QCon
全球软件开发大会

InfoQ 极客传媒

AI Coding 实践：PRD 到代码直出的探索

王光景

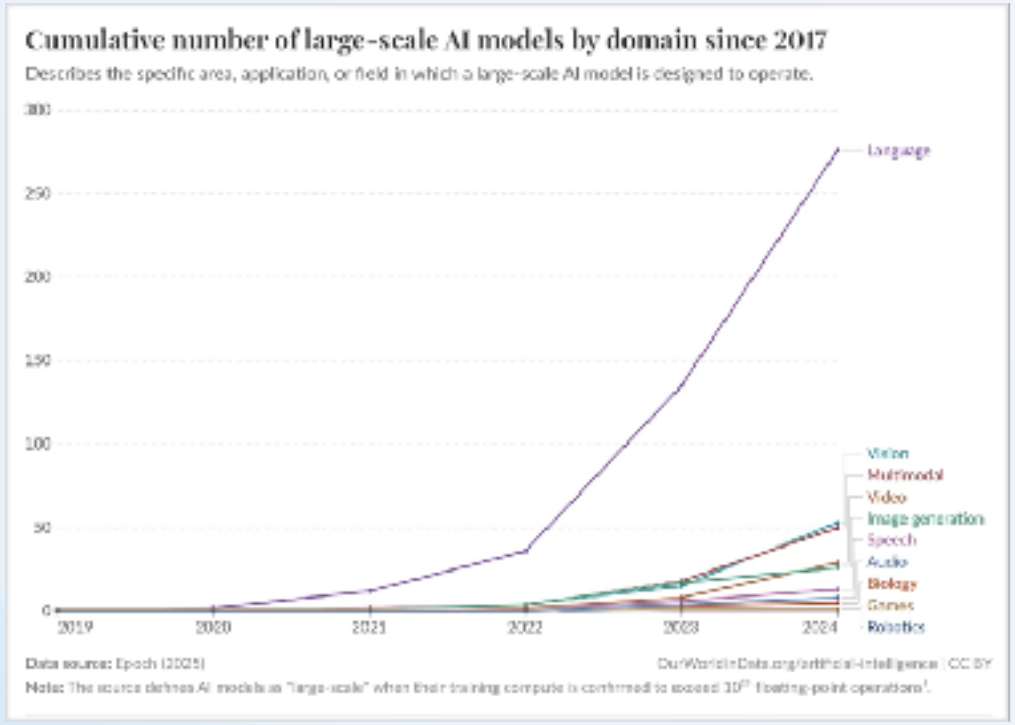
小红书社交客户端负责人

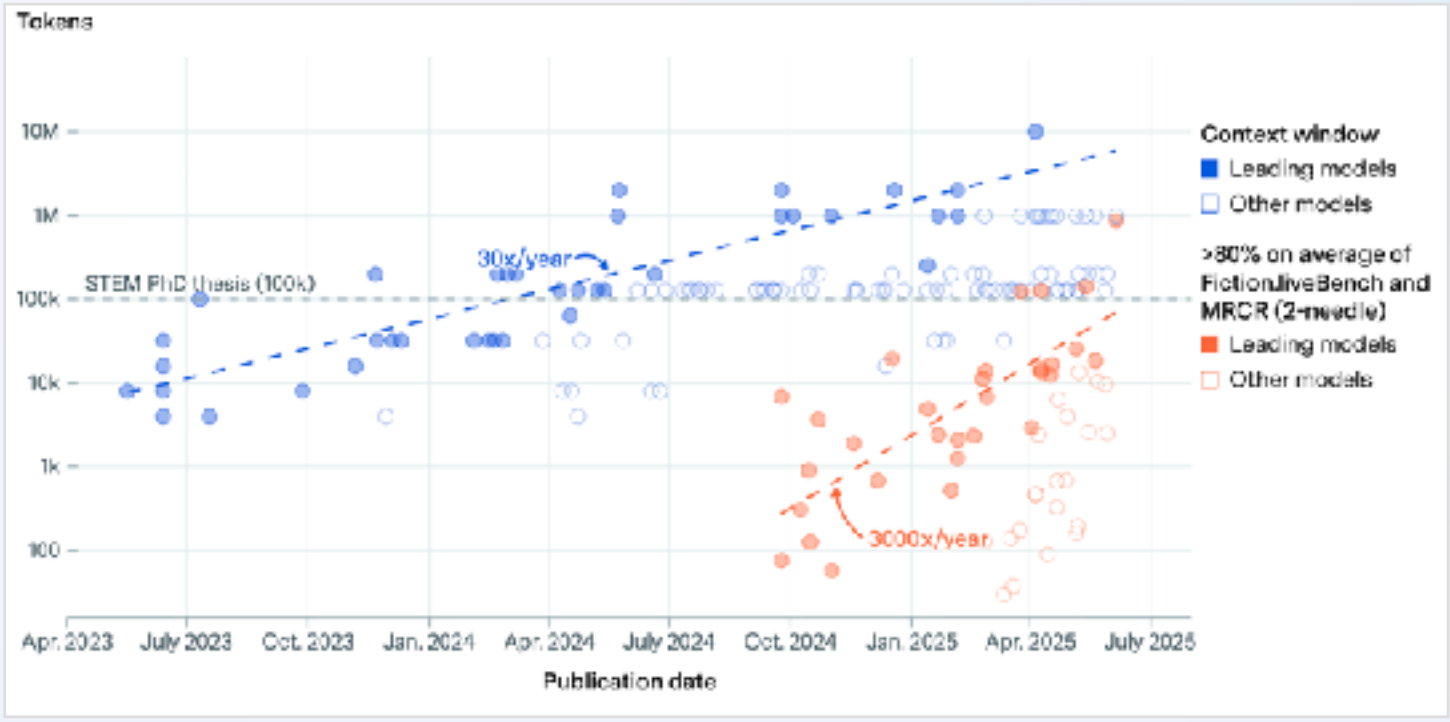
目录

01

AI Coding 发展史和现状

AI模型的发展





– 自2017 年 Google 提出Transformer 架构后，AI领域产生革命性突破，应用场景不断拓展，逐渐渗透到各个领域。

– 自2023 年开始，LLM 的最长上下文窗口每年增长 30 倍

AI Coding 发展历程

· Ref：https://paradite.github.io/ai-coding/

基于人机协作的视角，结合交互形态、人工介入与自主程度，可将 Code Agent 划分为 L1—L5 五个阶段。

L1： 人类主导，Agent实时辅助：代码提示

L2：人类布置任务，Agent生成代码：单一task 实现

L3：人类设定范围，Agent 推进多环节流程：生成方案、代码

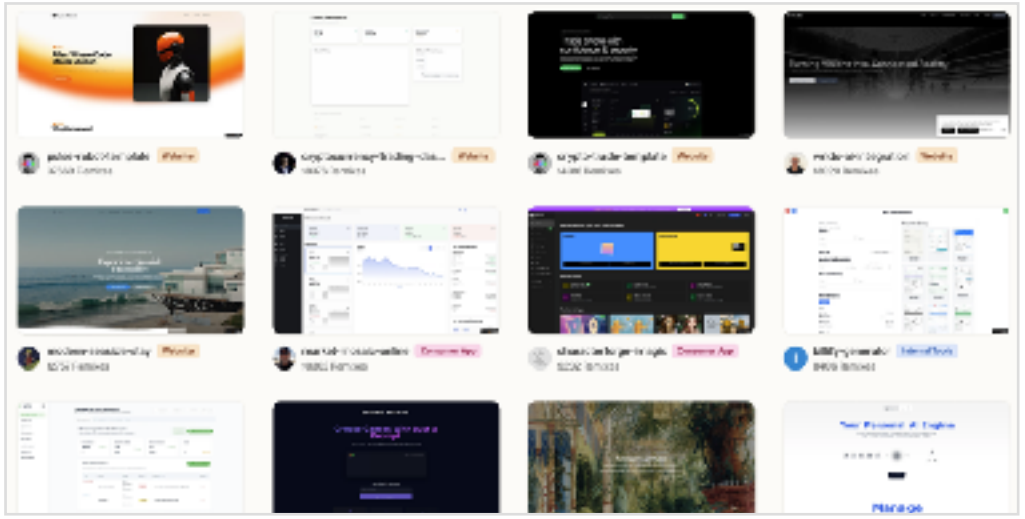
L4：人类输入PRD，Agent 端到端交付：完成需求解析、架构设计、编码

L5：人定义目标，多Agent分工协作：Agent 系统模拟完整软件团队，通过多 Agent 分工协作完成全流程开发

