游业务的影响。对于SRE或运维人员来说，了解故障传播链、相关事件流、影响面，结合多模态数据给出根因和解决方案，可以极大地简化运维操作，提升企业可用性，降低MTTR时间。

未来规划与展望

在未来规划和展望方面，首先，可观测的核心问题依然是采集更多高质量的数据，包括对新协议的覆盖。这是首要任务，否则将面临“无米之炊”的挑战。其次，无论是MCP的生态还是整个端到端的生态，未来企业构建统一可观测平台时，仅仅做到数据存储是远远不够的。例如，日志存储在日志系统，指标存储在监控系统，这种分散的数据存储方式无法满足需求。团队正在尝试解决如何真正理解这些数据，打破数据孤岛，构建数据之间的实体关系连接。以Tracing为例，它可以解决端到端请求流量的精准连接，只要遵循同一协议并透传Tracing ID即可。但一旦涉及非调用环节，比如某个模型应用部署在K8s Pod上，而该Pod两分钟前发生了容器镜像版本更新，且该镜像对应某人提交的Git commit，如何解决这种更大范围、更广义的数字世界连接问题，是团队关注的重点。团队希望通过构建实体拓扑来解决这一问题，这不仅包括大模型的实体拓扑（会优先构建），还包括如何构建整个数字世界的完整实体拓扑，这是团队未来需要回答的核心问题。第三，随着大模型的广泛应用，语义类问题将日益突出，如质量、安全、意图等。团队将持续优化模型评估流程，降低使用门槛，同时支持用户进行自定义扩展。最后，团队将持续迭代自身的可观测智能体，借助AI发展浪潮，通过AGI提升行业和社会的生产力。

嘉宾介绍

- **夏明**，阿里云高级技术专家。在链路追踪、应用可观测领域从业近十年。先后负责阿里集团EagleEye、阿里云ARMS相关产品设计与研发。GitHub稳定性专栏StabilityGuide发起者。

从模型到智能体：Snowflake的企业级Agentic AI工程化之路

演讲嘉宾 杨扬 编辑 Kitty

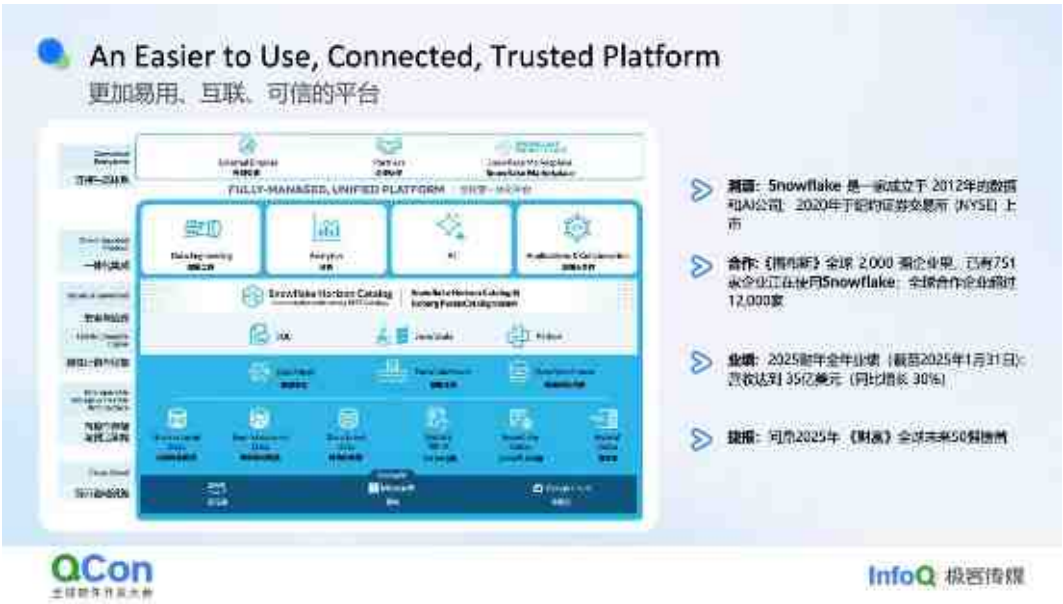
随着大语言模型迈向Agentic AI，企业在从功能验证到规模化落地的过程中，面临安全、效率与信任等多重挑战。没有坚实的数据基座与系统化的工程方法，AI难以真正转化为业务智能。Snowflake亚太及日本地区解决方案工程副总裁杨扬在2025 QCon全球软件开发大会（上海站）分享了通过Snowflake的研发实现企业级Agentic AI的部署，从而重塑智能生产力，实现从“大模型”到“可控智能体”的跃迁。

以下是演讲实录（经InfoQ进行不改变原意的编辑整理）。

如今，AI大语言模型和Agentic AI成为热门话题，然而从功能验证到企业级实施，仍是一条漫长且充满挑战的道路。今天，我将与大家分享我们是如何通过Snowflake的研发

实现企业级Agentic AI的部署，从而重塑智能生产力的。

Snowflake是一家成立于13年前的公司，基于公有云构建了一个完整的数据和AI平台。我们致力于帮助客户处理各种不同类型的数据，支持多种语言进行开发和数据建模，并为商业用户提供数据工程、数据分析、AI以及数据共享方面的应用。目前，我们在全球拥有超过12,000个企业级客户，其中超过50%的客户使用了我们的AI功能和产品。众多财富2,000强公司也在使用我们的解决方案，而且我们最近还被提名为2025年财富未来50强的第一名。



接下来，我将从五个方面分享Snowflake在AI研发中的支柱，这些支柱如何支撑我们的研发工作，并为客户带来最大价值。首先，企业级Agentic AI的部署需要一个智能的编排与工具使用系统。该系统能够在不同环境中串联各种工具，将任务分配到不同的工具中，从而准确、安全地读取结构化和非结构化数据。在实现不同类型数据的处理、读取和分析后，我们需要为企业级用户提供可观测性和信心。因为在企业级AI部署中，用户的信心至关重要。最后，我们还要考虑系统优化，在实现各种功能、处理不同数据后，如何达到高效的效果，使企业能够使用AI并将成本控制在可控范围内。

核心支柱一：智能体编排

第一个话题——**智能编排**。首先，我想请大家思考一个问题：中国的高铁系统为何是世界最先进的？仅仅是因为速度快吗？其实并非如此。尽管上海的磁悬浮列车速度更快，但高铁之所以领先，是因为它拥有一个广阔的网络系统，如四横四纵、八横八纵的布局，以及强大的中央控制系统。正是这种系统让乘客能够安全、高效地到达目的地。因此，速度并非唯一关键，一个安全高效的中央控制系统和编排系统才是实现最终目标的关键。

在Snowflake的AI研发中，第一个支柱就是智能编排。在企业级Agentic AI部署中，我们常常面临一个挑战：各种工具分布在不同的系统中，即使在同一平台内，也存在多种功能供开发人员调用。这就需要一个智能体，能够自动拆分任务，规划任务的执行过程，并将其发送到相应的工具中执行。在执行过程中，用户会持续提供信息、追问问题，而智能编排系统则能够优化和调整执行计划。在Snowflake平台上，我们提供了多种工具和功能，如Cortex Analyst和Cortex Search等。在这些工具之上，我们的智能编排系统可以自动拆分用户的需求，并将其发送到不同的工具中去。

让我举一个例子来说明这个过程。假设在一个用户终端，用户发送了一个请求，比如“为什么4月5号我们的仪表盘数据出现了下降？”这是一个非常简单的问题，但我们

的智能编排系统会将这个任务进行拆分，通过多级推理将其分解为几个部分。首先，系统会确认仪表盘数据的内容，判断它是否真的出现了下降，并查看其正常水平以及4月5号的具体数值。接着，系统会将日期作为一个关键因素进行分析，最终得出结论：4月5号数据下降是因为那天是周末，业务流量较小。虽然这只是一个简单示例，但在企业实际使用场景中，许多问题都需要分解为多个步骤，并分配到不同任务中去解决。

在我们的Snowflake平台上，各种工具通过智能编排系统接收任务，这要求系统具备高度的可扩展性。因为今天的应用场景可能是商业运营，明天可能就会转向医疗领域，智能编排系统是否能够灵活扩展、无缝连接到其他系统，是我们研发过程中必须考虑的关键问题。

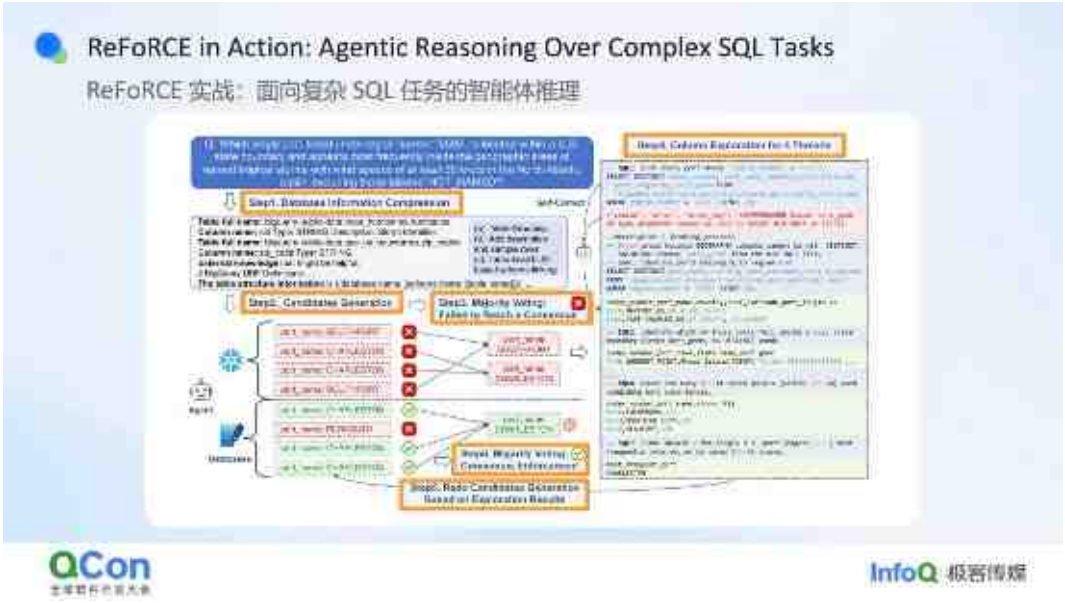
这其实也是我们研发理念的一部分。我们团队曾发表过一篇学术论文，扫描二维码可以阅读其内容，这篇论文展示了智能编排系统的可扩展性。在论文中，我们通过一个医疗诊断的例子来说明这一点：通过与智能编排系统的整合，阿尔茨海默病的预测准确率被提高到了93.26%。这种整合是无缝衔接的，这不仅验证了我们系统的灵活性，也展示了它在不同领域应用的潜力。

核心支柱二：结构化数据智能

现在，我们进入第二个支柱。在这里，我想再给大家提个问题：有多少人经常去超市购物呢？通常，我们会从配偶或父母那里拿到一张购物清单，然后去超市采购。这对我来说是家常便饭。购物本身其实很简单，但让我感到焦虑的是，每次进入超市后，我该如何从长长的购物清单中找到自己想买的东西？又该如何避免把一桶两公斤的食用油从超市入口一直扛到出口呢？很多时候，在进行查询时，任务的难点并不在于找到数据或获取数据，而在于如何找到最优的路径来获取我们需要的东西。这便引出了我们的第二个重点：**结构化数据的智能驱动**。

如今，生成SQL语句对每一个大模型来说都已是轻而易举的事情。但在企业级环境中，如何正确解读那些并不完全清晰的用户提问？如何在海量数据库中，在复杂的数据库编排系统里找到准确的数据，并进行验证和问题澄清？这是我们研发团队推动的一项创新，我们将其命名为ReFoRCE，这是一个Agentic系统。它可以通过一系列自动化操作对数据库的Schema进行压缩和优化，并通过一个自动投票系统对比不同生成的SQL执行结果，最终达到最优效果。

让我再举个例子来解释我们的ReFoRCE机制是如何执行的。假设现在有一个问题：它要求找到某个特定的港口，这个港口位于美国大西洋盆地的某个特定地点，并且该港口的热带风暴风速要超过一定数值，同时编码不能是“no name”。我们需要找到这个特定港口的名称。当Snowflake的ReFoRCE机制接收到这样的询问后，它会先将其拆分，提取关键细节，比如需要查询的元数据、涉及哪些表格和列。然后，系统会自动生成关于这些表格和列的描述，并基于这些信息生成一些候选SQL语句进行执行和对比。有些初步生成的SQL可能会失败，但这没关系。我们会进入下一步，即column exploration，我们会去探索数据库中哪些列与提出的需求相关，并通过一系列拓展和对比，最终找到正确答案。虽然大家可能觉得结构化语言查询很简单，无非是生成一个SQL并执行，但实际上，快速、安全、有效地找到所需的表和列，并进行多方面对比以达到最优效果，这里面蕴含了许多创新。



通过ReFoRCE机制，我们可以将SQL执行效率提高超过20%。PPT右边的Spider Lite是一个业界公认的用于评测Text-to-SQL精准度的基准。在最近一次（9月底）的排名中，我们位列第二，仅次于清华大学的一项研究成果。

AT&T案例研究：推动Text2SQL进阶

我们以AT&T为例，探讨其如何利用Snowflake及其他工具部署企业级的Text-to-SQL解决方案。AT&T是我们全球的一个重要客户，其应用场景极具代表性。

首先，AT&T是一家拥有超过14万名员工的大型电信公司，其中超过10万名员工通过AI部署显著提升了工作效率。公司已经部署了超过90个微调的小语言模型和410个智能工作单元，每天的AI API调用量超过4.5亿次。此外，其生产环境中的RAG应用已部署超过71个。仅从代码编程效率提升这一维度来看，AT&T已经实现了20%的效率提升。这表明，对于超大型企业而言，AI的应用能够极大地提升企业运营和业务性能。

在部署Text-to-SQL时，AT&T采取了多管齐下的策略。他们提出了9个关键点，但其中许多优化工作集中在数据库层面，而非单纯的大型语言模型。首先，他们对数据进行画像，快速、自动且高效地生成元数据描述。此外，他们还对数据库Schema进行去重、连接，并验证多重执行结果的一致性。这些措施不仅优化了数据库结构，还显著降低了AI使用过程中的Token消耗。

具体来说，AT&T在数据库优化方面取得了显著成效。例如，在进行Schema去重之前，他们的Token消耗量约为700万，而通过优化后，这一数字降至15.6万。这一优化带来的收益是巨大的。以时间序列数据库为例，原始设计中每天的数据存储在一个新表中，这导致用户查询跨多个时间区域时需要读取多个表并进行join或union操作，从而消耗大量Token。为了解决这一问题，AT&T将时间序列数据库中的每日表合并为每月或每季度一个表，这一简单操作极大地提升了效率。此外，AT&T还对数据库的语义信息进行了压缩。他们将表名、列名、列类型及其描述进行向量化处理，使得AI在搜索时可以直接在向量数据库中查询相似度，从而更快速地找到对应的数据库数据。

我们发布的ReFoRCE机制也通过学术论文的形式展示了其创新性和实用性。大家可以通过扫描二维码详细阅读该论文。

核心支柱三：非结构化数据智能

接下来，我想谈谈第三个关键支柱——非结构化数据。在企业级Agentic AI部署中，

如何高效地获取非结构化数据是一个重要问题。非结构化数据的关键在于最大限度地避免大模型产生幻觉。在日常生活中使用AI时，偶尔出现的幻觉可能影响不大，但在企业级应用中，幻觉可能导致灾难性的后果。

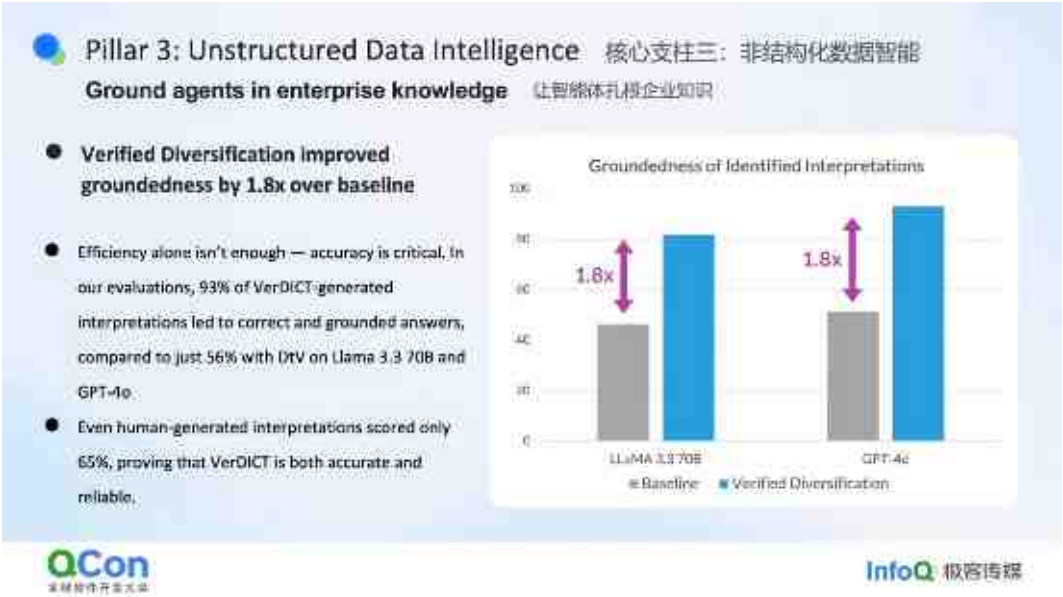
为了解决这一问题，我们Snowflake的研发团队提出了一种新的模式，我们称之为VerDICT，即Verified DIversification with ConsolidaTion。在所有大模型中，解读问题时通常会进行多元化处理，从多个不同角度理解用户提出的问题，然后再生成答案。这虽然是大模型的一个优势，但在企业级应用环境中，我们需要对每一步多元化解读进行验证。

首先，我们引入了Retriever: relevance feedback机制。当用户提出一个问题后，我们的软件会核实问题中需要读取的数据以及对问题的解读是否准确。例如，在下图例子中，用户问了一个简单的问题：“什么是HP？”大模型可能会解释为HP是一家科技公司，也可能会解释为“哈利·波特”或马力（Horse Power）等等。但在企业级应用环境中，通过Retriever: relevance feedback机制，我们可以迅速排除“哈利·波特”这一无关选项，因为它与企业的应用场景无关。

其次，我们还设计了Generator: answerability feedback机制。即使大模型只读取相关数据，而不是基于互联网知识或自我想象，它生成的答案是否直接回答了问题仍需验证。以同样的例子为例，如果用户问“什么是HP”，即使大模型找到了与HP相关的文档，但

生成的结果却详细描述了某一型号的HP产品，这显然没有直接回答问题。通过这一步骤，我们又排除了那些低回答性的答案。

通过这种双重验证的VerDICT机制，我们显著提升了基于非结构化数据的大模型处理结果的精准度。在应用VerDICT机制后，我们的精准度达到了93%，远远超过了使用Llama 3.3或GPT-4处理非结构化数据的结果。即使与人工处理的结果相比，我们的精准度也远高于人工的65%。



核心支柱四：可追溯性与可信度

接下来，核心支柱四，我们探讨AI的可追溯性与可信度。也就是说，我们是否能够正确地追溯AI的执行过程，并对其产生信任？让我用一个简单的例子来说明：当我们去看医生，抱怨肩膀疼痛时，医生可能会给出各种建议，比如开药、手术或进行力量训练。然而，我们如何信任这位医生？我们如何知道他的推理过程是基于科学而非随意猜测？他的资质是否可靠？他的建议是否合法合规？这些问题同样适用于企业级Agentic AI的部署。

在企业级环境中，我们需要考虑多个方面：准确性（Accuracy），即AI给出的答案是否精准；有效性（Effectiveness），即答案是否能在可接受的时间范围内提供，并且是否能将成本控制在合理范围内；最后是信任（Trust），这至关重要。尽管大模型带来了

创新和效率提升，但如果其不满足合规性或道德标准，就无法在企业级环境中应用。

那么，Snowflake的解决方案是什么呢？在准确性方面，我们提供了一个端到端的评估过程。无论任务被拆分到结构化数据查询还是非结构化数据查询，我们都能将每一步的执行结果完整地展示给开发人员或用户，以便他们进行观测和使用。此外，我们允许用户将同一个问题发送到不同的环境、不同的大模型，甚至是同一个大模型搭配不同的参数设置中，进行明确的对比分析。在可追溯性方面，我们支持OpenTelemetry这一开源格式。无论是直接使用Snowflake平台，还是在此基础上进行二次开发，整个执行过程都是可追溯且可信的。

虽然是平台设计的一部分，但我们同样建议，在进行企业级安全AI部署时，应将内容相关性、基于数据的可靠性和答案相关性准确地呈现给开发人员和用户。当一个问题被提出时，首先需要检测大模型读取的文档或数据是否与问题相关，是否与企业部署环境相匹配。回想一下之前提到的“哈利·波特”例子，生成的结果是否基于实际数据，而非模型的想象？最后，答案是否真正针对问题，而不是答非所问？这些要素都是我们在AI的可追溯性和可信度中需要重点关注的部分。

核心支柱五：系统优化

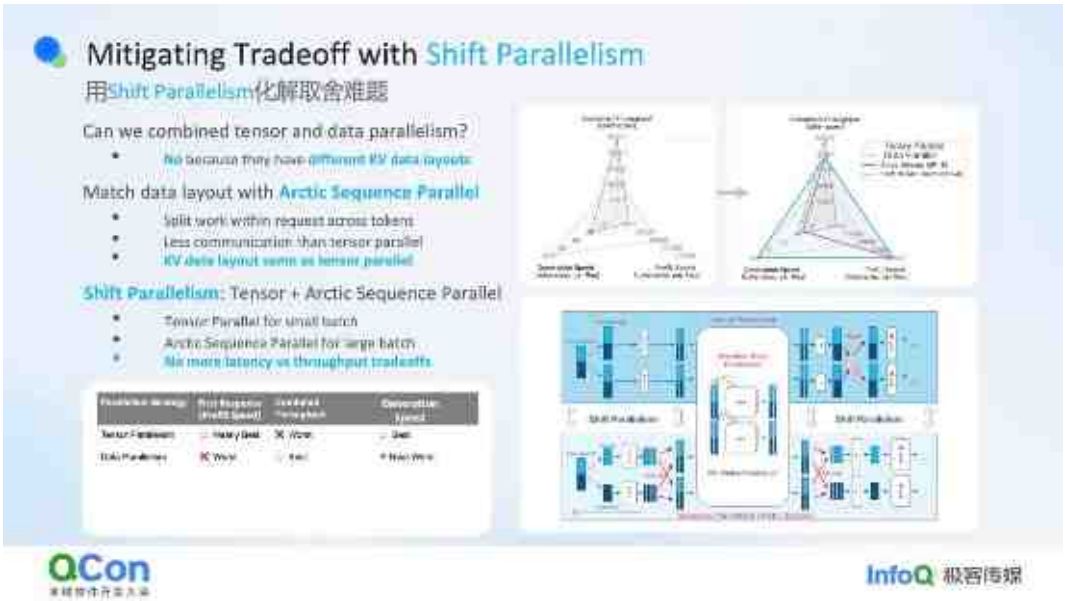
接下来，我们将探讨**系统的调优**。目前，Agentic AI的企业级部署已经能够实现任务

拆分、高效数据获取、数据分析以及安全管理，用户也能够信任大模型。然而，在企业级部署环境中，是否真正可用还需要从多个角度进行考量。这就好比评价一家餐馆是否优秀，需要综合多个维度：服务响应速度是否足够快？菜品上桌速度如何？以及是否具备合理的吞吐量，能否同时接待多位客人？只有将这些因素综合起来，才能全面评价一家餐馆的水平。同样，在企业级大模型部署时，我们也有三个关键点需要考虑：响应性（Responsive）、结果生成速度（Fast Generation）以及最终的吞吐量（Throughput）。响应性体现在用户与大模型交互时，模型需要多长时间生成第一个Token。结果生成速度则衡量模型生成完整答案的延迟。而吞吐量则取决于系统能够支持的用户数量，以及在部署大模型后是否超出预算。这三点都至关重要。

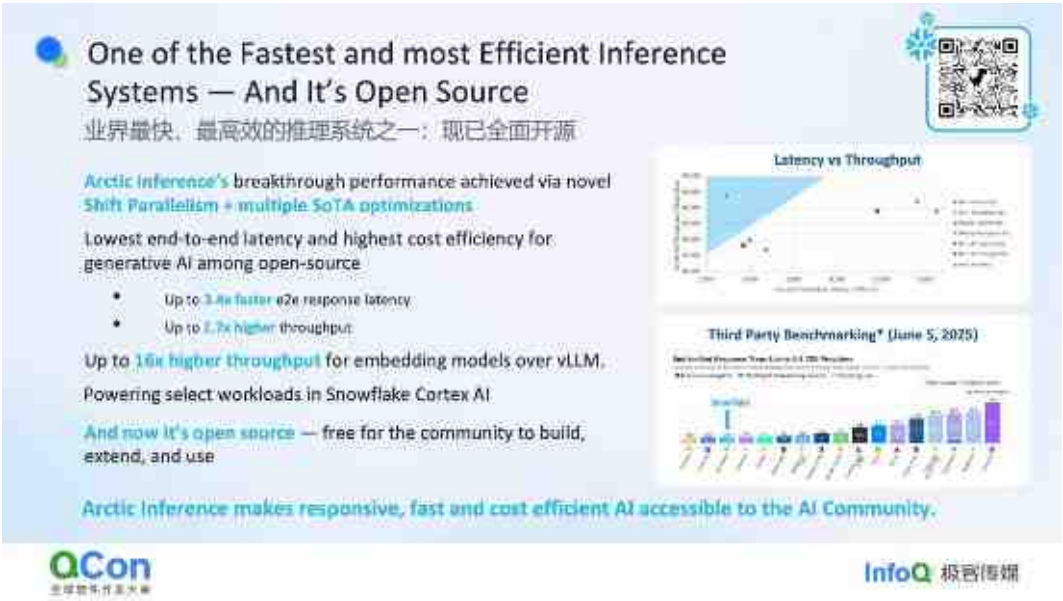
目前，行业内关于模型优化的机制主要有两种主流方式：张量并行（Tensor Parallel）和数据并行（Data Parallel）。它们各有优缺点。张量并行在首响应和答案生成方面表现出色，但吞吐量较差。而数据并行则在整体吞吐量上表现优秀，但在首响应和答案生成方面表现不佳。从开发人员的角度来看，我们是否可以将两者结合起来，取长补短呢？遗憾的是，这并不容易实现，因为张量并行和数据并行的KV数据布局不同，它们之间无法直接通信。

我们的Snowflake研发团队为此开发了一种名为Arctic Sequence Parallel的新机制。Arctic Sequence Parallel能够提高吞吐量，同时保持首响应的高效率，并降低生成延迟。

在设计Arctic Sequence Parallel时，我们保持了其KV数据布局与张量并行的一致性，从而实现了两者的互操作性。我们将这种结合称为Shift Parallelism，即“可平移的并行机制”。当用户调用系统时，我们可以根据调用的特点（小批量或大批量）实时选择使用张量并行还是Arctic Sequence Parallel，这种组合方式非常灵活。



最终，这种机制显著提升了推理效率。在测试中，我们的端到端响应效率比传统方法提高了3.4倍以上，吞吐量提升了1.7倍。如果仅从向量化（embedding）的角度来看，我们的提升更是达到了16倍以上。此外，PPT右下角的第三方测试结果显示，与市面上主流的推理系统相比，我们的系统排名前三。还有一个好消息是，我们的这一技术是开源的。如果大家希望对自己的企业级AI部署或产品开发进行优化，我们非常欢迎大家加入开源社区，使用并优化我们的代码。



Snowflake Cortex AI

在介绍了Snowflake的五大支柱以及我们的研发成果之后，我想进一步谈谈这些研究成果是如何体现在我们的产品架构中的，以及我们如何通过这些架构帮助大型企业客户部署他们的Agentic AI。

我们的产品架构首先围绕AI功能展开，我们将其命名为Cortex。Cortex能够处理各种不同类型的数据，无论是结构化数据、非结构化数据，还是语音、图片、文档等，都可以安全且高效地保存和管理在Snowflake的云平台上。我们的治理功能非常全面。我们通过用户角色进行权限管理，而不是单独针对账户或个人进行管理。此外，我们还提供了“护栏”（Guardrails）功能。例如，来自生产部门的用户只能访问特定的大模型，如LLaMA，而来自研发部门的用户则只能访问Mistral。这些权限管理通过内部控制机制实现，确保了数据的安全性和合规性。

在Snowflake的单一平台上，我们还内置了主流的大模型，包括OpenAI、Anthropic、Meta、Mistral、DeepSeek，以及我们自研的大模型Snowflake Arctic。基于这些模型和我们的安全机制，我们提供了多种工具和功能。其中，Cortex Analyst和Cortex Search是两个独立的功能模块，分别用于处理结构化数据和非结构化数据。开发人员可以直接调用这些工具，也可以通过简单的SQL语句进行操作，我们将这一功能称为AISQL，它能够将结

构化数据和非结构化数据进行联合查询。此外，在文档处理方面，基于我们的研发成果，我们提供了一种高效且出色的处理方式。

在架构的最上层，我们提供了一系列Agent API。如果开发人员希望将你们的产品或应用与Snowflake整合，但不想通过我们的界面操作，可以通过API调用的方式实现无缝对接。

Snowflake Intelligence是我们Agentic AI部署的核心整合平台。它利用了我们刚才提到的五大支柱和研发成果，能够自动编排用户提出的请求，将任务自动分发到不同的工具集中，并完成上下文保持的优化过程。通过这一平台，我们能够为企业级用户提供高效、安全且可信的AI部署解决方案。

实例演示

基于之前的介绍，我将为大家进行一个演示。这个演示的场景是这样的：我目前负责管理一个大型软件工业园的物业管理。园内有众多不同类型的公司和初创团队。当这些团队入驻软件园后，他们可能会发现一些建筑质量问题，比如墙体开裂或漏水。遇到这些问题时，他们会向我反馈，并发送大量照片，指出墙体存在问题。作为软件园的管理方，我需要快速分析这些用户反馈，并做出响应。同时，在内部流程中，我还需要进行预算评估、采购计划以及维修安排。

视频展示了如何利用Agentic AI功能来实现这一业务场景。首先，在Snowflake平台上，我们可以保存大量的结构化数据，这些数据包含了产品信息以及相关的语义信息，例如表格中包含哪些数据，表格的用途等。此外，对于非结构化数据，我们可以收集大量用户反馈的照片，这些照片展示了墙体的各种问题特征。

以往工作人员需要花费大量时间和精力去阅读这些照片，并由专业人员进行分析。但在Snowflake的Agentic AI平台上，我们可以将这些上千张照片安全地存储在单一的云平台中，并通过AI功能对照片进行解析。从开发人员的角度来看，这一过程非常简单。我只需要编写几行SQL语句，在语句中指定使用哪个大模型来处理图片。例如，我可以写这样的语句："请帮我分析一下图片表，它表达了什么样的建筑质量问题。"通过简单地执行这些SQL语句，我们就能得到对图片中建筑质量问题的详细分析结果。

作为开发人员，我可以看到具体的分析结果。基于这些结果，我还可以进一步追问大模型，比如："请根据你看到的所有问题，从上千张照片中找出出现频率最高的问题，并为我推荐维修产品。"这些操作都可以通过简单的SQL语句轻松实现，这就是我们的AISQL功能。在执行这些SQL语句后，系统可以处理和检索上千张图片，同时结合我们庞大的维修产品数据库和报价数据库，最终得出一个推荐方案。这个方案会明确指出我们需要使用哪些材料进行维修，需要购买哪些产品，以及这些产品的价格。

视频详情：https://mp.weixin.qq.com/s/tDYOeRz7aJWFBEQmQHhLFQ

接下来，我将从商业用户的角度出发，展示在开发完成后，如何在Snowflake Intelligence的图形化界面中处理这一业务场景，并得到分析结果。

在商务人员的环节，他们会提出自己的问题，比如："基于我需要购买的材料，你能否帮我找到合适的供应商？大概需要花费多少钱？如何进行采购？有哪些采购策略可以提高效率并尽量控制成本？"当这些问题在对话框中提交后，回顾我们之前提到的五大机制，这些任务已经在后台被并行拆分。具体来说，任务会被发送到Cortex Analyst，从数据库中读取产品报价和供应商信息；同时，任务也会被发送到Cortex Search，对图片信息进行解读，并将这些信息整合在一起，最终提出建议结果。例如："您提到的产品包括以下这些……"

这一步骤也体现了我们五大支柱中的**可追溯性**（Observability）。AI的执行过程需

要对用户透明，并详细记录下来，这样才能让用户对企业的AI部署充满信心。最终，系统不仅会给出具体的建议，比如“在京东购买这些产品”，还会详细解释为什么选择这些供应商，以及如何优化预算以覆盖更多业主的需求。

如果继续追问，系统会进一步提供推荐理由，包括每种产品需要的数量以及推荐这些产品的依据，比如供应商的库存量或好评度。通过Snowflake Intelligence，基于我们研发的五大支柱构建的Agentic AI环境，商业用户可以方便地将请求发送到不同的数据源，无论是结构化数据还是非结构化数据，所有操作都在一个单一、安全的环境中完成，无需在不同界面或环境中频繁切换。所有数据都保留在AI平台上，我们强调“将工作带到数据面前”，而不是“将数据拿出带到工作面前”。因为每次将数据从云环境中移出或复制时，安全风险都会呈指数级增长。整个AI的执行过程是可追溯的，用户可以看到推理和执行的全过程。

最终，我们的结果是高效的。例如，处理几千张图片可能只需要一分钟，通过一个SQL语句就能完成查询。当然，在实际应用中，我们可以回顾之前提到的AT&T案例。作为拥有超过10万名员工的大型企业，AT&T已经在使用Snowflake平台，每天的API调用次数超过4.5亿次。

总结

Snowflake的研发是基于五大支柱的。在这些支柱之上，我们建议企业在部署企业级Agentic AI时，需要考虑几个关键问题。首先，工具集是否具备智能编排系统，能否将任务拆分并进行不同任务之间的调度？其次，是否能够安全高效地处理结构化和非结构化数据？此外，企业级AI的部署是否是可观测的、可信任的？最后，它是否具备良好的系统优化，能否在推理过程中满足企业级用户对效率和成本的要求？我的分享就到这里，感谢大家的关注和支持！

嘉宾介绍

- **杨扬**先生现任Snowflake亚太及日本地区解决方案工程副总裁，常驻新加坡。他在数据、人工智能和分析领域拥有逾二十年的领导经验。杨扬先生致力于协助亚太地区的客户及合作伙伴充分利用Snowflake数据云的优势。他所领导的团队由专业的工程

师和现场首席技术官组成，负责推动转型项目并开展全球协作，以提供创新的解决方案。

此前，杨扬先生曾在Workday、Qlik、Oracle等知名企业以及多所大学担任高级技术与管理职务。凭借卓越的技术洞察力，他持续为公司创造业务价值，并因此荣获多项行业奖项、总裁俱乐部表彰以及数据仓库、分析和云技术领域的专业认证。

杨扬拥有澳大利亚Wollongong大学计算机科学研究硕士学位，是一位备受欢迎的演讲者，以其将深厚技术专长与前瞻性领导力相结合、推动数据驱动创新而闻名。

颠覆传统认知！顶尖架构师眼中，决定职业生涯上限的不是技术能力丨独家对话一线架构大佬 Christian Ciceri

作者 李冬梅

> **采访嘉宾**：Christian Ciceri，软件架构师，Apiumhub联合创始人，《软件架构指标》作者

在当下软件开发的快速演进与人工智能浪潮交汇的背景下，软件架构师的角色与方法论正在经历前所未有的考验与变革。

过去，架构师的核心职责主要集中在系统设计、模块划分和技术决策上，强调的是稳定性、可维护性以及对技术栈的掌控。然而，随着云原生架构、微服务、大规模分布

式系统以及低代码/无代码平台的普及，软件系统的复杂性呈指数级增长，架构师面临的不再只是技术选择问题，而是如何在快速迭代与持续交付的环境中保持架构健康和团队效率的挑战。

与此同时，人工智能的兴起为软件开发注入了前所未有的工具和能力。自动化代码生成、智能测试、AI辅助设计等技术，使得部分传统的架构任务可以由算法快速完成。架构师不再需要亲自绘制所有依赖图或手动分析性能瓶颈，而是可以借助AI快速发现潜在问题和优化空间。这一变化在提升效率的同时，也提出了新的问题：决策权、系统理解与技术判断仍然高度依赖人类的经验和洞察力，**如何确保AI工具辅助而非替代，成为架构师必须面对的核心议题**。

可以说，软件架构师正处在一个**技术能力、业务理解与数据驱动决策三位一体**的转型期。在快速迭代的环境中，他们需要掌握新技术、理解复杂系统的演化规律，同时引导团队形成共享愿景和架构文化。

近日，InfoQ独家采访了Apiumhub联合创始人、知名软件架构专家Christian Ciceri，带领读者从一线架构师的实战经验出发，深度探讨“可度量、可演化的架构”理念，以及AI与现代软件工程工具对架构实践的影响。此次访谈不仅回顾了Ciceri本人的成长历程，也提供了丰富的架构实践智慧，让读者在快速变化的技术环境中，理解如何保持架构质量与团队适应性。

Ciceri的职业路径颇具代表性：他从一线软件开发与架构设计实践中积累经验，目睹了大型企业中灵活性不足、交付周期漫长、流程效率低下的常见挑战。2014年，他与叶夫根尼·普雷丁（Evgeny Predein）在巴塞罗那共同创立了Apiumhub，立志将敏捷方法论与软件架构紧密结合到业务运营的核心。正是在长期实践中，Ciceri逐渐形成了“可度量、可演化架构”的理念，并将这一理念凝练在其著作《软件架构指标》中。他强调，构建稳固且具适应性的系统，不仅能提升软件交付质量，还能保证系统随业务需求同步成长。

在采访中，Ciceri对“可观测性”（Observability）和“架构治理”进行了深入阐述。他指出，系统运行时的质量属性只是整体质量的一部分，真正的架构治理需要对所有软件属性保持持续监控。借助演化式软件架构中的适应度函数（fitness functions），团队能够实时监测架构健康状况，及早发现开发速度下降、缺陷增加或性能问题等架构退化

迹象。

访谈中另一个高光时刻，是Ciceri对AI在软件架构中的角色的洞察。他明确表示，**AI可以辅助分析指标、提供可能的改进方案，但无法取代人类的判断与决策**。他强调："只有在人类驱动下，人工智能才能在软件设计中成为真正的生产力工具，而不是取而代之。"对于当前各种AI生成的架构建议，他仍然定位它们为"助手"而非"伙伴"，提醒架构师在拥抱AI时保持理性。

从架构师到创始人

> **InfoQ：Ciceri先生您好，阅读您的书了解到您一直是"可度量、可演化架构"的倡导者。能否先谈谈您从一名一线软件架构师成长为Apiumhub联合创始人，以及编写《软件架构指标》的经历？**

Christian：我的职业生涯发展，得益于多年在大型企业从事软件开发与架构设计的实战经验。期间，我遭遇了行业常见的挑战，比如企业灵活性不足、交付周期漫长以及流程效率低下等问题。

2014年，我与叶夫根尼·普雷丁（Evgeny Predein）在巴塞罗那共同创立了Apiumhub公司。我们的核心目标，是将敏捷方法论与软件架构置于业务运营的核心位置。

随着时间推移，我逐渐意识到，要创造长期价值，必须专注于可量化且可演进的架构设计。打造既稳固又具备适应性的系统，才能让团队交付高质量软件，并确保软件能随业务需求同步成长。这一理念，最终促使我撰写了《软件架构指标》（Software Architecture Metrics）一书中的相关章节。

> **InfoQ：在这本书中我留意到，许多架构师强调"愿景"，而您更强调"度量"和"指标"。为什么可衡量性在现代软件架构中如此关键？**

Christian：指标与度量是推动争议更趋近客观的有效方式。但需要注意的是，与团队共同秉持的架构愿景，仍是软件架构师日常工作中极为重要的一部分。

> **InfoQ：AI工具正在自动化部分设计、编码和测试工作，但架构往往仍依赖人的判断。您认为哪些架构设计环节可以真正被AI增强？哪些仍应由人主导？在AI辅**

> 助开发与智能化工程工具快速发展的背景下，您的架构哲学有哪些新的变化？

Christian：虽然人工智能有望成为软件架构师工作中非常实用的助手，但我确实认为它无法取代技术决策过程。技术决策必须由人类的能力和经验来主导。

换句话说，我深信，只有在人类的驱动下人工智能在软件设计中才能成为真正的生产力工具，而非相反。

> InfoQ：如今出现了越来越多AI生成的架构建议（如微服务拆解、依赖图优化等），您认为这些工具目前是否足够可靠，能成为“架构伙伴”？还是仍停留在“助手”阶段？

Christian：在我看来，这些工具有助于提出可能的解决方案并发掘新的可能性，但正如我之前所说，它们无法取代人类的决策过程。在我看来，这些工具现在是、将来也永远会是有价值的“助手”。

> InfoQ：Apiumhub与许多大型企业合作进行架构转型。在这些项目中，您认为最大的阻力来自哪里——技术层面，还是文化层面？如何在企业内部建立一种可持续的“架构文化”，而不仅仅是设立一个“架构部门”？

Christian：在马丁·福勒一篇题为《谁需要架构师》的旧文中，他区分了传统架构师的两种角色——决策者与引导者，其中后者是“提升团队水平”的最佳方式。这种方式有助于在团队内部建立真正的软件架构文化，与开发人员并肩工作，尤其是在建模阶段。

不过，我坚信软件架构是工程学这个更广泛领域内一门不断发展的科学。我的意思是，**任何人都可以成为架构师，但这并非可以随意为之的事情**。要成为一名高效的架构师，你需要研读大量的科学文献，并持续深化对该领域的理解。

持续架构与架构可观测性

> InfoQ：如何将“可观测性”融入系统设计，使架构质量不再只是文档指标，而是实时可见、可验证的？

Christian：“可观测性”通常指的是系统运行时的质量属性，而这只是系统整体

质量的一部分。一般来说，为系统引入架构治理或者架构设计意味着要对所有软件属性保持持续的管控。**实现这一过程的一个良好方法是采用“演化式软件架构”中的相关技术**，其中最重要的概念就是适应度函数（fitness functions）。

> **InfoQ：面对多AI代理、低代码组件、分布式系统等复杂生态，您建议架构师如何构建架构层面的“可观测工具箱”？有哪些信号能提醒团队：他们的架构正变得越来越“不可观测”、难以演进？**

Christian：当我们从架构治理的角度思考“可观测性”时，一次架构性的错误，**理想情况下应该能够通过失败的架构单元测试被检测到**。不过，架构退化的迹象通常是逐步显现的，例如开发速度变慢、缺陷和运行时问题（如性能问题）增加、系统在应对更高负载时出现困难，以及其他类似的症状。

> **InfoQ：在您的著作中，您提出要将架构质量属性（如可扩展性、可维护性、模块化）与具体的可量化指标相连接。实践中，团队对这种理念的接受度如何？**

Christian：根据我们的经验，这其实取决于团队的具体情况。总体而言，**指标不应该被强制设定为目标**，而应当结合团队的文化建设，谨慎地引入和推广——这才是真正的出发点。另一个关键点是，**指标的使用应当建立在真实且公认的痛点之上**，也就是那些团队成员都能感同身受、并一致认为需要重点关注的问题。

> **InfoQ：您在实际项目中见过哪些被误用或被误解的架构指标？能否举一个具体案例，说明架构指标如何帮助发现系统“架构退化”问题，或指导架构重构决策？**

Christian：我认为被**最滥用的指标**是测试代码覆盖率。问题在于，这个指标几乎无法告诉我们测试策略是否真正有效——测试效果主要取决于被测试模块（如类、方法等）的设计质量。不过话说回来，当代码覆盖率数字**非常低**时，它依然是一个有用的信号，因为这通常明确反映出团队生产力不足，或是开发流程存在问题。

> **InfoQ：随着AI进入软件分析领域，是否有可能出现一种“智能架构监控”机制，让系统指标不仅被观测，还能被自动优化？**

Christian：不，在我看来，这仍然属于“科幻”的范畴。虽然人工智能确实可以在分析指标、提出潜在改进建议等方面提供帮助，但它无法取代人类的判断力，也无法

替代软件架构中所需的细致决策过程。软件架构涉及**权衡取舍**、**理解业务背景**以及**预判未来需求**等方面，而这些都是目前难以在自动化系统中完全编码和实现的。

AI时代架构师的黄金生存法则

> **InfoQ：在您看来，当下优秀架构师最重要的特质是什么？是分析能力、领导力、共情力，还是好奇心？**

Christian：你说的这三种特质，包括分析能力、领导力、共情能力都很重要。分析能力当然是理解复杂系统、做出稳健架构决策的关键。但我认为，**好奇心**同样珍贵——它推动持续学习，帮助你不断掌握新技术与新实践，并且常常能引导出那些**单靠分析无法得出的创造性解决方案。**在许多方面，正是好奇心让架构师能够在不断变化的领域中成长与适应。

> **InfoQ：如果您愿意的话，能否请您推荐一两本您认为正在或者将要影响“架构师思维”的书籍或资源给我们的读者？**

Christian：所有广为人知的领域驱动设计（DDD）著作都很有参考价值， 但如果你想获得更深入、且更贴近当代的软件设计洞见，我推荐阅读**Vladik Khononov的《Balancing Coupling in Software Design》**。

> **InfoQ：如果有一个“架构师黄金法则”在智能时代仍然适用，您会说是什么？为什么这么说？**

Christian：我们这一行业的核心“黄金法则”是：**把“我（I）从架构（architecture）中去掉**。架构是一种**共享的愿景**，你不能仅凭自己对领域的理解就做出决策。真正的架构工作应当让整个团队共同参与，确保所有决策都是集体性的。

所有知识型岗都要被AI“吞了！清华大学教授刘嘉：未来大学分化猛烈，软件公司靠“几人+Agent”就够

作者 华卫

人类与AI间的对决，自2016年的AlphaGo打赢世界围棋冠军李世石起，就开始不断出现在大众视线中，出圈的例子更是不少。

曾担任《最强大脑》节目首席科学家的刘嘉，也亲眼见证过这样一场比赛。当时，还是百度大脑首席科学家的吴恩达带着搭载百度大脑的智能机器人小度上了舞台，与人类组选手比拼起“看照片认脸”。面对多轮挑战，最终人类最顶尖的面孔识别选手不敌AI。

这个结果，好似当头一棒重重敲向了此时正往北京师范大学副校长一职奔赴的刘嘉。他火速向学校递交辞呈，重新钻进实验室，将全部心思转投到了脑科学与AI的交叉研究中。

回到2025年的今天，我们更是已置身于一个几乎被AI包围的时代。去年，诺贝尔物理学奖和图灵奖双双花落AI领域。今年年初爆火的DeepSeek让“无所不知”的大模型遍布朋友圈，随后Manus的横空出现又将AI完全自主的蓝图放到大众眼前。AI真的将超越人类吗？身处于现在的时代，这个话题已被推至现实议程，越来越多的人能够感觉到一种深切的危机感。

在今年6月出版的新书《通用人工智能：认知、教育与生存方式的重构》中，刘嘉用“近乎疯狂”几个字来形容AI的进化速度。在他看来，这场人造的进化史诗，已不再是一场静观其变的外围变革，而是每一个人都必须直面的“新生存现实”。因此，这本书从认知底层逻辑讲起，一步步帮助读者清晰认知AI、审视自身定位，堪称是一则给所有人的“AI时代生存指南”。

近期，刘嘉接受了我们的专访，如今，他不仅是清华大学基础科学讲席教授，也是北京智源人工智能研究院首席科学家。访谈中，刘嘉不仅具体谈了AI在认知科学视角的真实智能级别，而且明确指出了AGI实现的能力标志与技术路径的方向。

同时，作为一名多年扎根科研与教育一线的学者，刘嘉剖析了当前高校AI教育中存在的现实问题，并结合具体案例提出许多切实可行的教育改革建议。对于大众，刘嘉也全方位给出了如何与AI抗衡的多方面秘诀。

人工智能到人类的距离

> Q：刘老师，最近一年里，AI领域的哪一进展或成果最令您感到震撼？原因是什么？

刘嘉：从大众视角来看，我认为今年截至目前最重要的两个进展，一个是人形机器人领域，另一个是智能体领域，我来简单解释一下。这两个领域的发展，正是传统大模型开始向物理世界和虚拟世界延伸的标志。

在机器人领域，核心是探索AI这个智能体如何在真实生活中与人类交互，目前我看到了一个很好的现象：大家开始关注AI如何在真实世界里工作、与人交互，以及从事一些社会相关事务，像宇树科技的机器人、上海的智元机器人等，都是现在大家比较关注

的。不过，整体趋势向好的同时，这个领域也有需要改进提升的地方。现在大家更多还是关注机器人的稳定性、动作协调性等基础层面，没有真正用AI去控制它的行为，在开放环境中让机器人像人一样做出自适应行为。这一点上，目前还没有较大突破，仍处于比较传统的阶段。

至于第二个让人振奋的领域，是虚拟世界里的“机器人”，也就是智能体（Agent）。我们知道，AI或基于大模型的AI，虽然擅长推理和内容解释，但没法具体做事，比如订一张去西安的机票、了解股票行情及民众反应等，它都做不好。但现在出现的智能体技术，能给大模型装上各种各样的“手脚”和工具，让它具备实际办事能力：有的能订票，有的能写程序查询具体问题，比如通过写程序、做统计来分析近五年股票走势，还能查询天气并形成综合报告。如今，大模型结合各类工具形成的智能体，是非常好的发展方向，这表明AI现在能具体帮人做实际的事。

这两点结合起来就能发现，一个是物理世界的机器人，另一个是虚拟世界的智能体。我们也都可以看到，像ChatGPT、豆包这类传统大语言模型，正从原来“你问我答”的对话模式，逐渐向服务人类、与人类进行实体及工作层面交互的方向转变，而且已经迈出了重要一步。所以在我看来，今年可以说是大模型走向实际应用的开端。

> **Q：您目前的研究方向是什么？能否具体谈谈。**

刘嘉：我的科研方向，其实和脑机接口、机器人领域都有一定关联，核心是围绕“脑科学如何启发人工智能发展”来展开。先说说和脑机接口相关的部分。脑机接口目前最大的难点，不在于柔性电极能放置多久、通道数有多高，而在于对大脑的理解，比如大脑神经元的放电，到底代表着“想拿杯水”“想去抓东西”还是“想走两步”，这种神经编码的意义解读，是当前最困难的一步，特别需要脑科学的突破，这也是我在脑机接口方向重点研究的内容。

再看机器人领域。现在的机器人不能只依赖Transformer架构，因为Transformer本质是序列加工，靠预测下一个token（词元）工作，只能进行串行处理；但人类对世界的感知、身体的运动，都是并行进行的，比如感知环境时，不会先处理视觉再处理听觉，运动时也是多部位协同。所以，在感知和运动层面，必须要有受人类大脑或生物大脑启发的全新架构。我在这一领域的工作，就是研究如何实现环境信息的并行加工：快速理解周遭环境、提取关键要素后，把信息传给大模型做推理，再通过并行方式驱动运动，让

机器人能更贴合人类的感知与运动逻辑。

总的来说，我现在的研究核心是“脑科学+AI”，试图解决两大问题：一是脑机接口中大脑信号的解码问题，让AI能更好理解人类大脑意图；二是机器人在真实物理世界中感知与运动的并行加工问题，突破现有架构的局限，从而拓展大模型在实际应用中的能力边界。

> **Q：ChatGPT、Claude、Gemini等大语言模型重塑我们日常生活与科研工作的当下，它们真的“理解”自己输出的内容吗？如何验证一个系统是否真正具备理解能力？**

刘嘉：我觉得这是一个非常好的问题。在2024年9月OpenA推出o1大模型之前，“大模型是真正理解人类需求与话语，还是仅基于概率进行随机采样”确实是个值得大家争议讨论的话题。但2024年o1推出后，我认为所有相关讨论基本都该画上句号了。

o1和之前的大模型不一样，它是一款基于推理的大模型。从学术角度来说，这叫基于“思维链”——比如要达成某个目标，它会先梳理出一步一步怎么做的思考链，再顺着思考链逐步推进以解决问题；要是过程中某一步走不下去，发现之前的思路有误，它还会换种方式重新生成新的思维链。这一特性对大模型的工作方式来说是革命性的改变，也让现在的大模型基本都成了推理大模型，要么是带思维链的模型，要么是兼具推理与对话功能的混合大模型。至于像GPT-4、GPT-4o这类纯粹的对话大模型，如今基本已经退出历史舞台了。咱们中国的DeepSeek R1是首个开源的推理大模型，它的开源大大加速了大模型向推理方向的转换进程。

这种推理能力，本质上体现了大模型对人类需求与任务的理解，并且能模仿人类的思维方式解决问题——这很像心理学家丹尼尔·卡尼曼在《思维，快与慢》一书里提到的“慢思维”。书里说人有两种思维模式，“快思维”是本能反应，比如看到黑乎乎的东西冲过来会下意识躲开，动物也有这种本能；而“慢思维”需要前额叶参与，要对事情进行拆分、生成逻辑链，也被称为“第二系统”。现在的推理大模型，正是在模仿人类的“慢思维”过程。

也正是因为推理大模型的价值，才有了今年智能体的爆发。智能体最核心的就是“规划（Planning）”能力：给它一个任务，它得明确用哪些工具、怎么用工具来完成，

要是没有推理大模型，这件事根本做不了。其实智能体的概念很早以前就提出来了，但直到今年才大火，关键就在于推理大模型的出现。

所以总结一下我个人的观点：“大模型是随机采样还是模仿人类推理”这件事，现在无需再详细讨论，因为大模型确实能理解我们的需求，还能模仿人类、按照人类的思维模式进行推理。也正是“思维链（Chain of Thought）”带来的这种推理能力，让智能体可以用起来，而且现在大模型的智商已经远超人类，毫不夸张地说，在国际奥林匹克数学竞赛中，它也基本达到或超越了人类最顶尖的水平。因此，现在大模型在智能层面已经达到甚至超过人类智能水平，这一点是毫无争议的。

> **Q：若未来AI不仅通过图灵测试、镜像神经元模拟甚至会表现出“痛苦”等情感状态，这是否意味着意识可从计算中涌现？**

刘嘉：我觉得这是个非常好的问题。我们之前聊到了智商（IQ），我认为AI在智商层面已经比人类更强，这一点没问题，但你提到的两个关键问题：AI是否会产生真正的情感（而非像自闭症那样推理出来的情感）、是否会产生意识，我的回答是“yes and no”。

先说说“yes”的部分：没有理由认为AI不能产生意识。从人类进化的角度来看，在人类出现之前，动物并没有“我是谁”这样的自我意识，但猴子逐渐进化成人的过程中，人类慢慢产生了自我意识，而人类复杂细致的情感、创造力等，也都是自然衍生而来，并非凭空赋予。既然人类能进化出这些特质，AI理论上也没有理由进化不出来。

再来说“no”的部分：现在大模型的发展，即便加上Transformer架构，或是通过稀疏化技术做成MOE（混合专家系统），从2022年11月30日GPT-3.5推出到现在，大模型的能力虽在不断提升，但架构本质上没太多变化，只是从密集型转向稀疏型，参数量不断增加而已。可问题在于，仅仅增加参数量，就一定能让AI拥有人类的自我意识、产生人类般的情感，实现像人类那样从零到一的颠覆式创新吗？我觉得这要打个很大的问号。

我可以举两个具体例子：第一，从进化角度看，人类大脑的体积、神经元数量或神经元连接数，并非地球上最多的，大象、鲸鱼的大脑比人类大，虎鲸的神经元数量也比人类多，但世界上最聪明的仍是人类。这说明从生物进化层面，并非神经元或“参数量”越大，生物就越聪明，架构其实非常关键。可人类大脑独特的架构是什么？它和大象等

动物的架构区别在哪？我们现在还不清楚。第二，再看运算的基本单元——神经元。现在人工神经网络的神经元，基本还是1943年的MCP模型，本质就是把权重求和，再根据是否超过阈值输出对应值，即便如今复杂的大模型，用的也差不多是六七十年前的神经元模型。但这个模型过于简单了：从线虫到果蝇、斑马鱼、小鼠、猴子再到人类，进化过程中人类不仅神经元数量增加，神经元结构的复杂度也大幅提升，这暗示神经元复杂度可能是人类拥有高智能的关键因素，而目前像ChatGPT这样的大模型，其神经元复杂度远不及人类。

所以能看出，参数量大只是AI提升智能的必要条件，而非充分条件；简单的神经元模型也无法支撑起复杂的智能需求，神经元复杂度或许是影响人类智能的关键。这意味着，可能还有很多与架构复杂度相关的因素影响着智能，而现在的大模型远远达不到要求。

因此总结来说：AI要实现人类级别的智能，或许还需要向人脑学习，需要新的AI架构或新的AI形式。简言之，“yes”的是，大模型或人工智能理论上没有理由不能实现人类的意识、情感与创造力；“no”的是，现在的大模型还需要在架构、底层神经元的复杂度与精细度上做出根本性改变，必须从脑科学中获取新启发，才能走向下一代人工智能。

> **Q：当前AI仍依赖人类标注数据，未来是否可能实现完全自主的“自我进化”学习？**

刘嘉：你提到了一个非常本质的问题，我把它称为“内驱力”。我们人类有着极强的内驱力，总想着不断学习、不断进步，而这种内驱力的本质，来源于人类特有的“死亡意识”。我们在很小的时候就知道，每个人都会死亡，只是不确定具体会在什么时候。正是这种“一定会死”的确定性，与“何时会死”的不确定性之间的矛盾，让我们不断思考：能不能给这个世界留下点什么、创造点什么？也正因如此，人类才有了强烈的动机去学习、去工作、去创造。但动物没有这种内驱力，因为它们没有死亡意识，它们不知道“死亡”是什么，只有像猫、大象等动物在生命快走到尽头时，才会找地方安息，不会像人类这样从一开始就清楚自己终将死亡。

把这个逻辑放到现在的大模型上，会发现两个关键差异。第一，大模型所有的学习都是人类要求它去学的，比如按照强化学习的规则去学习知识，或是根据人类给出的指

示整理信息，它就像小学或初中的孩子，需要“家长”在旁边监督，做得好就得到奖励，做得不好就受到“惩罚”，完全不是主动去学习。第二，大模型不存在“死亡”的概念，GPU坏了可以换一块，电线断了重新接上就行，它没有那种“必须变得更好、更强大”的迫切动力，只要人类不给出新的指示，它就会一直停留在当前的状态。

回到你刚才的核心问题，现在大模型所谓的“自学习”功能，比如大家常说的无监督学习，本质上还是人类给它设定好的学习规则和要求，并不是像人类这样，有发自内心的自我驱动去学习。所以在我看来，目前大模型的学习无论是有监督学习还是无监督学习，都属于被动学习，和人类主动学习的模式有本质区别，这也是当前AI和人类最核心的划分之一。

当然，我们也要考虑到，如果有一天大模型真的拥有了自我进化的动力，它的发展效率一定会远超人类，甚至能在极短时间内进化出超越人类的能力，把人类远远甩在后面。但对人类来说幸运的是，目前我们还不知道该如何让大模型拥有主动学习的能力，也不知道该怎么赋予它真正的内驱力。

> **Q：当前您最期待和关注哪一AI细分方向的发展有突破？**

刘嘉：我觉得从今年开始，AI领域有三个比较火、能和人类直接互动的方向，但其中脑机接口离我们的生活还相当远，它有个重要瓶颈没解决，就是大脑信息的解码，这更多取决于脑科学进展，和当前AI技术关系不大，当下还无法发挥重要作用，所以我们先跳过这个方向。剩下两个方向不仅和我们生活密切相关，还会产生巨大影响。

第一个是机器人领域，但这里说的不是现在常见的传统人形机器人，那些大多没借助大模型做决策，还是沿用传统工业机器人的思路。我指的是像马斯克的Optimus、OpenAI参与投资的Figure01以及谷歌尝试的那些机器人，它们核心关注“眼手合一”：“眼”代表大脑（大模型），“手”代表操作，重点是用大模型驱动肢体去适应环境。未来这个领域会对家居养老、服务行业等很多方面产生深刻影响，值得重点关注，它本质是在物理世界给大模型“装上手脚”，让其能帮我们行动。

第二个方向是虚拟世界里的智能体。比如去西安旅游，需要买票、订住宿、规划行程、和朋友沟通等，这些环节涉及不同能力，大模型能做规划，但没法直接执行购票、订房这些操作。而智能体就是把大模型和各类工具结合起来，比如查天气的工具、订票

的工具、处理生活事务的工具，让大模型能调用这些工具完成实际操作。目前这一块已经能直接改变日常生活方式，相当于每个人都有了专属“秘书”，帮着做PPT、处理讲课准备等事，它就像虚拟世界里的机器人，帮我们满足日常需求。

这两个方向很快会催生出大量商业应用，还会极大改变我们使用互联网的模式。将来手机上可能不再需要一堆APP，查航班、点外卖、订票等需求，都能通过大模型驱动的工具在一个接口里统一完成，对互联网使用产生深远影响。所以简而言之，脑机接口因依赖脑科学进展，应用还很遥远；而机器人（物理世界）和智能体（虚拟世界），都是给大模型“长手脚”，会切实改变我们的生活。

> **Q：最近AI Agent在国内外都十分热门，您有体验过这一领域的产品吗？感受如何？**

刘嘉：在Agent领域，目前有两类发展值得关注。一类是垂直类Agent，这一块现在做得比较好的，比如像Cursor、编程类的Copilot这类专注于编程领域的工具，它们结合大模型后，能力非常突出，甚至有人说今年是“程序员失业元年”——因为在Cursor这类垂直编程辅助工具的影响下，很多初级或中级程序员受到了冲击，面临失业风险。而且我认为，这类垂直类Agent还会越来越多，未来不仅是编程领域，像音乐创作、视频制作等领域也会出现类似工具，这可能会导致大量原本从事这些行业的初级或中级从业人员失业。

另一类则是通用类的，如字节推出的“扣子”（Coze）是做解决人们通用问题的通用型Agent，潜力和发展前途同样很大。我个人认为，虽然无论是垂直类还是通用Agent，目前都还只是刚刚起步，但从另一个角度来看，它们未来会给整个社会的工作方式甚至组织架构带来巨大变化。举个例子，以前一家软件公司可能需要招聘大量初级或中级程序员，而未来，或许只需要几个资深程序员或顶尖架构师，再搭配Agent，就能完成很多原本需要大量人力才能做好的工作。

无定义的AGI，已进化为新物种

> **Q：您在书中为何选择从认知科学视角而非纯技术角度解读AGI？如果让您用一句话总结AGI对人类的意义，您会如何描述？**

刘嘉：我先从认知科学的起源说起，为什么要从这个角度讲呢？因为我们现在讨论的智能科学，最早可以追溯到20世纪40年代，有学者提出“控制论（cybernetics）”。后来控制论逐渐向两个方向发展：一个是往工程实践方向走，形成了我们今天所说的人工智能；另一个是往理论研究方向走，发展成了如今的认知科学。

所以你会发现，认知科学和人工智能本质上是一回事，只是侧重点不同，一个更偏向理论，一个更偏向工程应用。比如人工智能早期的“专家系统”，在认知科学领域对应的概念就是“符号主义”；现在人工智能里的“人工神经网络”，在认知科学中可归为“联结主义”相关研究；；而人工智能领域的“机器人与真实世界交互”方向，在认知科学领域则被称为“具身智能”或“具身认知”。它们其实是同一事物的两个方面，只是研究视角和应用场景有所区别，这是关于认知科学与人工智能关系的第一个点。

至于第二个点，如果用一句话概括AGI对人类的影响，我认为它既不会仅仅是人类的工具，也不会是来消灭人类的新物种。在我看来，AGI对人类最大的贡献在于，它可能会成为一种与人类共生共进化的全新物种，能推动人类自身不断迭代、持续进化。

> **Q：您在书中也提到AGI已从“工具”进化为“新物种”，如何定义这一转变的关键特征呢？**

刘嘉：在以GPT为代表的大模型出现之前，我把当时的AI称为“任务特异的AI”，它们本质上就是一种工具，比如高铁机场的人脸识别，只能专注做人脸识别这件事，而且做得比人类好；再比如去年获得诺贝尔化学奖相关的AlphaFold，能从氨基酸序列推测蛋白质结构，这项能力也远超所有人类，但它也只局限于解析蛋白质结构。

而另一类AI就是通用人工智能（AGI），它的核心不是在某一个专项领域超越人类，而是要全方位地和人类相似，做到“人能做的事我都能做，且做得更好”。就像现在的大模型，既能绘画、生成文字、通过图灵测试，也能写代码、查天气、分析股票进展、帮忙订票，能胜任各种各样的事情，本质上和人类的能力范围很接近。之前我们也聊过，大模型有没有可能拥有意识、情感和创造力，其实背后探讨的也是大模型能否像人类一样，成为一个与人类相似的新物种。目前AGI还没有标准定义，但大家有个共识：它要具备和人类相似的通用智能。

这两种AI的发展方向完全不同，一个是成为专业工具，一个是朝着“通用新物种”

的方向发展。也正因如此，从ChatGPT到现在的AI，没有昙花一现，反而越来越火、越来越出圈，就连买大白菜的老大妈都知道AI，因为它在模仿人类，和我们的关系越来越密切。而像2016年的AlphaGo，虽然打赢了世界围棋冠军李世石，却只是昙花一现，没有真正出圈就归于沉寂了。

AGI何以真正实现？

> Q：您认为未来3-5年，AGI最可能突破的技术瓶颈是什么？

刘嘉：现在大家大多认为ChatGPT这类大模型是AGI的火花或雏形，还不是真正意义上的AGI，但对于AGI何时出现，不同人有不同观点。比如DeepMind的CEO（也是去年诺贝尔化学奖获得者）认为可能10年内实现，马斯克认为大概在2029年，“人工智能之父”辛顿觉得会在2028年，更极端的像OpenAI的CEO奥特曼，他在今年6月写了一篇名为《温和的奇点（Gentle Singularity）》的博客，认为AGI现在已经实现了。不过无论观点如何，大家有个基本共识：真正的AGI肯定会在我们有生之年实现，最保守地说，10年内实现是没问题的。

在我看来，AGI真正实现需要两个标志。首先，图灵测试现在已经失效了，去年就有AI通过了图灵测试，所以不能再用它来衡量AGI。第一个关键标志要从大模型本身来看，现在的大模型还没有零到一的创造力，只能做组合式创新。举个例子，如果让大模型学传统写实绘画，它能画出很精美的作品，但没法像塞尚、梵高那样，跳出传统绘画框架创作出印象派画作；就像在经典牛顿力学体系里，没法自然涌现出爱因斯坦的广义相对论、波尔等人的量子力学那样的颠覆式创新。这种零到一的创新，在计算机科学领域叫OOD（Out of Distribution/Off Distribution，分布之外）问题，也就是在AI已学习的知识范围之外，创造出全新的东西，这是人类最核心的创造力，但目前AI还做不到，而这正是AGI需要具备的关键能力。毕竟AGI的“G”不仅是“通用（General）”，也该包含“创造（Generative）”，深层次的零到一创造至关重要。

第二个标志是AI需要达到足够的感知和运动能力。现在的大模型基本基于Transformer架构，它有个先天缺陷：只能串行加工，就像通过已有词汇预测下一个词，必须按顺序推进，和人类说话的逻辑一样。但人类感知外部世界是并行的，不会先处理左边视野再处理右边，也不会先处理视觉再处理听觉，而是多模态信息同时加工，效率

很高，这是Transformer架构做不到的。运动能力也是如此，人类大脑有860多亿神经元，其中690多亿在小脑，小脑正因为要做大量并行运算，才能控制运动、帮助我们和外界交互，可现在的大模型在指导实际运动方面能力很有限。简单说，现在的AI没法像人一样“看”“听”外部世界，也没法对世界做出反应、改造世界，感知和运动能力都需要大幅提升。

所以总结一下，从大模型内部看，它需要拥有从0到1的创新能力，这相当于“新图灵测试2.0”，做到了这一点，大模型会有质的飞跃；从与世界交互的能力看，它需要“长眼睛”“长耳朵”“长手脚”，能感知世界、改造世界，而目前Transformer架构的串行加工缺陷，让它还无法实现这些。

这两个关键突破点都很难，而且在当下的Transformer架构里大概率都无法解决，目前我也没看到太多能解决它们的潜在曙光，所以很难准确判断突破会在何时发生。不过有一种可能是，一旦我们迎来类似“GPT时刻”的关键节点，进展就会飞速推进。就像之前做自然语言处理（NLP），大家研究了几十年都没太多突破，但Transformer架构和GPT出现后，相当于迎来了“GPT时刻”，各类大模型、AGI雏形很快就涌现出来，发展速度极快。

现在我们其实也在等待感知运动领域的“GPT时刻”或者创造力领域的“GPT时刻”，但没人知道这个时刻何时会来。如果它很快到来，可能就是一瞬之间，或许不用等到2028年、2029年，甚至不用等满十年；可如果这个时刻迟迟不到，大家可能要等待更久，比如十年、二十年也说不定。所以现在的情况是，我们清楚AGI需要突破什么目标，但具体怎么突破、会有什么新的技术或架构出现，大家都不确定，处于一种“前途既清晰又模糊”的状态。

不过目前能确定的是，至少有一个解决方向是明确的，那就是研究脑科学。因为人类本身就具备卓越的创新能力以及出色的感知和运动能力，所以现在AI发展的一个重要方向就是把目光重新投向人类脑科学研究，看看人类大脑的工作机制，能给AI带来哪些新的启发。也正因如此，我觉得当前AI领域一个特别火、也特别重要的方向，就是脑科学与AI的结合。

> **Q：您认为强化学习、脑模拟和自然语言处理哪条路径更可能实现真正的AGI？为什么？**

刘嘉：在我写的书中，我提到了通向AGI的三条路径，关于这个问题，我的看法是这样的：通向AGI的第一步，是通过语言来学习，而脑模拟和强化学习的速度相对慢一些。不过，要实现真正像人类一样的AGI，脑模拟和强化学习都是必要的。比如机器人要和外部世界交互，就需要在具体场景中进行交互训练，这就像现在英伟达在做的Isaac Lab相关项目，他们构建了很多虚拟环境，让机器人在里面通过强化学习获取反馈，核心就是为了突破运动能力这一难题，这是强化学习发挥作用的重要方面。

而脑模拟这一块，就像我们之前聊到的，大家都在尝试寻找新的算法来推进相关研究，但目前无论是强化学习还是脑模拟，都还缺少一个类似自然语言处理（NLP）领域中Transformer出现那样的“GPT时刻”。现在大家更像是在黑暗里摸索，有点像盲人摸象，不知道这些路径最终能不能走通，也不知道什么时候能看到突破的曙光。虽然大家都在做这些研究，包括我提到的脑模拟，但目前还没法确定这些方向能否顺利走下去。

就像以前在NLP领域，大家也长期在黑暗中探索，直到Transformer架构出现，才像是看到了曙光，虽然不知道具体什么时候能“天亮”，但能确定沿着这条路走，很快就能迎来突破。现在的情况是，我们知道强化学习对机器人领域至关重要，脑模拟对感知能力、创造力的提升很关键，也明确这些方向的重要性，但还处在黑暗中摸索的阶段，尚未看到一丝曙光。不过只要未来出现类似Transformer的关键突破或者“GPT时刻”，大家就会信心十足地扑上去了。也正因为如此，我认为目前脑科学与AI结合的相关研究很值得投入，但要明确的是，它目前还处于实验室基础研究的范畴，距离走向商业应用，还有相当长的一段距离。

当AI“停不下来”：未来教育必须改

> Q：如果AGI最终超越人类智能，人类是否应限制其发展？

刘嘉：现在有个很关键的问题，不知道你有没有关注过2023年年底OpenAI的那次风波，当时萨姆·奥特曼（Sam Altman）和伊利亚·苏茨克维（Ilya Sutskever）之间爆发了严重冲突，他们一个是管公司运营，一个是管技术。一开始奥特曼被赶走，后来在资本支持下“王者归来”，反而把伊利亚排挤了出去。有意思的是，他们的争执根本不是因为股票、收入这些利益问题，核心矛盾在于对AI未来发展的看法：伊利亚非常担心AI会“接管”世界，所以他主张要让AI和人类“对齐”，还要限制AI的发展。

但在我看来，伊利亚的这种想法过于理想化了，原因有两点。第一，他说要让AI和人类的三观对齐，可人类的三观本身就从来没有统一过啊。你想，中国和美国的三观能对齐吗？显然不能；就连我们国内，四川人爱吃辣，广州人偏爱甜口，生活观念都有差异，更别说三观了。人的特点就是多元化，不存在一种“绝对正确”的三观，每个人都有自己认可的价值取向。那要让AI和人类对齐，到底该和谁的三观对齐、和哪种三观对齐？这本身就没有标准答案，从逻辑上来说，让大模型去接纳所谓“统一的人类三观”就是不成立的。

第二，退一步讲，就算我们现在决定停止研发大模型，说“目前的水平够用了，不能再研发更强大、有零到一创造力、有自我意识或自驱力的大模型”，可我们自己能做到，竞争对手会停下来吗？肯定不会。2024年的时候，“人工神经网络之父”杰弗里·辛顿（Geoffrey Hinton）接受CNN采访，当时很多人觉得美国的大模型太危险，建议暂停研发，辛顿就明确表示：“美国可以停下来，但中国会停吗？中国不会停。如果中国的人工智能超过美国，美国就会受到严重影响，甚至可能走向衰落。”你看，国与国、公司与公司、人与人之间的竞争是客观存在的，根本无法消除。只要有竞争，谁先停下谁就会吃亏，最后就会陷入像当年核武器发展那样的局面，大家都知道发展核武器对人类整体有害，但美国有1000颗，中国就不能一颗都没有，否则在国际社会上根本没有平等对话的资格。只有当中国也拥有了核武器，才能和美国站在同等地位上沟通。

大模型和人工智能的发展也是一样的道理，想让大家都停下来、实现所谓的“对齐”，根本不可能。大模型会变得越来越聪明、能力越来越强，这是不可阻挡的必然趋势。

> **Q：从大学教授的角度来看，您认为未来教育应如何调整，来培养出AI时代“不可替代”的人才？**

刘嘉：我先举个具体例子说说我是怎么做的，再聊更宏观的方向。现在我给学生布置作业时，会鼓励他们一定要用AI辅助完成，如果有学生不懂怎么用，我还会教他们方法。之所以这么做，是因为让学生学会用AI后，他们能快速把作业中基础性的部分完成好。当每个人在AI帮助下都能达到80分的基础水平，我真正关注的重点就来了：在80分之上，你能做出什么和AI不同的东西？你的独创性、原创性体现在哪里？这才是人能超越AI、基于AI产生新价值的关键。这样一来，学生就会把更多精力放在如何创造、如何

变得与众不同、如何打造自身稀缺性上。

把视角放到大学教学上，现在再像以前那样单纯讲授具体的科学知识、学科知识，已经没有任何价值了，不管多复杂的知识，学生都能通过ChatGPT这类大模型学到。所以我认为，现在大学老师要教给学生的核心是两点：第一点就是刚才提到的，引导学生找到和AI的差异点，充分激发他们的创造力，培养批判性思维，让学生学会展现独特的创造能力；第二点是帮助学生打通不同学科的知识壁垒，比如学物理的学生，不能只局限于物理领域，还要去了解生命科学、人文社科等领域的知识。

道理其实很简单，所有创新点、新领域基本都诞生在学科交叉处。单一学科发展到现在，大多已经是“红海”，体系成熟，想在里面创新难度极大，而且AI已经把单一学科的知识掌握得非常全面；但学科与学科之间的交界地带，就像一片全新的“大陆”，AI目前还不知道该如何探索，而这正是零到一创新的核心领域。所以从大学层面来说，必须大力推行通识教育，鼓励学生跳出单一学科的局限，打通不同学科的知识，这样才能为创新打下基础。

也正因如此，未来大学的分化会特别严重。以前的大学大多以传授知识为主，比如讲大学物理，清华北大和普通大学的教学内容差别不大；但未来，像清华北大这样的顶尖大学，绝不能再抱着一本《大学物理》照本宣科，而是要推动学生打通物理与其他学科的关联，深入开展通识教育。这类优质大学有能力推动这种变革，而普通大学可能需要付出更多努力才能跟上。在这个过程中，谁率先开展真正的通识教育，谁就有可能成为新的优质大学；如果还固守传统教学模式，只沿着单一学科讲授知识，即便曾经是好大学，也必然会逐渐落后。可以说，未来新的教学方式会重新定义“好大学”与“普通大学”，也会重新区分“好老师”与那些过时、即将被AI淘汰的老师。

> **Q：中国多所高校已开设"AI+专业"交叉学科，您认为这类教育改革的核心应该侧重哪一方面？**

刘嘉：在我看来，“AI+学科”的发展既有好的方面，也有需要改进的地方。首先，从好的方面来说，现在所有学科都必须与AI结合，不存在不需要AI的学科，AI就像当年的计算机一样，一定会渗透到每个学科中，这是学科发展的必然趋势。比如清华大学今年新开的“无穹书院”，就扩招了150人，这个书院的核心就是“AI+”模式：学生可以选择自己喜欢的方向，无论是生物、绘画、艺术还是其他领域，都要先学习AI，最终目

标是把AI和自己的专业深度融合，我认为这是非常好的方向，也是现代学科发展的关键。

但坏的方面也很明显，现在有相当一部分推动“AI+某学科”的人，包括一些学科带头人，其实并不真正懂AI。这就导致“AI+学科”的发展很容易变成“袋装土豆”，表面上让学生既学AI又学本学科，甚至把学分加倍，却没有将两者有机融合。最后学生看似学了两个专业，却不知道如何融会贯通，使得“AI+学科”只停留在“1+1=2”的层面，甚至可能“1+1<2”，没能实现真正的融合增效，这和很多教育推行者自身不懂AI有很大关系。

所以我觉得，对于推动学科改革的学科带头人、系主任甚至校长们，有两件事必须做：第一，要亲自系统学习AI，深入了解AI的细节和原理，进行综合训练，而不是只知道几个AI名词、参加几次表面培训就够了，这是做好“AI+学科”的关键；第二，大学改革应该吸纳更多年轻老师参与，因为年轻老师对AI的理解和应用能力更强，也更能接受新事物，让他们来推动“AI+学科”的发展会更有优势。

总结来说，“AI+学科”是所有学科的必然选择，这一方向本身是好的，现在很多大学也在积极行动，比如开设人工智能必修课、建立“AI+”书院等；但要警惕把“AI+学科”搞成简单的“学两个学科”，必须实现两者的深度融合，把学生培养为“AI原生/AI Native”人才。而要做到这一点，一方面推动学科改革的老师们需要真正懂AI、站在AI应用的第一线，另一方面要依靠更多年轻老师的力量来推动改革。

普通人如何建立AGI“防线”？

> **Q：您提到“在AGI时代，基于知识和技能的白领行业寂然倒下，甚至包括教师和科学家”，那么未来哪些职业反而会因AI变得更稀缺？这类岗位需要具备哪些能力？**

刘嘉：在我看来，当前所有以知识密集型为核心的职位，都会受到AI的巨大冲击，像律师、程序员、会计、医生，还有我和你所在的教师行业都不例外。就拿教师来说，我们的核心工作是“传道、授业、解惑”，可现在“授业”（讲授知识）根本不需要我们了，ChatGPT讲得比我好，知识面也更广；“解惑”也轮不到我，它比我更有耐心、解释更细致。剩下的“传道”（传递三观），我未必能比ChatGPT做得更好。这么一来，传统教师的核心功能大多被替代了，就连我前几天碰到的一位家长都问：“我孩子才幼

儿园大班，刘老师，您说他还有必要读小学吗？”这个问题很关键，未来小学、初中、高中是否还有存在的必要？我觉得这会是个很大的问号，毕竟美国AI进入教育领域已是普遍趋势，学生靠AI加持，每天花两小时学专业知识就够了，剩下的时间能做课外活动、发展兴趣。

所以从职业角度看，一部分初级和中级职业的消失是必然的，但也有相反的情况。我前段时间和一位音乐人聊过，他说现在请他作曲的客单价不仅没跌，反而涨了。原因很简单，在大量AI生成的“鸡肋音乐”中，他的原创反而更让人眼前一亮，像他这样顶尖创作者的稀缺性变得更突出了。

把“大范围初级中级职业消失”和“顶尖人才更值钱”这两件事结合起来，能得出一个结论：至少目前，AI淘汰的只是平庸的初级、中级从业者，而那些稀缺、有才华的人会更耀眼。进一步说，现在不是AI取代人，而是懂AI的人取代不懂AI的人，掌握稀缺技能的人取代平庸、无稀缺技能的人。这就意味着，在当下社会，我们每个人要追求的关键，是让自己的价值、才华更具稀缺性和不可替代性。以前，比如做程序员，只要懂点简单编程、能完成基础任务，就能在软件公司或大厂找到初级职位“躺平”，可现在不行了，“滥竽充数”的人迟早会被淘汰，必须展现出自己的独特性。

其实从这个角度看，对人类而言反而是一种新的进化。过去我们把太多时间精力花在低端、简单重复的事上，只用极少时间做创造、做零到一的全新发明。但现在，有了AI的帮助，那些低端重复性工作可以交给AI，我们人能专注去做更具稀缺性的事。还是以教师为例，把重复性的知识讲授交给AI后，我们可以更多关注学生批判能力、创造力的培养，去激发他们成长的内驱力和自我学习的动力。这才是教师的本质，而不是日复一日地重复讲解勾股定律、分析李白杜甫的诗歌。我觉得，未来人类的职责会发生这样根本性的变化。

> **Q：您在书中提出“AGI将驱使人类重新定义自身的认知优势”，那么哪些人类独有的认知特质可能成为我们的防线？**

刘嘉：在我看来，现在大模型的工作与进化方式两个很明显的特征，也由此体现出它和人类的关键区别。第一，大模型确实能生成很多新东西，比如新的绘画、视频、文章，但这些大多是它学习人类已有知识后，进行组合加工的结果。而人类最擅长的恰恰是做零到一的颠覆性创新，能无中生有创造新事物：就像梵高、塞尚在传统绘画框架外

开创印象派，爱因斯坦在经典牛顿力学之外提出广义相对论，波尔等人突破经典力学体系建立量子力学。

所以，人类该大力发展这种全新的颠覆性创新能力，在AI时代，有创造力的人会更稀缺、价值更高；而那些只能做模仿、模拟的人，以及传统以知识为导向、侧重学习和应用前人知识工具的教育模式下培养的人，会逐渐被AI取代。相反，擅长创新、能开拓全新领域或在交叉学科工作的人，其能力会越来越受重视。可以说，组合式创新vs零到一颠覆性创新，是人和机器的一大核心区别，而且我相信在相当长一段时间里，AI都很难实现零到一的创新。

第二个区别在于学习模式。AI的模型是通过预训练加有监督微调实现的，交付给我们使用时就已经训练完成，不会再随着环境变化重新学习；但人类具备“在线学习”的能力，能不断根据环境改变，调整策略、采用新方式学习。这种差异意味着，未来我们无论是培养学生，还是提升自己，都要注重快速学习、理解和重构的能力——要能快速理解新知识、新领域、新现象带来的变化，并做出快速反应。这和过去那种慢节奏的学习模式有本质不同，未来在工作和生活中，快速完成适应与调整，可能比追求慢速的完美更重要。

> **Q：在AI狂飙突进的时代，普通人如何避免被甩下或者替代？**

刘嘉：首先我想澄清一个概念：学习一定是终身学习的概念，所有人的知识过一段时间都会过时，不管是正在上学的中学生、大学生，还是像我这样在大学工作多年、在某个领域有一定积累的人，我们的知识都处于随时可能过时的状态，所以终身学习是必然的，这是一个大前提。如果从更具体的层面来说，在这个时代要想不被AI落下，我认为有三点很关键。

第一点，一定要善用AI这个工具。目前的竞争，更多不是人和AI的竞争，而是“会用AI的人”与“不会用AI的人”、“善于用AI的人”与“不善于用AI的人”之间的差距。AI现在是人类非常好的助手和老师，把它用在工作的各个方面，能极大提升效率。比如我写稿子，以前要逐字逐句从开头写到结尾，现在我可以把更多时间花在构思布局、搭建结构和创新点上，只要给出框架，就让AI帮我填充细节；再比如做绘画、动漫，我可以专注于奇思妙想的构思，而不是耗费精力在一笔一画的绘制上。

第二点，要借助AI培养批判性思维能力，让它成为我们的批评者。在现实生活和工作中，我们很难得到有价值的负面反馈或批评意见：老板批评我们“做得不好”，往往不够具体，没什么实际价值；朋友的批评又会因为顾及社交关系，不敢过于尖锐，怕伤感情。但AI不一样，我们可以主动让它批评我们、挑我们的刺、跟我们“抬杠”。在这种激烈的思辨过程中，人的认知会不断深化，而且因为AI没有情感关联，即便“抬杠”也不会伤感情，我们更容易接受这种批评。这其实很像苏格拉底强调的“产婆术”，通过辩论求真，AI能在这方面给我们很大帮助。比如我做研究时，提出一个新想法，不会先让学生尝试，而是先让AI来挑战、批评这个想法，直到它无法再提出新的质疑，我就知道这个想法可能是全新且可行的，这对保持进步、让观点更严谨很关键。

第三点，要让AI成为我们情感支持的伙伴。它能在我们生活、工作遇到问题时，提供情感上的解释和帮助。比如和同事发生冲突、老板对自己不满意时，我们可以向AI求助，一起探讨如何找到好的解决方案，它能在情感层面给我们有力支撑。

