QCon
全球软件开发大会

InfoQ 极客传媒

为 Coding Agent 构建智能上下文：Qoder 的 Context Engineering 实践

夏振华

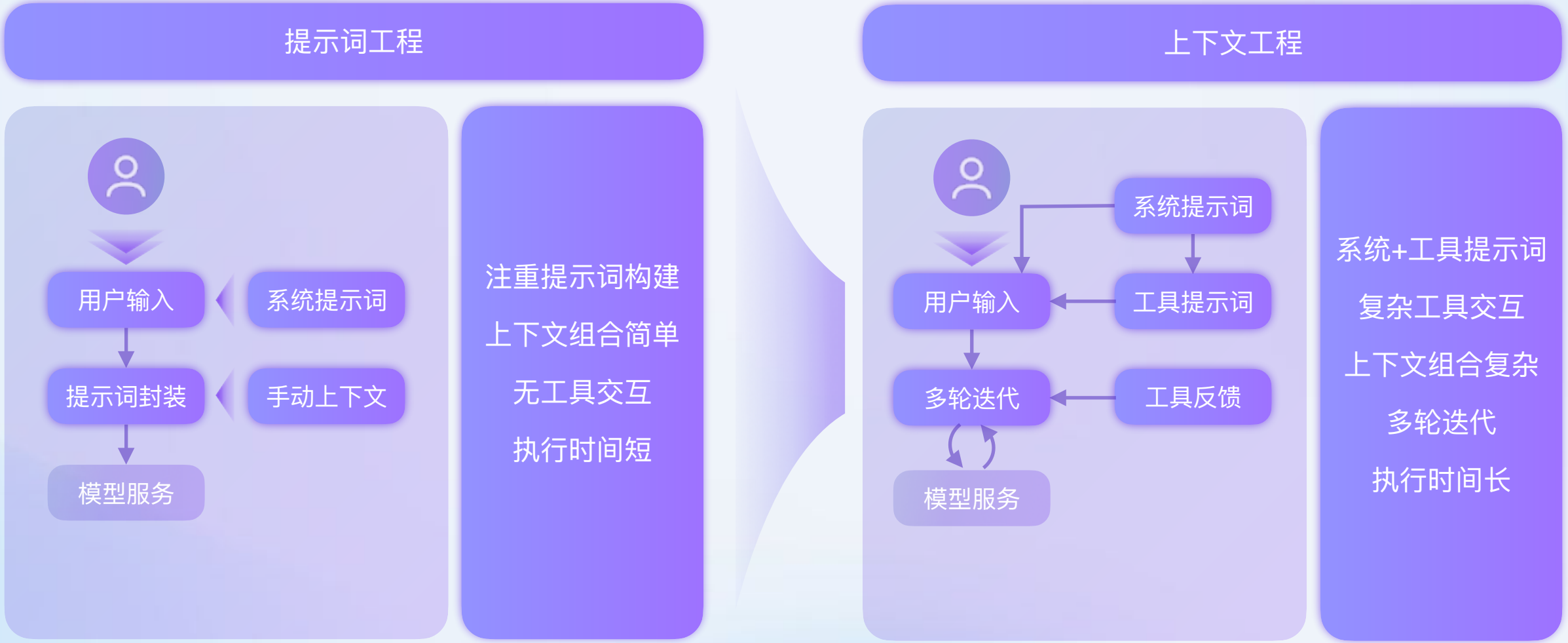
Qoder 工程师

目录

01

AI Coding 技术发展趋势

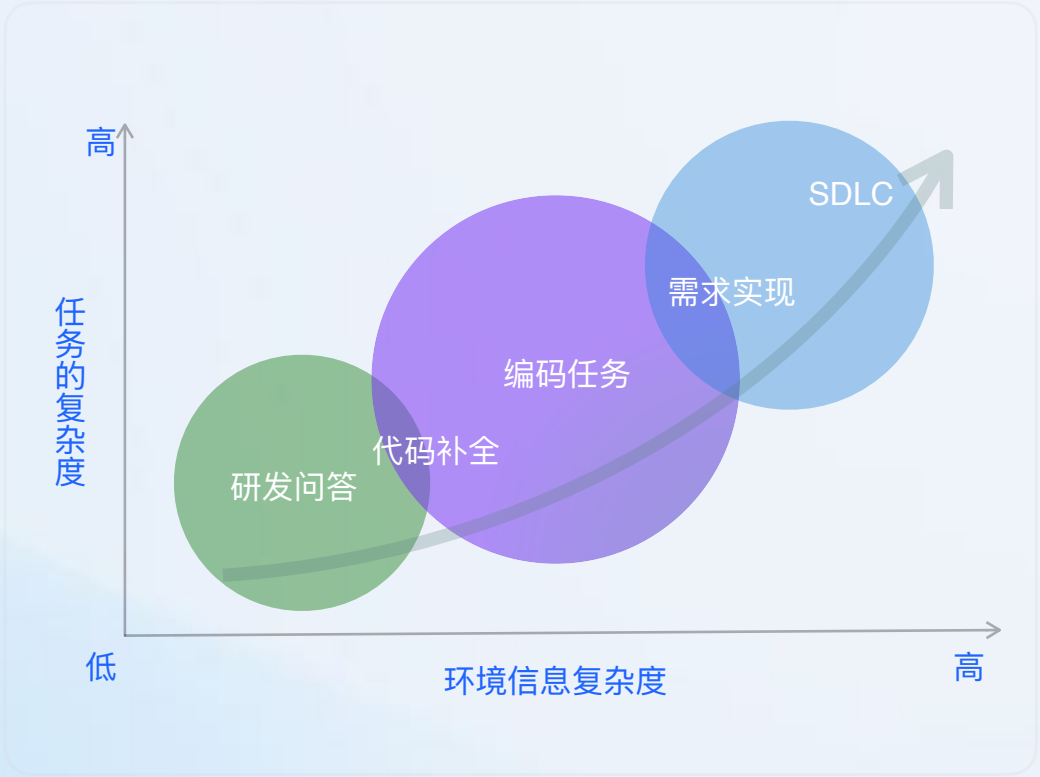
从提示词工程到上下文工程



从短任务对话协同到长任务异步委派

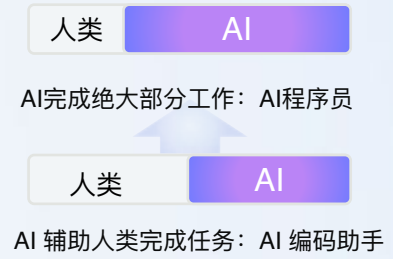