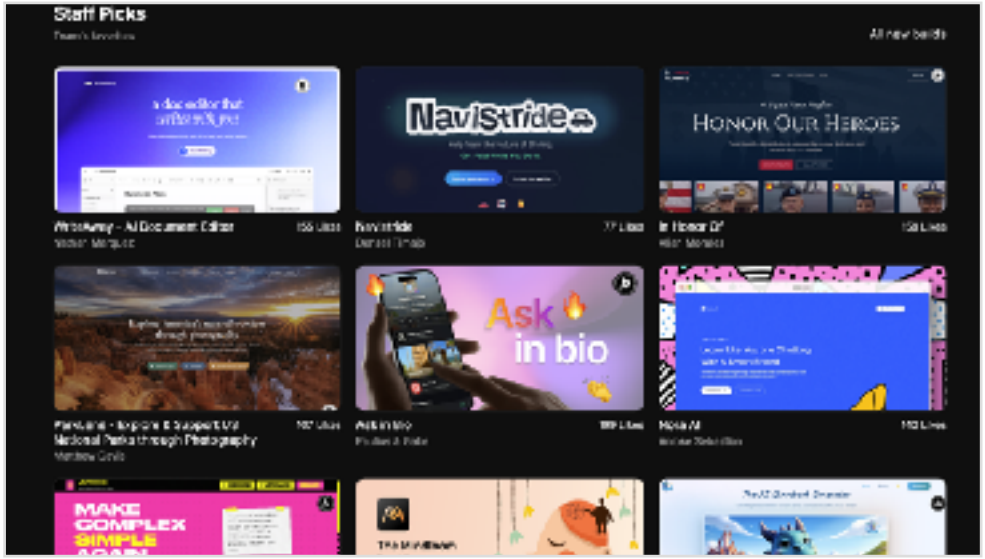
需求到交付——直出产品

端到端“直出”型 AI coding 产品：用自然语言直接产出可运行、可访问的应用，一步完成需求到交付，但是只出现在前端

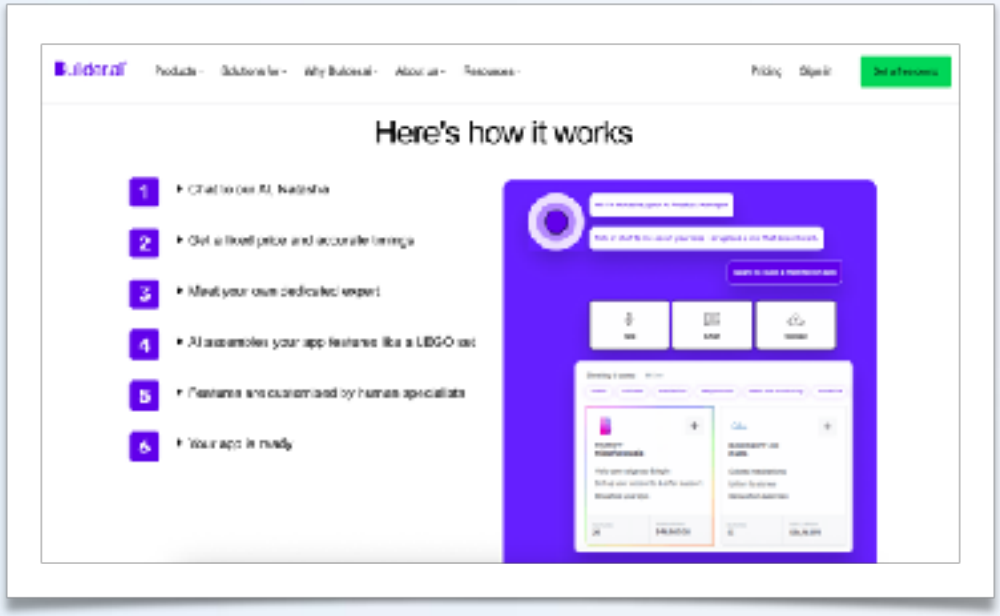
Lovable



Bolt.new



需求到交付——直出产品（客户端领域）



客户端领域的直出产品，但是很不幸暴雷了

客户端 AI Coding 的困境 – 技术栈视角

	前端	客户端
平台的标准化 vs 碎片化	1、标准化程度高：HTML/CSS/JavaScript 是统一标准 2、框架相对集中：React/Vue/Angular 占据主流	1、平台碎片化：iOS 16 的 API 和 iOS 17 的新 API用法不同 2、框架分散：SwiftUI、UIKit、Jetpack Compose、Flutter、React Native
构建与调试复杂度	构建工具成熟，调试简单	编译时间长，真机调试必要
开发模式差异	即时反馈：热重载、实时预览	1、生命周期复杂 2、依赖注入和架构：MVVM、VIPER 等模式多样
UI 构建的抽象	本质都是一种声明式UI 为主	UI 层面 SwiftUI、UIKit 和 Jetpack Compose 、Android View 同时存在，声明式和命令式混合