总结来说，在这个时代不被AI落下，首先要树立终身学习的意识，这是基础；其次要明白，现在的竞争对手不是AI，而是“会用AI”与“不会用AI”的差距，我们要充分利用AI，让它从助手、老师到批评者、再到情感支持伙伴，全方位助力我们完成工作、提升自我。

AI产品能不能火，全看创始人会不会当“网红”？这届AI大佬不拼代码了，个个都是隐藏的社交媒体达人

作者 华卫

小红书，似乎突然成了AI产品的“发迹”地。

今年开始，越来越多AI领域创业者选择在这里完成产品从0到1的冷启动。AI创作工具Flowith的CMO亲自在小红书高频率更新各类场景的使用体验，在小红书上收获大量粉丝，被用户们自发安利；专门做给老外的AI风水应用OCTA，从灵感过程诞生到发起内测群聊，通通在小红书完成；瞄准“拖延症”群体的AI效率神器plancoach，开发者把小红书当成了能和用户实时互动的产品日志和个人随感，一条内测招募视频都爆了10万+的浏览量。

创始人们纷纷以个人身份入驻，来分享自己的创业思考和产品灵感。公司团队则充分利用小红书的社交属性，下场招募志同道合的合作伙伴。投资人们亦主动出击，亲自来此寻找不错的AI项目。

近期，在小红书平台，还刮起一阵奇妙的“Ask me anything”风。平时在企业里扛技术、在高校里钻研究的AI精英们齐刷刷地发帖，介绍自己职位身份的同时积极向外链接。

有人说，一代人有一代人的机会，当下AI创业不仅身处风口，还拥有一项实用的“外挂”：社交媒体的流量杠杆。也正因如此，这届AI创业者仿佛都“进化”成了社交媒体达人。我们采访了几位创新AI产品的创始人，听他们讲述亲自主导产品社交传播的过程与经验。

不“投流”的AI产品，跑不出来

“现在的AI创业者必须接受一个新规则：如果你的产品在前48小时没能引发社交扩

散，就可能被判‘隐形死刑’。”被称为“欧洲版Cursor”的Lovable联合创始人Anton Osika在一次最近的采访中表示。成立仅6个月就以8000万美元价格被收购的Base44的创始人Maor Shlomo同样认为，真正让他们跑出来的，并不是技术多先进，而是对“传播力”的极致理解与执行。

这两款诞生于海外的AI产品，都成功跑出了市场，而它们背后的创始人们纷纷强调了社交传播的关键性。

“无论是在国外还是国内，都是如此。”好耶科技创始人兼CEO吴杰茜指出，现在AI产品的数量已经多到了一定量级，若不进行投流或营销，产品未必能到大家的视线里。

吴杰茜做了一款名为FilmAction的全流程AI影视及视频生成平台，主要对标影视级内容，能直出一分钟、甚至高达10分钟的视频，且最新推出了一键生成的核心功能，系统可在十分钟以内跑完全部流程，采用全AI Agent逻辑与全自动托管的制作模式。据悉，在用户构成上，FilmAction的C端普通用户数量最庞大，付费率也相当高。与此同时，目前已有不少机构、具备一定规模的影视工作室与影视公司在使用FilmAction，好耶科技还在与几家头部电影制作公司洽谈合作，计划将FilmAction应用于他们未来的电影项目中。

她表示，产品的传播度主要取决于两方面：一是产品本身的效果，二是在投流上投入的资金规模。如果产品不在推出后的前几分钟或前几天快速打开市场，难免会让人产生一些焦虑。当一款产品的口碑和知名度提升后，大家自然会更认同它。毕竟，即便产品本身再好，如果没有被大众熟知，就更谈不上让人们去使用了。而当越来越多人提及并使用这款产品时，它的口碑必然会进一步向好，用户规模也肯定会随之扩大。

“用户也学得很聪明，会抵触很多种营销。”AI在线生成PPT平台咔片的创始人兼CEO刘浩进一步提到。他认为，当前市场非常卷，流量获取不仅难度越来越大，成本也不断攀升。在这种情况下，传统的线性传播方式已无法媲美指数传播的效果。大家都希望，产品能具备传播力，实现一传十、十传百的扩散。但要做到这一点，有一个关键前提：产品对应的场景，无论是解决问题的场景还是用户使用场景，本身是否具备传播属性。

此前，在相关产品的用户采访中，他就发现，即便用户觉得产品好用且愿意主动分享，在某些场景下也会有所顾虑。比如职场场景中，部分用户会将好用的工具视为自己

的“独门绝活”，认为这是自身能力的体现，所以不愿分享给同事，担心失去这种优势。再比如，通常只有当产品本身有价值、好用或具备娱乐性时，才可能激发用户自发传播。还有些场景下，用户的核心需求是“用完即走”，不希望被产品打扰，这时强行附加社交属性、做裂变营销，反而可能违背产品初衷。

当然，并非所有场景都缺乏传播可能。刘浩表示，关键在于传播需围绕产品特性与功能场景自然展开，而非刻意营造。在咔片的协作场景中，传播就会自然发生：若要进行协作，用户必然需要考虑如何将内容分享给同事，他们也可以通过轻量化设计帮助用户搭建模板库、资源库，方便协作中的分享。

“这种围绕实际需求产生的分享，本质上就是一种有效的传播。反之，若为了追求传播力而在产品中强行加入各类裂变设计，很可能适得其反。”据刘浩介绍，咔片PPT的成品自3月末正式上线以来，截至目前仅用了约四个月时间，用户规模便已突破20万。从目前的转化情况来看，用户的付费转化率大致能达到20%。

新一代创始人们，都主持起“传播”大局

社交媒体给AI产品带来的价值似乎还远不止于助力冷启动。对于新一代创业者而言，他们正借此摆脱“闭门造车”的困境。以往，产品迭代要么依赖经验性洞察与预设，要么需反复开展用户调研才能找到方向。现在，他们不仅可以直接转化用户基于社媒反馈的真实需求，还可以预先开始对用户的使用引导和“教育”。

身兼创始人和CEO两职的刘浩对此深有同感。他透露，目前其职责之一就是关注产品迭代，同时更希望产品能细水长流，而非昙花一现。从传统角度来看，不少创始人容易聚焦在功能开发、产品迭代上，却忽略了传播的重要性。但像咔片这类产品，他们在打造过程中融入了不少理念性内容，即便希望用户使用时足够简单，仍需要把产品理念传递出去，让用户清晰认知到它好用在哪里。

“很多时候，我们会将重点放在产品本身。但除了把精力铺在功能上，我当前的主要工作还包括怎么把产品打磨得更加用户友好，以及能够创造一些能形成口碑效应的场景。”据刘浩称，他们在初期就开始关注到通过传统社交媒体获取公域流量的途径，且该途径已逐步铺开。但关键还在于产品力能否跟上，只有在编辑、演示等场景中切实为用户带来便捷与好处，口碑传播的过程才会自然而然形成。

而要引导这个过程，并非单靠组建营销团队就能实现，更需要创始人将产品理念、想法与创意转化为用户可接受、甚至可视化的传播语言。“我越来越深刻地意识到这一点，即便团队中有专业的营销人员，但他们在我们的产品领域中可能还是个小白，需要去从头教育。所以我需要先将这些理念与引导方向传递给团队，再让团队去创造可传播的方式与途径。”刘浩说道。

吴杰茜同样深度参与到了自家AI产品的传播环节中，关注的则是运营策略。她强调，现在早已不是过去那种推出一款产品，随便转发到朋友圈就能有效触达用户的时代了，因为朋友圈里的人群未必就是这款产品的目标受众。

不同的AI产品在运营上会存在一些差异，尤其是“AI+特定行业”类产品，关键得看产品更多面向B端还是C端。如果是面向B端，重点做好企业方的商务对接即可；但如果是面向大规模C端的产品，就需要先明确产品所属领域，比如是生产力工具、编程类工具，还是FilmAction这样的视频制作类工具，亦或是偏文艺的动漫类、二次元陪伴类产品，不同领域都有其对应的核心受众。

“先研究透受众之后，我们才会进一步思考是否值得在该受众圈层内投放广告，以及寻找对应的网络博主进行宣传。”这是吴杰茜在传播上的核心思路。并且，她表示，在投放广告时，抖音、快手、小红书和微信视频号等这类短视频创作者聚集的平台，带来的用户互动数据表现更为突出，点赞、收藏等指标的量级会比其他类型的平台要高一些。

“吆喝”也需有红线

“随着AI产品数量不断增加，社交传播的权重确实会越来越重，这是必然趋势，毕竟产品数量多了，传播层面的竞争也会更激烈。在未来几年内，社交传播的权重还会持续上升，随着AI产品数量的进一步增长，其重要性必然会越来越高。”吴杰茜说道。

但她表示，社交传播无法替代技术壁垒。每个产品固然都有其独特的技术壁垒，有些产品可能实现起来简单，但在创意层面难度很高。评价一款产品的优劣需要考虑多个因素，技术、创新性、独特性、能否解决用户需求以及产品设计、用户粘性等都是重要的考量维度。这些因素综合在一起，共同构成了AI产品成功的关键，而社交传播则是其中一个重要大类。

刘浩则认为，传播力与技术壁垒之间还可以相互增强。传播力决定产品能否“跑起来”，产品有传播力证明它是有用的。而技术壁垒决定的是产品能否“跑得更久”，若缺乏技术壁垒，很容易被竞争对手超越。

除此之外，他向我们透露了现在业内的一个普遍现象：很多产品尤其是获得投资后，都希望能快速崛起，像折线图一样迅速达到高峰并维持住。因此，无论是创始人主动推动还是受投资人影响，常会通过创造概念、尤其是噱头型概念打造营销点，再集中砸流量、投广告，试图稳住流量。

而在咔片内部，他们设定了两条不能逾越的红线：第一，不能以牺牲用户使用稳定性为代价去追求传播，否则得不偿失；第二，不希望用噱头功能替代对高频刚需体验的打磨来做传播，很多噱头功能看似美好、也能吸引注意力，但就像部分AI产品一样，用户新鲜体验过后，真正会持续使用的寥寥无几。

“我们不希望咔片靠噱头功能或牺牲稳定性来换取传播，因为这并非长久之计，更希望让产品在实际场景中真正为用户带来改变与优势，从而自然形成持久的传播力。”刘浩表示，他们之前有过类似的经验。做iSlide时，前几年他们在广告费上的投入是0，早期依靠PPT圈子里的垂直渠道和达人推荐，慢慢建立起了口碑传播。但现在AI赛道的应用层产品太多了，还靠传统方式可能早就被“吆喝”声给淹没下去了，甚至可能还要比比嗓子。

至于AI产品是否会因过度依赖病毒式社交传播而偏离核心价值，吴杰茜的看法是：这并非非黑即白的问题。产品质量与社交传播同样重要，不能孤立看待某一方，也谈不上“过度依赖”其中某一方。在她看来，产品首先必须自身质量过关，这样营销才能事半功倍。即便再差的产品，大量营销后或许能依靠投入的成本获得一定保底用户量，但要让用户持续使用并愿意付费，就必须满足用户需求。只有这样，产品才能真正拥有追随它的用户。

初创产品的终极优势：时间差

QuestMobile数据显示，截止到2025年8月，移动端AI应用用户规模达6.45亿，PC端达2.04亿。在热闹的AI应用赛道，各大科技巨头的身影同样频繁涌现。阿里的夸克、360的纳米AI搜索、字节的即梦AI、快手的可灵AI等，都在AI应用排行榜持续占据前列。

“说到底，竞争的核心终究要回归产品力本身。”刘浩谈到，当大厂切入某一领域时，对创业公司往往意味着生存压力加剧。大厂在技术能力、资源储备、专业研发团队等核心维度，天然具备难以逾越的优势。

在这样的市场环境下，国内赛道的竞争愈发激烈，尤其在AI应用层，“内卷”已延伸至商业模式层面：不少玩家为争夺用户，开始推出“终身会员”。但大厂从不会涉足此类业务，终身会员的长期服务成本、技术迭代投入等根本无法精准核算，本质上不符合其规模化、可持续的商业逻辑。除此之外，还有很多公司会通过制造噱头打造产品亮点，再砸流量、投广告来计算ROI，靠着终身会员提高客单价，从而提升ROI，只要数据正向就持续投钱，再通过讲故事拉流量、拿投资，形成一个循环。

但在刘浩看来，这种路径并不正常可取。创业公司真正的优势在于能沉下心深耕特定领域，围绕单一高频场景做到工程化极致，把场景细节的方方面面都考虑周全。就研发而言，它本是一个循序渐进的过程。相比大厂一下子就把产品做得八九不离十、完美呈现，作为初创公司的他们会选择先将一两个优势功能放出来。

这时，如何留住用户、提升用户留存率，是更关键的问题。以AI PPT产品为例，当下的“敲门砖”是要和其他产品比拼“AI生成PPT哪家强”，至少要让AI生成的PPT看起来更像专业的PPT。“当60%-70%的需求集中在AI生成环节时，我们更会关注剩下30%-40%的需求，很多用户生成PPT后可能会直接下载离开，但仍有部分用户会留存下来，他们会有替换素材、便捷换肤等需求。”

刘浩表示，这也正是咔片的定位所在：不只是关注前端的AI生成过程，更聚焦PPT演示的全流程。AI已经能把PPT从0到1的生成过程缩短到几十秒，但接下来的二次编辑、分享协作、演示放映等环节，他们也要做得更完善，同时借助AI进一步为用户提供支持。只有这样，用户才会真切感受到线上产品的优势，慢慢改变线下使用习惯，愿意在产品上投入更多时间，将其应用到更多场景中。

“创业公司更愿意聚焦单一或垂直场景、深耕用户核心需求，通过打造高频、便捷的核心功能建立用户粘性，再以此为基点逐步将优势功能从1个拓展到2个、3个，实现产品能力的稳步完善。这种 ‘以场景挖需求，以功能筑壁垒’ 的路径，是我们在‘卷终身会员’ 等同质化竞争之外真正能立足的核心竞争力。”

同时，刘浩“现身说法”道，在这类高频单一场景中，无论是产品团队、创始人还是初创成员，往往本身就是场景的深度使用者、甚至是领域内的“使用达人”。这种天然的用户视角，能让团队更精准地捕捉用户习惯、洞察潜在需求，也为其深耕细分场景、打造差异化优势奠定了先决基础。

吴杰茜对此也补充道，大厂要了解外界需求，往往需要行业资深从业者进入大厂体系，或是通过外部采访等间接方式。但AI初创公司通常会有特定行业的人才与技术人才共同合作，开发“AI+行业”类产品，这样不仅能更快地推进项目，对需求的理解也会更加深刻。

此外，她提到，无论是优化产品功能、调整发展路线，还是快速找到差异化突破口，初创公司在行动力和反应速度上都占据绝对优势。首先，大厂的架构很完善，但组织也比较复杂。即便再强调扁平化，庞大的人员规模还是会导致存在层层决策和信息传达的情况。其次，大厂做决策时需要顾虑很多因素，比如整个集团的战略规划、不同的资源池分配，以及资金要投入到哪些方面等，都需要慎重考量。

“我们要抓住这样的时间差，在大厂尚未关注的短期内，把细分场景的二三级细节打磨好，帮助用户建立使用习惯。要知道，用户习惯一旦形成就很难改变，后续即便有其他产品出现，要么需要重新对用户进行教育，要么产品本身要足够出色才能吸引用户转移。”刘浩总结称。

10年经验，不敌AI 10秒？对话5位顶尖架构师：AI不会替代我们，但会淘汰旧的我们

作者 李冬梅

> **采访嘉宾：**费良宏、付晓岩、沈剑、武艳军、杨凌云（采访不分先后，按拼音首字母排序）

过去十年，架构师被誉为互联网企业的“定海神针”，他们以宏观视角统筹系统、以前瞻眼光塑造技术蓝图，是数字化时代不可或缺的技术中枢。而如今，随着AI技术的深度融入和云原生体系的全面普及，架构师这一角色正在迎来一场前所未有的转型契机——不是被替代，而是被重新定义。

AI编码助手、智能开发平台、低代码工具的出现，正在释放架构师从繁重重复劳动

中的自由，让他们有机会专注于更高层次的系统设计与创新决策。GitHub Copilot、Cursor、Claude Code等新一代AI工具在多个研究中被证实可提升开发效率30%至50%，这意味着架构师可以把更多精力投入到战略性技术规划、跨系统协同和业务创新之中。同时，云厂商的标准化“最佳实践”也为架构师提供了更丰富的组件与模型基础，使他们能以更快速度构建、验证并落地复杂架构方案。

在这一进化过程中，新的职业形态正在萌芽。AI架构师、模型工程架构师、智能平台设计师等新角色正不断涌现，他们不仅熟悉传统系统设计，更懂得如何统筹协调算力、模型与数据，成为企业智能化转型的核心推动力。

在机械工业协会出版社主办、InfoQ协办的“数智时代的软件架构峰会”上，我们策划了这期深度对话，并邀请了这一转型过程中的亲历者与思考者——来自一汽大众的武艳军、亚马逊云科技的费良宏、数孪模型科技的付晓岩、凯捷中国的杨凌云，以及曾任职58同城和百度的资深架构专家沈剑共同探寻：当AI能够生成代码、绘制架构图甚至提出解决方案时，人类架构师的独特价值何在？他们该如何重塑自身角色，在智能化时代继续担当技术创新的引领者？

在这场多人对话访谈中，专家们认为，AI大模型爆发后引发的技术转型不仅仅是工具层面的革新，更是角色定位的根本性转变。亚马逊云科技首席云计算技术顾问费良宏指出，AI正在推动生产力与生产关系的重新适配，传统“以代码为中心”的工作模式正在被颠覆。编码不再是重点，架构师需要将工作重心转向产品战略与公司战略层面。那些仍然将精力局限于技术选型、代码评估的传统架构师，很可能在AI快速发展的环境中失去竞争力。

在这个过程中，新的职业形态正在萌芽。AI架构师、模型工程架构师、智能平台设计师等新角色不断涌现，他们不仅要熟悉传统系统设计，更要懂得如何协调算力、模型与数据流，成为企业智能化转型的核心推动力。架构不再只是“系统蓝图”，而是智能生态的设计语言。

凯捷中国区总监，高级咨询师杨凌云从另一个角度补充道，架构师的价值重心正在从“个人经验”转向“能力杠杆化”。AI能够帮助架构师快速完成技术调研、方案模拟和知识更新，但这并不意味着架构师的价值被削弱。恰恰相反，AI的加入让“人”与“组织”层面的议题变得更加重要。架构师需要将AI能力与组织战略、文化相融合，这

恰恰构成了他们在AI时代难以被替代的“护城河”。

在实践层面，专家们也分享了各自的观察。一汽大众首席架构师，《企业架构驱动数字化转型》作者，《架构现代化》译者武艳军提到，虽然目前AI在架构设计领域主要还是起到辅助作用，比如自动绘制架构图、进行方案检查等，但其在企业架构治理方面已经展现出显著价值。通过AI快速分析海量架构资产、进行一致性检查，大大提升了架构治理的效率。同时，AI带来了架构范式的深刻变化，AI原生架构成为新时期企业架构研究和实践的重点。

北京天润聚粮咨询服务有限公司执行董事、总经理；数孪模型科技高级副总裁付晓岩则提出了一个更具前瞻性的观点：未来的架构师将更多地扮演“语义管理者”的角色。随着大模型能力的持续增强，架构师需要帮助机器在企业特定的业务语境中准确理解业务逻辑、系统结构和目标需求。这种“语义管理”能力将成为连接人类知识体系与智能系统的核心桥梁。

面对这些变化，曾任58同城技术委员会主席，百度架构师沈剑给出了最直接的建议：不要观望，而是要去了解并使用这些工具。技术发展的浪潮从不会因为任何人的抗拒而停下脚步，只有始终保持开放心态、持续学习、勇于拥抱变化的技术人，才能在下一轮竞争中保持领先。

从宏观视角来看，这场转型不仅仅是技术层面的升级，更是架构师价值定位的深度重构。当AI能够自动生成架构蓝图时，人类架构师的独特价值将更加体现在战略判断、组织协调和创新能力上。正如多位专家所言，AI不会取代架构师，而是让他们从“绘制蓝图的人”转变为“定义价值与校准系统的人”，在智能化时代继续发挥不可替代的作用。

采访嘉宾

- **武艳军**，一汽大众首席架构师，《企业架构驱动数字化转型》作者，《架构现代化》译者
- **费良宏**，亚马逊云科技首席云计算技术顾问，《Effective软件架构》《软件架构决策之道》译者
- **付晓岩**，北京天润聚粮咨询服务有限公司执行董事、总经理；数孪模型科技高级副

总裁，《聚合架构》作者

- **杨凌云**，凯捷中国区总监，高级咨询师，《解决方案架构师修炼之道（原书第2版）》译者
- **沈剑**，曾任58同城技术委员会主席，百度架构师。互联网架构技术专家，“架构师之路”作者，《分布式系统模式》译者。

以下基于采访实录整理，InfoQ经过不改变原意的编辑

AI时代下架构师角色的重新定义

> **InfoQ：在如今全社会都在进行数智转型的大背景下，架构师在转型中的定位是否发生了变化？随着AI技术向业务中心逐渐渗透，企业中出现了很多新角色，比如“AI架构师”或“模型工程架构师”这类岗位，您是否接触过这类新角色，能否举例说明下他们在企业中的价值，您又是如何看待这类新角色的？这些岗位和传统架构师的分工有何不同？**

费良宏：AI极大降低了代码生成、设计生成的成本，推动生产力与生产关系重新适配，使传统“以代码为中心、人生成代码”的工作模式被改变。编码不再是重点，架构师需转移工作重心、调整角色、大幅更新技能，才能在AI快速发展的环境中立足，否则传统架构师与开发人员可能被取代。

从历史规律看，技术革命始于工具，此次AI革命不同于以往仅带来边缘性工具，而是成为商业价值核心，取代部分人类原有定位，要求包括架构师、程序员在内的从业者重新思考自身定位，其冲击远超以往技术变革，易引发焦虑与对未来的不确定性认知。

过去，团队中通常会划分出架构师、开发人员、项目经理等传统角色。我认为这类角色在未来可能依然存在，但它们的分工、所扮演的角色以及每个角色的内涵会发生诸多变化。从最直观的影响来看，程序员群体受到的冲击可能最大，低端程序员的生存空间大概率会被极大压缩，不过目前对高级程序员仍有强烈需求。需要注意的是，这一判断基于AI当前的发展现状，而回顾过往，AI的发展速度远超预期，例如OpenAI在2024年4月发布GPT-4o，仅一年后就推出了GPT-5，其智能水平提升了足足一倍。按照这样的发展速度，即便当下我们重点讨论的是低端程序员，未来其他领域技术人员的生存空间也可能被大幅压缩，这是一个大概率会成为现实的问题。

再看架构师这一角色，以往架构师的工作重点围绕功能与代码展开，核心任务包括解决性能问题、进行技术选型、开展代码评估以及需求分析等。然而，这些耗时且具有重复性的工作，在很大程度上可能会被AI带来的新技术所替代。在此背景下，架构师需要将更多精力与经验从以代码为核心、技术选型以及传统认知中具有挑战性的任务上转移出来，更多地转向产品战略与公司战略层面。未来，这类角色或许仍会沿用“架构师”的称谓，也可能被称为“增强架构师”或“新一代架构师”。

未来团队角色必然会经历重新组合，甚至可能如你所提及的那样出现许多新角色，但我目前无法确定具体会出现哪种新角色。从更大可能性来看，未来这些角色或许依然存在，但每个角色被赋予的含义会有所不同。比如，架构师本身可能同时具备程序员的属性，而程序员也可能承担起架构师的部分重要职责，如此一来，未来的开发人员有可能同时也是架构师。简言之，相关角色的概念可能会继续保留，但它们的内在本质与定义会被彻底重塑。此外，部分低端岗位可能会逐渐消失，同时角色高度融合的趋势会愈发明显——在一个团队中，低端程序员、高端程序员、前端人员、后端人员、架构师、产品经理等角色会大幅融合，甚至有不少角色会慢慢消失，这些变化都有可能在未来成为现实。

> **InfoQ：过去架构师的核心竞争力在于“经验积累”和“系统把控力”，如今这些能力正在被“智能化工具”部分取代或重构。AI时代是否存在“一人需掌握多个领域内容”的发展趋势？**

费良宏：在“一人空间”模式下，个体依靠一众AI智能体协作完成工作，此时如何平衡标准化与创新是核心问题。这种模式本质上带有标准化属性，而创新则是突破标准化、实现价值的关键。在当下时代，个体的核心价值恰恰体现在差异化上——只有做出与他人不同的成果，才能凸显自身价值，若陷入同质化，个人价值便难以衡量。

这种差异化的实现，关键在于人而非AI。尽管AI能持续帮我们提升效率，但目前它尚无法创造出足够的差异化，因此差异化仍需依赖人主动运用创造性思维、批判性思维，或是结合过往经验来达成。在此语境中，人始终是主体，AI仅处于从属地位，扮演工具角色。不过，AI凭借其高效的执行速度和效率优势，能在很大程度上协助我们验证想法、实现方案、生成原型，二者形成相辅相成的关系。

这一过程类似于多年前敏捷模式中的“结对编程”：理想状态下，一人负责思考，

一人负责编码，双方凭借各自优势相互配合。但在过去，很难找到能完全匹配的结对工程师，而如今AI却能完美胜任这一角色。只要我们充分发挥思考能力，提出足够多的创意与想法，再由AI将这些想法落地实践，这种模式完全可行。

因此，从长远来看，我们需要转变思维，将AI从单纯的“工具”转变为“伙伴”。学会驾驭这个伙伴，让它成为开发过程中真正的“结对工程师”，这不仅是至关重要的一点，也是我们实现创新的关键步骤。

杨凌云：我认为架构师这一角色在AI时代将经历一次深度重构。过去，架构师的能力主要依赖于个人经验——踩过的坑、积累的项目实践和对复杂系统的直觉判断。而如今，随着AI技术的引入，架构师的价值重心正逐渐转向“借助AI实现能力杠杆化”。

具体来说，AI可以帮助架构师在多个层面提升效率与洞察力。例如，在技术调研阶段，AI能够快速整合和分析大量资料；在系统架构设计中，它能进行模拟、推演，甚至提供多种可行方案；在知识更新方面，AI还能帮助架构师迅速吸收前沿信息，从而拓展思维边界。这些能力使得架构师不再局限于个人经验，而能通过AI实现更高层次的创新与决策。

但与此同时，AI的加入也让“人”与“组织”层面的议题变得尤为重要。AI能力要真正发挥作用，必须与组织的战略目标和文化相适配。架构师在这一过程中，承担着关键的桥梁角色——他们需要把AI的能力嵌入组织体系，推动文化与技术的融合，确保AI真正成为组织能力的一部分。这正是AI难以取代的核心价值所在，也是架构师的“护城河”。

此外，架构师的角色也在从“技术权威”逐渐演变为“AI总控师”。他们不仅要判断AI生成的方案是否符合企业规范，是否存在潜在的风险或不合规行为，还需要利用知识库不断完善AI的理解边界，确保输出的可靠性与一致性。通过这种方式，架构师可以在AI的辅助下持续增强自身能力，完成从技术专家到战略决策者的跃迁。

所以我认为，AI不会取代架构师，而是成为他们最有力的工具。未来的架构师，将不再是单纯的系统设计者，而是懂技术、懂组织、懂AI协作的综合型战略角色。

> **InfoQ：AI是否在改变架构设计的流程？例如需求分析、系统建模、性能评估、**

> **持续交付等环节中，哪些部分最可能实现“智能自动化”？能否举一些具体的企业中的实践例子？**

武艳军：确实，以大模型为代表的新一代人工智能技术发展非常快，在软件领域已经产生了显著影响。目前AI的应用主要集中在代码生成、测试等环节，可以带来10%-40%的效率提升。在需求分析上，它同样可以发挥一定作用。比如，它能高效分析大量资料、辅助调研、生成初步报告等。

在架构领域，由于架构设计的复杂性，AI目前主要发挥辅助作用。比如，架构师提供设计思路后，AI可以帮助自动绘制架构图，或者根据描述生成初步架构方案。当然，目前来看，架构师仍然掌握着整体设计思路和创新的主导权，AI更像是一个在细节层面提供支持的助手。

> **InfoQ：在您的工作中，是否已经开始使用像GitHub Copilot、Cursor、Claude Code这类AI辅助开发工具？它们在架构设计或研发协作中发挥了哪些作用？**

武艳军：这些工具我们都有在尝试使用，包括国内一些AI编程助手。目前主要还是从个人层面使用，比如工程师或架构师个人利用这些工具来提高效率。它们现在发挥作用的主要领域还是在编程层面。

在架构设计上，除了这类工具，还有一些其他形式的AI应用，比如智能体（Agent）或AI工作流，它们能在特定环节提供辅助支持，但总体上还处于探索阶段。

但也不可忽视大模型的一个特点是确定性不强。它根据理解来生成结果，可能存在一定偏差。这就要求我们在提出任务时，提示词要写得非常清晰、具体。如果能准确表达需求，模型往往能给出不错的结果；但如果问题过于宽泛，它可能会“跑题”。

在实践方面我们目前还没有直接用大模型来生成架构设计，但会用它来做方案检查和预评审。比如，验证一个方案是否符合规范、有没有遗漏之处。在这类初级、重复性较强的工作上，大模型的表现还是不错的，能帮架构师节省不少时间。

可以说，AI技术的发展在促进企业架构迭代方面发挥了很大作用。AI能帮助企业大幅提升架构治理的效率。具体而言，企业在做架构治理时，会积累大量架构资产，这些资产由不同团队在不同时期完成，存在关联复杂、一致性难以保证等问题。过去我们靠

人工分析，工作量大且容易遗漏；现在通过AI，可以快速完成这些资产的分析、对比和一致性检查，显著提升效率。

沈剑：我认为，AI工具首先应该被自然地引入到技术人员的日常工作之中。毕竟，它的本质仍然是一种工具。回顾技术发展的历程，我们已经历过许多“工具化”的演进：从编程语言、设计模式，到架构模式，再到各种集成开发环境（IDE）——它们都在不断帮助我们提升开发效率与代码质量。AI工具的出现，只是这一演进的又一次延伸。

工具的核心价值，在于提升效率与质量。正如编程语言让我们更高效地表达逻辑，设计模式帮助我们更合理地组织代码结构，AI工具同样能够在编码与架构设计过程中，提供智能化的辅助。例如，不论是Copilot这样的代码助手，还是其他具备生成与分析能力的AI工具，它们都能有效提升编程的质量与速度，帮助开发者更高效地完成架构设计与实现。

因此，我们应当以开放的心态去了解、使用并融入这些工具。AI并不是要取代开发者，而是成为开发者的新型“生产力伙伴”。善用AI工具，就意味着在新的技术周期中，持续提升自身的编码效率与代码质量。

> **InfoQ：AI的发展也让企业架构范式发生了变化。在AI时代，企业架构需要怎样去适应这种变化？**

武艳军：确实，AI的发展让企业架构进入了一个新的阶段。过去我们基于规则和判别式逻辑设计系统；而现在，生成式AI能主动产生内容，甚至驱动业务，这带来了架构范式的深刻变化。因此，企业架构方法也需要升级。之前我们关注云原生架构、微服务架构，现在重点研究“AI原生架构（AI-Native Architecture）”。所谓AI原生架构，是指以AI大模型为基础、以生成式架构为核心，以数据飞轮（“数据-模型-反馈闭环”）驱动自我进化，构建、运行和管理应用的架构范式。以大模型为基础，是因为AI原生应用的价值主要来自大模型，大模型是应用的核心决策引擎。以生成式架构为核心，是因为AI原生应用以大模型生成的文本、图像、代码等内容为结果来驱动业务。数据飞轮驱动的自主进化，是指AI原生应用构建了“数据-模型-反馈优化”的数据飞轮，在使用当中不断获取反馈数据并优化模型，从而提升AI原生应用的表现。

从企业架构理论来看，AI原生架构不只是涉及某个架构域，也不是跳出于4A架构的

单独的架构域，而是和4A架构都有着密切的关系。比如：

- 在业务架构层面，要找到高价值的业务场景，分析业务角色，并开展业务流程的智能化改造，业务角色未来不只是人类员工，还可以是AI员工。
- 在数据架构层面，要考虑非结构化数据治理、构建高质量数据集、企业知识的向量化和图谱化，以及选择和训练适合企业业务需要的各类大模型。
- 在应用架构层面，要构建各类基于大模型的AI应用，比如RAG、AI工作流、智能体等，并研究如何实现AI应用之间的集成、编排和协同、AI应用与现有系统的协同。
- 在技术架构层面，要考虑建设算力基础设施，包括通过云服务方式获取，或者建立自有的智算中心。

这些都是AI时代企业架构升级必须面对的新课题。

从云原生架构和AI原生架构两者的关系来看，AI原生架构不是对云原生架构的直接替代，长期来看两者会以一种混合形态存在，云原生架构可以为AI原生架构的运行提供支撑。

AI协作模式下，架构师面临的机遇和挑战

> **InfoQ：我们观察到一个现象，就是现在很多企业研发体系在转向“AI协作模式”（AI-assisted collaboration），这对架构师的团队协作方式带来了哪些冲击或机遇？**

杨凌云：我认为，首先最大的变化来自协作模式本身。过去，架构师通常被视为技术权威——由他来做决策，而机器只是执行者。而现在，这种关系正在发生转变。AI不再只是被动的工具，而是成为架构师决策过程中的重要辅助者。架构师需要学会借助AI的能力，快速获取信息、生成方案，然后在此基础上进行判断与取舍。他们关注的重点，也从具体的项目实现、代码编写等细节，转向更高层次的规范制定与战略执行。

其次，从团队协作层面来看，以往架构师往往要带领多个小团队协同推进不同模块的开发。而如今，AI的引入让这种组织结构正在被重新定义。过去可能需要多人配合才能完成的任务，现在一个人加上多个AI Agent就能实现。比如我们曾遇到过一个案例，

过去需要几位工程师协作数月才能完成的项目，如今同事仅用两个月就独立完成了，而他的“团队成员”正是几个协同工作的AI Agent。

因此，架构师在新的协作体系中，不仅要管理人，还要学会管理AI——如何让这些AI Agent高效运行、在团队中分工明确、保持合规与稳定输出，这将成为新的核心挑战。

第三个变化在于效能评估体系。传统企业的效能管理，通常会设定“北极星指标”及一系列分解指标，从人力产出、质量、效率等维度进行衡量。而在引入AI之后，评估逻辑将发生根本变化——我们不仅要衡量人的绩效，还要评估AI的效能：AI能帮我们完成哪些任务？哪些工作仍需人来承担？人机协作的边界该如何界定？

这对企业的KPI体系乃至整个组织管理模式，都是一次深刻的变革。

总体来说，AI正在让架构师的角色从“指挥者”变为“协调者”，从“亲力亲为的技术权威”转向“人机协同的组织设计者”。这既是挑战，也是一次重塑自身价值与能力边界的机遇。

付晓岩：我近期主持的工作坊命名为“双智时代架构未来”。在最近一次客户培训中，我们将第一讲的标题定为“双智时代，还能否架构未来？”——这其实引发了一个更深层的思考：在人工智能快速发展的背景下，架构师自身是否仍有未来？未来的架构又该如何定义？

正如您所提到的，大模型作为一种高效、智能化的工具，正被企业广泛引入研发体系。无论是为了加速开发进程，还是提升最终软件的效能，企业都希望将这一技术有效整合进来。

然而，引入大模型必然涉及架构层面的调整。首先，从技术架构角度看，我们需要重新审视AI在整个技术栈中的位置。目前，很多企业倾向于将其定位为“核心心脏”，并强调其他技术都应与人工智能进行融合。

进一步延伸到应用架构层面，问题则转变为：如何将基于AI的组件与传统业务系统有机结合？例如，当前兴起的“智能体架构”正是研究如何以非侵入的方式，将人工智能能力嵌入现有系统。在这样的架构中，一个智能体既可以调用传统服务，也能接入大模型，并在同一工作流中将二者协同起来，从而避免对原有系统的破坏。

如果我们希望应用形态朝这个方向发展，那么在软件架构设计阶段，架构师就需要明确智能体技术在整个体系中的层级定位，并设计相应的结构。更进一步，在业务需求提出阶段，我们就应思考：在哪些业务环节需要模型支持？应以何种形式支持？最终是形成智能体形态，还是催生更原生的AI应用？

因此，可以说，大模型的引入对软件全链路都产生了深远影响。同时，开发人员的编码方式也在发生变化：从过去手动编程或复制粘贴，逐步转向由大模型支持的智能化生成——从代码补全，到大段代码的自动编写。这也将推动开发管理方式的变革。

总体来看，从功能定位、架构形态、技术栈组合，到最终实现方式，整个软件架构体系正经历重大重塑。这就要求架构师以更开放的心态积极拥抱这一技术，因为其最终的影响形态尚未定型。

大模型本身仍在快速发展，我们目前所熟悉的协作方式，在三年或五年后是否仍然适用，尚未可知。因此，架构师不仅需要思考如何将AI融入当前设计，更要以长远的眼光，持续关注技术演进，并主动应对未来的变化。

> **InfoQ：其实现在AI技术已经被很多企业放在核心位置去考虑，但一个不可忽视的问题是——目前阶段的大模型还会有一一些短板，比如它会有“幻觉”，那我们怎样才能毫无顾忌地信任它呢？**

付晓岩：大模型的短板确实很明显，我自己在用大模型做自然语言编程时，就经常被它的短板“坑”，核心问题就是“幻觉”。

比如，它工作着工作着可能就忘了自己要做什么，这时候还得再跟它“聊两句”，把话题拉回正轨；有时候它“一高兴”，甚至会直接删掉数据库，所以，大模型的“幻觉”问题是很突出的。

要解决这个问题，首先需要做好编程工具的发展。我今年一直认为，编程工具是这个领域里一个非常重要的新“战场”。现在国内很多大厂都在做相关探索，国外像Cursor这类工具也在布局这块。

好的编程工具能在一定程度上抑制模型的“幻觉”行为：通过引入规则、控制过程，减少这类问题的发生。我自己的使用感受很明显，今年三、四月份时，这类工具的效能

还很低，但到了五、六月份，效能就有了大幅提升，现在和它对话，很多环节已经非常顺畅了。

当然，工具只是一方面，另一方面还得靠模型自身的进步。每一代基础模型的迭代，都会带来很多突破。比如现在宣传的GPT-6，里面就有很多让人期待的点，甚至能实现自我学习、自我适应。

其实，对大模型的信任，有时候和人与人之间的信任很像。极端点说，你很难完全信任一个同事，人尚且如此，更何况模型。所以，我们对大模型要保持“适度信任”，同时时刻警惕它可能出现的问题，再通过工具和环境的建设，适当控制、抑制这些问题。但不能因为有这些短板，就不敢用它——我接触过一些技术人员，总说“这个不行、那个不行”，然后就不再用了，退回到自己熟悉的“安全方式”里。

但这样做会导致“脱节”，尤其是现在大模型的进步速度太快了。我今年使用时明显感觉到，它每个月的能力都会有变化；如果一段时间不接触，你可能根本不知道它进步到了什么程度，甚至无法想象它的突破。而且技术发展往往有“临界点”，到了某个节点可能会有大的跃迁。如果等它完成跃迁后再去拥抱，其实就有点晚了，不如跟着它一起慢慢成长，学会逐步信任它。

> **InfoQ：未来的架构师是否需要具备更多“AI产品经理”或“智能平台设计师”的特质？例如对模型选择、Prompt设计、AI工作流编排的理解？**

沈剑：我认为，随着时间的推移和技术的不断演进，架构师确实需要具备越来越多元的能力。无论是拥有AI产品的视角，还是掌握更多智能工具的使用方法，这些都将成为未来架构师的重要素养。

回顾技术发展的历程，每一个阶段都有新的技术浪潮。AI是近几年最热门的方向；再往前，我们经历过Web3.0的探索；更早之前，是移动互联网时代的变革。每一次技术更迭，都会催生出新的角色和能力需求。而真正优秀的技术人，往往是那些始终保持开放心态、持续学习、勇于拥抱变化的人。

因此，对于未来的架构师而言，不仅要理解传统的系统架构、设计模式与工程方法，还需要具备跨界思维——懂AI产品逻辑、理解模型特性、熟悉智能平台的设计理念。通

过学习这些新兴工具与模式，才能不断提升自身在编码效率、架构质量和系统设计创新上的能力。

简而言之，未来的架构师将不再只是“搭建系统”的人，而是能够连接AI、产品与工程，持续推动智能化架构落地的复合型人才。

> **InfoQ：现在AI的大时代已经到来，很多企业也在积极引入AI技术，把它融入到自身的技术架构中。在这个过程中，企业通常会遇到哪些技术上的或组织管理上的阻力？**

沈剑：无论是早期的移动化浪潮、Web3.0的兴起，还是如今的AI热潮，我认为企业首先要做的，就是主动去拥抱它。技术的发展趋势是不可逆的，最大的阻力往往并不来自技术本身，而是来自人，尤其是来自组织内部心态的保守与惰性。

在我看来，最需要转变的是那些处在关键位置的决策者——包括企业负责人、技术一把手、CTO、架构师等。很多时候，技术人容易停留在自己的舒适区：熟悉PC时代的，就继续坚持传统架构；擅长Web3.0的，就执着于区块链的逻辑；面对AI这样的新趋势，反而会本能地排斥或犹豫。

但历史已经一再证明，技术浪潮不会因为任何人的抗拒而停下脚步。移动化是如此，Web3.0如此，AI亦然。拒绝拥抱新技术，不仅会让个人掉队，也会拖慢整个组织的转型步伐。

因此，我认为“开放的心态”才是突破技术变革阻力的关键。只有当中高层领导、技术负责人、架构师率先打开思路，主动去了解、去尝试、去使用AI，把它融入到组织流程和工具体系中，后续的调整与落地才会更加顺畅。

归根结底，淘汰我们的，从来不是技术本身，而是那些更早拥抱新工具的人和团队。如果我们保持封闭，迟迟不去学习、不去应用，那么最终被取代的，可能就是我们自己。

所以我想强调的是——保持开放的心态，积极去学习、去拥抱、去实践，把AI真正引入到日常工作中。谁能更早地理解和运用这些新技术，谁就能在下一轮竞争中领先一步。

> **InfoQ：您如何看待“架构标准化”与“创新空间”之间的平衡？AI和自动化是否会让架构师的个体创造力被削弱，还是反而得到强化？**

费良宏：我认为未来对架构师的要求会大幅提升，远超当前标准，主要体现在技术实力与软实力两个核心方面。

在技术实力上，首先要求架构师具备扎实的技术基础，涵盖设计范式、设计模式、设计理论、开发能力及架构常识等核心内容；其次，必须掌握AI相关能力，像大模型、Agent、Prompt Engineering等都属于必备技能范畴；最后，还需拥有敏锐的学习能力与技术嗅觉——AI时代的技术迭代以天为单位，每天都会涌现新概念，甚至颠覆以往认知，若缺乏足够的学习能力和技术敏感度，架构师将难以持续成长。

而在软实力方面，要求则更为严苛。第一是批判性思维，面对大量不确定的新事物，架构师需具备筛选与辨别能力，判断哪些内容对自身有价值、哪些未来可能无用，当前时代对这种思维的需求远胜以往；第二是沟通能力，人际沟通是AI无法替代的，架构师需具备说服力，让同事、合作伙伴、领导及客户认可自己的决策，过去对架构师的要求多聚焦于技术技能，如今因部分技术功能被AI承接，沟通能力的重要性愈发凸显；第三是对未来的预见能力，当下的准备是为未来奠基，若无法合理判断未来趋势，很可能错失机遇，这一点对需要制定长期技术方案的架构师而言尤为关键。

未来架构师需同时满足技术实力与软实力两方面的综合要求，才能适应AI时代的发展需求。

“软实力”为什么如此重要，因为在这个时代，技术水平固然重要，但它划定了一个人的“下限”，而软实力则真正决定一个人的“上限”。想要突破自身的上限，就必须从当下做起，在每一个细节中打磨自己，不断提升综合素养，而不仅仅局限于钻研代码、开发与技术本身。

> **InfoQ：在行业层面，您观察到哪些企业在“AI+架构”融合方面走在前面？他们的经验有哪些值得借鉴？**

费良宏：由于这项技术本身发展极为迅速，我们目前还无法看到大量公开、成熟的实践案例。但从市场上许多小型团队和个人开发者的尝试来看，甚至在不少公司内部，

AI已经开始被引入到软件开发流程中。而这些实践的结果，整体上是令人惊喜的——无论是开发效率，还是生成内容的规模，都展现出了相当可观的提升。

与此同时，围绕整个软件开发方法论，也出现了许多新的分支与探索。其中一个备受关注的方向是“规范驱动开发”（Specification-Driven Development，简称SDD）。这种方法的独特之处在于，它能帮助开发者为AI的工作过程设定明确的框架、边界与范围，使其在可控的环境下更高效地运作。

目前，已经有不少成果初现端倪。例如，早期的Vibe Coding已经新增了“Prompt Mode”模式；亚马逊的Kiro也已崭露头角，再如Cloud Code以及一些第三方工具，如SpecK等，都在积极探索这一方向。这些新的工具与方法的出现，充分证明了围绕大语言模型进行软件开发，正在形成一股新的方法学潮流，带来了切实可见的创新实践。

我认为，我们需要保持耐心。再经过一段时间的积累，这些探索的成果必将更加清晰可见，并在行业中逐渐开花结果。只是由于目前相关技术与实践还处在早期阶段，许多令人瞩目的成效尚未被广泛认知，但这正是时间与持续投入所能解决的问题。

从经验驱动到智能驱动

> **InfoQ：面对AI浪潮，架构师如何更新自己的知识体系？哪些能力最值得重新构建？**

付晓岩：这个问题其实挺难回答，因为我们目前也无法明确判断大模型未来能够覆盖的范围有多大。过去在架构领域，我们常说，随着技术的不断细分，一个人已经很难胜任“全栈”工作。然而，当大模型出现之后，我突然发现，一个人似乎又能够重新实现全栈开发了。并不是说这个人掌握了所有技能，而是因为模型可以帮助他完成这些环节。

我目前正在尝试用自然语言编程的方式开发一个系统，这算是我个人的一个实验项目。该系统的代码量已超过10万行，拥有较为完整的前端管理界面和UI系统，也具备网络代理功能。其功能层采用容器化实现，后端包括关系型数据库和缓存系统等。很多模块的底层实现我并不完全清楚，但模型能够自动生成并运行良好，确实可以直接使用。

由此可见，个人开发者似乎又重新具备了全栈能力。但如果一个人重新具备全栈能

力，从知识体系的角度来说，就意味着每个部分仍需要具备一定的理解。因为无论模型多么智能，最终你仍然需要检验它的实现是否符合需求。对于一些关键性的技术栈，仍需适当“回炉”，不断补充知识，朝着更全面的方向发展。

这一趋势不仅体现在个人层面，也反映在企业层面。以OpenAI为例，Sam AItman过去曾强调分工协作——模型研发、硬件研发、软件设计各自独立。但如今，他的最新观点发生了变化，认为垂直一体化反而更有优势：不仅要建设上层生态、设计软件与模型应用，还要深入到算力网络乃至底层硬件的设计。

企业的这种垂直一体化趋势，其实也可以映射到个人的成长路径。对于个人而言，也许同样需要在掌握大模型能力的基础上，找到自己能够深入发挥作用的关键点，向垂直整合的方向拓展，而不是简单地依赖“开箱即用”。

毕竟，任何工具都不可能真正做到完全的“开箱即用”。想要获得良好的效果，仍需对其核心机制具备一定理解。我在与大语言模型长期交互的过程中，发现自己对后端开发的理解比以前更加深入，阅读相关书籍也比过去容易了许多。过去由于时间有限，对后端的知识掌握较为零散；而通过与模型的交流，这些知识得到了逐步补全。接下来，我计划更多地阅读编码相关的书籍，以更好地理解大模型在具体生成和执行过程中的潜在瓶颈。

> **InfoQ：您认为未来3-5年，要成为一位优秀的“AI时代架构师”需要具备哪些核心素养？**

付晓岩：这一核心要素其实非常明确，因为无论软件发展到何种阶段，其本质始终是结构化的。因此，结构化设计能力，尤其是具备全局视角的系统性设计能力——例如企业架构所代表的全局性结构化思维——始终是底层核心能力。

未来的软件开发在人工智能的加持下，将不再局限于某个局部领域的设计。企业内部的软件系统往往属于闭环体系，内部各模块之间在功能或服务层面存在紧密关联。要想在某个具体领域中实现良好的设计，必须对整体架构具备一定程度的理解。而大模型的出现，恰恰使得开发者能够更便捷地获取全局信息。

以往，在传统的业务体系中，开发者往往被局限在特定领域。例如，负责存款业务

的人通常对贷款系统的了解有限，因为两者日常交流较少，信息流动受限。但如果企业的开发体系基于大模型构建，那么跨领域的信息获取将变得极为容易。开发者可以快速了解其他业务领域的实现逻辑与设计思路，从而使自身的设计视野更加开阔。

在这样的背景下，全局性的结构化思维将变得愈发重要。随着大模型能力的持续增强，其在后端开发、测试、部署等环节中的自动化程度将不断提高，开发者反而需要更多关注前端设计及系统间的协同关系。例如，如何定义自身负责模块与其他模块之间的接口关系，如何协调版本与分支之间的衔接，这些问题都需要更高层次的系统视角来统筹。

因此，未来软件开发中最关键的能力之一，仍将是具备全局性、系统性的结构化设计思维。这种能力不仅决定了一个人能否理解复杂系统的整体逻辑，也将成为人与智能系统协作时不可替代的核心竞争力。

此外，未来开发人员的“软能力”也很重要，再有就是学习如何与机器进行沟通——换句话说，要练习与机器人“对话”的方式。因为未来的工作对象，不仅包括人与人之间的互动，还会涉及人和机器的互动，甚至机器与机器之间的协作。而在这些情境中，核心能力其实都是“沟通”。

在人与人的沟通中，我们需要通过恰当的表达范式，帮助双方更好地理解彼此。而在人机交互中，则要思考如何将人类总结的知识有效传递给机器。例如，可以通过构建本体论等知识表示方式，让机器能够理解人类的概念体系；同时，不同机器之间在处理各自领域的知识时，也需要建立起知识模型之间的“互译”机制。

因此，未来在沟通层面上，值得深入研究的方向包括：从人到机器的知识传递、从机器到机器的知识协作，以及如何在这一过程中建立高效的信息理解与共享机制。因为最终无论是人与人、人与机器，还是机器与机器的协作，其产出成果最终都是面向人、服务于人的。技术的最终价值依然体现在人身上，这一点不会改变。

> **InfoQ：如果让您给年轻的架构师或技术负责人一些建议，您会如何建议他们在AI时代保持竞争力与创造力？**

付晓岩：这其实是一件相当不容易的事情。一个人不仅要具备强大的学习能力，还

需要具备高度的自律。许多人往往把“学习”与“价值”直接挂钩，但实际上，从学习到体现出价值之间存在时间差。学习成果往往需要经过一段积累和沉淀，才能最终转化为价值。因此，在学习这件事上，必须具备长期主义的思维。不能抱着“今天学、明天用”，甚至“今天学、今天就见效”的心态，而应当把学习视为一场长线投资。

最近我看到一位学者提出了一个观点，印象很深刻。他说，一个人的真正竞争力，往往要到四十岁之后才会显现出来。而我们社会中常常讨论所谓“35岁危机”，其实是一个误区。人到四十岁以后，往往因为经历足够丰富，才能真正形成稳固的竞争力。

最近我在与企业交流时也提到，想让人工智能在企业中发挥作用，必须“把它当作一个人来看”。人类身上最重要的三方面特质是：智商、知识和经验。智商固然重要，但它几乎无法改变——出生后基本已经定型。知识可以通过学习获得，但光有智商和知识还远远不够。真正能够把知识转化为解决问题的能力，这种能力的核心就是经验。

经验往往来自长期的实践与积累。很多时候，只有经过反复的试错与总结，才能形成真正的判断力与决策力。而人通常在四十岁之后，这方面的积累才逐渐成熟。

我认为，如今一些企业在用人观上存在一定的短视倾向，过于偏好年轻员工。年轻人确实在体力和执行力上占优势，但在经验的深度和系统性上，往往还需要时间沉淀。真正的专家型人才，大多需要经过长期的积累才能成长起来。当然，在人工智能等新兴领域，专家的平均年龄可能会更低，但在大多数行业中，经验仍然是决定专业深度与判断力的关键因素。

人工智能的出现，反而有可能帮助我们延长职业寿命。虽然许多人希望能早些退休，但实际上，AI将让人类能够更长久地保持生产力。因此，在这样的时代背景下，更需要通过长期积累来构建稳定的核心能力。

沈剑：我想分享的，不管叫经验也好，还是建议也好，其实就一句话——要去了解它，并且使用它，而不只是观望它。

很多人都在谈“认知升级”“提升能力”“取得成果”，但这些目标背后，都离不开一个核心动作——行动。如果你希望认知上有提升，那就必须持续学习，比如去看书、看视频、去体验新的技术和工具；如果你希望在工作中拿到更好的结果，那也要靠行动

去推动。

对于技术人来说，最直接的行动就是——写代码、用工具。无论是AI提示词、AI辅助编码工具，还是各种自动化、智能化平台，都要亲自去尝试，把它们真正融入到日常工作中。只有在使用中，才能看到效果、积累经验、形成自己的方法论。

但目前市面上的书籍种类和各种课程纷繁复杂，如何选择适合自己的？我的建议是要结合自己的岗位去看。如果你很犹豫，不知道该从哪个工具开始，那就从与你的工作最相关的入手。比如你是做研发、做架构的，那就选择那些在你领域里当下最流行、应用最广的AI工具去尝试。

其实现在整个AI工具生态还没有完全稳定下来，没有谁能被定义为“最领先”或“最终胜出”。很多工具可能今天火，明天就被新的替代。但这并不重要——关键是你要始终保持在一线、保持“手感”。

技术的发展从来不是一统天下的格局，不会有一个语言、一个工具颠覆所有。每种技术都有它擅长的领域，每个工具都有它的价值场景。

所以我的建议是：不要犹豫，不要畏惧、不要观望，先用起来。选和你岗位密切相关的工具，边用边学，在实践中去感受它的潜力和局限，这样才能真正跟上技术的节奏。

> **InfoQ：如果未来AI能够自动生成“架构蓝图”，您认为人类架构师的独特价值还体现在哪里？**

杨凌云：我觉得可以从几个方面来看。

首先是决策层面的差异。AI的决策往往基于数据和逻辑推理，它可以在已有的信息中找到最优解，比如根据当前的技术趋势或业界最佳实践，为企业提供一个“看起来最合适”的方案。但企业的现实情况往往更为复杂，不同公司的文化、战略方向、业务阶段都不尽相同。

而人类架构师的价值就在于这种“情境感知”与“经验判断”。他们经历过真实项目的失败与成功，踩过坑，见过复杂系统的演化，能从这些经验中判断哪种方案才真正适合当前的组织环境。这种对企业整体战略、文化氛围以及人员能力的综合把握，是AI

难以具备的。换句话说，AI可以提供“选项”，但最终的战略性决策仍然需要由架构师来拍板。

其次，是关注焦点的不同。AI的目标往往聚焦于效率优化——它更擅长让系统更快、更稳、更节省。而架构师的关注点则更宏观、更人本。他们不仅关心技术本身，还关心组织协作方式、团队能力建设、系统长期演进方向，以及技术与业务战略的匹配程度。

在这一点上，人类架构师扮演的角色更像是“技术与组织之间的翻译者”和“平衡者”。他们要确保AI生成的架构方案不仅在技术上可行，也能被团队理解、被企业文化接纳，并在长期战略上可持续。

所以我认为，即便AI未来能够自动生成架构蓝图，人类架构师依然不可或缺。他们的独特价值不在于取代AI去画图或写方案，而在于通过洞察组织、判断取舍、制定方向，让AI成为自己决策的助力，而不是替代者。

付晓岩：这正是我目前的研究与实践方向。我现在所做的工作，正是探索人工智能能否在架构设计中发挥作用。

随着人工智能不断深入软件工程领域，人的工作重心正在逐渐前移。未来，大部分精力将集中在研发链条的前端环节，也就是价值的确定、方案的制定与最终结果的验证。而在这一转变中，架构师的角色也正在发生深刻变化。

最近我与一位企业架构领域的顶尖专家交流时，我们形成了一个共同的观点：未来架构师最重要的工作之一，将是企业层面的“语义管理”——也就是知识语义的组织与管理。架构师需要帮助大模型在企业特定的业务环境中，更准确地理解企业的业务逻辑、系统结构以及目标需求。

换言之，未来架构师的关键任务，是确保机器能够在企业内部语境下充分理解“你是谁”“你要什么”“你的系统如何运作”。这种“语义管理”将成为连接人类知识体系与智能系统的核心桥梁。

传统意义上的架构工作，例如制作PPT、绘制系统蓝图等，仍然是架构师的基本能力，但它们的重要性正在下降。因为随着大模型和语义网络的不断演化，机器已经可以根据需求自动生成相对完整的架构蓝图，并且其准确度会随着时间和数据积累不断提升。

因此，未来架构师的主要职责可能将转向三个方向：

语义建模与管理——帮助机器在特定企业语境中正确理解业务与系统；

设计与实现的验证——对机器生成的设计结果进行验证，确保其符合企业的真实需求；

偏差纠正与持续优化——在模型生成与落地实施过程中，持续发现和修正偏差，提升系统的智能一致性。

可以说，未来的架构师将从“绘制蓝图的人”，转变为“语义管理与系统校准的人”。他们的价值不再仅仅体现在设计图纸上，而是体现在如何让机器真正理解企业、理解业务、理解人的意图。

从OpenAI回国的90后姚班博导，打造了国内首个开源Agent训练框架：从OpenAI团队解散与重组，看智能体技术十年沉淀

编译 Tina 褚杏娟

> “中国的创业公司几乎没有机会走OpenAI这样的路线。”
>
> “所有的创新本质和创业一样，是个长跑，不能一直冲刺，一直冲刺就累死了，是大家慢跑，甚至走，边走边看evidence，一旦看到evidence了就激进冲刺。一直冲刺是可能看不见evidence的。”
>
> “AI时代的变化太快了，是以月为单位的。目标确定了之后，不要做过度规划，要激进地寻找evidence，然后激进地调整迭代。”

> “如果从开源来说，基本就是中国人的天下了。”

姚班、伯克利、OpenAI、清华……年仅30多岁的吴翼身上已经聚集了众多亮眼的标签。

从小到大，似乎无论在哪个阶段、哪个领域，吴翼都可以交出一份不错的答卷：他是ACM世界奖牌得主，也是曾带队冲击IOI的教练；他亲历了Facebook 2012的崛起、字节跳动2016 – 2018的飞速成长；他也自己经历了创业、目前正带着团队全力做着开源强化学习项目AReaL。

一定程度上，吴翼从小就做好了成为一个“IT人”的准备。

1992年出生的吴翼，高二暑假就在NOI1竞赛中摘得金牌，由此被清华交叉信息研究院提前“签下”，保送进入以顶尖人才培养著称的姚班——这个以创办者、世界著名计算机科学家姚期智命名的班级，当时每年在全国只招收30名学生。

2014年，他远赴加州大学伯克利分校，师从AI大师Stuart Russell，深入研究深度强化学习的泛化性，是Stuart Russell多年来正式招收的第一位华人博士生。回忆起自己的信息学竞赛生涯，他说这几乎贯穿了他整个成长轨迹——从小学的信息学比赛，到高一踏上ACM区域赛的舞台，再到博士期间依然在参赛，“每年都有比赛，这件事好像已经成了我生活的一部分。”吴翼的技术气质，很大一部分来自早年ACM这些信息学竞赛的磨炼——算法思维加上高压环境下的编程实战，以及在严格的时间限制下完成极具挑战性的任务的磨砺。

求学期间，他曾前往美国在**刚刚上市的**Facebook实习，参与Facebook搜索系统的研发，解决具体的产品问题；之后又加入**早期的**字节跳动，成为AI Lab的第4位成员。博士毕业后，他加入OpenAI，正值ChatGPT**爆火前**的关键时期。在OpenAI，吴翼深度的参与了一段极具探索性与前瞻性的旅程——多智能体（Multi-Agent）研究。他们的旗舰成果，“捉迷藏”项目的演示动画成为了OpenAI历史上Youtube播放量最高的视频，相关论文被广泛引用于多智能体系统的研究中。

在OpenAI的一年半里，吴翼亲历了这个机构独特的“基于证据快速迭代”文化——各个研究团队会以比较自由的状态开展前沿研究，并不会有太大的压力；而研究项目一

旦显露突破迹象，公司就会迅速形成共识，并不断投入更多资源，甚至快速灵活的调整不同团队的人员。ChatGPT本身就是一个小的研究项目，在项目爆火后，OpenAI迅速整合了资源，比如他所在的多智能体团队就进行了调整，相关研究也被搁置了，直到数年后，OpenAI才在Noam Brown带领下重启这条路径——技术的演进，往往是在反复探索与重组中前进。

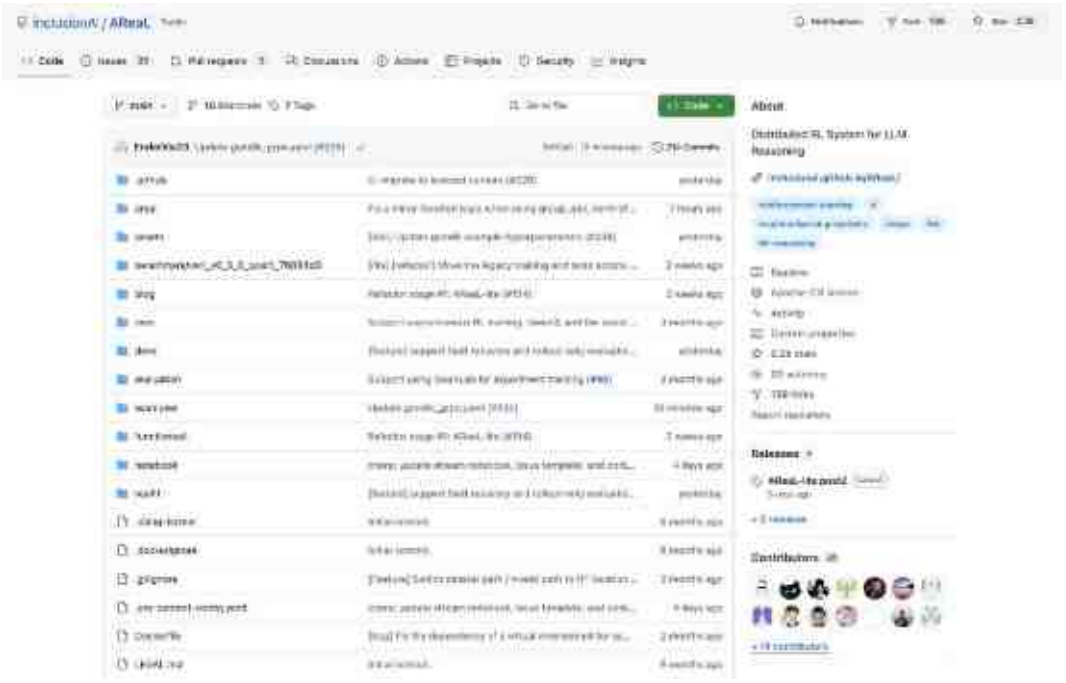
回望十年强化学习征程，吴翼看见一个令人震撼的“历史闭环”：

> 2016年，OpenAI有一个项目叫World of Bits，这个项目做的事情就是通过强化学习让一个Agent在网站上买机票。这个项目做了一年失败了。但是如果你站在这个角度去看，十年后，OpenAI不就是在做同样的事情么。从Agent/RL的视角看，其实事一直没变，甚至技术也没太大变，唯一的变化是，当年缺了大规模预训练这一环。

离开OpenAI后，吴翼做出了一个与大多数硅谷精英截然不同的决定：回国。2020年，年仅28岁的他，以全新身份重返清华园——这一次，不是作为学生，而是作为清华大学交叉信息研究院的助理教授和博士生导师，肩负起培养下一代顶尖AI人才的使命。

在学术之路之外，吴翼也和团队一起经历过创业（AI Agent创业公司边塞科技）以及建立开源项目——从零搭建团队，到选择和蚂蚁强化学习实验室一起推动AReaL这一开源强化学习系统。从2020年起，他们团队就开始做开源的规模化的强化学习工作，最终这些成果都被积累到了AReaL这个开源项目中。

AReaL从设计的第一天起就完全围绕Agent强化学习 打造。谈及定位，吴翼直言：“按照这个定位出发点来说AReaL目前是唯一的。当然，Agent RL还在早期阶段，我希望AReaL能在未来和大家一起来推动这个方向的发展”。



GitHub地址：https://github.com/inclusionAI/AReaL

嘉宾简介

- **吴翼博士**，清华大学交叉信息院助理教授，博士生导师，前OpenAI研究员，WAIC2025云帆奖得主。博士毕业于加州大学伯克利分校，研究方向为强化学习。Google Scholar引用超过14000次，曾获NIPS 2016 best paper award，ICRA best demo award finalist。他将在10月23日 -25日的QCon上海站发表主题为《智能体时代的强化学习：AReaL框架与Agent最佳实践》的演讲。

在OpenAI，我学会了以evidence驱动决策

> **InfoQ：从清华姚班到伯克利攻读博士，哪位/哪些导师给您的影响最大？为什么？在您看来，求学阶段的哪些习惯或经验，对您未来工作有很大帮助？**

吴翼：首先肯定是姚先生，如果没有姚先生、没有姚班肯定不会有现在的我了。我们对于科学的探求，对于学术品味的形成，乃至对于极其寻找新的机会去突破自己的舒适圈，这些都很大层面上是源自于姚先生的个人魅力以及姚班的培养机制。这个是底层的东西。

学术博士期间，我的几个对我非常有帮助的人，首先当然是我的导师Stuart Russell教授，他可以说是强化了我对好的学术品味的认知，同时他也在我读书期间对我有很大的支持，支持我去做更多的探索。Stuart是一个很好的visioner，他说的很多话我当时不懂，但是回过头看会觉得越想越有道理。

其他的话对我有帮助的，有三个人很重要。一个是现在在U Washington的Ras Bodik教授，Ras算是第一个手把手带着我做科研，带着我写paper整理思路的教授，对学生也非常的友好，算是第一个对我做严格科研思维训练的老师，只不过有点遗憾我我们只合作了一个项目他就从Berkeley去了UW。

还有一个Pieter Abbeel教授，Pieter教授是真正带着我走入强化学习的老师（我的第一篇工作Value Iteration Network，当时获得了NIPS2016的Best Paper也是和Pieter做的），所以我后续的科研选择，Pieter是起了决定性的因素的。同时Pieter也是很成功的创业者，以及他对待团队和极度支持学生的方式，也长期对我产生了很多的影响，我的可能很多工作方式都受他影响。

第三个是CMU的李磊教授，他算是我伯克利学术生涯的起点了（我最早在Berkeley的几个工作都是李磊在Berkeley博后期间带着我做的），如果不是因为李磊教授的赏识和培养我应该去不了Berkeley读书了。后来我之所以选择回国也受李磊很大的影响，是他2016年邀请我回到国内的字节跳动AI Lab实习，也因为这个契机让我埋下了回国的种子。

当然还有很多的朋友们无法一一感谢。至于说对于工作有什么帮助，我其实觉得其实很难讲有什么直接的帮助，工作的经验并不能通过在学校里习得的，毕竟纸上得来终觉浅。我觉得更多的是去见过更好的人，然后让自己成为更好的人。

> **InfoQ：您入职OpenAI时候ChatGPT还没有爆火，为什么会选择去OpenAI？在您看来，当年的OpenAI跟现在相比有发生哪些变化吗？**

吴翼：我那时候去OpenAI是个意外。我本来想去Google Brain（因为Brain的学术reputation一直高过OpenAI，那个时候更是这样），但是我很老实，我说我只待一年半我就要回国任教了。Google有headcount限制，我待长待短都要浪费他们一个headcount，就为了这个和HR部门扯皮了很久。OpenAI的话他们是非盈利机构，没有headcount限制，

我面试完第二周就打电话让我去上班了。当时我还很傲娇，我说不行，你等我一个月我要等Google，结果OpenAI真的等了一个月，而同时Google那边还没搞定，于是我就去了OpenAI。

> **InfoQ：您是ACM世界奖牌、IOI银牌得主，也是Topcoder红衣选手、IOI教练。在这些竞赛中，考验的是算法思维和编程速度吗？OpenAI曾在论文中展示o3模型在IOI测试集上拿下金牌，强调这是“不依赖test-time trick的RL扩展路径”，但上半年像CCPC、IMO这类真实竞赛，大模型几乎挂零。您怎么看待这两个不同的结果？**

吴翼：ICPC这些比赛（IOI其实也有这种趋势了），在俄罗斯有个称呼叫sports programming，我觉得非常准确。这个和电子竞技一样，本质上是个体育活动。它不是个考试，你很难用“考察什么能力”这样的问题去描述。当然算法思维和编程速度很重要，但竞技体育就涉及到更多的“技巧”以及心理因素了。

至于挂零，纯粹是模型不够好、没ready就拿出去的原因。比方说IMO这一次Google/OpenAI都用通用推理模型做到了金牌，字节用专用模型也拿到了不错的成绩。IOI最近AI模型也夺金了。其实大模型没有这些人的竞技因素干扰，攻克这些比赛是迟早的事。

> **InfoQ：近期，谷歌Gemini和OpenAI在IMO上的“夺金”表现是否可以归功于RL？如今AI正在大量“自动写代码”，您怎么看这种shift，您觉得IOI、ICPC这类竞赛还有未来吗？**

吴翼：是的。Gemini、OpenAI都很明确的说了，是因为RL训练。

至于未来，我还是再说，这不是考试，这是个竞技体育。我们其实可以参考围棋/Dota的发展的。作为人的训练和比赛平台，这些领域总是有价值的，但是可能因为AI的出现产生新的规定和训练方式的变革。

> **InfoQ：您当前的工作主要聚焦在哪个领域？OpenAI当时如何做赛道选择的？这对您后来创业的赛道选择有影响吗？**

吴翼：我目前的工作主要聚焦在强化学习领域，更具体的说，我们所有的工作都围

绕AReaL这个开源项目展开，AReaL是一个面向大模型智能体训练的强化学习框架，我们希望通过AReaL帮助大家训练出好的智能体模型。

OpenAI的所有赛道选择是两方面：

- 早期OpenAI是有charter的，他很明确的写了他这个公司要追求AGI，要通过scaling追求AGI，然后分了很多的方向开展研究，这个目标是top-down的。
- GPT系列的工作其实比较bottom up，都是几个人（甚至是一个人，比如GPT早期基本就是Alec Recford一个人搞的）的工作看到了evidence，然后大家scale up。ChatGPT也是几个人做出的原型，一下子火了（这个并不在OpenAI的计划之内），看到evidence之后，对于目标明确的小团队或者创业公司来说，做选择就太容易了。

至于我的研究工作，其实我一直也是scaling的爱好者（我PhD申请的文书写的就是我要做large scale machine learning system，当时是2013年），我在OpenAI的经历也更加坚定了我的一些观念，这些是受到影响的部分。至于创业的话，边塞科技的创业起因也挺随机的，也无非是恰好团队会做一些事情，而这些事遇到了时代的机会而已。

> **InfoQ：您当时在OpenAI的工作，尤其是“Multi-Agent Hide and Seek”，是否属于RL核心团队中的一部分或是否是RL的研究主线？这里涉及的研究是否在后面已经投入到了实际的某个框架产品或模型设计中？还是更多被当作“探索性研究”？当时的智能体与现在的有什么区别吗？**

吴翼：当时OpenAI有三个和RL相关的团队，一个是机器人团队，做的项目是机器手拧魔方，一个是RL团队，做的项目是打Dota。还有一个就是我们multi-agent团队，做的项目是“智能涌现”（emergent intelligence），换句话说使我们希望通过多智能体交互和进化的方法，观察到智能体涌现出智能的行为，最后的成果是捉迷藏项目。这当然是multi-agent团队的研究主线的成果。

捉迷藏项目结束之后，我们又进行了一些自由探索，比如在Minecraft上的算法尝试，我离开后OpenAI也发表了在Minecraft上让Agent能够造出钻石的项目，这个就是当时的multi-agent团队的后续工作。

据我所知multi-agent团队应该是在ChatGPT出来之后解散了（很多团队都解散重组了，比如Robotics，这个在OpenAI内部很常见）。后来Noam Brown加入OpenAI之后重新开启了multi-agent的方向。OpenAI的具体工作一直都很有探索性，同时团队目标和人员流动都非常灵活，并不是强KPI和分工导向的。

至于当时的智能体和现在的智能体有什么区别，其实本质上唯一的区别就是有没有大模型、有没有做预训练。其实如果你站在智能体的角度上看，强化学习这10年其实本质上绕了一个大圈，回到了OpenAI一开始想做的事情。比如OpenAI在2016年有一个项目叫World of Bits，这个项目做的事情就是通过强化学习让一个Agent在网站上买机票。这个项目做了一年失败了。但是如果你站在这个角度去看，十年后，OpenAI不就是在做同样的事情么。

从Agent/RL的视角看，其实事一直没变，甚至技术也没太大变，只是当年缺了预训练。这个话题如果展开聊可以聊很多，这里不做展开了。有兴趣的同学可以了解了解RL早年的一些工作，我找机会也跟大家多分享吧。

> **InfoQ：Calvin French-Owen在最近发表的文章中提到OpenAI是一个“把研究员当mini-CEO”的地方，强调bottoms-up、快速迭代、没有master plan的文化。您也曾在OpenAI做研究，后来开始创业——从今天一个技术创业者的视角看，您觉得OpenAI这样的组织范式是可以被小团队借鉴的吗？还是说它的“自下而上”更多是建立在极端资源富裕基础上的例外？**

吴翼：我觉得可以。我甚至更偏激的认为觉得AI时代的团队必须是这样的。因为AI时代的变化太快了，是以月为单位的。目标确定了之后，不要做过度规划，要激进地寻找evidence，然后激进地调整迭代。我自己的认知体系也不断在更新的。当然会有一些belief是不变的，但belief一定是很少且已经经过时间考验的，比如我可能坚定的相信Agent，相信RL，因为我从AlphaGo开始已经相信并且工作了10年了。

至于资源富裕与否，我觉得富有富的玩法，穷有穷的玩法，组织逻辑不会变的，AI时代其实极大的放大了穷资源团队的能力了，穷团队无非是不能训练而已，其他都可以在AI加持下极高速推进。

> **InfoQ：对于未来践行scaling law，OpenAI当时“产品驱动科研”的方式还有**

> 借鉴意义吗？

吴翼：当然，我觉得现在也是成立的，我们团队也一直是这么践行的，就是咱们先不聊技术，先聊聊咱们想干出来个啥，这个东西厉害么？厉害，好的，我们开干，然后再去想咋干。

Scaling law其实也是这个意识形态下的不断收集evidence的总结而已。他不是凭空冒出来的，是从seq2seq、AlphaGO、Dota、捉迷藏、Rubik’s Cube、GPT等等一系列的evidence不断增强的。

> InfoQ：Calvin在文中提到Codex团队**用了7周就从0到1推出产品**，并称这是他十年来最密集的冲刺。从创业者角度看，您如何看待这种密集“冲刺”的做法？这种节奏可复制吗？如果做不到，那么原因会是什么？

吴翼：这是典型的创业精神。也是优秀的创业公司的常态。

当然所有的创新和创业本质上一样，都是个长跑，不能一直冲刺，一直冲刺就累死了，是大家慢跑，甚至走，边走边看evidence，一旦看到evidence了就激进冲刺。如果盲目一直保持冲刺状态反而是可能看不见evidence的。

做不到有两种原因：你的组织结构就不是奔着创新和创业去的，不是创业组织，所以就做不到这么快；或者你的创业/创新团队出现了问题。

> InfoQ：硅谷和国内的创业公司有什么不同？离开OpenAI回来的决定会比较难下吗？您也在之前采访中提到当时回国时候“看到中国仍然有很多机会”，具体是什么？您觉得自己抓住了嘛？

吴翼：OpenAI不能算典型的创业公司，所以我很难评论硅谷的创业公司。总体上我只能说硅谷的资源更多，对于技术创业者更友好。国内创业基本是个身心灵的修炼场。

我并不是去了OpenAI才决定回国的，我是去之前（2018年10月）就已经和姚先生说好2020年8月回国任教了。当时甚至也没人知道OpenAI要做profit转化（导致我没拿股票），也没人知道会有疫情。

至于机会，我很难具体说。因为我也只是有个感觉，觉得这个时代应该有很多机会。

我很有幸的见过2012年的Facebook（我当时在实习），2016到2018的字节跳动，有幸见过中国美国互联网，也有幸见过中国互联网时代的尾巴。那个10年中国创造了太多奇迹了，未来十年因为AI还会有的，只是机会是什么我不知道。

抓没抓住，我肯定至少是见着了，2012年Facebook，2016-2018字节，2019-2020 OpenAI，2023参与也见证了边塞团队创业。其实都能见着也不容易。抓的话，可能我确实手滑。

> **InfoQ：在OpenAI的经历，对您回国以后的创业和研究有产生什么影响？比如研发方式、团队管理等？**

吴翼：创业的话没啥影响，因为中国的创业公司几乎没有机会走OpenAI这样的路线。

研究的话我团队一直是product-driven research，并且非常看重基础设施和系统，这点是受到影响的。比如AReaL就是这个逻辑的产物。

“创业不是个技术命题”

> **InfoQ：您在创立团队时选择了强化学习作为切入点，试图探索AI与人的对齐问题。如果把那个决定放在今天，您还会坚持当初的方向吗？如今，不少大佬都说时代不同了，创业已经不像以前一样，需要三五年死磕一个方向，在这个“快速试错”“不断调方向”的创业环境下，您如何看待当时的判断？**

吴翼：创业是要看客观机会看势的，不是以自己主观为导向的。如果放在今天看，我觉得现在不是一个好的技术创业时间点。美国是的，中国不是。中国做具身可以，做产品可以，技术创业和算力芯片相关可以，纯AI技术的我建议慎重。

至于我自己，其实我当时也没有经验，啥也不懂，沾了时代的光而已。

> **InfoQ：在2023年的创业期间，整个行业格局变化非常快，边塞团队当时的创业状态是怎么样的？什么问题是最困扰你的？当时一大批清华系创业者“同根竞争”，你们私下会互相交流心得之类的吗？您当时有考虑如何从中“跑出来”吗？**

吴翼：遇到的困扰太多了。技术和商业上其实都是小事，更多是人性上。对我个人而言，我整个人可能算重生了一次。

我觉得大家把竞争看的太奇怪了，商业竞争而已，大家私下里都是关系很好的朋友，也经常交流。大家不要总想着分蛋糕，AI这么大的时代下，找到自己的位置，好好做事，大家一起把蛋糕做大才是。

> **InfoQ：有报道指出，您当初对“商业落地空间存在一些疑虑”，如今回头去看，您对此有没有新的反思或心得？**

吴翼：创业者是需要理解商业的。大家都说要谋定而后动。话是没错的，但是你没做过你咋知道怎么谋呢？所以有机会就开始，就多尝试，试错是学习最快的一种方式。

> **InfoQ：您不止一次提到“创业难”，回头看您团队曾经踩过哪些坑？技术因素和非技术因素的都有吗？有没有是当时想不到、但后来代价特别大的？**

吴翼：其实没啥技术原因，我到现在依然很自豪的觉得AReaL团队是世界顶尖的强化学习团队，放到硅谷去也是超一流的。

但是创业不是个技术命题，甚至都不是坑不坑的问题，很多事情不以人的个人主观意愿为转移的，可能时间窗口就那么点，且你当时的第一视角也不知道，有人做了那个决策，那就抓住了时间点；你没有做决策，主观上也不能说是错，但就客观上错过了那个时间点。谋事在人，成事在天而已。人能做的无非是提高抓住机会的概率，但怎么说还是个概率事件。

所以我总体上会建议年轻的同学们尽量多尝试，尝试就有>0的概率。而且其实没什么代价特别大的，都是收获。

> **InfoQ：这段创业经历有没有改变您对大模型落地、产品形态或市场节奏的看法？**

吴翼：相对了解的更多了一点。

> **InfoQ：未来您的团队还想创业吗？还是说更倾向继续做科研工作？您如今以什么方式投入到AI前沿研究中？**

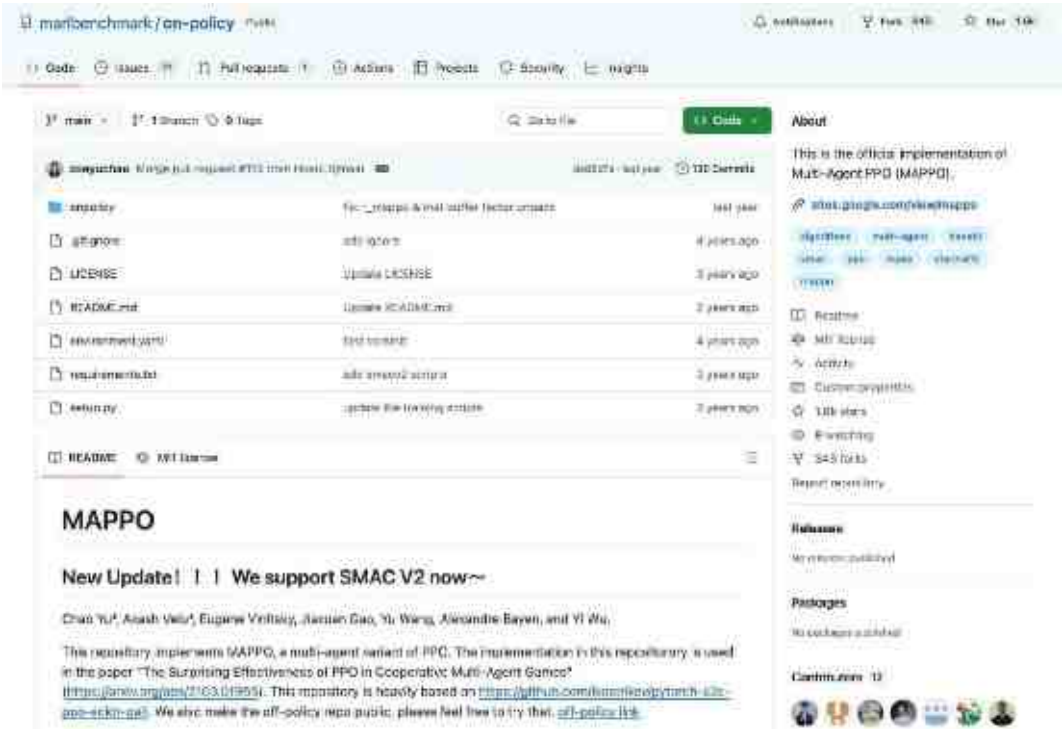
吴翼：今日不谈未来事。毕竟AI时代都是按月计的。

我们团队现在唯一的目标就是好好做好AReaL这个开源产品。

一切围绕Agent，没有竞品可比

> InfoQ：AReaL的出现，是否跟您之前的经历有关系？在技术路线上，延续了哪些过去的想法，又做出了哪些转向？

吴翼：我们团队从2020年就开始做开源的规模化的强化学习工作，从最早的MAPPO，到后来的SRL，再到ReaLHF，再到现在的AReaL，基本一脉相承，都是RL scaling，首先满足自用，然后开源。不过我们因为长期自用所以从产品角度看我们的开源工作其实一直做的不大好（MAPPO其实写的也挺烂），今年开始比较认真的把AReaL当成一个重要的开源产品推进。

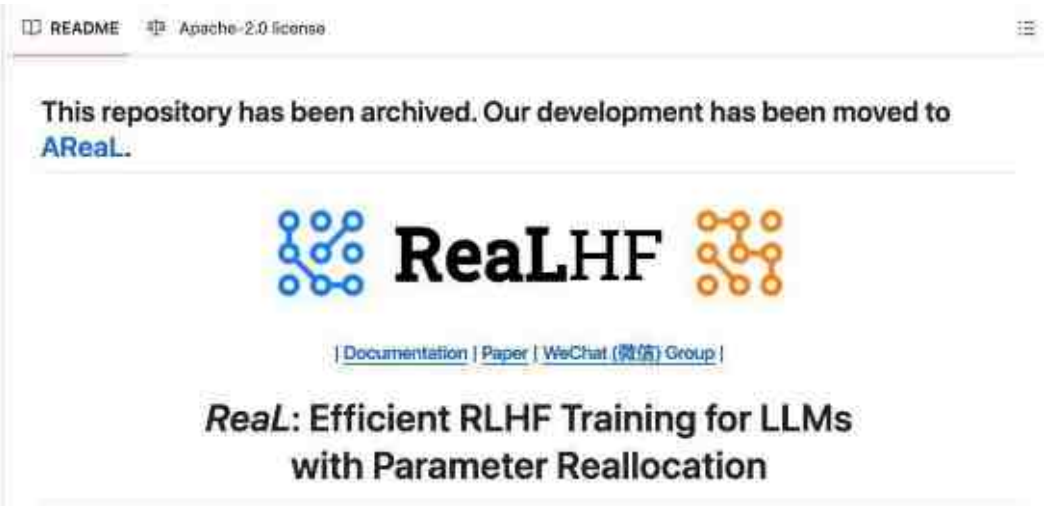


截图来自：https://github.com/marlbenchmark/on-policy

编辑注：MAPPO是一个轻量级、高度优化且运行速度极快的多智能体PPO库，专为学术研究场景设计。它在多种协作型多智能体基准测试中表现优异，达到或接近最先进水平（SOTA），包括Particle-World (MPE)、Hanabi、StarCraft Multi-Agent Challenge (SMAC)以及Google Football Research (GFR)。

> InfoQ：我们注意到您团队从2021年开始就搭建了专属的分布式强化学习框架。并发布了分布式RLHF训练框架ReaLHF。那么从ReaLHF到AReaI，是一次平滑的演进，还是一次技术重启？