从智能代码生成到全链路软件研发

应用场景在扩大，产品边界在延伸



编程智能体正在全面集成传统研发工具

在IDE环境中的编程智能体可以使用MCP协议集成大量传统研发工具。同时异步委派智能体可以与DevOps工具深度集成，增强代码评审、质量检测、部署验证等环节。



沉淀企业研发经验和优质代码改进模型尤为重要

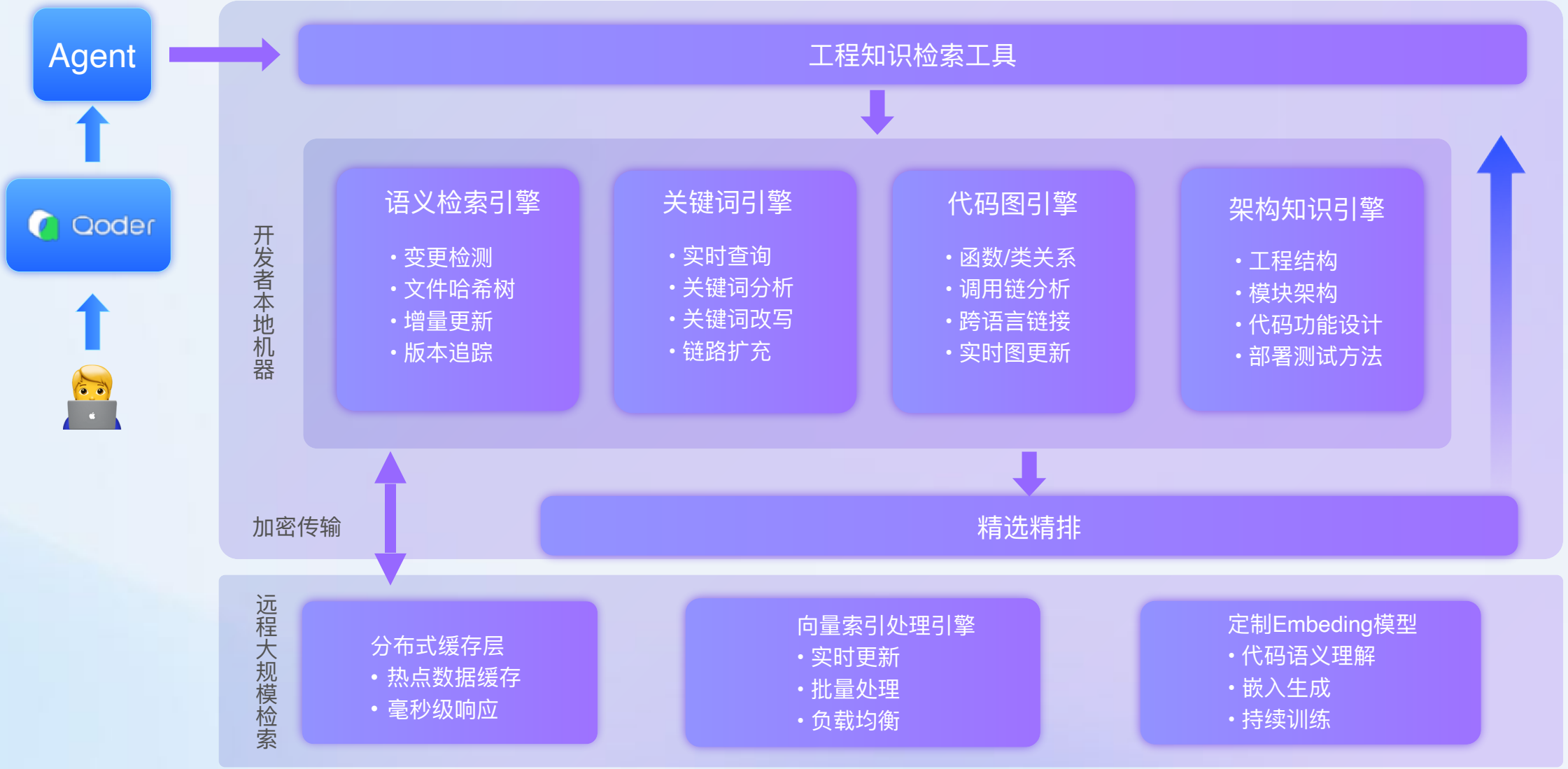
以数智大脑为核心，提供企业核心代码资产和研发组织资产，落地企业数字资产管理及治理能力，构建企业个性化私域知识库和研发过程经验，持续为数智大脑提供养分，提升大模型生成效果。