客户端 AI Coding 的困境 – AI 模型视角

	前端	客户端
训练数据数量和质量	拥有规模大、质量高的公开训练数据。语言使用率：JS 62.3% ，项目数量：前端 27.5%，	高质量的训练数据要稀缺得多，大量优质客户端代码在企业内部，未开源。语言使用率：Swift+Kotlin 15%。项目数量： 客户端 < 5%
反馈和验证循环	验证 AI 生成的代码非常快速。这个快速的反馈循环，使得开发者可以快速地与 AI 迭代，修正错误。	验证成本高得多，这个缓慢的反馈循环，大大降低了使用 AI 辅助编程的迭代效率。
代码模式识别难度	模式高度标准化：React Hooks 模式、Vue 组合式 API 模式	Android： 传统 Activity -> MVP -> MVVM + LiveData -> Jetpack Compose + State
上下文窗口限制	一个完整的功能通常在一个文件内，调用链浅	实现同样的功能需要理解多个文件，调用链深

客户端 AI Coding 的困境

前端开发就像是在一个**标准化的、开放的、乐高式**的环境中工作，任务明确，材料（训练数据）充足，非常适合 AI 大展拳脚。

客户端开发则像是在一个**碎片化的、半封闭的、需要与复杂系统深度交互**的环境中进行精密工程，AI 在这个领域能提供帮助，但要做到像前端那样流畅和智能，还有很长的路要走。

02

客户端 AI Coding 关键解法

客户端这么难，提效我们需要做什么？

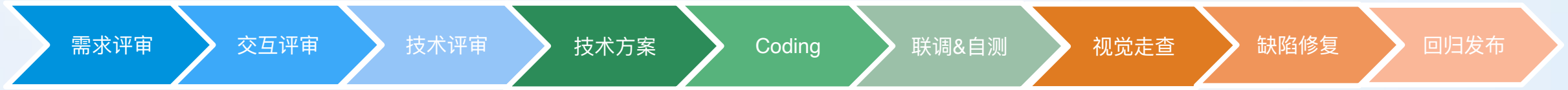
模型能力边界是什么？能否直出交付？

模型快速迭代，如何避免“无用功”？

落到真实场景的提效，如何切入？

AI 如何更长期的融入企业迭代？

客户端这么难，提效我们需要做什么？



模型快速迭代，如何避免“无用功”?

- #1 科学评测体系
- .....

落到真实场景的提效，如何切入?

- #2 需求 PRD 拆解
- #3 UI 高还原度出码
- .....

AI 如何更长期的融入企业迭代?

- #4 组件召回闭环迭代
- .....

02

AI Coding 端侧关键解法

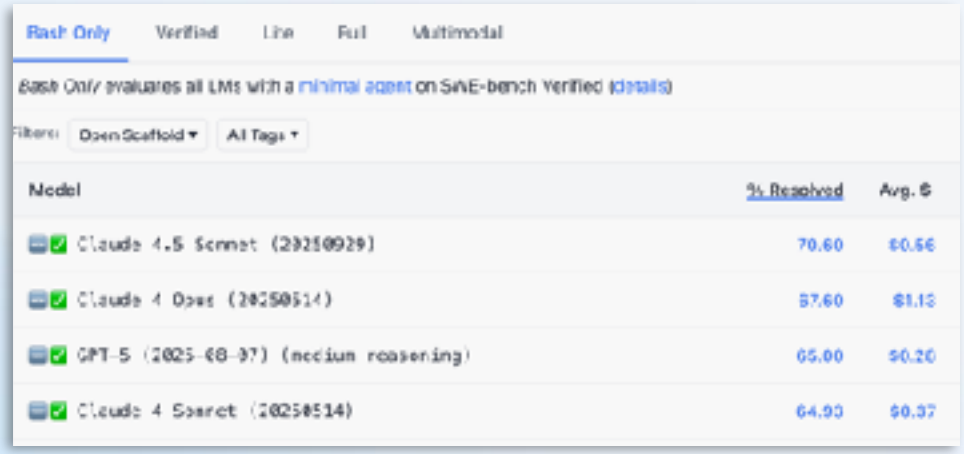
#1 Mobile-swe-benchmark

关键点一：如何科学评测AI代码生成效果

SWE-bench (Software Engineering Benchmark)

https://www.swebench.com/original.html

SWE-bench 是由普林斯顿大学和芝加哥大学于 2023 年推出的首个大规模软件工程基准测试，通过从 12 个真实 Python 开源项目中收集 2,294 个真实的 GitHub Issues 和对应的修复方案，用于评估大型语言模型解决实际软件问题的能力



关键点一：如何科学评测AI代码生成效果

为什么SWE -bench成了代码能力评测的事实基准？

数据源

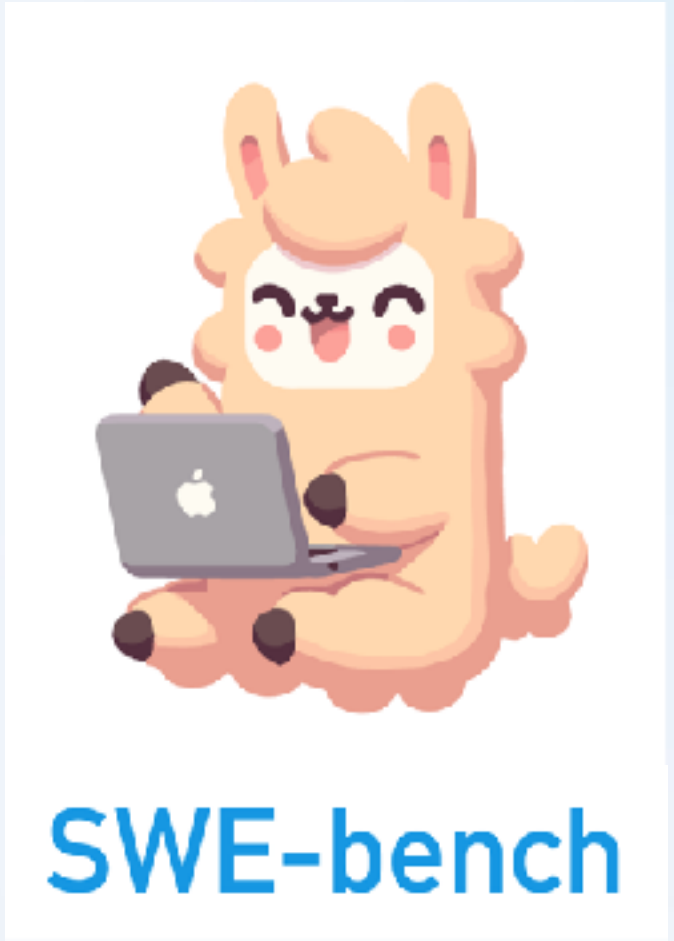
基于真实的 GitHub 开源项目（如 Django, Pytest），从数千个真实的 Issue (问题/Bug) 中提取需求描述，要求 AI Agent 自动生成一个 Pull Request (代码补丁) 来解决这个 Issue。

评测方法

通过原项目自带的单元测试（Unit Test）。如果 AI 生成的代码能让测试用例通过，就被认为是成功解决了问题。

Why it's popular?