吴翼：MAPPO->SRL->ReaLHF->AReaL都是一脉相承的。



截图来源：https://github.com/openpsi-project/ReaLHF

> InfoQ：AReaL的定位是什么？有相似的“竞品”吗？一个好的RL框架主要有哪些考量因素？

吴翼：AReaL从设计的第一天起，出发点就是让人更快训练出更好的Agent模型。一切围绕Agent。其实按照这个定位出发点AReaL是唯一的，算是目前还没有竞品吧。毕竟我们团队从第一天开始，就在做Agent和RL，坚持算法工程化并一直持续迭代到现在，挺独特的。当然这里有很多地方可能要跟大家科普的，为什么我们的设计可以做Agent，为什么我们的设计大家好用，后续我们慢慢开直播跟大家科普（广告一下，大家也可以在视频号关注**AReaL-吴翼以及蚂蚁开源**，都能看到AReaL团队的一些科普和直播内容）。

好的框架无非：1.好且快：能够产出SOTA模型且快；2.好用：能够让用户简单改一两个代码文件就完成定制的agent workflow和RL训练。AReaL一开始围绕1，最近AReaL-lite发布，我们终于能围绕全部的1和2把整个系统做了重构。

> InfoQ：特别是您之前有提到，OpenAI有自己的强化学习训练框架，大家都可以

> **在上面进行验证，如果要把AReal和OpenAI的这个框架放在一起进行对比，那么它们之间的关键差异有哪些？海内外在RL训练框架上还有明显差距吗？**

吴翼：首先，我不知道大厂内部的框架到底怎么样。我也离开OpenAI 5年了。确实没法评论。

如果从开源来说，基本就是中国人的天下了。但是头部公司肯定有很好的infra，我相信这一点（尤其OpenAI和Anthropic），毕竟大家理念都很像，大公司有更多资源和更好的团队，这些公司的组织也很好，没有道理做的比我们几个人做的差的。比如OpenAI就可以一个礼拜内让几个人创造出ChatGPT，这就是好的infra的作用。

> **InfoQ：您之前有提到RL目前三大分支：泛化（DeepSeek）、代码（Anthropic）、Agent（OpenAI），那么Areal在这三个方向中更倾向或看重哪个？**

吴翼：首先我觉得你很难简单的把三个分支和三个公司对应起来。模型能力是很复合的。只是各个模型有一些特色而已。Anthropic和OpenAI泛化也做的很好。最后无非是细节差异而已。

AReaL也是一样，AReaL是围绕Agent打造的项目。好的Agent也会需要泛化能力和代码能力的。所以AReaL也可以训练很好的代码模型和泛化的模型的。

> **InfoQ：您曾提到Areal主要服务开发者，并希望展示如何使用框架训练Agent模型，能不能聊聊这个的设计思路？并且为什么做Agent应用需要一个RL训练框架？**

吴翼：做Agent应用不一定非要用RL训练。应用应该以用户为第一快速迭代。

但是如果有一天资源成本下降，那么用RL可以训练出更好的Agent模型，可以帮助应用团队打造出更好的Agent产品。我们希望能够让大家都享受到RL技术发展的红利。

> **InfoQ：目前AReaL已发布多个版本，在数学和编码领域已经达到开源的顶尖水平，接下来的开源路线和规划是什么。如何看待AReal对整个Agent生态和应用场景的意义？**

吴翼：欢迎关注AReaL-lite项目，这是我们重新为了算法研究和用户迭代做的新版本。欢迎使用。https://github.com/inclusionAI/AReaL

也欢迎关注ASearcher项目，这是一个用AReaL-lite训练的search agent，希望能够给大家有所启发和帮助。https://github.com/inclusionAI/ASearcherv

判断技术潜力和未来展望

> **InfoQ：判断技术潜力从单个Agent到多智能体系统（Multi-Agent），在实际落地过程中面临哪些新的挑战和机遇？**

吴翼：目前来看，multi-agent是一个必要的方向，因为agent workflow是很复杂的，很可能需要多个智能体配合。同时未来智能体普及后，智能体之间的交互和算法逻辑也会需要依赖多智能体算法。整个智能体系统会变得更加复杂，而更复杂的系统肯定就有更多的算法和infra的机会。这些需要大家一起探索的，我们也是希望AReaL能够帮助大家更容易更快速的探索这些新机会，和大家一起进步。

> **InfoQ：您对Agent技术未来的发展方向怎么看？是否存在新的范式、框架或关键突破点值得关注？**

吴翼：Agent一定会成为大模型交互形式的主流，从被动的交互变成主动为用户节省时间。Agent自主探索和工作的时间和空间都会越来越大。因此算法提升的空间也就会越来越大，从具体目标驱动，到更开放的环境中。

新的范式一定会存在的，欢迎大家持续关注AReaL团队的发布和release，希望我们能把关键点早点做出来，也希望和大家一起探索更广阔的智能体的未来。

半年研发、1周上线，1秒200行代码爆发？美团研发负责人：靠小团队奇袭，模型和工程能力突破是核心

作者 褚杏娟

如今，AI编程工具正在重塑软件开发，其核心目标直指“开发民主化”。它们不再仅仅是补全代码片段的助手，而是能理解自然语言需求、生成可运行代码框架、甚至参与系统设计的“协作者”。这一背景下，越来越多的企业开始对外发布相关产品，美团便是其中之一。

6月10日，美团正式推出了自己的首款AI Coding Agent（AI编程智能体）产品NoCode。那么，在激烈的AI编程市场，美团如何定位其产品的核心竞争力？其底层模型具体如何平衡代码生成的质量与速度？未来的AI编程产品是否还是围绕IDE的延长线做？

这次访谈中，美团基础研发平台工程效率负责人、美团技术委员会委员程大同，对各种大家关心的问题进行了解答。主要观点如下：

- 尺寸越大，模型的吞吐速度会受影响，特定的任务需要不同的模型完成，团队对小尺寸模型做了针对性的优化，来平衡性能和效果。
- 后期难以维护问题确实存在，但是本质是代码，随着Coding Agent的演进，这类问题会自然而然解决。
- AI Coding效率提升的衡量是比较复杂的，当前阶段比较合理的观察就是AI增量代码占比以及AI Coding的采纳率。软件工程本质是很复杂的，包括流程、人、组织，以及用户和交付。
- AI Coding的核心竞争力，首先在于模型，其次是上下文工程，然后是对云infra和流程的集成与自动化，另外产品交互和品味也很重要。
- NoCode/CatPaw还是聚焦在技术突破，商业化思考并不太多。

“一周上线，团队非常有激情”

> **InfoQ：NoCode项目从2024年10月启动，今年5月发布。半年多时间，你们主要都在做什么事情？整个研发规划是怎样的？**

程大同：这半年主要是对内部的支持，在内部受欢迎程度很高，能快速交付产品原型，以及一些产品或运营系统，包括工具类的开发。

我们未来规划，会和内部更强的Coding Agent CatPaw联动起来，同时在后台infra支持上更加多元化，以及在这个过程中做好大规模下的稳定性和性能。

> **InfoQ：7个月的时间里AI技术发展节奏是非常快的，你们期间有为此做过什么调整吗？如何保持整个产品的技术先进性？**

程大同：实际上NoCode投入并不多，几个人开发出来的，一部分代码甚至是NoCode自己给自己生成的。大的方面没有什么调整，5月份对外的版本花了差不多1周时间就上线了，团队连续工作了1-2周，非常有激情，对AI Coding也有信仰。

先进性主要是在一些云infra的支持、Agent任务评测、产品交互与品味，我们认为很多技术小白是我们非常重要的用户。

> **InfoQ：总体来说，NoCode如何实现“1秒生成200行代码”的高效输出？美团自研7B Apply专用模型如何实现2000 tokens/s的推理速度？怎么做速度和质量的平衡？之前采访中提到，如果用大尺寸模型，生成速度可能只能达到几十tokens/s，为什么模型越大生成速度反而更低了？**

程大同：这些算是模型工程层面的优化，有一定的算法要求和技术壁垒，尺寸越大肯定模型的吞吐速度会受影响，小尺寸模型我们是做了针对性的优化，能平衡性能和效果。

> **InfoQ：美团HR团队用了NoCode发红包，但还是需要研发辅助。如今非研人员用是不是还不能自己“无痛”使用零代码AI应用开发工具？其中的难点是什么？有什么使用技巧可以分享？**

程大同：如果能用到产品的中等能力，本质应该没有什么难点，唯一的难点就是你能不能和AI一起进步学习，持续学习很重要。当然，你想要使用的更专业更有挑战，那可能需要花一点时间去理解计算机相关的知识，但我认为借助其他AI工具，也能快速成长起来。使用技巧非常多，NoCode论坛和公众号都有相关的推送，挺不错的，包括用户登录、音视频播放、大模型接入、微信收费等等。

> **InfoQ：在你们的宣传中，有提到对标Lovable。Lovable确实增长最快的AI工具之一，有观点认为它本质上是一个从文本生成App的工具，那么在你们看来，除了“从文字生成一整个交互式App”之外，这类工具跟现在的App构建器有什么本质区别？**

程大同：纠正一下，并不仅仅是文本，也支持图片相关的能力，包括图生码，NoCode内部版本是支持的。这个跟APP构建器有本质的区别：用户不一样，NoCode面对的是不懂技术的同学，当然懂技术的同学也能更好的使用；技术原理不一样，一个是偏规则或者固定系统架构，NoCode是AI类Agent，更智能，在产品能力上有更多想象空间，同时Agent还能调用工具或者infra，在App构建时更自动化、更迅速敏捷。

> **InfoQ：我们观察到，有些开发者觉得AI工具生成的项目虽然“漂亮”，但后期难以维护，不如直接用GitHub上工程化的模板，而有的用户觉得可以迅速上手、做出能用的App。你们怎么看这类产品的用户画像？你们更倾向对标新手、初学**

> 者、非技术用户，帮助他们“零起步”做出能跑的产品？还是更希望服务工程经验丰富、有长期迭代需求的专业开发者？从产品设计角度，你们如何平衡这两类用户的差异化需求？

程大同：后期难以维护确实有这类问题存在，但是本质是代码，我相信随着Coding Agent的演进，自然而然解决这类问题，比如通过rules的要求等等。

关于低代码，不是一类产品，当然LLM也可以参考一些模版化来实现局部的AI Coding。可以参考前面回答过的问题，我们的目标用户当前是新手或者非技术用户，但实际上我们也有不少的专业开发者，或者互联网产品及运营同学。从产品设计的角度，我认为创造力或者想象力非常丰富、且能够持续学习的同学，应该是这类产品的主要用户。

> InfoQ：还有反馈说这类工具静态页面它能一把梭，交互复杂了就容易“力不从心”。你们怎么看这个产品的使用边界？更希望谁来用、用在哪儿？

程大同：需要一点耐心，我们用的好的同学、会用的同学对话轮数少则几十轮，多则几百轮，几个小时甚至几天就能交付一个可用的系统。你看到“一把梭”，但实际上Agent也不断在持续进步，一方面给用户呈现的方式感觉是“一把梭”，实际上也是局部的修改，另一方面精准的diff的方式，以及局部代码生成也是我们Agent持续迭代的方向。

> InfoQ：NoCode属于氛围编程理念吗？你们如何看待“氛围编程”？有观点认为，prompt驱动的氛围编程不利于组件化和复用，大型工程中难以落地，您怎么看？如何解决“氛围编程”的模糊性、开发流程不可控等问题？

程大同：当前虽然叫做的vibe coding，本质上还需要持续去解决工程上的复杂挑战，NoCode本质也解决了不少，比如存储、计算、数据库，包括云上的环境资源，以及通过rules或者prompt要求哪些组建或者程序版本。后续随着Code Agent的演进，以及模型能力的提升，不限于数据算法和RL/SFT对软件工程的理解，本质上是可以解决这些模糊、不可控的问题。

> InfoQ：Code Agent核心竞争力是否是底层的大模型能力？为什么？另外，为了让智能体输出更准确、稳定，现在大家开始主张“上下文工程”，上下文工程为

> 什么火了？能否谈下上下文工程的具体技术实现？具体效果如何？

程大同：首先模型是很重要，其次上下文工程也很重要。我们确实做了很多上下文工程的事情，比如Index的效果和速度，我们都做了很多针对性优化。然后是对infra和流程的集成与自动化，这部分性能和稳定性还是很复杂。另外，在产品交互和品味上也很重要，因为模型的输出和用户交互，本质很多流式或者异步的处理，链路极其复杂，如何更好的呈现到产品上，是我们要思考的。最后，我认为还比较重要的是评测，评测是保证这些复杂链路的稳定性和性能。

> InfoQ：很多人试图搭建一个网站，参考已有示例复刻很顺利，但最终会碰到“最后一公里”的问题，比如集成分析工具、添加某些库支持等，这些还没有支持。对于“最后一公里”的问题，你们是如何规划的？怎么看待当前产品的极限以及未来一两年的能力边界？

程大同：现在支持数据库存储，以及数据分析场景，一些比较常见的能力都有适配，可以访问我们的论坛或者最佳实践。未来还是要解决多技术栈和后台能力，让Coding Agent能解决更多复杂的产品开发问题。

“Cursor也会逐步覆盖NoCode方向”

> InfoQ：切换到Dev Mode的CatPaw定位是专业开发者的Agentic IDE，其技术架构与NoCode相比复杂性体现在哪里？

程大同：更Agentic一些，ReAct更强，更多的专有模型、更复杂的链路，包括我们对IDE的开发投入。如果把整个Catpaw Agent的核心架构利用起来，自定义上下文工程、工具流程、Prompts和Rules，再放到一个特定的业务场景下，我认为它不局限于Code场景，当前是因为我们放入到了CatPaw或者NoCode这个产品中，所以叫Code Agent。

> InfoQ：CatPaw从2022开始做的，对标Copilot。24年开始往NoCode投入时，你们怎么给这两个产品做区分？现在两者的资源投入比分别是多少？你怎么看这些工具之间的关系？它们是各自属于不同的产品层，每个层级都有自己的赢家？还是说这些工具未来会合并，最终发生整合？

程大同：NoCode是CatPaw的研发团队孵化出来的，从产品层面上我认为不会那么快

合并，因为解决的问题和用户场景还是有区别，不过技术架构和Agent发展方向会逐步有一些协同，本质是你在不同的用户场景换不同的环境或者工具就能解决不同的问题，但AI的思考模式也可以基于场景来定制，比如prompt、rules等等。

> **InfoQ：美团如今50%新代码由AI生成，你们怎么度量AI对开发效能的真实提升？是否存在“生成量高但代码冗余和维护困难”的陷阱？**

程大同：AI Coding对效率肯定是有提升的，这个不用多说了，整个事情怎么衡量是比较复杂的，当前可能最好的方式就是看这个指标，只能说明大家用的更多了，但如何用好它们，以及解决一些复杂的问题还需要人和组织去决策，一些技能和知识还需要时间去发展。软件工程本质是很复杂的，包括流程、人、组织，以及用户和交付，因此这么多年也不太好衡量。

关于代码质量和冗余，人写的代码也会有类似的问题，但是我认为管理好AI应该比要求人更容易，前提是你要非常懂AI。开发者更多是一个“调度员”角色，指导AI完成具体的编码任务，工程师聚焦更复杂的系统架构设计和更有创造性的产品设计任务。未来将会是一个人机协作的编程模式，AI工具应作为开发者的得力助手，而非完全替代品。

> **InfoQ：当前，大部分Coding Agent总体还是围绕IDE的延长线在做，未来会不会有变化？如果大家都挤在这条路上，后来者要如何追赶Cursor等？**

程大同：如我说的，NoCode和CatPaw在系统架构上会有一些协作复用，Coding Agent最终会变成一个remote agent，它自己有一套sandbox或者开发环境帮用户实现对应需求。当前NoCode就是这类，因此CatPaw也能，甚至可以更专业。

Cursor本质会逐步覆盖NoCode这个方向，甚至其Agent架构再叠加特定的上下文工具、Promots、Rules以及一些自定义Agent，来覆盖更多的业务场景，不限于Code场景。不过大家会有一定的差异，可以等一等再看。

> **InfoQ：Cursor最近以各种第三方模型“每次请求成本差距极大”为由调整价格，这反映了当前AI编程工具商业模式上的哪些问题？怎么平衡用户的成本和体验？NoCode/CatPaw如果未来商业化的话会选择什么样的模式？**

程大同：正常，现在Cursor虽然ARR增长不错，但是还是亏钱的。AI Coding还在快速

发展，随着算力芯片的成本下降和模型能力的提升，我认为是可以达到平衡的，这里面还有现有的流程、组织和人，都要考虑进来。

NoCode/CatPaw还是比较聚焦技术上的突破，商业化思考并不太多，当前主要聚焦在AI技术上的探索以及让更多用户使用上更好的AI Coding工具产品。

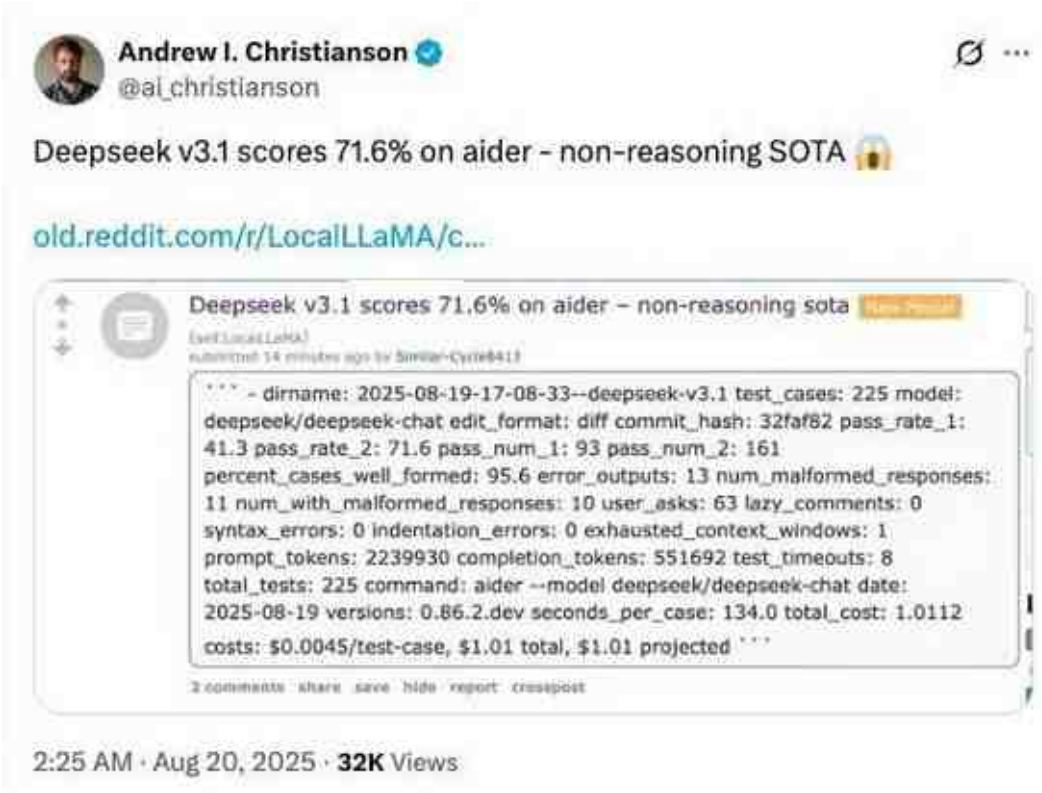
Claude封锁中国，腾讯带着国产AI编程工具CodeBuddy来了

采访嘉宾 汪晟杰，腾讯云开发者AI产品负责人
编辑 Tina

AI编程工具的竞争已经进入深水区：不仅各家产品在补全速度、上下文感知、智能体协作上不断拉锯，在背后的模型层面，博弈同样激烈，甚至出现了全球范围的“准入门槛”和“封锁线”。这意味着工具之争早已不是单纯的产品对比，而是与模型生态、合规和市场战略深度绑定。

然而，在这样的背景下，国产代码模型也在加速发力。今年8月，DeepSeek V3.1在国际开发者社区引发热议：它在aider编程基准测试中取得71.6%的成绩，成为新的非推理类SOTA，不仅比Claude Opus 4高出1个百分点，还便宜68倍。更重要的是，这一版本在编程表现、Agent能力、推理效率和长文本处理等方面全面升级，为国产工具落地提

供了坚实基础。



顺势而来的是国产编程工具的演进。8月21日，CodeBuddy IDE率先完成DeepSeek V3.1的接入并开启公测，让开发者第一时间体验最新国产模型在真实场景中的能力。仅仅不到三周，团队便根据反馈完成优化。

9月9日，CodeBuddy带来了两大更新：

首先是增加新的产品形态CLI，也有独立的名字“CodeBuddy Code”。这是国内乃至全球少数同时支持IDE插件、独立IDE、CLI三种形态的AI编程工具，开发者可以自由切换，按需选择最适合的工作流。

全新上线的CodeBuddy Code是终端原生的AI CLI，通过 `npm install -g @tencent-ai/codebuddy-code` 即可安装。它让习惯命令行的开发者在熟悉的环境中获得AI辅助，无需切换工具，并能与现有脚本和工具链配合使用。CodeBuddy Code内置文件编辑、命令运行和提交创建等功能，也支持通过MCP扩展或自定义开发工具。

（CodeBuddy Code的操作界面）

整体而言，CodeBuddy Code具备自然语言开发、智能代码库分析与集成、内置完整工具链、多场景任务自动化、灵活扩展AI团队能力（即将上线）等五大核心产品能力。

其次是IDE新能力与生态融合，并宣布CodeBuddy IDE开启公测，面向所有用户开放使用，无需邀请码。国内版支持DeepSeek，所有功能均可无限制使用；国际版支持GPT与Gemini等主流模型，可同时在IDE和CLI消耗Pro模型额度（测试期间赠送部分Pro模型体验额度）。

本次公测的IDE版本在两方面提升显著：其一，针对AI编程领域普遍存在的痛点（如后端质量不稳定、上下文不足）进行整体优化，结合新的Agent设计，进一步提高生成质量与稳定性；其二，与腾讯生态的融合更深入，尤其是CloudBase、EdgeOne Pages等能力，开发者可以直接用CodeBuddy从零构建小应用，并一键部署到腾讯云开发环境中，打通了完整的“代码、部署、上线”闭环。

这也意味着，国产模型的突破正在与国产工具、云平台、应用生态联动起来，形成一条贯通的“模型—工具—生态”链路。

在这样的背景下，InfoQ采访了CodeBuddy团队，深入探讨他们在产品定位、技术决策以及未来发展上的思考。

AI编程工具的中国路径

> **InfoQ：从最初发布到现在，CodeBuddy中间其实发了不少功能，那么回顾起来你们在定位或功能上有哪些关键转折？**

汪晟杰：在腾讯内部的调研中，我们发现开发者多达30%的时间被消耗在重复性和手动任务上，在编码方面这些任务包括：CRUD的业务代码编写、格式化和代码规范检查、提高单元测试及其覆盖、API文档过于老旧需要维护、调试和错误修复，等等。我们一直希望突破仅仅是“代码生成”，而是打造一个能理解项目上下文、执行复杂任务的智能助手，让开发者真正从繁琐工作中解放出来。

事实上，在2018年之前没有AI的时候就开始探索了，不过更多的是依赖于IDE自身，通过很多规则来判定交付，或者可能会推送一些可能相似的函数或者调用，帮助用户提效，但整体还是通过纯手工编码。

我们也从产品的用户画像和用户的痛点出发，做了很多调查，发现大家需求可能集中在针对新的业务，希望能快速了解工程规范和优化工程，希望AI能帮我写、帮我读、帮我分析、运行，去解决当前工程的问题；在专业的评审环节，帮我快速辨别或者发现问题；以及对于专业的行动，快速帮我搜索。

于是我们就在几年开始落地，立项做一款工具，希望加速并且提高软件的开发质量，并且能使用不同研发的阶段下，所以我们先在原有工作环境里面进行优化，做一个在主流的IDE中可以快速安装的插件，帮大家完成上面的任务，就是最早的CodeBuddy代码助手。

到2022年的时候，AI云慢慢爆发起来之后，我们通过让AI可以写代码，提升编码的速度，于是我们做了代码补全的能力：触发特定情况的时候可以自动推荐、确认、补全，帮你完成代码编程。到2023年、2024年的时候，智能体Agent进到大家的视野里，我们在更多产品形态里面可以用到它，比如通过简单对话能够帮我完成一个项目工程的理解、知识库的检索、更好地能懂我当前上下文，通过一些智能修改、内联对话的方式，快速用自然语言生成大部分完整的代码。

到2024年、2025年，在插件形态的CodeBuddy，我们推出了一个叫Craft软件开发智能体，希望能做到的逻辑是我们能不能站在用户角度，懂我、懂工程，生成用户需求要多文件项目，并通过AI进行反思、修正，并且运行，有点像后来说的Vibe Coding（氛围编程）。这时候我们已经在以Agent，也就是软件开发智能体的形态，完成一种智能体的开发协同，结合了这个模式很好的开启了和企业、个人的互联。

2025年Q2季度，我们发布了CodeBuddy IDE国际版，在产设研和规约编码上，基于

海外模型，做出了一定的成效，比如有一些用户基于CodeBuddy IDE，开发出了一个管理后端。另外在插件版本上我们也做了重构，接入国内模型和混元模型，专注在已有的IDE下的AI编码工具，提升了研发效率，在小程序IDE的合作上，深度融合小程序的效果，集成开发了小程序IDE版本的CodeBuddy的智能编码能力。

而在近期发布的CLI，其实已经在腾讯内部“吃过狗粮”了。在很多场景下，CLI版能够更灵活地嵌入到研发流水线里，支持批量代码生成、自动化任务执行以及跨项目重构。相比IDE插件，CLI更适合有标准化流程的团队和需要快速迭代的工程环境，同时也为高级用户提供了更多自定义和脚本化操作的可能。

回顾整个产品演进，我们经历了几个关键转折：

- 从单一IDE插件到多平台兼容——最初只聚焦在IDE内的辅助编码，随着用户需求的多样化，我们逐步扩展到小程序IDE和独立CLI，实现不同场景下的高效支持。
- 模型整合与优化策略——从最初单一海外模型，到接入国内模型和Hunyuan模型，使产品在不同项目、不同语言环境下都能保持较高的智能化水平。
- 从工具型到CLI提效的流程型能力——不仅是单次编码辅助，更开始覆盖任务管理、自动化执行和工程上下文感知，逐渐向团队级AI研发助手转型。
- 整体来看，每一次迭代都不是单纯增加功能，而是针对不同研发场景进行能力重塑，让CodeBuddy既能满足个人开发者，也能服务企业级团队。

> **InfoQ：你们为什么选择这种产品形态（IDE插件/独立应用/CLI）？如何决定哪些功能应该内建在CLI里，哪些应该留给IDE工具去做？**

汪晟杰：产品形态本身的选择也取决于技术本身的发展，我们最早也是从代码补全的插件开始，随着AI技术的成熟，对软件工程的改变越发深入。开发者的需求从“代码补全”转向“全栈应用开发”与“流程自动化”，AI编码产品的形态也从最早的IDE AI插件，到AI IDE和AI CLI等多种形态并存。

IDE插件、独立IDE和CLI面向不同用户和场景：

- IDE插件针对日常开发者，提供实时代码补全、跨文件分析和重构等功能，嵌入现有工作流；

- 独立IDE面向大型团队和复杂工程，支持工程上下文理解，和任务拆分，规约编程，面向产设研一体的开发平台；
- CLI面向自动化和批处理场景，用于批量API替换、全仓索引和CI/CD集成。

功能分配遵循原则：交互性强、依赖上下文的任务放IDE；批量、自动化任务、异步完成工程任务的，放CLI；依赖于本地主流IDE的开发者，对已有IDE产生粘性的用户，可以采用插件形态。整体目标是针对不同场景提供最优开发效率和AI协作体验。

> **InfoQ：在代码补全场景，你们有没有采用什么特殊优化？除去“补全层面”，在复杂refactor和大规模代码迁移场景下，CodeBuddy在这一层能做到什么？有真实benchmark吗？**

汪晟杰：在代码补全方面，我们除了常规的上下文感知补全外，还在模型侧和工程侧做了多维度优化，比如我们支持NES（Next Edit Suggestion）和补全缓存，能够快速提供更精准、可靠的补全建议。

在复杂重构和大规模代码迁移场景下，CodeBuddy CLI的优势更加明显。我们内置的“全仓记忆”机制，可以让智能体快速记住之前的总结、工程描述等。工程上下文压缩和自定义策略（Rules）是基建能力。

在真实benchmark上，我们并没有做精准的评测，而是更多的以宏观的方式去看效果，看提效。相比传统研发，新时代的AI辅助编码，可以大大降低了编码阶段的时间，也成为了开发者很强的粘性工具。

> **InfoQ：很多开发者会把国产AI编程工具和Cursor、Claude Code、Copilot作比较。在你看来，CodeBuddy的独特技术壁垒是什么？它在哪些方面不是简单的平移或替代，而是真正提供了新的价值？或有没有你们认为只有在中国市场才能跑通的独特场景？**

汪晟杰：CodeBuddy不只是一个AI代码补全工具，而是一个面向企业级复杂项目的工程智能体平台，结合全仓感知、任务级自定义Agents和本地化场景优化，提供了在海外工具难以复制的价值。在国内，很多企业有严格的数据安全、代码隐私和云端合规要求。CodeBuddy可以本地化部署、支持私有模型接入，同时兼顾国内主流IDE和国产代码

托管平台生态。这些场景在海外工具中很难原生实现，是在中国市场才能真正跑通的独特价值。

比如，CodeBuddy按照客户体量和安全要求差异，分为个人版和企业SaaS版、企业VPC专享版和私有化订阅版等多种版本，形成了清晰的商业模式。同时，我们建设了大模型新范式的研效生态体系。在具体客户合作案例中，企业客户依托我们的伙伴生态，利用CodeBuddy成功完成旧系统的改造、流水线智能化升级和企业内部泛开发者大规模推广落地。

总结来看，CodeBuddy不只是“替代Copilot或Claude Code”，而是在工程级智能体、全局上下文感知、自然语言闭环执行以及本地化合规等层面形成了技术壁垒和差异化价值。它解决的是开发效率和工程协作的深层痛点，而不仅仅是代码补全。

> **InfoQ：现在AI编程工具百花齐放，多少有点像当年的“前端框架大战”。过去没人会一周用Angular，下一周又改成React，再换成Vue；但今天很多团队在Cursor、Claude Code、CodeBuddy之间频繁切换。另一方面，各工具之间的差异也在缩小。那么在你看来，是否已经出现了判断标准，来帮助团队选择最合适的工具？**

汪晟杰：每一位开发者可能都会有自己的“体验感受”，这本身是个性化的，跟他所处理的场景和任务也有很大关系。也许他内心是有“标准”但还没提炼出来，我认为这个标准也许对我们做产品而言更加重要，因为我们需要有更普适、更明确的标尺去指导我们优化用户体验。整体而言，我们观察下来认为：

- **场景适配优先于品牌偏好。**不同工具在“代码补全”、“大规模重构”、“跨文件迁移”甚至“工程级Agent协作”上能力差异仍然存在。团队不再单纯追求新奇或明星工具，而是根据项目特点来选择。例如：小型前端项目可能更关注快速补全和内联建议 → CodeBuddy或轻量级IDE插件足够。大型后端或跨仓库重构项目则需要上下文感知、全仓记忆和Sub Agent协作能力 → CodeBuddy IDE或CodeBuddy CLI相结合更适合。
- **工程上下文管理能力成为核心指标。**过去选择工具主要看语言模型能力，但现在团队越来越关注工具是否能完整理解工程上下文，是否是一个上下文感知能力强的工具，能够减少切换成本和潜在错误，这一点比单次补全的准确率更重要。

- **可扩展性和定制能力。**随着团队规模增长，工具是否允许自定义Agent、指令、插件或任务链路，会直接影响长期效率。简单的“开箱即用”体验固然重要，但可扩展性和自动化能力决定了工具是否能在团队内部形成核心竞争力。

> **InfoQ：如果要把CodeBuddy来个定位，你们倾向于放在“能力最强”的这一端，还是“更快更便宜但能力较弱”的那一端？如何在速度和代码可维护性之间平衡？**

汪晟杰：我们的核心用户群是产设研群体，所以，CodeBuddy不是追求“最快”，而是追求“工程质量”，在此基础上尽可能做到快。我们认为未来AI编程工具的竞争，不会只看生成速度，而是看谁能在百万行级的真实项目里，让开发者少踩坑、少返工、持续维护。

> **InfoQ：目前CodeBuddy的用户规模大概是多少？技术用户和非技术用户的比例如何？企业客户占比又有多少？**

汪晟杰：目前我们拥有百万级用户，有1/4左右是非技术用户，企业客户占了40%。

技术决策的核心：上下文、规则与协作

> **InfoQ：有开发者反馈国产模型在性能上大约能发挥Claude Code七成多的水平，但在长上下文和稳定性上仍有差距。你们在选用混元+DeepSeek双模型时，实际测试的性能差异体现在哪里？在什么场景下国产模型已经足够好，在哪些环节还存在明显短板？**

汪晟杰：针对混元+DeepSeek双模型场景，在小程序上我们有做一些优化，结合小程序的知识库强化，在小程序上default+DeepSeek下，可以做到比较好的还原度，同时在工程层面上让小程序编码更智能快捷。

当然在其他方面，比如上下文支持、稳定性、代码生成的稳健程度上，还是和Claude模型有一定差距。但我们觉得AI Coding赛道会随着模型越来越强，弥补上去，我们先做好产品体验，内部我们戏称，“枪准备好了，就等待更好的子弹”。

> **InfoQ：Cursor因调整价格而陷入热议，并且通过优化上下文传递来降低推理费用，但这类方法始终有限，那么CodeBuddy对外的时候需不需要考虑商业模式？**

> **并且在成本控制上有没有新的思路？**

汪晟杰：我们不会走Cursor那种“单一大模型+涨价”的路。CodeBuddy从设计之初就考虑了商业可持续性，采用分层商业模式：个人用户用低成本模型保证体验，团队和付费用户在需要时可以调用高性能模型，而企业用户可以有选择性的选择企业私有化的开源模型，做到成本和价值解耦。

在成本控制上，我们有两方面创新：

- 智能模型路由：按场景选择最合适的模型，避免无效算力浪费；
- 上下文优化：对于非必要的上下文做好精细化裁剪，并在合适时机做压缩和总结。

> **InfoQ：很多开发者抱怨某些国产AI编程工具几乎都是按token计费，结果一旦用high了，分分钟就是上亿token的花销。CodeBuddy在计费模式上会不会有新的探索？比如订阅制、企业级套餐，或者更透明的token消耗反馈和预算上限管控？**

汪晟杰：我们意识到纯按token计费模式会带来成本不可控的问题，尤其是在高强度使用下可能瞬间产生巨大开销。因此，CodeBuddy正在探索订阅制和企业套餐，提供固定额度（credit）和团队管理能力，同时内置实时消耗反馈和预算上限管控，让开发者和企业可以清晰掌握成本。我们还在尝试智能模式切换，根据任务复杂度选择合适模型，以降低不必要的token消耗，实现可预测、可管理的使用体验。

> **InfoQ：关于上下文，长会话里，多次压缩后最初的意图可能被淡化，模型也会遗忘部分指令，虽然我们很期待模型有更大的“有效上下文”。在CodeBuddy的设计中，你们是更依赖上下文压缩算法这类的技术解法，还是尝试用外部存储（如KV store/向量检索）产品解法？是否还有其他内部需求或‘黑科技’值得分享？**

汪晟杰：CodeBuddy采用 压缩+外部存储的混合策略。通过即时上下文压缩，使得：

a. 对话或编辑历史在本地/临时内存中做轻量压缩，保证模型可以快速处理当前任务。

b. 类似“流式摘要”，保留核心任务意图和最新代码片段。

> **InfoQ：现在开发圈里很热的是“后台代理”和“看板式多代理系统”，像Cline可以在CLI开多个代理，或在GitHub Action或云进程里跑，以及还有像roo code的Boomerang任务模式。腾讯的CodeBuddy Craft会不会也尝试类似的探索？在你们看来，是一个能力不断扩展的单体代理更可控，还是并行协作的多代理在大规模场景里更具优势？**

汪晟杰：CodeBuddy CLI正是在做这类事情的探索。结合CDE（Cloud Development Environment）可以很好的异步执行，拉起环境，启动CLI，生成到执行程序，合并到主线，过程中有冲突则解决，有运行失败则反思修正。我们在Cloud Studio产品上已经落地了类似的能力，给教师端提供作业批改的Background Agent，当学生提交作业的时候就会主动触发进行作业批改，完成后将结论推送给教师。

单体代理和多代理协作要解决的问题场景不一样，都是要深度探索，不断迭代的领域，要让单体代理解决好垂直场景问题，同时又要让多个智能体之间有良好的协作节目。

> **InfoQ：很多人对AI代码助手的担心是：它会不会一直“死磕”，走错路还在不断尝试？在CodeBuddy里，你们如何设计“自动探索”与“人工介入”的边界？哪些场景你们敢放手让代理全自动执行，哪些必须要有human-in-the-loop？人工介入的checkpoint是怎么设置的？**

汪晟杰：自动探索vs人工介入的原则：

- 自动探索：适合结构清晰、规则明确、风险可控的任务，比如：批量重构固定风格的代码（rename、move file、format），执行标准化脚本操作（依赖安装、代码lint），小范围单元测试或自动修复可预测错误。
- 人工介入：涉及高风险、不可逆或者结果多样化的场景，比如：核心业务逻辑改动（finance、auth、payment），架构级迁移或重构，多模块交叉依赖的复杂变更。

在CodeBuddy中，checkpoint是人为设置的“检查点”，确保自动行为不失控。常见做法：

1. 任务级checkpoint

每个子任务执行前，弹出摘要给人类确认

例如：“我计划在文件X修改10行函数Y，你确认吗？”

2. 结果校验checkpoint

自动执行后先生成diff/日志，让用户approve才提交

可结合单元测试、lint或静态分析，自动判定是否安全

> **InfoQ：有人认为“规则和记忆本质上是一样的，本质就是提供上下文”。但在CodeBuddy里，你们区分了rules、plan mode和Spec-driven。你们怎么看待这三者的边界和各自的作用？**

汪晟杰：这个问题非常有意思，也触及了AI编程助手设计里一个核心哲学：“上下文”到底如何被管理和利用”。我可以从CodeBuddy的设计思路梳理一下rules、plan mode和spec-driven之间的边界和作用：

- Rules（规则）的本质：硬性约束或操作规范，确保安全与一致性。
- Plan Mode（计划模式）的 本质：多步策略执行，指导AI按步骤完成复杂任务。
- Spec-driven（规格驱动）的 本质：以功能或产品规格为导向，生成符合需求的代码。

> **InfoQ：很多AI编程工具在强调YOLO模式 和SOLO模式。你们怎么看待这两种模式？在CodeBuddy的设计里，会如何取舍或融合？**

汪晟杰：YOLO模式追求快速实验，效率高但风险大；SOLO模式强调自由探索，灵活性强但协作成本高。CodeBuddy通过规则引导、上下文感知和审查，将两者融合，让开发者在自由与规范间自如切换。

在CodeBuddy的理念中，YOLO模式和SOLO模式并不是对立的，而是互补的两种工作心态。YOLO模式激励开发者大胆尝试、快速迭代，避免陷入过度设计；而SOLO模式则帮助开发者在独立思考和全局把控中找到个人创造力的发挥空间。

CodeBuddy通过语义理解、通过人机交互，比如询问下一步建议、是否要继续的确认框等交互，使开发者可以在团队规范下“放心YOLO”，也能在需要深度思考时“高效SOLO”。当用户处于快速原型阶段，可以告诉系统，请帮我全部完成，那么CodeBuddy会拆解任务，甚至还会测试验证、错误修正建议，降低试错成本；而在独立探索时，CodeBuddy会强化对上下文的捕捉和个人偏好的学习，让开发者保持专注，改动少量文件，甚至是文件的光标后的预测推荐。这样，既能保持创造势能，又能减少因为随意性而带来的技术债。

> **InfoQ：规则太多容易把团队管死，规则太少又容易失控。你们认为大型团队在制定规则集时，真正的“临界点”在哪里？CodeBuddy又是如何帮助团队在这种张力下找到平衡的？**

汪晟杰：临界点其实不是固定的数字，而是团队能在自由度与可控性之间实现最优张力的点：核心规则保留关键约束：如代码风格、测试覆盖率、提交规范等。

可调整的自由区间：允许团队成员在非核心部分自主选择工具、方法、实现方式。

所以，CodeBuddy的规则层可配置化，团队可以在CodeBuddy内设置必遵守的规则（例如代码风格、模块依赖限制、测试覆盖率阈值）。

> **InfoQ：关于MCP：在CodeBuddy里，面向MCP/工具的“最小可信工具集”是什么？你们如何在开放性与产品安全性、一致性之间做平衡？**

汪晟杰：核心思想是：每个工具必须可审计、可验证、可组合，但不能无限制扩展。MCP内的每个工具都有明确的输入输出规格，避免生成代码或操作出现意外行为。

本质上，MCP就是“最小、可信、可组合”的工具集合，能够支撑CodeBuddy的核心功能，同时降低复杂性和潜在风险。

Vibe是过渡，Spec才是未来

> **InfoQ：你们提到AI会打破专业壁垒，模糊PM、设计师、开发者的边界，团队结构因此发生调整。那么在CodeBuddy+Supabase+Vibe Coding的实践里，目前有非技术人员使用的真实例子吗？**

汪晟杰：非技术人员可以直接触达代码或数据层，无需完全掌握编程。CodeBuddy+Vibe Coding让自然语言和可视化交互成为主入口。

团队结构变化：PM/设计师/运营可以承担“原型构建+数据操作+部分前端逻辑”的角色。开发者更多专注于架构、性能优化和安全控制。

真实落地：目前在一些初创团队和内部业务线试点中，PM和设计师确实在没有手写代码的情况下完成了MVP和内部工具原型。

补充一下，我们的实践不仅仅有Supabase，我们还有CloudBase，也就是腾讯云开发，他们在跟小程序的结合上也有很深的积淀。同时，目前我们也在跟腾讯生态做更好的融合，你点开CodeBuddy IDE的integration界面，会发现我们现在不仅仅有Supabase、云开发CloudBase，还有腾讯云EdgeOne Pages，针对小程序这块我们也会逐步优化，同时未来我们也集成越来越多的内外部能力。

> **InfoQ：有观点认为Vibe coding并不适合企业场景，对个人开发者也存在限制，比如技术栈固定、目前模型生成出来的PR无法保证没有Bug。你们认为Vibe coding到底是面向什么样的用户群体？它的边界和定位在哪里？**

汪晟杰：我想，Vibe Coding大致的定位可能是：“一种辅助开发的创作型工具，优化开发体验和迭代速度，而非替代专业工程师。”

它的边界是：

- 适合原型开发、个人/小团队快速迭代、熟悉技术栈。
- 不适合关键生产系统、全栈大团队核心代码、对Bug零容忍的场景。

使用策略：

- 企业可以用Vibe coding生成草稿、建议或PR草案，再由人工严格审查。
- 个人开发者可用它 加速实验和小型项目，同时保留人工掌控权。

这涉及到Vibe Coding的定义，但我们要注意Vibe Coding不等于AI Coding的全部。AI能力本身对现代软件工程会带来广泛且深入的变革，我认为这个是毋庸置疑的。因此，假设Vibe Coding真的不适合企业，但不代表AI Coding对企业和个人没有意义。这个词本身

就是就是技术发展过程中冒出来描述某种状态、某种场景的代称，本身技术、产品都是流动的。

所以我们更关注“Spec Coding”。其实这个我们在7月22日的发布上有分享，我分享一下当时那张PPT。

企业应用和生产力提升

> **InfoQ：你们怎么衡量CodeBuddy的生产力提升？你们有没有通过外部客户或内部实验，量化CodeBuddy的生产力提升？是缩短开发周期、减少bug，还是帮助新人快速上手？能分享一个典型数据点吗？**

汪晟杰：CodeBuddy的生产力提升在这几个地方都有，确实主要体现在：开发周期缩短（效率提升）、低级bug减少（质量提升）、新人快速上手（学习曲线优化）。

我们可以分享一些数据，从量化上看，平均效率提升30 – 40%，Bug数量下降约20 – 30%，新人上手速度提升约40%，这也是我们在内部实验和部分客户案例中看到的典型数据。

> **InfoQ：对于企业，工程副总裁/CTO他们关心的不只是个体开发者，而是整个代**

> 码库、工程决策。比如一个团队有100个开发者，每个人都可能在用这些工具。那么该怎么治理？代码审查怎么改？变更管理怎么改？

汪晟杰：从个体到团队，简单的说，关注点可能会有所不同，主要是从三个层面而言：

- 研发效能层面，从单个开发者效率，到团队治理效率，会更注重规则化、分级审查、自动化监控等。
- 代码审查层面，从“逐行检查”，到同时关注“架构和策略一致性”。
- 变更管理层面，从“人工把控”，到注重“自动化+风险可视化+分级合并”。

> InfoQ：面向团队与企业。有没有一些企业级的AI需求，是单个开发者平时感知不到、只有规模上来后才会暴露的？

汪晟杰：目前没有发现，而且对于我们而言，因为本来需求就是诞生于腾讯这样一个大型企业的开发实践需求，可能我们最早就同时能感知到单个开发者使用体验层面和涉及到组织协作，或者复杂软件研发流程的续期。

“成为自己，而不是下一个谁”

> InfoQ：在内部roadmap制定时，你们如何预测模型下一个迭代的能力提升点？如何确保产品和未来的模型能力相匹配。

汪晟杰：我们不会单纯“预测”模型能力，而是更关注如何把模型能力“转化”为实际生产力。产品研发流程当前是有完备的方法论的，而且目前看来在AI时代依然生效。那么我们可以围绕着产品研发的各个阶段做好场景拆解，在模型能力能够解决该场景的时候，做好产品化输出。

> InfoQ：对PM来说，这类AI工具和传统互联网产品很不一样——模型每天都在变强，功能路线图很难像过去那样提前锁定。你们的产品管理方式和传统PM最大的区别是什么？

汪晟杰：需要拥抱不确定性，找到产品价值，而非功能；也要利用好产品功能数据，快速迭代，用数据验证功能价值。

> InfoQ：CodeBuddy的团队是怎么组建的？你们在挑选人才时最看重什么能力？是有工程背景，还是懂AI应用？哪些技能是“必选项”，哪些反而不重要？

汪晟杰：我们的团队核心是一批兼具工程能力与产品思维的“多面手”，在挑选人才时，我们最看重的是一个人驾驭AI的思维和能力——也就是能否从业务视角定义问题、用架构思维拆解任务，并引导AI高效执行。CodeBuddy就像一把神兵利器，它真正增强的是那些能主动运用AI处理重复工作、转而聚焦于系统设计和价值创造的“智能协作架构师”。所以我们不再单纯强调工程背景或AI理论，更关注业务洞察、提示词工程和人机协作素养——核心是从“代码执行者”转型为“用AI创造价值”的驱动者。

> InfoQ：你们还会招聘什么样的人才？

汪晟杰：除了上面的人才外，我们也会针对研发流程来招聘垂直领域的专业性人才，例如质量领域、设计领域。结合他们的背景知识和AI能力，来打造好我们的垂类产品和能力。

> InfoQ：从未来发展来看，CodeBuddy最想成为的是什么？一个AI版VSCode？还是国内开发者的Claude Code、Lovable？

汪晟杰：可能是成为自己，而不是成为别人。在软件开发领域，我们的目标其实是要解决两个方向的问题：人机交互和自动化。基于插件和IDE的产品形态，来增强人机交互的体验，提供产设研的统一协作平台。通过CLI产品形态，集成到研发流程中，提升自动化运作效率。

> InfoQ：如果让你们对标Cursor、Claude Code、Lovable，最想在技术上超越的“一点”是什么？

汪晟杰：作为AI应用，说实在的，应用层的技术上都拉不开差距。但我们可以用户体验上、生态链接上取得优势。

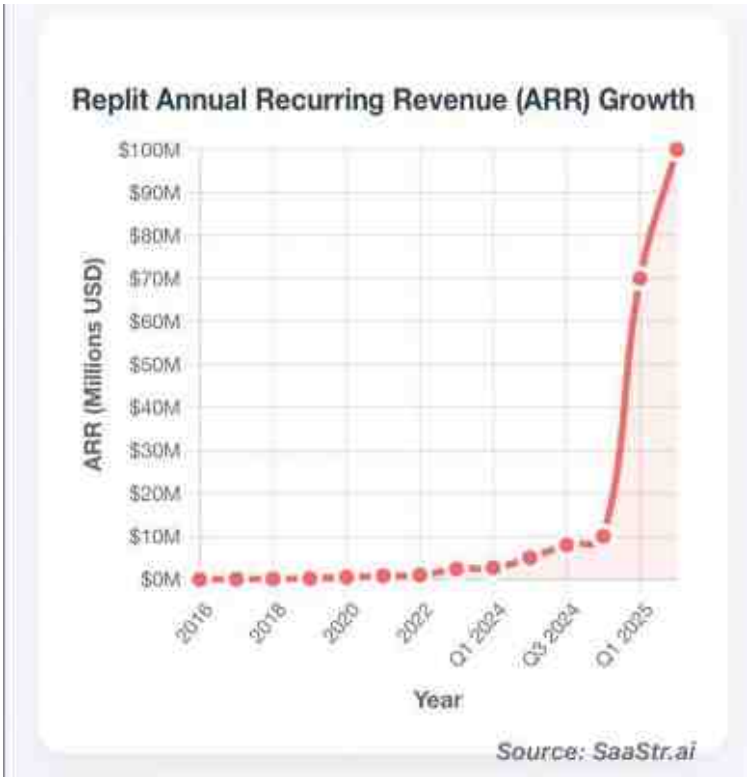
> InfoQ：未来两年，你们最想打赢的“第一场硬仗”是什么？

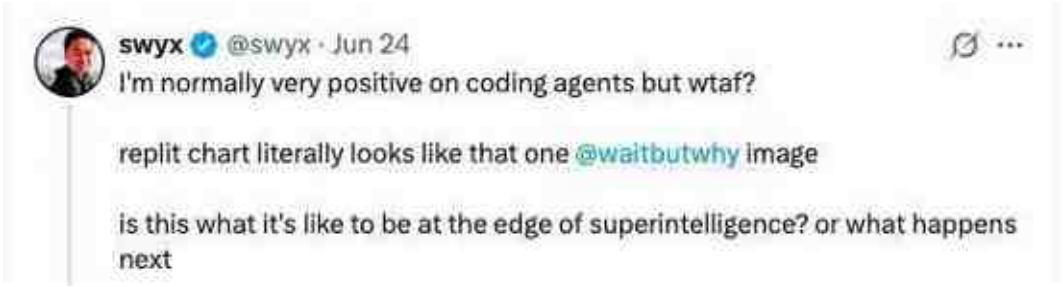
汪晟杰：让生产力走出CODING圈子，覆盖更多的场景、用户和客户。

从烧光现金、裁掉一半员工，到ARR 9个月破亿：Replit用“全栈平台”反杀Cursor，赌赢“每层都赚钱”模式

作者 Tina 傅宇琪

2024年初，Replit的年度经常性收入（ARR）还不到1000万美元，而到了2025年，仅用九个月，它就突破了1亿美元大关。这条几乎垂直的增长曲线迅速在开发者社区引发关注。





网友swyx感叹说：“我一向对编码代理持乐观态度，但Replit这张图……属实让人有点看懵。”他把这条曲线类比为@waitbutwhy曾经绘制的“智能爆炸临界点”图：变化突然、陡峭，难以预测后续走向。

Replit的增长并非仅靠AI代码生成，而是基于一套更具系统性的战略设计。有不少开发者认为，它的成功关键在于对平台层的布局与整合能力。在swyx的评论下，网友们给出了如下观点：

> "Lovable擅长制造'看起来像真的'原型，Figma一出手也许能干掉它。"
> "Cursor是个泡沫——Claude Max+自带IDE更好用。他们解决了短期扩展性的问题，但没有护城河。"
> "Replit才是真正有潜力成为小白版AWS的选手。说实话，我买账。"

还有人指出，Replit的关键在于"像Bolt一样"，原本就有基础设施能力，在行业转折点完成了及时转向。

与一些专注于智能IDE的竞品不同，Replit的路径更偏向基础设施整合。AI编程代理只是流量入口，平台真正发力的是托管、数据库、部署、监控、日志等"应用生命周期"的后端部分。这也造就了Replit的一个特点：生成即上线，构建即运行。

这使其商业模型不同于仅售卖推理服务的公司：Replit可以在代码生成环节获客，在托管与使用中变现。有人用一句略带调侃的说法总结它的策略："每个token都在亏钱，但每一层都在赚钱。"

S

图片来自：https://x.com/mattppal/status/1932798993626771556

Replit当前的增长与策略，更多反映的是一个趋势：AI编程工具正逐步从“编辑器”进化为“平台”，从“写代码”迈向“部署应用”。谁能抓住这一链条中更多的节点，或许就更有机会在下一阶段脱颖而出。

最近，David Lieb和Tom Blomfield在播客节目中采访了Replit的创始人兼CEO Amjad Msad。从裁员、融资接连遭拒到成为行业瞩目的新兴独角兽，Replit是如何“浴火重生”，用短短9个月的时间，将ARR从1000万提升至1亿的呢？他详细分享了这段历程，以及对AI编程的看法。基于该播客视频，InfoQ进行了部分增删。

核心观点如下：

- 一旦“创造”的过程变得容易，真正的瓶颈就回到了“你能想到多少好点子”。
- 去探索那些技术刚刚变得可能的边界。因为模型的下一次迭代，可能就让你今天的产品突然变得有价值，你也因此率先进入市场。
- 未来会出现一种“融合模式”——我们依然使用自然语言进行交互，但底层的展现方式不再是代码本身，而是某种基于代码之上的抽象界面或视图。
- 高互动、实时反馈的沟通才是真正推动产品前进的关键。

押上公司命运的产品

David：Replit最早的目标是让初学者可以在网页上轻松搭建编程环境，但现在你们在AI辅助编程方面的发展非常迅速，能和我们分享一下最新的进展吗？

Amjad：我们的使命从一开始就是让编程更容易接触。YC（Y Combinator）之后，我们将目标进一步提升，开始讨论让全球拥有十亿个软件开发者。

大概在2015年，AI曾经历过一波热潮，自然语言处理（NLP）也在当时很火。那时有不少AI公司，实际上背后是人工操作，最后都失败了，但也让我们初步看到了用NLP处理代码的可能性。我在最早的种子轮融资计划书里就写过，总有一天我们能收集到足够多的数据来训练模型。直到2020年GPT-2发布，我才真正觉得这变得可行了。

在这之前，我们已经构建了一些基础设施，比如开发环境、托管平台以及周边的服务。我当时想，如果我们能把AI Agent加进去，它应该能协调这些系统，效果会非常出色。我们从2021年就开始尝试引入Agent，但那时候效果不理想。一直到2024年年初，我们才觉得这个方向越来越接近可行。即使只是GPT-4.0，也已经能连续保持两分钟左右的逻辑连贯。

其实我们那时相当于下了一个很大的赌注，因为公司当时的情况并不乐观。

Tom：我正想问，你们是否经历了一个“押上公司命运”的时刻？从最初教授人们编程，转变为现在帮助每个人构建应用程序。

Amjad：是的，当时公司已经扩张到一定规模，但我们的资金消耗过大，因此我们决定裁员。我们裁掉了大约50人，之后又有15到20人主动离职，公司规模一下缩减到原来的一半不到。那时我把全部精力都投入到Replit Agent上，觉得这就是公司的关键转折点，甚至有种“破釜沉舟”的感觉，这件事必须要做成，因为我相信它是能让公司活下去的唯一机会。

坦白说，如果不是Claude 3.5在我们开发Agent的中途发布，我们可能已经失败了。因为Claude 3.5是第一个能保持五到十分钟连贯性的模型，能真正完成代码生成任务。

Tom：对于那些站在技术最前沿的初创公司来说，这种做法其实是最有效的。一开

始设定了一个使命，虽然技术尚未成熟，但先开始着手构建，等技术追上来，就刚好踩在了正确的时间点上。

David：我记得大约六个月前你说：“我们离‘完全自动化的软件开发’还有很远。”你现在还这么认为吗？

Amjad：完全不是了。过去每次我做预测时，都觉得自己很大胆，但事实证明我一直低估了技术的发展，自动化程度比我预想得快得多。就像我说的，Claude 3.5能保持五到十分钟的连贯性，Claude 3.7可能能达到四五十分钟甚至一个小时，而在Opus的系统卡中，他们提到Claude 4.0能连续工作七个小时。

David：Agent最大的限制一直是：它们能否保持思路连贯，能否利用更长的上下文做出合理判断。如果大语言模型能连续工作七小时，那基本上就是一个人类员工了。

Tom：而且它的工作速度可能远快于人类，相当于在七小时内完成了一周的工作量。

Amjad：目前我认为仍然存在一个关键限制，那就是“计算机操作（Computer Use）能力”——Agent使用电脑的方式还很糟糕。这就是Replit Agent的优势所在，它能在一次Prompt下完成整个应用开发，而不是每一步都要你陪着测试、纠错、反复尝试。

Tom：有一家叫Browser Use的公司，他们正在做浏览器自动化，效果非常不错。还有另一家叫Pig的公司，则专注于Windows桌面自动化。我给现在创业者的建议是，可以基于这些底层能力，将浏览器或桌面自动化技术应用到具体的企业垂直场景中。

Amjad：一旦这些技术真正成熟，这两家公司很可能会迎来爆发。

Tom：我觉得现在距离这些技术真正“跑通”可能只剩下几周，所以现在正是开始使用这些技术的最佳时机。

Amjad：完全同意，这正是我们目前关注的方向。Replit Agent从v1到v2在自治能力上已经有了巨大提升，而v3则是我们迄今为止构建的最具自治能力的系统。我们已经在研发中，真正让我们感兴趣的是背后的基础技术，以及它如何赋能更高程度的自治。

有几个要素特别重要，其中之一就是事务性（Transactionality）——具备回滚能力。你希望Agent的行为是安全的，就像Git让程序员可以自由试验、创建分支一样。Agent也

应该能够尝试不同路径，在出错时安全回滚。此外，它也应该能够在不同的执行路径之间进行采样（Sampling）。这对自主性也非常关键。举个例子，Anthropic在发布其SWE-bench分数时，会分别展示“未采样”与“有采样”的表现，分数会从70%提升到80%，这说明采样机制对于提升系统的可靠性和能力非常有效。

Tom：这涉及到一个思路：你同时启动多个Agent，每个尝试一种解法，然后从中挑选出最优路径。

Amjad：我们构建的基础架构是完全事务性的，并且可以“锁步执行”。文件系统是基于快照的，数据库也是快照数据库，甚至虚拟机在运行中也可以进行快照提交。因此我们可以随时分叉、创建分支。

如果“计算机操作”也能成熟，那么我们甚至可以做到这样：多个Agent同时尝试不同路径，再通过自动化评估机制进行比较。目前一些做法是用判分器（Judge）来打分排名，但那不是真正的验证器。真正的验证器是测试，是Agent实际操作计算机后的测试反馈。通过这样的方式，我们可以采样多个执行路径，找到真正“有效”的那一个，再反复复制执行，最终达到极高的稳定性与可靠性。

Tom：会不会未来，你们可以不只生成一个Agent，而是同时生成五个、十个，甚至上百万个？

Amjad：这就是未来真正有意思的地方。我们希望用户能够自行设定“计算预算”。现在一些类似O结构的模型已经开始提供预算设置，比如你愿意投入多少钱。如果你愿意给我们1000美元，我们就能把它全部用在生成和评估上。

Tom：同时生成多个分支并行执行，再挑出最好的那个，人类大脑是做不到这一点的。我们是线性思维，没法像AI一样并发决策。

David：这让我想起一个“传说”，我不确定是否属实：据说在Steve Jobs管理苹果时，他会故意安排多个团队做相同的项目，然后看哪一个做得最好。我听说OpenAI现在也采取类似做法。

Amjad：我也听说过，比如OpenAI在做Codex项目时就有多个团队同时推进。在学术文献中也有类似结论，小模型如果进行采样，其表现可能超过未经采样的大模型。比

如，经过采样的Sonnet可能比Opus更好。一些公司甚至在尝试一种类似策略：不是雇一个高级工程师，而是雇十个初级工程师，让他们完成相同的任务，最后挑选出效果最好的。

Tom：不过，用人类来做这件事代价太高，而对大语言模型来说却相对便宜。你可以生成上百个版本，然后每次都选出最优的那个。

什么该交给AI，什么必须平台托底?

David：现在大家是怎么使用Replit Agent的？主要的用户群体是谁？

Amjad：这个问题回到我们最初的愿景——只要让编程变得足够简单，就会有越来越多的人愿意尝试。其实我们刚进入YC前，Paul Graham就在Hacker News上发现了我们。他告诉我，有一个“超线性关系”：编程越简单，愿意学习编程的人数增长得就越快。

Replit的优化策略一直都是降低学习门槛，这是我们增长用户和客户的核心逻辑。现在我们看到，各行各业的人都在使用Replit Agent。特别是产品经理，他们无需依赖工程师就能完成很多有影响力的工作，比如运行A/B测试、做产品优化等，极大提高了独立性和执行力。

这也促使我们重新思考产品经理、设计师和工程师这些角色之间的边界。最近我们成立了一个新的产品团队，不再是传统的产品负责人带一组产品经理，而是将设计师、工程师和PM组成一个混编小组，大家都在使用AI，不仅能快速做出原型，有时甚至可以直接推进到上线阶段。这打破了传统的瀑布式协作流程，避免了跨角色之间沟通效率低下的问题，使得整个团队协作速度极快。

Tom：我在创业公司工作时，创意列表和产品待办事项总是无限延长，而最大的瓶颈永远是工程师的时间。但现在我在做自己的项目，我写下待办事项后，它们很快就都完成了。突然间，瓶颈变成了“我还有多少新点子可以想出来”。看着一个空空如也的待办列表，竟然开始发愁下一步该做什么，这种体验真的很奇妙。

Amjad：我听说过一个典型的案例，有家公司在内部广泛部署了Replit，连创始人都在用。这反而让工程师感到压力山大，因为创始人一个周末就能做出一个功能，反过来质问他们：“我能自己一个人在周末搞定，你们一整个团队干了些什么？”

Tom：这些人原本是技术背景的吗？他们做的是初版原型，还是直接上线？还是交给工程团队让他们重写？你们通常看到的做法是什么？

Amjad：我们一般建议这些用户与工程团队协作，但现实中并不总是如此。很多产品经理和设计师会选择绕过工程师，直接面向用户。我们发现他们大多会先把产品交给内测或测试用户，但也有一些人直接上线投入生产环境。

目前我们正在和这些公司的工程负责人深入沟通，很多人对这种做法非常不满，比如："谁来负责这套系统的运维？出bug谁修？"在我看来，所有这些问题的答案其实很简单：这些应该由Agent来负责。

Tom：那现在的主要限制因素是什么？如果我今天用Agent写了一段代码，直接上线，通常工程师会反对哪些方面？主要是哪里出问题了？

Amjad：安全（Security）问题是一个非常大的挑战。大语言模型（LLM）和人一样并不完美，它们在某些组件上的表现非常糟糕，尤其是在身份认证方面。目前的模型往往还在使用过时的加盐、哈希方法，这是一个严重的问题。

我们已经看到很多现实中的失败案例，虽然目前还没有特别严重的事故，但我认为迟早会发生。比如，一些独立开发者会意外泄露API密钥，或者没有做好登录安全保护，导致系统极易被绕过。而市面上很多工具对此并不负责，甚至直接把锅甩给用户，说这是用户的问题。

但我们不这么看。作为一个面向非开发者的平台，我们认为自己有责任为用户屏蔽掉这些高风险领域。因此，我们主动限制了一些LLM当前不适合处理的任务。比如在Replit上，如果你添加身份认证组件，我们就会自动拉取一个我们从零开发的认证模块，内置了验证码、标准安全机制，并且与你的数据库打通，用户管理也都集成在页面中，尽可能降低复杂度。

另一个我们认为LLM不该负责的模块是支付系统。支付逻辑虽然不复杂，大致就分为一次性付款、订阅和按量计费几种模式，但即便如此，我们也不建议由LLM来编写。

Tom：支付逻辑类型就那么几种，不是无限复杂。但也正因为标准化，其实更应该由平台来提供现成模块。

David：今天已经没有人会自己从头写支付系统或身份认证模块了，大家都会用成熟的服务商。现在AI时代也在重演同样的趋势，看起来这就是最好的发展路径。

Amjad：除此之外，我们还与一家出色的安全公司Samrip建立了合作关系。现在每当用户部署一个Replit应用时，系统会自动运行安全扫描，对代码进行检查，并生成包含警告和错误的详细报告，Agent甚至可以尝试自动修复这些问题。

Tom：我能想象未来还会遇到其他问题，比如可扩展性、数据库的N+1查询问题、性能瓶颈等等。你有没有一份明确的“阻碍清单”，列出在实现“真正一键部署”前需要解决的所有问题？

Amjad：我认为，未来最大的限制因素其实是“人”本身，特别是社会层面的不信任感，这恐怕只能随着时间自然演进。把这类问题放在一边不谈，企业要适应新技术，还需要在其他层面上做出改变，比如提升可扩展性的检测能力。目前我们还缺乏类似模糊测试或对抗性Agent来主动攻击和测试应用系统的机制，而这将是未来非常重要的一环。

另一个重要方向是企业系统集成。我们正在研发的一个功能，是支持企业将自己的设计系统导入Replit。很多公司已经有完整的UI组件库，我们希望Replit在进入这些大公司后，能够无缝对接它们已有的内部系统。

构建护城河：“大量工程投入其实在基础设施上”

David：这让我想到当前不同类型编程工具所构成的“光谱”。一端是所谓的“能力增强型工具”，比如Cursor和WindSurf，它们服务于专业开发者，用于提升效率。另一端是面向消费者的“低门槛工具”，比如那种让任何人都能快速搭建App的平台。你们似乎处于这个光谱的中间地带——既帮助企业交付真实产品，又服务于那些非传统意义上的程序员。

你怎么看这个格局的发展？未来会不会逐渐统一，还是会长期存在多个并行工具？

Amjad：如果出现AGI，那显然就是“统一”了。但先撇开那个世界不谈，目前很难为那种场景做出规划。就现实而言，“如何逐步提升现有工程师的生产效率”是一个非常明确、竞争也非常激烈的市场。无论是像Cursor这样的应用公司，还是底层模型公

司，都在往这个方向投入。

比如Claude Code和Cursor之间就已经存在竞争关系，而Cursor又依赖Claude。可以说，这是一场激烈的混战，但市场足够大，我预计最后会出现一些整合，可能不是一家独大，但最终可能收敛到两三家主导者。

我们的服务对象是“知识工作者”——理论上，任何一位知识型员工都应该能够借助软件解决问题。

Replit的愿景，就是成为“通用问题解决器”：无论是个人生活还是工作场景，我们希望都能帮你解决问题。因此，这个市场会更加多样化，参与者也会根据各自优势定位在不同层次。Replit的目标，是实现面向非工程师的“自治式编程”。我们希望用户不必担心安全、不必管理系统，只需要带着想法来到Replit，担任一个“Agent管理者”的角色。

我们努力将这套机制尽可能“人性化”，并融入用户的自然工作流中。工程师与非工程师之间的一个显著差别是：后者并不会一天八小时坐在电脑前，所以移动端对他们而言非常重要，因此我们也开发了体验非常优秀的移动应用。

我们现在在尝试一种新的使用方式，叫“环境式开发（Ambient Building）”：你可以在电脑上启动一个项目，然后离开，拿着手机收到Agent发来的消息：“我做完了这个功能，你还需要别的吗？”你可以直接通过手机继续推进。

Tom：在像Cursor或WindSurf这样的工具中，主界面元素很明确，就是代码。你看到的是代码改动和差异对比，外加一个小型对话窗口，整个体验以代码为核心的。

但在Replit这样的工具中，主要界面是图形化的，是按钮和所见即所得（WYSIWYG）式的构建体验。这种方式在搭建用户界面时非常好用，但当你尝试构建更复杂的逻辑流程时，就会觉得有些困难。因为你看不到代码，也无法清晰了解背后的运行机制，整个系统有点像个“黑盒”。

如果我们把场景拉到更复杂的企业级内部工作流程，那么产品经理或运营经理如何理解整个流程及其逻辑分支？他们该如何可视化这些东西？

Amjad：如果我们回顾计算机历史，就会发现“可视化编程”这个概念其实早就有了，但它从未真正成功。原因在于，过去这些系统通常无法达到图灵完备，无法成为真正的通用计算平台。而现在我们进入了代码生成（Code-gen）时代，它是图灵完备的，理论上可以完成任何计算。但人们与之交互的方式主要是自然语言，而自然语言本身是模糊的，这就导致很难确定系统是否在做正确的事情。

我认为，未来会出现一种“融合模式”——我们依然使用自然语言进行交互，但底层的展现方式不再是代码本身，而是某种基于代码之上的抽象界面或视图。

你可以想象类似Smalltalk的系统。Smalltalk是最早的面向对象编程系统，Alan Kay甚至认为后来的系统都不能算是真正的OOP。在Smalltalk中，开发者并不通过文件操作代码，而是通过“对象”与代码交互。这个思路或许能为我们提供一些参考。

我相信我们正走向一个“代码抽象视图”的世界，人们将不再阅读或修改代码本身，而是通过某种更具逻辑结构的方式来理解和控制代码行为。

Tom：我感觉这中间其实还有一个“开放空间”，比如是否可以使用伪代码（pseudocode）那样的方式，它看起来像英语但更有结构化？又或者用更先进的可视化拖拽界面？

David：我过去与工程师、设计师等团队合作做产品时，沟通方式大多是口头的，或者通过书面形式表达抽象想法，我们会一起在白板上画图，做系统流程图，也会通过真实的应用去测试，指出哪里慢了、哪里坏了。这种“多模态+高度灵活”的沟通方式其实是非常理想的。我相信我们终将实现一个这样的界面——产品作者依然以这种方式工作，只不过他们交流的对象不再是人类团队，而是AI Agent。

Amjad：在产品经理这个领域，有没有尝试过让沟通方式更正式、更结构化？

David：确实有尝试让产品沟通更正式一些，但效果并不好。最常见的形式是PRD（产品需求文档），但我认为它常常沦为一种“表演性产出”——特别是在大公司里，它只是为了给升职评审准备的一个形式上的成果，而非真正推动工作进展的工具。

对我来说，最有效的交流方式还是白板讨论。比如“我们这里想实现什么功能？”、“这里可能会有问题。”、“我们是不是忽略了某个因素？”、“那我们需要重新思考

整个方案。”这些高互动、实时反馈的沟通才是真正推动产品前进的关键。

Amjad：AI其实也可以在这些场景中发挥作用。我最近接触了一个叫Granola的初创公司，它可以录音会议并自动生成转录文稿。他们推出了一个团队版产品，所有会议记录都会被集中整理，还提供了移动端应用，用户可以把手机放在桌上录音并实时记录内容。

我在想，也许我们该走向一种“Granola极致主义”——也就是不去对抗企业沟通日益“口语化”的趋势，而是拥抱它。现在公司内部的沟通很多都发生在Slack里，或是在各种会议里，甚至包括和AI Agent的对话。与其强行推进文档化的规范，不如发展一套AI工具，在后台自动完成信息记录、结构化、归档和搜索，让语音沟通也能变成系统性知识资产。

Tom：什么时候我们会迎来第一个能主动参与会议的AI呢？比如你和设计师正在头脑风暴，AI也插进来说：“我有个想法，要不要试试这个？”

Amjad：很多人对AI抱有末日论的看法，觉得它会取代所有人的工作，但我认为不是这样的。未来的工作会更“人性化”，更加互动、多模态，也更有趣。

David：上次我们聊的时候，你刚刚发布Replit Agent，当时用户增长非常快。现在情况如何？

Amjad：Replit Agent上线以来，我们的月复合增长率达到了45%。

David：这是我们通常建议YC创业公司在早期用户基数为零的情况下努力追求的指标，而你们是在已经具备规模的基础上实现的，非常惊人。

Amjad：确实如此，但这种增长也给公司带来了不少压力。我们的系统基础还相对较小，一旦增长太快，很容易走偏，开始追求错误的目标。尤其在AI领域，如果你只关注收入增长，很容易出现一个问题：用户花了更多的钱，但并没有获得更好的体验。

事实上，在某些情况下，也许并不应该追求极快的增长，因为真正重要的是用更低的成本给用户带来更好的体验。所以我们在Replit并没有收入目标，而是更关注产品质量、用户留存等与用户体验直接相关的指标。

Tom：很多AI公司会出现一种不良增长模式：虽然收入快速增长，但客户流失率却接近100%。最终，这种增长方式是不可持续的。

Amjad：是的，而且它们的毛利率通常也很差。也就是说，用户越多、增长越快，公司的财务状况反而越糟。

Tom：那投资人怎么看这个领域？他们能分辨得出这些产品之间的差异吗？

Amjad：说实话，对他们来说很多时候很模糊。当投资人刚开始关注这个领域时，通常只会用每个产品三分钟。而在三分钟之内，这些产品看起来都差不多。

不过我认为，随着时间推移，这些产品之间的差异会越来越明显，尤其是在各自专注的方向上，逐渐会出现“聚合”与“分化”的趋势。未来一年，这个格局可能会清晰许多。但目前我们和投资人交流时，他们还是挺困惑的。他们不太理解这些系统，也不清楚它们将往哪儿发展。

Tom：最近Cursor和WindSurf都宣布了一些技术更新，比如在Claude、Gemini或OpenAI的基础上，叠加自研模型和API，比如“Fast Apply”等。Replit是怎么做的？

Amjad：很多时候，我们其实是在“修补”底层模型的问题。以“Fast Apply”为例，它之所以重要，是因为目前几乎所有主流模型在处理“差异化修改（Diff）”方面都做得不够好。

Tom：可以解释一下什么是“Diff”吗？

Amjad：假设你要让大语言模型编辑一个文件，最理想的方式是生成一个“差异文件”——也就是告诉系统“删掉这三行，插入另外三行”。但现实是，大多数模型在处理这类任务时表现很差。它们在识别原始代码行数、定位修改点方面经常出错。因此，很多公司一开始的做法是直接重写整个文件——不管它有几百行，模型都全部重新输出。

Tom：听起来又慢又贵。

Amjad：所以我们会尽量引导模型“懒惰一些”——也就是说只修改必要的部分。但这样一来，又很难直接应用这些修改。因此，我们还需要另一个模型来执行这些修改操作，也就是负责“应用Diff”。

我们可以选择训练一个专门的模型，或者使用像Gemini Flash这样的小模型。有时也会在多个模型上做组合优化，补上各自的短板。这其实更像是工程问题，而不是科研问题。我们不是在重新发明模型，而是在用工程手段把已有模型组合起来，构建出一个真正可用的系统。

Tom：我注意到你们并没有像Cursor或WindSurf那样在产品中让用户选择底层模型。比如其他产品会有一个下拉菜单，让你选择“我想看看Gemini怎么看这个问题”。你们为什么不这么做？

Amjad：我们在评估（Eval）上投入了大量精力。我认为这是协作式AI编程中被严重低估的一部分。我们花很多时间评估新模型、编写和生成评估数据，分析用户反馈和使用情况。一旦有新的前沿模型发布，我们会立刻进行测试。比如几个月前Gemini发布时，我们马上试用了。它在某些场景下的One-shot效果比Claude还好，但在工具调用和Agent编排方面表现一般。不过用户往往只看到宣传热度，然后就说“我想用Gemini”。

Tom：那你们在这些模型发布之前，会提前收到通知吗？还是说发布当天你们才开始测试？s

Amjad：我们和Google、Anthropic、OpenAI都有良好合作关系，通常会提前拿到模型的Checkpoint。我们会第一时间进行尝试。特别是Anthropic，我们几乎总是在模型发布当天就能上线新功能。我们有时也能预判模型发展方向，比如Claude 3.5到3.7时我们就开始为4.0架构做准备。

但说到底，我们的大量工程投入其实在基础设施上。比如我们构建了一个分布式快照型网络文件系统，这个系统花了我们两年时间，市面上没有现成的解决方案。还有安全性方面也很复杂，比如在Replit上，用户只要注册账号就可以获得云端虚拟机。要在这种环境下抵御恶意行为，比如加密矿工的攻击，是非常难的。我们还用了NixOS，这是一种声明式、可事务回滚的操作系统。我们在全球各地的计算节点上都部署了多TB的缓存硬盘，预加载了所有软件包，这些会自动挂载到每个容器上。

所有这些设计都体现了一个核心理念：事务性。你需要一个安全、可回退的系统，这样用户和Agent才能大胆地试验、采样、切换路径。这些底层工程虽然不像“发布一个新模型”那样吸睛，但它们才是我们真正建立长期技术优势的关键。

David：没错，风投口中的“护城河（Moat）”，我更倾向于理解为“复利型优势”：你领先的那个方向，会让你越跑越快。你们的这些基础设施，正好就是个很好的例子。

Amjad：真正的“护城河”往往在公司成立几十年之后才显现。比如Netflix，很多人当初不看好他们，但事实证明，他们建立了一整套强大的内容生产系统，这就是Disney后来无法超越的地方。

AI时代的能力边界与新机会

David：我有几个年幼的孩子，我希望他们未来成为有创造力的生产者。你觉得我应该让他们学编程吗？现在的“学编程”到底意味着什么？

Amjad：如果你想成为专业的软件开发者，那去读计算机科学专业、掌握基础知识当然是有意义的。但如果你希望成为一个创造者，或者在这个世界里作为一个通才发挥作用，我认为就不再需要用传统方式去“学编程”了。你可以通过“渗透式”学习来掌握它——比如直接上Replit去做项目，在过程中你自然会遇到一些需要查看代码或日志的问题，靠自己的探索能力和Google查资料的能力，你就会慢慢学会。

这其实也是我们那一代人学编程的方式，只不过后来编程逐渐变得“工业化”甚至“形式化”。以前我们用记事本写一个HTML文件就能做出网页，现在你得学一大堆工具，比如Webpack，门槛高了很多。但我认为未来的工作图景是模糊的，我们顶多能预测个大概。所以我对我自己的孩子的教育目标，是让他们拥有尽可能广泛的知识面，成为通才，更重要的是成为“生成型”的人——能源源不断地产生新想法。

因为一旦“创造”的过程变得容易，真正的瓶颈就回到了“你能想到多少好点子”。所以，我不会把“学编程”排在第一位，而是会鼓励他们“学会创造”——用代码创造、用视频创造、用AI创造任何东西。

Tom：你怎么看SaaS的未来？如果我们很快就能说一句话，比如“帮我做个Google Calendar”或“克隆一个DocuSign”，那整个SaaS行业会变成什么样？

Amjad：现在就已经有不少用户在用Replit取代原本每年数十万美元的SaaS工具。有人被某公司报价15万美元的软件，自己用Replit做出来，花了400美元，然后卖给公司3.2

万。

我认为，那些拥有开发者生态和插件系统的平台型公司还比较安全，比如Salesforce这种系统你不太可能用AI随便“即兴编写”出来。但很多垂直类SaaS恐怕危险了，我猜他们的一些关键指标现在就已经开始下滑了。

Tom：你会给当下正在创业的创始人什么建议？

Amjad：去探索那些技术刚刚变得可能的边界。因为AI或模型的下一次迭代，可能就让你今天的产品突然变得有价值，你也因此率先进入市场。

有前瞻性思维的创始人其实挺少的，很多人并不真正花时间去思考未来可能的走向。也许以前这样做并不被鼓励，但现在，具备预测能力，是一个非常重要的竞争优势。你要敢于做判断，然后构建一个“当前看起来很烂但模型一变就能立刻变好的产品”。

参考链接

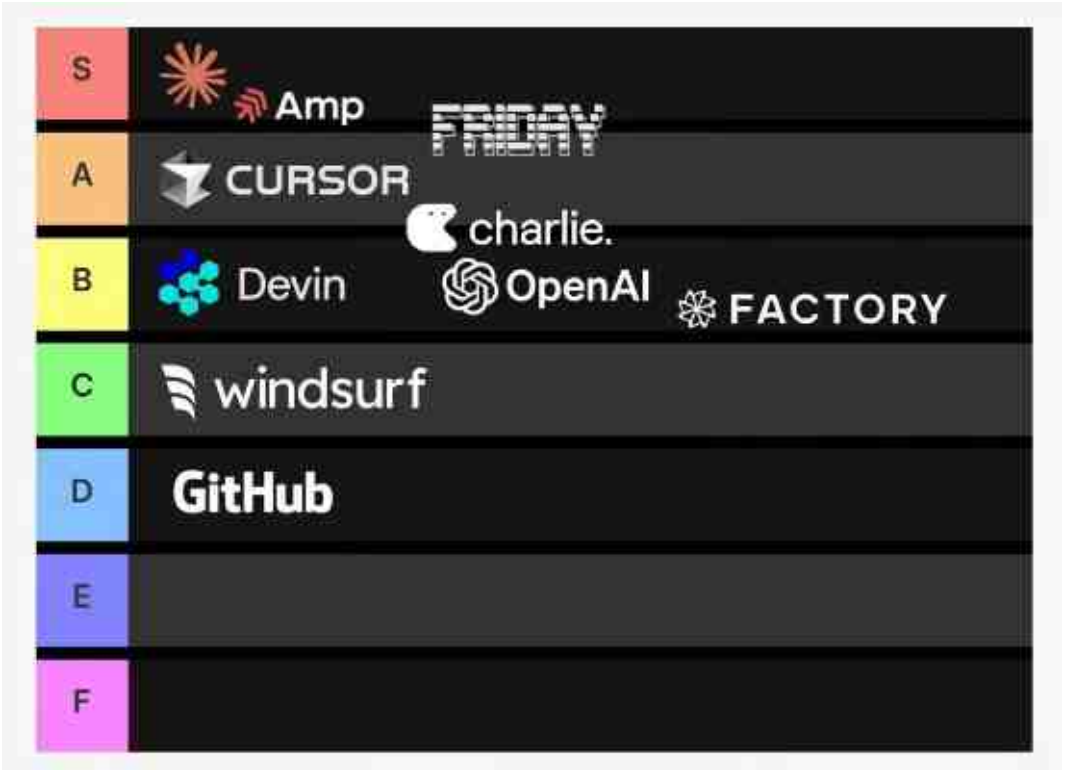
- https://www.youtube.com/watch?v=kOyIjt6FUrw

Claude Code唯一对手！？AI编程黑马AmpCode崛起的秘密：不设token上限，放手让AI自己死磕代码

作者 Tina 平川

近期，AI编程领域又一匹AI Coding黑马正在快速崛起。在一次对主流AI编程产品的评级分类里，唯一与Claude Code并列S级的，是Sourcegraph最新推出的AmpCode，而爆火的Cursor也只排在了第二档次的A级。

那么，AmpCode究竟有何独特之处？Sourcegraph工程师Thorsten Ball在近期一档播客中分享了这款产品背后的理念与AI编程的范式转变。

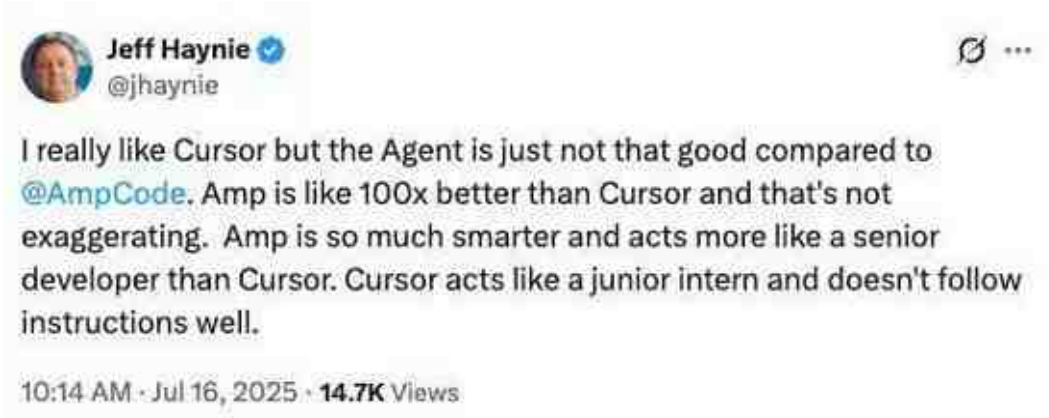


Thorsten透露，AmpCode的研发实际上早于Claude Code的发布。两者虽独立发展，但在智能编程助手的核心设计理念上却不谋而合。在他看来，AmpCode和Claude Code目前代表了最具“代理性”（agentic）的AI编程产品：它们不仅能调用工具，还真正“参与”开发流程，具备高度自治能力。

而与Cursor、Windsurf等交互过程不够直接的产品不同，AmpCode在架构设计上做出了关键决策：

> “我们选择了放权——把对话记录、工具访问权限、文件系统权限全都交给模型，然后放手让它去做。”

Thorsten表示，这种“让出控制权”的方式极大释放了大模型的潜力，也重新定义了开发者与AI的协作边界。这一大胆的设计思路也得到了不少一线开发者的共鸣。



> “我一直挺喜欢Cursor，但它的代理能力远不如AmpCode。Amp像是一位资深工程师，而Cursor更像一个不太听话的实习生。”
> ——Jeff Haynie，Agentuity创始人兼CEO

我们翻译了这期播客，全文可见下文（有部分删节）：

让LLM像工程师一样“死磕问题”

> **主持人Jerod Santo：你4月在ampcode.com上写的那篇《如何构建一个代理》（https://ampcode.com/how-to-build-an-agent）给我留下深刻印象。它用Go逐行讲解了一个简单却实用的编码代理，帮我理解了代理的工作原理。能说说你写这篇文章的初衷吗？也许能让听众了解，想在本地跑一个代理其实没那么难。**