02

Qoder Agent 的技术架构

Qoder Agent 技术大图

新一代智能工程知识、代码检索技术



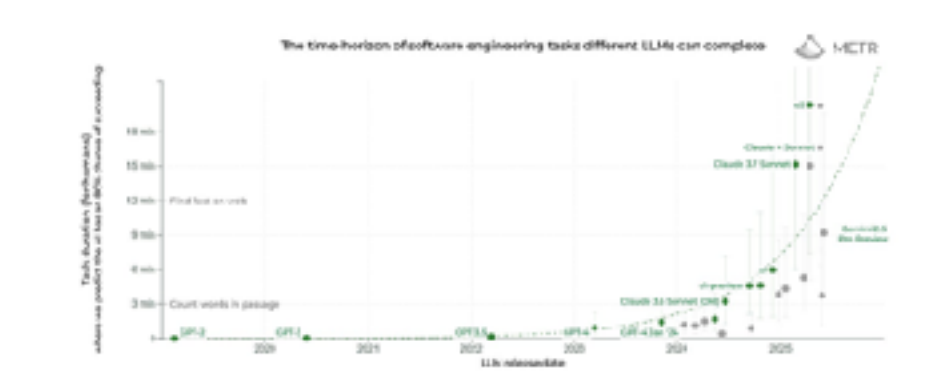
具备自优化能力的记忆感知技术

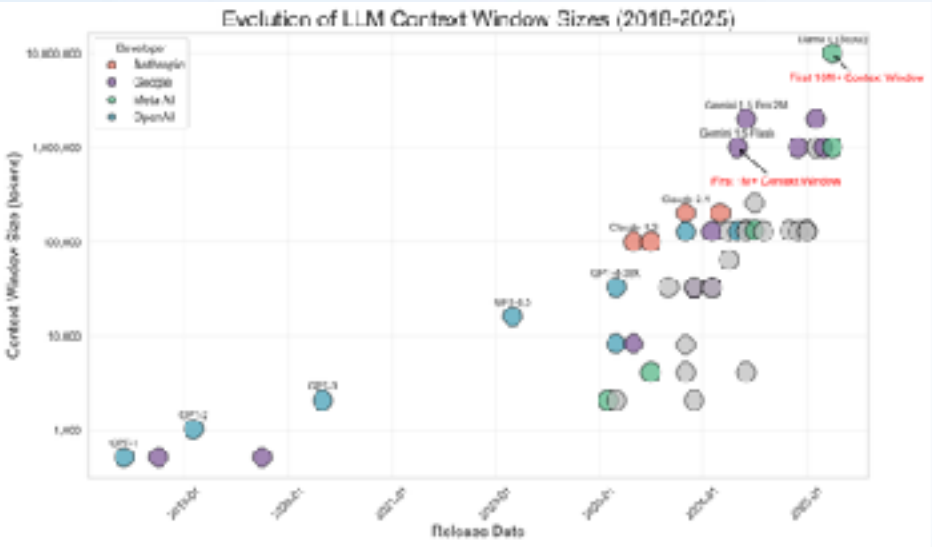
03

Qoder 上下文工程实践

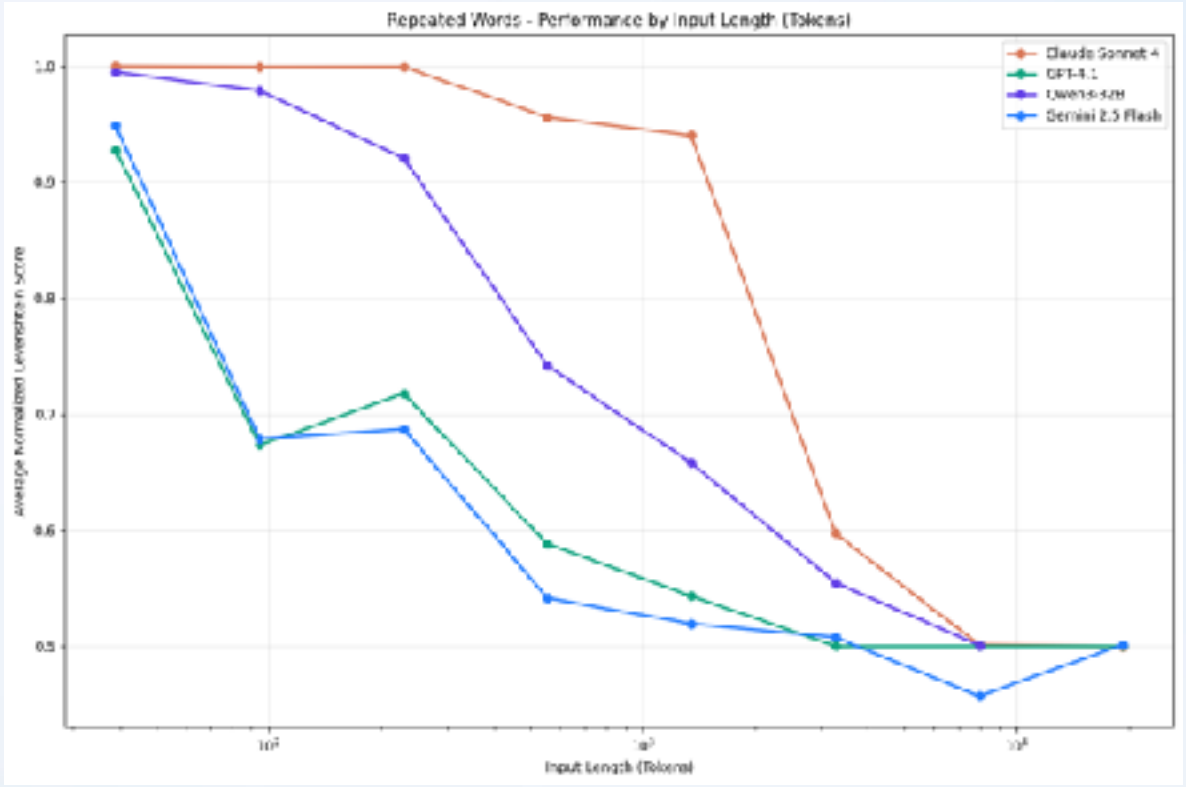
上下文相关的核心问题

模型可以完成的软件工程任务的时长 &
各模型厂商支持的上下文长度都在快速扩大





模型性能随着上下文长度增长却呈现快速下降趋势



参考：
https://research.trychroma.com/context-rot
https://www.oneusefulthing.org/p/the-bitter-lesson-versus-the-garbage
https://www.meibel.ai/post/understanding-the-impact-of-increasing-llm-context-wir