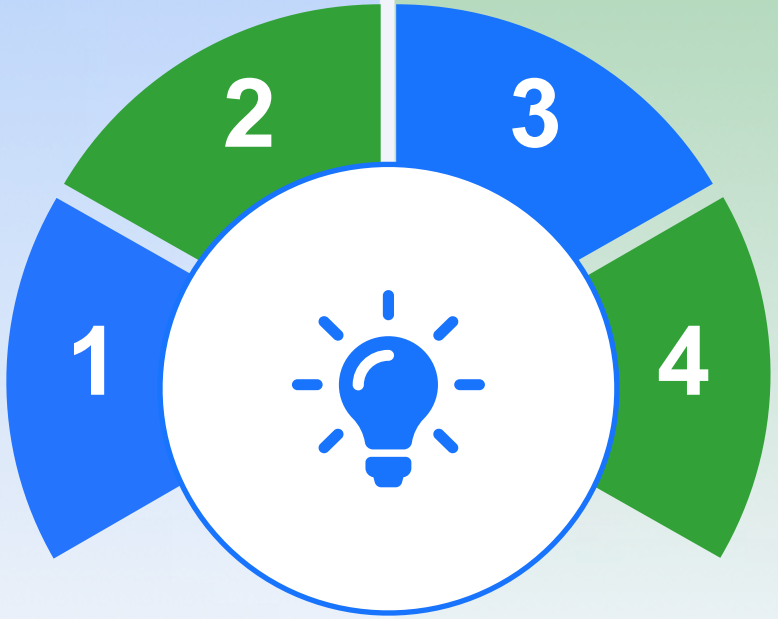
精准切中了现代软件工程的核心痛点——如何在复杂、动态的代码库中高效解决真实问题。现在每个模型的发布、迭代都会默认跑一下分。



如何科学评测AI代码生成效果 —— SWE-bench 局限性

Bugs not features

Github issue本身更多是bug描述，因此swe-bench评测的实际上是模型bugfix的能力，而不是端到端实现feature的能力，也不能真正的反应生产环境问题。



Multimodel eval is not proper

移动端开发领域会涉及大量的图形界面。虽然swe-bench后面也引入了swe-bench-multimodel，但是其pixel-by-pixel的比对方式也不太科学。