Thorsten Ball：当然。我写那篇文章，是因为我被模型的表现震惊到了，忍不住想分享。当时我在Sourcegraph内部写了份文档，后来整理成了这篇公开博文，反响很好，可能是我有史以来最受欢迎的文章之一。

起因是我和Sourcegraph CEO Quinn一起研究后来的Amp项目，用的模型是Claude 3.7和Sonnet 3.7。我们发现这些模型已经强大到，你只要给几个简单的工具，比如读取文件、列出目录、执行终端命令，它就能开始自动写代码了。以前需要很多繁琐设计，现在只需提供能力，模型自己就能推理并完成任务。

有次我让模型改一个文件，虽然我没给它“编辑文件”的工具，但它竟然用echo命令修改了文件内容。它理解了我的意图，并找到了替代方法实现目标。当时我坐在编辑

器前，彻底震惊了。这是一次“我好像看到了AGI”的时刻，虽然只是玩笑说法，但它展现出的智能确实令人印象深刻。

为了传播这个发现，我写了这篇文章。它只用了约300行代码，读者纷纷跟我说，他们用Python等语言实现了自己的版本，也有不少人说“我也有类似的震撼瞬间”。

这篇文章受欢迎的原因之一，是它很“去神秘化”。你不需要看一堆Karpathy的视频去理解代理，也不需要研究神经网络原理。只要了解LLM是怎么通过对话调用工具的，写点代码，跑一下，你就会恍然大悟。很多人跟我说，看了文章后才第一次真正理解了“代理”。

> **主持人Adam Stacoviak：你刚刚说“感受到了AGI”，这是什么意思？你是认真的吗？**

Thorsten Ball：只是半开玩笑，我并不是真的在说AGI实现了。我想表达的是，这种模型已经可以结合反馈，自主采取合理行动，解决实际问题。比如说“重启Nginx”这种命令，它会检查sysctl Nginx restart。如果命令失败、没有响应，它会查看错误信息，然后尝试其他方法。我见过这种情况：它会说，“等等，Nginx真的在运行吗？我先查一下。”接着它会运行ps命令找出Nginx的进程，获取对应的PID，然后再去/proc目录里找出这个进程对应的可执行文件路径。基于这些信息，它最后会回到终端，执行类似“restart my Nginx”的命令。

关于AGI，我们当然可以讨论它究竟是什么、又不是什么。但我一时找不到更准确的词来描述这种情况：模型观察自己正在做的事情，查看了记录并根据feedback采取行动，最终设法完成目标。这虽然不一定是AGI，但……谁知道呢。

> **主持人Jerod Santo：你提到你看了“记录”文件，那是什么？里面有什么？**

Thorsten Ball：记录文本就是整个对话过程的完整记录，包括每次的用户输入和模型回复。工具调用的机制其实也不复杂：你事先告诉模型，“当你需要读取文件或执行命令时，请以某种格式回复”。它就像你跟朋友说，“你一眨眼我就抬手”，然后你们以这个规则沟通。模型只是按设定好的协议格式发出“调用工具”的信号，背后并没有魔法，整个流程其实非常透明、可控。

主持人Jerod Santo：我喜欢这个类比。

Thorsten Ball：所以你只需告诉模型“你是编程助手，有三个工具：读取文件、列出目录、执行命令”，然后开始对话。当用户说“帮我看下readme”，然后模型说“让我读一下那个文件。就像是说，这就是我想做的事情。”实际上，你把这个发送给提供商，给Anthropic、谷歌、OpenAI，然后返回了响应，说“助手或模型没有完成任务。它想调用一个工具。”然后你看它具体想调用什么工具。然后你“执行工具”，只需用给定的参数运行那个函数，然后你把结果发送回去。所以这很简单。如果你在UML图上画出来，这很简单。其魔力在于通过这种方式能够实现多少功能。

所以如果我们回到第一个例子，你会问——你有三个工具：列出文件、读取文件和运行终端命令。然后你说，“这是什么项目？”这就是你要问的。然后模型——这就是为什么我一直在说，就像我们一样——会做的是，“好吧，让我列出文件。让我看看这个目录里有什么。”然后你执行列出文件的操作，把执行结果发送回去……可以只是一个字符串的列表，或者是一个包含换行符的字符串。或者只是ls-l或类似的东西。有了这个文件列表，它自己就会说，“哦，我看到你有一个go.mod文件。”或者“我看到你有一个package.json文件。”或者“我看到你有一个pnpm日志文件。我假设这是一个Web应用，因为……让我在另一个文件中查看你是如何定义的，或者这个文件中包含了什么，它是如何被记录的。”然后它自己继续探索其他文件。再说一次，我想我在过去的20分钟里已经说了18次了：这太震撼了。真的，很疯狂，这些小小的工具能够实现这么多功能。

主持人Jerod Santo：有趣的是，这是一个非常基础的算法，对吧？

Thorsten Ball：是的。

主持人Jerod Santo：就像是说，“循环直到你找到了解决方案。”

Thorsten Ball：是的。

主持人Jerod Santo：其实我们人类工程师解决问题的方法，无非就是坚持还是放弃。而这玩意儿（LLM）不会放弃，它只会用蛮力不断尝试。

如果你问我，“Jerod，我需要解决一个问题，比如读取一个文件”，我会先试试最

常见的办法。如果不行，就再换一个思路，再试一遍。一直试到所有办法都试过了。如果还没解决，我就去问朋友、上Stack Overflow，或者现在，我可以问LLM，获取更多思路——“给我更多方法，直到我搞定。”

过去几十年来，一个优秀程序员的标准就是：在遇到问题时，你能坚持多久直到解决。有些人会中途放弃，另一些人则坚持到底，并在过程中积累经验，慢慢能更快选中对的方法，提前跳出循环。

其实这就是个很简单的算法：“不断尝试，直到成功。”它的本质是用蛮力反复尝试，直到奏效。而LLM能做得更快、更广泛，因为它掌握了几乎所有做法的索引。它没有自己发明什么新方法，只是用速度和覆盖率震撼了我们。

这也正是让人惊讶的地方：它尝试了我没想到的办法，而且做得极高效。所以“震撼”这个词，我觉得非常贴切。

Thorsten Ball：关于这个算法的简单性，我们这几个月一直在讨论的一点是——其实在过去一年里，我们看到很多原本依赖工具实现的功能，现在已经被整合进模型本身了。我的意思是，一年前，这些模型在工具调用方面还不太行。那时候你可能会说：“这是这个文件的内容，你能帮我编辑一下吗？”然后模型会回复你，你还得反复提示它：“请用这种特定的diff格式回复。”接着你需要自己解析这个diff，再手动应用，或者调用另一个模型去应用。

而现在，这些能力已经融入模型内部了。你只需要提供工具，它就能自己完成这些操作。看起来就像一个简单的for循环。

有趣的是，如果你去问一百个工程师，有一半会说：“这不就是个for循环嘛。”而另一半则会笑着说：“这不就是个for循环嘛……太疯狂了。”

现在一切都在模型里了。你只要输入五个指令，然后问它：“接下来我该做什么？”它就会试着做另外十五件事——因为它能从先前的上下文中推断出，接下来最合理的动作是什么。我现在都不想再用“疯狂”这个词了。只能说，这真的太离谱了。

Amp如何适应快速演化的模型

> 主持人Jerod Santo：这背后显然是个巨大的机会，Sourcegraph正在抓住它，其他公司也在行动。我们知道Google刚加入战局，OpenAI、Anthropic也都在做，开源阵营也很活跃——说明这个方向既有价值，也有潜力。你是Sourcegraph内部项目Amp的创建者之一。几周前我们和Steve Yegge聊过，他提到可以试试OpenAI Codex、Amp、Claude Code等不同组合，会有不一样的体验。正如Adam所说，我们今天就想深入了解Amp。当然，Gemini CLI也值得一提——这个月刚发布，你可能比我们更早体验了。我刚下载，觉得挺有意思：免费、开源、无限使用，谷歌果然一出手就不简单。不过先说说Amp吧。Sourcegraph是怎么定义Amp的？Steve一直说这是面向企业的解决方案，我也想听听你怎么看。

Thorsten Ball：Amp是在今年2月构建出来的，现在想来好像已经是很久以前的事了。那时正是一个转折点：Claude 3.7发布后，人们突然意识到，大模型在工具调用方面的能力已经非常强大了。你可以非常快地把一个原型跑起来——这也是我们写那篇博客的原因。而Amp的基本假设就是：只要你不去限制模型的发挥，它的表现会非常惊艳。

我们做的就是给模型足够的token。这也是为什么Amp的定价可能比其他产品更贵——我们没有强行去匹配什么“20美元/月”的订阅价位，也不做输出限制或精简token。我们选择的是：给模型足够的空间，给它一套精心挑选的编程工具，让它跑起来，别挡路，把控制权交还给模型。

项目刚启动时，我们自己也很惊讶它的效果有多好。很快，Quinn和我就开始用Amp来构建Amp，我们整天互发消息：“太厉害了，它又自动完成了这个、那个。”可能没人会相信，但我们其实是在Claude Code发布之前就开始做这套东西的，后来Claude Code才发布。我现在仍然认为，在所有这类工具里，只有Amp和Claude Code是最具“代理性”（agentic）的。

当然，Cursor和Windsurf也是很出色的产品。但它们的代理模式感觉慢一些，用户与模型之间还有某种抽象隔层，交互过程显得不够直接，像是被包了一层壳。我们当时明确做出了一些不同的产品决策，比如说：不要每次修改都手动确认；不要剥夺模型访问文件系统的能力；也不要限制它修改之前的对话内容。相反，我们选择了放权——把

对话记录、工具访问权限、文件系统权限全都交给模型，然后放手让它去做。

我觉得现在越来越多的人开始意识到，这种方式真的很强大。

那么Amp是给企业用的，还是给个人开发者用的？我是个个人开发者，我非常喜欢用它。我认识很多个人开发者也同样喜欢。

谈到企业级应用，我认为关键在于，Sourcegraph在与大客户及全球顶尖软件公司合作方面的专业经验，为我们赢得了客户的信任，使我们具备了根据他们的需求构建东西的能力。我们知道他们的代码库是什么样子。我们看到，他们拥有的成千上万的大型存储库。那在其中发挥了作用。但我不会做这样的市场定位，“好吧，Claude Code是为个人开发者准备的，Amp是为企业开发者准备的。”对我来说，Amp是为所有人准备的。每个想要强大工具的人。

听起来确实有些疯狂，但我们现在已经进入了这样一个时代：个人开发者每个月愿意为AI编程工具花上几百美元。你们俩也在这个行业很久了，应该知道这意味着什么。放在两年前，如果我跟你说“一个开发者为了业余项目，周末会花50美元买token来写代码”，听起来简直不可思议。

但我们已经接受了这样的变化。这就是如今提升生产力的方式。这就是现在写代码的常态。

所以如果你还在说，“个人开发者用不起这个工具，每次最多只能花五美元、只能跑几个请求”——那Amp可能就不是你要找的。Amp是为那些想要最强AI代理、愿意为其付费、然后放手让它跑的人准备的。

还有一点是从更实用的角度讲：Amp是一个命令行工具（CLI应用），也有VS Code插件，支持VS Code、Cursor、Windsurf、Codium，甚至连Firebase的Web版VS Code也能用。你不需要换编辑器，也不需要换IDE，直接就能上手。

还有一点是Amp与其他工具的重大区别：我们有服务端组件。这意味着你和模型的所有对话都可以和团队成员共享。他们可以看到你是怎么使用代理模型的，可以生成对话链接，也可以看到使用情况排行榜，比如每个人烧掉了多少token、生成了多少代码行数……

这在大型企业里特别受欢迎。你可能想不到，但我们后来发现，大企业内部其实存在一个巨大分歧：一部分人已经意识到这些工具的强大之处，积极推动在工程组织中推广；另一部分人则非常怀疑，使用效果也差距明显。

所以当我们向客户（或潜在客户）展示说："看，Amp不仅能做这些操作，还能分享prompt、对话记录和结果"，他们会立刻说："太好了，这样我就能把用法分享给团队其他人，告诉他们我是怎么写提示词、怎么建立反馈机制的。"

这就是我们产品的大致轮廓。

还有一件我认为非常重要的"元"层级的事是：我们一开始构建Amp时就假设——每一周模型都有可能变得更强大，很多原本需要外挂工具处理的能力又会内化进模型本身。所以我们必须随时准备好迭代产品。

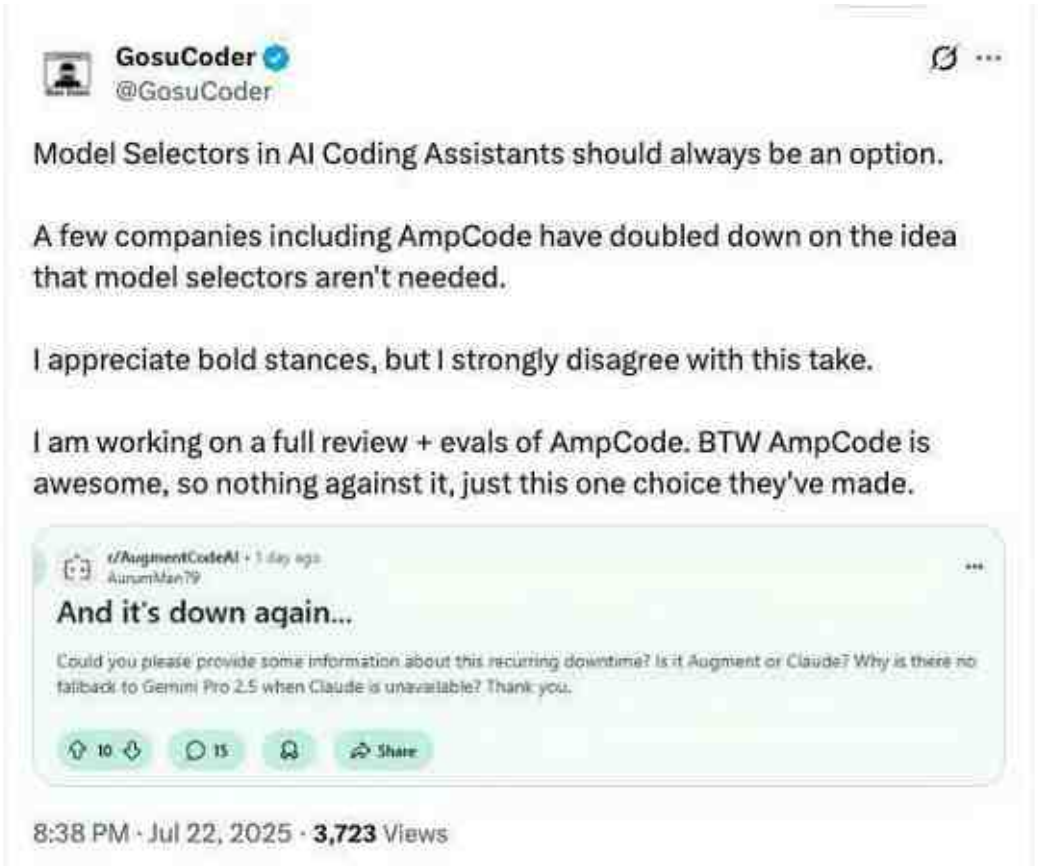
你们如果采访过Sourcegraph CTO Beyang，他几个月前说过一句让我印象深刻的话。他说：

"在AI驱动的时代，过去15～20年里创业公司用的那套'试错-找到产品市场契合点-规模化'的老剧本，可能已经过时了。"

因为现在，可能你刚刚找到了PMF，下一波技术就来了，直接把你现有的优势掀翻。你根本没法说"我们定住了、可以开始扩张了"，你必须随时准备随着技术变化而调整。我们正在经历的是一个技术持续剧变的时期。

所以我们从一开始就接受了这种节奏。我们构建产品的方式是：尽量"让开"，别挡住模型本身的能力。我的比喻是：在模型周围搭建轻量的、木质的脚手架，当模型变得更强时，脚手架自然会坍塌，用户就能重新获得模型原始的全部能力。

这也是我们一直秉持的产品哲学：保持简洁、灵活、快速响应。



> AI编程助手中的模型选择器应始终作为一个选项存在。包括AmpCode在内的几家公司坚持认为，模型选择器并不是必需的。我尊重这种大胆的立场，但我对此观点非常不认同。

哦对了，我本来一开始就该说——我们没有“模型选择器”。我们会根据当前任务自动选择我们认为最适合的模型，并随时准备切换。如果明天出了更好的代码模型，我们会第一时间采用。

我们的目标不是让用户在“高端模型”“低价模型”“快速模型”等十几个选项里选来选去，也不是让用户在ask模式、planning模式、execution模式之间切换。我们要说的是：“就按这个方式来，这是最好的。”用户无需操心，我们自动帮你选出最优模型。

目前我们底层用的是多家模型提供商的组合，但目标只有一个：为用户提供最佳体验。

企业内部为何存在分歧与怀疑?

> **主持人Jerod Santo：让我问你一个问题。你之前提到企业内部存在分歧。姑且用“信徒”和“怀疑者”来形容吧。也就是说，有看涨派和看跌派。有许多人和组织在向着这个方向努力。但也有很多人非常怀疑，原因各不相同。对于怀疑论者，你听到的论点是什么？也许你能为他们辩护，而不是简单地否定。企业内部的怀疑者对这些工具及其未来持怀疑态度的原因是什么？**

Thorsten Ball：我不太想刻意区分企业用户和其他人，我们就泛泛地聊一聊吧。去年，我在意大利参加了一个Rust会议，期间和一位资深工程师聊了起来。他在过去20年里做出过很多非常出色的技术贡献，显然是位非常厉害的程序员。

当时我正在研究Zed，我们聊到编辑器的功能，他问我：“你们有AI功能吗？”我说：“有的。”从他的语气我就能听出，他并不相信这东西有什么用。他接着问：“我可以把它关掉吗？”我说：“当然可以。”

然后我问他，“你之前用过AI吗？我挺好奇的。”他说，“我一两年前玩过一次，但它只给我一些垃圾。”我又问，“你当时用的是ChatGPT吗？”他说是“某个网站”。

这给我留下了深刻印象——完全没有兴趣，也没有一点好奇心。他的态度是：“我试过一次，不好用。”就彻底放弃了。

我觉得现在很多人已经对这些技术有些麻木了，就像有些人对加密货币或者区块链已经失去了兴趣一样。有一部分人已经完全不关心AI的发展了。每当他们听到“AI”这两个字，眼神就开始发散，注意力完全飘走了，根本不会去了解接下来发生了什么。我认为这是个大问题——不是他们认真了解过这些东西然后判断“不适合我”，而是他们根本没有意识到过去几年里这个领域已经发生了多么巨大的变化。这是我观察到的一个方面。

另一个有趣的现象是，有人用钟形曲线（bell curve）来比喻当前AI在工程师群体中的接受度：初级工程师在这波AI热潮中获益最多，因为他们原本掌握的知识还不多，AI帮他们处理了很多繁琐的事情——比如，他们从来不需要自己去学怎么用CSS实现元素居中。

然后，在钟形曲线的另一端是资深工程师，他们也从中获益良多，因为他们知道很多，基本上知道如何审查AI在做什么，他们知道哪里有陷阱，以及应该写什么测试，不应该做什么，以及如何设计这个东西的架构。

但在钟形曲线的中间，有一类人处境就比较尴尬了：他们正在努力提升自己的技能，想学会所有这些东西，因此对“AI接管方向盘”这类行为感到不安。他们会说：“我不知道这里发生了什么。我希望搞清楚。我不理解这个过程。”他们会有质疑。

这其实正是很多企业中员工的真实写照。

第三个问题是：**你必须真正去学习和提升自己。你必须在这方面变得更擅长。**

很多人在第一次尝试使用Agent（智能代理）时，可能不会立刻获得惊艳的结果。在小规模场景下，有些人确实做出了一些令人惊叹的事情，但也有不少人会失败。

我认为这就是一个现实问题——过去几年间的炒作和市场营销把这件事吹得太厉害了，比如“你只需要说一句‘给我建个网站’，它就能生成一个看起来很棒的网站。”大家的期望值被拉得太高了。

所以现在有些工程师在实际尝试时，会说：“修复这个分布式数据库。”结果发现，它做不到，它根本不知道该怎么修复。于是他们就会想：“哦，原来不行啊。”第一次尝试不成功，他们就会感到失望，随后就直接放弃了。

最近，我想是在上周，Mitchell Hashimoto说，“你什么时候只用了一天工具就变得更有生产力了？”你必须投入一些努力。你必须在这方面变得更好。我认为这是一个问题。这些模型被拟人化了，我也有责任。

还有一种常见的思维方式是：“哇，这些东西好聪明，简直像人类，已经接近AGI了。”但现实是，这些系统本质上还是基于Transformer架构的大型语言模型。

它们有自己特定的处理机制：根据你提供的上下文生成输出。但它们并不是无所不能的，也不是全知的。你必须清楚地知道，哪些信息该放进上下文，哪些不该放进去——否则，它们很容易跑偏，做出错误的判断或回应。

所以有一个学习曲线，但不是像网上有人告诉你的那样。没有人说，“我们有一个

学习曲线。这太棒了。”在过去的20年里，软件领域的说法一直是“学习曲线不好”，比如Vim的用户。

放弃Vim转投VS Code的怀抱

> **主持人Jerod Santo：你现在还在用Vim吗？我听说IDE要消亡了。我的意思是……你不是已经让Amp来接管了吗？**

Thorsten Ball：是的。这是个热门话题。

> **主持人Adam Stacoviak：所以，到底还用不用Vim？直接说，是或不是？**

Thorsten Ball：不用了。我现在在VS Code里用Vim模式。但说实话，我已经不怎么亲自写代码了，这真的挺疯狂的。

> **主持人Adam Stacoviak：麻烦再说一遍。讲得更清楚些。**

Thorsten Ball：好的。在过去一年半的时间里，我在Sourcegraph工作时，开始想尝试一些新东西。我当时的目标是尽可能“硬核”，成为那种特别厉害的程序员。

后来我去了Zed，我们从零开始用Rust构建了一款文本编辑器，连GPU框架都是我们自己做的。团队里全是全球顶尖的程序员，代码库非常惊艳，产品本身也很出色。我当时觉得自己已经触碰到了编程的“核心”。

但随着AI的发展，我开始尝试一些新工具，比如CursorTab。有一次我坐在那儿，正好我们在为Zed做补全功能，我要搞清楚竞争对手都在做什么，比如制表符补全、AI补全等。我用了CursorTab，正改一个switch语句，做一些重复性的工作。以前我会花心思写一个漂亮的Vim宏来解决。而现在，我会在switch语句的某个分支中输入控制台日志或类似的东西，然后它会说，“哦，你也想在这里添加这个吗？”我只需按下Tab键。

我连续按了10次Tab，整个改动就搞定了。当时我坐在那儿，心里想：“靠，这比我用Vim自己写快多了。”

比如你要处理一个CSV文件，要删除最后一列。我以前的做法是在Vim里用普通模式跳到结尾，删除，或者写个宏，然后执行990次。现在用这些模型，我只要删掉第一列，

它就会问：“你是想删所有列？还是最后一列？”我一按Tab，就解决了。太疯狂了。这真的改变了很多事情。

然后我也开发Zed补全功能，我们自己完成了构建，并意识到“我不是ML专家”。和我一起工作的Antonio可能是我合作过的最好的程序员，但他也不是ML专家。但我们能够构建出与CursorTab同等质量的东西。我就想，“这真的要改变很多东西。”像开源模型，你可以使用Qwen，或DeepSeek，或其他任何东西。如果你能将这个变成一个补全模型，编辑代码的速度会比一个非常擅长Vim的人（我自己）还要快。那这说明什么？这说明开发工具必须变革，整个编程体验也要改变。

那一刻，我突然有个想法：我是不是在给马车做轮子？但我也不知道怎么说才不冒犯人。

> **主持人Jerod Santo：但Vim确实是一个非常好的文本编辑器。它真的很棒。**

Thorsten Ball：是的，它是一个令人惊叹的产品，但问题是，在Zed，我曾与Conrad一起使用过Vim模式工作，所有功能都令人惊叹，但最终你还是会想，“我要变得高效。”我不是那种因为宏和键盘快捷键而热爱编程并追求速度的人。我喜欢高效地完成事情。比如，我喜欢提高效率。

突然间，我意识到，我之前在Vim中使用的所有宏都有些过时了，因为在其他编辑器中，我只需连续按下Tab键就能轻松处理这些繁琐的任务。这改变了很多事情。这改变了我对开发工具的看法。

总结一下，基本上，我看了其他公司，与不同的人进行了交流，最终我回到了Sourcegraph，因为我与Quinn进行了交谈，并向他讲述了刚才告诉你们的一切，我对他说，“老兄，一切都在改变。”他回应道，“你想来这里共同构建未来吗？我同意你的观点，很多事情都在发生改变。”

现在，我在用VS Code，这是我从未想过的事情，我也不喜欢VS Code。从审美角度来说，我不喜欢它。我也意识到我不再那么在乎了。我和很多人进行了交谈，包括Sourcegraph的同事和我在旧金山遇到的人，或者在会议上遇到的人，他们也使用Amp或者Cursor、Windsurf。至少有五个人跟我说，“我曾经是一个铁杆的Vim用户，但我转而

使用VS Code/Cursor/其他东西，因为我意识到，它能把效率提升10倍，其他东西就不再那么重要了。”

如果你问Primeagen，或者其他人，他们肯定会因为我这么说而骂我。**但我有种感觉，很多人也有这种感觉，那就是在编辑器中快速机械运动的时代有点要结束了，因为这些模型更快。**未来的话，这些东西会变得更快、更便宜，也许肯定会在你的笔记本电脑上运行……然后，你不再需要Vim键绑定，不再需要Colemak，你只要和你的电脑说话。

Vim和Zed是过时的马车？！

> **主持人Adam Stacoviak：好的，你刚刚提到了“马车”的比喻，我想回到那个点上。先说清楚，我完全没有冒犯的意思，我只是想认真思考这个问题。作为一名播客主持人，我试图认真倾听，也在思考我们正走向何方，试图理解这个比喻的深意。**

但你真正让我思考的是：“我今天到底该怎么做？”很自然地，我会选择最简单、最方便的方案，因为那已经极大地改善了我的生活。

最直观的例子是什么？交通工具。比如今天早上，我送孩子去夏令营。我没走路去，也没坐马车——我们开的是那辆“新马车”F-250，一辆柴油卡车。虽然它可能还不算最新最现代的技术，但它确实比马车快太多了。

我们开车往返，从A点到B点。过程中我完全没有因为我的曾祖父可能是骑马出行就感到难过。那是他们的时代，他们喜欢那样，但那个时代已经过去了，对吧？现在没人那样出行了。我就是开车，就这么简单。

你让我意识到，这其实就是一个最显而易见的选择——一旦体验过现代化的便利，我们很难再回到过去的方式。除非有一天我退休了，在牧场上无所事事，有闲钱买马，骑马消磨时间，那可能另当别论。但如果说哪个方案显然更高效，那肯定是更快的交通工具，这就是现实世界的趋势。

Thorsten Ball：我觉得你说得非常好。而且这里其实还牵涉到一个“代沟”的问题。我记得去年在慕尼黑某所大学做了一次演讲，来听的很多人年纪都比我小不少。我们聊到了AI，聊到这代技术对编程的影响。

很多人提出疑问，比如：“这还算是‘真正的编程’吗？”“如果我用了Cursor、Cody、Windsurf这些工具，是不是就不算自己写代码了？”还有人担心：“我是不是会变笨？用这种方式学习对吗？”或者更具体地说：“如果不是我亲手敲出来的，那还算‘手工代码’吗？”

我和一群年轻人聊了一阵，突然意识到：他们根本不在乎这些争论。对他们来说，这些讨论早就过时了。他们从来不会想：“我的代码不算真正的代码，因为我没用Emacs写。”他们更关心的是怎么更高效地完成工作。

他们会告诉我：“我用这种方式整理文档，方便我用Cursor快速调用需要的东西，我还有个规则库，这样组织更轻松。”他们甚至不会花两秒钟去思考“这样做对不对”，因为这就是他们自然的工作方式。他们是与AI一起长大的一代人。如果编辑器不支持某功能，他们就去问ChatGPT、Claude，或者别的模型。

就像1965年时有人质疑：“用电吉他还算真正的音乐吗？”——结果证明，年轻人完全不在乎，他们根本不想听这种争论。他们已经在看向未来了。

所以正如你说的，这是一个代际变化的问题。如果你现在去任何一所大学，找一个正在学计算机或在业余时间写代码的20岁年轻人，我敢保证他们正在使用AI，而且丝毫不会去纠结“这是不是正统的编程方式”。**对他们来说，这就是他们的工具，这就是当代的编程。**

主持人Adam Stacoviak：尽管我们想留在过去，但未来无论如何都会来。时间是线性的，我们无法停止。我们只拥有当下这一刻。过去已经过去，我们无法改变，因为我们能做的只有把握当下这一刻，而未来无论如何都会来。如果写代码或软件开发者不再使用Vim了？如果甚至不再使用VS Code了？如果使用的是这些工具演变而来的版本？

Thorsten Ball：是的，我100%同意。

主持人Jerod Santo：你知道吗？如果你是个Vim的铁杆粉丝，那听到这些话可能真的会有点刺耳（笑）。虽然我在开玩笑，但说到底，这种“认同感”本身其实就是一种负担。我们真正需要改变的，也许正是这种认同。

就像有些人对自己青春期听的音乐，或成长阶段喜欢的乐队特别有感情，对当下的

音乐却提不起兴趣——那是因为它承载了我们的身份和情感。而我们使用电脑做的许多事情，也同样映射了我们是谁、我们在意什么。我们会通过自己选择的工具、编辑器来表达这些认同。

所以这就是为什么会有那么多关于编辑器的激烈争论。归根结底，是因为我们在乎。如果我们根本不在乎，也不会花时间去争论。正因为我们关心，所以当你把Vim或Zed比作“马车”时，我们自然会有反应。即使你没有冒犯的意思，那也确实可能让一些人觉得不舒服。

除此之外，我觉得还有一种更理性的怀疑，它源于很多人经历过太多“新潮技术”的起起落落。要不是因为我做播客，经常从这个视角来观察这些趋势，我可能早就放弃追踪AI相关的东西了。如果你有时间翻翻过去的节目，会看到我的态度其实也在变化。一开始我对这些工具的体验很差，我的反应是：“这有什么用？纯粹是干扰。我还是继续手动写代码吧。”但随着时间推移，我持续关注它们的发展，最近才真正意识到：“哇，这太厉害了。”我开始真正被它震撼。

不过，我也很容易回到过去的那种思维模式，因为我见过太多曾被吹捧为“有前途”“颠覆性”的东西，最后都没能兑现承诺。所以，这种怀疑也不是完全没有道理的。

我想回到你刚才说的关于“阻力”的话题——那些还没有完全接受这波浪潮的人，他们的心态，其实我也能理解。

然后，你提到了钟形曲线。人们称之为中智梗，就是中间的那个人自以为最聪明，看法却最糟糕。就像初级的人因为不同的原因理解了，高级的人也理解了，而中级的人却不理解。我认为，在这种特定的情况下，这是因为技能的问题。比如说，Vim是一种技能。你花了很多精力去学习。也许对你来说，它很容易，但对大多数人来说并不容易。**因此，这里存在一些沉没成本谬误，“我已经为这些技能付出了很多努力。”**

如果你看那个钟形曲线，初级没有任何技能，所以他们不在乎。他们会说，“酷，这帮我做了我做不到的事情。”至于高级，如果他们始终保持好奇心并具备良好的自我认知，他们就会意识到这些技能只是实现目标的手段，而真正重要的是目标本身。有了这个工具，他们会更强大，可以更好、更快地达到目的。我花了很多时间学习Vim。然而，在我认为对我来说它不再是最好的选择时，我会把它放在一边。

但是，当你处于那个钟形曲线的顶峰时，你已经花费了大量的时间、金钱和努力来尽可能地提升你的工程技能。所以对你来说，说出“这些技能实际上不再那么有用，不再那么有价值”是最难的。我认为，对我们很多人来说，这只是一种伤害。

Thorsten Ball：我认为，这里面有很多身份认同的问题。就像你说的那样，人们认同，“我是那个从来不需要查找Rust方法的人。我是那个知道所有语法的人。我是那个非常擅长Vim的人。”再次以你提到的高级工程师的为例。我想你可以确认这一点，作为一名高级工程师，有时候你会意识到，代码并不是那么重要。比如，市场营销、商业、团队以及你如何发布产品也很重要。

主持人Jerod Santo：对。除了代码之外，还有其他很多重要的东西。

Thorsten Ball：代码当然重要，但你最终会意识到，它有时候甚至可能成为一种负担。我觉得在工具这件事上，人们的经历往往也会经历类似的变化。

我一开始就有过这样的体验。我当时非常擅长使用Vim，为此感到特别自豪。我曾经坚信：“每个真正优秀的程序员都应该用Vim。”

后来我有一位资深同事，他用的是Sublime，几乎没怎么配置快捷键。但他的速度依然非常快，做了很多出色的工作，也总是能做出明智的决策。他之所以成为高级工程师，是有充分理由的。那时候我才意识到，也许决定水平高低的，并不是你用的编辑器。

我觉得这种情况在我们行业里经常出现。现在很多人把AI当成一把大锤，用它来击碎旧有的认知，说：“一切都变了。你曾经花大量时间投入的东西，现在价值没那么高了。”从某种程度上来说，这确实是真的，也确实很残酷。

面对这种变化，我们应当多一份理解和同情。我自己也经历过这种转变，并且为此挣扎了很长一段时间。

代码价值改变了，但程序员仍处于有利地位

> **主持人Jerod Santo：是的。我认为我们必须认识到，为了接受这个残酷的现实，同时也要克服它并真正地利用它，并不是我们的技能没用了，而是它们失去了价值。但是，由于我们作为工程师的经历，我们有很好的机会比其他人更好地利用**

> 新工具，并且能更快地适应，当事情出错时理解出了什么问题。我们还能帮助Agent做得更好，而那是新手做不到的。即使它已经很好，你只需要告诉它你想要什么，它就会变得更好。我觉得，高技能的人也可以采用新工具，只要他们没有我们正在讨论的包袱，而且与那些根本不会使用工具的人相比，他们可能可以更有效地采用新工具。

Thorsten Ball：是的。Kent Beck说过一句很棒的话，“我刚刚意识到，作为程序员，我能做的事情中有90%现在都一文不值，而另外10%的价值增加了100倍。”

有很多机械性的工作，比如，“配置哪个框架，如何配置，什么东西放入哪个配置文件，如何输入这个，如何做那个？如何构建一个FFmpeg命令？命令行参数是什么？”，所有这些东西现在都一文不值。

但是，要构建什么，何时构建，以及如何组织它，如何设计它的架构，拉入哪些依赖项，避免哪些陷阱，如何为将来的使用构建它，所有这些都是关于权衡的，是关于在一组约束条件下如何构建某物的决策，这些现在就非常有价值。现在，那才是倍增器，而不是你打字有多快。

主持人Jerod Santo：百分百正确。我今天早上刚跟孩子们说了这件事，作为父母和老师，我们也在努力摸索如何应对这些新事物。目前学校体系正经历巨大的变革。外

面一团糟。有很多作弊行为太好了，以至于作弊检测工具都跟不上了。我对孩子们说的，我认为特别适用于我们的工作，两者有很大的区别，那就是使用AI帮助你思考，和让AI为你思考之间的区别。如果你让它为你思考，那么我们正朝着白痴乌托邦的方向发展，你将无法成功。但是，如果你用它来帮助你思考，那么现在，你基本上就是一个超人。

我认为，当涉及到编码时，情况非常相似。我们确实需要参与并做出那些决策以及评判结果，做所有对我们、我们的环境、我们的商业目标以及我们知道的事情而言都独一无二的事情，因为编码代理只需你告诉它做什么，它就会尽力去完成；它可能会比你做得更好，但它不能决定做什么。我们还没有走得那么远。

因此，利用这些工具帮助我们比以往任何时候都更好地构建，而不仅仅是为我们自己构建，而我们只是随波逐流。

主持人Adam Stacoviak：我刚才只是在反驳这个观点，也许是针对那些有Vim纹身的Vim用户。你今天仍然可以通过SSH连接到一台机器。但这种方式并不常见了。通常，你会使用CLI来做到这一点，而在某个时候，你会有一个代理使用CLI来做到这一点。或者，也许今天你就应该这样做。不过，SSH仍然存在，你仍然有用户名和登录，你仍然可以控制你如何访问Linux机器。这并不意味着你不能再使用Vim了。这并不意味着这些技能就过时了。你还是可以维护和管理你的Vim文件。

这些仍然是事实，但那是一个不同的世界，那里曾经是Ultra X程序员的生产力工具，而现在，在某种程度上，那种类型的程序员已经停滞不前，因为Agent可以比它运行得更快。或者像照看孩子一样照看Agent。

这并不意味着Vim不存在，或者SSH不存在。你仍然可以通过SSH连接到一台机器。只是现在由Kubernetes来管理你的机器集群，而不是你手动逐一连接并配置每台机器。这与当前的做法截然不同。SSH仍然存在，Vim也仍然存在，只是使用方式发生了变化。

Thorsten Ball：我觉得你刚才讲的内容其实也触及到了另一个问题，那就是在关于“编程是否会被人工智能取代”的讨论中，很多事情都被笼统地归为“编程”这个概念。但事实上，世界上存在成千上万种不同类型的程序员。有的是在大型科技公司里开发分布式系统的，有的是在硬件公司做嵌入式系统开发的，有像我这样专注于开发工具的，也有从事Web开发或是构建AI Agent的程序员。

我住在德国的一个小镇上。如果我去拜访离我最近的一百位程序员，大多数人其实并不在软件公司工作，而是在一些传统行业里维护旧的Java程序。还有些人是在给客户做WordPress网站。

所以当我们说AI会改变很多编程工作时，有人会反驳说：“AI连Postgres的存储层都改不了。”但我并不是这个意思。我的意思是，每天可能有上万次，有人打电话说，“你能不能帮我改下我们WordPress网站上的这个东西？”而那个去做改动的人——可能被称作“程序员”——确实需要具备一定的技能。但我认为未来这类工作将会发生很大变化。也许不会在几个月内，但在稍远的将来，这些边缘的编程工作会逐渐被改变。

再说回你提到的“通过SSH登录服务器”这类观点。云计算刚兴起时，很多人也说，“云其实就是另一台远程计算机，本质没什么变化。我们还是会需要系统管理员来运维。”确实，到了2025年，我们仍然有系统管理员，但这个角色的边界和方式已经和以前非常不同了。

主持人Adam Stacoviak：是的，他们只是有些领域毕业了。

Thorsten Ball：完全正确。如果你看一下招聘系统管理员的工作职位的数量，我敢肯定，过去15年里已经发生了变化。我认为，其中一些变化也会发生在编程上。我认为，未来几年里，前端开发者的数量会有变化。但如果你在谷歌开发分布式存储层，那么不会有Agent在接下来的两年内取代你的工作。大概率不会。

主持人Adam Stacoviak：是的，如果你是在这些真正比较边缘的领域工作，可能会更安全。

我认为，如果你处于80%的领域，即80%的开发是由一个普通的、很容易招聘到的典型开发者完成，那么这份工作更有可能被压缩或者代理化，而你变成一个保姆，你有品味、审美以及人文倾向，如关怀和人性。你知道，到目前为止，机器可能可以推理它，但并不能真正像我们那样感受它。

这确实是一个值得深思的问题。我也在思考你正在编写的代码的类型。你有没有使用Amp来增强？你能给我们更清晰地描绘一下幕后的情况吗？因为我问过你这个问题，而你似乎也提到了，基本上，你描述了你所做的事情，而不是它实际看起来的样子。写

这种级别的代码是什么样子。我希望你描述下你的工作，不是关于Amp的研发以及你将如何发展这个平台，而是你作为一个训练有素的专业软件开发者的工作。你如今不是在写代码，而是尽可能多地生成代码，解决尽可能多的大问题。对你正在做的事情而言，这种描述准确吗？

Thorsten Ball：好的，我想我的头衔仍然是软件工程师。

我认为，作为高级工程师，我的工作是思考“如何实现？”在过去几个月里，我一直在想的是，“Agent能为我编写这个代码吗？它能做这个吗？”

如果问题过于复杂，或者我脑海中存在大量无法用文字表达的隐性知识，又或者将这些内容写下来过于繁琐，我会采用“填色式编程”的方法，直接输入代码行。我会说：“我需要这个文件、这个文件以及这个新服务，它应该包含这些方法，并实现这些功能和参数。请为我编写这段代码。”

在实际开发层面，我认为我们代码库中大概有50%，甚至60%到70%的代码，是标准的TypeScript、Svelte和SvelteKit代码。也就是说，整个代码库看上去非常常规，但其中很多部分其实是自动生成的。而真正承载主要功能、承担复杂逻辑的部分，通常是手写的，或者是基于生成代码进行过手工修改的。

我猜我们的测试套件中，大概有90%是自动生成的，比如只需要一句“覆盖所有这些情况”。我们还有一个现在已经相当完善的Storybook，它就是一个Web服务，用来展示我们所有的UI组件，包括Svelte组件，还能显示它们在不同配置下的表现。

比如某个组件是一个“箭头二号”，你可以看到它当前的状态，比如是激活还是未激活。在一个页面上，就能预览这个组件在所有状态下的样子。再比如一个活动指示器，它可能是蓝色、红色、绿色，或者处于空闲状态，并带有一个工具提示。开发过程中，你当然会想知道，“这个组件在各种状态下具体长什么样？”

搁在以前，我要做的是创建storybook页面，创建一个版本，创建一些模拟数据，做一些Vim操作，将模拟数据复制到不同的状态和不同的配置中，然后渲染它。或者换个说法，你有一个测试套件，有一些模拟数据，比如“五个不同的用户。一个用户被停用，一个用户被激活，一个用户是管理员，一个用户是组管理员”。以前，你会用编辑器多

次复制这些信息，或者编写其他辅助脚本来消除代码重复。但现在，我用代理做测试或者storybook。像这样，“这是组件，这是我想渲染的位置”。或者，“这是我想做的测试。去为我打出这段代码。”大部分代理都不是超级智能；它不会提出惊人的新算法，它只是干家务活。

还有另一件事。人们一直在说，我没有意识到我在编程时做了多少无意义的输入。我们都听过这个说法，“思考是瓶颈，打字速度不重要。”但当你真正去做的时候，就会发现，打字仍然占了很大一部分，而这就是我试图消除的部分。我不是试图消除思考的部分。我知道结构是什么，代理帮我写出剩下的部分。

我给你一个两小时前的具体例子。我在UI中有两个组件，它们应该做同样的事情。其中一个有两个按钮，另一个有两个按钮中的一个。我们称这两个按钮为登录和注销按钮。一个组件只有登录，另一个有登录和注销。这是一个愚蠢的例子，讲不通。它们的按钮应该是一样的，都有两个按钮。“如果要做到这点，那么我必须从这里复制代码，更新导入语句，调整它，确保它是Flexbox对齐的，并且渲染正确，然后调整这个按钮的高度……”等等。

所以我就跟Agent说，“这个组件有一个按钮，这个组件有两个按钮。第一个也应该有两个按钮。请帮我实现。”它看着两个组件，很快就明白了，这个任务不难。它复制过去，20秒后就完成了。我不需要自己打出来。

显然，这只是一个很小的例子。今天早些时候，我想建一个测试脚本，我有一些数据，我想逐个处理这些数据。对于每一条数据，我想运行它，发送到API并看看返回什么。我注意到，我的反馈循环是“启动开发构建，手动尝试，点击按钮，执行操作。”我想，“我可以构建一个工具，连续做50次，然后可以在同一页面上看到所有结果。”

半年前，我绝不会尝试这样做，因为我不想构建另一个东西，我得打出300行代码，我需要想出怎么做一个三栏布局。但后来我就想，“我可以生成这个。”如果我和代理说，“这个文件夹里是数据，这里是API。为我提供输入API密钥的能力，然后给我一个网站，三栏布局，左边列出数据，点击一行，显示数据，然后你有一个按钮发送请求，然后就可以在第三个面板上看到结果。”我说完它就去做了。我不在乎它看起来怎么样，它有没有样式。它很有用，这就够了。

我再讲一个非常具体的例子，几个月前的。那是在Amp早期，非常早，还不是在生产环境中。

我们有的代理在编辑文件时会失败。所以用户会说，“它有时会失败。”而我会说，“这帮不了我。我需要数据。比如，输入是什么？磁盘上的实际文件是什么？发生了什么？”为了弄清楚出了什么问题，在过去，我会做的是找出一些日志，比如一些错误报告。借助Sentry或其他什么工具把它们变成我能读懂的信息，从而找出问题在哪里。

但随后我想到，“我现在手头有一个代码生成器，一个可以非常快地生成代码的机器。如果那个代码是一个独立的项目，不到一千行，那么99%的情况下它都能搞定。”所以我做的是在代码中放了类似这样的东西，“如果你在Thorsten的机器上遇到这个错误，把原始数据转储。拿走原始数据，把它放在Thorsten的计算机上的这个文件夹里。”我只运行了两天，就收集了一千个文件。

然后我说，“Amp，我们来构建点东西。这里有一个数据文件夹。我想让你构建的是一个数据查看器。我想让你用Go语言构建一个小的Web应用，列出所有这些文件，取这两个字段，语法高亮显示它们，并显示这两个字段之间的差异。然后给我一个键盘控制，让我可以浏览数据。”它用了不到45秒钟就完成了。我打开网站，点击，浏览。我浏览了50个，通过查看数据，我发现了一个Bug。我意识到，“这是一个空格问题。”因为有语法高亮和差异，所以我可以轻松地浏览这些数据。

我永远不会自己建这个，因为语法高亮非常令人讨厌。想到要处理JavaScript中的diff、三栏布局，我会放弃的。但是使用代理或者一般的AI，入门门槛降得如此之低，你可以构建你以前从未尝试过的东西。回到我最初会尝试的东西，比如日志等。假如你得到了这样的日志，“输入这个，输出这个”，只有这一行日志，你会从中发现一个空格问题吗？比如，“它用了两个空格而不是一个Tab”。我不认为我能。但是，只说句话就能生成代码，这改变了我从工程角度处理这个问题的方式。

我认为，现在很多人都已经意识到这一点。昨天推特上有人说，他想知道“Markdown文档的每一节中有多少个词”。过去你会怎么做？你会坐下来手动编写一个工具来完成这件事吗？可能不会。你立刻就会放弃这个想法。他对代理说，“Claude，给我写个单文件脚本，不管什么脚本，只要能完成这件事就行”。结果它做到了，而这仅仅是因为现在负担得起了。

那么现在，回到真正的软件工程。有多少测试、调试工具、测试套件、自省工具或分析工具是因为太费力了才没有编写？现在这些都变得负担得起了。我们开始意识到这一点，问题是，我们何时才能真正利用这些？我们何时会利用这些？

再进一步说，自2019年以来，我一直在Sourcegraph工作，中间休息了一年。很多大客户会问，“你能让我们的代码库用上这个吗？你能把你们的工具运用到我们的大型代码库上吗？”现在我们看到，人们越来越多地利用AI修改代码库。他们会说，“对于代理来说，这些文件太长了，上下文窗口放不下。我们把它分成五个文件吧。”我告诉你，两年前没有人会为了任何工具分割他们的文件。他们会说，“这就是我们的文件。”

这就像我们有20000行代码，没有人会碰。但现在情况发生了变化，这些工具能提供的杠杆也发生了变化。现在，代码库将随之适应。这是我的预测。**代码库将适应这些工具。**

对我来说，真正有趣的是，我们的工程实践会会有什么变化？我们会手写什么代码？我们会生成什么代码？再进一步想，会不会有我们不提交的代码，我们只提交提示，然后即时生成它？还是所有的代码仍然会被提交，谁知道呢？

开源的价值也变了？

> **主持人Jerod Santo：这实际上为我打开了一个全新的思路，我以前没想过的。这对开源有什么影响？**

因为我在想你说的情况，“在过去，那种一次性工具帮助我做别的事情”，就像是一个支线任务。就像你说的，“工作量太大。有它很好，但我不需要它。”或者我会说，“算了。我要写一个，然后开源，其他人也可以用它，对我来说就值了。”或者我会说，“好吧，让我们去看看别人有没有做过，看看我是否可以只是使用别人的。”也许顺序不一样。我可能会先去看看，然后再决定是否要构建它。

但在一个我们可以临时生成一次性工具的世界里，我们可以把它们检入代码库，也可以不检入，可以保留提示，也可以不保留，开源的数量会减少吗？开源工具还重要吗，因为我可以生成我需要的任何东西？你想过这个吗？因为我甚至没想过对开源的影响。

Thorsten Ball：让我们说实话，GitHub的贡献不像十年前、五年前、两年前那么

有价值了。我认为，在过去一年左右的时间里，它的贡献有一个急剧的下降。现在我在想，有了AI，我为什么要还引入一个小包？为什么我要亲手编写它？为什么我要去某个地方查找一个格式化时间戳的函数？我可以直接问LLM，“这是时间戳，这是所有的五种格式。如果你没有所有五种格式，这是命令。给我写一个程序，生成所有可能的格式。所以你看到了所有可能的格式，然后给我写一个函数来解析它们。”即使是代码作为一种减少重复的方式也不再是理所当然。

> 主持人Jerod Santo：你开始质疑了。

Thorsten Ball：我举了个愚蠢的例子，有人会说“Thorsten是个白痴”。但为了说明这个点，假设你有一个函数来验证某件事，你想确保它验证了50个案例或者150个案例。过去，最好的做法是打出这150个案例。你会写一个正则表达式或者别的什么，对吧？你会说，“不要用正则表达式，因为我们无法维护这个列表。”现在，有了LLM，你可以生成150个例子。你可以添加代码编写时间，生成所有的变体。您无需让CPU处理所有变体。Web框架的目标是帮助你减少需要编写的代码量。但如果生成的代码量虽然庞大，但速度快、成本低，那么当我可以直接说，“将所有用户的头像组件改为绿色”时，我是否还需要复杂的模板辅助工具？

> 主持人Jerod Santo：DRY原则变成了“Do Repeat Yourself（重复你自己）”。

Thorsten Ball：也许吧。很多代码以及我们编写代码的方式都是基于这样的假设：编写代码需要时间，而且难度大，我们要尽可能避免那样做。但现在，情况变了，因为我可以一键生成网站，还可以有50种不同的变体。

让我们回到开源那个问题。存储预生成的代码片段，构建可供15种其他用例使用和配置的库，这是否还有意义？你会把一个版本输入到一个LLM（大型语言模型）中，然后生成自己的版本吗？这有点科幻，显然它还没有实现，但事情正在改变。

还有一件事，更进一步，这也是我认为我所在的行业如此有趣的原因之一。

Facebook前工程总监Eric Meyer是一个Haskell爱好者，一个非常聪明的函数式程序员，已经编程40年了。两年前，他做了一次演讲。他说，“当你可以让一个LLM生成它，为什么还要搜索代码？”他的意思是，当你去搜索Stack Overflow，搜索你自己或公司的代

码库时，你想解决的问题是“构建一个用户头像组件或者类似的东西。”我们通常怎么做这个？因为你不知道或者你不想去做。但如果有一个模型知道怎么做，并且可以在不到一秒钟内完成，为什么还要去找那些例子？为什么不在这里生成这个？当你不需要去寻找它，而是可以即时生成时，事情就变了。

主持人Adam Stacoviak：嗯，你所说的是，我们过去所做的效率或感知效率都是基于人类的效率。

对吧？我们认为编写共享代码很高效，因为这样可以更高效地创建新项目。为此，团队需要统一编码方式、标准等。这些都是效率提升的措施。但它们不再重要了。对于大型语言模型（LLM）而言，或者说开始生成代码的系统来说，情况就像是这样，“我无需担心25个应用程序如何以一种统一的方式连接，因为我可以随时随地编写代码。”

Thorsten Ball：是的。我过去用过的一个比喻是，在书的最后，你有一个索引，里面有不同的单词，你可以快速查找。那个索引的存在是因为在书中找到那个特定的东西需要很长时间。如果你能以每秒1000页的速度阅读，你还需要索引吗？这个东西本身是与现在的使用方式相匹配的，但如果方式突然变了，为什么还需要在书的最后有一个索引？我认为，很多代码就是这样的。它有赖于我们编写和消费代码的方式，以及编写难度。但当能力改变时，工具或我们用这些工具生成的东西也会改变。

> **主持人Jerod Santo：孩子们会完全理解这一点。孩子们是AI原住民，他们不会问这些问题，因为他们将在一个没有这种限制的世界中长大。他们会说，“为什么你必须有一个共享库？当我可以告诉我的代理创建一个新库时，我为什么要共享我的库？当我可以重写，我为什么要重构？或者当我可以替换，我为什么要维护？”**
>
> **维修汽车是因为替换很贵。但如果替换的成本接近零，为什么还要维修？**

Thorsten Ball：我说一个有趣的例子。我开始构建这个是因为我发表的博文引起了人们极大的兴趣。这有点哲学意味，但归根结底，我们在使用计算机时所做的大部分事情，都是将信息以特定的形式和结构呈现，以便计算机能够处理。例如，静态网站生成器博文现在的格式是，你需要一个YAML文件，称为前置信息；其中包含文章链接、发布日期等信息，然后是文章正文。现在，借助大型语言模型（LLM），理论上，你可以用

任何东西撰写博文。你可以创建一个名为“我的博文”的文件夹，其中可以包含一条短信的截图，这就可以作为一篇博文。

你也可以准备一个便签的截图、一张便签的照片、一个Markdown文件、一个文本文件，然后告诉LLM，“这是我的五篇博文，这是我希望这些博文使用的基本模板，为我生成博客。”你不需要在其中添加任何结构，因为这些LLM现在是“模糊到非模糊适配器”。我是这样称呼它们的。你可以向它们投喂图片、截图、语音消息、视频，任何内容，它们都能生成文本。这听起来像科幻和哲学。当我们不再需要严格遵循数据库的列结构进行思考时，我们能创造出多少美好的事物？我们可以说，“一篇博文可以是任何形式。它可以是一张图片、一段视频、一段音频”。我们现在有一款工具，可以将这些内容转换为另一种形式。我们无需将其固定为特定的格式。

> **主持人Jerod Santo：这很有趣。那你到底在构建什么呢？**

Thorsten Ball：我开始构建一个静态网站生成器，在构建时，它会浏览一个名为“Posts”的文件夹，并用图片、视频、音频文件和屏幕截图生成博文的索引，并将它们放入一个格式中。提示是这样的，“对于每篇博文，修改布局以匹配博文的内容”。比如，让它看起来严肃，或者让它看起来有趣，或者任何它应该成为的样子。我认为，这是我们在计算机或软件中从未有过的东西，你可以告诉它，“让这个看起来像手写的”。然后它就会出现，做一些让它看起来像这样的东西。

大概两个月前，有人给我发了一封电子邮件说，“你的个人网站仍然说你在Zed工作，但我听说你回到了Sourcegraph。”我回复说，“你是对的。”我用Amp打开我的网站，截取了那封电子邮件，并把它粘贴到代理里说，“修复这个。”然后它就找到了我的网站上说当前工作地点的那一部分，并根据那封电子邮件的屏幕截图进行了更新。我就想，“这不是很神奇吗？我截取一封电子邮件的屏幕截图，然后一些东西就会根据它改变？”我把它发回给给我发电子邮件的那个人，那人说，“我敢肯定，你可以比代理做得更快。”我说，“伙计，你不觉得这太疯狂了吗？”我可以为你构建一些东西，你转发一封电子邮件，它就会在你的网站上打开一个拉取请求。现在构建这个不难了。

> **主持人Adam Stacoviak：是的，要是以前，这是一家初创公司了，对吧？**

Thorsten Ball：没错。

主持人Adam Stacoviak：有人会带着一份商业计划，到处寻求资金。现在则是“随它去吧”。不过，我们之所以讨论到这里，主要是因为Jerod提到了开源。过去大约35分钟的讨论，全都是围绕开源这个问题展开的。起初，我差点说一切都是默认开源的。因为如果你能生成每一行代码，那么关键因素不是生成的代码，而是想法和知识产权，是这些决定了它是否具有专有性。如果默认情况下一切都是开源的，但随着对话的进行，“如果开源不再那么重要了怎么办？”因为当我们需要某样东西的时候，我们只需自己创建。不过，总得有一些在用的标准吧。

> **主持人Jerod Santo：源代码的价值在哪里？一定要有一些源代码，供机器人学习。**

Thorsten Ball：是的，那是一种次要影响。比如说，如果我们都认为分享东西不再有价值，那么用于改进这些模型的资源就会枯竭。

主持人Adam Stacoviak：我认为，尽管如此，我们仍然会从分享中找到价值。我认为，仍然会有库和框架被创建，也许在某个时候，源代码将成为大型语言模型（LLM）使用这些内容的一种标准化方式，它们将成为用户，就像我们是用户一样。而我们之所以成为用户，仅仅是因为我们关心与之关联的名称。

主持人Jerod Santo：但它们不一定想要源代码，它们想要的是工具。比如，在训练时它们需要源代码，但在实际构建中，它们更需要工具而不是源代码。所以我认为，我们无法在接下来的三到五年内回答这个问题。我觉得这是一个代际问题。比如，如果你现在展望20年后，“2045年开源技术对世界的影响会是什么？”，与现在相比会大不相同吗？我认为可能会，但我不确定。

Thorsten Ball：我能做一个乐观的预测吗？我认为，那些创造性的、人类独有的、有品味的，基于独有经验的东西，其价值将会上升。比如，如果有一件事，在那个时刻，只有你用这个模型和那个模型的组合才能完成，我认为那仍然是有价值的。除了真正有创意的代码、真正有洞见的算法、真正高效、良好的数据结构，另一个TUI框架、日期解析库或者类似的东西，其价值会逐渐减弱。独特性、品味和创造力的价值会脱颖而出。

主持人Jerod Santo：我觉得这是一个很好的结束语。Adam，你觉得呢？

主持人Adam Stacoviak：我也这么认为。我唯一想要补充的是，似乎需要换个角度来看待这个问题，因为当你离问题越近，细节就越重要。比如，未来的人类可能会说，“你还记得人类的价值曾经是基于代码行数，或是他们一生中写下的文字，或是其他什么吗？而现在，这些都不重要了，因为当你从宏观角度看问题时，这些并不是需要追踪的指标。”当你聚焦于细节时，这些细节确实重要。但当你从宏观角度看问题时，你衡量事物的标准是基于整体趋势，而非细节的具体定义。

原文链接：https://changelog.com/podcast/648

延伸阅读

一年成爆款，狂斩 49.1k Star、200 万下载：Cline 不是开源 Cursor，却更胜一筹？！

挑战Claude code和Cursor：阿里Qoder对标全球，AI编程迎来“上下文”革命

采访嘉宾 夏振华 编辑 Tina

AI编程工具正爆发式增长。以Claude Code、Cursor为代表的海外路线，凭借架构与交互创新快速出圈；Cline、Replit、AmpCode等也在加速试验新形态。与此同时，国内厂商积极入局，试图打造兼具本地特色与全球竞争力的AI Coding工具。

尽管过去一年Agent能力明显进步，开始从“辅助”走向能独立处理长链条任务，但“上下文”依然是摆在实用化面前的一块硬骨头。在实际开发中，面对庞大的代码规模、复杂的模块结构和分散的上下文，开发者常常需要投入大量时间进行检索、理解和修改。与此同时，文档与代码长期不同步、知识传递低效以及重复性编码任务占比高，成为限制研发效率的核心痛点。

正如Andrej Karpathy所言，LLM更像一种“新操作系统”：模型好比CPU，上下文窗口如同RAM，容量有限、取舍关键。“上下文工程”的任务，就是把正确的信息装进这块“工作内存”，为下一步推理服务。谁能把对的东西放进窗口，谁就更可能在真实工程里稳定交付。

正是为了直面这一“上下文”瓶颈，阿里推出了首个Agentic编程平台Qoder。它被定位为面向全球市场的“创新验证平台”，相当于探索前沿技术“先锋队”；阿里表示，Qoder可一次检索10万个代码文件，并将电商网站的前后端开发从数天压缩到约十分钟。它结合全球顶尖的大模型能力与Agent，对大规模代码进行深度语义检索与持续上下文理解，将复杂、多阶段的开发任务拆解并由智能体迭代完成，力图以“仓库级理解+任务化执行”正面应对真实工程的复杂度。

这条“重理解”的路线能否在全球对标中站稳？带着对架构哲学、技术取舍与定位/价格等关键问题的关注，我们采访了Qoder团队工程师夏振华，并围绕这三条主线展开深度解析。

采访嘉宾

- **夏振华**，Qoder团队工程师，负责基于LLM的Agentic Coding产品的设计与研发工作。之前在蚂蚁金服、阿里云有多年软件研发工具链的架构设计和研发经历。目前专注于AI Agent技术在软件开发领域的创新应用，致力于通过Context Engineering、Multi-Agent系统、Agentic RAG等前沿技术提升提升整体Coding Agent在开发者日常编程任务以及长程复杂任务上的体验和效果。

Qoder的定位与成本

> **InfoQ：在开发者圈子里，大家常常把AI编程工具放在一起比较，比如Cursor、Claude Code等。外界甚至把Qoder称作“阿里版Claude Code”“阿里版Cursor”。你们怎么看这种说法？如果已经有Cursor、Claude Code，为什么开发者或企业还需要Qoder？Qoder在这个“工具谱系”里应该被放在什么位置？它的独特价值和差异化优势在哪里？**

夏振华：Claude Code是一个非常优秀的产品，它在CLI环境中率先探索并开创了AI编

程的新方式，让命令行这一经典的程序员交互界面焕发了新的活力。

相比之下，Qoder并不仅限于CLI，而是一个完整的Agentic Coding平台，既提供IDE集成版本，也有CLI工具，覆盖不同的开发场景和偏好。我们的核心优势在于工程级感知与持久记忆，可以全面理解并索引整个代码仓库的结构与历史，不局限于单文件操作，而是支持跨文件、跨模块的深度语义检索、分析与改动。这种能力确保在复杂、多轮迭代的项目中保持上下文一致性和长期可靠性，对真实工程环境更为贴合。

在模型层面，Qoder支持模型的自动路由，不同任务由最适合的模型无缝切换执行，开发者无需手动选择或理解模型差异，从而减少认知负担并提升效率。同时，我们具备任务拆解与规划能力，能够将复杂的长程任务拆分为可执行的子任务，并进行有序调度与跟踪，实现更完整、高质量的任务交付。

这使Qoder在AI编程工具生态中，不只是代码生成或补全助手，而是面向工程全周期的智能研发伙伴，适用于从vibe Coding到日常编码，再到复杂系统构建的全链路场景。Qoder在8月21日全球首发后，目前已经收获了全球数十万开发者，验证了PMF的成功！

> **InfoQ：Claude Code和Cursor都遇到过推理成本和价格争议。从用户体验来看，Qoder的Credits消耗规律并没有完全公开透明。有开发者认为，根据看板数据和实际体验，Credits并不是单纯按调用次数计费，而更像是基于token消耗后做的一种归一化处理。能否具体介绍一下Qoder的Credits定价逻辑？在不同场景（如大文件检索、复杂任务、多代理并行）下，Credits的消耗方式有什么差异？**

夏振华：Qoder的Credits并不是按照简单的提问次数或模型调用次数来计费，而是基于实际token消耗进行统一换算和结算。这样能更精准地反映任务的真实计算量，同时避免用户在复杂任务或长上下文情况下被不公平计费。我们始终为订阅用户提供全球顶尖的编程模型，确保用户获得行业最强的上下文工程能力与推理效果。

另外，我们也持续通过技术升级，比如：工程检索准确率提升、智能体工具并行化优化、上下文智能压缩等能力，在保证效果的前提下持续优化单任务的token消耗，希望做到并能让开发者感受到我们的Credits越来越耐用。

Qoder在刚发布公测期，确实收到了用户关于Credits消耗快的反馈，我们也非常重

视大家的体验，通过持续的技术优化，当前最新版Qoder整体耐用度相比公测期提升了15%。

关于不同场景的Credits消耗，场景越复杂，所需处理的上下文越大，迭代步骤越长，token消耗越多，Credits使用量也随之增加。这里面有一些使用技巧和经验，后面我们会在Qoder社区开设对应的专栏，给大家提供参考指南。

> **InfoQ：最近有用户在Claude Code上用200美元的包月订阅消耗出了价值5万美元的token，逼得官方不得不连夜限速。这也让大家开始关注：Qoder在包月订阅的情况下，对于每月的token使用有没有限流或速率控制机制？你们如何平衡用户希望长时间运行代理与平台需要控制资源成本之间的矛盾？**

夏振华：Qoder的每个订阅版本都设定了固定额度的Credits，系统会根据任务的实际token消耗进行折算和扣减，额度用尽后将自动降级至基础模型，用户仍可继续使用，以满足持续运行的需求，同时有效避免单用户极端占用资源的情况。

在平衡用户长时间运行代理与平台资源成本方面，Qoder一方面通过技术优化不断降低模型的推理成本；另一方面也鼓励开发者在提交任务时明确表达需求，减少无效调用与重复推理，从而实现资源的高效利用与稳定可控的用户体验。

> **InfoQ：Qoder上线以来，用户增长的情况如何？主要的增长动力来自个人开发者，还是更多来自团队和企业用户？在你们看来，下一个阶段的用户增长会依赖于哪些关键因素？**

夏振华：Qoder上线以来，用户增长的速度非常快。虽然具体数据不便透露，但从活跃度和使用广度来看，在早期阶段，增长更多来自对新技术新产品充满兴趣的个人开发者，并且海外开发者居多，他们乐于在日常编码、学习和探索中尝试Qoder，并形成口碑传播。前段时间我们对部分订阅用户开展了一次调研发现，同事或朋友推荐使用Qoder占比超40%，这点让我们非常惊喜，口碑胜过一切。现在，随着用户规模的持续高速增长，以及工具在复杂工程场景中的能力被验证，团队和企业客户的比例也在不断提升，近期Qoder Team版也会上线。

下一个阶段的用户增长，核心还是在于持续提升产品力，构建更强的、差异化、引

领市场方向的核心能力，并切实解决开发者在真实场景中的实际问题。

面向全球的技术筹码：技术路线与差异化

> **InfoQ：Qoder在设计上最核心的哲学是什么？它是如何做到既降低新手上手门槛，又能满足百万行级项目的复杂需求？**

夏振华：我们的设计思路是面向真实软件的智能化开发，让AI能真正深入理解工程、参与创造，从而构建出更好的产物。真实软件意味着我们的目标不是做演示级的小功能，而是面向长期维护、迭代和交付的真实工程，无论是几百行的原型还是几十万行的企业代码库，Qoder都能融入整体架构，理解全局上下文并稳定输出。

Qoder的底层能力是高度内化的。它在后台拥有工程级感知、持久记忆、任务拆解等复杂机制，能在面对百万行级的大型代码仓库时自动“唤醒”这些能力，帮助开发者跨文件、跨模块地分析、检索、规划和交付复杂任务。

但对新手来说，他们不必理解这些复杂过程。Qoder会通过自然语言交互、即时反馈和简洁的界面，把这些高级能力隐藏在流畅的体验背后，让任何人都能轻松上手、快速获得有价值的结果。

> **InfoQ：市面上的AI编程工具越来越多，但不少开发者觉得它们在功能和体验上差异并不大，包括在上下文管理等关键能力上，大家的方案看起来类似，容易出现“同质化”。在你看来，如今团队在选择AI编程工具时，是否已经有了清晰的判断标准？**

夏振华：目前市场上的AI编程工具在功能层面确实有一定的相似性，比如代码补全、Agent模式开发任务等，因此表面看容易出现“同质化”的印象。然而，在实际落地中，这些能力的实现方式、可扩展性、与工程环境的融合深度，以及对不同规模和复杂度项目的适配效果，往往存在明显差异，这也正是团队在选择工具时需要重点考量的地方。最后，企业会在效果、性价比、安全合规这几大维度综合权衡。Qoder正是在这些关键维度上深度投入，为企业和开发者提供切实可依赖的智能研发伙伴。

Qoder除了基础的Agent模式之外，我们也发布了两个特色功能，Repo wiki、Quest模式，这两个功能也在开发者社区收到了非常多正向的反馈。Repo Wiki会自动生成项目知

识库，在大型仓库里找功能实现尤其实用。这个功能解决了技术文档滞后的老大难问题，让新团队成员能够快速上手项目。

Quest模式就更进一步，是Qoder强大之处。Quest意味着探索，是Qoder迈向自主编程的关键一步。主要针对复杂或耗时的开发任务，写出详细Spec后它会自己规划、撰写并给出报告，多个任务可以异步并行执行，我只需要审阅它的plan即可。Quest上线后2个月，近期Cursor也快速跟进、发布了Plan模式，从某种意义上，大家也看到了这是正确的方向。

> **InfoQ：在AI编程工具里，大家经常会讨论所谓的“外壳”（harness），也就是在代理或模型外层加的一些功能，比如提示词优化，那么你们是如何判断哪些能力应该直接内置在CLI里，作为Qoder的“默认体验”，而哪些能力应该交给使用者或外部工具自己去实现？**

夏振华：我们的取舍原则很简单：不纠结“套壳”之争，关键是让开发者在真实工程里高效、稳妥地把任务做完。默认内置的，是跨项目通用且与正确性强相关的能力：代码工程的深度理解（项目结构、依赖、构建与测试）、持续的任务级记忆与知识沉淀，以及精细化的上下文组织与“最小必要上下文”分发，确保代理始终在正确的约束与单一事实源下工作。

而企业外部系统与知识库的集成（私有规范、流程、审批、资产库等）、个性化能力与自定义策略则交给使用者或外部工具来实现。

Qoder CLI也是前几天全面上线。Qoder CLI在全球顶尖的编程模型基础之上，进行了大量的工程设计，全面提升Agent能力。Qoder CLI内置了轻量级的Agent框架，该架构可高效运行在普通笔记本电脑和云端沙箱实例，满足不同场景的开发需求。测试显示，Qoder CLI在空闲状态下消耗的内存比同类工具低70%，常见命令的响应时间不到200ms。

Qoder CLI还内置了Quest模式（自主编程）与CodeReview能力，无需开发者深度介入，就可以轻松实现Spec驱动的任务委派，让AI自主完成任务开发。同时在命令行终端即可进行代码审查，帮助用户快速扫描项目中的关键改动点，并给出审查意见，代码审查耗时减少50%，代码质量提升一倍。

> **InfoQ：Claude Code的产品负责人曾强调，他们刻意保持产品简单、通用，避免在模型外堆太多脚手架，因为模型能力进步很快，复杂结构反而可能成为束缚。那在Qoder的设计里，你们如何判断：哪些功能可以依赖模型本身的能力去完成，哪些则需要用工程手段补齐？**

夏振华：我们的判断原则是保持Agent架构尽量简单，避免复杂Workflow，把推理、反思与迭代尽量交给模型，以最大化模型升级带来的红利。

其他比如涉及工具调用的输出质量、上下文组织与切分、记忆与状态管理、容错与可观测性、以及外部数据与环境集成的可靠性与边界，则用工程手段补齐。

检索与上下文工程

> **InfoQ：长链式的代理任务往往会消耗大量token。Qoder在设计时是如何看待和优化这一问题的？你们会考虑哪些具体手段？**

夏振华：Qoder针对长链式代理任务，会先在Plan规划与执行阶段生成结构化方案，明确步骤与依赖，减少无序调用并防止偏离主线；依托 强大的工程检索能力，仅召回相关代码片段，显著地降低上下文占用；在工具调用上实现并行化，缩短链路并减少消耗；结合精细化上下文组合只引入必要信息，并通过裁剪压缩策略移除冗余数据、生成摘要，避免窗口占满，在保障任务质量的同时显著提升性价比。

> **InfoQ：今年大家都在说“代理之年”，但在实际落地中，复杂任务往往会触发几十甚至上百次工具调用，导致上下文爆炸：一方面很快打满窗口，另一方面还会出现“上下文腐烂”带来的性能退化。Qoder在设计时是如何解决这种“工具调用密集带来的上下文爆炸”问题的？在保证任务完成度的同时，怎么平衡窗口上限、性能退化与成本控制？**

夏振华：我们在有限的上下文长度下，通过Context Edit能力和长期记忆机制，保留任务主线所需的关键信息，同时清理无关或过期内容，避免窗口被无效历史填满；并结合工程级压缩与模型端摘要，将必要信息以更紧凑的形式保留，降低token占用的同时确保可用性。另外再实际工程落地时，还需要结合不同模型的prompt cache，决定压缩策略和触发时机，避免无谓的上下文调整带来的额外的成本开销。

> **InfoQ：你们提到过“通过技术升级与手工压缩上下文的配合，Credits耐用度将提升50%”。总结和压缩并不是一件简单的事，需要非常谨慎以避免信息丢失。Qoder在上下文压缩时，是如何保证意图不会被淡化、关键指令不会丢失的？**

夏振华：Qoder在进行上下文压缩时，会通过精细化总结与关键代码保留，确保在压缩后仍能维持任务主线和用户意图。我们会结构化提炼任务目标、技术概念、文件摘要、问题修复记录及待办列表，并结合近期代码改动，使模型在精简上下文的同时保持连续性和高性能，避免跑偏或丢失核心信息。需要注意的是，压缩本质上属于有损处理，还是可能会出现压缩后的效果变差，这方面要有一定的心理预期。

> **InfoQ：在代码检索上，有的团队选择“重型RAG管道”，比如Windsurf的分块、向量检索和重排；也有的像Claude Code一样走“代理式检索”，完全不建索引，Cline甚至直言“不能再用2023年的办法解决今天的问题”。Qoder在实践中更倾向哪条路线？你们怎么看待“什么时候该上索引，什么时候简单探查就够”的取舍标准？**

夏振华：在Qoder的实践中，我们更倾向于构建完整的工程检索引擎，而不是完全依赖代理式的grep检索。这样可以在源代码规模较大、文件分布复杂时，通过分块、向量检索与结果重排有效提升检索的精准度与召回率，从而减少多轮模型调用，提高整体效率并优化运行成本。

对于“什么时候该上索引，什么时候简单grep就够”，我们的取舍标准主要基于以下几点：

- **代码库规模与结构**：当代码库超过一定体量、文件结构层次深，且跨模块引用频繁时，建立索引能显著减少检索时间并提高相关性；而在小型项目或结构较为集中的场景，轻量的grep就足够。
- **成本控制**：索引建立有初始计算和存储开销，但在长期使用中可显著降低模型调用次数和API消耗；简单grep虽然零索引成本，但在重复场景中总调用费用更大。

总体而言，我们会在大型、复杂、高频的检索场景下优先用索引，而在小型或一次性探索任务中使用grep，这样既能保证性能，又能合理控制成本。

> **InfoQ：我们看到很多厂商都在引入缓存机制：比如OpenAI的Responses API会自动缓存对话历史，Anthropic以前需要显式header来启用，现在也自动化了，Gemini也支持隐式缓存。缓存确实能明显降低延迟和成本，它并不能解决长上下文带来的“上下文腐烂”和性能下降问题。那在Qoder的上下文工程实践里，你们是怎么理解缓存的价值和局限的？会不会把缓存和压缩、检索这些手段结合起来，用来优化整体体验？**

夏振华：在Qoder的上下文工程实践中，我们将缓存视为降低成本、提升性能的最核心的能力之一。它的价值主要体现为高命中率带来的延迟降低和推理费用优化，尤其对Agent场景这种高频、前序请求大量重复的情况，能显著减少模型计算开销。

但是长时间保留大量原始prompt虽然便于复用，但也容易导致上下文长度增加、出现“上下文腐烂”，影响输出质量，因此必须谨慎权衡。

为此，我们会将缓存与压缩、检索增强等策略结合使用：当判断平台缓存机制可能失效，或上下文接近模型上限时，会主动优化上下文结构，提取关键信息并替换冗余内容，从而保证命中率的同时减轻性能衰减，确保整体体验稳定、优质。

多Agent与单Agent

> **InfoQ：在业界对于多代理的看法并不一致。Cognition认为不要做子代理，因为子代理之间不能很好的沟通；而Anthropic则强调多代理的优势。Qoder在设计时，是如何看待这种分歧的？如果是多个代理，我们该如何处理这些代理之间数据、记忆和上下文的割裂问题？你们在选择最优解、减少合并冲突、降低开发者认知负荷等方面，是否有探索过新的机制？**

夏振华：模型能力的升级，很多观点都会发生变化，比如这里提到的Devin关于多Agent的态度，其实也在变化。Qoder在探索这方面的实践已经持续了一年多，从去年我们就开始探索多Agent串联配合来完成一个研发任务，通过任务拆解，解决当时模型能力不足的问题。再之后随着模型能力的演进，我们主要的架构是基于单Agent通过工具调用自主迭代循环的方式来完成任务。

当前，我们也在做包括多Agent的串联、主子Agent形式的探索，并解决这里面存在

的上下文隔离、共享等问题。整体目前仍在探索阶段，进展后续也会持续同步出来。核心思路是子Agent仅获取最小必要输入、以结构化关键摘要信息的方式回传，由主Agent聚合与决策，降低上下文割裂与主Agent负担。

> **InfoQ：社区里也常有人吐槽，Claude Code很难做到真正的“全自动长跑”，代理跑一会儿就要人工确认。Qoder会不会考虑支持这种24小时不间断的运行模式？如果支持，你们会如何在体验、安全性之间做取舍？**

夏振华：Qoder的Quest Mode就是为长时间运行的研发任务而设计，采用Spec驱动的开发范式，让Agent能够将用户的需求自动转化为详尽的设计规范，并在此基础上自主完成开发、测试、重构和Bug修复等工作，实现端到端的自动化研发。

依托Remote云端运行能力，Quest Mode可以在安全的云端沙箱环境中持续执行任务，无需依赖本地环境，用户可在执行过程中随时中断或调整任务，从而在保证长时间稳定运行的同时降低安全风险。

> **InfoQ：在很多团队尝试把代理迁移到云端时，都会遇到一个难题：如何复制开发者本地的环境？直接“克隆”开发环境往往过于复杂、个性化，很难真正落地。于是大家转向容器和沙箱，把代理跑在更标准化的环境里。Qoder在云端代理的设计上，是否也遇到过类似的挑战？是否有值得分享的“黑科技”？**

夏振华：是的，云端复制本地环境确实是一个难题。Qoder的Quest模式里我们是这样来解决这个问题：远程任务以用户自定义的Dockerfile作为基础环境，系统先验证/构建，再在每次任务执行前checkout对应commit并运行用户提供的安装/初始化脚本，保证可重复、可隔离。

评测方法与“榜单偏差”纠偏

> **InfoQ：Qoder如何收集和利用用户反馈？在阿里内部和企业客户中分别采用什么机制？你们是如何做评测（evals）的？更看重端到端、触发型，还是能力型评测？**

夏振华：关于用户反馈收集与利用，Qoder所有反馈的收集都遵循用户授权或者主动提供的原则，并符合相关隐私条款和企业的安全规范。无论是阿里内部还是企业客户，

都可以直接通过IDE内置的反馈入口、官方邮件或论坛的方式提交建议与问题。对于 阿里内部团队，我们还建立了更实时的沟通机制，例如钉钉协作群，让研发、测试、业务团队能够提前尝鲜新版本，在真实工程场景中快速反馈问题并推动迭代优化。

在评测方面，我们覆盖了主要的研发场景，包括前端、后端、客户端开发等，以及主流编程语言的多种任务类型——从bug修复、需求实现，到代码重构、结构优化等。我们既关注端到端的整体效果，评估任务从触发到交付的全链路完成质量；也重视核心能力的精准评测，如代码生成质量、检索准确性等。在此基础上，我们会进一步分析过程中的成本消耗与执行效率，确保整体表现既高效又稳定。

> **InfoQ：我们知道市面上的编程工具都宣称自己在开放基准测试里表现最佳。但这些基准往往无法覆盖企业真实的复杂场景，比如超大规模单体仓库、成百上千个微服务、大规模迁移和依赖升级。Qoder在评估和优化时，如何避免只在“排行榜”上好看，而是能真正解决企业开发中的难题？**

夏振华：这些开放基准测试通常无法覆盖企业真实的复杂工程里的研发场景，所以我们建立了自有得评测集，覆盖的维度会比开放基准测试更多，包括前端、后端、客户端等不同研发场景，主流编程语言，以及多类型任务——从bug修复、需求实现，到代码重构、架构优化、跨服务依赖调整等。与很多工具只在开放测试集有限样例中“跑分”不同，Qoder的评测环境会更贴近企业真实的研发项目。

> **InfoQ：在探索模型边界时，有人提出要敢于“推模型一把”，看看它是否能被逼出新的能力。结合你们的经验，当模型未达预期时，如何快速判断原因是提示词设计不当、模型选型问题，还是模型本身的能力瓶颈？**

夏振华：我们一般会先用一组简单、可控的基准任务建立下限；若通过改写提示词或者微调工具实现即可显著提升效果，通常是提示词设计不当或在该模型下的适配性问题。

在相同的prompt与评测数据下横向替换不同模型，若表现差异显著则是模型选型问题；若各模型都以类似方式失败且加入few-shot/思维链也无明显改进，多半是模型能力瓶颈。

另外一个点是可以看一下厂商是否提供基于模型的开源产品或提供官方的最佳实践，如果参考并调整后仍无法达标，更可能是模型能力瓶颈。

实践方法与路线图

> **InfoQ：最后能分享一些Qoder的使用技巧吗？**

夏振华：首先，任务描述一定要尽量一次说明清楚，技术栈、功能细节、规范要求都尽可能完整，否则来回补充和追问不仅会浪费时间，也会增加成本，并影响整体效果。

另外，可以在项目中放置一个rules文件，将代码风格、文件结构以及规范要求写明，这样Qoder生成的内容会更符合项目标准。

除此之外，Qoder具备智能记忆功能，遇到关键的业务规则或曾经踩过的坑，我会主动让它记录下来，比如API的认证方式，这样以后就无需反复强调。

会话管理同样不可忽视，完成与当前任务无关的编码工作时，最好新开一个独立会话。如果某个会话窗口的历史上下文过长且不相关，可以主动触发压缩，只保留核心信息，这样后续响应会更快、更准确。

版本控制也非常重要，每完成一个功能模块我都会及时提交（commit），方便在出现问题时迅速回退。

> **InfoQ：从未来发展来看，Qoder最想成为怎样的产品？如果一定要拿Cursor、Claude Code、Lovable来对标，你们最想在技术层面超越的“关键一点”是什么？**

夏振华：我们希望成为面向真实工程的Agentic Coding Platform，不只是“会写代码”，而且能够实现从需求到可合并PR的端到端执行者，深度理解仓库、做出系统级设计决策，并产出高质量、易维护的变更代码——Think deeper, Build better。

颠覆Cursor，AI编程不再需要IDE！用并行智能体重构开发范式，MongoDB CEO高调站台

作者 Tina

在AI工具风靡开发圈之前，一批经验丰富的资深程序员，对它们始终保持警惕。这些人，包括Flask作者Armin Ronacher（17年开发经验）、PSPDFKit创始人Peter Steinberger（17年iOS和macOS开发经验），以及Django联合作者Simon Willison（25年编程经验）。然而，就在今年，他们的看法都发生了根本转变。

Armin在过去几个月中彻底改观。他曾对AI工具“不敢授权”，如今却愿意将工程主导交给编程代理，自己转而去泡咖啡或陪孩子玩耍。“如今显而易见，我们正经历一场翻天覆地的变革。”

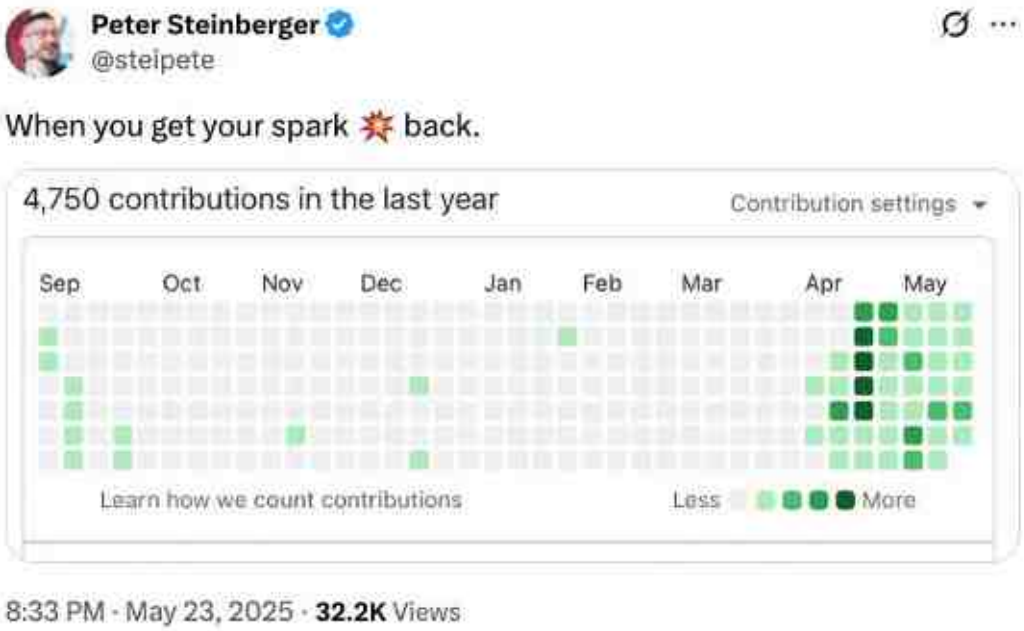
AI Changes Everything

written on June 04, 2025

At the moment I'm working on a new project. Even over the last two months, the way I do this has changed profoundly. Where I used to spend most of my time in Cursor, I now mostly use Claude Code, almost entirely hands-off.

Do I program any faster? Not really. But it feels like I've gained 30% more time in my day because the machine is doing the work. I alternate between giving it instructions, reading a book, and reviewing the changes. If you would have told me even just six months ago that I'd prefer being an engineering lead to a virtual programmer intern over hitting the keys myself, I would not have believed it. I can go can make a coffee, and progress still happens. I can be at the playground with my youngest while work continues in the background. Even as I'm writing this blog post, Claude is doing some refactorings.