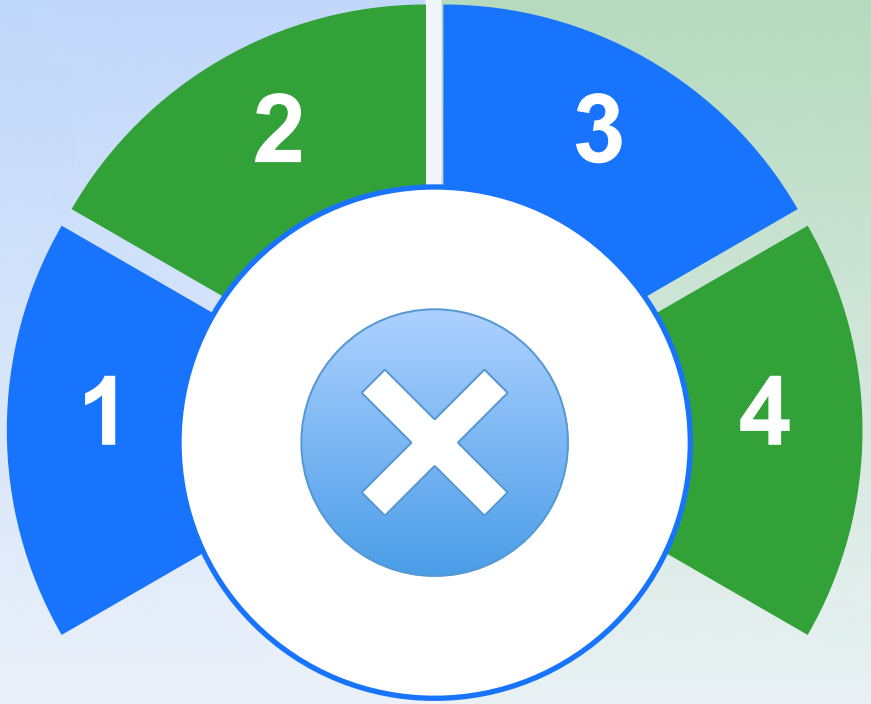
上下文相关的核心问题

Context Conflict（上下文冲突）

指上下文中包含了相互矛盾或不一致的信息，这会破坏模型的逻辑推理链，导致其无法得出正确的结论或执行正确的操作。

Context Poisoning（上下文中毒）

指一个初始的错误或幻觉信息污染了上下文，并在后续交互中被反复引用，如同“毒素”一样不断累积和放大，最终导致整个任务失败。



Context Failures

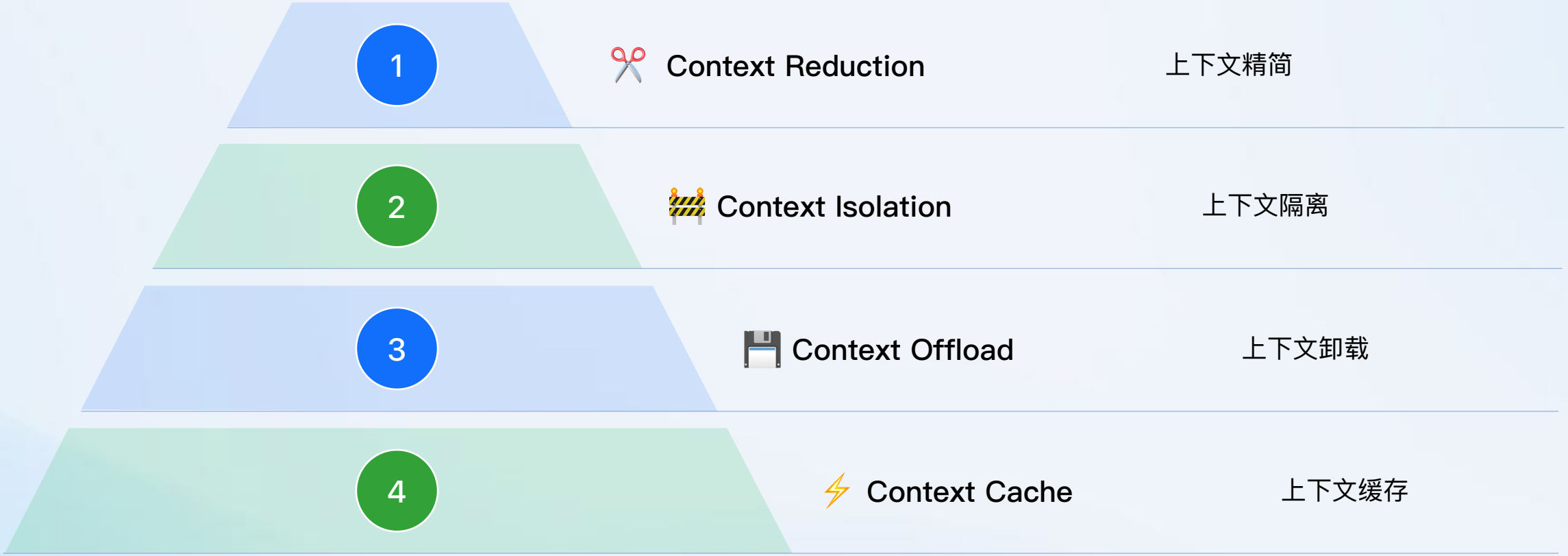
Context Distraction（上下文干扰）

指当上下文窗口过长时，模型会过度关注历史对话信息，倾向于重复历史行为模式，而忽略了当前的用户核心指令。

Context Overload（上下文过载）

指上下文中存在过多无关或冗余的信息（例如提供了过多的工具），这会干扰模型的判断，使其难以识别和调用最关键的信息或工具，从而生成低质量的响应。

上下文优化的核心手段


1
✂️ Context Reduction
上下文精简
2
🚧 Context Isolation
上下文隔离
3
💾 Context Offload
上下文卸载
4
⚡ Context Cache
上下文缓存