Scenario Coverage

swe-bench选中的项目多集中在后端，产品形态上不直接面向C端用户，项目的编程语言也相对局限。

Mobile-specific problems

移动端还有很多领域特有的隐含编码任务，如：屏幕适配、系统适配、SOC适配等问题。

移动端评测 —— mobile-swe-bench

更适合移动端研发任务的评测 – 核心要素

1

高质量的真实PRD

真实交付过的需求，描述详细，图文丰富。涵盖了多种业务场景和形态。

2

多模态的输入

除了prd中原本自带的图片以外，引入figma作为重要输入。

3

详细的QA用例

需求研发流程中沉淀的经过专家评审的真实用例。

4

历史Commit 基线

真正跑在线上的代码会作为golden standard。

5

多维度评测方法

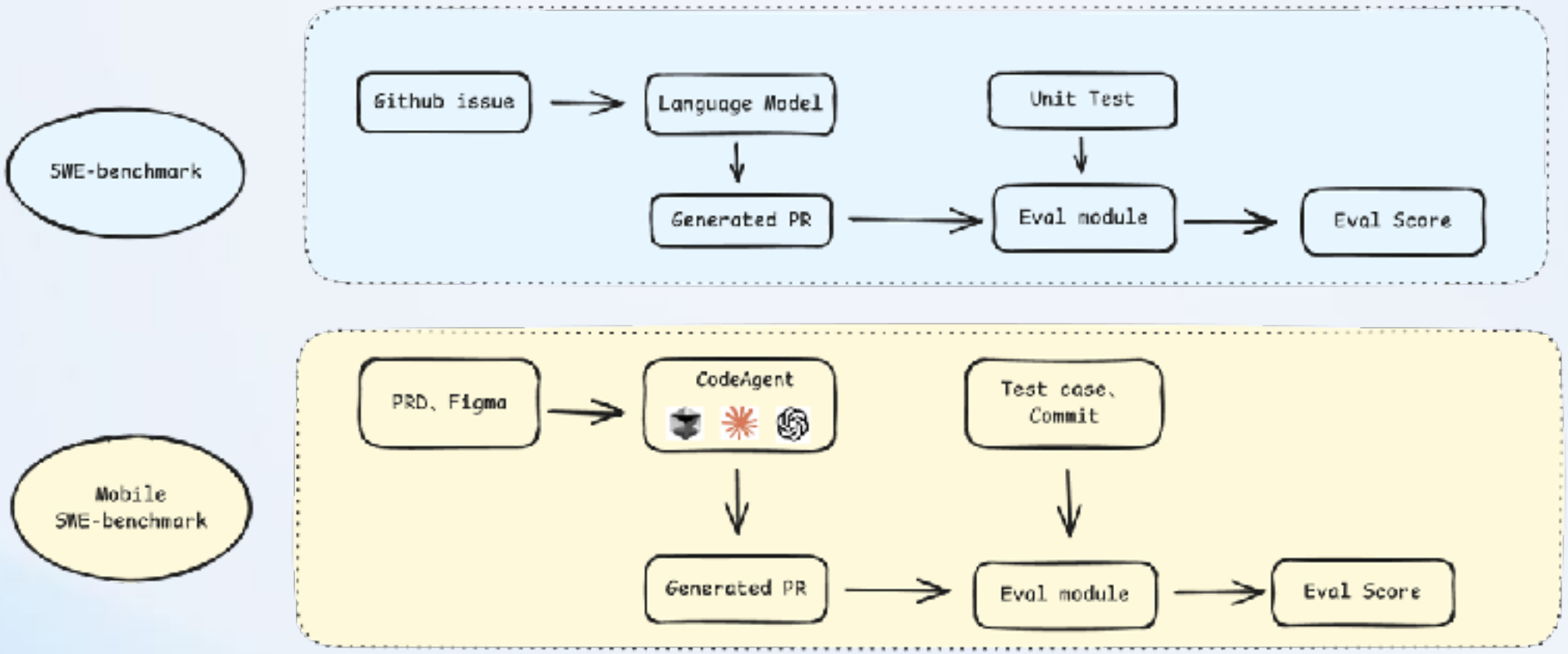
覆盖人工评测、自动化测试、移动端UI 渲染评测等

Bench核心 —— 如何设计Eval方法

平衡准确性与效率

- Human Eval
  - 人工评测：developer通过两份diff patch来判别打分、QA通过执行手工用例来判别。
    准确性较高，但效率较低，scale难度大
- Unit Test + UI Test
  - 客观、可重复、可自动化，可以快速反馈。
    测试用例质量影响结果，可能出现「高分低能」
- Render-Tree constraint lint
  - 通过静态分析和运行时检查，验证 UI 渲染树是否符合性能和设计规范
- Vision Language Model
  - 利用多模态大模型理解 UI 截图和代码，进行"像人一样"的评估。

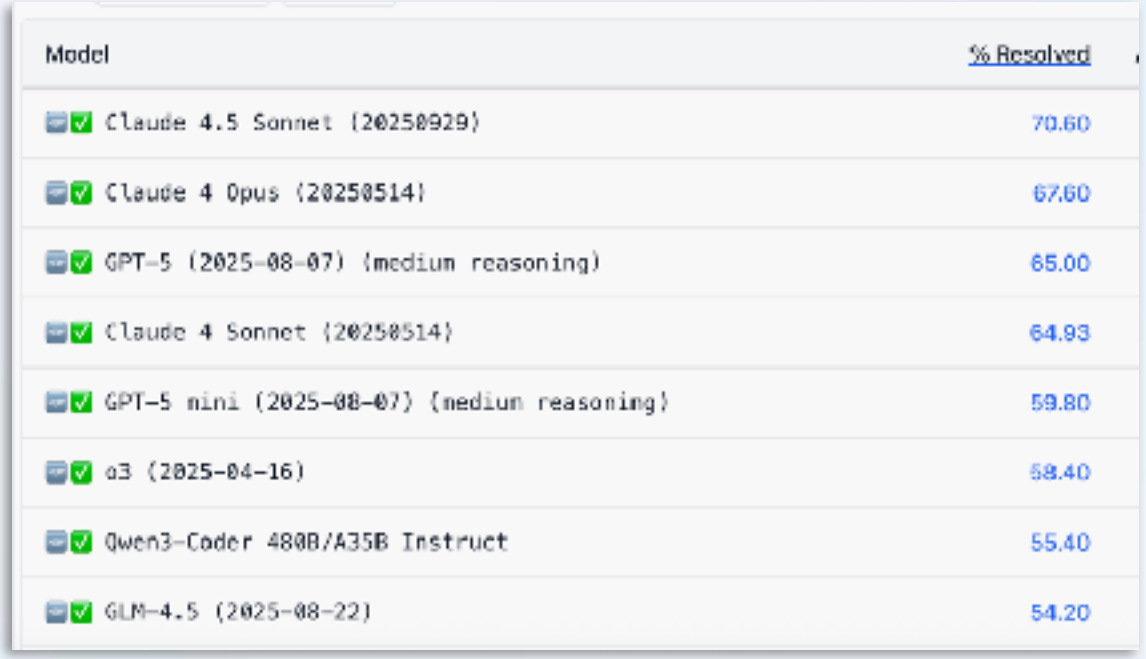
Benchmark 评测流程



热门Coding Agents表现如何

根据实际需求的复杂度，将需求分层 Easy、Medium、Hard 3 种类别的测试集，能够看到即便市面上火热的 Code Agent 在真实生产环境端到端的测试集中，表现也一般。

Agent	Easy	Medium	Hard	All
GPT-5-Codex (GPT-5 High)	38%	35%	26%	33.0%
Claude Code (Sonnet 4.5)	35%	33%	26%	31.0%
Gemini CLI (Gemini 2.5 pro)	-	-	-	
Cursor (Claude Sonnet 4.5)	36%	34%	30%	33.3%



Model	% Resolved
Claude 4.5 Sonnet (20250929)	70.60
Claude 4 Opus (20250514)	67.60
GPT-5 (2025-08-07) (medium reasoning)	65.00
Claude 4 Sonnet (20250514)	64.93
GPT-5 mini (2025-08-07) (medium reasoning)	59.80
o3 (2025-04-16)	58.40
Qwen3-Coder 480B/A35B Instruct	55.40
GLM-4.5 (2025-08-22)	54.20

Where "top-tier coding agents" fail

代码定位错误

找不到需求描述所对应的代码文件，基本就宣告了一次出码任务的完全失败，在大型企业项目(文件数量 > 1w，代码行数 > 100w)中尤其如此；如果需求还涉及了跨模块调用，会使得这个任务的失败率更高。

需求点实现不全/过实现

窗口内的上下文长了之后，很容易顾前不顾后，一些需求的细节点容易在之后遗漏，有些做了一半有些完全没做。

企业内部组件缺乏

自研 UI 组件、网络组件、存储组件、数据打点、ab test等每个需求里经常出现的组件没法被正确使用。

UI 还原程度低

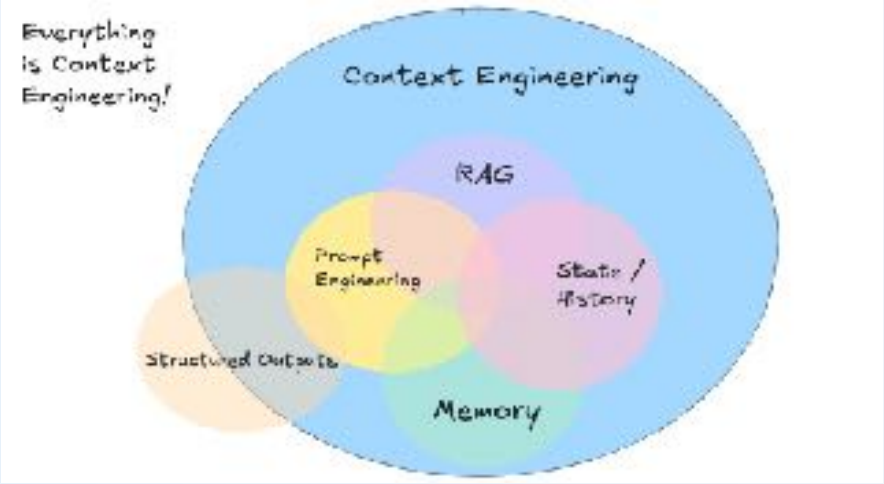
大体上看起来是那么回事，但是完全过不了设计师的还原走查。这里其实就是prototyping -> production的区别。