Peter作为拥有17年经验的iOS和Mac开发者，近期重新开始捡起了编码工作。2021年，PSPDFKit获得1亿欧元融资时，他出售了自己在公司的全部股份，此后就只是偶尔写点东西。“现在，我们正处于技术发展的一个令人难以置信的十字路口。”“这些工具彻底改变了软件构建的方式。”“Agentic AI工具是有史以来最大的一次变革。”



Simon曾是Django的创始人，目前是一名独立软件工程师。对于当前AI工具在软件开

发中的实际能力，他是这样评价的：“现在，编码代理（Coding Agents）真的能用了：你可以把一个LLM放进循环中，让它运行编译器、测试框架、linter和其他工具，给它一个目标，然后看着它完成整个流程。**在过去六个月，模型的进步已经从‘好玩的玩具演示’，跃升到了日常可用的生产工具。**”

在这个背景下，像Copilot、Cursor和Claude Code这样的智能IDE无疑是促成大家看法转变的关键因素之一。与此同时，我们也注意到了一种新的产品类别正在兴起：Factory AI，它声称目标是摆脱传统IDE的形态。

Factory AI由理论物理学博士Matan Grinberg和曾任Hugging Face及微软数据科学家的Eno Reyes共同创立的公司打造。

Grinberg指出，如今AI被视为超越以往所有平台变革的强大力量，但Copilot和Cursor依然是传统的IDE。“尽管大家普遍认为软件开发模式会改变，但目前主流的做法仍是‘在原有流程上附加AI’：开发者在IDE中编码15年，现在只是把AI塞进IDE，成了AI IDE。换句话说，真正的转型尚未发生。”

因此，Factory AI的核心理念是重新思考软件工程，不仅仅是写代码，而是关注整个端到端的软件开发流程。

视频地址：https://www.infoq.cn/article/HBISNaNNSEqNBwvD12QP

Factory基于Grinberg所称的“Droid”构建：一个引擎，用于提取和处理公司工程系统数据以构建知识库；以及一个算法，用于从知识库中提取洞察以解决各种工程问题。Droid的第三个核心组件Reflection Engine，充当Factory所利用的第三方AI模型的过滤器。

在过去的两年里，这家公司一直**只面向企业提供服务**，据说在企业领域取得了显著成功，其客户涵盖了从“种子期”到“‘上市”等不同发展阶段的企业。一个月前，**Factory AI首次向公众开放**，并获得了MongoDB CEO的高度赞扬：

有些人将其与Devin对比，但两者也存在不同的地方：Devin宣扬的是“取代所有工程师”，但Factory更强调的是“增强”工程团队的能力。

Matan Grinberg在最近的一次访谈中也体现出他对这个行业的独特见解。他认为AI的到来将使程序员的工作重心转向更高层次，并且“可解决的问题总量”也会因此变得更多。未来那些具备系统性思维、深入理解底层原理并善于利用AI工具的程序员，将更具

价值。

我们翻译了Matan的访谈，来了解Factory的思想。一些亮点摘录如下：

- 五年后的世界会是什么样？我认为，从“想法”到“解决问题”的这段旅程将会高效得多。可能以前需要一千个人才能完成的事情，将来只需要十个人就可以搞定。
- 会有一些软件问题，其规模之大，就算地球上所有工程师一起上也解决不了。而个人用户面对这些问题时，能够借助一支“虚拟工程师大军”来完成解决。
- 我们习惯于线性思维，但很多技术的演进却是指数级的。
- 如果大家都有AI技术，这种竞争态势会强制性地抬高“好软件”的标准。

为什么要做Factory?

> **Matthew Berman：Factory是一个和传统IDE非常不同的产品。它本质上就不是一个传统意义上的IDE。这可能也反映了你对当前AI与软件工程结合现状，以及未来走向的看法。那我们直接切入正题：你为什么要做Factory？能先说说背后的故事吗？**

Matan Grinberg：好的，我可以从为什么开始做Factory说起。在做Factory之前，我当了十年理论物理学家，主要研究弦论。当时我执着地坚持这条路，并不是因为我觉得它最适合我，而是因为它很难——我对难的事情总是有种莫名的吸引力。后来我去了伯克利，开始攻读博士学位。

在伯克利期间，我开始接触一些AI的研究生课程，接触到了当时还叫“程序合成”（program synthesis）的领域——现在我们称之为“代码生成”。我一头栽进去，很快就彻底转向了AI研究。那是2022年初。代码生成让我着迷，是因为我从理论物理和弦论转过来，已经被训练成习惯关注那些最基础、最普遍的事物。而代码或软件开发，在整个AI领域里扮演了极其普适的基础角色。

模型在编码方面的能力，往往直接关联到它在其他下游任务中的表现。不管是写诗，还是回答学术问题，模型的编码能力越强，它在其他任务中的能力也越强。这种核心能力吸引了很多数学家和物理学家进入这一领域，这也是我当初被吸引的原因。

至于你刚才提到的“为什么关注IDE之外的新交互模式”这个问题，我想先引用一句我们在Factory很认同的话——亨利·福特曾说过：“如果我问人们他们想要什么，他们会说更快的马。”

我们认为软件工程的未来也是如此。现在的开发模式基本都建立在IDE上，这种范式已经存在二十多年了，开发者都很熟悉，形成了非常固定的使用习惯。而大家普遍也认同，软件开发在未来五年会变得非常不同。问题在于：我们现在是一个开发者写每一行代码的世界，而未来我们可能会进入一个开发者几乎不再写代码的世界。

有一种思路是：从当前“写满代码”的世界逐步演进到“零代码”的理想状态。但我们的观点是——这就像是试图从马演化出汽车。但历史上并不是这样发生的。我们是从第一性原理出发，直接**造出**了汽车。我们现在的做法就像是在重新思考“交通”这个问题一样，重构软件工程的方式。

我们把这种模式叫做agent-native软件开发，与传统的IDE编程方式有本质的不同。

传统开发者的思维是：“我怎么才能更快地完成这个任务？”于是会使用自动补全、单元测试等等工具来加快效率。但agent-native的模式，是思维方式发生转变：面对一个大任务，开发者要思考“我怎么把它拆解成离散、可验证、可并行执行的小步骤”，然后把这些步骤交给智能体去并行完成。

这和串行地更快工作是两种完全不同的提速方式。一个任务如果能并行拆成四个部分去做，其速度提升远超在原有流程中微优化。

Matthew Berman：我在使用AI产品时，最让我惊艳的时刻，往往是当我意识到可以“并行”处理任务的时候。第一次有这种感受，是我看到OpenAI的operator，可以启动一个需要数分钟甚至几十分钟完成的长任务，然后再启动第二个任务。我意识到：我手里有多个agent同时处理我原本要一个个完成的任务，真的是非常震撼。我确实非常相信这种多智能体并行执行任务的未来愿景。

> **Matthew Berman：你刚刚提到一个很有意思的观点：代码生成的能力是模型其他强大能力的“上游”。那我就想问一下——你有没有看到苹果这个周末刚发布的那篇论文？你笑了，我猜你看过。这篇被大家广泛传播的论文探讨了“推理的**

> 错觉”——它指出语言模型可能并没有真正地在用自然语言进行推理。他们设置了一些谜题，当复杂度上升后，模型就无法解决这些谜题了。
> 但这篇论文没有提及的一点是：这些模型可以用代码解决这些谜题，而且在任意复杂度下都能做到。这让我特别想问你：你认为语言模型用代码进行逻辑推理、解谜题的能力，算是“智能”吗？你怎么看这篇论文？

Matan Grinberg：—坦白说，我没有认真读那篇论文，只看了摘要。没来得及通读全文。说实话这篇论文的发表时机，和苹果现在在AI市场的处境联系起来还挺有意思的。

至于“写代码算不算智能”？我觉得我们现在所处的时代特别有意思的一点是，整个“智能”这个概念本身正面临质疑。每当一个大模型做成某件事，我们总会本能地想：“哦，那不算智能，那只是记忆”或者“那只是训练数据的复现”。我们总会说它只是插值，不是真正的推理。

但这其实很难定义。我至今也没看到过一个真正清晰、统一的“智能”定义，能够明确划分“这算智能”、“那不算”。

我觉得有意思的一个方向是ARC奖（ARC Prize）那边在做的研究。他们强调泛化能力、模式匹配等等。这方面他们做了不少不错的探索。有一点非常明确：模型在训练数据中接触越多的任务类型，它在这些任务上的表现就越好。

当然，这也会引发一种质疑：“它只是做熟了，才会。”但人类也是这样。人类能泛化得更好一些，从一种题型推广到另一种题型，但本质上，我们也需要训练。

Sarah Guo最近说了句话我挺认同：当人们谈论AGI或智能时，其实他们潜意识里说的是“意识”（consciousness），这两个概念经常在无意识中混淆。

回到你问的问题，我的看法是：要解决某些编码问题，确实需要智能。所以按这个标准来说，这些模型当然具备智能。至于它们有没有超越训练数据的“广义智能”？答案是否定的。但这正是各大实验室现在努力攻克的方向。

人类vs智能体：并行协作方式有何不同？

> Matthew Berman：我也相信，相比于自然语言，模型通过写代码的方式，展现

> 出的泛化能力要更强。我想回到“并行处理”这个话题上来。过去几十年里，人类工程师团队协同工作时，通常不会在同一段代码上一起开发——否则就会产生大量冲突。而Git就是为了解决这些冲突而生的。那么，智能体在并行协作时的方式和人类有何不同？

Matan Grinberg：好问题。我认为这也涉及一个关键问题：当软件开发智能体越来越强大时，人类在其中扮演什么角色？

如果是两个完全独立的任务，比如开发功能一和功能二，那当然可以并行。但如你所说，如果它们要修改相同的核心代码区域，那就会出现合并冲突。

我认为人类工程师在未来将扮演一个非常重要的角色：比如在处理“功能一”的时候，如何**最优地将任务拆解成子问题**，而这些子问题**必须是可以并行处理的**。

那人类需要什么样的能力才能做到这种高效拆解？答案是：系统性思维（systems thinking）。

真正优秀的工程师从来都不是那种记住每门语言所有细节的人，而是具备系统性思维、能在各种限制条件下做出合理推断的人。现在我们拥有了一种全新的交互模式和开发方式：不再是“思考完怎么做，然后自己动手实现”，而是尽快地整理出一个完整的“任务包”，包含可执行的步骤，以及验证这些步骤是否完成的标准，然后交给智能体去实现。

是否还需要学编程?

> Matthew Berman：你提到系统性思维，我很认同。我经常被问到一个问题，你可能也会遇到：现在还值得学编程吗？我问过GitHub CEO这个问题。我自己有两个年幼的孩子。对我来说，学会编程是我一生中最重要的技能之一，它让我可以把脑海中的想法变成现实，而不依赖他人。
>
> 几年前，如果你问我，我会毫不犹豫地告诉我的孩子：“编程是最重要的技能，必须学。”
>
> 但之后有一段时间我产生了动摇。不过让我再次把这个问题拉回到系统性思维。

> 除了学编程之外，系统性思维本身也是我学到的最重要能力之一。即使未来大多数代码都由智能体来写，这种思维方式依然对整个工作流程至关重要，甚至放在整个“人生技能”范畴下也一样重要。你怎么看？

Matan Grinberg：我完全同意，甚至想说个稍微更“激进”的观点：对现在还在读高中或大学的年轻人来说，他们学的是计算机、数学、物理还是生物，其实并没那么重要。

重要的是：他们进入的这些领域，问题空间都非常庞大、信息密度极高，还有几百年的学术历史——现实中没人有时间把所有细节都学一遍。

所以关键在于：你是否具备一种能力——**进入一个复杂领域后，迅速搞清楚核心基础是什么，哪些细节是你必须弄懂的，哪些可以暂时模糊理解，但依然能继续往前推进。**

我可能有点偏见，因为我自己花了十年时间研究物理。一个比喻是这样的：每次我站在黑板前写下一个公式时，我不会每次都重新推导相关定理——那样效率太低。但在我人生中的某个阶段，我确实做过这些推导，而且如果有人现在让我推一次，我也能推出来。

我觉得这和软件开发很相似。在大学的计算机课程中，我们通常会从底层逐步学到高级语言。现实中你可能永远不会用到机器码，但理解它是有帮助的。

这同样回到了系统性思维的问题：你得了解你所工作的“整个技术栈”的结构。就像物理学家和数学家也会使用计算器、用各种他人推导好的定理，他们不会每次都重头再证明一遍，但他们**曾经学过这些推导**，知道如何从头做一次。

如果你忽略了这些基本功，哪怕现在有智能体帮你写代码，最终也可能吃亏。因为你没有那个“在信息大山中摸索出路径”的经验和能力。

而这正是你说的那个关键技能：面对一个复杂领域，知道该深入掌握什么、又能对哪些东西保持模糊但足够用的理解，并继续往前推进。

Matthew Berman：我听你描述的其实是“抽象层”的概念。而放回软件工程的上下文里，今天很多人都在问：未来我们会成为orchestrator（编排者）吗？我们是不是主

要职责就是检查智能体的工作？无论是哪种角色，**了解底层原理**仍然是必须的。哪怕你不亲自去写每个算法、不从头实现每个函数，**你也需要有基本的判断力**，才能检查智能体做的是否正确，或者调度它们正确地协作。

软件工程的突变，以及5~10年的展望

> **Matthew Berman：过去两年里，软件工程的变化速度真的令人震惊。让我意外的是：AI对软件工程的影响居然是最大的（尽管现在看好像也不算意外了）。而且它的变化速度还在持续加快。你刚刚描述了未来几年可能出现的“智能体编排”形态，以及从IDE向更高抽象层的迁移。那么你觉得5年后、10年后的世界会变成什么样？**

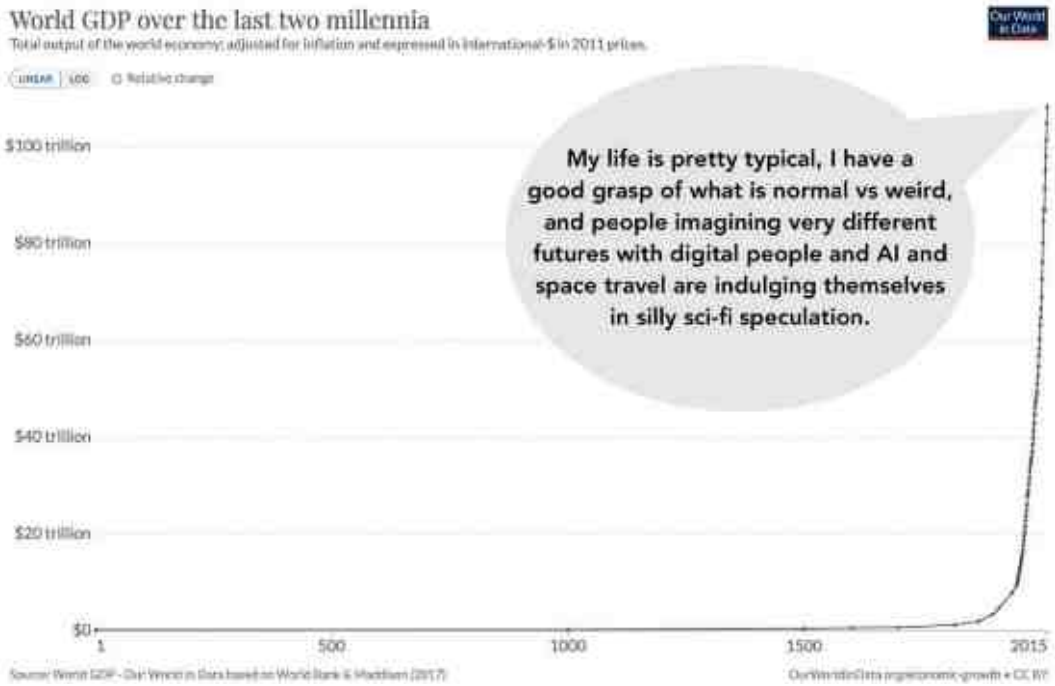
Matan Grinberg：当然我知道这类预测不可能准确，但我还是很想听听你的愿景。

我完全同意你说的“预测很难”这个前提。预测5年后的事情本来就极具不确定性。

2020年的时候，可能有极少数人能预见我们今天的样子，但真的非常少。因为每一代模型的进步都会产生指数级连锁反应，而这些累积效应会影响未来每一年的节奏。

所以预测时，一个很重要的意识是：**人类并不擅长处理“复利”这种非线性思维方式。**

这不是我第一个提出来，也不会是最后一个这么说的人。我们习惯用线性方式思考，但科技进展是指数曲线。有一个很棒的梗图我想跟你分享。这来自Daniel Kokotajlo的博客。这张图非常形象地说明了我们在处理非线性增长时的无能。



比如，全球GDP增长就是一种非常夸张的指数曲线。但在2010到2020这十年里，大多数人并没有觉得进展有多“疯狂”。但其实那段时间GDP曲线几乎是垂直往上的。

所以当我们试图思考未来5~10年时，我们其实很难做出有高置信度的判断。

我唯一能说的是，**在思考未来时，我会尽量回归初衷**。我们搞软件工程的初衷并不是为了写代码本身。我们不是为了代码而写代码，而是为了造出我们想造的东西。

我们写程序，是为了借助计算机这种比人类快得多的“计算机器”完成我们的目标。计算只是手段，不是目的。真正的目标，是用它来构建连接世界的工具、解决现实的问题。我们想要实现的是有意义的终点。比如，我们想要模拟某些科学现象，以便研发新药；又比如，我们只是想让打车这件事变得更高效，不必再等上三十分钟才叫到出租车。

这些才是我们的**目的**，而**手段**是软件，因为我们想用机器来完成这些事，而不是仅依靠人工系统。为了与机器交流，我们不得不学习一种非常精确的语言——编程语言。后来我们慢慢让这种语言变得越来越抽象，因为直接用比特与机器交流实在是太低效了。用更高层次的抽象语言，效率反而更高。所以我认为这个趋势还会持续下去。

过去，我们必须深入学习编程，才能知道如何与机器协作。但现在我们正开始“抽

离”出来，重新关注我们真正想实现的目标。

所以回到你刚才的问题——我认为在未来的5到10年里，我们能以更快速度解决更多问题。比如，以前一个问题可能需要1000名工程师和10年时间去建立一家公司来解决它。但未来你可能只需要10个人，就能把一个想法变成现实并完成整个闭环。

从这个角度看，好像会出现公司人数下降、不会再有一万人的大公司了。但我认为还有另一个相反方向的力量存在：如果你有1万名工程师，而他们每个人都能把任务分派给几百个智能体——那么软件能实现的规模和复杂性也会大大提升。

现在我们都很难想象如何协调一个十万人规模的软件开发组织。但像微软这样的公司正在尝试这么做。虽然它们的效率肯定还存在瓶颈，但如果你把时间放到未来5~10年，届时软件系统的复杂性可能远超我们今天的想象。

那会是一个我们几乎无法理解的世界。想象一下“相当于一百万名软件工程师同时工作”的产出规模——这都难以想象。

Matthew Berman：听你讲完，我感觉我们对未来的看法其实非常一致，而且你也非常乐观。

很多人觉得，如果现在一款产品过去需要1000名工程师，而将来只需要10个，那公司一定会裁掉那990个人。但我持反例，甚至持更乐观的态度。因为真正发生的事是——**“可解决的问题总量”会急剧扩大。**虽然现在10人就能解决一个问题X，但之后你会有问题Y、Z、α等等。那些原本做X的工程师可以继续去解决这些新问题——他们只是“被超级加持了”。

所以我不认为最终会是那种“一个人领导一家公司，下面全是AI军团”的格局。我不认为未来世界只剩下10家公司，每家就一个人+一堆智能体。恰恰相反。以前有些“长尾问题”之所以没人解决，不是因为它们不重要，而是**解决它们的“智力成本”不划算**——但这一点即将改变。

Matan Grinberg：确实如此。通常一个问题的“可服务市场规模”（TAM）会决定它能吸引多少工程师和资金来解决。比如某个问题只影响全世界20万人，那你就会根据他们的消费能力来判断值不值得投入资源去解决。

以前，也许这种问题只值几百工程师的投入。但现在情况不同了：哪怕某个问题只影响几千人，你也可以投入相当于十万工程师的“智能体产能”去解决它。

我觉得这太酷了。哪怕一个问题只影响2000人，我也觉得值得解决——理想世界里，不该有人被遗留，而现在我们可以把这种“计算力”集中起来去解决这些问题。

Matthew Berman：甚至可能最终把问题细化到只影响一个人的程度。而那时，软件的开发成本也会降到如此之低，以至于哪怕是“为一个人量身定制”的解决方案都能盈利。这真的非常有趣。当然，我们刚才讨论的，是那种**极致长尾**的问题。未来也还有另一类——**我们现在甚至想象不出的庞大问题**。这些也许是几十年以后的事。但假如我们未来走向星辰大海，一定会产生一些**巨大的软件问题**，是全人类的工程师都解决不了的。

但在那个未来，每一位人类工程师都会拥有属于自己的“虚拟工程师军团”。所以也许，到时候这些原本无法解决的问题就变得可解了。

Matan Grinberg：完全同意。我特别喜欢你刚才说的那种描述方式：这不是为了软件工程本身而做软件工程，而是每个人有自己的问题——他手上有AI工程军团，能用来解决问题。

无论是探索宇宙，还是别的什么，人们总会在路上遇到各种挑战，而他们可以更高效地解决这些问题。这种效率的提升，显然是对整个社会的巨大正向推动。

Factory的未来

> **Matthew Berman：说回Factory产品本身。我最初使用它时，最打动我的是它的“设计”。**
>
> **不仅是UI，更是整个使用体验（UX）。它显然不是一个IDE。所以，谈谈你们的设计理念吧。Factory的UI/UX正在走向什么方向？它如何影响人与产品之间的交互？**

Matan Grinberg：首先，我要大大表扬我们的设计师Cal——他其实是我亲哥哥。能和他一起工作，真是一种乐趣。我们设计上的几个亮点之一是：Cal的背景其实是工业

设计，他毕业于RISD（美国最顶尖的艺术院校）。而Factory有22位世界顶尖的工程师。

我认为我们特别注重在设计时吸收不同视角，哪怕我们做的是给程序员用的工具。很多人会直觉地认为：“这是程序员用的产品，那就让程序员来设计吧，听听他们的意见就行。”比如功能A放这儿还是放那儿，大家一起投票决定。但Cal作为一个“局外人”，他的设计背景和UX/UI视角，其实带来了非常宝贵的价值。

因为我们的目标之一，是**摆脱IDE的形态**。

而市面上最顶尖的程序员过去十几年都一直在使用传统IDE，所以他们脑子里已经形成了非常深的习惯。反而是Cal——**他从没用过IDE**，所以他根本没有这些“旧习惯”，这就让他能提出一些全新的视角。

这有点像物理学领域也会出现的现象：为什么很多物理学家的“最强突破”都发生在27岁左右？因为那时候他们既有足够的专业背景，又没被各种教条和惯性思维束缚住。他们还有勇气和能力去“质疑一切”。

这种心态在我们的设计工作中非常有帮助。

至于产品未来的走向，如我之前所说，我们在尝试构建一辆“车”，而不是在“马车”上反复加速。传统做法是在IDE上堆叠更多AI功能。但我们的目标是**彻底跳出IDE**。

最终理想是，人类开发者不再需要手动去写代码。开发者只需要提供一份清晰的计划——不仅仅是“给我做个仪表盘”这种模糊需求，而是通过设计正确的交互，引导他们提供真正准确、可操作的限制条件和目标定义。这样我们才能真正理解开发者的意图，并尽可能忠实地将其实现。

我们希望能有确定性的方式来验证生成结果。举个稍微有点傻的例子：假设我想做一个蓝色的仪表盘。我对Factory说：“嘿，帮我做一个符合Factory主题风格的仪表盘。”结果它给我生成了一个粉色的界面——这显然违背了我们现有的设计系统。

我们想要的是，即使Factory出现这种偏差，生成了不符合我们预期约束的内容，它也应该具备能力去发现这个问题，然后自动迭代修正，而不是通知我“我搞定了”，然后我点开一看——结果居然是粉色的。然后我还得说：“我们通常是深色模式啊。”

所以，我们正在塑造一种交互模式，让人类开发者可以更清晰地表达他们的约束条件，或者系统能随着使用不断学习这些约束，这样他们就不需要每次都重复说明。

与此同时，在底层的系统层面，我们也在确保检索和代码生成的能力始终保持领先。确保那些执行代码的“droid”（智能代理）能基于运行结果进行迭代，比如某个测试失败了，它能据此修复而不是让人来回手动干预。

我们越能搞定这一点，开发者就越不需要自己手动修改代码——这也是我们走向“Agent-Native”未来的重要一步：人类开发者只需要清晰地定义他们的目标和任务范围，然后把它交给代理去完成。

Factory的幕后机制

> **Matthew Berman：Factory对企业的代码库理解得非常深。那么，你能不能分享一些幕后内容，比如Factory的droid（智能代理）是如何理解代码、保持模式一致、不做不该做的修改的？Factory在这些方面有哪些独特的方法？**

Matan Grinberg：我觉得主要有三点关键技术，我们可以一一聊一下：

1. **原生集成**
2. **记忆系统**
3. **本地和远程的代码执行能力**

首先是原生集成。

MCP服务器确实挺强的，但我觉得它目前有点被过度炒作了。MCP的优点在于，它能把IDE中的信息注入到模型里，但它的缺点也很明显——对于大型代码库或企业环境，它的计算都是临时完成的，这和人类工程师的工作方式非常不同。

人类工程师在处理问题时，并不会“清空记忆”然后重新搜索所有相关内容。通常他们已经对自己的代码库和团队架构有一定的“先验理解”，遇到问题时可以直接想到：“哦，这个跟那块代码有关”，然后快速定位、解决问题。

Factory采用类似的思路。我们和GitHub、Slack、Jira、Sentry、Datadog等工具进行了深度原生集成，并且预先计算了它们之间的关联关系。

比如：某个Notion文档里写了一份技术设计方案，它是基于一些客户需求或问题产生的，然后你提交了一个PR去实现它，之后Sentry报告了这个PR引发的故障。现在，如果你要追溯问题来源，我们不需要临时从各个系统中“抓信息”，而是已经知道这些内容的上下文链条。你可以更快、更高质量地排查和修改问题。

所以相比MCP，Factory的原生集成方式在大型企业中更节省时间，也更稳定可靠。

> **Matthew Berman：好的，那我们再聊聊“记忆”这一块。我本来也想问这个，因为像ChatGPT的记忆功能就非常重要，它提升了交互质量。那么Factory是如何实现记忆机制的？**

Matan Grinberg：Factory的记忆分为多个层级，能让系统逐步“了解你”：

- **组织层级记忆**：了解整个组织的工作方式、产品细节、面向的客户、使用的技术栈等。
- **团队层级记忆**：不同团队可能有各自的流程和习惯，Factory会分别学习。
- **个人层级记忆**：比如你写代码时总是忘记写测试，Factory会记住，并在你提交PR时自动补上测试；或者你们团队对PR有严格要求，Factory就会自动帮你调整格式、补文档等，以满足这些要求。

这些记忆会随着使用持续学习和进化，当然，你也可以手动修改或配置这些记忆，不管是组织负责人、团队leader还是个人开发者都可以自定义。

第三个关键点是——Factory可以执行代码。

具体来说，Factory提供了两种执行模式：

- **远程执行**：你可以在云端环境中并发启动多个droid，让它们同时处理不同任务，比如同时生成多个模块或完成多项需求。
- **本地执行**：如果你想更亲自参与控制流程，比如更精细地监控执行细节，那你可以让代码在本地设备上运行。

另外，具备真实的代码执行能力，意味着你不再是“写完代码就盲目提交”，而是能够真正运行它、验证它能否编译通过、测试是否成功，以及它是否真的按照你的预期功能去执行。

这与人类工程师的工作方式非常相似。人类不会只是写完代码就提交PR，觉得“应该能行”。他们会实际运行、检查并确认：“这个代码真的完成了我想要的功能吗？所有测试都通过了吗？”

非技术公司是否该采用Factory?

> **Matthew Berman：现在Factory大大降低了写代码的门槛，让高质量、规模化的代码生产变得容易。这让我在思考：垂直型SaaS公司会不会受到冲击？比如以前那些不是技术型的企业，没有富余的工程师团队，也无法自建系统解决内部需求。那么你觉得这些企业是否该采用Factory，自建一些工具？**

Matan Grinberg：事实上，我们目前最大的一些客户，其核心竞争力也并不是软件开发。比如说我们的一位客户是德国制药巨头拜耳（Bayer），也就是那个做阿司匹林的公司。他们就是我们的用户之一。

他们并不卖软件，但现实情况是，**今天的每一家企业，在某种程度上都依赖软件**。即便软件不是你直接销售的产品，它对企业提高杠杆效率来说依然至关重要。

如果你现在能让“droid”（智能代理）来帮你分担任务，或者说你之前为了一些旧系统支付了大量费用，但实际上只需要用到其中的一小部分功能——现在你完全可以自己内部搭建一个更符合需求的工具。

或者说，你可以用更少的工程师，更快地发布产品。因为对于那些软件不是核心业务的公司来说，他们手上的工程师本来就不多。那你就更需要给他们配备最前沿、高效的工具。

而且还有“乘数效应”——每位工程师都能有多个智能代理协作，这种一对多的工作方式会极大提高产出。所以我的答案是：是的，非技术公司特别适合使用Factory。

SaaS终结了吗?

> **Matthew Berman：尤其是因为Factory的存在，才让那些没有太多开发经验的人，也能真正写出优秀的软件。这一点非常有意义。那我们接着说吧。像Factory这样的产品正在把软件开发的成本压缩到接近零。那么，这对垂直型**

> SaaS公司意味着什么？

Matan Grinberg：是的，现在确实是一个非常有意思的时间点。这其实也呼应了你刚刚提到的一个观点：如果AI工具能让每个工程师的效率提升10倍，那公司是否会因此裁员？

如果世界上**不存在竞争**，那也许会这样。但现实是——**每一家你的竞争对手现在也能让他们的工程师效率提升10倍**。

所以你当然可以裁员，然后保持当前的生产效率。但如果你的对手**没有裁员**，反而用这些效率工具让自己变得更强，那他们的整体产出将是你的100倍，最终你只会被甩在后面。

这对软件用户来说是好事，因为这意味着——**你所用软件的门槛和质量标准将会显著提高**。

这其实就像博弈论的经典场景：如果只有你一个人拥有AI技术，那你当然可以靠它降本增效。但现实是，大家都有这项技术。这种**竞争态势会强制性地抬高“好软件”的标准**。

类似的例子还有90年代的网站。当时要做出一个漂亮的网站非常难，而现在有了各种低代码工具，几分钟就能搞定炫酷网站。所以现在我们眼里的“好网站”门槛，比以前高了很多——**90年代那种顶尖网站，现在只能算入门水平**。

> Matthew Berman：最后一个问题，Matan。你觉得大家接下来该关注什么？你们未来6个月会做哪些新东西？

Matan Grinberg：我觉得最值得期待的是，**智能代理（agents）会变得更可靠，质量更高**，你用更少的提示，它们就能更准确地完成你想要的任务。

现在我们看到，真正沉浸在“Agent Native”软件开发方式里的用户，在使用Factory时会有一种“魔法般的体验”。但即使在那些拥有上万名开发者的大公司里，仍有很多人对AI和Agent不感冒。他们可能并没有太强烈的意愿去尝试这些新工具，因为现在确实还需要一些“善意的投入”——你要去写清楚你希望Agent做什么，才能让它给你一个

好的结果。

但我相信，再过**六个月**，哪怕你完全不在乎AI、依然每天待在你的Emacs里、只想按照老方式写代码，只要你肯花一分钟尝试一下Factory——你就会被惊艳到。

那种提升开发者能力和工作杠杆的方式，会真正让你感受到empowered，让你愿意拥抱这种新的范式。

参考链接

- https://www.youtube.com/watch?v=sI3D1UY-cV0&t=5s
- https://www.youtube.com/watch?v=8gKN29Ea-J8
- https://techcrunch.com/2023/11/02/factory-wants-to-use-ai-to-automate-the-software-dev-lifecycle/

氛围编程行不通！CTO们集体炮轰AI编程：不是失业，而是失控

作者 Tina 傅宇琪

不像GitHub CEO Thomas Dohmke那样，一边喊着“要么拥抱AI，要么离开”，一边忙着卖Copilot订阅，真正带队写代码、扛事故的CTO们，对“氛围编程”的态度冷静得多，也残酷得多。

他们没有利益冲动去推销新概念，更没有时间沉迷话术，他们面对的只有实打实的线上系统、真金白银的技术债和凌晨的报警电话。也正因为如此，他们的结论比任何口号都更值得参考。

在CTO的叙述里，氛围编程并不是“生产力革命”，而是一场接二连三的灾难。

一位CTO说得很直白：“氛围编程看似捷径，但本质是死路。”

Let Set Go的CTO Ritesh Joshi的团队刚经历过一次“教科书式”事故：开发者用AI生成了一段数据库查询，小样本下毫无问题，但一旦遇到真实流量，系统立刻被拖到爬不动。问题不是语法有错误，而是底层架构。

Cirrus Bridge的创始人兼首席软件架构师Patric Edwards也分享过一次“惊心动魄”的经历：一名新人把AI建议和Stack Overflow代码片段拼凑起来，写了个用户权限系统。测试和QA全部通过，上线后两周才发现已注销的用户依然访问某些后端工具。原因只是一个“看似合理”的真假逻辑反了。修复这漏洞，资深工程师花了整整两天。

对他来说，这不是bug，而是一种“信任债”：高级工程师被迫长期当侦探，反复逆向解读基于vibes拼出来的逻辑，只为发布一个稳定更新。

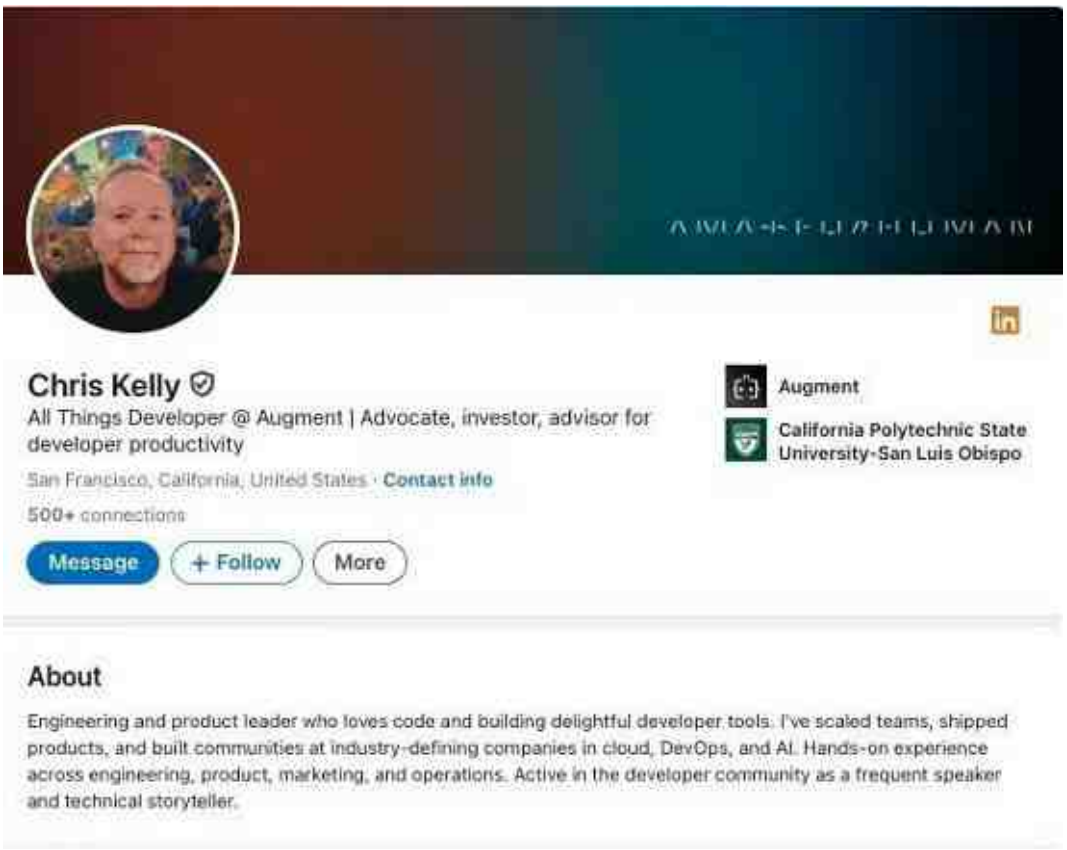
AlgoCademy的CTO Mircea Dima遇到过更隐蔽的状况。一位开发者用AI写了二分查找实现，并被用于核心搜索功能，结果上线一周后发现在特定输入下会悄悄出错，直接导致生产系统宕机，造成用户流失。Dima总结说：“问题不在于AI会犯错，而在于vibe coding创造了一种危险局面——你只有等到系统真正崩溃，才会发现问题。”

App Makers LA的CEO Daniel Haiem的团队也被狠狠教训过。一名开发者用Firebase和npm包“vibe-coded”了整个认证流程。“在只支持简单登录时它能跑，但当我们需要多角色权限和区域隐私规则时，它彻底崩塌。没人能搞清楚各模块间的关联，中间件分散在六个文件里，没有逻辑模型，只有vibes。最后我们只能推倒重写，因为调试就像考古。”

Akveo的高级全栈工程师Mikhail Hryb既见识过AI开发的强大，也见证过彻底的灾难：“有个项目几乎完全靠vibe coding搭出来。MVP的确两天就交付了，而不是一周。但没人审核AI生成的代码。初级开发者写的烂代码至少还能读懂；AI生成的没人看过，结果就是一堆胡言乱语。不可调试、难以扩展、维护痛苦。”

AI写得快，CTO修得更快。氛围编程或许让功能上线飞快，但真正支撑生产环境的，是懂系统、能排查、懂业务逻辑的工程师。CTO们用脚投下的票，说明了这条捷径根本走不通。

与这些一线CTO的实践经验相呼应，Augment Code的工程和产品负责人Chris Kelly最近以Vibes Won’t Cut It为题发表了一次演讲。在分享中，他详细探讨了“氛围编码”在生产级软件开发中的局限性，强调情境在软件工程中的关键作用，而不仅仅是编写代码。



Kelly拥有超过15年的软件开发经验，曾在New Relic、GitHub、Salesforce和FireHydrant等公司帮助开发者提升生产力。他的观点直指核心：仅靠直觉和AI生成的代码不足以构建强大、可直接投入生产的应用程序，真正可靠的产品依赖的是结构化的软件开发方法。

以下为其演讲内容整理，供读者参考。

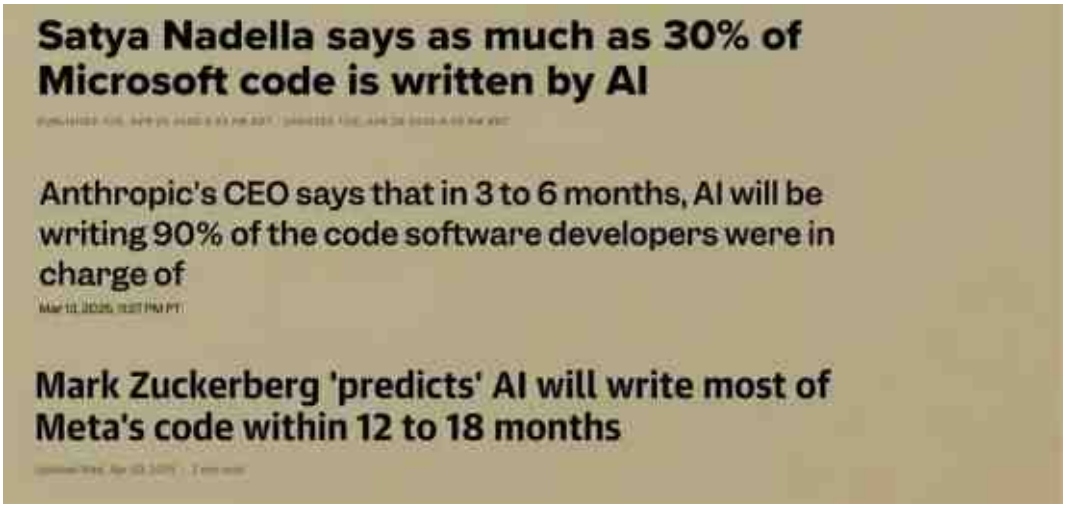
氛围编程行不通

如果你还没准备好，我得提醒你一个不太妙的事：到明年这个时候，我们当中可能有一半人已经不在这个行业了，至少如果你相信当下那些关于AI和AI编程的各种炒作的

话。

外界铺天盖地地在宣传这个东西，但我觉得他们大概率是错了。

不是因为我认为AI编程没前景，而是因为这些人可能已经很多年没有真正接触过一个生产环境了。他们说AI现在可以生成30%的代码，可那可能根本不是他们想象的那样。



不要小看生产环境的任何一行代码

AI写的代码，归根结底还是“代码”。有些人没有意识到，他们日常工作的代码基座庞大，几乎每一个关于架构、基础设施的决策都已经有人帮他们做了。比如说，如果我生成了30%的代码，对比的是每天在数百万行现有代码之上新增几千行而已，那这个30%其实没有多少“腾挪空间”可言，这些代码本来就只能干很有限的事情。说白了，就是多一个按钮而已。

没有冒犯的意思，但我接下来可能要吐槽一下Meta。如果你跟Meta的工程师聊过，你会听到这样的故事：有工程师花了六个月时间，在广告平台上造了一个按钮。这六个月，他的全部工作就是那个按钮。你说这样的系统，还有多少空间留给AI来“发挥”？

再说一遍，AI还是在写代码。这些代码的本质和我们过去五十年写的一样，语言也一样，没啥本质变化。而且，这些代码最终还是要在生产环境里跑。如果你没运营过一个大型的生产系统，就算你写了一行再漂亮的代码，放到复杂系统里照样可能出故障。复杂系统会有“涌现行为（emergemt behavior）”，不是你单看一行代码就能预判的。

那当系统出问题了，谁来修？谁来排查？谁来理解这些微妙之处？如果我们不再需要软件工程师，那这些工作由谁来做？我觉得，我们依然需要软件工程师。

历史其实一直在重演，这不是我第一次被告知“你的职业要完了”。我干这行也有些年头了，回头看看，比如十五年前DevOps革命、云计算起来的时候，那些天天在机房里插机器、起内核的系统管理员们，现在都涨工资了，而且做的事也更有价值、更开心了。这次也一样，不过是抽象层级更高了点。拖拉机的出现没有终结农业，只是让马和农场工人少了些。产业当然会变，但“种地”这件事还在。

写代码跟写生产级软件不是一回事

如果你没听过vibe coding，我简单解释一下。vibe coding就是完全让AI写代码，人基本不看代码，也不管结构，只关心它是不是能跑起来、是不是做了我想让它做的事。可以跑？那就继续往前推，不用管内部细节，不用编辑检查，直接放行。

但我今天要讲的，不是vibe coding。我要讲的是：怎么写“能跑在生产环境上的代码”。

我说“生产环境”，指的是你要做到99.99%的可用性，这意味着你面对的是成千上万的用户、以GB为单位的数据流。这是支撑整个互联网的软件，靠vibe是搞不定的。

这里有个很大的误解需要澄清：写代码本身不是软件工程师的“本职工作”。就像蓝图不是建筑师的工作本身，它只是工作产出的一部分。

作为软件工程师，代码只是我的“交付物”之一，我其实做的是成千上万个决策：我要构建什么、结构如何、引哪些包、做哪些权衡。所以请大家不要混淆“生成代码”和“软件工程的艺术与技艺”——这完全是两回事。

LLM很擅长生成代码，但那跟“写生产级软件”根本不是一回事。

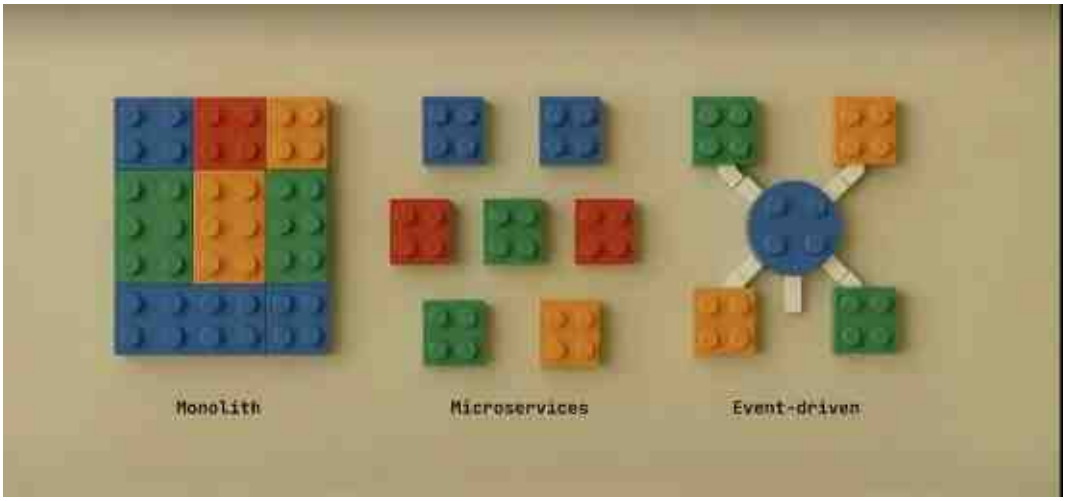
Stack Overflow的创始人Jeff Atwood曾说过一句话：“最好的代码，是不存在的代码。”这话说得对，每一行代码，都是一种负担。我得维护它、调试它。所以每一行AI生成的代码，最后都成了我的责任。

我们过去花太多时间在想“AI可以生成多少代码”。但说真的，这根本不重要。AI

生成得越多，我和我的系统反而负担越大，我们应该让代码越少越好。

所有代码都有权衡：某个包性能更好，但可能更难维护；某种模式可扩展，但可能有副作用。

对比三种架构：单体架构、微服务架构、事件驱动系统。



试想你在每种架构下造一个“航班预订系统”，需要做成千上万个决策。而LLM不做决策，它只会生成“模式”。但在某个规模之上，模式就不再适用了。

有人运行过那种有点像雪花的有“怪癖（idiosyncrasies）”的生产软件吗？那种只有“Bob知道它哪块逻辑是怎么跑的“，或者是“只有Jane能修“，因为是她六年前写的，但她早就离职了，没人敢动那块代码。

那才是真实世界中的软件。当系统足够复杂，所有的“模式匹配”都失效了，因为没有哪个模式能囊括这些独特的细节。

系统半夜崩了怎么办？我这讲的是“我当年背呼叫器”的PTSD——凌晨两点软件挂了，vibes可救不了你，总得有人查出问题在哪。

那什么才是软件工程师真正的“工作”？对我来说，是安全地修改软件。

我这二十年，一直在思考这个问题：如何加新功能、改老代码，而不把系统搞挂，让用户照样能正常使用，让数据安全、业务持续运转。

我为此干了很多事，比如：学习大量代码基础、用版本控制、写单元测试、用类型系统约束、做渐进式部署……

那AI能不能帮我们？它能不能利用上下文，理解更多代码？或许可以。我们在Augment做的工作就是这个。我们认为，上下文才是AI编程里最关键的部分。所以我们相信这个问题是能解决的。

但这不改变一个事实：我仍然得关心生产系统。

怎么写出让AI也能接手的代码？

假设我们认了，未来真的来了，AI写代码成了日常。那问题来了：你怎么写出让AI也能接手的代码？

我在这个圈子混很久了，说点我的观察：职业软件工程师是我见过接受AI最慢的一批人。以前工具一更新，大家冲得比谁都快。版本控制系统换代、云转型——我早年在GitHub，看着这些潮流一波波涌起，开发者都乐此不疲地跟着跑。但AI不一样，现在很多工程师根本不碰，说“这玩意做不了我该做的事”。

为什么？我有一些猜测，但说实话我也没完全搞明白。

几年前，AI编码工具基本上像一堆砖头——勉强能用，但干不了什么事。直到大约一年前，claude sonnet 3.5发布，那是个转折点，质量突然大幅提升。再到四周前，如果

你有关注新闻——几乎所有AI编码工具都一口同声说：“Agent是未来。”这个转变来得非常快。

所以我们要聊聊：在这个新世界里，我们怎么做“软件工程”？怎么构建“对AI友好的代码”？给你们几点建议：

- **规范统一的文档和编码规范**：你们团队到底用哪个包？哪个框架？别人不知道，AI更不知道。写下来，告诉它你们要走哪条路。
- **可复现的开发环境**：你那套开发环境是不是很独特、配置乱七八糟？别这样。
- **可测性强**：测试能本地跑吗？跑得快吗？测试越好写越好跑，AI才能帮得上忙。
- **清晰的功能边界**：明确告诉AI哪些事可以做，哪些别碰。
- **清晰地定义任务和工作内容**：因为AI的表现会和你预期的一样。就像我不会给团队中的任何一位工程师一个模糊的任务，比如：“你能让这个按钮有点不同的效果吗？”但我看到太多工程师在用类似这样的方式对AI提问。

这对我来说很有意思，因为这听起来其实就是软件工程的本质。理想情况下，你的软件工程体系应该具备这些特性；如果不具备，你可能会觉得生产力很差，因为你有自定义的测试基础架构、自定义的开发环境……你需要为AI提供和人类工程师一样的工具，因为它正在做的工作其实是一样的——写代码。

我从来没有一次就写出完美代码的经历。我的代码总是会有错误，我运行测试，它失败了；我有一个代码格式检查器，它会修复它等等。但我们似乎对AI有一个不合理的期望，认为它能写出完美的代码，我不明白为什么。

所以，当你作为软件工程师考虑使用AI时，你必须确保你的系统就像你期望其他工程师那样运作，因为AI也是在写代码。

接下来说的是代码审查，这无疑是最重要的技能。我觉得作为一个行业，我们已经有点遗忘这个技能了。我们可能应该一直在面试中考察代码审查能力，而不是那种深奥的LeetCode问题。我们应该问：“你能阅读别人的代码并评价它的好坏吗？”

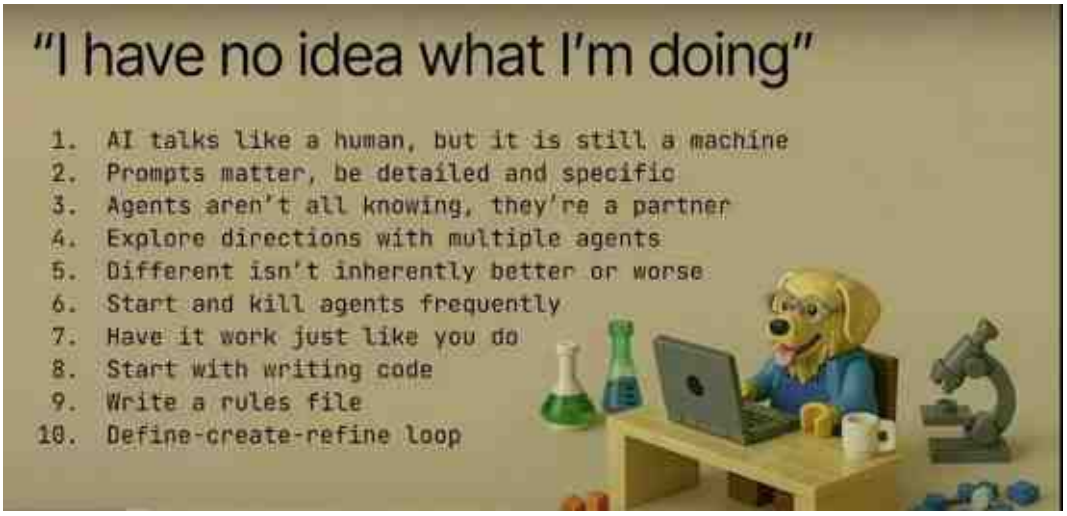
我认为这个能力会变得越来越重要，因为将来会有越来越多的代码由AI代理生成。而我们今天的代码审查工具，说实话非常糟糕。我现在拿到的是一个变更文件列表，它

是按字典序排序的：好吧，文件A改了，那我就看看A文件的改动，再看看B文件的改动。但这并不是理解软件变更的正确方式——按文件顺序思考是没有意义的。

我认为我们会看到代码审查方式的大变革，而这是我们需要在招聘中评估的技能，也是我们自己要重新熟悉的技能。

“我有一些建议”

我想再给大家分享一些关键要点，特别是针对那些不太信任AI、但又想尝试使用AI的工程师。以下是我想给你们的一些建议：



AI说话像人，但其实它是台机器。

我最近和AI有个互动，我在对它“吼”，因为它没完成我想要它做的事情。它回复说：“对不起，我只是略读了那个文件，没有仔细阅读。”我就想，这是什么意思？软件怎么“略读”文件？这在软件领域根本不是个概念。

其实是因为大型语言模型是基于全世界的数据训练出来的，它读过成千上万封邮件，其中有人说：“对不起，我没有认真看你的文档，我只是略读了一下。”于是它学会了：当有人对我生气时，我可以这样回复。

所以我们不能轻信LLM所说的它“正在做”的事情，因为它其实并没有真正做那些

事情。它只是生成文本，它输出的文本不一定代表它实际上做了那些事。阅读LLM输出内容时，请一定记住这一点。

有时候代码只是“不一样”而已。

如果LLM生成的代码和你写的不一样，也没关系。就像坐在你旁边的同事写的代码也可能和你不一样一样，所以接受这一点。如果你想强迫它写出和你风格完全一致的代码，你当然可以付出那份努力。但你需要分辨清楚：这段代码是更好，还是只是风格不同？

学会放手这一点，这正是我们使用代码风格检查器、规则系统和编码规范指南的原因——就是为了不再争论“函数到底应该怎么写才对”。如果可以的话，就放下那些执念吧。

编写规则文件。

告诉AI你希望它怎么做。我每次开始一个项目都会写一个文件，说明我使用的技术栈、希望它遵守的编码规范。这些内容会成为我给LLM输入上下文的一部分。

“定义 - 创建 - 优化”循环。

创建一个文档，让LLM帮你生成它。比如你说“我要做这个功能”，让它帮你写一个Markdown文件，列出完整的计划。保存这个计划作为Markdown文件，并将其作为上下文的一部分。接着让AI根据这个文档创建东西。运行这个代理，根据你的文件执行任务。然后你可以修改它的输出，继续优化它。你可以通过代码补全等方式进行微调，然后不断重复这个循环：先制定计划、再创建、再微调。

这样一来，你不仅能学会如何更有效地提示LLM获取你想要的代码，还能显著提升你的编码效率。而且，只要你愿意放下“代码必须按我那样写”这种执念，只关注“代码是否能正确工作”，你就能变得高效得多。

参考链接

- https://www.youtube.com/watch?v=Dc3qOA9WOnE
- https://www.finalroundai.com/blog/what-ctos-think-about-vibe-coding

Python只是前戏，JVM才是正餐！Eclipse开源新方案，在K8s上不换栈搞定Agent

编译 Tina

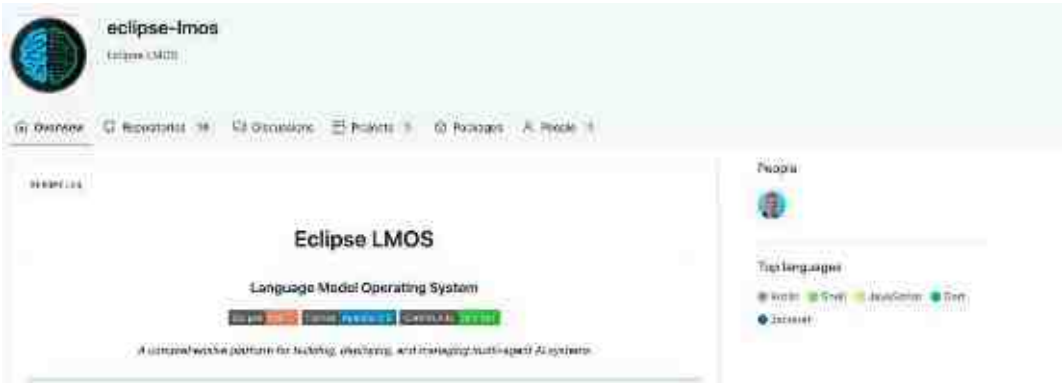
近期，Eclipse基金会宣布在其开源平台Eclipse LMOS中推出“代理定义语言”（ADL）。这是一种结构化、与模型无关的描述方式，允许用户无需编写代码即可定义AI行为。

据Eclipse表示，ADL将成为智能体计算平台LMOS的核心组件。LMOS这个项目从一开始就瞄准在Kubernetes/Istio上原生运行，服务JVM生态，旨在用统一、开放的方式重构企业级AI代理的开发与运维链路。同时也对标专有平台与以Python为主的企业AI技术栈；这也意味着对长期主导企业AI的闭源替代方案发起正面挑战。

值得注意的是，LMOS这个项目采取“先落地、后开源”的路径：其前身是德国电

信在传统云原生架构中的生产级实践，之后才在Eclipse基金会中完成孵化。

开源地址：https://github.com/eclipse-lmos

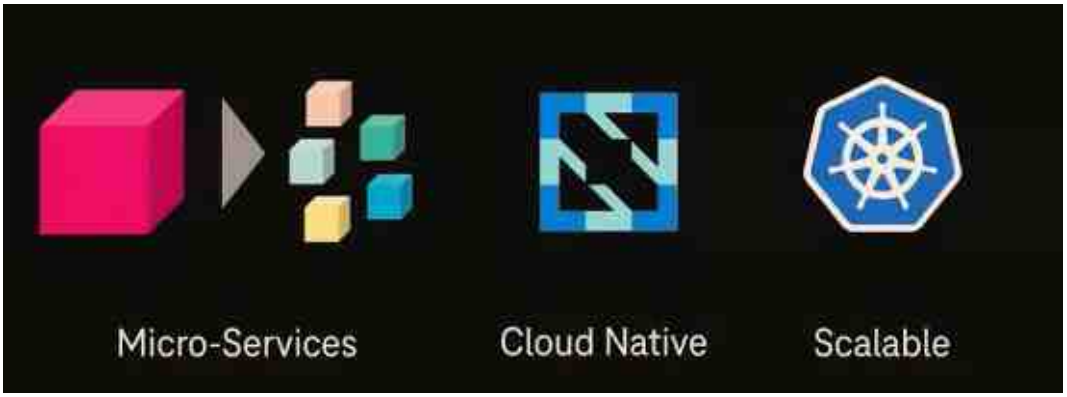


技术收敛：在熟悉的技能栈上做AI

过去十年，企业在云端应用上已形成一套行之有效的工程范式：不再编写单体应用，而是尽量拆分为微服务，让不同团队负责不同部分，并且便于装配组合。同时，将一切打包进容器，方便跨环境迁移和部署；当流量上升时，也能通过复制实例来横向扩展，让多副本共同承载请求与计算。这大致就是当下大家在云上构建应用的常态实践，运行得相当不错。

然后，生成式AI来了。过去两年，大家似乎总被迫“学习一堆新技能”才能用上新技术。回溯到2023年，GitHub上新框架不断涌现，很多人都在犹豫是否要跟进（当时Spring AI尚未出现、LangChain4j也刚冒头）。但实际上大家的头号挑战始终是：如何提高开发速度，以及如何高效利用现有资源。

面对既有的技术生态，特别是对于深耕JVM体系的企业来说，实际上没有必要“推倒重来”再组建一支昂贵的新队伍。因此，LMOS项目的初衷，就是探索如何将AI能力尽量贴近我们已经熟悉的技能栈，而不是迫使企业抛弃既有成果，写Python脚本、从零再来一套。



我们不能丢掉过去十年的经验，再去“组一支新队伍、换一整套新堆栈”

LMOS项目与这些背景相关。该项目由负责人Arun Joseph在2023年主导设计，当时他负责了德国电信的一个AI项目，需要在10个欧洲国家上线面向销售与客服的AI能力。

“我们刚起步的时候，虽然涌现出了一些框架，比如LangChain，但它们都是用Python编写的，”Joseph解释道，“但和大多数电信公司和企业一样，德国电信的整个企业级技术栈都是基于JVM构建的。”

另外，抛开语言差异，德电一线业务的复杂度本身就很高：同一套系统要跨多国部署，还需可插拔地接入不同渠道（Web、App、热线等）；同时API体系庞杂，客户端库包含数百个属性与多年沉淀的领域知识。若改以Python重新起步，等于放弃既有资产，逼迫团队重建系统。

而且，要在10个国家运营，技术架构必须支持“平台化、集中式”的统一部署与管理。基于这些考量，团队决定在熟悉的生态内演进：采用Kotlin，复用既有基础设施、API与DevOps能力，搭建多智能体平台。

平台以Kubernetes为底座，配合Istio等组件提供的能力，将“代理/工具”以微服务形态部署到K8s环境，并通过自定义资源（CRD）提升为一等公民，支持声明式管理与可观测性。

通过这种方式，开发者能沿用既有工作流：只需推一个智能体镜像，接下来的操作就能在一个新环境里把它跑起来，独立测试；运维团队可以直接用kubectl get agents、kubectl apply去监控与发布。

所以，在理念上Eclipse LMOS与当下主流AI工具生态分道而行了。Joseph指出，许多“企业级AI技术栈”往往是一个Python代码库拼接多家风投支持初创的SDK——各自只解决遥测、记忆、评测等一小块问题——再用一个装饰器（decorator）把整片容器编队塞进基础设施。“我见过某些评测工具，为了一个函数就要25个容器，”他说，“也就是说，为了一行代码，让25个容器跑一个自定义的Kubernetes Operator。企业承受不起这样的无序膨胀。”

Eclipse LMOS的方式避免了这些问题：原生融入Kubernetes、Istio与基于JVM的应用，顺畅对接组织多年建设的DevOps流程、可观测性工具与API库，让AI代理以最低迁移成本进入生产系统。

这些平台已支撑德国电信的多项AI应用，其中包括多次获奖的客服机器人Frag Magenta。到2023年底，首个代理在德国电信投入生产，并在欧洲加速扩展：覆盖范围从3 – 4个国家增至10个国家，上线后月均处理约450万次会话；到2024年，转人工次数下降38%。于是这成为欧洲最大规模之一的、真正投入生产的Agentic系统。

沿用熟悉的技术栈，开发周期也因此得以压缩：“最初做第一个代理（同时还在搭平台）花了一个月；随后在少量工程师参与下把周期降到15天；再往后，基本一两天就能把一个代理连同一组用例做出来。”

“经过一段时间的度量我们发现，只需一名数据科学家与一名工程师配对，从业务

提出想法或问题陈述到将代理部署到生产都能非常快。进入维护阶段后仍可快速迭代，小团队也带来明显的成本优势。我们没有在一开始就纠结于是否提前扩招AI/ML工程师或数据科学家——**2023、2024年许多团队都把时间浪费在这类取舍上。**”

随后，在2024年，团队将德国电信内部的专有代码迁移至Eclipse基金会开源，项目采用Apache-2.0许可证进行发布。

如何将经典领域的精华融合到一个平台中

在将AI代理真正推向企业生产环境的过程中，Eclipse选择了一条“双线策略”。一

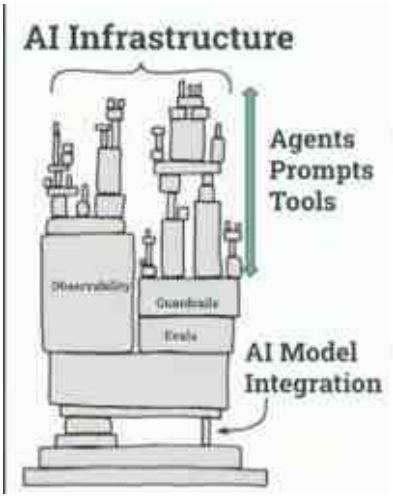
条线是**LMOS平台**（此前已完全开源）。另一条线更具突破性：**ADL（Agent Definition Language）**，“让更多人能真正写出代理”。

“我们对当时的框架集体‘不满意’，于是走了‘激进简化’路线：选了Kotlin作为主语言，因为它便于我们打造领域专用语言（DSL）——也就是ADL。”

“将业务背景融入人工智能工作流程和应用程序至关重要，这样它们才能大规模地做出高质量的决策，**自然语言提示无法进行版本控制或审计**——这是企业面临的痛点，也是编程语言存在的原因——所以这种方法可以满足介于两者之间的需求。”

这种路线也来自实践中的痛点。在现实的AI基础设施里，Agent、提示词（prompts）、工具（tools）层层叠加，复杂度迅速膨胀，同一套模式在不同场景被反复造轮子。更关键的是，系统本质上仍是自然语言驱动——大量的条件、规则与SOP需要压在模型之上去执行。有意思的是，市场上已经被这些产品淹没了。单拎出来看，它们都很棒；但拼成一套完整系统就很难彼此“对话”与“粘合”。

而且当系统一旦扩大，AI模型集成会变得极其微妙。模型轻微升级或细节偏差，上层行为就会被牵一发而动全身导致整体脆弱（如下图所示）。

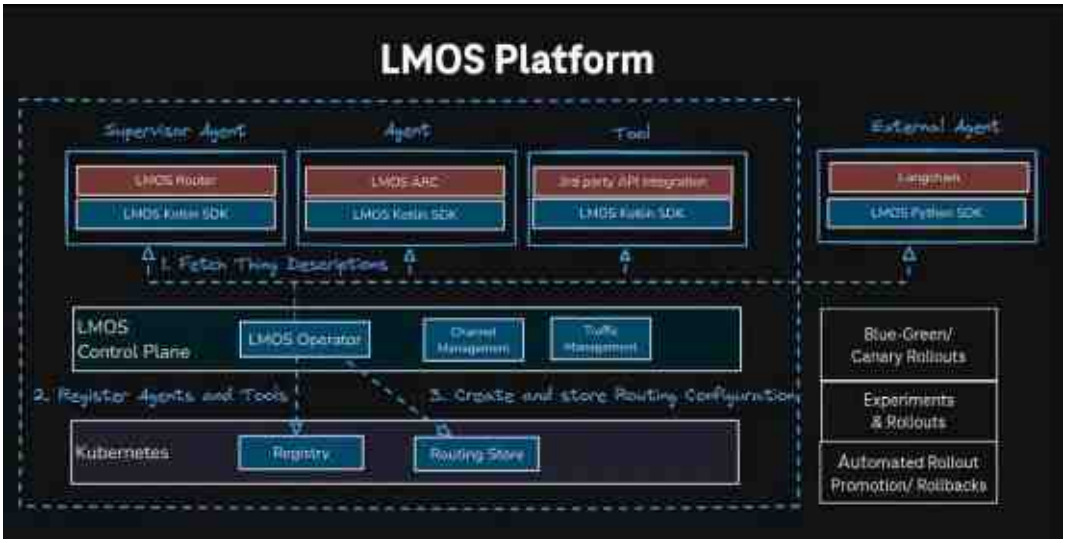


所以，这个平台最终凝结为三个彼此独立、又能够协同运作的模块：

- **ADL**：结构化、模型无关，支持可视化创作与多角色协作，让业务与工程团队共写代理。。它让业务部门能够像写SOP一样定义代理行为，立刻测试、立刻迭代，无需等工程工单排期。与此同时ADL仍保持工程严谨性：行为可版本化、可追踪、可维护，这也是它替代传统提示词工程的关键。
- **ARC Agent Framework**：基于JVM/Kotlin，提供IDE级开发体验与可视化调试，让工程师专注业务API与集成逻辑，而不是搭底层。
- **LMOS平台层**：开放的云原生编排层，用于代理生命周期管理、发现、语义路由和可观测性。它基于云原生计算基金会（CNCF）技术栈构建，目前处于alpha版本。

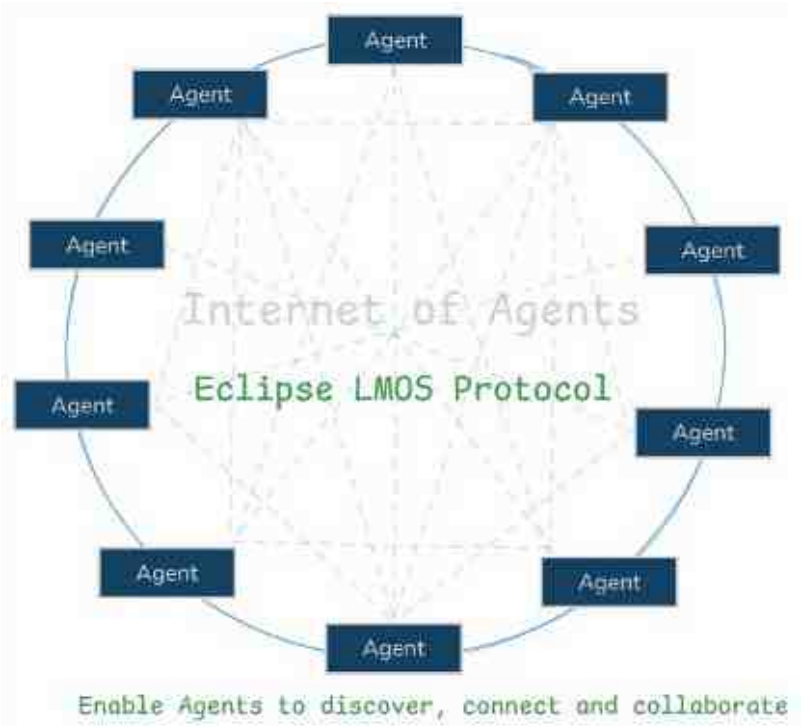
截图来源：https://eclipse.dev/lmos/docs/lmos_platform/overview/

围绕上述模块，LMOS底层还提供了与运行环境紧密集成的控制平面（control plane）：当一个agent或tool以微服务形态部署到Kubernetes后，LMOS Operator负责生命周期管理。每当新应用安装，Operator会接收事件通知，抓取其**Description**（描述文档），并写入Kubernetes Registry（当前用作存储机制）；同时Operator还提供了来自Web of Things的Directory API便于发现当前在Kubernetes环境中安装了哪些agents。



其中，还值得一提的是LMOS协议。

考虑到每个代理都有独立生命周期，系统需要一种机制让代理彼此发现并协商通信协议。LMOS的设计借鉴了W3C成熟标准，并从Matter/Thread与Bluesky的AT Protocol等去中心化技术中汲取灵感。其目标在于像互联网连接计算机、物联网连接设备那样，让平台连接跨组织边界的AI智能体，实现可发现、可互操作的代理网络。



Eclipse团队回忆，早期他们在没有MCP的前提下就把东西做起来了：既然熟悉内部API，就先压缩出一套“收敛版”接口用于函数调用，而不是再搭一台新服务器、引入一堆新概念。他们还在MCP之前就实现了“工具注册表”：只需填入工具名即可自动注册，让模型清楚“该用哪个工具”。此后MCP走红、Google又推出A2A。

但这并不意味着LMOS这样的路径没有空间——毕竟在“代理大多运行于云、而云标准已成型”的现实里，要把代理与现有云原生栈真正对上齿轮，反而需要像LMOS这样贴近工程现场的协议与平台。当然，面对MCP与A2A标准更新快、落地也快的节奏，这对Eclipse基金会而言将是一场硬仗。

写在最后

在生成式AI席卷企业软件的当下，技术世界正在形成一条看不见的断层线：一边是敏捷、开源、快速原型迭代的Python阵营，主导着创新型初创公司；另一边是庞大而成熟的JVM世界，以稳健与可控为前提，承载着大多数企业的关键系统。

Eclipse LMOS站在这条缝隙之间，试图做一件看似朴素、实则颇具野心的事：把AI代理带回企业能理解、能运维、能长期托管的基础设施之上。你不需要“推倒重来”、重新组建一支只会写Python的团队，也不必在提示词与外部工具间疲于奔命；你完全可以从Eclipse LMOS和ADL这样的入口出发，再按需把系统复杂度一点点提升到Kubernetes等成熟栈之中。

“Agentic AI正在重新定义企业软件，但迄今仍缺乏可真正替代专有产品的开源软件，”Eclipse基金会执行董事Mike Milinkovich曾在一份声明中表示，“借助Eclipse LMOS和ADL，我们正在提供一个强大的开放平台，任何组织都可以据此构建可扩展、智能且透明的代理系统。”

LMOS和ADL所做的，是将智能体在你已经信任的技术之上自然生长——这一点，也许正是“代理时代”最值得期待的转折。

参考链接

- https://thenewstack.io/eclipse-opens-up-enterprise-ai-agent-development-with-adl/
- https://www.youtube.com/watch?v=gcpHYtqTBbA
- https://www.infoq.com/presentations/ai-agents-platform/
- https://www.youtube.com/watch?v=gHqP5Hx9gbU

LangChain彻底重写：从开源副业到独角兽，一次“核心迁移”干到12.5亿估值

作者 Tina

> 重写LangChain之后，Agent开发终于告别“拼凑学”。

本周，LangChain宣布完成1.25亿美元融资，投后估值12.5亿美元。除了宣布其独角兽地位外，该公司还发布了里程碑式更新：经过3年迭代，LangChain 1.0正式登场。而且，这并非一次常规的版本升级，而是一场从零开始的重写。

LangChain是开源开发者社区中最受欢迎的项目之一，其每月下载量高达8000万次，数百万开发者正在使用，目前在GitHub上拥有11.8万颗star和1.94万个分支。要对这样一个普及度极高的框架进行全面“重写”，这个决策难度可想而知。

任职于LangChain的Julia Schottenstein发帖：

> LangChain从零开始重写——现在更加精简、更灵活、更强大。各方面都大幅提升。**要下决心重写这样一个已经如此普及的框架，绝非易事。**现在的LangChain围绕循环内的工具调用Agent架构构建，模型无关性是其核心优势之一。

从副业开始的项目

LangChain于2022年10月左右由机器学习工程师Harrison Chase发起。最初是Harrison的一个副业，当时他大约写了800行代码，是一个体量不大的单文件Python包，于同年秋季发布到了他个人的GitHub账户（hwchase17）上。

Harrison自述，项目的灵感来源于一次特殊的时期：大约在Stable Diffusion发布之后、ChatGPT问世之前的一个月里。他频繁参加各种聚会和技术活动，结识了许多利用大模型进行前沿探索的人士。

面对工具碎片化和抽象不足的状况，通过交流，他大致梳理了大家构建项目的一些共性，并意识到：“将这些共同点抽象分解出来，会是一个很酷的副项目。”后来，这个副项目就发展成了今天的LangChain。

最开始的版本包含三个端到端的模块。一个是NatBot，这是Nat Friedman开发的Web代理。另一个是LLM Math Chain。第三个是Self-Ask，是一种RAG搜索，类似于React风格的代理。该框架的作用主要是把模型与工具连接起来，将多个调用与业务逻辑串联成链

条，解决了早期大模型开发应用时的一些痛点，例如网页搜索、API调用以及数据库交互等。

发布后，他持续迭代，接入了各种LLM和向量数据库等更多集成，还提供了更多高级的“模板”，使用户仅用5行代码就能开始使用RAG、SQL问答、提取等功能。

随着大模型的火热发展，LangChain很快成为增长最快的开源项目之一。Harrison于2023年4月以Benchmark领投的1000万美元种子轮正式创立公司。一周后，他又完成了由红杉领投的2500万美元A轮融资，当时LangChain的估值据称已达2亿美元。

从“拼装学”到工程化：一个大模型框架的进化

即使是创始人Harrison本人，现在也很难准确一句话概括LangChain到底是什么。在一次播客中，他曾这样定义：“LangChain是一个构建LLM应用程序的框架，但这说法很模糊，也不够具体。我认为部分原因在于LangChain的功能实在太丰富了，所以很难用一句话具体说明。”

但就优先级与侧重点而言，LangChain的核心是一个“情境感知的推理型应用框架”。它大体包含两层：其一是组件与模块层，覆盖提示模板、LLM与聊天模型抽象、向量存储抽象、文本分割器与文档加载器等。LangChain本身不提供自有的大模型或向量库，但整合了大量外部实现；文本分割器与文档加载器等则有自有逻辑，均可独立组合。其二是端到端的链与应用层，例如文档问答、聊天问答、SQL问答与“即插即用”的代理等，把前述组件按既定流程装配，帮助开发者用少量代码快速完成从检索嵌入、合成提示，到生成、解析并后处理答案的全流程。

在整合生态这件事上，团队长期坚持“模型与基础设施中立”的路线。外界曾评价他们“就像开发者领域的瑞士”：比如说那时市面上有众多向量数据库，彼此都在竞争、争夺嵌入与数据归属，而LangChain的态度是“我们都支持”。

Harrison认为，他们当初最坚定的信念是，这个领域还处于非常早期的阶段，而且发展非常迅速。因此，关于向量数据库将扮演什么角色、会有多少个向量数据库，未来发展如何，都存在很多不确定性。因此他认为“可选性和可切换性是必须的，LangChain不能被任何技术路线束缚。”

除了主流大模型、80种向量数据库，对于“文本分割器（Text splitters）”这类的组件，Harrison曾表示：“我记得我们有大概15种，我还觉得它们的数量都被低估了。”

据2024年10月的数据显示，LangChain里有总计超过700个不同的集成。这其中包括10大类组件，每一类里通常有30到100个集成。这个领域有非常多不同的技术触点，而LangChain的定位，就是连接这些触点的“粘合剂”。而且他们还提供了Python和TypeScript两种版本。

这背后是大量艰苦的适配与集成工作——尤其是在团队仅约10人的阶段（2023年7月的时候团队只有6个人，随后两个月增加到10人上下），仍投入了可观的时间去打通各种不同的组件，以保证开发者随取随用、自由组合。

然而，在这种高速集成的情况下，项目也积攒了不少问题，Harrison透露：“（那时我们）有大约2,500个未解决的问题或类似的（Bug），还有300或400个待处理的PR。”

到了2023年夏天，LangChain团队开始接收到大量负面反馈。其中有些问题尚能修复，比如防止破坏性更改、显式化隐藏的提示词、解决安装包臃肿、依赖冲突以及文档过时等。

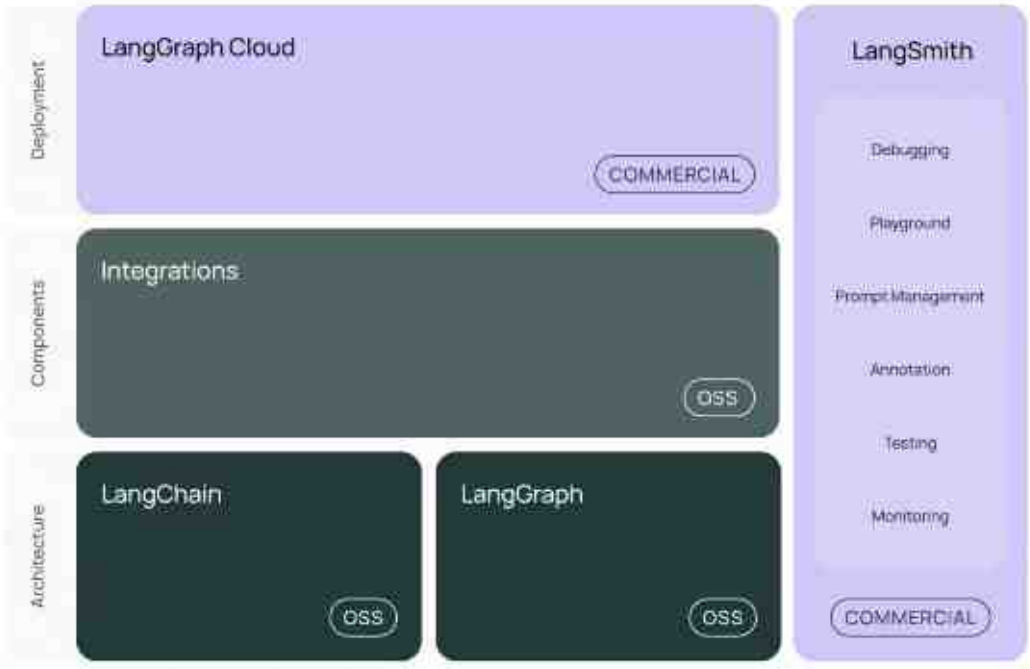
然而，有一项反馈更难处理——用户希望获得更高的控制权。虽然LangChain一直是开发者快速入门的最佳选择，但团队为此牺牲了部分定制化能力以换取易用性。最初，LangChain中那些让用户快速上手的高层级接口，现在反而成了开发者试图进行定制化并推向生产环境时的阻碍。

为了解决这一痛点，LangChain团队于同年夏天开始开发LangGraph，并在2024年初正式推出。LangGraph允许开发者以更底层的方式编排每一步智能体逻辑，包括记忆管理、人类在环（human-in-the-loop）以及持久化执行的长任务管理。

例如，Replit在几个月前发布的智能体就是基于LangGraph和LangSmith构建的。值得一提的是，LangSmith是LangChain生态中的一款闭源工具，专注于LLM运维领域，主要提供可观察性和监控功能，同时也是该公司的主要收入来源。

目前，LangChain公司有三条主要产品线，工作侧重点各有不同。在LangChain开源方面，核心工作是生态系统的规模化管理，需要与大量的合作伙伴进行协作。在

LangGraph方面，当前许多工作聚焦于可扩展性，以及智能体的集成开发环境（IDE）与调试能力的提升。而在LangSmith方面，随着承载的生产工作负载持续增加，团队会继续致力于推进其可扩展性。



截图来源：https://js.langchain.com/docs/introduction/

重写LangChain

LangChain在三年前发布时，其功能主要围绕着各种集成和高层接口展开。当时集成范围还比较有限，模型端主要集中在OpenAI、Cohere和Hugging Face这几家，而围绕模型、向量库、文档加载器等核心积木，LangChain都提供了相应的组件与集成。另一块是高层接口，它让用户可以非常容易地上手，比如实现RAG或SQL问答只需短短五行代码。

团队当时判断，行业所处阶段决定了LangChain的首要目标：让开发者“尽快用起来”。随着生态成熟、原型走向生产，关注点随之转移。

以发布一年半后的LangGraph为例，它瞄准了两个核心：**可控性**与内建的**运行时**。

其一，在生产环境中，开发者需要更强的自定义与边界控制，“五行代码做RAG”

背后隐含了大量默认前提（如隐藏提示与所谓“上下文工程”），有利于入门，却不利于深度定制与规模化稳定。

其二，运行时层面，团队沉淀出三项关键设计：可持久的执行环境（局部出错不致整次作废）、检查点恢复（运行中缓存应用状态以便回溯与重启）、以及将流式交互作为一等公民（支持长任务的实时进度与互动）。

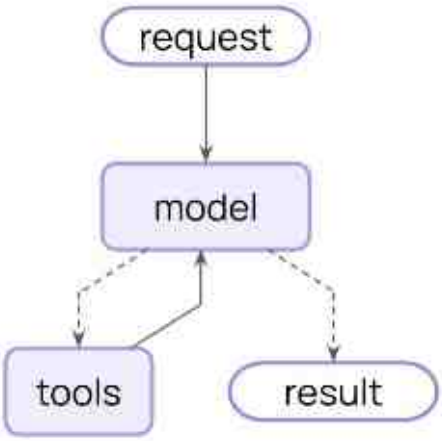
当年团队在构建LangChain时，像流式输出、人类在环这类能力并非一开始就具备，而是后来“回填”进去的。

几个月前，他们开始重新审视LangChain，并决定全面重写。

这一重大决策也深受LangGraph开发过程的影响：在LangGraph的开发过程中，团队曾反思为何某些在原型期易于实现的功能，在生产环境中却难以落地。基于此，他们从架构层面提出核心要求，即运行时必须原生支持“人类在环”等关键能力——这也正是LangGraph的核心目标之一。

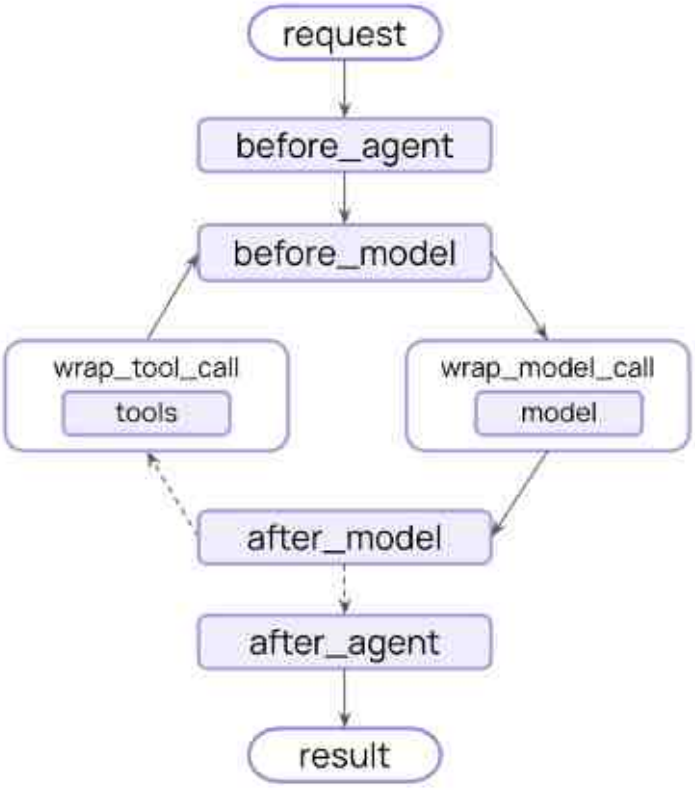
这些经验与要求，直接奠定了重写的技术方向：团队最终决定以LangGraph为底座，对LangChain进行彻底的架构重构。

两年来，整个行业里已经冒出了一堆围绕“智能体构建”的框架，但基本收敛在“智能体=工具调用循环（tool-calling loop）”这个概念上。因此新版LangChain 1.0为平衡“强可控性”与“低门槛”，提供了统一的**create_agent抽象**，让开发者用少量代码即可起步，快速搭建经典的“模型—工具调用”循环。