Context Reduction（上下文精简）

Context Compression（压缩）

对历史上下文里的工具执行结果进行压缩，保留关键工具的调用，裁剪工具输出结果

Context Summarization（摘要）

对历史上下文的完整交互历史，结合模型进行压缩，要求保留会话过程核心信息

Context Consolidation（合并）

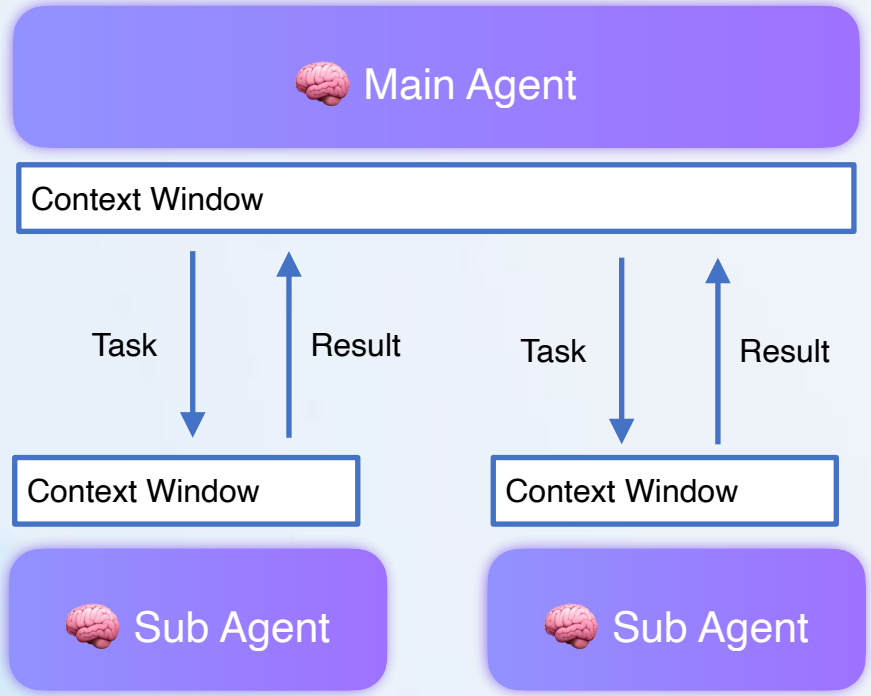
将多个相关的上下文片段进行智能合并去重，消除冗余信息，保持语义完整性

Context Filtration（过滤）

基于相关性评分过滤低价值上下文，保留与当前任务强相关的历史信息

Context Isolation（上下文隔离）

多 Agent 隔离上下文



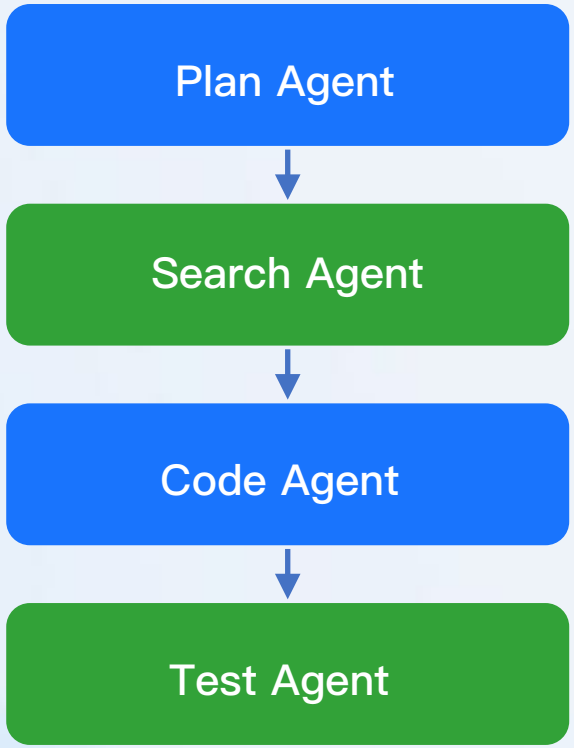
The Bitter Lesson Warning：

多数 Multi Agent 系统是在为当前模型局限打补丁。
当限制消失时，架构复杂度本身将成为限制。

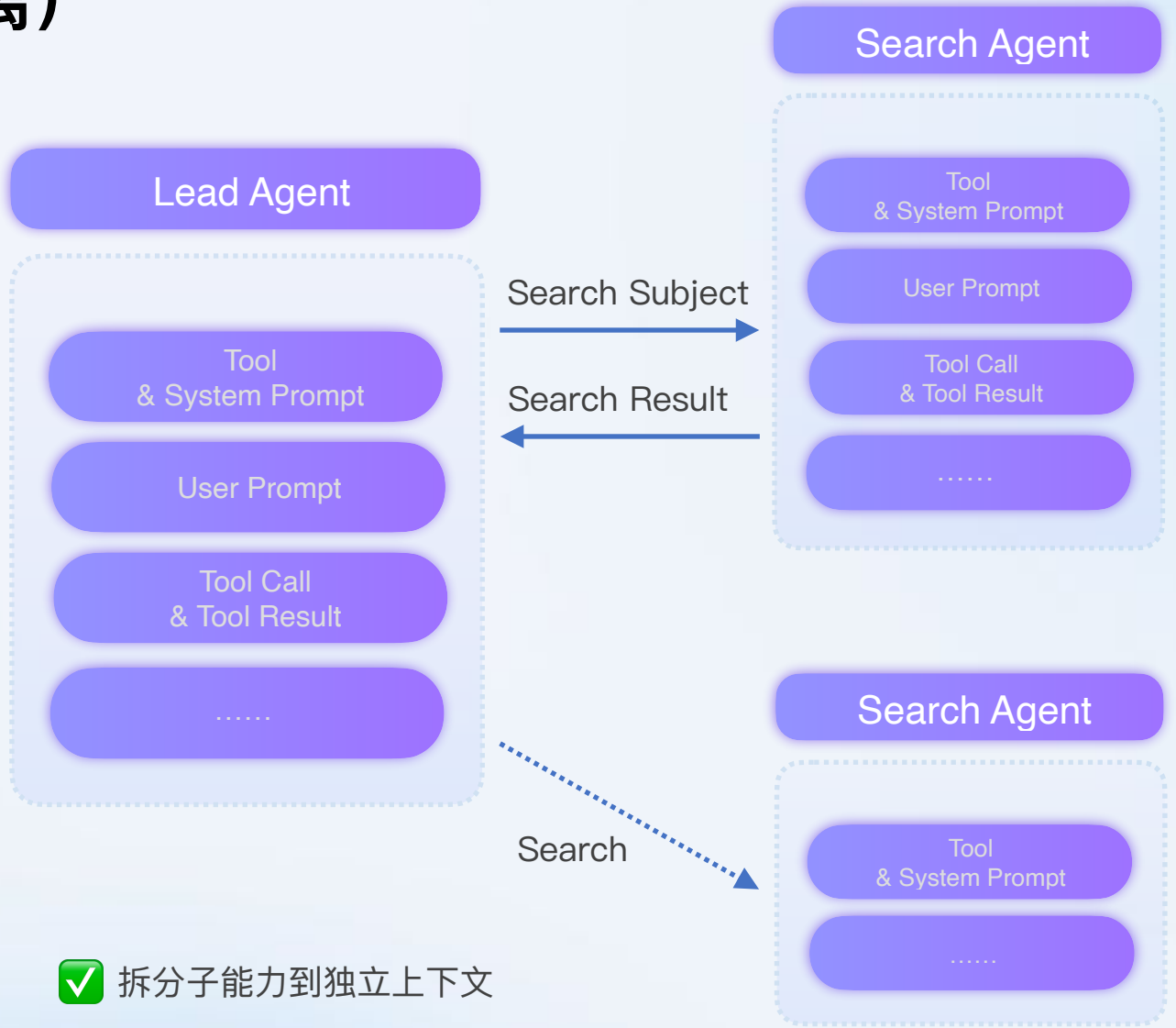
我们认为：

Multi Agent 的价值不是分解智能，而是隔离污染。

Context Isolation（上下文隔离）



按角色拆分上下文



✅ 拆分子能力到独立上下文

Context Offload（上下文卸载）

外部存储压缩
External Storage

将大量原始数据保存到文件系统或外部存储中，仅向LLM传递摘要或元数据，可实现较高压缩比同时保持信息可恢复性。

可逆性原则
Reversibility Principle

压缩信息时保留完整数据的访问路径(如URL、文件路径)，确保需要时可以随时检索完整内容，并精心设计摘要以保持信息召回能力。

分层记忆模式
Layered Storage Pattern

实现临时 TODO 记录和持久化长期记忆两种模式，使 Agent 能够在单次会话中做笔记，并跨会话访问历史沉淀内容。

易恢复性原则
Easy Restoration Principle

卸载内容应采用模型友好的格式和结构，便于通过原生工具快速加载并恢复上下文理解。

准确率、成本和性能的平衡

成本（Cost）

优先采用缓存机制和压缩策略降低token消耗。

通过KV-cache复用减少重复计算开销，在窗口边界实施智能压缩算法（如滑动窗口、注意力蒸馏）。对于超大规模上下文，利用外部存储卸载非热点数据，按需加载。
结合多模型调度，根据任务、代码仓库的复杂度等多维度信息智能路由，进一步优化成本。

准确度（Accuracy）

通过完善的检索机制和记忆系统，

确保上下文信息的完整性和相关性。采用语义搜索、向量数据库索引和多层级信息过滤策略，提升模型对关键信息的召回率。在长上下文场景中，结合分块存储和智能摘要技术，保证信息传递的精确度，减少幻觉和错误推理。

性能（Performance）

通过缓存优化和最大化并行处理能力提升响应速度。

优化prompt结构保证前缀稳定性，提升KV-cache命中率，减少重复计算开销。采用多Agent并行执行和单Agent内工具并行调用相结合的方式，最大化任务并发度。针对特定工具和场景，使用轻量级小模型加速生成过程