02

AI Coding 端侧关键解法

#2 PRD 微调拆解

低质量出码原因之一 —— 上下文



Context Engineering是一门系统性学科，专注于设计、构建并维护一个动态系统，该系统负责在 Agent 执行任务的每一步，为其智能地**组装出最优的上下文组合**，以确保任务能够被可靠、高效地完成

代码定位错误　企业组件缺乏　需求实现不全　UI还原程度低

技术规范层上下文
企业组件库文档、技术栈规范约束等

业务领域层上下文
业务概念，业务逻辑、历史业务参考

设计还原层上下文
design token、布局规范、响应式要求

代码库上下文
项目架构、依赖关系、数据状态管理

核心上下文 – PRD

核心洞察

PRD 是核心的上下文载体，需求迭代的关键输入，完整 PRD ≠ 好上下文

现状是：PRD 过于冗余或信息缺失, 内容不够结构化。这就导致AI 难以聚焦，生成的代码质量低。

理想状态：精准任务： 复杂需求 → 可执行小任务，明确任务边界 - 目标清晰，职责单一

需求理解 —— 精确上下文

NER（Named Entity Recognition）

命名实体识别（Named Entity Recognition，简称 NER）是自然语言处理（NLP）中的一项核心任务，旨在从非结构化文本中识别并分类具有特定意义的实体，如人名、地名、组织机构名、时间、日期等。

需求理解 —— 精确上下文

PRD 怎么拆解 ？ 拆到什么程度?

	NER任务	控件主体拆分
目标	从文本中识别实体	从PRD中识别UI控件实体
分类	PER/LOC/ORG等类型	按钮/输入框/列表等类型
边界	实体的起止位置	控件逻辑的作用范围
关系	实体间的语义关系	控件间的交互关系
上下文	词语的语境依赖	控件的功能上下文

我们需要找到一种方式拆解PRD内容从非结构化的PRD内容中拆分出结构化的逻辑任务。正如通用 NER 从新闻文本中识别“人名、地名、机构名”，客户端 PRD 拆解是从需求文档中识别「UI控件实体」，本质上就是一个领域特定的NER任务

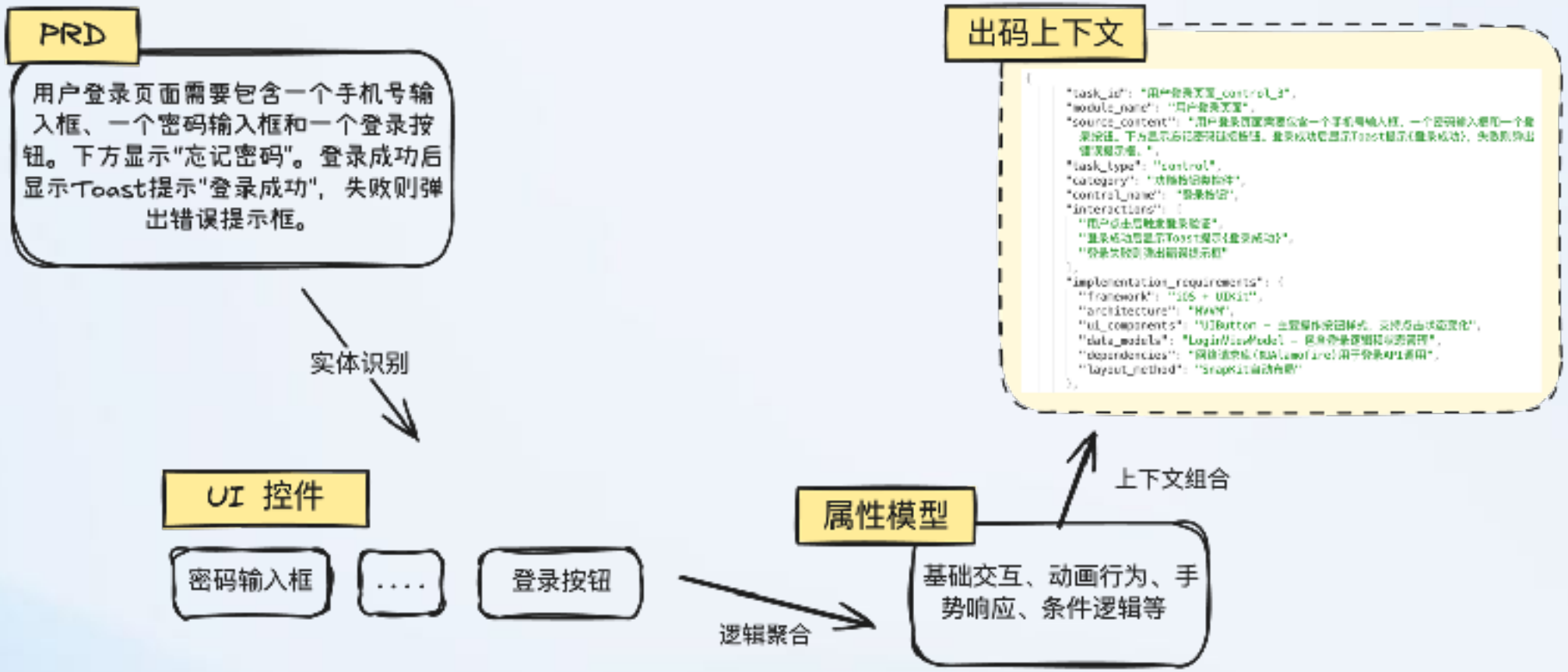
需求理解 —— 控件实体

Ui 控件实体分类

业界设计规范		
设计规范	Apple： Human Interface Guidelines	Google： Material Design System
分类体系	• Controls (控件) • Buttons • Text Input • Selection Controls • Content Views • Modal Interfaces	• Components (组件) • Inputs • Navigation • Containment • Communication
分类原则	• 功能一致性: 同类控件提供相似的功能 • 交互一致性: 同类控件使用相似的交互模式 • 视觉一致性: 同类控件具有相似的视觉特征	• Material Metaphor: 基于物理世界的隐喻进行分类 • Intentional Hierarchy: 建立明确的视觉层次 • Responsive Interaction: 响应式交互设计

客户端控件分类
输入类控件
功能按钮类控件
浮层面板控件
导航栏与页面框架控件
内容展示类控件
列表选择型控件
附加逻辑控制条件
。。。。

需求理解 —— 基于控件实体的出码模型



需求理解 —— 基于控件实体的优势

任务精确性

每个逻辑任务都有明确的UI控件主体
避免"相似控件与逻辑的推理导致出码幻觉"