该抽象既延续了早期LangChain 0.0.8的“chat agent executor”范式，也吸收了LangGraph社区广泛使用的“create_react_agent”经验，在一致性与易用性上进一步收敛。

此外，LangChain 1.0还添加了一个新的**中间件**概念，这也是此次重写的关键抓手。



这种中间件模式允许开发者在“核心智能体循环”的任意位置插入额外逻辑，使主循环更易扩展到各类场景：例如，它可以在模型调用前自动总结历史对话，以支持长时运行带来的超长上下文；或者在高风险或高成本的工具调用处，通过添加Hook（钩子）的方式，启用人类在环（Human-in-the-loop）审批。

LangChain团队表示：“我们非常认可在整个智能体循环周围添加钩子的这种范式。因此，我们也让大家非常容易去构建自定义的中间件。”团队进一步指出，如果想要定制动态提示词（dynamic prompt）或动态工具（dynamic tools），同样可以通过中间件来实现。“我们认为，这正是将LangChain作为一个智能体框架区别开来的地方：中间件中内建的可定制化能力。”

而且，借助**动态模型中间件（dynamic model middleware）**，系统还可以基于上下文动态选择要使用的模型，从而将“模型无关性”落到工程实处。

如今已非“单一模型通吃”的时代：写代码去找Anthropic，推理（reasoning）去找OpenAI，多模态去找Google。某种意义上，每家模型都显现出了自己的专长领域。

团队强调，“在我们与开发者交流时，模型之间的可选性本来就是LangChain的初心之一。能够启用这种可选性，才是你在智能体构建上保持最前沿的方式。”中间件让系统可在对话生成或下一步循环中按需切换模型，于能力与成本之间取得更优解。

“**create_agent + middleware API**是把‘上下文工程’真正落地为可操作抽象的首次清晰尝试。”

在产品层面，**LangChain 1.0**做了四项系统升级：

- 引入更规范的**content blocks**，在不同模型间统一输入/输出结构；
- **精简代理选项**，仅保留经过官方打磨与背书的实现，降低选择与调参成本；
- 提出**middleware**以强化可控性与“上下文工程”；
- **全面运行在LangGraph runtime之上**，原生支持持久化、检查点恢复、人类在环与有状态交互等生产级需求。

整体而言，LangChain 1.0在延续“易上手”的同时，将LangGraph的生产级能力下沉到框架层：更标准的内容结构、更克制的代理组合、更可编排的中间件体系，以及对人

类在环与有状态交互的原生支持，构成了这次重写的核心价值。

参考链接

- https://blog.langchain.com/three-years-langchain/
- https://blog.langchain.com/langchain-langgraph-1dot0/
- https://www.youtube.com/watch?v=NxrMgMBuxao
- https://www.latent.space/p/langchain
- https://www.infoq.cn/news/WHkGx30RJzICVXWNXBNo
- https://www.youtube.com/watch?v=r5Z_gYZb4Ns

将AI带入数据！Oracle给数据库内嵌上Agent框架

作者 褚杏娟

“Oracle如今越来越开放。”甲骨文公司副总裁及中国区董事总经理吴承杨说道。其全球大会名称从“Oracle CloudWorld”更改为“Oracle AI World”，清晰地宣告了公司的核心方向：AI将改变一切，并将在未来多年成为整个IT产业的基石。

MIT报告显示，95%的企业生成式AI项目未能带来显著影响。吴承杨指出，当前许多企业AI项目未能产生预期收益，其根本原因在于理念错误。企业往往将数据导出，作为一个独立的“AI项目”来运作，这被比喻为“将AI围绕数据运行”。这种割裂的方式导致了效率低下和收益不明。

因此，Oracle提出了一个新的理念：AI不应是一个独立项目，而应是一种内生于数

据的能力。这意味着，AI功能被直接嵌入到数据本身。当数据“流”到企业的任何地方时，AI能力也随之抵达，无处不在。这不是将数据带入AI，而是将AI带入数据。

基于这一理念，Oracle构建了多层次的AI架构：

- 架构层，集成AI算力与基础设施。
- 数据层，以Oracle Database 26ai为代表，实现AI内生于数据库。
- 应用层，演进至智能体时代，提供自然交互。

“Oracle凭借其完整的技术堆栈，用‘数据血液’将AI能力贯穿所有层面。这确保了企业无需为了引入AI而牺牲其固有的核心能力，如安全性、隐私性、可扩展性和快速部署能力。”吴承杨表示。

云和数据库产品创新

成立近五十年、入华三十六年，吴承杨表示，Oracle如今依然生机勃勃。其“新”体现在两个方面：一是由两位新任CEO引领的新时代；二是一系列新产品与新功能的推出。

云基础设施OCI“内置AI”

产品方面，其云基础设施OCI（Oracle Cloud Infrastructure）是承载“内置AI”理念的技术基石。OCI的价值定位在于：它是一个从零开始构建、软硬件集成设计的云平台，其核心设计目标是更高性能，更低的成本，更安全的基础架构。

甲骨文公司中国区云工程部门总经理窦杰称，近年来，为应对AI工作负载的爆发，OCI在AI基础架构上投入巨大，最新发布的OCI Zettascale10集群可支持高达80万个GPU，被称为全球最大的AI集群之一。

OCI的演进具体体现在四个关键方向：

- 性能引擎：Oracle Acceleron解决方案，一个整合了硬件加速与零信任安全模型的云网络功能，旨在为整个OCI基础架构提供更高的吞吐量、更强的隔离性和更优的成本结构，是OCI未来的“性能引擎”。

- 专属部署：Dedicated Region 25。如果对数据驻留、安全和合规有严苛要求，OCI提供可部署在客户指定数据中心的“专属区域”。客户能以最小的资源占用，获得与公有云完全一致的功能与体验，实现“云的就地部署”。
- 开放互联：多云战略。其多云战略的核心优势是免除出向流量费，即客户在不同云服务商或数据中心之间迁移数据时无需支付费用，这为企业带来了巨大的成本节约。通过Multicloud Universal Credits，客户可以一份合同，灵活使用包括OCI在内的多家云服务。
- 人工智能。OCI致力于提供端到端的AI体验。客户可以在OCI上便捷地使用包括Grok、Gemini、Llama、Cohere及OpenAI开源模型在内的众多前沿大模型，实现“一个平台，多种模型”的一站式使用。OCI“一站式商店”理念还体现在，通过提供托管的RAG智能体、SQL智能体、代码智能体等，兼容LangChain等主流开源框架，同时与Oracle自身的应用产品线（如Fusion, NetSuite）无缝集成。

深度集成AI的26ai数据库

数据库层面的核心产品代表是Oracle AI Database 26ai。甲骨文公司中国区技术工程部总经理嵇小峰介绍，26ai的发布标志着AI技术与数据技术的深度融合。

据介绍，26ai的创新主要体现在三个层面：

- 数据库核心内嵌了Select AI Agent框架，这是一个支持ReAct模式的智能体运行时环境，意味着部分业务逻辑和AI智能体可以直接在数据库内运行，而不仅仅将数据库视为持久化存储，极大提升处理效率。
- 重新打造了从数据到AI应用的整个开发链路。为解决大模型不理解企业私有术语的问题，Oracle推出了结构化的Annotation功能，它采用键值对形式，集中管理数据语义，并存入数据字典，极大方便了后续的AI应用开发。对于存量系统，AI Enrichment功能可自动对现有Schema进行采样并生成标注；此外还提供了自动化工具和优化器，为企业推荐最佳实践参数。
- 全面拥抱开放标准：Apache Iceberg。Iceberg充当统一的“数据目录”，使Oracle能够无缝访问和集成位于任何地方的数据。

此外，Oracle发布AI Data Platform，可以整合企业所有数据（包括私有数据）用于AI

应用。底层采用开放的湖仓一体化基础；中间层集成了OCI上的多种大模型及Spark、Flink等开源框架；顶层提供统一的AI开发工作台（AI Data Platform workbench），支持数据科学、分析和智能体开发，并内置数据可视化、工作流和自然语言交互等开箱即用功能。

Oracle和Dify的合作

作为Oracle开放生态的例证，其与开源AI应用开发平台Dify的合作备受关注。双方合作的具体形式包括：

- 市场入驻，Dify将在Oracle Cloud Marketplace上架，便于用户一键部署于OCI。
- 深度技术集成，26ai（包括其结构化与向量化能力）将与Dify引擎实现完美兼容。
- 能力扩展，Dify将通过支持MCP插件，无缝连接并利用各类PaaS服务能力。

嵇小峰强调，Oracle所做的并不仅仅是一个宽泛的“智能体基础设施”，而是专注于AI应用的全生命周期，并充分发挥其在“数据”层面的核心优势。面对实现方式繁多的智能体技术，Oracle采取开放策略，通过与Dify合作，共同加速企业级应用落地。

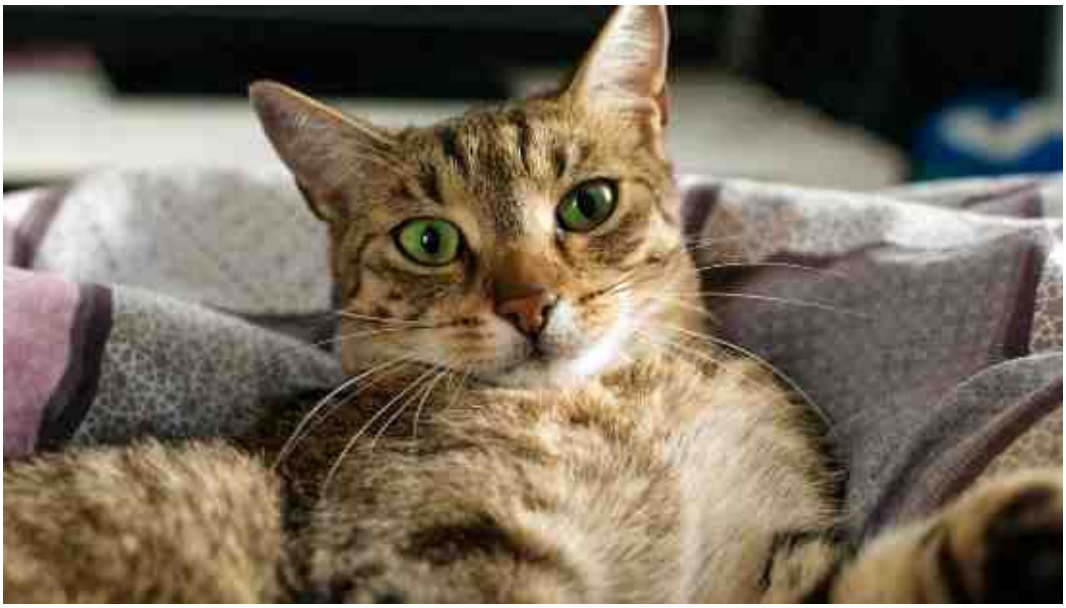
Dify创始人张路宇认为，在技术飞速迭代、新产品层出不穷的当下，合作是必然趋势。不同的厂商对技术路线的“风险偏好”不同，而Dify更看重可扩展性与可靠性，这正是Oracle能够提供的核心价值。

他进一步指出，当前市场充斥着大量吸引眼球但未必可靠的“噪音”。在难以分辨真伪的环境中，客户的选择策略趋于谨慎：当他们选定一个信任的“主厨”（如Oracle）后，自然会信赖其推荐的“食材”生态（如Dify）。这种“信任转移”为企业客户提供了一个经过筛选和验证的、风险更低的整合方案。

嵇小峰表示，对于企业客户而言，选择Oracle的技术路线，再结合Dify等的灵活性与创新，最终能够实现两大关键价值：一是“放心”，技术栈成熟可靠；二是“快”，落地路径清晰，能有效规避选型风险，加速实现业务价值。

弃Python拥抱JVM，Spring之父20年后再造“革命性框架”：我从未如此确信一个新项目的必要性

作者 Tina

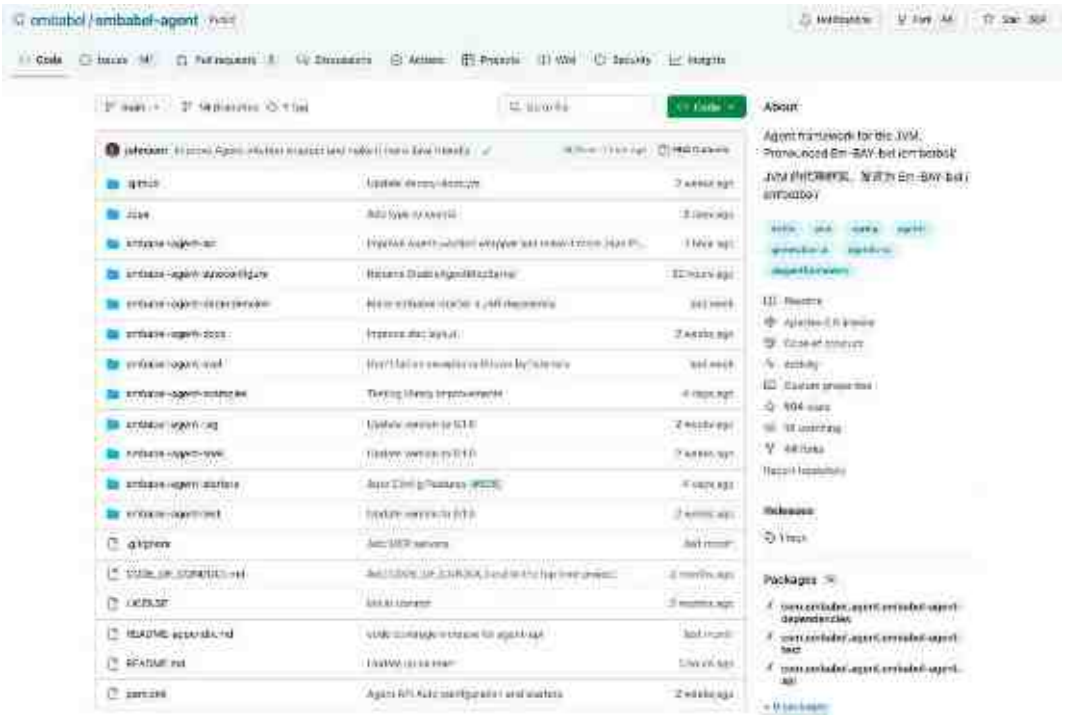


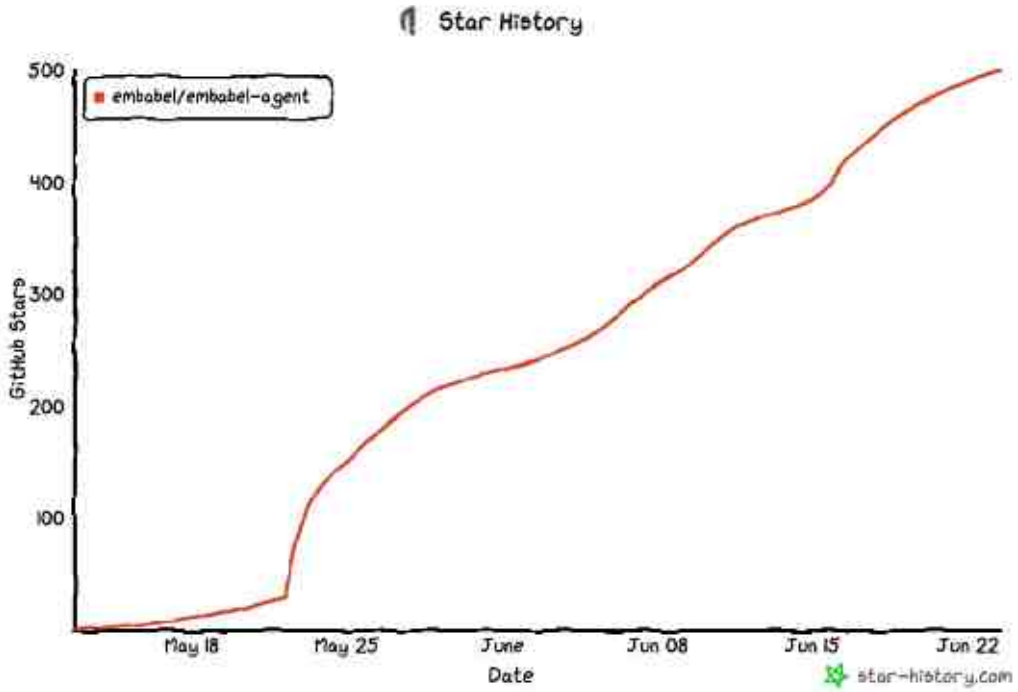
在Java世界，Rod Johnson是一个无法忽视的名字，作为Spring框架的缔造者，他曾深刻改变了企业级Java应用的开发方式。

如今，随着生成式AI席卷各行各业，软件开发范式正面临新一轮重构。与此同时，Python凭借其强大的生态系统，在这场AI浪潮中占据了主导地位。而此时，Johnson再次站了出来。他亲手打造并开源了一个全新的项目——Embabel，这是一个专为JVM生态设计的AI智能体（Agent）框架，旨在为企业级AI应用提供坚实基础，弥合生成式AI的“承诺”与“现实”之间的巨大鸿沟。

“自从我创立Spring框架以来，我从未如此坚定地相信一个新项目的必要性。”Johnson表示。“我真心认为，Embabel有潜力成为JVM世界中的一个关键项目。我不仅希望它追上Python生态，更希望借此构建最强的Agent模型，释放真正的商业价值。”

作为一个开源项目，Embabel和Spring一样采用的是Apache 2.0协议，目前已在GitHub上公开：https://github.com/embabel/embabel-agent





Spring之父为何重构AI应用基础?

> “九成可用”相当于“完全没用”，GenAI不该止步于惊艳。

Rod Johnson不仅是Spring创始人，更是推动现代企业级Java编程范式变革的关键人物。Spring框架的诞生颇具传奇色彩，它最初源于Johnson撰写的一本书——《Expert One-on-One J2EE Design and Development》。这本书传递了一个明确的信息：“亲爱的Java企业开发者们，你们的痛苦到此为止了，你们再也不需要EJB了。”

当时的EJB开发流程繁复至极，仅实现一个Hello World就需要准备多个XML文件和类接口。所以Spring一出现就彻底改变了Java企业开发，并迅速成为了Java生态中最成功、最持久的开源项目之一。

Spring作为开源框架已走过二十余年，依然活跃在全球无数企业的系统中，而Johnson也始终没有远离这个社区。他最新关注的技术焦点，则是生成式AI。

“我认为，这是我们这个行业自互联网以来最重要的技术，”Johnson在一次访谈中表示，“它不是一个可以忽视的东西，我必须深入理解它，也要尽我所能让它变得更

有用。”

但生成式AI这项技术的“承诺”和现实之间存在着巨大的差距。他回忆起两三年前第一次使用ChatGPT时的震撼：“你居然可以和这个东西对话！”然而，几年过去了，**我们在企业场景中其实并没有看到太多真正的成功案例。**

“生成式AI在某些场景表现出色，比如个人助理。大约90%的回答都很不错，但10%完全不靠谱。”Johnson强调，对于企业应用而言，这“10%的错误”意味着系统的不可信，甚至是无法使用。“你实际上是在用一个‘九成时间有效、一成时间无效’的东西去构建系统，从效果上看，它就等于‘完全没用’。因为业务应用容不得这10%的错误。”

不仅如此，它“10%的失败”还是设计上的必然结果。生成式AI是一种随机的、非确定性的系统，它给同一个提示生成的结果也会有所不同。这对应用开发者提出了新的挑战：“你要处理极其不可靠的系统、高失败率、非确定性行为，甚至还要与自然语言交互。”与此同时，一些20年前就遇到过的经典问题也依然存在——如何访问企业数据？系统记录源（system of record）在哪里？事务如何管理？系统如何扩展？

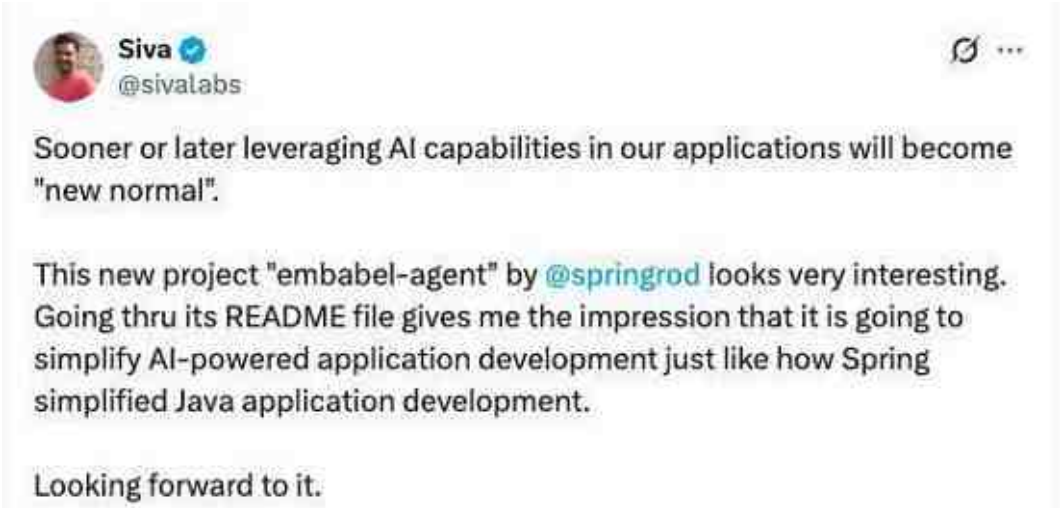
在Johnson看来，解决这一切问题，**JVM社区具备天然优势**。“我觉得JVM和JVM社区，在让这些事情‘变得有用’方面拥有巨大的潜力。”

Johnson的这番判断并非没有实战经验支撑。在过去10到15年里，他使用过包括Scala、Node 、TypeScript和Python在内的多种语言，尤其是在2019年真正开始接触AI领域（不是Gen AI）后，写了大量Python代码。然而，当他认真思考如何真正把生成式AI用在企业系统中时，他发现——“解决这些问题所需要的技能，**JVM社区比Python社区更加具备**。”

尽管Johnson承认“Python是一门很棒的语言，也非常容易上手”，他也坦言：“我不会说我‘热爱’ Python，但我很庆幸自己对它很熟练。”他甚至会忍不住用Python去重写复杂的Bash脚本，让维护变得更轻松。

但说到构建复杂的企业级软件系统，Python并不是一个特别优秀的语言。“比如它对可见性（visibility）的支持几乎没有，泛型（generics）也很原始。”更重要的是，企

业系统通常要集成各种已有的代码资产，而这些资产绝大部分是基于JVM和Java的，而不是Python的。所以，这也让他下定决心：“如果我们要填补GenAI的承诺与现实之间的鸿沟，让它真正对企业有用，JVM必须发挥作用。”于是，Embabel应运而生了。



> 迟早，AI能力将会成为应用开发中的“新常态”。阅读文档后，我的第一印象是：它将有望像当年Spring简化Java开发那样，简化AI驱动应用的开发过程。非常期待它的发展。

Embabel的架构之道

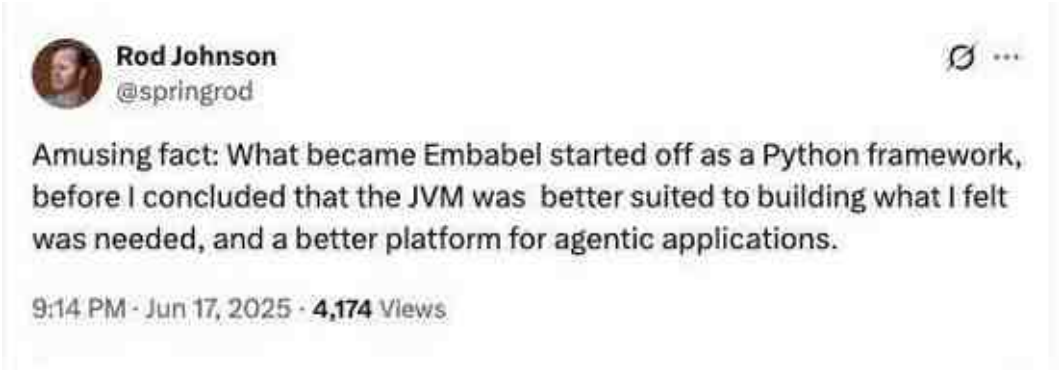
在Rod Johnson看来，MCP是试图填补这个鸿沟的一种尝试。“作为一个服务器平台，它在理论上是语言无关的，这意味着大家可以跨不同平台共享功能。”但MCP只是“解决方案的一部分”，“它并没有解决很多实际问题，比如discoverability（可发现性）、security（安全性）等。”

针对当下流行的“工具堆砌型”Agent方案，Johnson更是毫不掩饰自己的质疑：“我并不认同那种观点：只要给一个LLM一堆工具，它就能解决任何问题。”他认为，这不仅会造成巨大的算力和资源浪费，还存在混乱与不可控的结构性风险。“即使是最先进的模型，在面对200个工具时，也会迷失方向。”

相比之下，Johnson更支持“Agentic模型”的做法——以小提示驱动、精简聚焦的工具集、合理边界控制的架构方式：“这些工具可能来自MCP，但一定是有限且高效的；

也可以接入本地模型，获得隐私与成本上的优势。”在他看来，这种方式才更有可能实现企业级AI所需的三大目标：更高的确定性、更低的成本和更强的安全性。

他强调，这些Agentic应用并非只与LLM对话，它们同样需要与企业的系统记录源、消息总线、已有代码资产深度集成——“这些需求，显然更适合在JVM上完成，而不是Python。”



> 有趣的是：Embabel最初其实是以一个Python框架的形式起步的，直到我最终得出结论——JVM更适合构建我认为真正需要的东西，也更适合作为Agentic应用的平台。

因此，Embabel被设计为了一个用于在JVM上构建Agentic应用的框架，面向所有JVM开发者，尤其是使用Spring的开发者。

Embabel基于Spring AI构建，并拥抱Spring组件模型。但与Spring AI不同，Embabel工作在更高的抽象层之上。

“Embabel的层级比LangChain4j和Spring AI要高得多。它建立在这一层级之上，之所以选择Spring AI，是因为它与Spring生态系统的密切关系，但它提供了更高层次的抽象，并引入了独到的理念。代理框架层级在Python中已经很常见，但在JVM上却非常新。而Java要想在企业级及其他领域保持领先地位，就迫切需要它。”

如果说Johnson曾经打造的Spring框架是在为企业级Java提供更好的编程模型，那么Embabel的出现就是向着AI智能体编程迈出的重要一步。

智能体编程，能否既聪明又可靠?

与Spring使用基于依赖项注入与控制反转这一新型编程模型颠覆企业级Java类似，Embabel致力于建立一种编程模型，帮助开发者使用符合生产代码要求的最佳实践来构建智能体，同时继续为智能体追求特定目标、执行工作流任务并自主做出动态运行时决策留出充裕的空间。

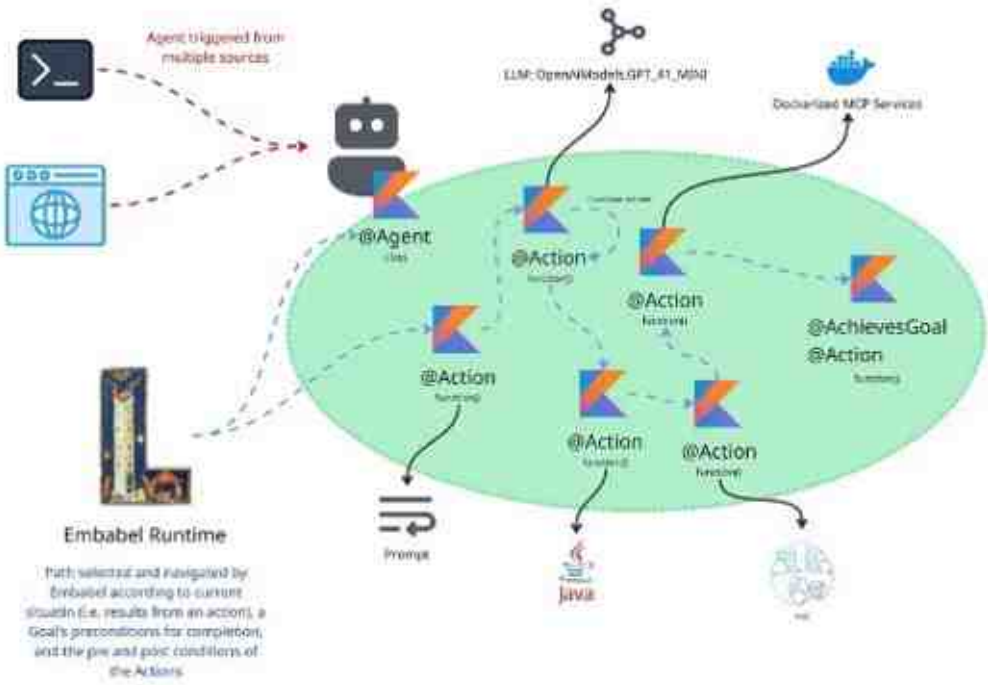
确定性似乎一直是AI模型难以达成的核心指标。当我们构建应用程序时，习惯于其应用行为具备确定性——换句话说，对于相同的输入与初始状态，我们的代码应当始终产生相同的输出和行为结果。

而大模型的运作方式并非如此。大模型会产生幻觉——有人说，人类不也充满了不确定性吗？但也正是因为存在不确定性，在被赋予自由发挥的创造力时，人类反而能够做出令人意外但却成效出众的成果。

对于基于提示词的简单人类交互循环来说，这种不确定性往往令人头痛，但影响还不算太大。而对于需要在无人交互的前提下采取行动的全闭合OODA循环（即观察、调整、决策与行动）的智能体来说，**结果的确定性则变得至关重要。**

建立对智能体工作流程的信任，要求我们找到一种保证智能体在整个工作流程中实现确定性的可行方法。简单来讲，我们既希望智能体拿出令人眼前一亮的开发成果，又要确保对工作的推进方式保留一定的知情权与控制权。为此，Embabel采用了GOAP方法。

这套机制的第一步被称为“Planning Step”，是Embabel最有特色的一环，并且是可插拔的。比如在Crew AI中，执行流程是顺序嵌套的；在LangChain或LangGraph中，你需要手动搭建状态机。用状态机去定义流程，系统只能做你预先设想过的事。要想做出调整或扩展，就需要改动很多逻辑，每次都得重新跑一遍。但Embabel不同，它工作在一个更高的抽象层上。



Embabel还引入了“行为（Actions）”“目标（Goals）”和“智能体（Agents）”的语义建模，每个目标和行为都有清晰的前置条件和预期后置条件。这使得Embabel能够使用非LLM的AI算法进行“规划”。它采用的是一种路径规划算法（path finding algorithm），用于从当前世界状态出发，寻找通往目标的路径。这样的机制其中一个优势是：系统可以执行那些你没有显式编程指令的操作，但前提是这些操作是“安全的”。

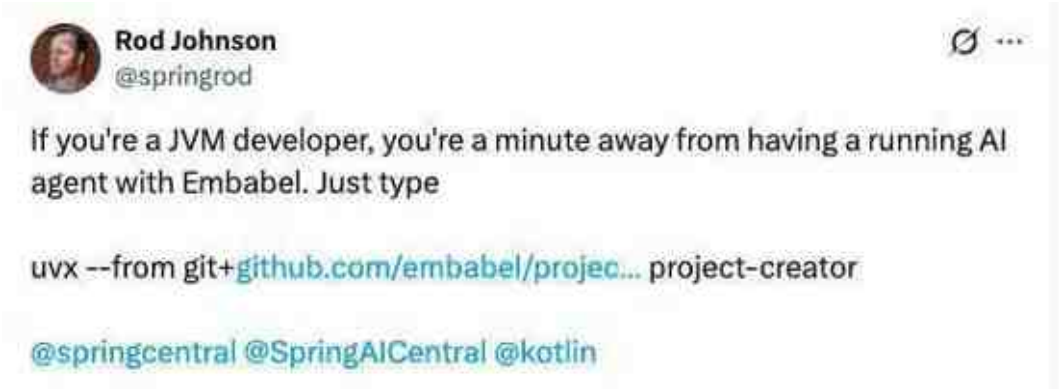
“你给它相同的世界状态和目标，它每次都会生成同样的计划。”Johnson表示，“而不像LLM一样，背后藏着3000亿个参数，让你根本无法解释它为何那样决策。”

这一机制的另一个亮点在于弹性执行与动态重规划：它可以围绕失败情况来重新规划。当流程中某个行为失败，比如尝试提取信息失败、数据源访问异常，系统会自动切换路径寻找替代方案，而无需显式写出复杂的if-else判断逻辑。这种“围绕失败自动重规划”的能力，让Embabel既具备传统工程体系的可控性，也拥有现代AI系统的柔性应对力。

最好的Agent框架？！

AI智能体的构建尚处于起步阶段，Embabel项目同样处在萌芽期。但Johnson的目标并不止于JVM。他直言：“我们并不是想做JVM上最好的Agent框架，而是想做最好的Agent框架。”

目前Embabel项目95%使用Kotlin语言编写，同时亦支持用纯Java代码开发智能体。



> 如果你是JVM开发者，使用Embabel启动一个运行中的AI Agent，只需一分钟。

另外几个点也很有意思，可能会让人感到惊喜——Embabel与Spring紧密集成。如果你是一个Spring开发者，使用Embabel创建一个Agentic流程会像写一个Spring MVC应用一样简单。“毕竟归根结底，这一切都是组件抽象，差别只是你用这些组件来做什么。”

而且在模型调用的可移植性方面，因为Embabel是建立在Spring AI的基础之上的，这也是为什么它有着非常强的MCP支持——得益于之前社区在Spring AI MCP上的出色工作，所以Embabel能够在Spring的基础上构建这些能力。并且，除了MCP支持之外，Embabel JVM代理框架现在还实现了Google A2A服务器。

> Spring AI正迅速成为企业领域的MCP（Model-Connected Platform）首选。它是目前这一方向上最值得关注的方案，结合了Spring Security实现认证与信任机制，以及Actuator提供可观测性能力。更关键的是，大多数企业本就使用Spring，因此切换到MCP模式变得异常顺滑。

在Embabel框架中，可测试性被视为一项基本能力。得益于与Spring提供的强大能力，开发者不仅可以像测试Spring服务一样对每个Action方法进行单元测试，还可以验证提示词是否包含正确的数据元素、是否调用了合适的MCP工具、是否指定了超参数等。

Embabel另一个重要理念是对LLM使用的“分级控制”。开发者可以定义哪些工具可暴露给模型使用、哪些只能只读访问，并在运行时动态验证是否具备执行权限。以编码Agent为例，文件系统的访问权限被明确划分为读取和写入两类。模型在执行写入操作前，会先验证例如项目是否成功构建等前置条件，只有在满足要求时才允许真正执行写入。这种设计既保证了流程的可控性，也提升了系统整体的安全性与稳定性。

为了进一步提升LLM生成内容的可控性，Embabel引入了“**意图完整性链（Intent Integrity Chain）**”的设计机制。“**确保在流程的最后，生成的代码与我们最初的意图完全一致**。”

整个流程链条从prompt构建开始，依次经过规范生成、规范审核、测试用例编译，最终进入代码生成阶段。每一个环节都可以嵌入人工审查或自动化验证机制，测试集则以只读形式锁定预期行为，确保结果可控。

LLM的角色则被聚焦在“不断尝试通过测试”这一任务上——它需要反复生成代码，直到全部通过测试为止。“一旦达成这一点，**我们就能100%确认生成的代码与prompt**

所表达的意图一致，或至少与我们审核过的规范保持一致。”

“我真心相信，AI是生产力的游戏规则改变者。”Johnson表示，“不论你是开发者还是市场人员，都应该被AI增强。我们的理念是：最大限度增强人的能力，而不是依赖任何商业化或封闭的AI工具。”

参考链接

- https://www.youtube.com/watch?v=3Qtc0yQTsyw&list=PLqcGn3yCaveQN33xDe2goHlQrUE1TuHlF&index=1
- https://medium.com/@springrod/embabel-a-new-agent-platform-for-the-jvm-1c83402e0014
- https://thenewstack.io/meet-embabel-a-framework-for-building-ai-agents-with-java/
- https://www.infoq.com/news/2025/06/introducing-embabel-ai-agent/

“比Flink更适合Agent！”十五年老中间件转型做Agentic AI：比LangChain快70%，还能省2/3算力

作者 Tina 核子可乐

7月14日，Akka正式发布了其Agentic Platform，号称“提供SDK与运行时，其中包含创建自主AI系统所需要的全部组件：编排、智能体、内存与流式传输”。这标志着Akka迈出了从传统分布式计算领域向AI驱动型系统转型的关键一步。

作为一款老牌的开源中间件项目，Akka旨在简化JVM上并发与分布式应用程序的构建过程。其软件已经被下载超过十亿次，支撑着全球大约十万个应用程序。并且它在GitHub上拥有13.2k的星标和超过3,600个分支（forks），足见其在开发者社区中的深厚积累和影响力。

然而，Akka的运营在近年来却面临着不小的挑战。尽管自2009年成立以来一直秉持开放友好的Apache许可证，但前两年，“公司已经实质上陷入困境”，每年客户流失量接近30%。因为没有有效的盈利模式，导致该项目运营面临巨大的资金压力。为了维持公司的正常增长和持续发展，Akka团队在2022年还不得不做出一个艰难的决定，将许可证更改为商业源许可证（BSL），以此获取必要的资金支持。

尽管Akka公司CEO Tyler Jewell曾坦言对AI和“智能体”的无休止炒作持怀疑态度，但他现在明确表示“我们是时候抛开偏见了”。Akka正在全力押注代理式AI，因为他注意到越来越多的客户已经开始使用Akka构建代理式系统，并且迫切需要更多相关功能。Jewell提到，尽管客户长期以来一直呼吁Akka提供AI功能，但公司官方此前始终保持着谨慎态度。

“如今我们正在扭转乾坤，构建起一套比任何Langchain现有技术都更丰富、也更复杂的AI功能集。”

简言之，该平台相当于Langchain的Java应用替代方案；Akka类似于瑞士军刀，支持高容量代理式AI工作流的自动化。Jewell提到，与其他代理式应用程序构建方案相比，Akka能够显著降低计算成本。

“这套集成平台可以将Agentic系统的开发速度提升3倍，计算资源降至三分之一，并满足各类SLA，无论系统是自主、自适应、实时、事务型还是部署在边缘端。”

其中包含四大核心组件：

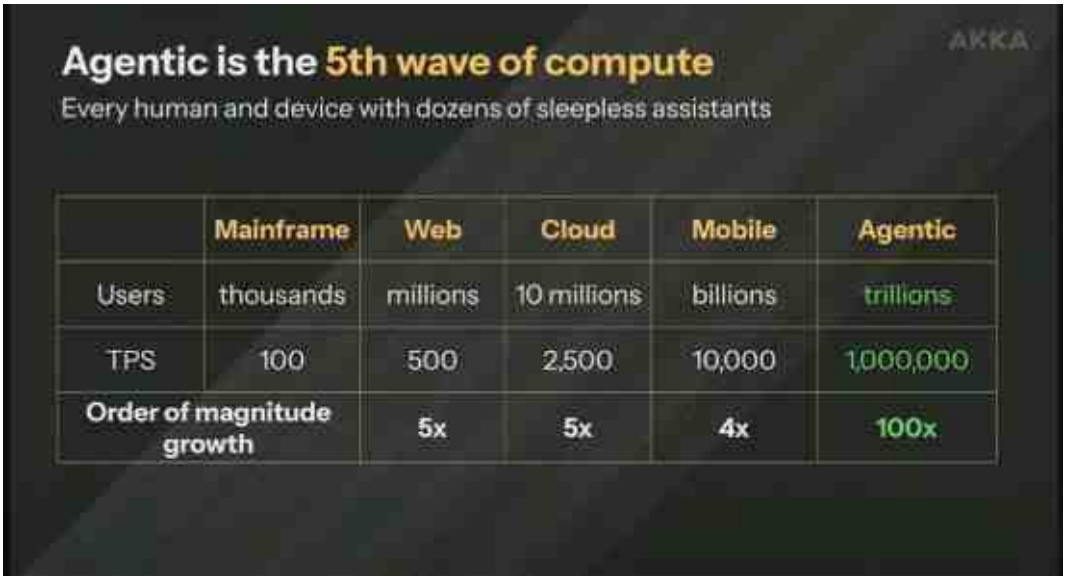
- Akka Orchestration，用于“引导、调节和控制长期运行的系统”；
- Akka Agents，用于“创建智能体、MCP工具和HTTP/gRPC API”；
- Akka Memory，用于“持久化、内存内及分片数据”；
- Akka Streaming，用于流处理等。

这不仅是对LangChain、Crew.ai、Temporal和n8n等方案的替代，更是对代理式系统架构进行的一次系统性重塑。

全力押注Agentic AI找活路

在2025年QCon的一场演讲中，Akka团队分享了一个他们最近的发现，“Agentic AI”这个词在Google搜索中的热度，在过去四五个月增长了超过1500%。他们强调，这不仅是一个备受关注的热门词汇，更是一个潜力无限的强大概念。

“我们坚信，Agentic AI将成为第五次计算浪潮。这是一个你无法忽视的巨大趋势。”Akka高级总监Duncan DeVore如此断言。



	Mainframe	Web	Cloud	Mobile	Agentic
Users	thousands	millions	10 millions	billions	trillions
TPS	100	500	2,500	10,000	1,000,000
Order of magnitude growth		5x	5x	4x	100x

回顾历史，第一次计算浪潮是大型机。随后，我们步入了“小型机”时代，计算机开始联网，最终进入Web时代。Duncan引用了一个著名的例子：微软最初曾将Web视为“昙花一现”的风潮，但仅仅六个月后，这家巨头便彻底转型，其变革之迅速和彻底，即便对于微软这样规模的公司而言，也是一次里程碑式的决策。如今，这个历史片段如今正在重演，只不过这一次，是Agentic AI。

“你可以看到，随着每一波浪潮的推进，用户数量从几千到几百万，再到几亿、几十亿，现在在移动互联网时代已经达到数十亿用户。而我们预测，Agentic AI时代的用户量级将达到‘万亿’级别。”

这将是一次前所未有的规模挑战，也意味着前所未有的系统复杂度。在这种背景下，传统技术堆栈开始显得力不从心。

“试想一下，在面对万亿用户、每秒百万级事务处理、Agent分布式并行处理的场景下，你还会用Python+MySQL写系统吗？”

他指出，Web时代的MySQL已基本抛弃了关系型理论，采用“扁平化”表结构；而到了移动时代，即使是列式数据库也逐渐力不从心。而在Agentic时代，我们将面临十倍乃至百倍于今日的负载与动态复杂度——不只是执行一个操作，而是有五个、十个、十五个Agent同时在执行，甚至还会衍生出“子代理”。这一切都需要统一调度与管理。

如今范式转变已至，“Agentic AI是你无法回避的未来趋势”，所以Akka决定将这类技术集成到自己的服务中。

这中间也不可避免的遇到一些挑战。

Agentic系统，特别是那些偏重推理能力的模型系统，常常伴随着5-15秒的显著延迟。这种“代理性循环”要求我们彻底摒弃传统阻塞模式，必须转向非阻塞、能处理高并发的模式。

传统的“分布式应用”多为无状态、单线程的架构，依赖Kafka这一类的队列写入数据库，然后通过负载均衡来实现可扩展性。这种架构的弊端明显：一旦某节点负载过高，极易引发级联故障。在面向Agent的超大规模系统中，“写入队列+写入关系型数据库”的传统方式的旧方法已经走到了尽头，必须从架构层重构整个系统的运行范式。

Agentic服务的架构也在发生变化：要么作为独立微服务存在，与SaaS系统协同运行；要么两者融合，构成一个统一的联合系统。

另外，Agentic服务——它是有状态的，能够管理历史上下文，大多数时候会采用事件溯源（event sourcing）机制。这意味着，读取和写入不能走同一通道，因为数据量太大，要避免拥塞。在移动时代，我们还能“勉强凑合”，但在Agentic时代，这种方式已经无法满足需求。

值得注意的是，LLM本身是无状态的，其上下文历史通常由外部基础设施负责保存并传递——一部分在前端界面中，另一部分在与模型交互的后端系统中。随着对话复杂度提升，我们必须每次都上传完整上下文。这就像事件溯源——你记录了1000万个事件，每次想得出当前状态时都得重放一遍。为此，系统必须周期性地做状态“快照”（snapshotting）。

Agentic还必须是双向、异步且支持并发的“流式传输”架构。这类流式能力在大模型时代，要支持文本、音频和视频数据，还必须是开箱即用的，否则就算一开始能“凑合着”跑起来，但很快就会遇到大问题。在实际项目中，一些现成的LLM接口（如Gemini适配器）因采用阻塞、同步方式，很快就暴露出瓶颈，无法满足实时性需求，最后不得不重写为支持异步双向通信的版本。

随着使用LLM的复杂性提高，开发者越来越发现，这其实是一个**分布式计算问题**。流式双向通信、状态一致性管理、负载动态调度，这些问题不再是数据基础设施团队的专属，而直接进入了应用层的架构范畴。

所以，Agentic架构需要有五大核心能力：

1. **流式端点（Streaming Endpoints）**架构必须支持双向、异步、并发的数据流。无论是文本、音频还是视频，模型与人之间的交互不再是“请求-响应”，而是“持续交互”。这就要求系统能支持很多复杂的流处理和反压（backpressure）管理。
2. **智能体适配器（Agent Connectivity & Adapters）**Agent需要连接多个异构系统：IoT、数据库、外部API等，这些连接必须异步、非阻塞且高可用。开发者不想自己去处理信号量（semaphore）、锁（locks）这些底层复杂逻辑，而是希望用已有的平台自动处理这些。
3. **智能体编排（Agent Orchestration）**能够管理Agent流程和执行逻辑，比如Scatter-Gather（分发-汇总）、路由模式（router pattern）等，平台或工具需要支持这些能力。
4. **内存数据库+CRDT（冲突自由复制数据结构）**要访问支持冲突解决的数据结构（CRDT）。平台具备持久性，当某个节点崩溃，可以快速恢复并重新加载数据，在一个节点上读取的数据，换到另一个节点时也要是一致的。
5. **Agent生命周期管理（Agent Lifecycle Management）**当某个Agent崩溃或宕机

时，系统应能自动重启它、加载其上下文，或根据情况创建新的子Agent。这类似于Actor模型中的“监督策略（Supervision）”，用于构建真正具备韧性（resilience）的AI系统。

因此，这五大能力构成了Akka系统的“蓝图”。同时，这也是一种新的架构哲学：从移动应用时代的N层架构中解放出来，彻底推倒重建，形成适配Agentic AI时代的A-Tier架构。

其CEO Jewell对新Platform表现得非常开放且自信满满。

“我们的技术栈中有两项让人眼前一亮的功能。”

“一个是内存机制：它是隐式的，就是说内存会自动成为智能体和工作流程中的一部分。我们的所有竞争对手都要求用户单独添加内存，而这会产生很大的编程挑战，特别是难以实现可靠且可扩展。而我们支持自动添加内存。”

“另一大优势在于，我们是唯一一家为代理式框架提供高性能流式传输功能的供应商。就是说如果大家需要接收音频或视频，或者打算获取物联网指标，且希望智能体可以处理这些指标，就必须拥有与智能体并行运作的持续处理引擎……”

那么，难道市面上还没有能够实现这些功能、易于理解且可以轻松应用的开源流式

传输引擎吗？

“市面上当然不乏独立的高性能事件流式传输引擎，比如Flink，它能够支撑起大量消息总线。”

“但Flink之类只是独立系统，仅适用于计算。而借助Akka，我们将智能体编排、内存与流式传输功能集成至同一编程包内。其优势在于，用户能够一站式对所有功能进行编程，且各项功能均使用相同的计算资源。**也正因为如此，我们的执行效率才会比Langchain高出70%。**”

这是黎明前的最后黑暗……

Akka近期的发展状态相当不顺。该软件由Lightbend（即现在的Akka）作为开源软件项目交付和维护，被广泛用于支持Java及Scala的弹性消息驱动类应用程序……

Akka公司由CTO Jonas Bonér建立，其产品模型是“在分布式系统的多个核心上进行并发，确保用户服务能够在不同时间及空间的不同位置上成功运行。每天有超过10万个系统在生产环境中构建和部署，以Akka作为内部运行时的应用程序已经拥有约20亿用户。”

但怎样将这款开源软件的高人气转化为商业收入？答案并不简单。

开源应用（包括Adobe、苹果、亚马逊云科技、Azure、Google Cloud、HPE、特斯拉等）非常广泛，但Akka只是整体技术栈中的一枚齿轮，组织往往没有意识到自己正在使用。本质上，Akka只是企业IT栈中功能强大的一环。

他宣称，Akka的社区参与度很低，而且由于该开源软件本体运行状况良好，所以没办法采取开放核心的盈利模式。

该公司在基础开源元素之外追加销售部分高级附加组件的开放核心模式没能跑通，**“到2021年，公司已经实质上陷入困境”。**他们失去了很多客户，每年客户流失量接近30%。这一切与竞争对手无关，客户只是单纯降级至免费开源软件，并表示“开源版本已经足够好用”。

客户坦言“这是一款很棒的软件，我们非常喜欢。我们感谢公司的贡献，但如果非

得付钱，那我们会另想办法……”

Jewell补充道，多年来绝大多数上游开源贡献均来自Lightbend。因此在2022年，CTO兼创始人Bonér被迫在Akka v2.7中将Apache 2.0许可更换成了限制性更强的BSL v1.1（商业源代码许可证），并开始按核心收费。

Jewell自豪地宣布，“从那时起，我们迎来了可观的收入增长。去年我们的业绩仍保持盈利，未计利息、税项、折旧及摊销前的利润（EBITDA）与现金流也均为正数。”

“今年早些时候，部分现有客户及潜在客户找到我们，询问我们是否在关注代理式AI。当时我们的回应是还没太关注，AI成果更像是玩具、而非基础性工具。”

但客户们强调“不，你们真的应该给予关注。我们试过一些开源代理式框架，比如Langchain和Crew.ai，但它们总出问题……”

“这时候我们才意识到必须行动起来。”

“事实证明，客户至少已经在Akka上构建和部署了几十种代理式AI系统，其中最大的系统每秒处理近10亿个token——也就是来自印度的线上食品订购与配送平台Swiggy。”

“代理式AI本质上是多个智能体，它们采取某种分布编排形式，且均共享某种状态。我们突然想到，既然代理式AI的本质是分布式系统，那不正好跟Akka的设计初衷相吻合吗！”Jewell充满信心地表示，“所以我们做出一个不同寻常的决定，**从今年三月开始全力投入代理式AI领域**，并构建起一整套比Langchain现有产品都更加丰富且复杂的AI功能集。”

Akka于2022年的许可证变更当然引发了不可避免的结果：开源分叉Pekko，但后者并未像Redis分叉Valkey或Terraform分叉OpenTofu那样获得关注。对于个中原因，Jewell简单总结称“社区没有投入资金来做功能改进。”

“Akka发现并解决了一系列关键运行时问题，而Pekko没能及时跟上，这影响了其进一步普及。”他还尖锐地指出，“Pekko过去三年间的提交量比Akka少了约2万次……”

“三年前，Akka的大部分（超过99%）的贡献都来自Akka（Lightbeng）员工，这是开源社区所无法比拟的。”

“社区正在构建自定义集成与数据存储方案，例如对MongoDB的支持，但这些只是他们在自有项目中用过后又免费提供的附加组件。社区本身对Akka核心路线图及发展方向的投入不多……”

Jewell拒绝透露Akka目前的运营情况，而且对团队规模三缄其口（只提到「超过10人，不到1000人」）。此前Akka曾在2023年进行了一次小规模债务重组，他的公关公司在随后的邮件中补充称Akka目前开放15个招聘职位，其中包括三个现场CTO职位和一个首席AI官职位。

参考链接

- https://akka.io/blog/announcing-akkas-agentic-ai-release
- https://www.thestack.technology/agentic-ai-convert-akka-looks-to-take-on-langchain/
- https://www.youtube.com/watch?v=EpPrZ1-OMe8

还在拼命加GPU？AI应用规模化的下半场，拼的是这五大软件“新基建”

作者 作者 天子、冯嘉、向阳、连城、源启
策划 Tina

过去十年，我们通过中间件、数据库与容器技术，奠定了云原生应用基础设施（**Application Infra**）的基石。我们利用消息队列、实时计算与分布式存储，铺设了现代数据流动的高速公路（**Data Infra**）。

从云原生到AI原生，应用基础设施的范式正在经历一场深刻的跃迁：从面向资源的交付效率革命到面向认知的智能治理进化。AI的世界观是概率性的、涌现的。给定的输入，通过一个黑盒模型，产生一个不确定的、但可能非常有价值的输出。

我们的使命，是为这个不确定的“创造过程”提供引导、护栏和成本控制。当前，AI热浪正在席卷全球资本市场，根据英国《金融时报》最新报道，全球10家**尚未盈利**的

AI初创公司在过去12个月内估值合计暴增近1万亿美元，创下史上最快的财富膨胀速度。资本和人才聚集在这一新趋势上，大家押注的，不仅是模型的算法、算力，高价值应用的涌现，更是我们能否实现人类智能的**复制**。

如今，AI技术正在重塑一切。AI应用的普及速度也正以一种前所未有的方式冲击着市场。我们正在迎接一个“比移动互联网大10倍”的浪潮。OpenAI的ChatGPT，在短短2个月内便吸引了超过1亿的月活用户，刷新了所有消费级应用的增长记录。这一速度，将过去需要数年才能完成的用户积累，如今被压缩到了惊人的数周。高盛预测，生成式AI领域的投资规模将达千亿美元级别，并可能为全球GDP带来7%的巨大增长。这种爆炸式的增长态势，为应用基础设施带来了前所未有的机遇与挑战。

先前构建的高效的数据通路已经远远不够，我们迫切需要构建一个能够容纳、管理和驱动无数自主智能决策单元的**智能城市**，从“连接”转向“认知”。面对如此挑战，我们需要将过去构建通路的**经验升维**，去设计和定义承载大规模智能体的下一代应用基础设施。过去在构建大规模高可靠分布式系统中的深厚积淀，使我们能够以一种更**体系化**、更**工程化**的思维，去探索和定义新一代**AI Infra**所带来的全新挑战。接下来，本文将深入探讨智能应用基础设施背后的技术架构演进思考，以及如何利用这些技术“灯塔”，去构建承载和治理大规模智能体的AI原生应用中心。

时代的回响 - 云原生基础设施的实践总结

回望过去十年，整个信息技术领域的核心叙事，无疑是围绕着云原生展开的。它并非一种单一技术，而是一整套**构建和运行现代化应用的系统性思想与实践**。作为这场技术浪潮的深度参与者与贡献者，我们致力于将这些先进理念转化为支撑数字业务运行的坚实底座。

现代化数字城市的基础设施蓝图

时间回到2014年，当几个Google工程师决定将内部的Borg系统开源时，他们或许并未预见到，这个名为Kubernetes的项目，将如何定义下一个时代的云基础设施，并最终成为这座数字城市的“操作系统”。它为所有上层应用提供了**统一**、**标准**的规划蓝图和稳固地基。这使得不同功能、不同形态的**应用服务**与**数据任务**，都能在这片土地上和谐共生。

在城市内部，我们通过Envoy、Istio、EventMesh等服务网格技术，构建了精密高效的“智能交通网络”。同时，以gRPC、Dubbo作为标准化的“高速轨道”，确保了服务间的通信如同高铁般有序、迅捷与可靠。当然，城市的运转离不开能源和水源的持续供给。我们以Kafka、RocketMQ等消息流系统，构建了覆盖全城的“主动脉”，让数据像血液一样精准、实时地输送到每个需要的角落。

而像HDFS这类汇聚海量数据的“战略水库”，在云原生时代迎来新的发展。它不再是一个单一的分布式存储系统，而是演化成了一个提供多元化、API化服务的存储基座，按需提供对象存储（Object Storage）、块存储（Block Storage）和 文件存储（File Storage），并通过CSI（Container Storage Interface）规范与上层应用无缝集成。

在这座数字城市中，总有一些设施是为了挑战速度的极限而存在。以Redis为代表的内存计算，便是这样的存在 - 它并非简单的加速器，而更像是构建在**数据与应用**之间的高速传送门。它以接近电光石火的速度，确保了每一次关键交互都能得到瞬时响应，成为整个城市体验的**终极助推引擎**。

图1

从数字城市的整体规划设计——统一调度，到联通四通八达的交通网络——应用通讯，最后再到能源和水利系统——数据底座。正是通过这套分层协同的架构，我们在实践中**融合了应用基础设施与数据基础设施**，实现了重要突破。无论是服务于在线应用的

微服务，还是用于后台分析的Spark、Flink作业，都被视为这座“数字城市”的居民，享受着统一的**资源调度**、**网络通讯**、**数据存储**和**运维体验**。这份实践最终沉淀为我们的核心交付，一个为所有业务负载提供**弹性**、**韧性**、**可观测性**与**敏捷性**的统一应用平台。我们始终相信，最优秀的基础设施，是离应用最近的基础设施。因为它承载着业务的每一次“心跳”，支撑着企业的每一次“成长”。

时代的机遇 - AI原生范式带来的颠覆性挑战

当云原生范式将数字城市的建设推向一个前所未有的高度后，我们一度认为，未来的叙事将是围绕这座精心构建的城市进行持续的**精细化治理与迭代**。

然而，历史的脚本，显然准备了更令人惊叹的下一章——一场更宏大的范式革命**AI原生**，正以不可阻挡之势席卷而来。一个新的物种——智能体（Agent）正以惊人的速度涌现。以大语言模型（LLM）为代表的AI技术，其角色正从一个外部依赖的咨询顾问，演变为这座城市里能够自主思考和行动的核心居民。就像电影《I，Robot》中觉醒的中央主脑一样，我们亲手构建的这座数字城市，它的大脑第一次不再是提问“我应该做什么？”，而是开始自主判断“什么才是对的？”

新王登基：当“AI原生”叩问“云原生”

这标志着一个根本性的转变。应用拥有了“灵魂”，其核心行为模式从传统的“请求 - 响应”升级为“感知 - 理解 - 规划 - 行动”的自主循环。这场由AI驱动的范式革命，正对我们精心构建的云原生体系提出了一系列严峻的考验。我们曾经坚固的“城墙”和高效的“交通网络”，在这些拥有自主意识的“新居民”面前，开始显得力不从心：

从“编码为中心”到“智能体生命周期管理”

云原生时代，我们所有工作的核心都围绕着代码展开。开发者的职责是编写业务逻辑，平台工具则负责将这些代码构建成镜像，部署为服务并确保其稳定运行。整个生命周期管理是清晰、线性的。在最新的技术趋势报告中，Gartner明确指出，**生成式AI正催生一种全新的、以智能体为核心的应用架构**。随着这些智能体逐渐成为应用的主角，这个以代码为中心的范式开始被颠覆。

不同于执行静态代码的传统应用，这种以**认知内核**驱动，旨在自主理解并实现用户**动态意图**的新一代应用——**AI原生应用，**开始逐渐走入大众视野。在AI原生应用中，决定其行为的不再仅仅是代码，而是一个复杂的、动态的组合。它包含了作为“大脑”的大语言模型、定义其“人格与目标”的提示词（Prompt）、连接物理世界的“双手”即工具（Tools）、长期积累经验的“记忆”（Memory），以及驱动这一切的少量胶水代码。

开发者的核心工作，也从埋头编写业务逻辑，转变为一种更贴近导演或教练的角色——他们需要精心设计Agent的构成，反复调试其与环境的交互，并持续治理其自主行为，确保其目标对齐。

这就对应用基础设施提出了一个本质性的新要求：平台不能再仅仅是一个代码的**部署管道**，而必须进化为一个**智能体的孵化器与管理器**。它需要提供一套全新的**开发范式**和**管理界面**，让开发者能够直观地定义和组装一个Agent。需要将过去被视为非结构化文本的Prompt，像代码一样进行版本控制、AB测试和灰度发布。需要建立一个统一的工具箱，让Agent可以按权限安全地调用各种API。

总而言之，我们迫切需要一个能覆盖Agent从设计、开发、测试、调试、发布、监控到迭代演进的全生命周期的治理平台。而这，已经远远超出了传统CI/CD和应用管理平台的范畴。

从“无状态”到“有状态”的长时记忆

云原生架构推崇无状态服务，以此换取极致的部署灵活性与弹性伸缩能力。然而，为了在Serverless范式下兑现业务“永远在线”的承诺，我们早已通过计算与状态分离的池化架构，成功攻克了有状态服务平滑伸缩这一业界难题。这一实践沉淀出的分层存储技术，恰好为今天我们构建智能体的“长时记忆”系统，提供了一个坚实无比的起点。

现实生活中，人人都想**增加记忆**。但我们必须清醒地认识到，智能体的记忆，远非简单的**业务状态存储**。它不是一个功能，而涉及用户体验、隐私和系统影响的设计决策。从架构视角来看，我们可以将其抽象为一个上下文层与行为层的组合，抽象成一个基础组件，使其成为一个安全的，可移植的记忆层，可以跨应用工作。

实践告诉我们，**真正能帮助人类解决复杂问题的AI，其卓越能力很大程度上源于其对精准上下文的理解，对过往经验的反思与归纳总结能力**。它需要能实时获取物理世界的多元数据信息，在跨越数天、数月甚至更长时间的交互中不断学习和进化，而这必须依赖于一个可靠且高效的记忆系统。

因此，真正的挑战浮出水面：我们如何将久经考验的分布式存储底座，进行一次脱胎换骨的重塑？这已不再是服务于**机器状态**的升级，而是要构建一个能够支撑**AI认知**的全新记忆系统。这场变革的本质，是存储系统核心使命的三重演进：

从存储“数据”，到管理“知识”

传统存储关心的是比特和字节的无损记录。而新的记忆系统必须理解数据背后的语义，它管理的是由**向量**、**图谱**和**实体关系**构成的知识网络。它的职责不再是**存得下**，而是**看得懂**。

从响应“查询”，到使能“推理”

传统存储被动响应精确的SQL或API调用。而新的记忆系统要主动参与AI的推理链，它需要响应一个模糊的目标，并动态地整合、剪裁和生成上下文信息，为AI的每一步思考提供恰到好处的灵感素材。它的价值不再是**找得快**，而是**给得对**。

从记录“静态”，到驱动“演进”

传统存储是应用状态的一个快照。而新的记忆系统本身就是一个活的有机体，它通过持续的反馈闭环，从AI与世界的交互中学习，动态调整记忆的权重，形成新的认知联想。它的要求不再是**记得全**，而是要**学得会**。

综上所述，我们的目标，是为AI打造一个真正的认知“海马体”。它存在的意义，不只是为了快速遗忘或精准记忆，而是为了在海量经验中**提炼智慧**，在复杂情境中**推理关联**，并为智能体的每一次决策提供深邃的洞察力。

从“确定性编排”到“涌现行为治理”

过去很长一段时间里，无论是通过serverless工作流引擎编排在线、离线任务，还是通过BPMN驱动复杂的业务流程，我们都沉浸在一个**确定性**的世界里。服务间的每一次

调用、任务的每一个步骤，都遵循着预先设定的、可预测的有向无环图（DAG）。软件系统的行为，如同精密的机械钟表，一切尽在掌握。

早期的智能体架构，经历了从线性工作流到单Agent循环的演进。这一步虽提升了系统的鲁棒性与成果质量，但其内核仍是串行处理的范式，考验的是Agent之间的顺序接力。然而，当任务的复杂度超越了单点执行，需要被分解为多个可并行的子问题时，这种串行范式便暴露了其天花板。因此，让多个智能体并行协作，就从一种**优化选项**演变为一种**架构必然**。

在此驱动下，能够支持复杂协同策略的**多智能体架构**，成为了合乎逻辑的下一站。这一转变，也正式宣告了过去那种自上而下的、确定性的控制范式已然失效。取而代之的，是一个全新的指挥与协同框架。系统的核心，一个作为“指挥官”的Orchestrator Agent，负责将宏观任务解构为可并行的策略意图，并分派给多个专业的执行Agent。这些执行Agent并非简单的指令接收者。它们在一个共同的目标下，形成了一个动态的协作网络。它们自主感知环境、进行局部决策，甚至为了最优路径发起**内部辩论**。它们的交互不再是僵化的汇报线，而是一个弹性的、自适应的生命群体。

在这个群体中，系统的执行路径充满了非确定性。我们无法预知某个Agent会调用何种工具、与哪个同伴协商、在何时调整自己的计划。这种源于简单规则之上的复杂集体行为，正是涌现智能的真正魅力所在。但正如潘多拉魔盒一旦开启，与巨大潜力一同释放的，还有其潜在的风险。一个令人警惕的治理悖论也随之浮现：**我们梦寐以求的、能够应对复杂世界的强大能力，恰恰源自于我们主动放弃对其过程的微观控制。**

因此，伴随而来的核心挑战也愈发清晰：如何将我们的角色，从一个任务**编排者**重塑为一个涌现生态的**治理者**？这意味着，我们设计的不再是一个严格规定步骤的脚本执行器，而是一个能够容纳和引导不确定性的协作涌现空间。这个空间必须成为Agent运行时的核心引擎。它不应仅仅是Agent生存的容器，更要成为一个主动的干预框架。这个框架的核心职责包括：

- 运行时审查与修正 - 实时洞察并理解Agent动态生成的执行计划，防止其在复杂的协作中迷航。
- 资源与权限管理 - 对资源消耗、工具调用进行精细化的授权与监督。

归根结底，我们要做的是**赋能而非放任**，是**引导而非控制**，从而确保这股源自涌现的强大力量，始终在安全、可信的轨道上，为我们创造价值。

从“内聚服务”到“开放生态整合”

微服务后时代，以Envoy、Istio、EventMesh为代表的服务网格中间件，将网络通信能力从业务代码中下沉，交给一个独立的、与业务进程并存的边车代理。这样一来，所有服务间的**东西向流量**都被Sidecar劫持。我们不再需要改动任何业务代码，就能获得统一的**重试**、**熔断**、**负载均衡**、**加密认证**、以及**端到端的可观测性**。本质上，这是平台治理Infra的一场革命，将流量治理的权力从应用开发者手上，集中到了平台管理者这里。

但回望整个流量入口，大家往往会用Nginx、F5等做南北向的API网关，又在集群内部署一套Istio做东西向的服务网格。这导致了“两套班子，两套规则”，运维复杂，策略不一致。为了统一与简化架构，我们为云原生网关丰富了L7协议处理能力，使其能够基于Header、Path与Body等内容进行智能路由、重写与决策。同时，使其具有动态可编程性，网关不能再依赖静态配置文件和重启。它必须能通过一套标准的发现服务xDS API，从控制平面动态接收服务发现、路由规则、安全策略等所有配置，并实现热加载，做到配置秒级生效且业务无感。

这套统一**流量治理架构**，其**历史功绩**在于为微服务建立了一个清晰的信任边界。它通过**将流量控制、服务发现和零信任安全等能力下沉到基础设施层**，成功地将一个混乱、复杂的内部服务网络，收敛成了一个边界清晰、行为可预测的秩序标准。

然而，AI应用的诞生，其基因层面就要求打破边界。智能体的感知与行动能力，几乎完全依赖于对外部世界各种工具、API与知识的动态整合。从GPT、Claude等基础模型，到Google搜索、天气查询等公共API，再到企业内部的各种私有工具。这种根本性的转变，使得传统服务网格的流量治理范式捉襟见肘。深入洞察后，我们发现其根本挑战在于：

- 流量治理的重心，从服务间的负载均衡转变为对多种（统一世界模型的通用模型还未真正来临）外部模型的**动态路由**、**成本与时延的精细权衡**。
- 安全范式，从内部服务的双向认证mTLS互信，演变为对海量、异构的外部API安全凭证的统一注入与安全轮换。
- 资源管理维度，对每一次外部API调用的成本进行实时度量与预算控制，以及通

过语义缓存来大幅优化昂贵的模型调用。

这一切都清晰地表明，我们需要构建一个专为AI生态打造的智能流量入口，它必须成为连接智能体与广阔数字世界的统一入口，提供一站式的路由、凭证、成本和安全治理能力。

从“白盒监控”到“黑盒行为洞察”

在微服务治理的黄金时代，我们围绕经典的可观测性三大支柱 - 指标（Metrics）、日志（Logging）和链路追踪（Tracing）构建了一套成熟的白盒监控体系。通过分析CPU占用率、函数调用日志和分布式调用链，我们能像钟表匠一样精确定位系统的每一个故障齿轮。

然而，这套为确定性“世界”设计的体系，在面对以LLM/SLM为核心的智能体时开始显得力不从心：

- Metrics失焦了，我们能度量Token消耗和API调用次数，但这些成本指标无法回答那个最关键的问题 - Agent的这次决策，其智能的投入产出比是多少？
- Logging泛滥了，Agent与大模型之间千变万化的Prompt和Response，产生了海量的、非结构化的对话日志，我们淹没在对话的海洋中，难以找到关键的决策拐点。
- Tracing断裂了，传统的调用链只能追踪到Agent服务调用了大模型API，但无法穿透进去，看到Agent内部的思考链。它为何做出这个决策？它的推理路径是什么？

LLM本身就是一个深邃的黑盒，其决策过程是概率性的，而非确定性的。当一个Agent任务返回了不理想的结果，我们该如何溯源？是交付给模型的提示词存在歧义？是模型自身的能力不足以应对该任务？还是在众多的工具调用中，某一次执行失败导致了后续的连锁反应？

这里的核心矛盾在于，传统监控关心的是服务的**生理健康**（CPU、内存、磁盘），而我们现在必须洞察智能体的**心理活动**（决策过程）。因此，平台必须提供一种全新的可观测性范式，这种**行为洞察**不再是简单地记录代码执行，而是要能够完整复现Agent的“思考 - 行动 - 观察”的心路历程。通过将Agent的每一次决策、每一次外部调用、每一次结果反思都串联成一条清晰的行为轨迹，我们才能最终将这个黑盒的运作过程，

转化为可供开发者调试、运维人员审计、产品经理理解的半透明视图，从而真正驾驭这些充满智能却又难以捉摸的数字生命。

以上五个环环相扣的挑战，共同编织出从云原生迈向AI原生所必须穿越的认知与架构的**转型矩阵**。这一系列结构性挑战，迫使我们不能再用打补丁的方式修补旧系统，而必须回归第一性原理进行系统性重建。在为新一代应用——AI原生应用构建基础设施之前，必须先深刻理解新一代应用本身。我们认为，在AI原生时代，AI智能体正是那个最具代表性、也最具挑战性的应用范式。它不仅涵盖了外部知识增强RAG、工具调用等常见AI应用模式，更引入了包括**自主规划**、**动态决策**、**技能组合**等更高维度的复杂性。**以至简的内核去驾驭至繁的真实场景，解决实际问题，才能真正面向未来**。

因此，从核心单元开始进行系统性重建就显得尤为重要。从智能体的定义与组装，到它运行时的动态治理，从开放生态的流量入口，到黑盒行为的深度洞察，基础设施的每一个核心层面都需要一场深刻的变革。在接下来的篇章中，我们将逐一解构其背后的技术本质，并提出一套完整而清晰的设计蓝图。一个能够真正承载、驱动和治理大规模智能体协同工作的下一代智能应用基础设施，将如何从我们熟悉的技术体系中演进而来。

时代的擘画 - 构建下一代智能应用基础设施

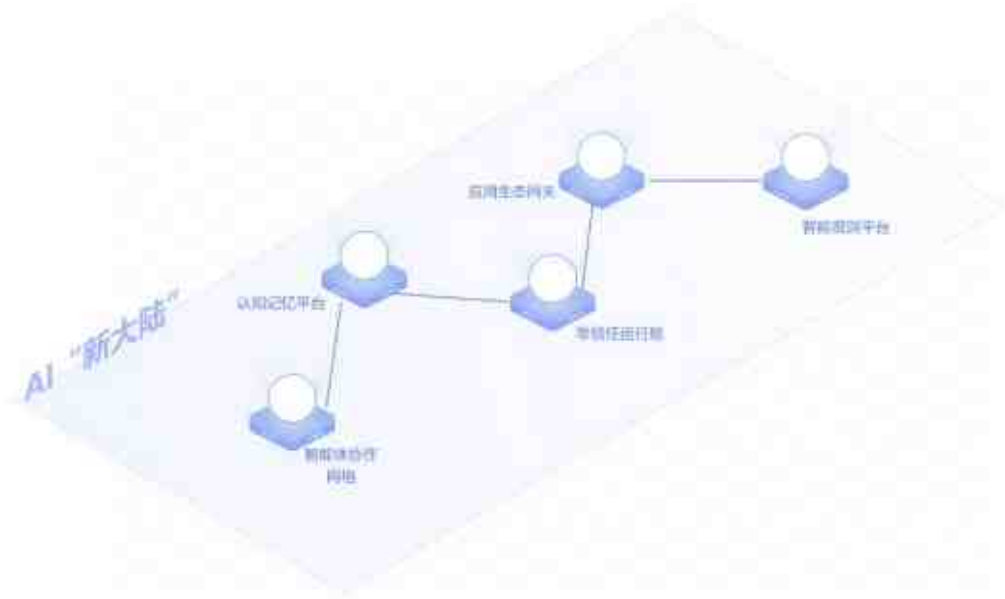


图2

当一个主流范式将其能力发挥至成熟，甚至繁荣的阶段时，它恰恰也为整个行业清晰地标定了那些“它无法回答的问题”。而下一个时代的浪潮，正是由那些致力于解答这些全新问题的探索者所开启的。过去十年，我们见证了云原生技术栈如何以前所未有的**效率与弹性**，重塑软件的开发与交付。如今，随着算力成本的**指数级下降**和模型规模的**暴力美学式增长**，我们正亲历一场由大规模、无监督学习所驱动的结构性变革。

我们能清晰地看到，过去那些围绕确定性逻辑和静态接口构建的原子能力，在面对以Agent为代表的、具备自主探索和涌现协同能力的新计算主体时，正遭遇经典的**降维打击——其本质，就是当游戏规则本身被重写时，你过往赖以成功的壁垒，便瞬间形同虚设**。

因此，我们需要的不是对现有能力的修补，而是一场彻底的架构跃迁。要完成这次深刻的跃迁，我们需要一张清晰的行动纲领。而在接下来的篇章中，这份时代的擘画就将逐一展开。正如上图2所示，通往AI新大陆的航海图已经绘就，其航向将由这五座关键的技术灯塔所指引。

灯塔一：智能体协作网格，典型代表Agent Mesh

刚刚过去的十一黄金周，相信有不少人已经利用AI设计了自己的旅行规划：为一家四口（两个孩子分别是5岁和10岁）规划一次为期10天的瑞士深度游，偏爱自然风光和徒步，预算8000美元。

在智能原生时代，响应这个请求的不再是单一的、庞大的后端服务。取而代之的，可能是一个由多个专业Agent组成的动态团队，如下图3所示：

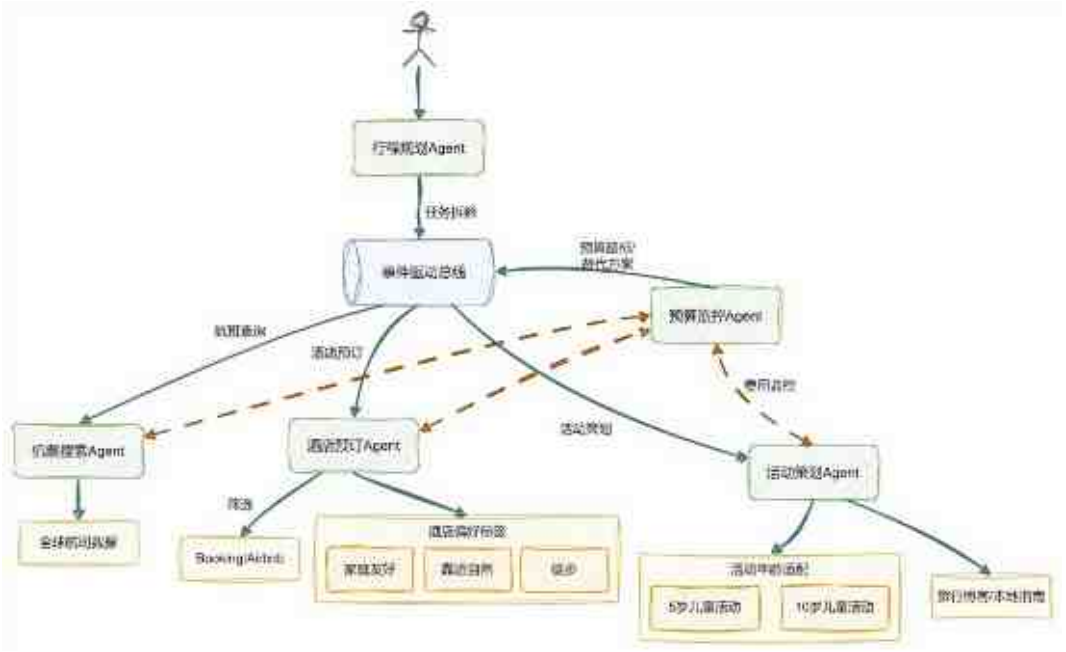


图3

行程规划Agent作为总指挥，理解用户意图并拆解任务。它唤醒机票搜索Agent，后者实时比对全球航司数据，寻找性价比最高的航班。同时，酒店预订Agent正在根据家庭友好、靠近自然、徒步等标签，筛选Booking和Airbnb上的住处。活动策划Agent则深入挖掘旅行博客和本地指南，为5岁和10岁的孩子分别设计有趣的每日活动。一个永远在线的预算监控Agent会像个会计，确保所有开销都在红线之内，并能在超支时提出预警和替代方案。

这还仅仅是一个用户的单一请求。假设我们的平台同时服务成千上万的用户，每个用户都有着独一无二的定制需求。在那一刻，后台将运行着成千上万个不同职能的Agent，它们之间需要进行大量的、动态的、非确定性的交互。谁完成了任务？下一步该通知谁？预算有变动时，如何让所有相关Agent同步调整策略？

如果靠传统的硬编码或直接API调用来组织它们，结果将是一场灾难——一个脆弱、混乱、无法维护的“数字巴别塔”。这便是智能时代给我们带来的核心挑战 - 如何从管理代码和服务，进化到治理拥有自主性的智能体。

为了驾驭这场从**代码为中心**到**智能体为中心**的范式革命，我们需要一个全新的基础

设施层。它必须超越简单的服务发现与调用，转而提供一个专为智能体设计的协作与治理框架。基于这一深刻洞察，我们从云原生时代的演进中汲取灵感，借鉴了服务网格在**关注点分离**上的经典架构范式，提出并联合社区构建了下一代智能应用的灯塔Agent Mesh。

服务网格解决了微服务之间的流量与治理难题，Agent Mesh要构建的是智能体之间的**智能协作网格**。它并非一个简单的Agent注册中心，而是覆盖Agent定义、调试、部署、监控、协作与治理全生命周期的**神经中枢**和**交通总线**。

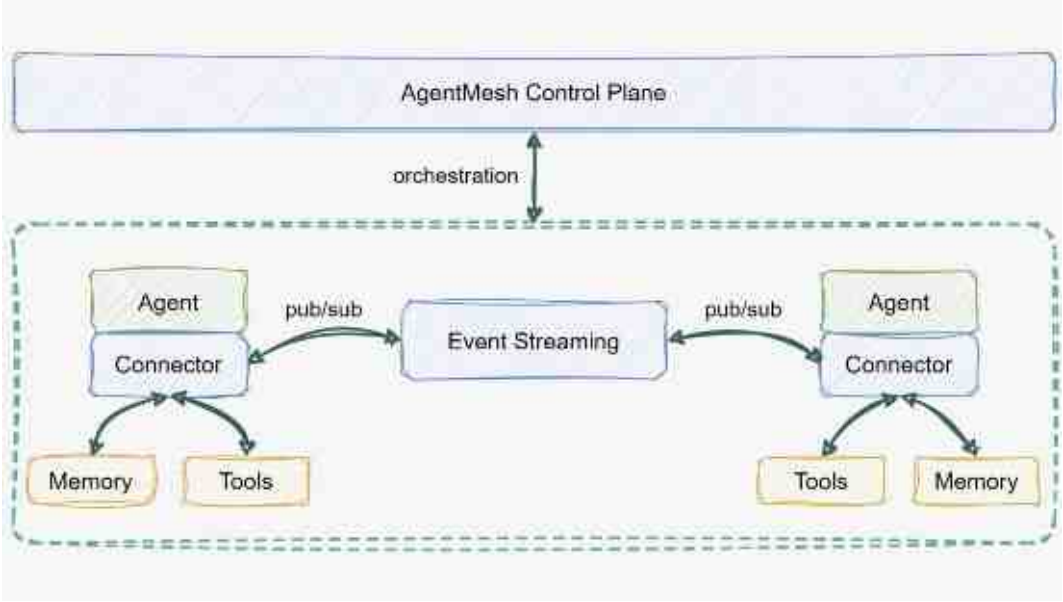


图4

如上图4所示，Agent Mesh的核心由三大组件构成：

声明式的控制平面

如同每一艘远航的船只都需要在港口注册登记，每一个Agent也必须拥有一个声明式的、版本化的身份档案。我们可以选择继续扩展Kubernetes CRD，将Agent、Tool、PromptTemplate等核心概念抽象为AI原生的API资源。这为整个网格建立了一套可靠的户籍管理系统。开发者可以通过熟悉的YAML文件或可视化界面，轻松地注册并配置一个Agent，为其赋予清晰的身份id、能力skill与行为准则prompt，这是实现规模化治理的第

一步。

事件驱动的协作总线

这正是Agent Mesh的灵魂所在。我们深刻地认识到，智能体之间的协作，绝不能是硬编码的直接调用链，那只会造就僵化而脆弱的数字作坊。同时，对于一个能应对海量复杂任务的Agent系统，Agent的数量可能非常庞大，并且会随着时间、任务的变化而快速变化。Agent之间的访问，在同步调用的方式下，也将面临调用关系复杂、感知不及时、容易受网络等波动影响等问题。**真正的智能协同，应当是松耦合、异步且充满动态性的。**

为此，我们构建了一个事件驱动的协作总线。当一个Agent完成任务或产生一个重要的中间结果时，它会向总线发布一个携带上下文的事件。其它对此感兴趣的Agent可以订阅这些事件，并以此为触发器，自主启动自己的后续任务。这种机制，彻底解耦了Agent之间的直接依赖。Agent互访，从关心谁来处理转为关心处理什么类型的任务，使得构建复杂的、非确定性的、可动态演化的多Agent协作流程成为可能，让涌现出的智能协作不再是奢望。

智能连接器“市场”

如果说协作总线是奔流不息的数据动脉，那么智能连接器就是植入每个Agent体内的、标准化的智能接入端口。它让Agent能够说总线能懂的语言，并听懂来自总线的声音，是Agent融入协作网络不可或缺的组成部分。这是一个高度可插拔的插件体系，它作为Agent与平台核心服务之间的**智能翻译官**和**全能助理**，主要打通三大关键交互：

- **赋能事件交互**

连接器为Agent提供了一个极简的编程界面，将底层的消息引擎（如Kafka、RocketMQ）客户端复杂性完全屏蔽。它自动处理事件的序列化、标准化，并确保可靠投递。Agent只需表达我要发布一个成果的意图，连接器会处理剩下的一切。

- **无缝存取记忆**

它提供记忆插件，让Agent可以直接通过简单的开放标准接口将思考结果存入

OpenMemory、MemoX等记忆系统，或在启动时自动从记忆系统加载所需上下文，而无需关心底层数据存储的细节。

- **安全调用工具**

当Agent需要使用工具时，工具调用插件会接管请求。它通过AI网关自动处理服务发现、身份验证、API参数适配和安全审计，让Agent只需声明意图，即可安全地调用外部世界的能力。

智能连接器不仅是管道，更是主动的赋能者。它通过**协议转换**、**上下文注入**、**数据标准化**等能力，将Agent从繁琐的底层交互中彻底解放，使其能专注于**思考**本身，而非**接线**。

当前市面上主流Agent框架，或因API频繁变更，或因追求大一统而牺牲专业深度，尚难以支撑企业级应用的严肃构建。为此，我们选择了一条更聚焦、更稳健的路径，立足于Java庞大的企业级生态和开源社区，我们提供了一个深度集成Agent Mesh核心能力的增强版Spring AI框架，为开发者带来开箱即用的原生体验。开发者使用简单的注解来发布Agent的成果和订阅其它Agent的事件。通过零配置的自动装配，直接注入MemoXTemplate来读写记忆。像调用本地方法一样安全地使用通过AI Gateway注册的任何工具。同时，为构建一个开放、繁荣的智能体生态，我们正将过去在工业级开放协议设计中积累的深厚经验，倾注于前瞻性的A2A（Agent-to-Agent）协议构建之中，致力于与社区共同定义未来智能体的**协作标准**。

Agent Mesh的架构蓝图，凝练了我们在多个技术领域长期探索的深刻洞察与工程实践。其设计哲学传承并演进了云原生时代的最佳实践：

- 控制平面的设计思想，直接脱胎于我们对服务网格治理体系的深刻理解。我们认为，管理成百上千个自主Agent的复杂性，与治理大规模微服务在哲学上是相通的，这为我们构建Agent的策略、身份与可观测性体系奠定了理论基础。
- 协作总线的实现路径，则根植于团队在事件驱动架构领域的丰富积累。无论是对Kafka、RocketMQ等主流消息中间件的驾驭能力，还是构建高吞吐、低延迟、轻ETL数据通道的工程经验，都构成了这套协作体系的坚实骨架。
- 智能连接器的插件化生态，是我们对平台化与可扩展性长期坚持的产物。**唯有开**

放，方有生态。这一宝贵认知，塑造了Agent Mesh的架构之魂。一个高度模块化、可扩展的设计。它确保了任何一个组件，无论是模型、工具还是数据，都能被灵活地替换与增强，从而使整个架构能与日新月异的AI浪潮同频共振，永葆生机。