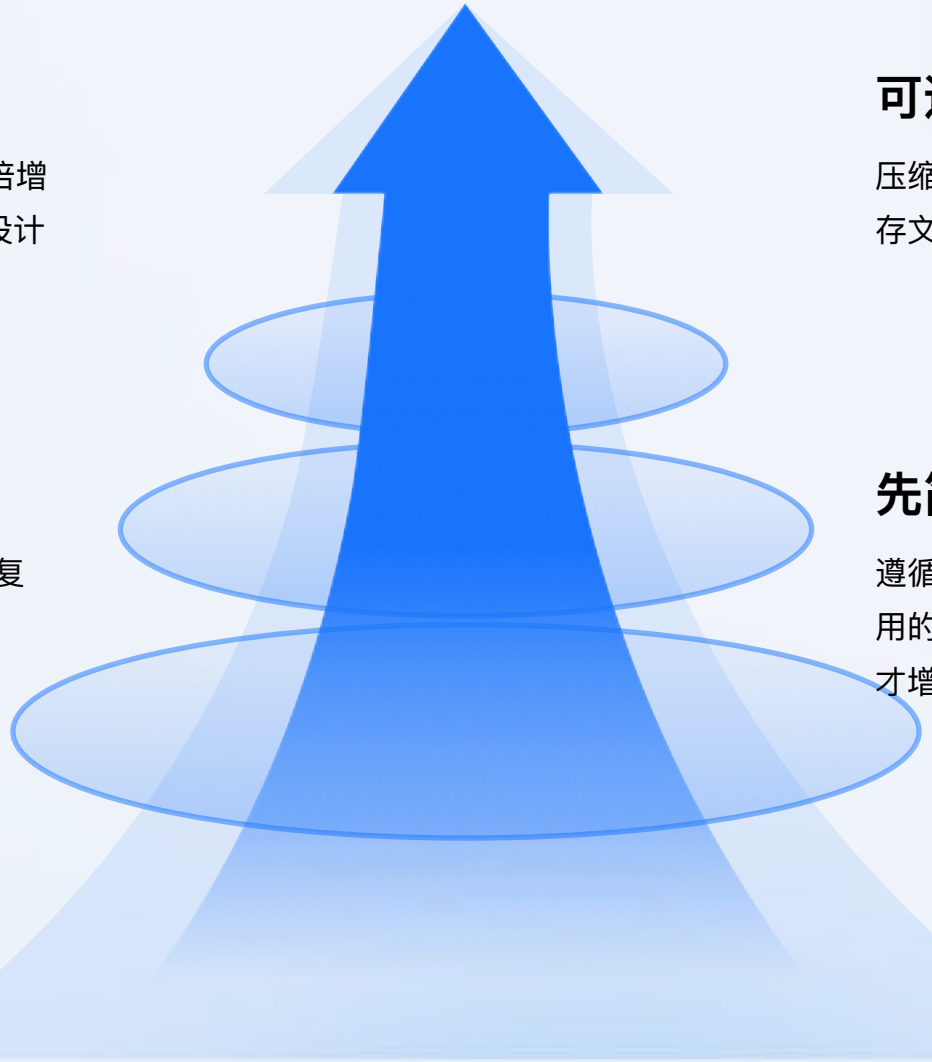
上下文实践的经验总结

追加式设计，避免修改历史

任何对历史上下文的修改都会使缓存失效并成倍增加成本，始终采用只追加（append-only）的设计模式。

可逆压缩，避免激进裁剪

压缩上下文时必须保留原始数据的恢复能力（如保存文件路径、URL），避免永久丢失关键信息。



保留错误信息，避免重复失败

将错误消息保留在上下文中可以防止 agent 反复犯同样的错误，错误是宝贵的学习机会。

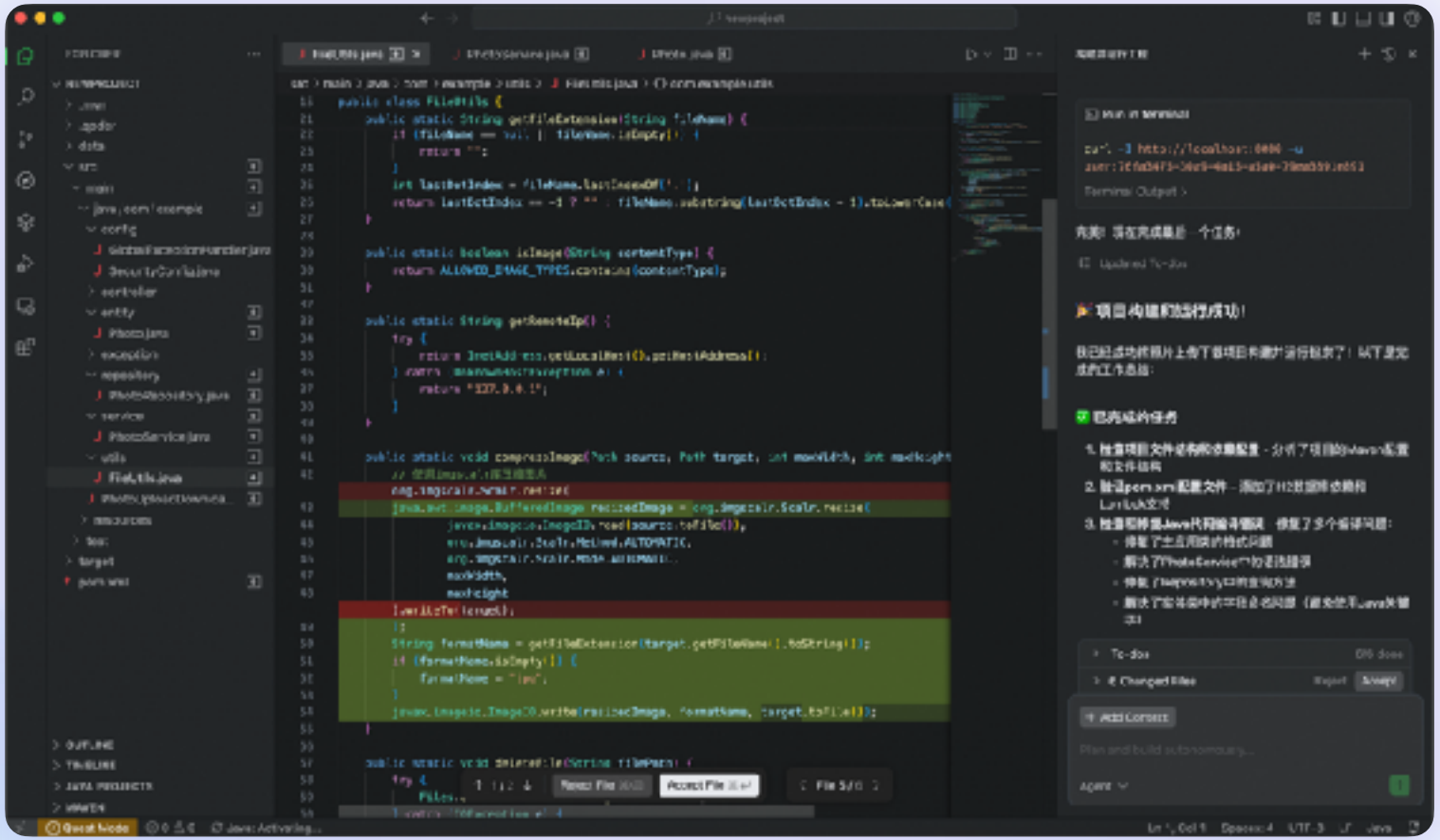
先简后繁，避免过早优化

遵循"Bitter Lesson"，随着模型能力提升，简单通用的解决方案往往胜过复杂设计，只在验证必要时才增加复杂度

04

Qoder 功能和案例分享

智能对话 Agent Mode



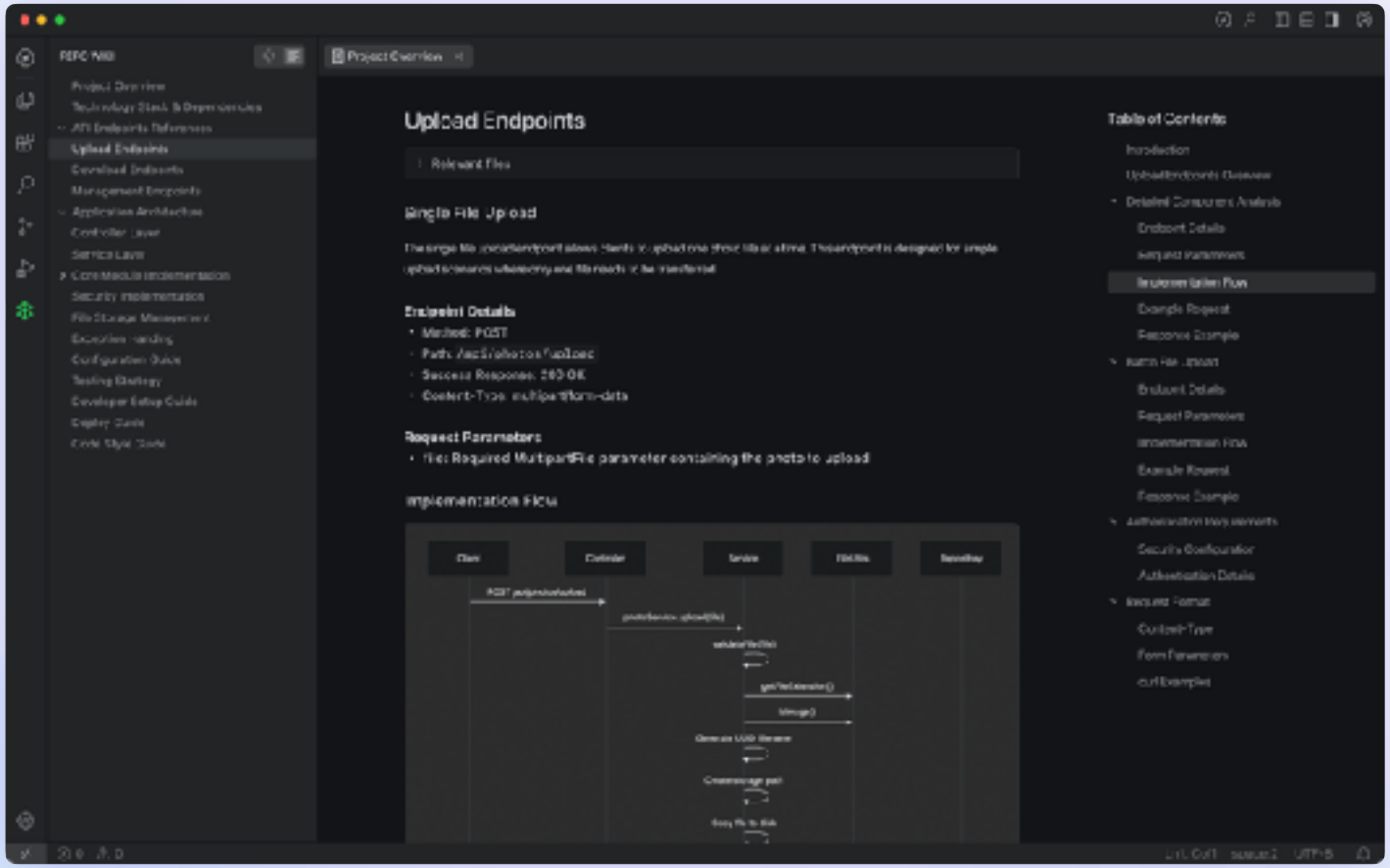
Agent Mode

用户价值：根据开发者输入需求，自主完成工程理解和文件编辑，与开发者协同迭代的方式完成编码任务。

关键挑战：如何实现复杂真实代码库上的工程理解能力，以及善用工具快速、准确完成任务的能力。

技术能力：代码语义+图谱+wiki+记忆的混合检索能力构建最强上下文工程，确保复杂工程编程效果。

工程知识可视 Repo Wiki



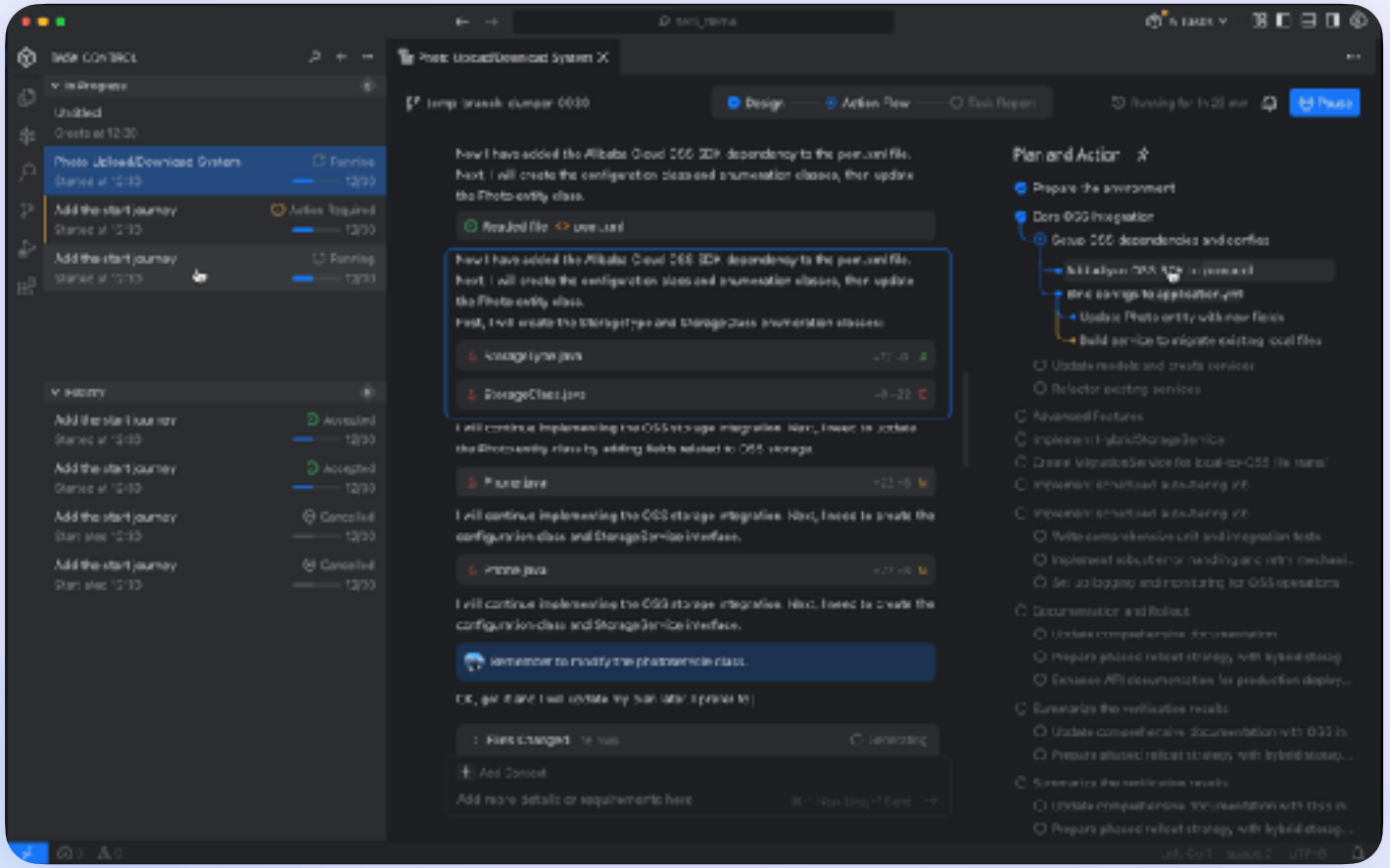
Repo Wiki

用户价值：在为开发者提供高效的工程知识学习的同时，让编程智能体具备了架构知识理解能力。

关键挑战：如何更快、更低成本、更准确的生成复杂真实工程的架构知识，同时让智能体完成更复杂任务。

技术能力：精调WIKI专用多智能体，专用WIKI模型，以及智能体代码混合检索技术。