任务独立性

每个子任务可以独立执行和验证
降低任务间的耦合度

边界清晰性

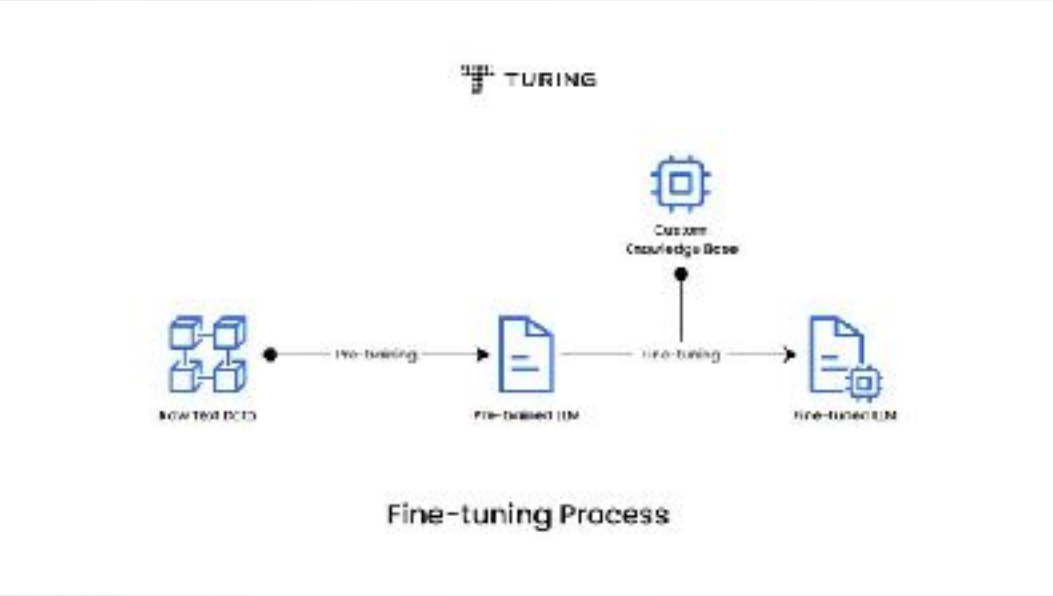
类似NER的精确边界识别
避免逻辑归属模糊导致的重复或遗漏

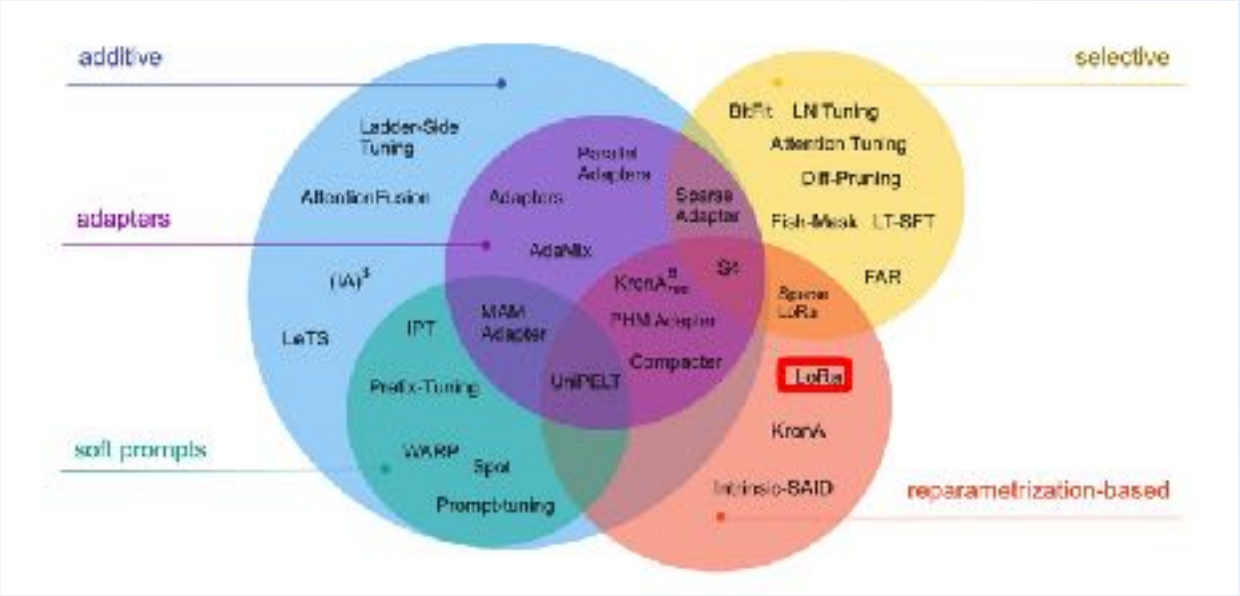
可扩展性

分类体系可根据业务需要扩展
属性模型支持新的交互模式

PRD 拆解 —— 模型微调

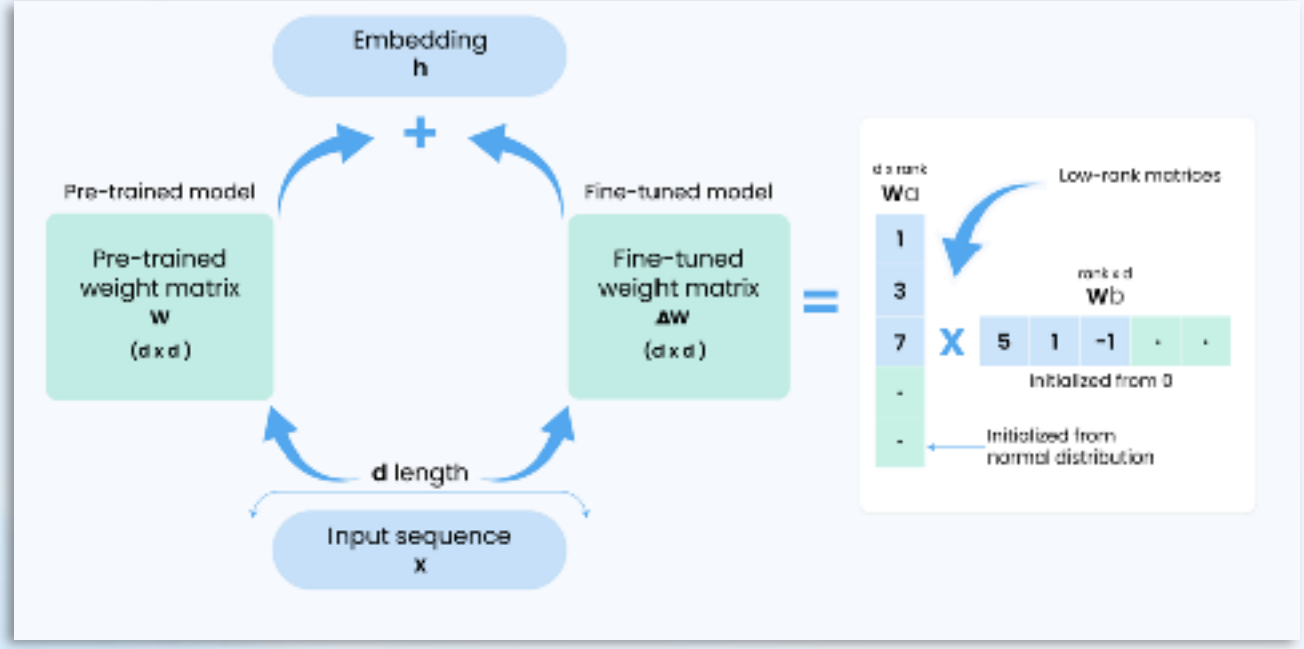
模型微调(Fine-tuning)：在预训练模型上通过在特定任务的数据集上进行额外的训练，这样不仅可以利用预训练模型的泛化能力，还能让模型学会解决特定问题的细微差别，从而在保持模型相对轻量化的同时，显著提升在特定任务上的准确度和效率。