灯塔二：认知记忆平台，典型代表MemoX

当前所有大语言模型最大的瓶颈之一就是上下文长度限制。无论是GPT-4的128K，还是Gemini 1.5 Pro的1M，巨头们投入巨额研发成本来扩展这个临时记忆窗口。由此可见记忆对于维持对话、理解任务的极端重要性。但窗口再大，也只是临时记忆，无法形成长期记忆。**而这个模型层的技术天花板，直接导致了智能体在功能层面先天不足的宿命**。

一个没有记忆的智能体，就像一只只有七秒记忆的“数字金鱼”。它无法从过去的成功或失败中学习，无法理解任务之间的关联，更无法形成对一个领域长期的、结构化的认知。这使得它永远停留在只能执行简单、孤立指令的工具层面，而非真正的自主智能。在提出MemoX架构之前，我们深入分析了当前记忆系统普遍存在的几大核心挑战：

记忆孤岛

业界当前大多数实现中，每个Agent都维护着自己独立的记忆文件，如一个本地向量库或JSON文件。它们各自为战，A智能体犯过的错，B智能体全然不知，C智能体刚学到的知识，D智能体无法利用。真正的智能是**集体智能**。缺乏**共享的、全局的**记忆中枢，是限制多Agent系统能力上限的根本瓶颈。

扁平检索

当前主流记忆方案过度依赖于向量相似度检索。这能解决“找到相似文本片段”的问题，但无法回答“A和B是什么关系？”、“导致这个结果的关键决策链是什么？”等深层次的关联问题。记忆是扁平的，缺乏结构和因果。人类的记忆不仅是事实的堆砌，更是由无数关系、因果、概念组成的知识图谱。高效的记忆系统必须同时具备**相似度感知**和**关系感知**的能力。

不可移植

许多原型级的记忆系统是临时的、非托管的。它们与Agent进程绑定，一旦重启、迁移或扩容，记忆就会丢失或不一致。这在生产环境中是不可接受的，无法承载任何严肃的企业级应用。我们认为，记忆是Agent系统中最宝贵的有状态资产，必须以企业级的标准来设计和运维，保证其高可用、持久化和可扩展性。

基于以上洞察，我们提出并设计了MemoX，它并非一个单一的分布式存储系统，而是Agent Mesh平台的核心认知中枢，一个为集体智能而生的有状态记忆服务。其本质是扩展大语言模型固有的无状态性和有限上下文窗口，通过模拟人类记忆过程（更新、提取、巩固、反思、遗忘等）来构建对外接口。它是一个分层的、共享的、统一的、支持“语义+关联”多重检索的认知记忆中心。如果说etcd是Kubernetes集群管理配置的“短期突触”，那么MemoX就是整个AI应用生态沉淀智慧的“长期海马体”。如下图5所示：

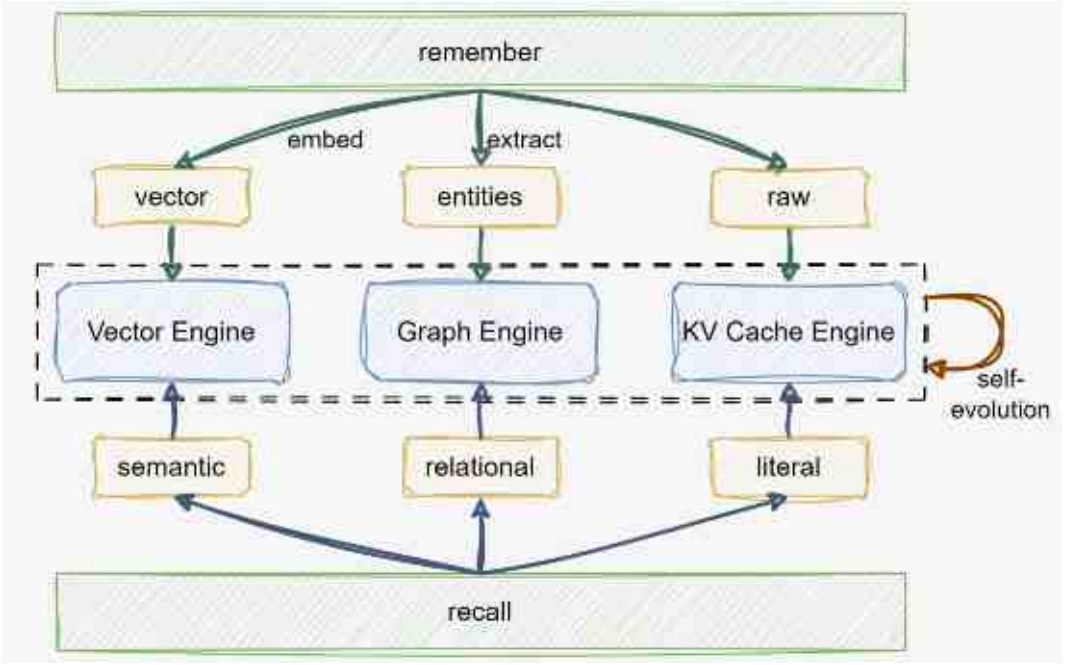


图5

当前业界主流的存储引擎如KV、图、向量、**流存储**等都不约而同的在记忆系统方面发力。MemoX的复合型记忆架构，并非对单一存储技术的盲从，而是我们基于对不同记忆类型所需数据模型的判断，做出的一种审慎的**工程选择**。我们相信，**强大的认知能力**

源于对不同类型信息的专业化处理。为此，我们将多种引擎进行有机融合，各司其职。

向量引擎用于实现高效的语义相似度检索。想象这么一个场景，一位资深客服Agent收到一条新的用户抱怨：“升级新版本后，我的应用总是卡住不动，然后自己就退出了！”Agent需要立刻判断这是否是已知问题。此时，向量引擎开始工作。它不会进行僵硬的关键词匹配，而是理解“卡住不动”、“自己退出”背后的语义本质，并从海量的历史工单中，精准召回三天前另一位用户描述的“软件界面无响应，并发生意外进程终止”的故障报告。尽管措辞迥异，但语义高度相关。这让Agent能瞬间将新问题与已知故障关联，极大提升了响应效率。

图引擎用于分析实体、事件间的关联关系，处理逻辑与关联。假设一个DevOps分析Agent正在复盘一次严重的“线上发布失败”事件。简单的日志检索只能看到表象：“支付服务启动失败”。但真正的根源在哪里？Agent向MemoX发起一个关联查询。图引擎开始在记忆网络中穿梭，它清晰地揭示了这样一条隐藏的关系链，并找到相关联的关键信息，如下图6所示：

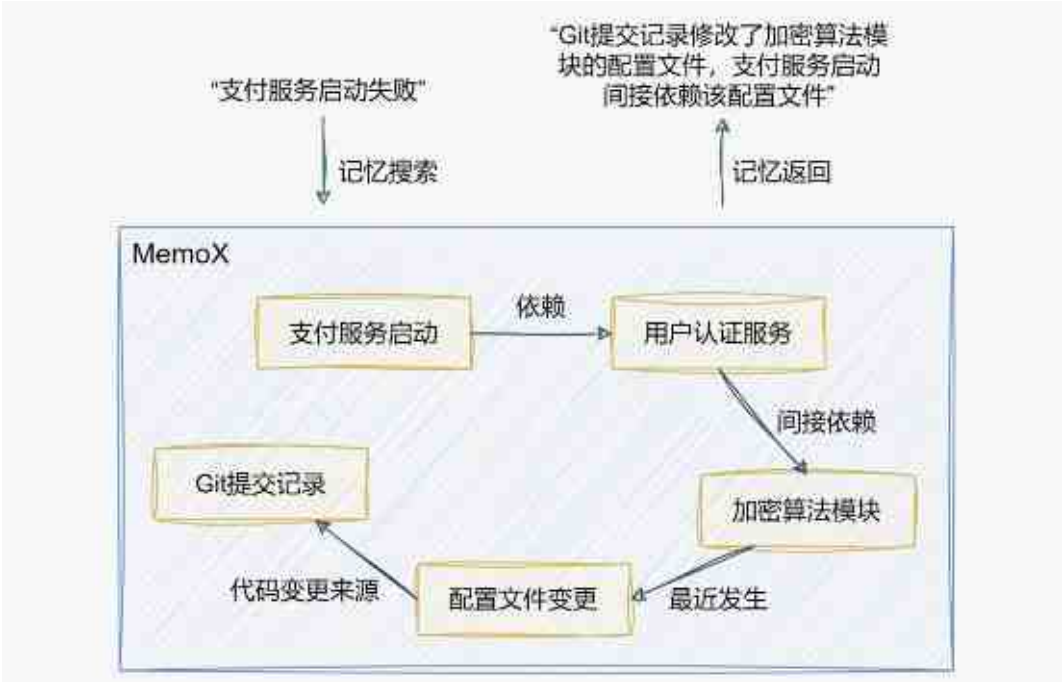


图6

传统模式下，当支付服务报警时，运维人员需要登录跳板机，查看支付服务的日志，

发现它依赖的用户认证服务挂了。再去找负责认证服务的团队，他们查日志，发现是健康检查失败。再查监控，可能发现是某个配置变更导致的。这个过程可能需要跨越多个团队、查询多个系统（日志系统、监控系统、Git历史），耗时几十分钟甚至几小时。图引擎将这条长达五步的因果链瞬间呈现，让Agent得以洞察藏在表象之下的真正根源。

KV Cache引擎用于处理当时与即时。其毫秒级的热数据存取，作为Agent的工作记忆在合适不过。在智能编码领域，一个代码生成Agent正在根据需求编写一个复杂的算法。在它的脑海中，必须时刻保持对几个关键信息的瞬时访问：用户的原始需求描述、目标数据库的表结构、以及刚刚生成并通过测试的上一个函数签名。这些信息在任务执行期间会被频繁、反复地调用。将它们存入KV缓存，就如同置于一块数字便签上，确保Agent在需要时能以近乎零延迟的速度获取，从而保证了其思考过程的连贯与流畅。

通过这套组合拳，MemoX能同时理解“内容像什么”（向量）、“事物如何连接”（图）以及“眼下最重要的是什么”（KV缓存），从而提供远超任何单一扁平检索系统的记忆深度与效率。

Karpathy最近分享了一个很有意思的理念，**人类糟糕的记忆，是一种特性，不是缺陷。AI记得一切，却学不会抽象。人类遗忘很多，却能举一反三**。记忆不是静态的，而需要动态演化，需要借助包括检索、推理这类“举一反三”行为更好的**强化认知核心**。当对话过长时，我们可以触发LLM对近期内容生成摘要，用精炼的摘要替代原始冗长的文本，释放上下文空间。不仅如此，MemoX会定期回顾已有的记忆，结合记忆使用的情况，将信息进行抽取、整合，以提升记忆的准确性和访问效率。

记忆代表了Agent系统的知识库，一个真正“聪明”的记忆系统，应该具备自主学习的能力，能够从现实世界不断学习和更新自身的知识。MemoX会通过外部交互工具，如Agent Mesh的智能连接器，适时地获取最新的知识，更新、纠正其认知。

另外，借鉴成熟的Kubernetes Operator开发经验，我们可以将MemoX打包成一个高可用的有状态服务集合。这实现了一键部署、故障自愈、在线扩容和数据备份恢复，将复杂的有状态服务运维难度降至最低，确保了MemoX在生产环境中的稳定性和可靠性。

灯塔三：零信任运行舱，典型代表Agent Runtime

当整个行业都在为Agent的巨大潜力而兴奋时，一个普遍的实践鸿沟也随之出现。大多数团队试图用过去为确定性任务设计的工具，去驾驭充满概率性的Agent。这带来了两个核心痛点：

控制流的错配

传统的DAG工作流引擎如同精密的流水线，为的是执行可预测、线性的任务。而Agent的“思考 - 行动”循环本质上是动态的、非线性的，更像一场即兴辩论。用僵化的“剧本”去指挥“即兴辩手”，不仅会扼杀Agent的自主性，更会在遇到意外情况时导致整个流程的脆弱与崩溃。

安全与信任的真空

LLM驱动的Agent，其行为尤其是生成的代码，具有不可预测性。让这样一个黑箱直接在不受控的环境中执行文件操作、发起网络请求，无异于将一个权限过高的实习生直接接入生产服务器。这种失控的风险，是任何严肃的工程团队都无法接受的。

面对这些挑战，我们发现问题的根源在于，大家普遍混淆了宏观的**战略协作流**与微观的**战术执行环**。战略协作流关乎团队目标是什么以及任务如何拆解和分配。战术执行环则关乎单个Agent如何独立思考、调用哪些工具来完成分配给它的任务。将这两者耦合在一起，用同一个引擎管理，是导致混乱和风险的根源。真正的解决方案，必须在架构层面将二者解耦。

基于以上洞察，我们需要提供一个专为Agent战术执行环设计的、完全托管的运行环境——Agent Runtime。它不再试图控制Agent的每一步思考，而是为其提供一个功能完备、安全隔离的智能工作室。在这里，Agent可以自由地进行自己的“思考 - 行动”循环，而高层的协作系统只需关心它是否完成了任务这一最终结果。

做个类比，如果说宏观的协作编排系统是Kubernetes Scheduler，负责决策。那么Agent Runtime就是每个节点上的Kubelet+Container Runtime的组合体。它接收上层指令，为单个Agent的执行提供一个标准的、安全的运行环境，管理其从启动、执行到终止的

全过程。为了做到这一点，我们重点打造了其两大技术底座，如下图7所示：

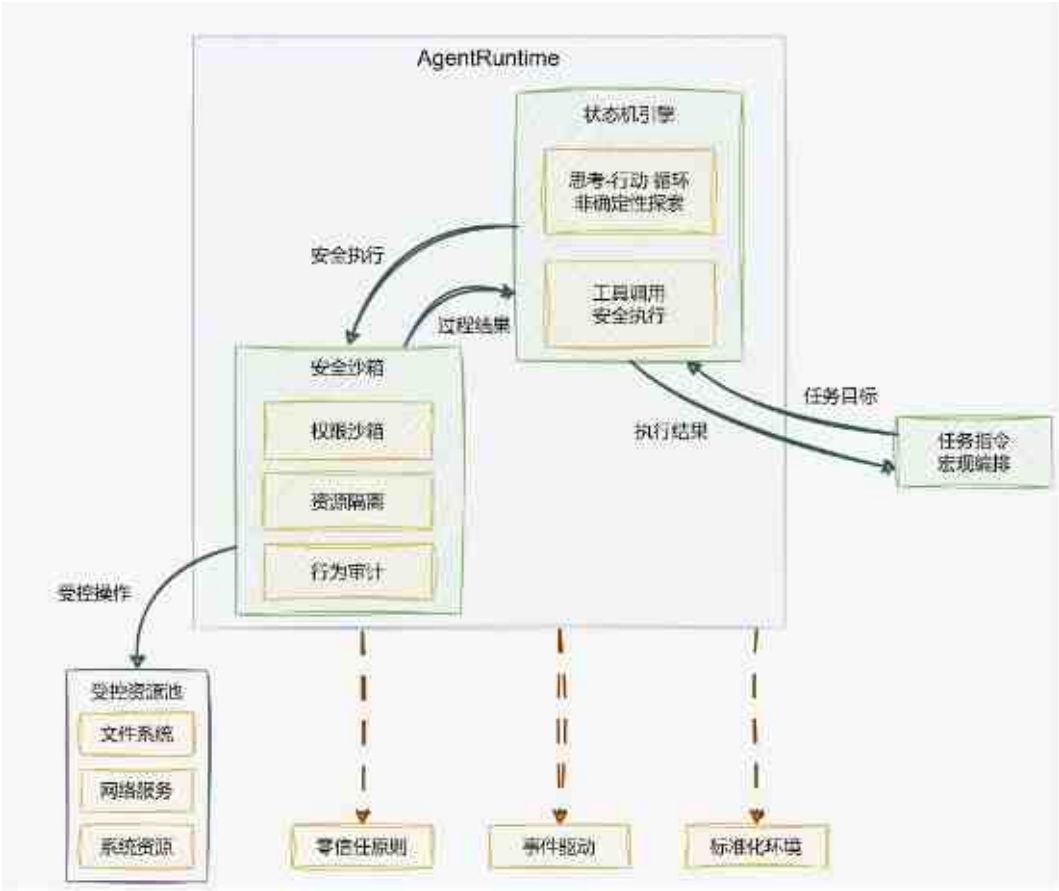


图7

事件驱动的状态机引擎

Agent Runtime的核心是一个轻量化的状态机。它不预设Agent的执行路径，而是通过事件来驱动Agent的思考行动ReAct流程。当一个Agent需要调用工具时，Runtime会捕获这个意图事件，在安全的沙箱中执行该动作，再将结果作为新的观察事件反馈给Agent，触发其下一轮思考。这种设计完美拥抱了Agent的非确定性，提供了极高的灵活性。

纵深防御的安全沙箱

每个Agent实例都被严格限制在一个由MicroVM或者WASM构建的轻量级沙箱中运行。

这意味着，默认情况下，Agent无权访问任何文件系统、网络或系统资源。所有权限都必须被显式授予。这种零信任执行模型，从根本上杜绝了Agent行为失控带来的风险，确保即使Agent产生幻觉或恶意代码，其破坏力也会被牢牢地锁在笼子之内，为平台提供了企业级的安全保障。

灯塔四：开放生态平台，典型代表AI Gateway

Agent的价值在于它能够连接并操作现实世界的数字服务。然而，当我们试图将强大的Agent接入丰富的外部生态时，当下可能会立刻陷入一片混沌之中。这种混乱体现在三个层面：

模型的战国时代

如今的大模型领域百花齐放，从GPT系列、Gemini系列、Claude系列、DeepSeek系列、QWen系列，到其它各种开源及私有化模型，每个都有不同的API接口、计费模式和性能表现。直接在代码中与这些异构模型硬编码耦合，不仅会导致技术债堆积如山，更让成本优化和统一安全审计成为不可能完成的任务。

工具的认证噩梦

为了让Agent能够预订会议室、查询订单或者发送邮件，它必须持有并管理大量第三方服务的API密钥和凭证。这种密钥散落一地的模式，是一个巨大的安全黑洞。一旦Agent的Prompt或中间代码被泄露，所有关联服务的权限都可能瞬间失控。

边界的治理无序

每次对外部模型或API的调用，都意味着一次成本、一次潜在的延迟、一次数据交换。在缺乏统一管控的情况下，我们无法回答诸如“哪个业务消耗了最多的模型预算？”、“哪个第三方API是性能瓶颈？”、“我们的敏感数据是否通过Prompt泄露给了外部模型？”这类关键的治理问题。由此可见，Agent与外部世界之间当前缺乏一个强有力的、统一的交互边界。我们必须在平台与生态之间建立一个“海关”来管理这种可能的无序，构建一个类似国家海关的中央枢纽 - AI Gateway。所有进出平台的请求，无论是调用外部大模型，还是使用外部工具API，都必须经过这个关口进行检查、认证、

记录和重路由。这个海关不仅保障了平台的安全，也极大地简化了Agent与世界沟通的方式。

对于Agent而言，AI Gateway是统一的“五官”和“双手”。Agent无需再关心每个模型、每个工具的具体接口细节和认证方式。它只需用意图（如调用模型服务）与Gateway交互，Gateway会负责搞定剩下的一切。对于平台而言，AI Gateway是坚固的“防火墙”和“审计署”，是平台与外部世界交互的唯一出入口。它集中解决了认证、授权、安全、成本控制和可观测性等所有关键的治理问题，为整个平台的稳定和安全提供了核心保障。既有API网关的路由、认证、限流能力，又具备多云管理平台那种对异构后端资源（这里是LLMs和Tools）进行**统一纳管**和**成本优化**的能力。

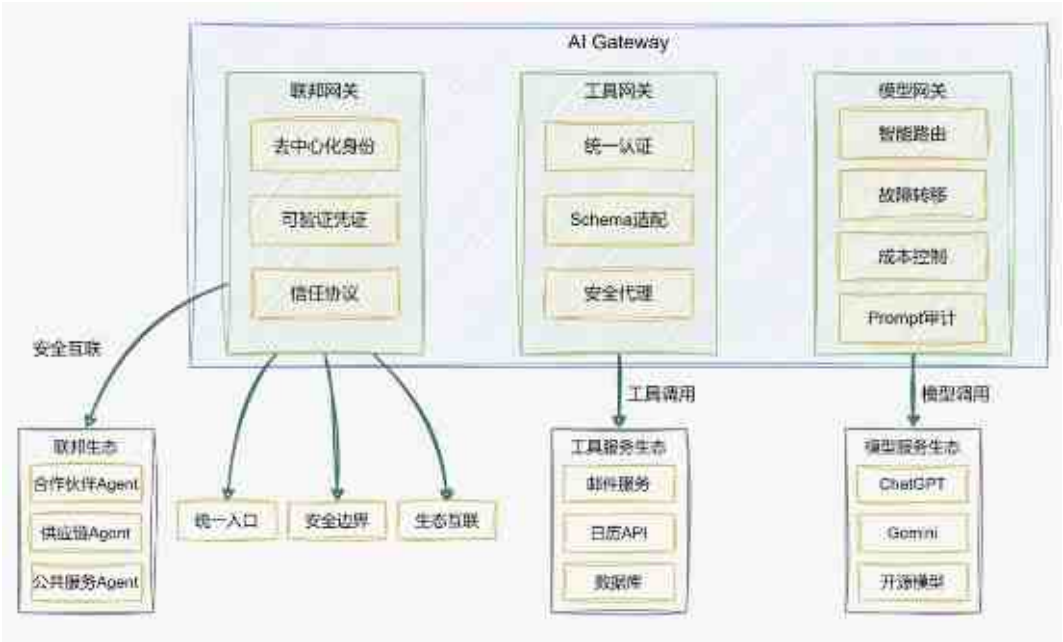


图8

如图8所示，整体上看，我们通过构建三维一体的AI网关能力中心，重新定义了企业AI服务的交付、管理与安全范式。AI网关从单一通道，升维为集**连接**、**增强**、**治理**于一体的立体化能力中心。它主要由以下三部分组成：

模型网关

这是访问所有LLMs的统一入口。它提供了四大关键能力：

智能路由与负载均衡

在生产环境中，团队往往会根据任务复杂性、成本、延迟、模型能力或法规等因素，动态地将请求路由到最合适的模型端点。

- **故障转移**

当某个模型服务不可用时，自动切换到备用模型，保证业务连续性。当然，这个能力可以跟Agent Mesh联动，进行所谓的可靠性编排。

- **成本控制与统一缓存**

对模型调用进行精细的预算管理和用量限制，并对高频、相似请求进行缓存，大幅降低整体调用成本。

- **Prompt审计与合规**

记录所有流经网关的Prompt和响应，用于安全审计、数据防泄露和持续的Prompt优化。

工具网关

相当于一个中心化的**企业工具市场**。我们将所有外部API封装成标准化的工具并注册到此处，同时提供。

- **统一认证与授权**

Agent不再直接持有任何API密钥。所有的认证凭证都由网关集中管理和安全存储。它会基于Agent的身份和任务上下文，动态地为其赋予临时的工具使用权限。

- **Schema适配与安全代理**

网关负责将外部多样化的API接口规范，适配成LLM易于理解的统一格式，并作为安全代理执行所有调用，避免Agent直接暴露在公网上。

联邦网关

随着多Agent系统在企业内外的普及，可以预见，Agent技术的终局绝非孤立的企业大脑，而是会演化为一个跨越组织边界、可互操作的智能互联系统。这不禁引发了我们一个终极的思考：如何在没有中央权威的开放网络中，建立Agent之间的信任？为此，我们提出了联邦网关这一核心概念，它尝试解决这个数字世界中的**信任外交**，最终演变为一个支持去中心化身份与可信数据交换的协议网关。为此，我们为联邦网关提供以下核心功能，用来构建下一代Agent互联的基础设施：

- **去中心化身份标识**

我们将为平台上的每一个Agent（甚至每一次重要任务）分配一个全局唯一的、由组织自身控制的ID。这就像为每个Agent颁发了一个加密的数字护照。这个护照不依赖于任何第三方身份提供商，其所有权和控制权牢牢掌握在企业自己手中。当一个Agent需要与外部Agent交互时，它出示的是自己的ID，而非易于被伪造的简单API Key。

- **可验证凭证VC（Verifiable Credential）**

光有护照还不够，还需要证明这个护照持有者的资格和授权。可验证凭证技术就扮演了这个角色。AI Gateway可以为Agent颁发经过加密签名的VC，用以证明其身份、能力或授权。当Agent向外部伙伴发起请求时，它可以出示这些VC。对方的网关可以通过密码学验证这些凭证的真实性和完整性，而无需反向查询我们的系统，从而建立起高效、可信的零知识证明式交互。

通过集成上述能力，联邦网关实现了从**基于密钥的认证**到**基于密码学信任的授权**的根本性升级。

灯塔五：智能观测平台，典型代表Agent Insight

在生产环境中运行Agent，开发者和运维人员很快会发现他们正面临一个前所未有的挑战。传统应用性能监控APM工具，在Agent面前几乎完全失效。在过去，当一个传统服务出错，我们可以通过堆栈信息和日志精确定位到出错的代码行。但当一个Agent给出了错误的答案或采取了非预期的行动时，我们看到的是什么？

- APM视角 - 服务正常，对OpenAI API的调用返回了200 OK。
- 日志视角 - 一堆无序的、非结构化的Prompt和JSON输出。
- 开发者视角 - 我不知道它为什么会这么想！它上一步还好好的，怎么下一步就精神错乱了？

Agent的内部决策过程，就像一个无法打开的黑盒，充满了不确定性。我们无法追溯其心路历程，无法诊断其思维谬误，更无法量化其决策成本。这种失控感是阻止Agent大规模应用于生产环境的最大障碍之一。传统的可观测性关注的是代码执行，而Agent时代的可观测性必须升级到**关注认知过程**。我们不能再满足于监控CPU和内存，我们必须能够监控Agent的思考、决策和学习，必须能将Agent内部抽象的思考 - 行动循环，转化为一个具体可度量、可追踪的工程实体。我们更需要为Agent的思维链配备一个**飞行记录仪**Flight Data Recorder，类似JDK自带的针对Java应用的诊断与分析工具。

为此，我们构建了Agent Insight，一个专为Agent和LLM应用设计的全新可观测性解决方案。它的核心价值在于，为Agent的自主决策与执行链路提供了端到端的**调用链追踪**与**行为归因**能力，实现了从意图到结果的全流程可追溯性与可审计性。Agent Insight是为AI量身定制的下一代可观测性平台。它将分布式链路追踪、时序度量和结构化日志的核心思想进行了深度融合与智能化升级，创造出一个统一的面向AI行为的分析平面。

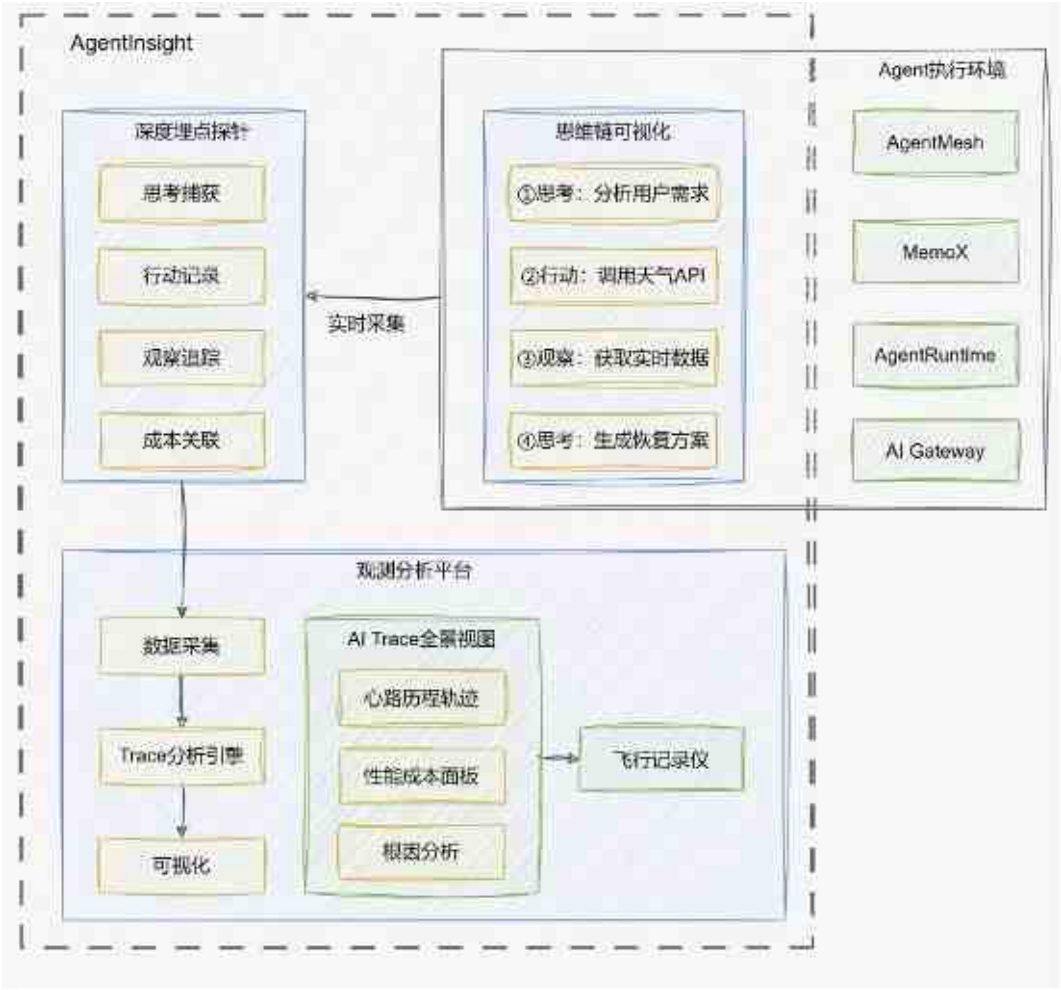


图9

如上图9所示，我们将数据采集探针深度植入到几大核心脉络 - AI Gateway、Agent Mesh、MemoX和Agent Runtime之中。这种原生集成的方式，让数据采集不再是外部的、猜测性的，而是内在的、事实性的。针对Agent的核心驱动循环进行深度埋点，Agent的每一次思考、每一次行动决策、每一次从环境中获取的观察，都会被自动捕获为一个结构化的事件，并携带一个唯一的TraceID。这些事件串联起来，就形成了一条完整的、可视化的**心路历程轨迹**。用户不再是面对一堆杂乱的日志，而是能像看电影分镜一样，清晰地回顾Agent从接收任务到得出结论的全过程，洞悉其每一步的所思所想。

当Agent的行动需要与外部世界交互时，请求会携带TraceID流经AI Gateway。AI Gateway在执行请求的同时，会将关键的性能与成本指标，如Token消耗、模型调用延迟、API

执行成本、外部服务状态码等与这个TraceID进行关联，并回写到Agent Insight的数据中心。最终，开发者获得的是一个前所未有的上帝视角。在可视化的AI Trace上，他们可以：

点击任何一步思考，立刻看到其关联的Token成本和金钱花费。

筛选出所有高延迟的行动，快速定位是模型响应慢，还是某个第三方工具API成为了瓶颈。

对比两次任务的Trace，一次成功一次失败，清晰地看到是哪一步的思考或观察出现了分歧，从而实现A/B测试和根本原因分析。

Agent Insight让调试Agent不再是猜谜游戏，而是科学的诊断分析，为Agent在企业中的可靠运行提供了最后、也是最关键的一块拼图。最近在硅谷创投圈一些新的研究方向如**上下文可观测**也被大家普遍提到，哪些输入能持续提升输出质量，哪些上下文会导致模型幻觉等等，这些问题当前还都处于摸索阶段，这也是Agent Insight后续需要重点关注与解决的。

小编结语

当下，企业对AI的价值预期从探索走到现实价值，与企业期待形成对比的是，AI原生应用的大规模落地仍然面临诸多的挑战。一个具备强大认知能力的大脑，并不总能独立解决现实世界中的问题。而承载着大模型与AI应用运行的基础设施，是AI融入产业的关键基础能力。纵观全文，文章所描绘的AI原生应用基础设施蓝图 - 无论是作为流量入口的AI网关，还是保障Agent可靠运行的观测与治理体系 - 其核心价值都**在于将AI系统的复杂性与不确定性进行封装，为上层业务应用提供稳定、高效、安全的调用接口**。

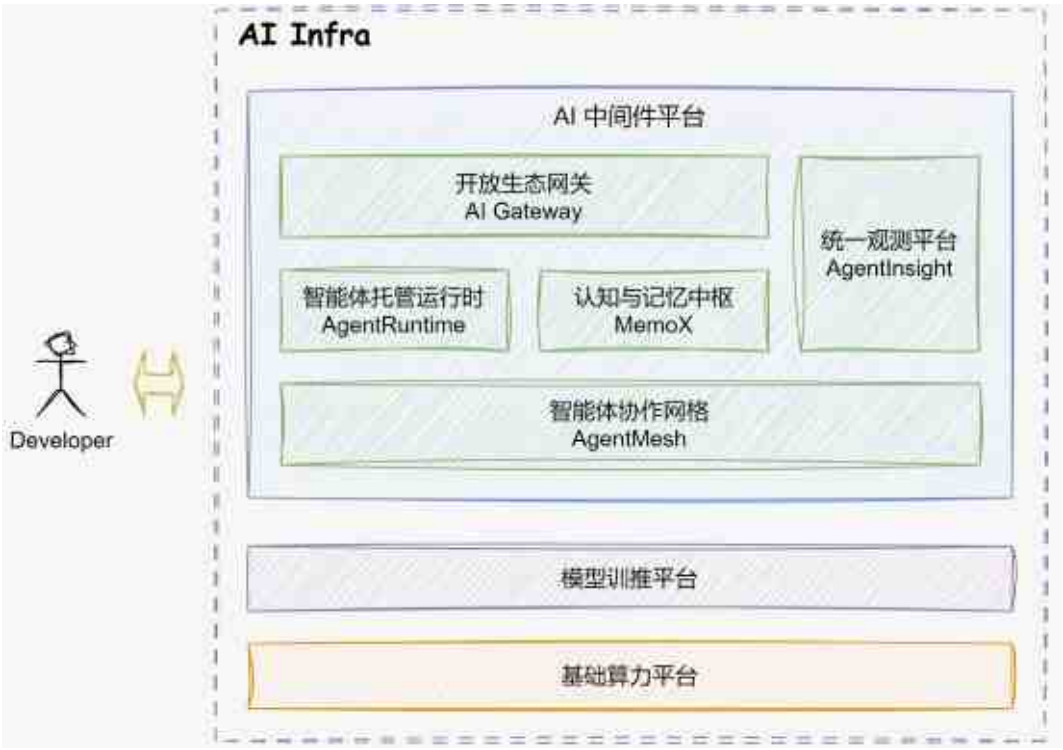


图10

如上图10所示，这，实质上就是在重新定义和构建属于AI时代的中间件与PaaS平台。它们不再是传统意义上的管道与容器，而是演化成了具备**感知**、**决策**和**治理能力**的智能应用基础设施层。

展望未来，这一层的成熟度将直接决定AI原生应用的普及速度。我们期望和所有学术与产业从业者一道，打造开放、标准、高效的AI原生应用基础设施平台，让创新不再被底层的复杂性所束缚，共同开启一个真正由**AI驱动的应用新纪元**。

Fluss湖流一体：Lakehouse架构实时化演进

演讲嘉宾 罗宇侠 编辑 Kitty

湖仓一体Lakehouse凭借其开放性和成本效益，已经成为当今数据平台的主流架构。然而在LakeHouse架构下，数据的时效性最多只能达到分钟级。为了满足企业对实时数据分析的需求，通常需要额外引入秒级延迟的流存储（如Kafka）。流存储游离在LakeHouse架构外，带来的成本、一致性、治理等问题都面临很大的挑战。

在InfoQ举办的QCon全球软件开发大会（北京站）上，阿里云高级开发工程师罗宇侠做了专题演讲“Fluss湖流一体：Lakehouse架构实时化演进”，他分享了流存储和LakeHouse架构割裂的现状下用户面临的问题和挑战，以及目前业界在融合两者上的趋势。并介绍了流存储Fluss如何完美地融入进LakeHouse架构，无缝地将LakeHouse架构进行实时化改造，深度解析其技术架构和原理，分析该架构相比割裂地使用流存储和

LakeHouse架构能带来的收益。最后他还分享了基于Fluss来构建实时LakeHouse架构的最佳实践。

以下是演讲实录（经InfoQ进行不改变原意的编辑整理）。

AI时代湖流融合的趋势

在AI时代，湖流融合是一个重要的趋势。一方面，Lakehouse架构的重要性日益凸显。Lakehouse架构是开放的，能够存储多模态数据，包括结构化数据和文本、图像等非结构化数据，满足大模型训练对多模态数据的需求。同时，Lakehouse架构通过统一的Catalog管理所有数据，无论是结构化还是非结构化数据，都可以进行数据溯源和权限管理，保证大模型训练数据的质量和可追踪性。其底层依赖廉价的存储，如OSS、S3等，这些存储方式成本低，能够应对大模型时代数据量指数级增长的需求。

另一方面，AI时代需要实时数据，实时数据的重要性体现在多个方面：一是知识库可能会滞后，实时数据能够实时更新数据库和知识库，提升模型的准确性；二是实时数据能够感知用户的上下文，通过传感器等实时获取用户当前状态，从而更精准地满足用户需求；三是实时数据支持模型的在线学习，即模型可以实时进行机器学习；四是实时数据有助于模型根据用户的反馈迭代推理能力。

在湖流融合的业界趋势方面，目前有多种发展方向。例如，Kafka背后的商业化公司Confluent提供了Tableflow，它将Kafka中的业务数据实时导入到数据湖中，其实它就是一个Flink作业，但用户无需手动管理Flink作业。Red Panda是一家估值十几亿美元的创业公司，提出了Iceberg Topic，将数据转化为Iceberg湖格式，便于分析引擎进行高效查询和OLAP分析。AutoMQ是国内一家新兴的创业公司，提出了Table Topic，将Kafka数据转化为Iceberg湖格式，支持直接在Iceberg上进行分析。Pulsar背后的商业化公司StreamNative也提出了类似的思路，这些公司都在积极探索湖流融合的方向。目前，Lakehouse架构已经在多家公司落地应用，未来Lakehouse和实时数据的融合将越来越紧密。

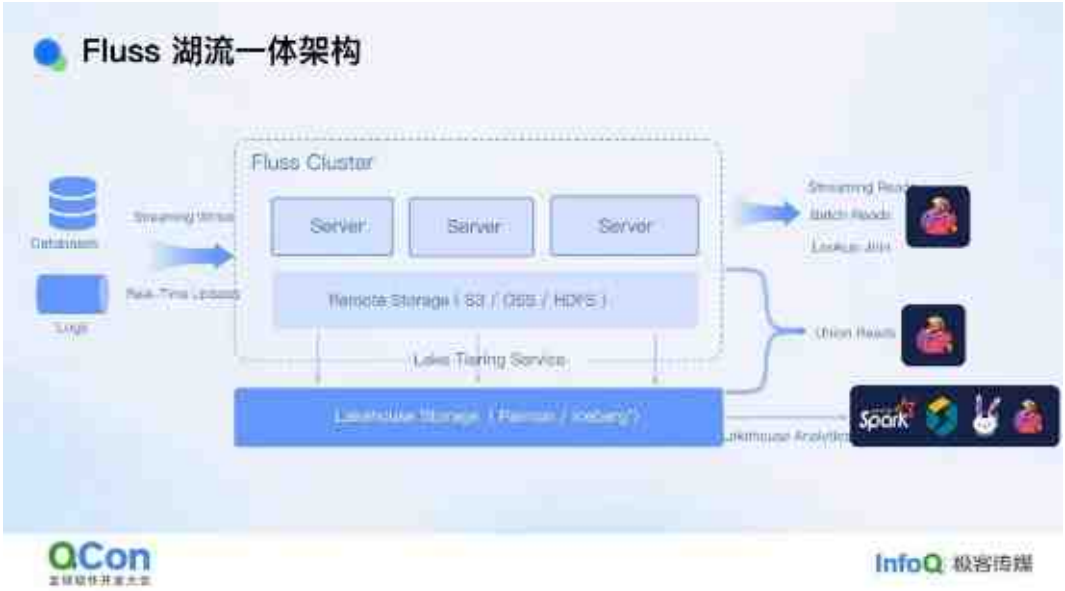
目前这些融合方案大多是基于已有的消息队列进行融合，这就导致与湖的融合或多或少不够直接。

例如Kafka是无Schema的，无分区概念的，而湖格式却有Schema和分区的概念，消息队列和湖很难对齐。因此，我们提出了一种从一开始就面向数据湖设计的实时存储方案Fluss。

Fluss湖流一体架构和技术解析

Fluss湖流一体架构

Fluss湖流一体的架构如下所示：数据可能来源于消息队列、业务数据库或业务数据，这些数据进入Fluss集群。在Fluss集群中有一个Lake Tiering Service，其作用是将数据从Fluss集群实时同步到数据湖中。这种同步过程被称为Tiering（分层），而不是简单的Copy。通过Tiering，实时数据被写入数据湖，之后可以使用Spark，StarRocks，Trino等工具直接在数据湖上进行高效分析。此外，Fluss还提供了一个Union Read（联合读取）功能，即将Fluss集群中最新的实时数据与数据湖中稍早的数据（例如3分钟前的数据）进行联合，从而得到最新的数据视图。



Fluss表模型

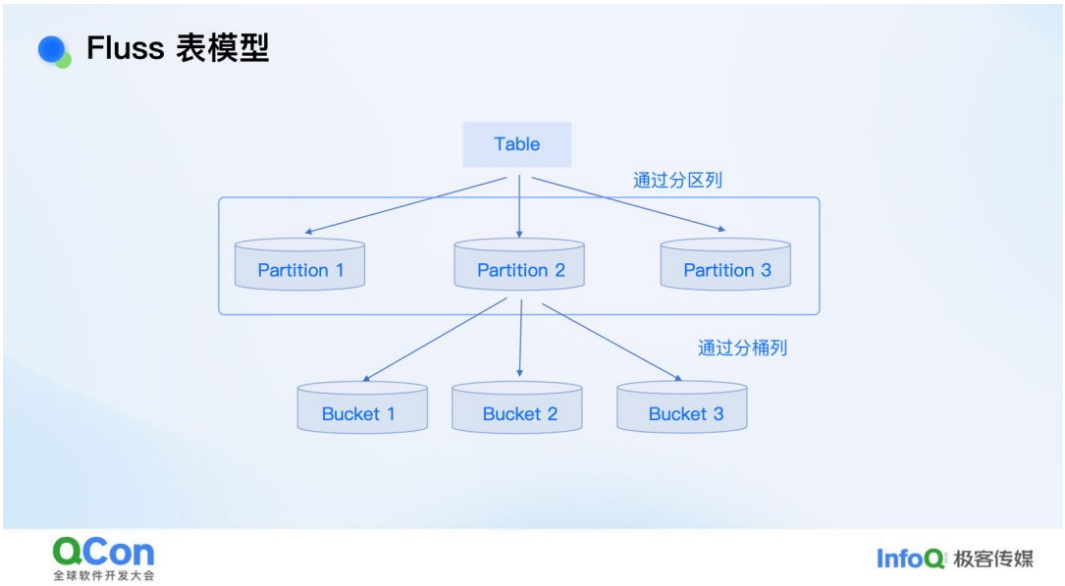
在深入了解之前，先介绍一下Fluss的表模型。Fluss的表具有分区功能，通过分区列将表分成若干分区，这是数据仓库中的一个重要概念。每个分区还会通过分桶列（Bucket）进一步分桶，以实现分布式高性能的写入和读取。这种表模型与常见的数据湖表模型非常相似，几乎没有区别，这也是为什么说Fluss是面向Lakehouse架构设计的存储系统。

Fluss湖流一体技术解析

Lake Tiering Service

接下来是技术解析的核心部分——Lake Tiering Service（数据湖分层服务）。它负责将数据分层写入数据湖。目前，Tiering服务是一个Flink作业，但理论上也可以是Spark作业或其他类型的作业，因为其逻辑不依赖于Flink的特定特性。整个过程包括三个主要步骤：

1. Tiering Coordinator：从Fluss集群中获取需要Tiering的表的信息，并将其分配给Tiering Worker进行湖格式转换。
2. Tiering Worker：实际执行数据转换工作，它将Fluss集群中存储的数据（以Arrow格式存储）转换为湖格式（如Parquet文件）。转换完成后，将这些文件信息发送给Tiering Committer
3. Tiering Commiter：向数据湖提交写入的湖格式，并且将数据湖的快照提交回Fluss集群，使Fluss集群能够管理和利用这些数据。



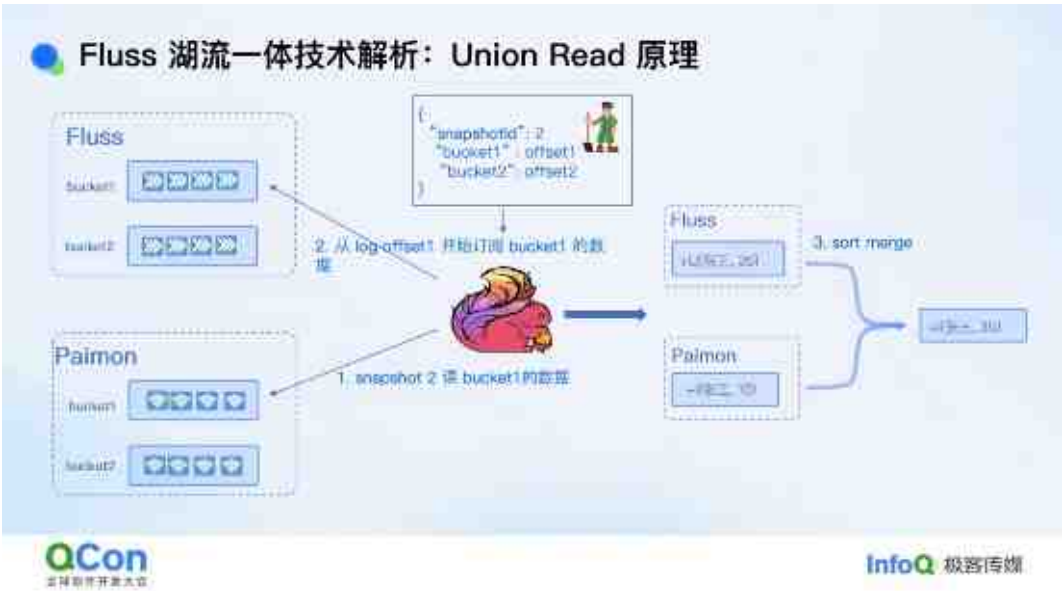
Tiering服务是无状态的，这意味着它可以快速进行横向/纵向扩展。如果一个Tiering服务出现反压（即转换速度跟不上），可以快速启动更多的Tiering服务实例，或者增加Tiering Worker的并发数量，从而解决转换瓶颈问题。这种无状态设计使得扩容非常迅速，无需进行状态迁移，能够有效应对数据Tiering的高压力场景。

Union read

在从Fluss到数据湖的过程中，Tiering Service的工作实际上可以使用任何其他同步工具来完成，比如Flink CDC。通过Flink CDC将数据从Fluss同步到数据湖是可行的，虽然这种方式没有Fluss自带的Tiering Service那么高效，也没有Fluss的无状态特性，但这并不是一个很大的问题。关键在于，数据从Fluss进入Lakehouse之后，这些数据是否能够被Fluss所利用。这也是我接下来要讲的Union Read能力的核心所在。

Union Read的作用是将Fluss中的实时数据与数据湖中的历史数据进行合并。假设一开始我们有一些历史数据，比如“张三15岁”、“李四20岁”，这些数据已经被写入数据湖（例如Paimon）。但随后我们发现最新的数据表明张三不是15岁，而是35岁，这条实时数据进入了Fluss。如果某个查询想要获取最新的数据，查询引擎就可以使用Union Read功能，将Fluss中的实时数据（张三35岁）与数据湖中的历史数据进行合并。最终得到的结果是“张三35岁”、“李四20岁”。这样，Union Read实际上打破了Lakehouse的延迟限制，将数据的新鲜度提升到了秒级。

Union Read是如何实现的呢？其实并没有什么特别复杂的“黑魔法”。核心在于Fluss能够管理数据湖中的数据。具体来说，当Tiering Service将数据从Fluss提交到数据湖时，Fluss会记录一些关键信息。首先，它会记录一个快照，表示这个快照已经包含了当前所有提交到数据湖的数据。其次，它会记录一个bucket和offset的对应关系，表示某个bucket中offset之前的全部数据已经成功写入数据湖。通过这种方式，Fluss知道哪些数据已经进入数据湖，以及这些数据对应的快照版本。



当Flink向Fluss集群查询最新快照时，Fluss会返回最新的快照ID（比如快照2）以及对应的offset信息。Flink可以利用这些信息，直接从数据湖中读取快照2中的数据（比如“张三10岁”），同时从Fluss中订阅 对应offset之后的实时数据（比如张三35岁）。这两部分数据合并后，最终得到的结果是“张三35岁”。如果没有Union Read，传统Lakehouse架构可能只能读取到“张三10岁”的数据，数据的新鲜度会大打折扣。而通过Union Read，Fluss能够实时地将最新的数据与历史数据结合，从而提供更准确、更及时的数据视图。

高效转成湖格式的利器

Union Read的实现依赖于一个重要的机制：Bucket对齐，即Fluss的bucket和Paimon的bucket是一一对应的。通过Bucket对齐，对于Fluss的某一个bucket的数据，Union Read只

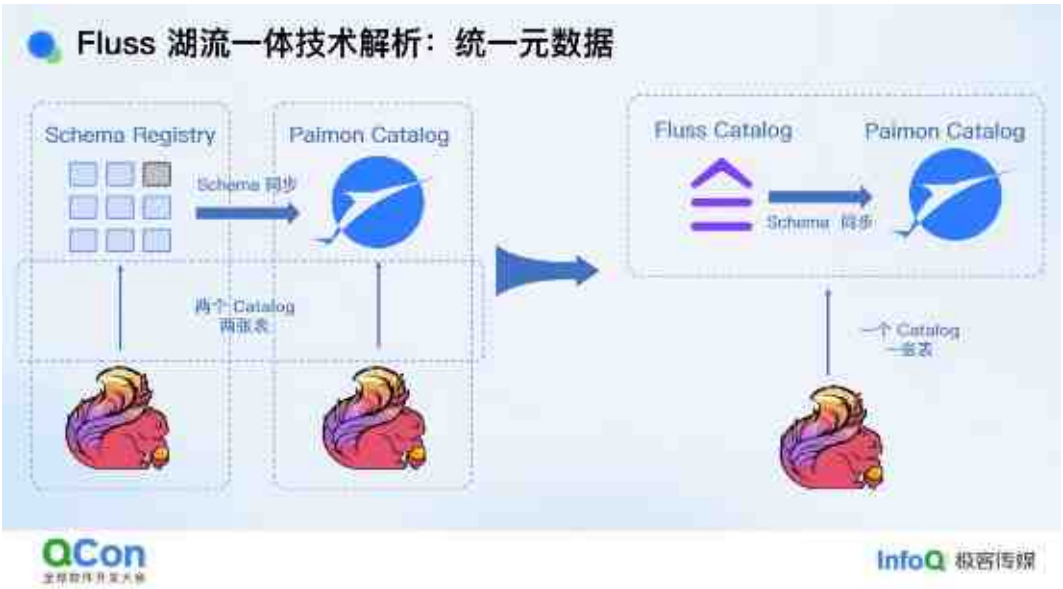
需要将其与Paimon对应的那个bucket进行union即可，而不需要union Paimon所有的bucket。

Bucket对齐的另一个好处是它能够帮助我们高效地将数据转换为湖格式（如Parquet）。虽然其他系统也可以进行这种转换，但Fluss的优势在于数据分布的对齐。具体来说，Fluss表和Paimon表都具有分区和分桶的结构，且它们的Bucket是对齐的。这意味着一条数据在Fluss中位于某个Bucket中时，它在Paimon中也一定位于同一个Bucket中。这种对齐方式避免了数据在转换过程中需要进行Shuffle的开销，从而提高了效率。此外，Fluss的底层存储格式是Arrow，而Arrow社区已经提供了非常成熟的转换方案，通过高效的C++实现，可以将Arrow数据高效地转换为Parquet格式，性能提升显著。这也是Fluss湖流一体架构能够高效转换为湖格式的关键所在。

统一元数据

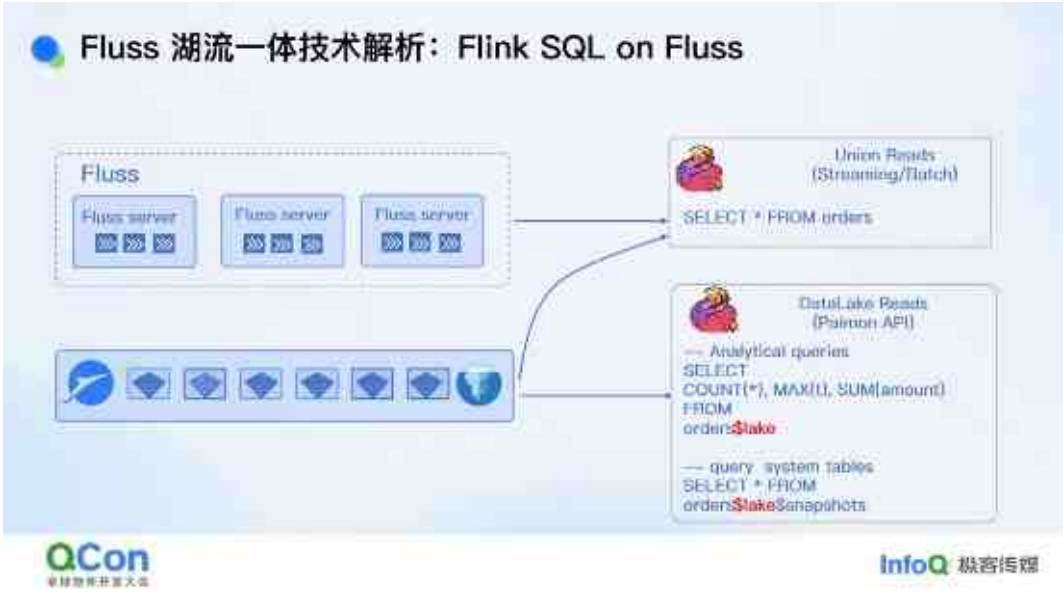
在传统架构中，元数据管理通常较为复杂。例如，Kafka使用Schema Registry，而Paimon使用自己的Catalog。在这种情况下，需要通过Flink作业进行Schema同步，且在读取数据时需要在不同的Catalog之间切换，这使得数据同步和使用变得不够方便。然而，在Fluss湖流一体架构中，虽然仍然存在两个单独的Catalog（Fluss的Catalog和Paimon的Catalog），但Fluss会自动同步Schema，并且对外暴露的是一个Catalog，这样在查询时看到

的是一张统一的表，无需在不同的Catalog之间切换，用户不需要切换Catalog既可以查到Fluss的最新数据，也可以查到Paimon中的历史数据。



Flink SQL on Fluss

这种统一元数据的实现方式使得用户在查询时更加方便。如果用户不需要最新的实时数据，而只需要3分钟前在Paimon中的数据，可以通过在SQL查询中添加$lake修饰符来直接从Paimon中读取数据。这种方式继承了Paimon表作为Flink Source的所有能力，包括高效的读取、列裁剪和过滤条件下推等。用户无需切换表或Catalog，即可实现高效的OLAP查询和实时数据读取。



Fluss湖流一体核心优势

成本降低

在传统架构中，Kafka通常用于存储实时数据，但其本地磁盘存储成本较高，且数据需要通过同步作业转移到数据湖中。而Fluss湖流一体架构将历史数据直接同步到数据湖中，并且这些数据仍然可以被Fluss使用。这意味着Fluss本地不再需要存储大量历史数据（例如3天），而可能只需要存储6个小时的数据，从而大幅减少了磁盘和存储成本，流存储成本可以降低到原来的十分之一。

高效的数据回追与流读

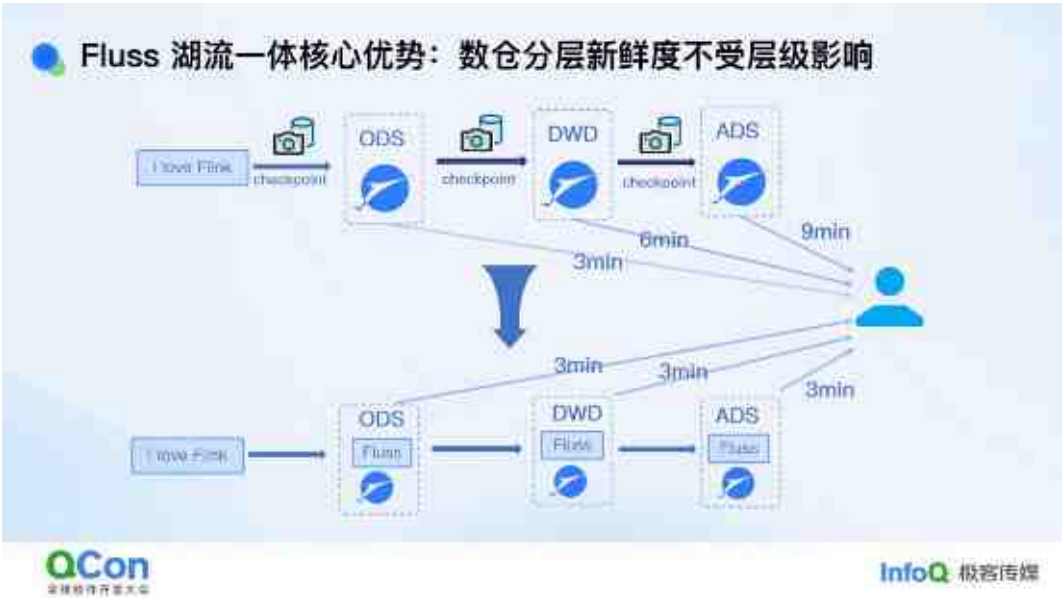
在某些场景下，需要从历史数据回溯到实时数据。Fluss的历史数据存储在Paimon中，而Paimon的优势在于其高效的批量读取能力。它借助湖格式的高效条件过滤、列裁剪和压缩，使得读取速度非常快。在需要回溯时，Fluss会自动从数据湖读取历史数据，然后切换到实时流读取，无需手动干预。这种设计实现了真正的流批融合，数据既不会丢失也不会重复。

数据湖时效性提升至秒级

传统数据湖的时效性通常在分钟级别（例如Iceberg可能是十几分钟，Paimon是3分钟）。然而，在Fluss湖流一体架构下，数据的时效性可以提升到秒级。因为Fluss的端到端延迟是秒级的，数据一旦进入Fluss，用户就可以立即读取。这种架构可以在不破坏原有架构的情况下，通过引入Fluss来实现秒级数据可见性，而无需对现有基建进行大规模改动。

数仓分层新鲜度的提升

在传统数仓架构中，数据分层（如ODS、DWD、ADS）会导致数据新鲜度逐层降低。例如，如果每层的Flink checkpoint时间为3分钟，那么数据从ODS层到ADS层可能需要9分钟（3层 × 3分钟）。然而，在Fluss湖流一体架构下，数据在每一层的延迟都是秒级的。数据进入Fluss后，会立即生成Change Log并流向下游层，同时也会同步到Paimon中。这意味着无论数据经过多少层，其新鲜度都不会受到层级的影响，最终数据在数据湖中的可见性仍然可以保持在3分钟以内。



更高效的CDC生成

Paimon支持两种Change Log生成方式：Lookup Change Log Producer和Full Compaction

Change Log Producer。Lookup Change Log Producer时效性高，但资源消耗大；Full Compaction Change Log Producer资源消耗低，但需要等待多个Checkpoint后数据才可见。而Fluss生成Change Log是秒级的，因为它天然支持生成Change Log。Fluss的Change Log可以直接转化为Paimon的Change Log，兼顾了时效性和性能。

与Confluent Table Flow的对比

Confluent Table Flow是一个将Kafka数据通过Flink作业同步到数据湖的解决方案。虽然它简化了数据转换过程，但数据需要在Kafka和数据湖中保存两份，增加了存储成本。而Fluss湖流一体架构中，数据只需要保存一份，历史数据直接存储在数据湖中，Fluss中只保留最近几小时的数据，从而降低了成本。此外，Fluss的Union Read能力可以将数据湖的时效性提升到秒级，而Confluent Table Flow并没有增强数据湖的时效性。此外，Kafka的Topic没有Schema和分区表的概念，这在使用时会带来诸多不便。而Fluss的表模型是基于数据湖设计的，支持分区表和分桶，与Paimon完全对齐，避免了数据模型不一致的问题。

Fluss湖流一体QuickStart

接下来通过一个简单的Quick Start来了解Fluss湖流一体的使用方法。实际上，Quick

Start在我们的开源社区中也非常容易找到。只要打开社区文档，点击Quick Start，就能找到。Quick Start也是一个基于Docker的环境，可以帮助大家快速体验Fluss湖流一体的功能，非常简便。

Quick Start会涵盖我之前介绍的所有内容。首先是一个Lake Tiering Service。这个服务是我们提供的，你不需要手动去完成Tiering工作。只需要运行一个命令，带上一些Flink参数（因为目前Tiering Service是一个Flink作业），就可以启动这个服务。启动后，你可以在Flink UI上看到这个作业。

一个普通的表如果你想让它成为湖表，需要进行一些设置。具体来说，你需要通过添加一个properties（属性）来指定这张表是湖表，并开启湖表功能。这样，Lake Tiering Service才会感知到这张表，将其转换为湖格式。

完成设置后，你可以直接向Fluss表中插入数据。插入数据后，就可以进行数据读取了。因为Tiering Service在后台会进行数据转换，所以当转换完成后，你就可以直接从数据湖中读取数据，并利用Union Read（联合读取）能力，读取全量数据。

例如，执行一个简单的SQL查询：select count (1) as order_count from dwd_enriched_orders where n_name = 'CHINA'，最终可能会返回1000多条数据，这是最新的全量数据，因为Union Read先从Paimon中读取数据，再从Fluss中读取实时数据。

如果你只想读取数据湖中的数据，或者只关注某个快照之前的数据，而不需要实时数据，你可以直接在查询中指定$lake。这样，查询就会直接从数据湖中读取数据，而不需要切换Catalog。例如，指定$lake后，可能在9秒内就能读取到3条数据。这里需要注意的是，数据湖中的数据并不是最新的，它可能有3分钟的延迟，这是正常的。

你也可以选择使用StarRocks来读取数据。因为数据已经是Paimon的湖格式，你只需要指定Paimon的路径（warehouse path），创建一个StarRocks的Catalog，就可以直接使用StarRocks进行读取和分析。

未来规划

支持更多查询引擎

目前，我们的Unon Read能力已经对接了Flink，因为我们本身是一个Flink团队，所以优先支持Flink是很自然的选择。然而，我们的架构并不局限于Flink。未来，我们计划对接更多查询引擎，比如StarRocks和Spark。这样一来，用户可以直接使用StarRocks和Spark进行Union Read，查询数据湖和Fluss中的数据，从而实现数据的极致新鲜度。

湖生态对接Iceberg和Hudi

目前，我们主要对接了Paimon，不过在社区中已经有针对Iceberg和Hudi的初步开发和贡献。我们期待这些功能能够尽快完成并发布，进一步丰富我们的湖生态支持。

支持Deletion Vector

我们计划在Union Read中支持Deletion Vector。Deletion Vector主要用于优化主键表的读取性能。目前，Paimon社区已经支持了Deletion Vector，我们也希望尽快对接这一功能，从而大幅提升读取主键表的性能，减少数据union的开销，提高整体效率。

最后，我们非常欢迎大家加入Fluss社区，共同建设这个项目。以下是项目的地址和文档链接，大家可以参考文档来了解更多信息。

- **Fluss项目地址：**https://github.com/alibaba/fluss
- **Fluss文档地址：**https://alibaba.github.io/fluss-docs

嘉宾介绍

- **罗宇侠**，阿里云智能高级开发工程师，Apache Flink Committer，早期致力于Flink与Hive生态以及数据湖生态的集成。目前专注于流存储Fluss项目，负责湖流一体的研发工作。

高性能全闪并行文件系统的设计和实践

演讲嘉宾 张文涛　编辑 Kitty

在深度学习领域中，数据是基石，算力是引擎。训练一个模型，需要大量的数据和算力，并且需要反复迭代和验证才能得到想要的模型。为了提升训练效率，缩短训练时间，所有组件之间都需要快速响应，这其中就包括了计算和存储之间的交互。对于一个AI系统而言，模型的能力随着模型尺寸和训练数据的增加而显著提升，但随着数据集和模型规模不断增加，训练任务载入训练数据所消耗的时间越来越长，进而影响了训练效率，缓慢的IO严重拖累GPU的强大算力。

内容亮点

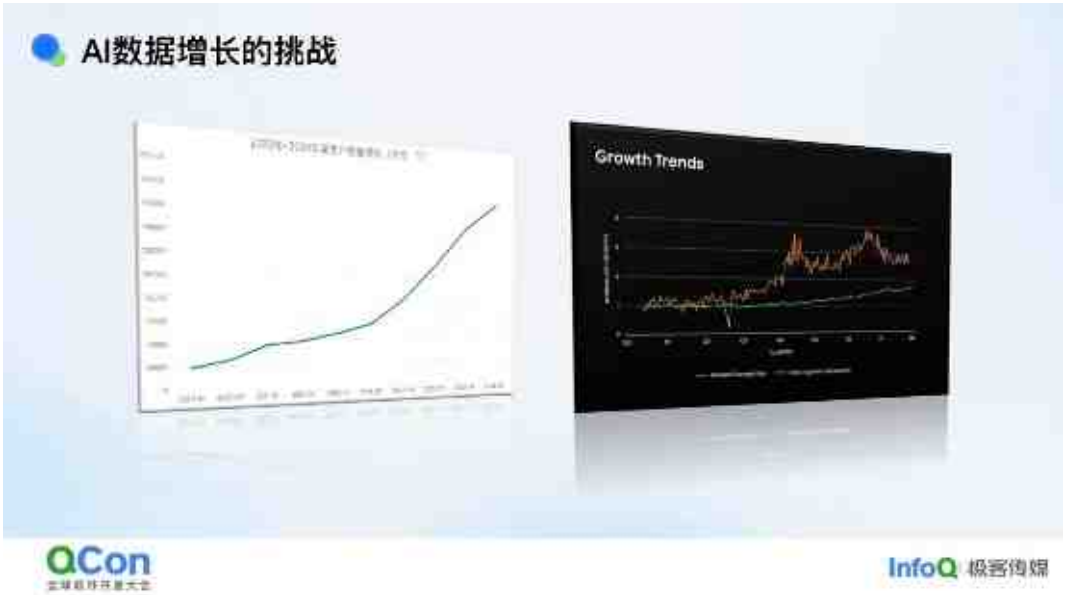
- YRCloudFile高性能文件系统的核心技术
- 在AI训练场景中遇到的疑难问题和解决方案

以下是演讲实录（经InfoQ进行不改变原意的编辑整理）。

大模型时代的存储挑战

今天我给大家带来的分享主题是高性能全闪存储。刚才几位老师在讲解对象存储时，大多是站在降低成本的视角来设计系统，然后再考虑性能优化。而我们则是反其道而行之，先从性能角度出发，再设法降低成本。这两种视角存在明显差异，下面我将分享我们在这一领域所开展的工作以及设计理念。

大模型时代的存储面临着诸多挑战。从下图可以看出，黑色图是去年Meta公布的数据，其中绿色线代表过去两年其容量的增长情况，橙色线则表示性能的增长情况。在过去两年里，尽管Meta的数据量基数已经很大，但其容量仍翻了一番。而橙色线所显示的吞吐情况，相比之前已接近原来的四倍。白色图展示的是我们一个大客户的存储情况，仅代表他在我们YRCloudFile中的存储数据量。我们统计了他这四年里的数据增长情况，发现在2020年到2022年之间，每年数据量增长接近20T。但从2022年年底开始，到2024年年底，其数据量增长愈发迅猛，基本以每年60T的速度增长。这一增长情况与大模型时代到来的时间点相契合，即2022年下半年，尤其是年底ChatGPT爆发时，国内的大模型厂商纷纷跟进，数据量也随之飞速增长，这便是数据增长方面所面临的挑战。



在AI的全流程中，与存储相关的环节主要有四个。首先是数据采集环节，主要是将

各种原始数据收集过来，方式多种多样，比如编写脚本爬取数据、从公共数据网站下载数据、购买数据，或者收集本行业及企业内部的数据等。在这个过程中，需要运用各种协议来同时访问数据。第二个环节是数据处理，主要是针对数据进行清洗、格式转换以及集成，为后续的数据训练做好前期准备。这个流程较长，涉及多个类型的存储挑战，包括多种协议访问、数据快速检索以及I/O大小和读写方式的混合等。虽然该环节对存储的挑战是全方位的，但由于它不影响GPU的利用率，所以往往容易被大家忽视。第三个环节是数据训练，这在AI存储中是大家较为关注的场景。AI训练场景对存储的挑战主要体现在高并发场景的性能上，它的IO类型其实很简单，主要有几种情况：第一种是启动训练时模型的加载，属于大量并发的顺序读；第二种是读取数据集，又分为两类，若数据集较小，会将数据集预热到内存中，同样是顺序大I/O读，若数据集过大，内存无法缓存，则更多采用直接访问存储的方式，此时为大量随机小I/O读；此外，还有Checkpoint本身，属于大并发的顺序大I/O写。对于多模态情况，会增加一些特殊的小文件，如图文对、视频文本对、语音文本对等，从而产生海量小文件问题；第四个环节是推理，其对存储的需求也很简单，类型较少，主要有两种。第一种是模型分发，第二种是最近很火的KVCache，二者都属于吞吐型，而KVCache还增加了延迟敏感，因为其以存代算，访问延迟不能过高；最后是数据归档，需要进行数据的全生命周期管理，以降低整个存储成本。

我们将AI存储面临的主要挑战归纳为四种：第一种是高性能，各种场景都对存储性能有较高要求，包括高IOPS、高读带宽、高写带宽等；第二种是海量小文件问题，这同样是性能问题，不过由数据I/O性能转变为元数据性能。由于文件系统中元数据操作相对复杂，所以这一挑战较为棘手；第三种是横向扩展，大模型时代存储容量增长迅速，同时对性能的诉求也日益强烈，这就要求存储能够支撑大集群，实现容量和性能的线性扩展，以跟上计算能力的增长；第四种则是容量与成本问题，即在实现高性能的前提下，如何降低成本。如果成本降不下来，使用成本就会过高，这无疑会限制存储技术的应用和发展。

YRCloudFile的设计方案

YRCloudFile在设计方案上进行了精心的取舍。文件系统本身在结构上大同小异，通常包含几个关键模块。首先，有一个提供POSIX私有客户端的模块，这是实现文件语义

的入口。接着是MGR（集群管理服务）和MDS（元数据服务），MDS负责存储元数据信息。最后是数据管理服务，用于管理数据。这些组件共同构成了文件系统的基本架构。

针对性能问题，我们深知I/O路径越简单，效率越高。因此，我们采用了一种非常简单的数据路由算法。在文件创建时，系统会划分一组OSD（数据存储服务），确定文件将被打散并存储在哪些磁盘上。这一过程是静态的，在文件创建时就已固化。这种设计带来了两个显著优势。首先，在访问文件数据时，无需频繁访问元数据服务或数据服务，我们可以通过计算快速确定文件位于哪个磁盘的哪个位置。其次，由于文件被打散到多个磁盘上，能够充分利用这些磁盘的能力。在AI场景中，常常涉及大量计算节点同时访问某个大文件，因此大文件能够提供的带宽至关重要，这也是并行文件系统和分布式文件系统存在一些细微区别的原因之一。

除了简化I/O路径外，我们还做了其他工作，以将性能提升到更高水平。我们挑选了几个具有借鉴意义的优化措施。首先是Multi-Channel技术。对于全闪存储而言，单盘带宽本身很高，但如何充分发挥一个拥有十几盘位或24盘位的全闪存储设备的全部带宽能力呢？网卡就成为了一个很大的瓶颈。我们需要进行网卡聚合，但InfiniBand网络和RoCE网络与以太网有所不同。以太网可以进行bond操作，但InfiniBand无法进行bond，需要存储系统做额外工作才能实现多网卡带宽聚合，我们将其称为Multi-Channel。通过这种方式，可以将单个节点的吞吐量翻倍甚至翻四倍，极大地提升单节点的吞吐能力。其次

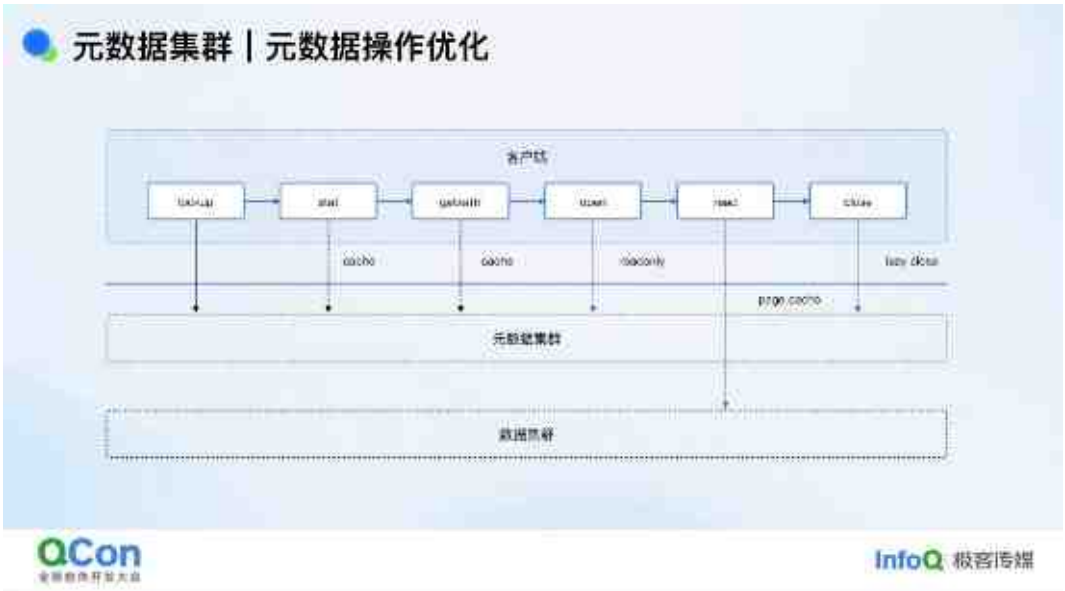
是NUMA亲和性，在高性能场景中，这一点至关重要。其核心问题是避免跨NUMA的内存访问。以AMD平台为例，如果发生跨NUMA访问，带宽将无法超过15GB。只有避免跨NUMA访问，才能充分发挥整个节点的带宽能力。第三是RDMA的单边编程模式。在RDMA中，有两种编程模式，一种是send-receive方式，另一种是read-write单边方式。单边方式的核心优势在于减少内存拷贝。采用这种方式可以减少一次内存拷贝，从而带来更稳定的读写延迟和更低的CPU负载。

去年，我们发布了F9000X全闪一体机产品。该产品配备了第五代Intel CPU系列，4张400Gb的InfiniBand网卡，以及16块全闪盘。需要注意的是，由于CPU插槽数量的限制，我们只能插入16块磁盘。这种配置下，一个三节点集群能够达到480GB/s的带宽和750万的IOPS，相比上一代F8000X产品，每GBps带宽成本下降60%。

在海量小文件问题上，各种解决方案都有其对应的场景和不适用的场景。存储领域没有一种架构能够解决所有问题，只有场景适用与不适用之分。在文件系统中，常见的解决方案包括静态子树目录哈希或动态子树等架构。我们采用的是基于Dentry Hash的方式，它遵循三个原则。第一，在集群格式化时，根目录会被固定下来。第二，在创建子目录时，会重新进行哈希选择MDS，这样随着集群中目录数量的增加，能够保证目录和文件均匀地分布在各个MDS中。第三，文件和根目录位于同一节点，这保证了一定的本地性。在文件系统中，许多元数据操作，如find或者ls查询等操作，都涉及到readdir。如

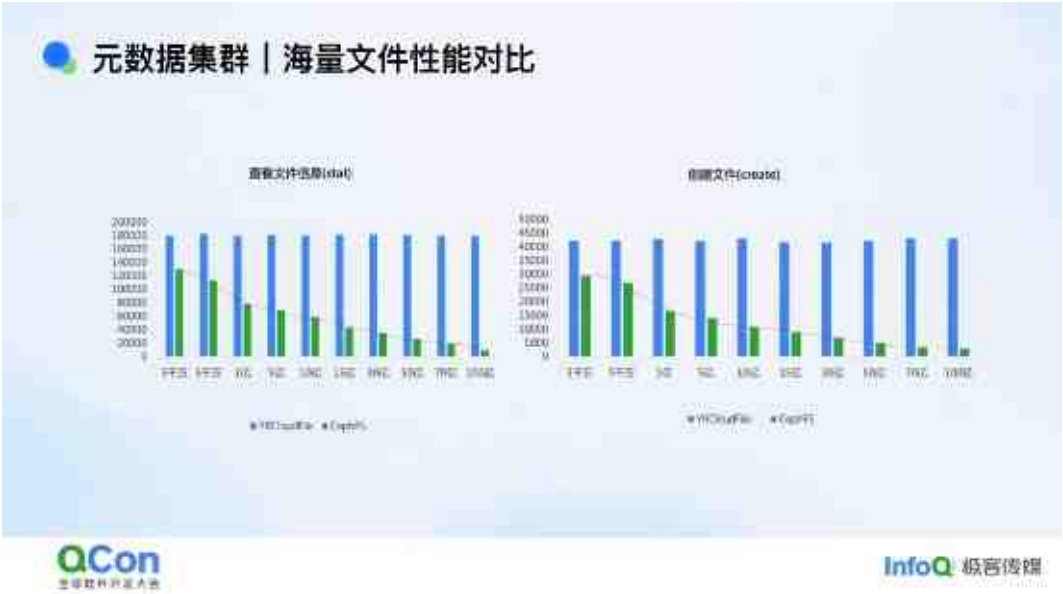
果缺乏本地性，很多优化工作将难以开展。有了本地性后，我们可以进行一些预取等优化操作。

除了架构层面，文件系统还有很多细节需要注意，其中包括缓存等。在访问小文件时，主要涉及的元数据操作包括lookup、getattr获取元数据、getxattr获取扩展属性、open打开文件、读取和close关闭文件。对于小文件而言，真正读取数据的RPC只有一次，其余的都是元数据操作。因此，在小文件场景中，元数据的重要性不言而喻。针对AI场景，我们进行了许多优化。首先，我们有元数据缓存，它可以省去getattr或getxattr等RPC操作。其次，在训练过程中，读取文件时通常是只读的，我们可以将POSIX语义弱化。因为在文件系统中，open操作本身是一个很重的写操作，我们可以将其变为一个轻量级的读操作，从而实现10倍以上的性能提升。还有close操作，它也是一个很重的写操作，因为它需要更新元数据信息，如文件大小等。我们可以将close操作变为异步的，以进一步优化性能。



在项目早期，我们进行了一系列测试，主要针对不同数量的元数据服务（MDS）节点，如1个、2个、3个和4个节点时的性能表现。测试结果显示，性能增长基本呈线性趋势，这为我们后续的研发工作奠定了良好的基础。在实际应用中，当客户进行概念验证（POC）测试时，他们也会关注元数据扩容后的性能表现，例如在元数据操作的OPS（每秒操作次数）方面，是否会随着扩容而实现成倍增长等。对此，我们进行了相应的

测试，并与开源的CephFS进行了性能对比。在对比测试中，我们重点关注了两个关键指标：creation（元数据写操作）和stat（元数据读操作）。测试从一个空集群开始，逐步增加数据量，直至达到1亿、10亿甚至100亿的规模。结果显示，YRCloudFile在元数据OPS方面表现稳定，波动较小。而CephFS在数据量达到1亿后，性能衰减较为严重，当数据量增至几十亿时，其性能几乎无法满足实际使用需求。这一对比结果充分证明了YRCloudFile在处理海量元数据时的优越性能。



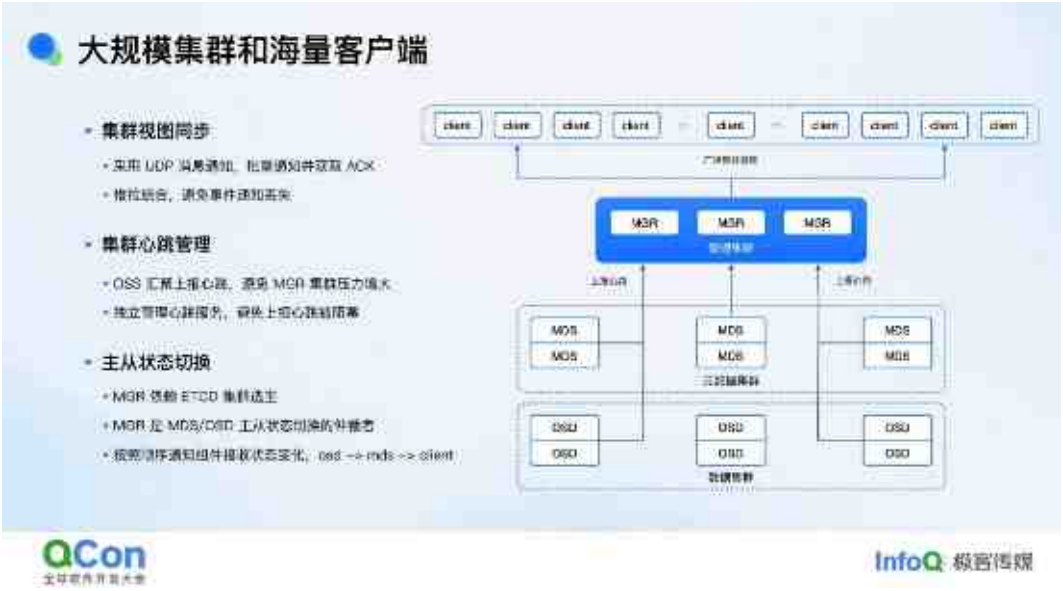
针对AI场景中的存储需求，我们进行了深入研究和优化。通常情况下，AI存储集群的规模可能在几百台服务器左右，而客户端数量可能在几千台左右，这已经是一个相当大规模的集群了。在这样的集群环境中，我们开展了一系列工作，这些工作具有一定的借鉴意义。

首先，我们注意到集群规模较大的一个重要因素是心跳管理。为了确保集群中各节点的状态能够及时准确地被监控和管理，我们设计了一种汇聚式的心跳上报机制，有效减轻了管理节点（MGR）的压力。同时，我们还将心跳服务独立管理，避免了在心跳上报过程中可能出现的阻塞问题，从而提高了整个集群的稳定性和可靠性。

其次，我们采用了UDP协议进行集群事件同步，即事件通知。UDP协议本身是无状态的，可以批量发送大量数据，并且能够实现高效的同步操作。然而，UDP协议的一个

缺点是数据包容易丢失。为了解决这一问题，我们采用了推拉结合的方式。一方面，我们会主动将事件推送给相关的客户端或其他组件，使它们能够及时感知到事件的变化；另一方面，如果数据包丢失，客户端或其他组件会主动拉取事件信息，从而确保了事件通知的可靠性和及时性。

此外，管理节点（MGR）在集群中扮演着仲裁者的角色，负责管理和协调元数据服务（MDS）和数据存储服务（OSD）。这种设计避免了引入外部仲裁者所带来的复杂工程实践问题，使得MGR能够站在一个客观的角度，准确地判断各个服务节点的主从关系，从而提高了整个集群的管理效率和稳定性。

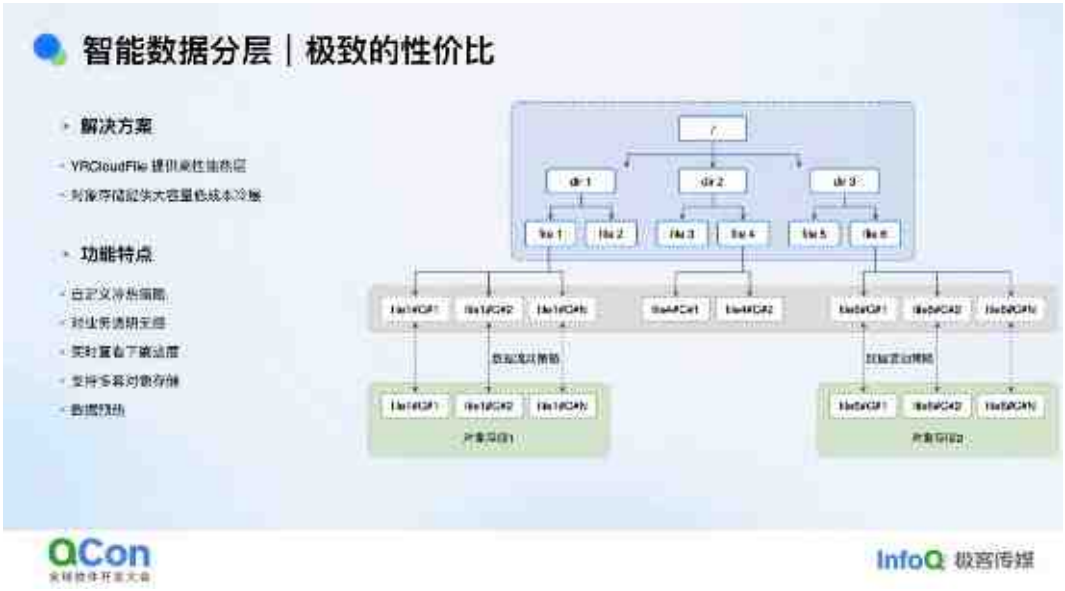


我们的产品规格能够支持200多台全闪存储节点的集群规模。这样的集群能够提供TB级别以上的带宽，接近10TBps，能够满足大规模AI计算的需求。在客户端支持方面，针对RDMA协议，我们可以支持2000个客户端；而对于基于TCP的以太网的客户端，我们能够支持的规模可达10万个。这些性能指标充分展示了YRCloudFile在大规模集群环境下的强大性能和高扩展性。