异步委派任务 Quest Mode



Quest Mode

用户价值：通过Spec–Driven Development模式，实现智能体端到端交付各种编程任务。

关键挑战：如何实现Spec生成准确性，以及长程任务的完成准确、成本、安全问题。

技术能力：顶级代码大模型，多智能体架构，远程执行容器，以及云端代码检索技术，让异步委派编程模式得以落地。

命令行工具 Qoder CLI

Qoder CLI

用户价值： 让开发者在终端内完成从构思到实现的完整开发流程，无需切换工具，保持专注和高效的开发状态。

关键挑战： 如何在纯命令行环境中实现代码库深度理解、自然语言交互、多工具集成，以及复杂开发任务的自动化编排。

技术能力： 顶级大模型驱动的代码理解和生成能力，MCP 协议实现的工具生态集成，智能代理（Agents）系统支持任务分解与自动执行。

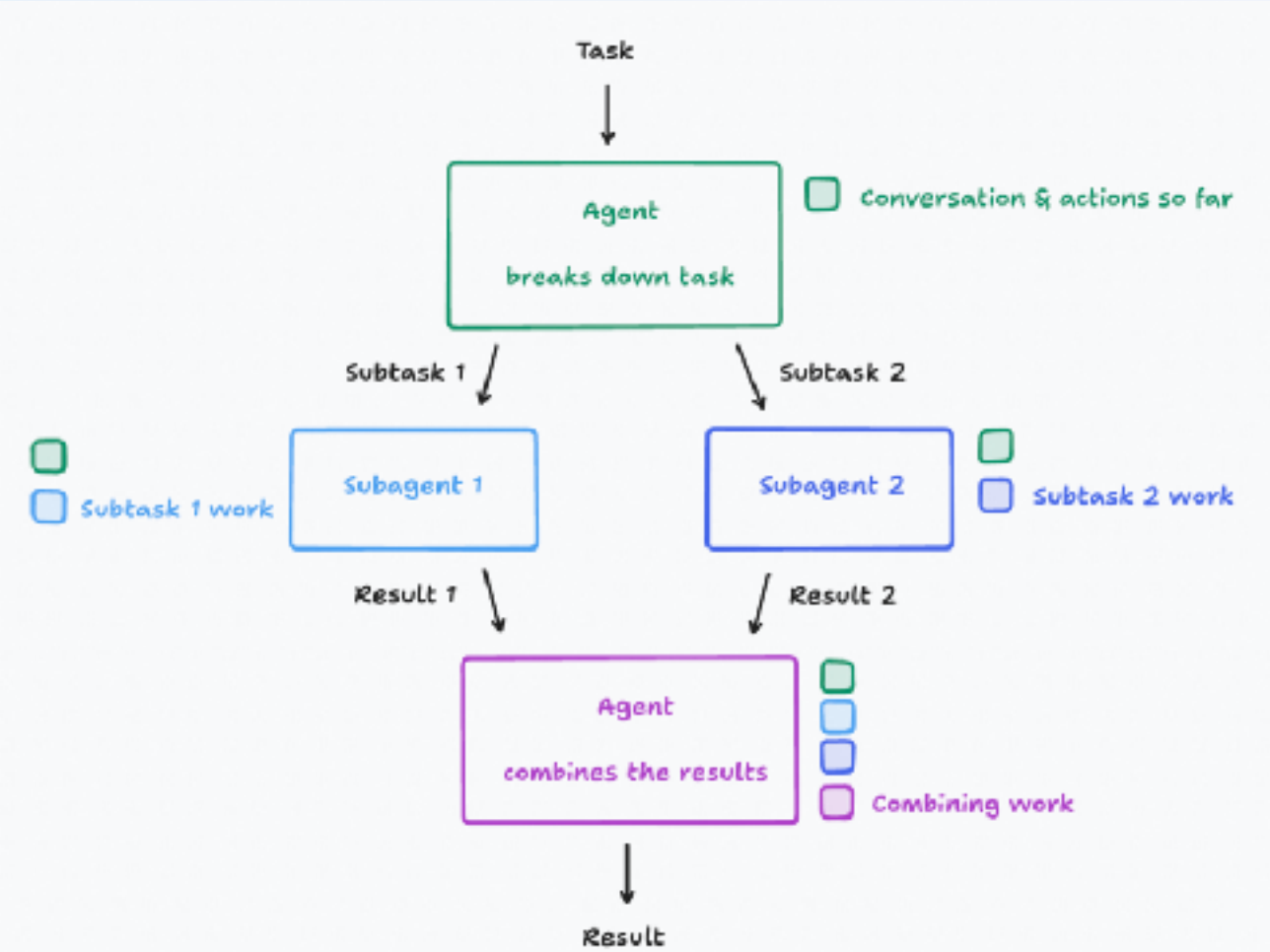
05

Qoder 未来演讲方向

从设计、执行到验证全链路软件研发实现更高准确率

Design	Act	Verify
人机协同针对复杂问题进行设计与澄清	在确定性代码任务下多轮迭代完成代码编写	修复lint错误，测试错误，界面错误
针对复杂问题，发挥模型的扩展性思考能力，结合现有工程架构给出详细技术设计。做好任务拆解，要求模型分步骤完成编写任务。	将工程规范、问题解决思路、任务要求输入给智能体。智能体使用本地工具完成代码编写。通过多轮迭代的方式逐步完善，弥补模型能力问题和幻觉导致的偏差。	AI 按照任务要求生成对应测试用例，并自主完成测试。发现错误后进行自动修复，与人类配合完成最终成品验收工作。
人类 AI	人类 AI	人类 AI
AI 根据需求进行设计，人类指导优化	人类提出明确要求 AI 完成工作	AI 编写测试用例并测试，人类进行验收