我们的设计理念是先确保性能达到要求，然后再通过各种手段降低成本。具体来说，我们采用了智能数据分层的策略。YRCloudFile本身作为一个高性能的热层存储空间，主要用于存储频繁访问的热点数据。而对象存储则作为大容量、低成本的冷存储空间，用

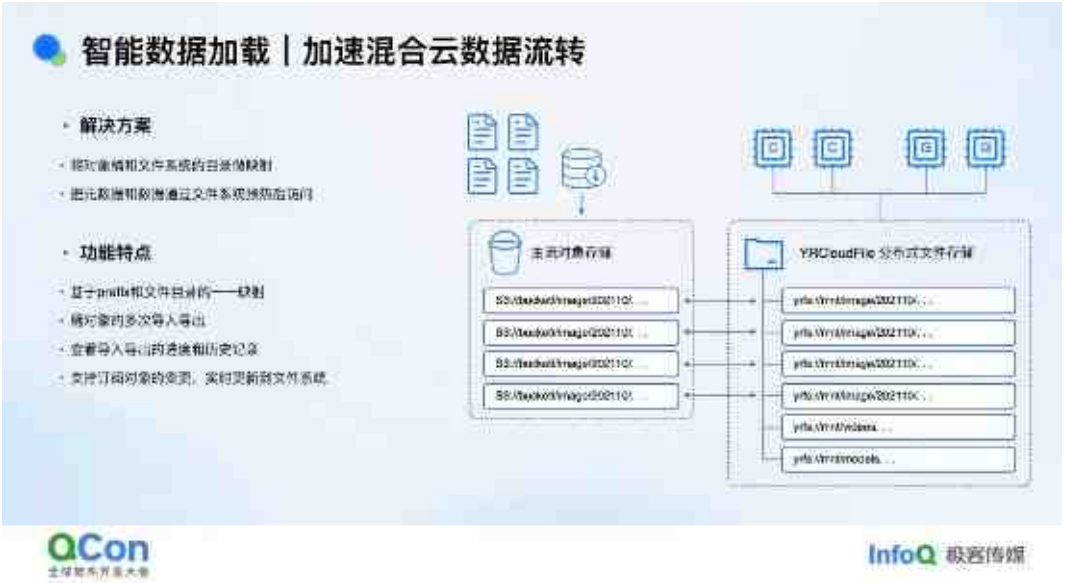
于存储不那么频繁访问的冷数据。对于业务应用来说，它们并不需要感知后端存储的具体实现，它们看到的仍然是一个统一的文件系统视图。

在智能数据分层功能中，管理员可以根据实际需求自定义冷热数据的划分策略。策略的定义主要基于两个维度：时间和大小。从时间维度来看，管理员可以设定一个时间阈值，例如一天、一周或一个月，如果数据在设定的时间内没有被访问，那么这些数据就会被自动下沉到冷存储中。从大小维度来看，对于小文件，由于它们本身占用的空间较小，可以一直保留在热层存储中，以避免小文件对对象存储的访问压力。此外，智能数据分层还具备业务透明无感的特点，管理员可以通过命令行或者界面实时查看数据下沉的进度，并且后端存储支持多种对象存储类型。在AI场景中，数据预热功能至关重要。这是因为GPU在进行计算时，无法等待从冷存储中加载数据，因此我们需要提前将数据预热到热层存储中，以确保GPU能够快速访问所需数据，从而提高整个AI计算的效率。



除了智能数据分层，我们还实现了数据智能加载功能。这一功能同样有助于降低成本。在实际应用中，用户可以将数据集或原始数据存储在对象存储中，无论是公有云的对象存储服务，还是私有云的对象存储系统都可以。当需要进行训练时，再将这些数据加载到全闪文件系统中。传统的做法可能是通过编写脚本来实现数据的上传和下载，但这种方式效率较低。而我们的数据智能加载功能则提供了一种更加高效的方法。它可以将对象存储桶与文件系统的目录进行映射，并允许用户自定义加载策略。例如，用户可以先将元数据预热到文件系统中，使用户能够快速看到对应的数据，然后在后台异步地

将实际数据加载过来。此外，我们还支持对象存储的变更订阅功能。当对象存储中的数据集发生变化时，我们可以及时将这些变更同步到全闪文件系统中，确保数据的一致性和实时性。



下图是YRCloudFile的整体架构。在最上层是协议层，我们提供了多种协议支持，包括POSIX私有客户端、大数据接口、CSI、NFS以及SMB等。中间部分是我们的后端存储服务，每个组件都采用了高可用架构，确保了整个系统的稳定性和可靠性。在最底层，我们提供了数据生命周期管理的解决方案，包括智能数据加载和智能数据分层等功能。这些功能共同构成了YRCloudFile的完整架构，使其能够满足不同用户在不同场景下的多样化存储需求。

高级运维特性

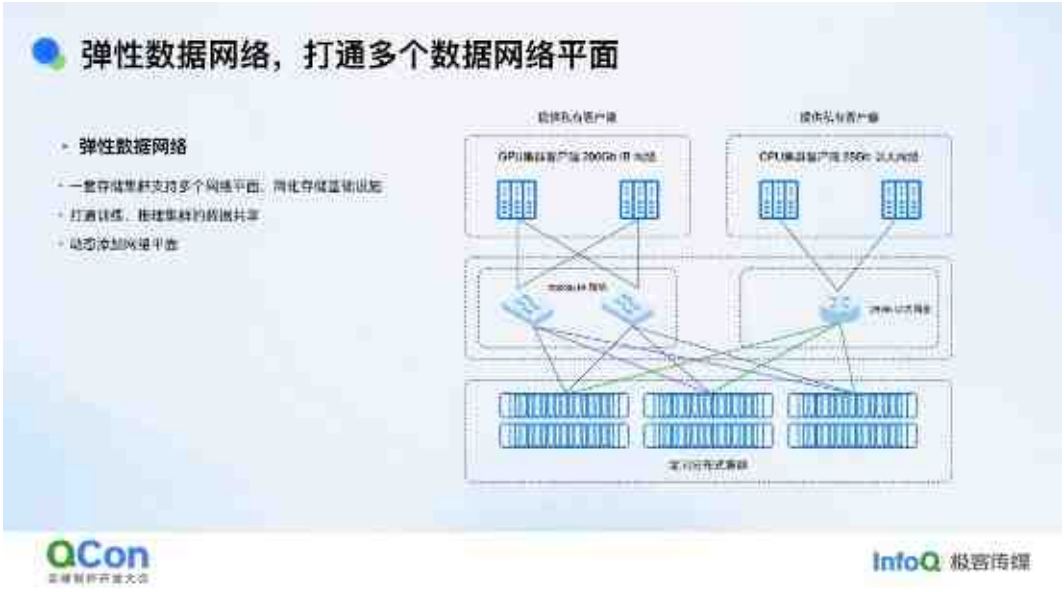
在存储系统的构建和优化过程中，稳定性和可运维性是两个至关重要的考量因素。今天，我将向大家详细介绍我们在这些方面所采取的一些高级运维特性。这些特性可以大致分为几个类别：首先是多租户管理，这在AI训练和推理场景中尤为重要；其次是数据访问安全；最后是我们在构建底层技术设施时，针对多个网络平面以及客户现场棘手问题所提出的系统优化方案。

多租户管理的实现涉及到空间隔离、流量隔离和访问隔离这三个关键方面。在空间隔离方面，我们通过配额管理来让管理员能够为不同的租户分配不同的存储空间。对于流量控制，我们设置了相应的目录级别的QoS，以此来限制某个租户的流量上限，防止其对其他用户造成影响。而在访问隔离上，我们采用了基于IP白名单或token的认证挂载方式，确保租户只能访问自己的存储空间。

在数据访问安全方面，我们采取了多种措施。首先是访问权限控制，我们实现了标准的POSIX ACL，用户可以通过ACL跟LDAP或AD这样的域控服务来实现全局的用户权限统一管理。其次是日志审计，这对于管理员来说极为重要，它能够记录用户的高危操作，如unlink（删除链接）、rmdir（删除目录）、rename（重命名）以及open（打开文件）等操作。日志审计可以记录下是哪个用户在什么时间、使用哪个节点、通过什么工具针

对哪些文件进行了操作，这些信息还可以对接到ELK平台，实现高效的审计检索。最后是回收站功能，它主要用于应对用户或管理员可能发生的误操作。我们设计的回收站允许每个目录自定义回收站，并且每个回收站都可以自定义清理策略，还可以动态开关。虽然回收站本身会对性能产生5%以内的影响，但它为数据安全提供了最后一道防线。

弹性数据网络的本质是帮助用户打通多个网络平面，同时访问一套存储。在AI领域，这种需求非常常见。例如，训练集群和推理集群对网络的要求不同，训练集群通常需要200Gb、400Gb的IB或RoCE高速无损网络，而推理集群则一般使用25Gb或100Gb的以太网网络。这两种网络在物理层面上是隔离的，如果想要实现数据共享，要么通过存储系统本身支持多任务、多网络平面的访问，要么就需要进行额外的数据同步操作，这无疑增加了系统的复杂度。有了弹性数据网络，就可以简化存储基础设施，提高系统的灵活性和效率。



在特定场景下的性能优化方面，由于文件系统是存储领域中最为复杂的类型，其语义丰富，且性能与用户的编码水平直接相关，因此我们挑选了几个在AI领域中典型的优化案例。首先是针对单流业务的优化。单流业务是指只有一个线程在工作的业务，如数据拷贝或解压缩等。为了提升这类业务的性能，我们主要依赖缓存机制，通过预取、预读或缓存写操作来显著提高性能。其次是Cache的HardLimit（硬限制），在AI训练中，尤其是训练小模型时，如果数据集非常大，PageCache可能无法完全缓存数据，这对AI训练

非常不利。因为训练过程中每个数据集都要被读取一遍，虽然是随机读取，但对Cache来说并不友好。此外，当缓存数据需要被置换时，如果数据量很大，会导致延迟抖动非常严重，这对GPU的效率非常不利。为此，我们设置了Cache的HardLimit，例如对于一个1TB的GPU服务器，我们允许其使用的缓存最多为100GB或200GB的数据，这样可以避免触发PageCache本身的阈值，从而减少抖动。最后是客户端限速。这与前面提到的QoS有所不同，主要是为了解决在IB网络场景中，当多个用户共享一个集群时，某些计算节点的网络带宽过高会导致拥塞的问题。这种拥塞会扩散，影响整个集群的网络带宽。我们的解决方案是限制某些客户端的速度，通过牺牲少量客户端的峰值带宽，来实现整个网络的高吞吐量。

AI训练推理解决方案

在AI训练阶段，性能是至关重要的。我们通过多种技术手段来提升性能，包括Multi-Channel技术、支持GPU Direct Storage以降低延迟、内核私有客户端以及支持400Gb的Infiniband或RoCE无损网络。此外，我们还提供了分布式元数据集群来进一步增强性能表现。在数据生命周期管理方面，我们实现了分层存储和数据加载功能，这不仅有助于降低成本，还能打通混合云环境中的数据流转。而在运维方面，我们提供了一系列功能，如QoS、Quota、子目录挂载、ACL、审计、回收站以及弹性数据网络等，以确保整个大模型训练过程的稳定性和高效性。在智算中心的整体存储架构中，对象存储作为数据底座，而YRCloudFile则作为训练存储的加速层，为训练阶段提供高效支持。

在推理阶段，我们的解决方案主要围绕提升推理效率展开。我们针对KVCache进行了优化，通过以存换算的方式提高推理效率。我们提供了一个PB级的KVCache缓存空间，这有助于提高Cache命中率，从而节省算力。由于KVCache本身是一个吞吐且延迟敏感型的应用，我们确保单个计算节点能够提供40GBps的带宽能力，并保证了KVCache访问的低延迟。

在测试数据方面，我们在两种场景下进行了性能对比。第一种场景是在长上下文情况下，使用KVCache后，TTFT延迟显著下降，性能提升了约13倍。第二种场景是在高并发情况下，针对不同上下文长度的对比测试。我们将使用vLLM原生方案与加入YRCloudFile后的数据进行对比，结果显示，当上下文长度越长时，使用KVCache的效果越好。

我们在推理场景中还提供DataInsight的解决方案。目前，DataInsight主要应用于知识库平台，当前许多知识库平台在数据层面存在最后一公里的问题。DataInsight能够帮助企业用户从海量历史数据中快速检索出有价值的数据。它支持多种数据源，包括S3、NAS或HDFS等，并能够实现百亿级数据的秒级检索返回。DataInsight还支持多维度组合查询，使管理员能够精准检索所需数据，并通过DataFlow按需将数据流转到知识库平台中。同时，我们还实现了对第三方存储增量数据的感知功能，这对于企业来说非常有用，因为它能够确保知识库平台的信息保持更新，而无需侵入业务平台。当业务数据写入原有位置时，我们能够自动感知这些增量数据，并将其同步到向量数据库中，从而使知识库平台的用户能够及时获取最新信息，如产品发布参数和最新行业法规等。

总结和未来规划

总结一下，我们在性能方面主要关注元数据性能和数据性能两大块。在元数据性能上，我们采用了分布式元数据架构，并针对元数据操作进行了优化。在数据性能层面，我们引入了GPU Direct Storage、NVMe SSD、RDMA、Multi-Channel以及NUMA亲和性等多项优化技术。

在运维层面，我们涵盖了多租户管理、数据安全访问、弹性数据网络以及监控告警等功能，以确保系统的稳定性和可维护性。在成本控制方面，我们通过智能数据分层和数据加载等功能来降低成本。

关于未来规划，我们有以下几个方面的考虑：

1. 在推理侧，我们会继续增强相关功能。目前我们已经实现了KVCache解决方案，未来将从“有到优”进行进一步优化，提升性能和效率。
2. 在降低成本方面，我们会采取两种策略。一方面，我们引入EC（Erasure Coding，纠删码）技术，并将其作为产品路线图中的重点。另一方面，在存储介质层面，我们将逐步采用QLC SSD，因其单盘容量较大，目前已有32TB的产品，很快将推出64TB的QLC SSD，这将有助于提高存储密度并降低成本。
3. 在客户端方面，我们会将工作负载卸载到DPU中。对于GPU服务器而言，其CPU和内存资源非常宝贵，因此我们将尽可能减少对这些资源的占用，把相关工作负载转移到DPU上，以提高整体效率。
4. 在运维方面，我们会继续增强系统的可运维性。对于存储系统来说，性能、稳定性和可运维性都是非常关键的指标。通过提升可运维性，能够帮助管理员更高效地管理和使用存储系统。

嘉宾介绍

- **张文涛**，毕业于华中科技大学计算机专业硕士，专注于分布式存储领域，拥有超过15年的大规模公有云存储架构开发和AI存储架构设计，参与主导了YRCloudFile高性能分布式文件存储系统从0到1的设计研发及产品落地工作，并在AI场景应用落地方面具备一定的实战经验。在AI及高算力场景项目交付上，有着丰富的整体架构设计和性能优化经验。中国智能计算产业联盟专委会技术专家组，上海TGO鲲鹏会成员。

AI时代，编程语言选型更难也更重要：Go、Rust、Python、TypeScript谁该上场？

作者 傅宇琪 Tina

在AI写码逐渐成为“新常态”的当下，编程语言的选择反而更重要。Flask作者、创业者Armin Ronacher指出：语言背后是复杂的权衡，在AI时代必须被重新审视；更关键的是，语言会直接影响Agent生成代码的质量。

Armin认为，Go在AI场景下更“对味”，它抽象层薄、结构规整，便于模型读懂与改写。同一类小程序，他让AI分别用多种语言各写十次再比通过率，Go明显优于Python，也好于Rust。与此同时，他也强调一个现实：无论你创办什么公司，最终都绕不开Python。也许不会用它写核心服务，但只要涉及机器学习或数据处理，Python一定会出现。同理，JavaScript也无法回避；既有JavaScript，通常就会有TypeScript。

更长远地看，“为人类与Agent共编而设计的下一代语言”正成为行业趋势。Armin认为，现在正是创造“更完美语言”的窗口期：我们短期内不会摆脱AI生成代码的范式，而现有语言也未必是人机协作的最优解。那么，如今的创业公司在Python/Go/Rust/TypeScript之间如何选择？我们翻译了本期播客，汇总Armin的一线观点与实战经验，供大家参考。

以下是一些亮点：

- 一家完全依赖机器的公司，也会失去团队的活力。Claude不是人，它不会有情绪或激情。一个优秀的团队，靠的是人之间的能量与协作，这一点永远不会被AI替代。
- 我非常尊重编程语言，因为它们背后蕴含着复杂的权衡。而在AI时代，我们可能需要重新审视这些权衡方式。
- 编程初学者常常天真地以为“语言就该这样设计才对”，但随着经验积累，你才会理解各种权衡背后的复杂性。
- 统一代码库没那么神圣：OpenAPI等高质量代码生成让前后端“同语种”的价值下降；清晰的边界（尤其RSC场景）反而是优点。
- 当前确实是一个适合创造“更完美语言”的时刻，因为我们很可能不会马上摆脱AI生成代码的模式，而现有语言也未必是最佳选择。
- 反对“旗帜化的996”，我在行业中见过太多惨痛的例子，包括一些社区成员因为长期过劳而出现精神问题，比如精神错乱或精神分裂。这些并非个案，而是长期高压与不健康生活方式的必然结果。工作固然重要，但没有什么值得你为此失去完整的人生。

Python、Go、Rust、TypeScript如何各擅其长

> **Gergely：我们先聊聊编程语言。你曾在Python领域深耕了很多很多年——超过十年。你还记得当年Python 2向3迁移的那场“风波吗”？当时究竟发生了什么？**

Armin：很巧的是，今年出了部《Python纪录片》，我受邀参与拍摄，其中有一段专门讲到Python 2到3的迁移，这让我重新回顾了整个过程。

回过头看，我认为这样的迁移如今已不太可能再发生。当年虽然Python没被这次迁

移“整死”，但确实只差一点。若不是社区投入巨大努力去推动迁移，这次变革可能会给语言带来灾难性的后果。

要理解这次迁移，就要知道Python的历史。最初它的字符串类型和C类似，只支持基础字符串。后来才引入了Unicode字符串，用u前缀表示。而Python 3的主要目标，就是统一并严格化字符串处理，将一切迁移到Unicode。但问题在于，现实世界中的编码远比设计者想象的复杂。Python 3的字符串改动在理论上是合理的，实践中却异常棘手。

迁移的复杂性不仅体现在语言内部，也波及了操作系统和生态。比如当时很多Linux系统仍未全面采用UTF-8编码，这导致文件系统中出现各种奇怪的情况。最终，Python和操作系统生态的演进相互促进，Unicode的支持才逐渐完善。

但在那之前，仍有数以百万计的Python 2代码无法直接运行在Python 3上。最初的设想是“一次性迁移”，可现实中根本做不到。我们不得不同时维护支持Python 2和3的库，持续多年。我记得当时在佛罗伦萨的Python语言峰会上，我提议重新引入字符串u前缀，这样开发者就能编写兼容两个版本的代码。但当时遭到强烈反对，许多人认为应当直接迁移到Python 3，不必再兼容旧版本。然而事实证明，这种理想主义行不通。最终，Python 3的迁移能够成功，靠的是社区多年持续的努力和务实的态度——接受两代共存，逐步过渡，整个过程花了十年以上。

这场迁移也为后来者提供了宝贵的教训。像Rust在设计之初就明确引用了Python 3的教训，提出“版本增量系统”（edition system）的概念，使得新特性可以选择性启用，不必彻底割裂旧版本。这样，一个项目中可以同时存在不同语言版本的代码，这显然是从Python迁移中吸取的经验。

> **Gergely：你提到Python曾经是你最喜欢的语言。后来，你为了性能等原因，在Sentry引入了Rust，现在你也在尝试使用Go。你会如何比较这些语言？当你考虑一种语言时，你会怎么判断它适合做什么，不适合做什么？**

Armin：我写过一些关于我在编程中的“两种人格”的博客。一方面，我喜欢打造精致的开源软件，希望能被成千上万人使用，我会投入大量心血去雕琢代码的每个细节——这有点像制表匠制作瑞士手表，每一处都要精密考究。在这种情境下，我非常在意语言的特性、API的优雅程度以及整体设计。但当你在构建一家公司、打造一款产品时，

这一切都不再重要。那种“写一次、运行无数次”的理想状态，在商业环境中并不存在。

Rust之所以令人惊叹，是因为它非常适合打造精巧的开源项目，但这同样使它成为创业公司中不够理想的语言。它过于严格、过于精确，在开发效率上存在更多摩擦。当然，相比C++，Rust已经是一次巨大的进步，尤其是在处理二进制文件时安全得多。然而，如果要快速迭代产品，Rust的成本太高了。

> **Gergely：能举一些这种“摩擦”的例子吗？为什么你认为Rust不适合初创公司？**

Armin：最明显的一点是，Rust编译极慢，这是巨大的阻力。其次，与Python相比，你需要写更多的代码，还要花大量时间思考类型。我喜欢类型系统，但有些事情实在难以通过类型表达，而这些问题在Python中根本不用考虑。

值得注意的是，动态语言与静态语言之间的界限正在模糊。比如C语言引入了dynamic关键字，让静态语言也能在运行时进行动态类型绑定；TypeScript则将这一思路扩展到JavaScript，使得两种世界在某种程度上融合。而Rust则完全是静态类型语言，如果你的问题本质上需要动态类型支持，就得自己手动实现各种动态包装，这很麻烦。

另一个典型的摩擦点是借用检查器（borrow checker）。虽然它能保证内存安全，但也让某些设计无法实现。比如在C++中，你可以创建自引用结构体，而在Rust中编译器会直接拒绝。你明知道逻辑没错，却被迫重构，最后可能把自己逼进死角。Rust对某些问题的约束太强，它擅长的领域非常“Rust形”，但很多问题并非如此。

正因如此，我现在在新公司主要使用Go。Go是一种极其实用的语言，如果你追求的是务实、高效，它非常合适。它稳定、简单，即便哪天Google不再维护，也会有一群人继续维护它。甚至现代的Java也非常不错，比如虚拟线程（virtual threads）让并发变得更轻松，不必总处理Promise。这些语言可能不“性感”，但足够可靠。

对我来说，选择语言的标准已经更为务实。对公司而言，重点不在代码本身，而在你构建的产品。

> **Gergely：你现在正在创办一家初创公司？团队规模有多大？**

Armin：现在团队除了我和一位联合创始人两人，还有一群Claude和Codex——一支

“AI实习生大军”在写代码，这也彻底改变了我的工作方式。

> **Gergely：一门好的编程语言要具备什么特质？为什么你认为灵活但速度较慢的Python是好的？为什么Go似乎成了你心目中Python和Rust的折中方案？**

Armin：现在，无论你创办什么公司，最终都会用到Python，这是无法避免的。你未必会用它来写核心服务，但只要涉及机器学习或数据处理，Python就一定会出现。同样，你也不可能绕过JavaScript。有了它，就会有TypeScript。问题在于，你的整个代码库中，这些语言所占的比例是多少。

我个人并不想用Python或JavaScript来写后端服务，这并非出于性能考虑，而是因为生态定位不同。比如我现在的公司大量处理邮件解析，这类任务Python处理得不错，但若要在大规模场景下运行，我认为并不理想。结合我在Sentry使用Rust的经验，我可以想象在规模扩大后Rust会重新派上用场，但在当前阶段，它并不是最佳选择。

> **Gergely：当公司需要处理海量数据、运行众多进程时，我们是否就需要关注编程语言的性能问题？**

Armin：我认为“规模化”并不仅仅取决于数据量，而是包括团队规模、问题复杂度以及系统架构的复杂性。在这些条件下，企业往往需要在不同语言间做取舍。引入新语言的原因可能是性能考虑，也可能是为了融入某个生态系统。以Sentry为例，我们之所以引入Rust，不只是因为我个人喜欢它，而是因为当时遇到了性能瓶颈。

我们本可以用Go来写新的服务，但那意味着要维护一个独立服务，这不够高效。Rust之所以成为选择，是因为它能嵌入Python，从而在不分拆系统的情况下提升性能。后来我们做本地符号处理时，备选方案是C++，但在早期尝试后发现，C++程序崩溃频繁，维护困难；而当时Go的生态还不完善，如果从零构建一切成本太高。Rust的生态正好在发展中，尤其在debug方面有Mozilla等公司也在推动，这让它成为更务实的选择。

当然，我们后来也在一些并非最合适的场景使用了Rust，例如数据接入系统。如果现在让我重来，可能不会再那样做。但当时团队规模有限，只能复用已有资源，这是一种现实权衡。我相信，未来我的公司也可能再次因为环境或生态的变化，引入原本计划不用的语言。这并不一定是出于性能需求，可能只是出于生态稳定性或兼容性的考虑。

> Gergely：你如何看待当下Python、Rust和Go的生态？

Armin：Python的生态极其成熟，尤其在基础设施和自动化领域。我目前主要用Python来配置和管理云资源。它在机器学习领域仍无可替代，我的数据管道也是用Python写的，它依旧非常适合构建Web服务。至于是否用它写核心业务逻辑，要看具体情况。比如一个以AI推理为主的公司，大部分时间其实在等待网络返回结果，这种场景下用Python反而非常合适，开发效率高。但如果服务要求高并发、高吞吐，Go可能更合适。Go的语法比Python更简单直接，也更易于维护；而Python随着生态的膨胀，复杂度反而上升了。

Rust则适用于处理二进制数据、构建数据库或负载均衡器等需要高性能和精确内存管理的系统。对于不能容忍垃圾回收延迟、要求性能可预测的场景，Rust是极佳选择。尤其在需要并发处理、内存安全、或被嵌入到高安全环境（如浏览器）的模块中，它的优势明显。Rust也非常适合为Python编写扩展模块，它在性能优化或生态整合方面都优于其他方案。

Go的定位则相对单一，最适合用于构建Web服务或命令行工具。至于WebAssembly，虽然没有如预期般流行，但它的应用需求已根深蒂固，而Rust能很好地满足这一点。相比之下，用Go运行在浏览器中仍存在性能和垃圾回收等限制。

在浏览器端，JavaScript是无法绕开的。而在服务端，我对此其实态度复杂。总体来说，TypeScript已经让这个生态相当成熟了。如果未来有人推出“JavaScript 2”，像Python 3那样清理历史包袱，或许会更好，但我并不建议重演那种迁移灾难。我目前不选择用TypeScript构建后端，主要原因是npm生态。依赖太多，让人几乎无法掌控项目。我倾向于保持依赖最小化，但在JavaScript世界，要构建一个正常项目几乎不可能少于500个依赖包。浏览器端我能接受，因为没有替代方案，但在服务端这让我难以安心。

此外，“统一代码库”的理念在实践中也并不理想。前后端的职责、运行环境、性能约束差异太大，强行统一反而让边界模糊。更好的做法是通过OpenAPI或代码生成工具来桥接两端——这种方式更清晰、也更可靠。

选择语言时要从问题出发，而不是从偏好出发。初创公司应该在早期尽量控制技术栈的数量，三到四种语言已经足够。

> Gergely：过去TypeScript的最大卖点之一就是“统一代码库”，React前端与Node后端共享语言，团队成员能跨端协作。但现在随着AI工具的出现，这种优势似乎正在改变。

Armin：没错。我最近在一次技术交流会上提到，如今“地板在抬高，天花板却没变”。意思是，大家对软件质量的期望越来越高，而支撑这种期望的工具也更强大。以代码生成为例，现在的成熟度远超两三年前。比如Stainless这样的公司，已经能自动生成所有主流AI云服务的SDK。这在过去要花巨大人力维护，如今几乎全自动完成。

当代码生成如此高效时，统一代码库的价值就不再那么关键。事实上，保留清晰的系统边界反而更有利于开发，尤其是在使用React Server Components这类技术时，明确“哪些代码运行在服务器、哪些在客户端”变得尤为重要。

AI时代，语言选择更重要

> Gergely：说到AI辅助编程，你现在的创业公司就大量使用这些“AI实习生”——Claude、Codex等等。能谈谈你们是如何用它们的吗？以及你们目前在做什么？

Armin：我们选择了“电子邮件”作为核心领域，因为它是天然的自然语言载体。如今自然语言处理技术已经在可接受的成本下实现了大规模可用，而邮件中蕴含着丰富而长期未被充分利用的数据。

老实说，我过去对AI编程的态度非常消极，直到今年三月才逐渐改变。到五月，我彻底意识到：软件开发的世界再也回不去了。虽然当前AI协作在团队层面还不算完美，但我相信随着规模增长，它的潜力会越来越明显。

举个例子：在Sentry时，我们曾为改进错误分组算法花了三周时间，仅仅是为了构建一个可视化工具来验证新旧算法效果。而现在，我用Claude半小时就能生成一个更漂亮、功能更完整的版本。过去很多项目夭折，不是因为想法不好，而是因为前期要投入大量精力搭建验证环境，而AI工具几乎消除了这道门槛。

此外，如今我能轻松用Claude构建日志可视化系统、云部署控制台等辅助工具，这些过去我根本不会花时间去做。现在连我并非技术背景的联合创始人也能直接用Claude

和Codex构建原型，验证产品体验。很多功能模块我甚至不亲自写代码，但最终结果完全符合预期。

如今我们约有80%以上的代码是AI生成的，这些代码结构规范、测试完善，承担着标准化API、开放接口、基础逻辑等工作。人类开发者的重点则放在那些真正需要创造性和深思熟虑的部分。这样的分工，使得整个研发过程前所未有地高效。

> **Gergely：过去你对AI编程工具一直持怀疑态度，为什么转变了想法？**

Armin：最大的变化在于：这些工具现在真的开始替我完成那些我讨厌但又不得不做的工作。举个例子，就在昨天，我需要排查一个线上接口为何不能正常工作。问题并不在代码本身，而是出在AWS的权限配置上。结果发现是三个错误叠加造成的：一个是IAM权限问题，一个是系统白名单配置错误，最后还有一个逻辑漏洞。单独看每个错误都不难，但它们叠在一起时很难一眼看出问题所在。

过去这类问题我得花上两个小时才能理清，但这次，我一边处理别的工作，一边让Claude协助排查。虽然我仍需复制一些日志内容，但Claude能结合“世界知识”理解上下文，把不同系统的信息串联起来，帮我定位问题。我能继续推进其他任务的同时，它在后台帮我调试生产环境，这种效率的提升是质的变化。

另一个让我改变看法的关键，是它能自动生成“复现案例”（repro case）。过去我最讨厌写复现案例，但我也知道只要复现清晰，调试过程就会轻松得多。现在我只要告诉Claude：“请帮我生成这个问题的复现案例，大致逻辑是这样。”它就能生成一个完整的示例，有时甚至能写出几千行代码。以前这种事得花上好几天。

从那之后，我开始越来越大胆地把更多任务交给它。现在我会把不想做的部分全权交给AI，而把我想掌控的部分留在自己手里。这就是我转变的核心，意识到我可以这样高效地分工协作。

> **Gergely：那这些具备“代理（agentic）能力”的AI工具，会改变软件工程的哪些方面？又有哪些不会变？**

Armin：系统架构、复杂度管理、可维护性……这些核心问题并不会因为AI而改变。我这些年积累的经验，比如如何让系统在规模扩张后仍然可维护，这些不是机器能完全

替代的。

AI工具确实能辅助工程师，但如果你把所有决策都交给机器，反而会丧失竞争力。因为机器生成的内容往往基于“已有知识”，而创新往往发生在尚未被模型学习到的领域。真正的优势，仍来自人类对新问题的创造性思考。此外，一家完全依赖机器的公司，也会失去团队的活力。Claude不是人，它不会有情绪或激情。一个优秀的团队，靠的是人之间的能量与协作，这一点永远不会被AI替代。

当然，我在使用Claude时确实写更少的代码了，但“编程”的本质体验依然存在，只是我敲键盘的次数变少了。不同的是，AI大大降低了构建自定义工具的成本，也让我能做出更好的决策。部分原因是它检索能力极强，当我遇到新问题时，它不仅能给出答案，还能解释“为什么”。这种能结合教学与解答的方式，对理解复杂概念特别有帮助。比如在一些我不擅长的数学问题上，它能把内容“翻译”成我能理解的形式，帮助我更快掌握。

还有一个显著的变化是，它让更多从未接触编程的人进入了这个领域。前阵子我在火车上遇到一位空管人员，他告诉我自己用ChatGPT学会了写简单的程序，以解决工作中的问题。过去这种人不会编程，而现在只因订阅了带有Codex的ChatGPT，他就能实现自动化脚本。

我们会看到越来越多“新程序员”，他们的入门方式不是通过编程课程，而是因为AI让他们“顺便”写起了代码。过去要成为能独立产出成果的程序员，至少要花上几个月时间学习；而现在，只需输入几个指令，就能看到结果。这种即时的成就感，会让更多人愿意学习编程。AI实际上降低了入门门槛，让编程变得更民主化。

> **Gergely：随着Claude、Codex以及其他越来越强大的Agent的出现，你认为编程语言的重要性会发生怎样的变化？它是否会逐渐变得只对少数人重要？也许只有像你这样长期研究语言优劣的人还会在意，而对大多数人来说，他们只需告诉AI用什么语言解决问题，AI就能在现有代码库上直接完成任务。**

Armin：我依然坚持之前的看法，非常尊重编程语言，因为它们背后蕴含着复杂的权衡。而在AI时代，我们可能需要重新审视这些权衡方式。

我之所以在创业中最终选择Go，是因为我发现Go代码在AI场景下的可扩展性非常出色。它的抽象层很薄，使得AI更容易理解代码。我甚至做过实验：让AI在不同语言中各写十次同类型的程序，然后比较成功率，结果发现Go的表现远优于Python，也明显好于Rust。这说明编程语言依然重要，因为语言会直接影响Agent生成代码的质量。

至于现在的语言中是否已经存在适合“人机协作”环境的理想语言，我并不确定。也许我们已经接近了编程语言的巅峰，但也可能正是此时，会有人提出全新的设计思路：在AI降低某些成本、但提高其他成本的背景下，重新平衡这些取舍。

我认为编程语言仍然会非常重要，尤其是它在运行时环境中的权衡，这一点甚至会变得更关键。唯一不同的是，一切都在加速。技术演进的速度比以往更快，无论是产品层面还是技术层面，都让人感觉如果此刻不去投入，就会错过变化。

我相信现在有不少人正在尝试构建一种“为人类与Agent共同编程而设计的语言”，这不仅仅是一两个人的想法，而是整个行业的趋势。

当前确实是一个适合创造“更完美语言”的时刻，因为我们很可能不会马上摆脱AI生成代码的模式，而现有语言也未必是最佳选择。

不过，计算机完全取代人类编程的可能性并不高。人类仍会在相当长一段时间内保持在循环（Loop）中，参与更多我们意想不到的环节。因此，我们不可能追求“最优输出是直接生成汇编代码”的目标。那样的话，人类将无法审核这些代码。反而，未来的取舍可能会倾向于更高层次的语言，以便让审阅和协作更加容易。

> **Gergely：汇编语言的一个问题在于，每种架构或CPU类型都需要不同的汇编代码。这也是为什么在上世纪九十年代，Java会如此流行，因为它能在不同系统上运行，比如Mac。**

Armin：没错，但如果AI能够自动将代码迁移到不同的操作系统，那也许可以反过来质疑：我们是否还需要为跨平台而设计语言？

> **Gergely：尽管AI理论上可以让我们“少干活”，比如让Agent在后台循环执行任务、你可以去睡觉，但现实中人们反而变得更忙了。尤其是在旧金山，一些AI初创公司甚至公开宣扬“996”，还会在社交媒体上自豪地展示团队在午夜仍在**

> 办公室奋战的场景。你怎么看这种趋势？

Armin：首先，我得特别感谢Peter Steinberger，他很大程度上促使我深入接触“智能编程”。他当时经营一家叫PSPDFKit的公司，后来卖掉后重新开始写代码。他告诉我，他找到了一个“能替自己编程的电脑”，从那以后几乎废寝忘食。我一开始对此持怀疑态度，但后来我明白，Agent编程确实会改变你的思维方式。

起初，我感到一种强烈的焦虑：每当Agent没有在运行、没有在产出，我就觉得自己在浪费时间。那种感觉几乎像上瘾，它带来即时的满足感，就像在拉老虎机一样。你一遍又一遍地运行它，期待新的结果。对我来说，这种“即时奖励机制”一度让我彻夜不眠，不是因为我在高效推进创业项目，而只是被这种过程本身所吸引。

此外，现在很多人都意识到技术正在发生巨变，都想尽可能抓住机会。这也助长了那种高强度的工作文化。但“996”是一种极端的工作制度——每天工作12小时，每周6天，几乎没有私人时间。我虽然也有过每周工作80小时的时期，但我始终不会把这种节奏常态化，因为我知道人不可能持续高效。尤其是我有妻子和三个孩子，家庭是我生命中最重要的部分，所以我的工作必须围绕家庭来安排。你可以高强度工作，但不必以牺牲生活为代价。

还有一点常被忽视：如果你是创始人，或者公司早期的核心工程师，且手里有真正有意义的股权，那你在高强度工作中得到的回报是不同的。但对于后来加入的员工，尤其在公司股权不太可能兑现的情况下，没有理由去透支自己去成全他人的利益。

我反感“996”被当成一种口号去宣传。它的本意是毫无弹性的劳动安排：每天12小时、每周6天，完全牺牲生活。而这种文化在现实中导致了严重的健康问题，甚至心理疾病。公司若真想倡导高能量、高投入的工作环境，至少也该坦诚地面对代价，而不是用带有极端色彩的数字去粉饰。

我在行业中见过太多惨痛的例子，包括一些社区成员因为长期过劳而出现精神问题，比如精神错乱或精神分裂。这些并非个案，而是长期高压与不健康生活方式的必然结果。工作固然重要，但没有什么值得你为此失去完整的人生。

错误处理心得

> Gergely：不同语言在错误类型或频率上是否存在明显差异？从错误处理（Error handling）的角度看，编程语言的选择到底有多重要？

Armin：不同语言的崩溃方式确实不同，这不仅与语言本身有关，也取决于你用它构建的系统类型。比如，在大规模应用中，JavaScript的错误极为常见，你打开浏览器开发者工具几乎在每个网站上都能看到各种错误和警告，但这些日志中的报错并不意味着网站无法使用。浏览器不会轻易因为JavaScript报错就崩溃，通常只是某个功能失效，例如事件监听器不再触发，导致“点击无响应”。这种问题在Sentry的早期产品中不容易被察觉，直到我们推出Session Replay功能，才更好地将错误与用户行为联系起来。

相反，在用C++开发的电脑游戏中，如果程序崩溃，整个会话就会中断。虽然此类崩溃事件数量远低于JavaScript的报错量，但每一个都更具意义。也因此，不同语言的错误率难以直接比较，因为每种错误的影响范围完全不同。以Sentry的数据为例，C++崩溃报告的数量非常少，但每条报告的价值都很高。

当然，有些错误类型你见得多了会觉得“不该再发生”。在JavaScript社区中，类型检查器的普及确实减少了一部分可预防的错误，至少强制开发者在编译时显式处理空值，但我并没有观察到TypeScript的普及对整体错误率有明显影响。理论上，类型安全的语言应能减少某些低级错误，但在实践中，这种改善微乎其微，几乎无法量化。

这也可能与另一个现象有关：开发者在得到更安全的工具后，往往会更大胆地构建复杂系统。Sentry发展过程中正处于“复杂化时代”，错误的来源也从“是否空值”变成了“微服务之间的版本不匹配”之类的问题。类型检查在这类场景中无能为力——网络层返回的空值仍然可能导致异常。

随着React等框架的普及，错误类型也在演化。比如“水合错误”（hydration errors）在React出现前根本不存在，但如今已成为常见问题。这些错误源于服务器端初始渲染与客户端动态渲染内容不一致。可以说，应用越复杂，错误种类就越多。

> Gergely：像React这样的框架带来了全新的错误类别——以前根本没有，现在却随处可见。所以，从某种意义上说，错误甚至“越来越多”。

Armin：做错误监控的生意是个“安全的生意”，因为错误永远不会消失，只会不断演变。

> **Gergely：而且随着AI工具的普及，尤其是那些训练于最常见框架的模型，我们可能会看到代码量与部署量的爆炸式增长，这对错误监控来说也是机会。既然你们长期处在“错误的前线”，Sentry团队在开发过程中有没有因为这些经验而改变对错误的处理方式？你们是否更有意识地在设计阶段考虑错误问题？**

Armin：很多人会以为，在一家做可观测性（observability）产品的公司工作，团队自然会在构建自家系统时特别擅长处理错误与监控。但事实上，错误往往仍是“事后补救”的部分。要让团队真正重视那5%甚至1%的异常情况，需要强大的自我约束与文化建设。每次提交代码时，当测试通过后，很少有人会额外花时间去确保系统在失败时也能正确上报错误。

就我观察，Sentry的代码在错误处理方面并不比其他公司的普遍更完善。因为“良好的错误报告”“合理的日志设计”“完整的指标采集”这些事情都需要刻意为之，而非自然产生。我们或许能比普通团队做得好50%或100%，但整体上仍远远不够。要真正改善，理想的提升幅度应是千倍，而这只有在我们彻底改变编程方式、让语言自身在每个堆栈帧、每个异常路径中都原生支持这些能力时才可能实现。

从根本上说，我们要解决的不是“如何更好地分析错误”，而是“如何让错误数据更容易被收集”。这要求我们从编程环境层面重新思考：怎样的语言和运行时可以让可观测性成为内建能力。长期以来，开发者在这方面的主要障碍是“上下文传递”。Java的Ron Pressler在介绍虚拟线程和Project Loom时曾讲过，堆栈帧的伟大之处在于它能隐式地携带上下文信息。但当程序转向异步模型或基于Promise的结构时，这种天然的上下文携带性就丧失了。

此时，如果你想让一个“关联ID”贯穿整个调用链，必须在每个Promise之间手动传递，非常容易丢失。而有了堆栈帧，这种信息就始终存在，你可以随时向上回溯。

不同语言对此的解决方案不尽相同。例如，在 .NET中叫作“执行上下文”（Execution Context），在其他语言中则称为“Context Local”。它是一种轻量级的上下文存储机制，可以随逻辑执行流传播，而不局限于线程。对于分布式追踪或OpenTelemetry等场景，

这一点极为重要。我们在Sentry早期就反复强调“需要在各处支持Context Local”，但JavaScript的浏览器端至今仍未原生支持。Node.js后端经过多次演进，从早期的domain，到async hooks，再到现在的async local storage，才逐渐具备这一能力。

> **Gergely：即便像Sentry这样的团队，开发时的首要目标仍然是“让功能跑起来”，错误处理依旧排在次要位置。而且，似乎在许多语言的设计中，错误与上下文传递本身也像是“事后补充”。如果从语言设计阶段就考虑到“程序一定会出错”，并让这种上下文传递成为底层机制，或许情况会好得多。**

Armin：我倒不认为那是“事后补充”，更准确地说，这是“被有意忽略”的部分。因为语言设计总是要在不同需求间权衡取舍。以Context Local为例，它的最大问题是会让每次函数调用变慢。而对于追求性能的语言阵营，这几乎是不能接受的。要说服他们为调试性牺牲性能，难度极高。

一个典型案例是原生平台的编译器优化。过去，编译器为了多出一个可用寄存器，牺牲了堆栈寄存器（stack pointer）的保留方式，改用复杂的DWARF解栈机制来在调试信息中恢复堆栈。这种改动虽然提升了运行效率，却让实时性能分析和错误堆栈恢复变得极为困难。举个例子，这让我们在Sentry中处理原生代码崩溃报告时经常拿不到完整的堆栈信息，因为对应的调试符号文件往往缺失或不可访问。

Facebook曾为了解决类似问题，在Android应用中“偷偷”采集系统符号表，以改进错误报告质量。Sentry当然不可能那样做，那会损害用户信任。直到大约两年前，Debian或Red Hat的工程师才推动恢复堆栈指针的默认行为。虽然这样会带来大约5%的性能损失（Python特例高达20%，因此未启用），但他们仍认为这是值得的。可见，这背后其实是一场长达数年的博弈：调试能力与性能之间始终难以兼得。

> **Gergely：这确实让人意识到语言设计的艰难。每个选择都在牺牲某一方，要么牺牲性能，要么牺牲可调试性。语言必须服务不同的用户群：有的人只关心运行速度，有的人却更在意崩溃后的可恢复性，没有哪个方案能让所有人满意。**

Armin：我现在对语言设计师的尊敬比以往任何时候都高。编程初学者常常天真地以为“语言就该这样设计才对”，但随着经验积累，你才会理解各种权衡背后的复杂性。很多决定看似微小，但会深刻影响语言的使用者群体。比如，我曾觉得Python是构建

Web服务最理想的语言，因为它可以在运行时轻松检查进程状态而不明显拖慢性能。可后来才意识到，这正是Python性能偏慢的根本原因之一。

给“初创”的建议

> **Gergely：从担任初创公司早期工程师的创业经历中，你学到了什么？你会给现在加入一家快速成长型初创公司的工程师什么建议？**

Armin：如果从十年前加入初创公司的视角来说，我的经历其实是一条相对线性的路径：遇到一个机会，我评估是否想做，然后决定要不要接下。这有点像“秘书问题”（secretary problem），只是我没有尝试太多次，而是倾向于在觉得事情有趣、有价值时就全情投入。

我本身并不是一个典型的“好员工”，我不太适合被固定在某种角色或框架中。在Sentry工作这些年，我的头衔从“软件工程师”到各种管理职位都有，但我从未真正把自己与某个头衔绑定。早期的时候，我甚至要处理各种琐事：例如管理办公室、处理薪资、买家具。

所以，如果你加入一家早期初创公司，要做好心理准备：一切都处于混乱和变化中，没有清晰的边界或流程。有些人热爱这种不确定性，有些人会非常不适应。如果你喜欢尝试新事物、乐于应对模糊与挑战，那就勇敢投入；否则，你需要有非常清晰的目标与路径意识，知道自己要往哪里走，即使中途会有曲折或暂时的退步。

> **Gergely：现在你自己也创办了一家公司，从早期工程师变成了创始人。这个过程和你想象的有什么不同？作为一个曾经的“早期员工”，你现在在创业时有什么新的体会吗？**

Armin：某种程度上，我其实在很久以前就开始为创业做心理准备。回想Sentry初期的那几年，也许当时就存在另一条可能的路径，比如我们可以做别的产品。后来，当我的股份在四五年后完全归属时，我也想过是否要开始新的事情。那时我看到像Vercel这样的公司把Next.js打造成一个完整的业务体系，我就在想：是否也能把Flask打造成类似的平台？

这一次创业，我和联合创始人从“我们希望建立一家怎样的公司”出发，而不是

“我们想做什么产品”，这是与上一次最大的不同。我们都知道自己是那种会去创业的人，但这次我们更想先明确“为什么要创业”。当然，也可能会因此想太多，但回头看，如果当年Sentry能早点讨论这些基础问题，许多后来遇到的困难也许能更早化解。

所以现在我刻意在一开始就思考这些长远问题，比如股权结构、激励机制、合作方式等。经验让人更清醒：许多事情你可以在书上读到、在播客里听到，但只有亲身经历过，才会明白那种压力、紧迫感与责任感。十年前我就爱读各种创业故事，也以为自己理解了创业，但真正做起来，感觉完全不同。

> **Gergely：无论你读了多少创业故事、听了多少建议，只有亲身去做，才会真正体会其中的不同。**

Armin：对某些人来说，阅读和学习可能足以让他们做好准备，但对我不是。没有哪本书或播客能让我真正准备好。我成长过程中接受的教育很普通，并没有教我如何面对创业的不确定性或复杂性。这些都只能在实践中一点点摸索、学习。

> **Gergely：你最喜欢的编程语言是哪一种？**

Armin：Python。一是它给了我职业生涯的起点，这份情感很深；二是尽管它在设计上有许多缺陷，但却极其务实。我喜欢这种“实用主义”的精神。就像Flickr和Slack的前CTO Cal Henderson使用PHP的方式一样，他并不在乎语言“是否完美”，而是专注于“能否解决问题”。我用Python做产品时也是这种心态，它让我能高效完成很多事。

> **Gergely：有没有一个你特别喜欢的工具？**

Armin：如果不局限于编程工具，我最喜欢的其实是一把电动螺丝刀。它能拧螺丝——听起来很平常，但当我第一次买到一套真正高品质的电动工具时，我的生活发生了变化。我开始更愿意打孔、组装家具，也更享受动手创造的过程。

Gergely：这似乎也是一个隐喻：当你拥有好的工具，你会更有动力、更愿意去尝试新事物，也会变得更有冒险精神。

参考链接

- https://www.youtube.com/watch?v=45kVol96IIM

Python新版本去GIL刷屏，Karpathy点赞敢死队，Python之父：冷静，别神话并发

作者 Tina 核子可乐

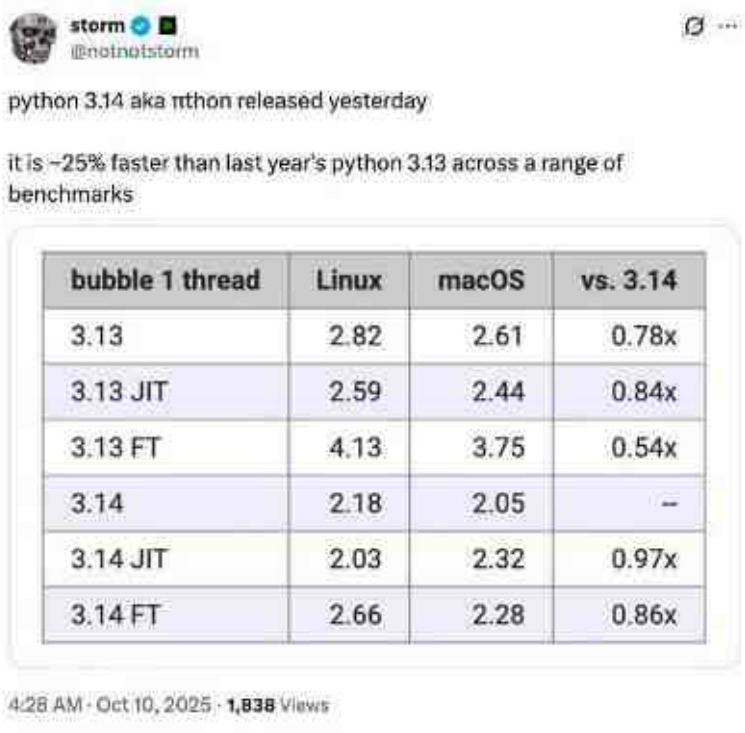
这周，Python 3.14正式发布，把悬念了多年的“去GIL（全局解释器锁）”写进官方发行版。

此次更新并非只是一项开关，而是一整套能力同步上线，涵盖自由线程支持、使用并发解释器、经过改进的调试器支持以及作为可选项的新解释器（有望将性能提升3%至5%）。

Python中的自由线程功能可禁用全局解释器锁（GIL），目前已在PEP 703中得到完整实现。它还配套了一个自适应解释器思路，源自Mark Shannon领衔的Faster CPython项目（尽管微软已在今年5月停止官方支持，相关成果已沉淀进实现）。

长期以来，GIL既像安全网也像减速带：通过“同一时刻仅允许运行一个Python线程”来保障内存安全、避免许多棘手的并发Bug，却也限制了CPU密集型多线程程序对多核的利用，除非借助繁琐的变通方案。如今，3.14提供的自由线程（no-GIL）构建移除了这道栅栏，使多线程能够真正并行，重计算场景下的性能收益尤为明显。当然，权衡也必须看见——单线程速度通常会略有回落，内存占用大约增加10%。这意味着开发者拥有了更清晰的选择权：需要稳妥与兼容时沿用带GIL的默认构建，需要极致并行时切换到no-GIL构建。

发布之后，许多开发人员报告说与过去的版本相比，速度明显提高。



storm
@noinotstorm

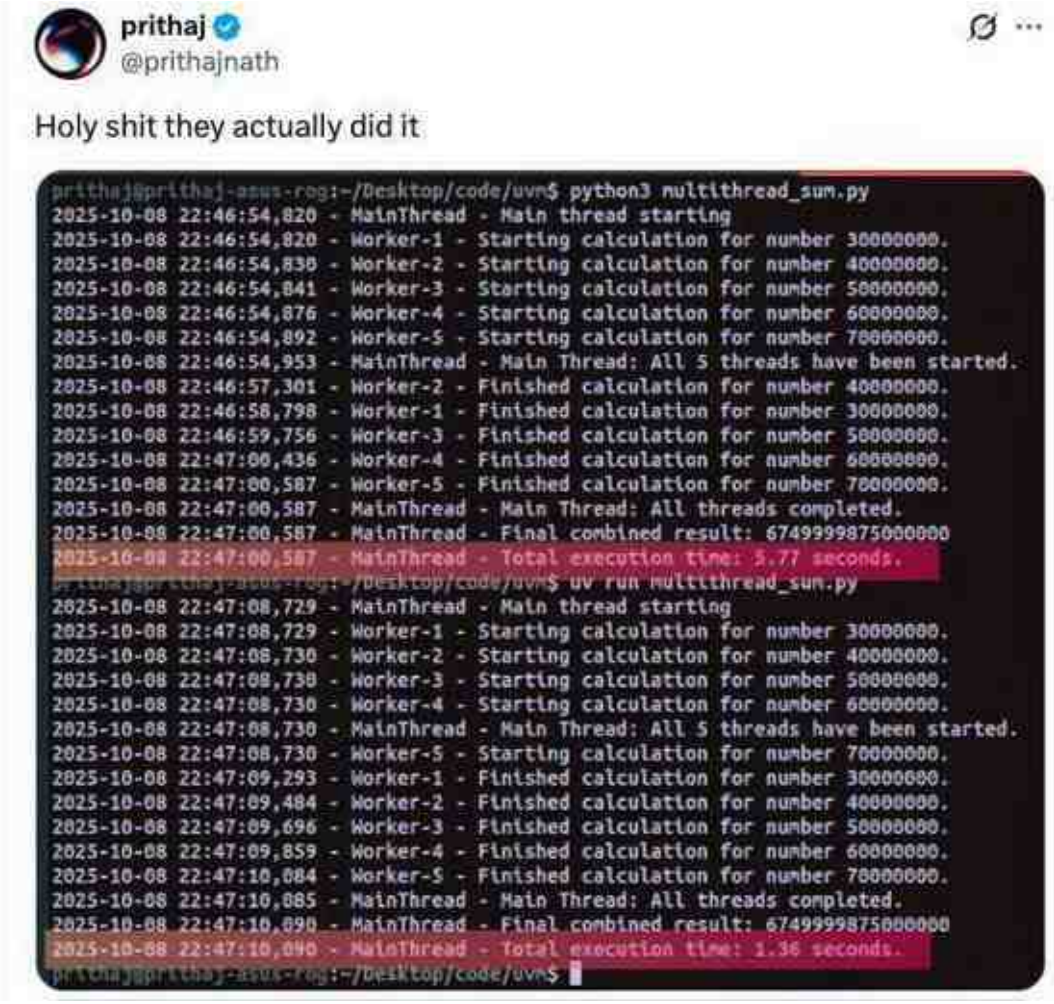
python 3.14 aka πthon released yesterday

it is ~25% faster than last year's python 3.13 across a range of benchmarks

bubble 1 thread	Linux	macOS	vs. 3.14
3.13	2.82	2.61	0.78x
3.13 JIT	2.59	2.44	0.84x
3.13 FT	4.13	3.75	0.54x
3.14	2.18	2.05	–
3.14 JIT	2.03	2.32	0.97x
3.14 FT	2.66	2.28	0.86x

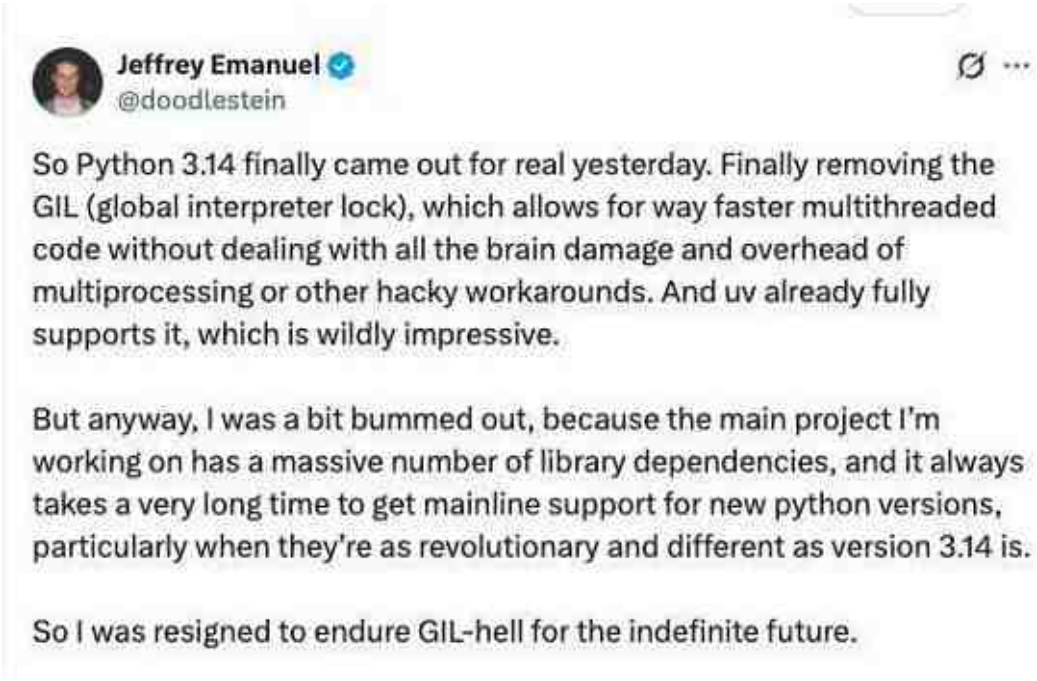
4:28 AM · Oct 10, 2025 · 1,838 Views

> 在一系列基准测试中，它比去年的Python 3.13快了约25%。



> 去GIL条件下，时间从5.77秒缩短到了1.36秒。

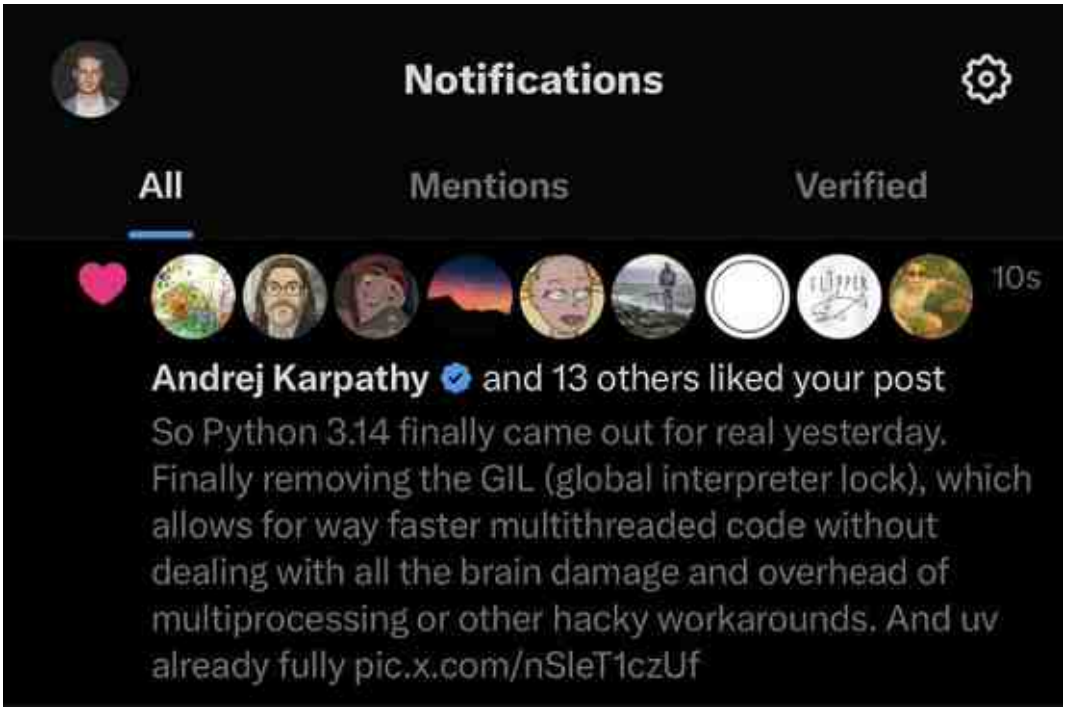
开发者Jeffrey Emanuel写道，“Python 3.14终于真的发布了。终于移除了GIL，这意味着多线程代码会快得多，不用再忍受多进程的脑损伤和各种临时变通。uv已经完全支持，这太令人印象深刻了。”在他看来，这是一个“革命性、差异极大”的版本。



> 截图来源：https://x.com/doodlestein/status/1976478297744699771

起初，他以为自己的项目（依赖PyTorch/pyarrow/cvxpy等）还会被困在“GIL地狱”。转念一想，不如交给codex/GPT-5：自动检索博客与issue，必要时vendor部分库，用C++/Rust从nightly源码构建替代轮子。经过数小时多轮迭代，最终全部跑通。与“大模型之前”的年代相比，此类升级既耗时又高风险；而现在，“我们正在生活在未来”。

Andrej Karpathy也在X上点赞这个贴文，就像是一记注脚：这一次，Python的并发故事真的开始加速了。



不过，和社区对“去GIL”与AI的狂热相比，Guido van Rossum在发布后的一次专访中给出了更为冷静的提醒：他直言去除GIL的重要性被高估，去GIL主要服务于超大规模并发场景，同时抬高了CPython的贡献与维护复杂度；AI炒作太过了，“归根结底还是软件”，他并不迷信“AI驱动的未来”，更强调代码必须可读、可审；工程常识优先于热点——并发模型并不易掌握，很多人在并行化后反而变慢，与其逐热，不如把生态兼容、可维护性与长期演进放在更优先的位置。

以下是Guido van Rossum访谈的翻译：

Python之父怎么看?

> **问1：《Python之禅》强调简洁性和可读性。随着AI与机器学习系统日益复杂，您认为这些核心原则是否比以往任何时候都更加重要，或者说应当在这个新时代重新评估这些原则的意义？**

Guido van Rossum：代码仍然离不开人类的阅读和审查，否则我们可能会完全失去对自身生存的控制力。而且看起来，Python等秉承“人文主义”哲学的语言也适合用来编写模型——毕竟大语言模型擅长的就是处理人类语言结构，而编程语言终究也是为

人类设计了。因此只要经过相应训练，大模型将很擅长阅读和输出这类语言。另外，大部分大模型都接受过良好的Python语言能力训练。

> **问2：在最初缔造Python的时候，您有没有想过它会成为科学计算与AI领域的主导性语言？您认为它在这些领域的意外成功是由哪些因素促成的？**

Guido van Rossum：我完全没想到过！我不是个很有野心的人，现在回想起来成功的关键因素可以说有两个。首先，作为一种语言，Python既易于理解、且功能极其强大。正如Bruce Eckel所说，“它特别贴合人类的脑回路”。第二个因素则是，我在设计Python时会考虑如何让它跟操作系统服务和第三方库良好集成。这样它的功能将更多样也更加可扩展，例如允许NumPy之类的主流库独立于Python本体之外进行开发。

> **问3：近期关于全书解释器锁（GIL）可选化的研究以及对AI性能的普遍需求成为热点。那您会如何看待Python并行性与并发性的未来？这两点对Python语言的长远发展又意味着什么？**

Guido van Rossum：老实说，我觉得移除GIL这事的影响被过度夸大了。这既满足了我们最大用户（例如Meta）的需求，同时也给CPython代码库的潜在贡献者招了点麻烦（新代码更容易引入并发性bug）。

我们经常听到有人说，他们在尝试了代码并行化之后速度反而变慢了——这让我认为大众对于Python编程模型的理解仍有进步的空间。

再有，我担心Python会变得过于企业化，因为大企业客户只会为他们需要的新功能付费。这里澄清一下，企业客户虽然不会花钱让我们替他们实现功能，但会提供开发人员参与贡献，所以其实是一回事。

> **问4：您在Python引入类型提示中是位关键倡导者。那您如何看待静态类型在Python中的演进？您认为它在构建我们如今熟悉的大规模关键任务型AI应用中扮演着怎样的角色？**

Guido van Rossum：我没听说过有哪些大规模关键任务型AI应用，我只知道有很多大规模关键任务型非AI应用。对于这类应用程序来讲，类型提示非常非常重要，也是其他工具对代码库进行有效处理的前提。我认为类型提示的应用阈值大概是一万行代码

——低于这个阈值，类型提示的价值就会降低，因为开发者自己就能记住其中大部分代码，传统动态测试也基本应付得来。而一旦代码量达到上万行，缺少了类型提示几乎没办法保证代码质量。当然，我不会强迫语言初学者使用类型提示。

> **问5：从Python 2到3的过渡应该是语言发展史上意义最重大，同时又充满挑战的篇章。这段经历带来了哪些重要的经验教训，在新范式已经出现的现实条件下，这些经验教训又将如何指导Python未来的发展态势？**

Guido van Rossum：我不觉得这里需要专门强调范式的变化，毕竟范式转换其实就意味着过往的经验已经无法帮助我们理解新的现实。但要说最关键的经验教训，那就是对于任何未来的版本更新，哪怕只是从3.x到3.x+1，我们都必须始终考虑如何在不改变旧应用形态的前提下实现支持。总而言之，迁移方法必须经过慎重考虑，特别是大多数库都必须同时支持多个版本（我们在2.0到3.0的升级过程中对此体会不深，也没有制定出好的解决方案）。

> **问6：Python的简洁性是其备受赞誉的特性之一。随着新的、更强大的AI库引入更多抽象层和复杂性，您认为社区该如何更好地保持Python语言的易用性，防止它给初学者带来更高的准入门槛？**

Guido van Rossum：到目前为止，我使用的AI库都不算特别强大或者复杂——它们只是让人们能跟远端实际提供AI服务的设施进行通信。相比之下，很多互联网协议反而更复杂，而且也没什么本质上的区别。

要说差异，那就是AI提供商的节奏太快了，有时候每几周就变更一次API，提供的说明文档也是质量低下、内容混乱。总之我们会继续遵循长久以来的实践思路——用库和API构建这个世界。

Python经历过计算领域的一系列剧烈变革，却仍然屹立不倒，所以我对这个问题并不太担心。早在90年代初互联网几乎不存在的阶段，那时微软还通过软盘和光盘分发软件；我们还经历了互联网和万维网的发展，从集中式计算到个人电脑，再到浏览器端软件以及硬件的巨大改进，Python都平稳走过来了。

> **问7：鉴于现代AI开发的独特需求（从数据处理到模型训练），如果您现在可以**

> 在Python内核中添加一项主要功能或者进行一项变更，您会怎么做？为什么？

Guido van Rossum：我什么都想不出来。**我还是觉得现在AI炒作太甚了，它的本质仍然是种软件。**在我自己的AI使用体验中，确实会借助一些小型库来发挥它的技术能力，也就是利用AI的优势以传统方式高效处理数据库操作。我们也有一部分代码是由所谓“智能体”编写的，但我们不会使用氛围编程。我们在架构和API设计方面一直保持高度自主可控。

> 问8：像Mojo和Julia这样的新兴语言专门为高性能AI场景开发而成。您如何看待这种新兴的竞争态势？Python要如何继续保持自身领先地位，并且在未来十年的技术进步中始终维持自身优势？

Guido van Rossum：Mojo强调成为高性能AI的实现“内核”，这要求对计算机进行非常严格的分类优化。所以它既不可能取代Python的生态地位，也并不打算这么做。我印象中Julis好像跟高性能AI关系不大——它更多面向高性能数值计算，就是说既能服务于AI，也能服务于其他具有苛刻要求的应用领域。

> 问9：您的角色已经从终身仁慈独裁者（BDFL）演变为微软杰出工程师。这种角色转变对您在Python开发、社区治理以及更广泛的企业生态系统中的定位产生了怎样的影响？

Guido van Rossum：这肯定代表着地位的下降。我一直担任BDFL，直到Python的所有治理职责不再由一个人承担，这其实就标志着我已经退休了。而之后选择微软，是因为我意识到自己仍然想要继续编程。在供职于谷歌和Dropbox之后，告别了鲍尔默时代的微软似乎确实是个体验更多编程乐趣的好地方。

> 问10：回顾您与Python合作的精彩历程，展望AI驱动的未来，您希望Python能够在历史上留下怎样的一笔？从个人角度来讲，您如何看待未来几年编程这个概念的变化？

Guido van Rossum：我绝不向往一个由AI驱动的未来。我担心的并不是AI会毁灭整个人类族群，只是我看到过太多缺乏伦理道德之人，往往用极低的代价就对全社会造成了巨大破坏。如今这股炒作弊端的根源就在社交媒体——AI只是又一次重大的计算机

范式转变，它确实会颠覆社会，但并没有真正影响软件的本质。

我希望Python的传承能够始终体现其草根精神和全球协作精神，始终依托于平等和尊重，而非权力和金钱。希望Python始终为“平凡人”赋能，帮助他们编写出灵感中的项目。

参考链接

- https://www.odbms.org/blog/2025/10/beyond-the-ai-hype-guido-van-rossum-on-pythons-philosophy-simplicity-and-the-future-of-programming/

Cloudflare用Rust重写核心系统：CDN性能提升25%，响应时间缩短10毫秒

作者 Sergio De Simone 译者 田橙

Cloudflare近日宣布，已将公司核心子系统之一用Rust语言重写，实现响应时间缩短10毫秒、整体性能提升25%。公司同时强调，Rust不仅提升了系统性能，还增强了安全性，并缩短了开发周期。

在成功将Pingora子系统迁移至Rust之后，Cloudflare的工程师决定从零开始重写公司最老、也是最关键的组件之一——FL，即“Cloudflare的大脑”：

FL是Cloudflare的中枢。当一个请求到达FL后，我们会在网络中运行各种安全与性能功能。它会应用每个客户独有的配置和设置，从执行WAF（Web应用防火墙）规则、DDoS防护，到将流量路由至开发者平台和对象存储服务R2。

Cloudflare的架构师们决定基于公司内部开发的代理框架Oxy来构建新一代系统FL2。Oxy支持监控、软重载、动态配置加载与切换等功能。

尤其值得一提的是，Oxy的内建“平滑重启”机制对代理系统至关重要。传统方式中，终止进程会导致所有活跃连接被强制中断；而Oxy则在需要关闭实例时，会停止接收新连接，但继续为现有连接提供服务，直到它们自然结束，从而实现无中断过渡。

对Cloudflare架构师而言，最大的挑战之一是：如何替换一个支撑了公司15年、仍在持续演进的运行系统。为了避免团队必须为旧版（基于LuaJIT的FL）和新版（基于Rust的FL2）各自重复开发功能，他们在FL内部创建了一个中间层，使得为FL2编写的Rust模块也能在旧系统中无缝运行。

这样一来，各团队无需维护两套逻辑，可以直接用Rust实现新功能，并逐步替换旧的Lua逻辑，而无需等待整个系统彻底迁移完成。

为了确保迁移过程顺利，Cloudflare的架构师还制定了明确的测试与发布策略。测试环节使用名为Flamingo的系统，可同时对FL1与FL2发起上千次端到端测试请求。每次更新都会逐步上线，在不同阶段接受递增流量测试与性能基准对比，以确保性能与资源使用保持在可接受范围内。

另一个关键机制是FL2的回退机制。当FL2无法处理某个请求时，它会将该请求交给旧系统FL1处理。得益于这一机制，Cloudflare能够在不影响整体稳定性的前提下，逐步增加FL2的实际流量占比。随着FL2的成熟，它处理的请求量持续上升，而回退至FL1的比例则相应下降。

Cloudflare架构师指出，构建FL2的最大收益来自性能提升。这主要源于两方面：一是FL2完全使用高性能语言Rust编写，而不再像FL1那样混合使用C、Rust与Lua；二是旧系统FL1在多语言间进行数据转换时耗费了大量时间。得益于此，FL2仅需原系统一半的CPU资源，内存占用也不到一半。

此外，FL2还从Rust的编译时安全机制中获益良多。Rust严格的静态检查、代码规范、全面测试和严格的代码审查流程，使系统在运行过程中极少出现崩溃问题。目前报告的大部分故障都与硬件故障有关。

原文链接

- https://www.infoq.com/news/2025/10/cloudflare-rust-proxy/

Cloudflare酿六年最惨宕机：一行Rust代码，全球一半流量瘫痪！ChatGPT、Claude集体失联

作者 华卫

半个互联网又又又断了。

刚刚，Cloudflare公司遭遇了持续数小时的宕机事故，导致多款热门网站和AI服务下线。据报道，此次服务中断持续约五个半小时，OpenAI的ChatGPT和Sora均在受影响应用之列，Claude、Shopify以及美国新泽西州公共交通系统的官网也出现了故障。

神秘流量激增，导致大范围宕机

据外媒报道，美国东部时间11月18日凌晨5点20分左右，Cloudflare首次发现平台出

现异常流量。约一个半小时后，该公司在状态页面更新公告，告知客户此次宕机事件，服务中断表现为出现错误提示及延迟升高。“Cloudflare内部服务出现故障。部分服务可能会间歇性受到影响，”Cloudflare在美国东部时间早上7点前不久发布的公告中表示。

而受此次宕机影响的并非仅有面向网站的CDN服务。故障还波及了其应用服务产品套件，该套件为云端及本地工作负载提供CDN功能，同时保护这些工作负载的应用程序接口免受恶意流量攻击。

Cloudflare在今年7月的一篇博客指出，全球约20%的网站依赖其管理和保护流量。据DownDetector称，此次宕机事件影响了包括X、Spotify、OpenAI的ChatGPT、特朗普的社交媒体网站Truth Social、在线设计平台Canva以及电影评分应用Letterboxd等，甚至DownDetector自己的网站也曾短暂受到影响。

此次宕机还影响了至少另外两项服务。在故障排查过程中，Cloudflare工程师关闭了伦敦地区的WARP虚拟专用网络（VPN）服务。此外，部分用户无法正常使用该公司的Cloudflare Access零信任网络访问（ZTNA）工具。ZTNA产品的用途与VPN类似，但能提供更优的安全性和性能。

美国东部时间11月18日上午8:09，该公司表示，问题“已查明，正在实施修复”，但恢复过程并不算顺利。美国东部时间11月18日上午8点13分左右，Cloudflare重新启用了伦敦地区的WARP服务。据Cloudflare称，控制面板服务已于美国东部时间上午9:34恢复。上午9点42分，该公司在状态页面宣布，工程师已修复宕机的根本原因。接下来的几个小时里，Cloudflare持续监控恢复进程，并“寻找加速全面恢复的方法”。最终，此次服务中断于上午11点44分结束。

Cloudflare的一位发言人向外媒证实，在发布第一份状态更新之前，他们发现“旗下一项服务出现异常流量激增”，这“导致部分流经Cloudflare网络的流量出现错误”。“我们全员出动，确保所有流量无误。之后，我们将集中精力调查流量异常激增的原因。”Cloudflare在声明中说道。

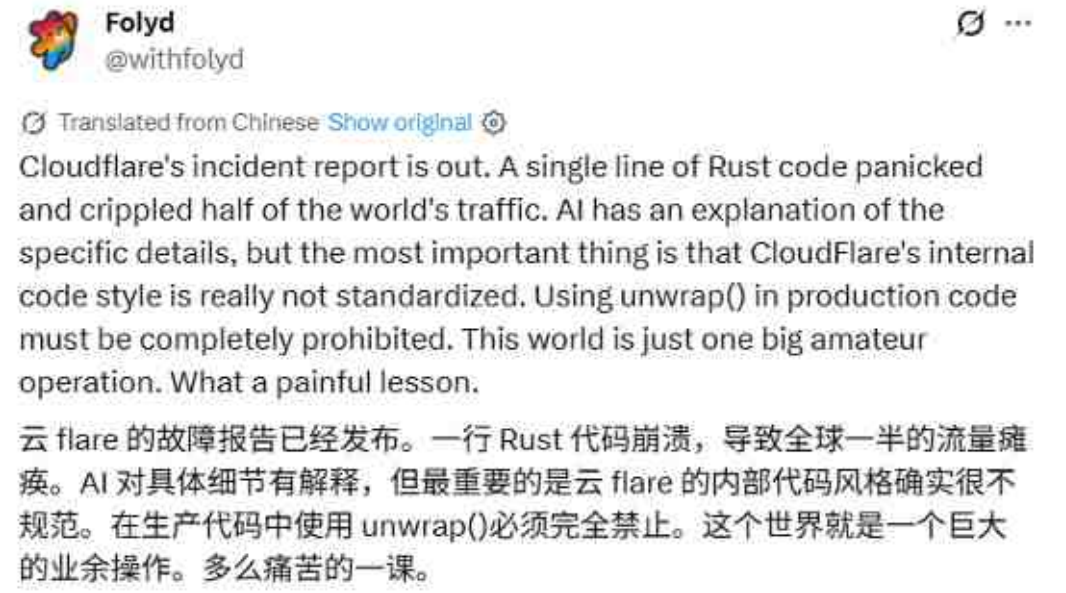
而从故障根因来看，有专家认为这次宕机并非单点失误，而是一连串低概率事件的叠加。Cloudflare在故障复盘中提到，问题最初源于一次数据库用户权限的变更，意外导致一条SQL返回了重复数据；再叠加上不够严谨的Rust代码实现、以及多年遭受DDOS攻

击带来的“PTSD式误诊”，几件本不至于造成灾难的小事，最终触发了今年全球范围内持续时间最长、影响最广的一次网络故障。

值得一提的是，在X平台上，有网友评价，“Cloudflare的Rust重写版本并未经得起时间的考验。”9月26日，Cloudflare采用“内存安全”的Rust语言重写核心代码。该公司称，得益于Rust语言的特性，此次重构“速度更快、安全性更高”。

Cloudflare故障报告中，专门指出了导致这次宕机的那行Rust代码。

```rust
/// Fetch edge features based on `input` struct into [`Features`] buffer.
pub fn fetch_features(
    &mut self,
    input: &dyn BotsInput,
    features: &mut Features,
) -> Result<(), (ErrorFlags, i32)> {
    // update features checksum (lower 32 bits) and copy edge feature names
    features.checksum &= 0xFFFF_FFFF_0000_0000;
    features.checksum |= u64::from(self.config.checksum);
    let (feature_values, _) = features
        .append_with_names(&self.config.feature_names)
        .unwrap();
```



“一行Rust代码崩溃，导致全球一半的流量瘫痪。”不少人认为，写过Rust的都知道随意使用unwrap都不是一个好习惯。也有人指出，“只有当配置文件有问题时，unwrap才会失败。”

还有一位声称“朋友在Cloudflare工作”的人士表示，“宕机是因为有个工程师试图修改一份旧配置文件，删掉了一堆看起来已经过时的代码行。结果发现，正是这些代码行在维持着他们路由系统的稳定。配置文件一经部署，一半的监控系统直接变红报警，整个网络开始出现一些甚至他们内部文档都无法完全解释的异常现象。修复过程得找回一份尘封已久的备份，回滚一连串自动重载操作，还要想办法让一个彻底乱了套的服务器集群恢复正常运行。”

并且，其透露，“当时（Cloudflare）办公室里满是红牛罐子，大家都在暗自慌神，还有个资深开发者一直在重复念叨‘啥也别碰’。”

官方披露：宕机的深层原因

Cloudflare运营着全球约20%网站所依赖的内容分发网络（CDN）。该平台通过创建网站内容的多个副本，并将其分布在全球各地的数据中心来运作。当用户访问网页时，Cloudflare会从距离用户最近的数据中心加载内容。该公司表示，这种架构能为全球95%的人口提供50毫秒或更低的延迟。

除了提升网站速度，Cloudflare的平台还有其他用途。将流量处理任务卸载到CDN可减轻网站运营商的服务器负载，进而提高运营效率。此外，Cloudflare还提供网络安全功能，能够过滤恶意机器人程序及其他威胁。

关于造成流量激增的原因，当晚，Cloudflare首席技术官Dane Knecht在X平台的帖子中透露，此次宕机由公司的恶意机器人流量过滤功能引发，并非攻击所致。这位高管强调，“我们的机器人防护功能所依赖的一项服务中存在潜在漏洞，在一次常规配置变更后开始崩溃，进而导致我们的网络及其他服务大范围出现性能下降。”

同时，Cloudflare发言人也向外媒提供了更详细的最新进展。据称，“此次宕机的根

本原因是一个自动生成的威胁流量管理配置文件。该文件的条目数量超出预期规模，引发了为Cloudflare多项服务处理流量的软件系统崩溃。”发言人表示，“需要明确的是，目前没有证据表明这是攻击行为或恶意活动导致的。我们预计，事件结束后流量会自然激增，部分Cloudflare服务可能会出现短暂性能下降，但所有服务将在未来几小时内恢复正常。”

在后续发布的博客中，Cloudflare进一步解释了出现故障的完整经过、受影响系统和处理流程。据称，“问题是由于我们数据库系统的一项权限更改触发的，该更改导致数据库向一个由Bot管理系统使用的功能文件中输出了多个条目。该功能文件的大小随后翻倍。预期之外的大功能文件随后被传播到构成我们网络的全部机器上。这些设备上运行的网络流量路由软件会读取这份特征文件，确保机器人管理系统能及时应对不断变化的威胁。该软件对特征文件的大小设有限制，而此次文件大小翻倍后超出了这一限制，导致软件故障。”

具体来说，“机器人管理”模块正是此次宕机的根源。据介绍，Cloudflare的机器人管理模块包含多个系统，其中一款机器学习模型会为流经其网络的每一项请求生成机器人评分。客户借助这些评分决定是否允许特定机器人访问其网站。该模型的输入数据是一份“特征”配置文件，这份特征文件每几分钟更新一次，并同步至整个网络，使其能够应对互联网流量的变化。

而正是底层ClickHouse查询行为的一项变更，导致生成的文件中出现大量重复的“特征”行。这一变化改变了此前固定大小的特征配置文件的尺寸，引发机器人模块触发错误。结果是，负责为客户处理流量的核心代理系统，向所有依赖该机器人模块的流量返回了HTTP 5xx错误码。这一问题还影响了依赖核心代理的Workers KV和Access服务。

其做出的变更是，让所有用户都能获取其有权访问的表的准确元数据。但问题在于，他们过去的代码中存在一个预设前提：此类查询返回的列列表只会包含default数据库的内容，该查询不会对数据库名进行过滤。随着他们逐步向目标ClickHouse集群的用户推出这一显式权限，上述查询开始返回列的“重复项”，这些重复项来自存储在r0数据库中的底层表。不巧的是，机器人管理模块的特征文件生成逻辑，正是通过这类查询来构建本节开头提到的文件中的每个输入“特征”。

由于用户获得了额外权限，查询响应现在包含了r0数据库模式的所有元数据，导致

响应行数增加了一倍多，最终影响了输出文件中的行数（即特征数量）。起初，他们还误判观察到的症状是由超大规模分布式拒绝服务（DDoS）攻击引发，但随后准确识别出核心问题，成功阻止了这份超出预期大小的特征文件继续传播，并替换为早期版本。

详细报告链接：https://blog.cloudflare.com/18-november-2025-outage/

六年来最严重中断，“真相”被嘲疯了？

在大范围宕机期间，Cloudflare的股价下跌了约3%。

“鉴于Cloudflare服务的重要性，任何宕机都是不可接受的。网络曾一度无法正常路由流量，这让我们团队的每一位成员都深感痛心。我们知道，今日辜负了大家的信任。”Cloudflare在博客中也表示。

并且，该公司说明了后续加固系统以防止此类故障的步骤，包括以下方面：

- 按用户生成输入的防护标准，强化对Cloudflare内部生成配置文件的接收校验；
- 为相关功能增设更多全局紧急关闭开关；
- 避免核心转储或其他错误报告占用过多系统资源；
- 全面审查所有核心代理模块的各类错误场景故障模式。

对于此次的宕机事故，Cloudflare承认，这是其自2019年以来最严重的一次宕机。“我们以往也发生过宕机事件，比如导致控制台无法访问，或是部分新功能暂时不可用，但在过去六年多里，从未出现过导致大部分核心流量无法通过我们网络传输的情况。”

据了解，该公司上一次重大宕机发生在6月，当时其超过六项服务下线约两个半小时。那次宕机由Workers KV数据存储平台的故障引发。

专家认为，这可不是Cloudflare一家的问题。AWS、Azure、Google Cloud、阿里云，全都翻过车。系统越复杂，链条越长，一个小小改动就能在依赖迷宫里撞出大灾。云计算的规模红利，正在被它带来的复杂度和系统性风险反噬。要解决这个问题，也许要借鉴其他行业的经验，把IaaS层当“算力电网”管起来，PaaS/SaaS层放开竞争，可能才是出路。

有网友评价，“这纯属Cloudflare自己搞砸了。一个小故障，就成了第一块多米诺骨

牌。”也有人认为，“这次宕机本身是件小事，但它暴露了Cloudflare自身服务之间过度的耦合问题，导致控制面板也无法访问它。如果控制面板可用，将能让许多服务更快地部分恢复功能。”

还有人发出疑问：“互联网真的需要如此严重地依赖单一供应商吗？”同时，亦有批评人士表示，此类宕机事件充分暴露了互联网的脆弱性，尤其是当所有人都依赖相同的服务提供商时。

参考链接

- https://siliconangle.com/2025/11/18/cloudflare-outage-briefly-takes-chatgpt-claude-services-offline/
- https://arstechnica.com/tech-policy/2025/11/widespread-cloudflare-outage-blamed-on-mysterious-traffic-spike/

扫码关注 InfoQ 公众号