多智能体架构应对更长更复杂任务



图表引用：https://cognition.ai/blog/dont-build-multi-agents#a-theory-of-building-long-running-agents

问题、挑战、应对

关键问题：在 Coding 任务中是否需要引入多智能体架构，其收益和问题有哪些，适用于什么场景?

关键挑战：多智能体架构可以有效解决单智能体上下文窗口受限问题，但引入了多智能体之间信息共享、同步问题，反而让多智能体架构更易失败，消耗更多 token。

应对思路：独立边界任务，完整的信息传递，谨慎采用并行化。

更多专用领域模型突破复杂任务成本和性能瓶颈

专用模型的必要性

性能：在基础模型之上通过合成专项数据进行后训练，可以让模型在特定提示词下表现更加稳定，并提升基础模型专项性能。

成本：更小参数模型更强性能的专用模型必然带来成本和速度优势。

智能体大模型

支持长上下文下复杂长程任务的运行

记忆类模型

在记忆提取、记忆融合、记忆评估等方面更优

检索类模型

针对复杂代码场景具备更高的召回率，成本更低

Wiki生成模型

具备复杂代码理解和wiki生成能力，速度更快

工具专用模型

代码应用、会话总结等专用模型，带来更优上下文能力

补全 / NES 模型

模拟用户代码编辑轨迹专项训练，兼顾准确率与速度

Qoder

Think Deeper. Build Better.

官网：https://qoder.com/
Twitter：https://x.com/qoder_ai_ide

QCon
全球软件开发大会

InfoQ 极客传媒

THANKS

大模型正在重新定义软件

Large Language Model Is Redefining The Software