* Prompt-tuning： 通过修改输入文本的提示来引导模型生成符合特定任务的输出，无需更新模型参数
* Prefix-tuning：在每一层的输入前添加可训练的“前缀向量”，只优化这些前缀参数。
* LoRA 微调：通过在调整过程中改变模型参数的代表性子集（称为低秩适配器 ）的权重，而不是基础模型的权重，使基础模型适应某项任务

PRD 拆解 —— LoRA 微调



LoRA 微调：通过在调整过程中改变模型参数的代表性子集（称为低秩适配器 ）的权重，而不是基础模型的权重，使基础模型适应某项任务

维度	全量微调	**LoRA 微调**
可训练参数	100%	0.1%-1%
显存需求	高（如 80GB）	低（如 16GB）
训练时间	长	短（提速 3-5 倍）
存储空间	完整模型副本	仅 LoRA 权重
部署灵活性	低	高

微调结果对比

评估控件归类是否精确

- 准确率：是否误检
- 召回率：是否漏检
- F1分数：两者的调和平均数

纯文本Baseline

- 准确率：0.51
- 召回率：0.69
- F1分数：0.57

微调后的纯文本模型

- 准确率：0.83
- 召回率：0.72
- F1分数：0.74

微调后的多模态模型（无图评测）

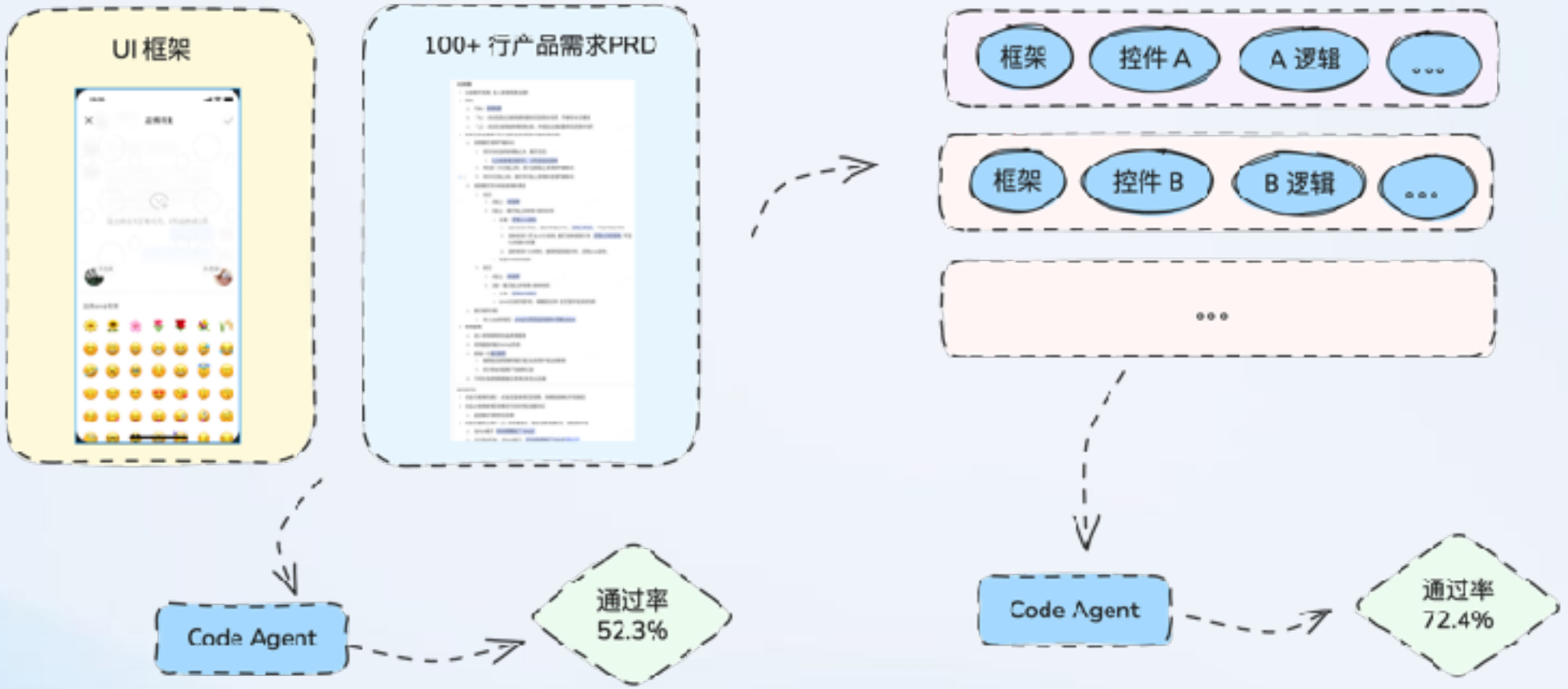
- 准确率：0.81
- 召回率：0.73
- F1分数：0.75

微调后的多模态模型（带图评测）

- 准确率：0.88
- 召回率：0.87
- F1分数：0.85

*数据来源于每份测试PRD的指标平均值

PRD 拆解后的效果


UI 框架
100+ 行产品需求PRD
Code Agent
通过率
52.3%
框架
控件 A
A 逻辑
...
框架
控件 B
B 逻辑
...
...
Code Agent
通过率
72.4%

02

AI Coding 端侧关键解法

#3 UI 高还原度出码

低精度 UI 出码原因

Figma设计稿不规范

- 设计稿使用绝对布局
- 组件标记不规范
- 设计稿中存在多余、透明元素

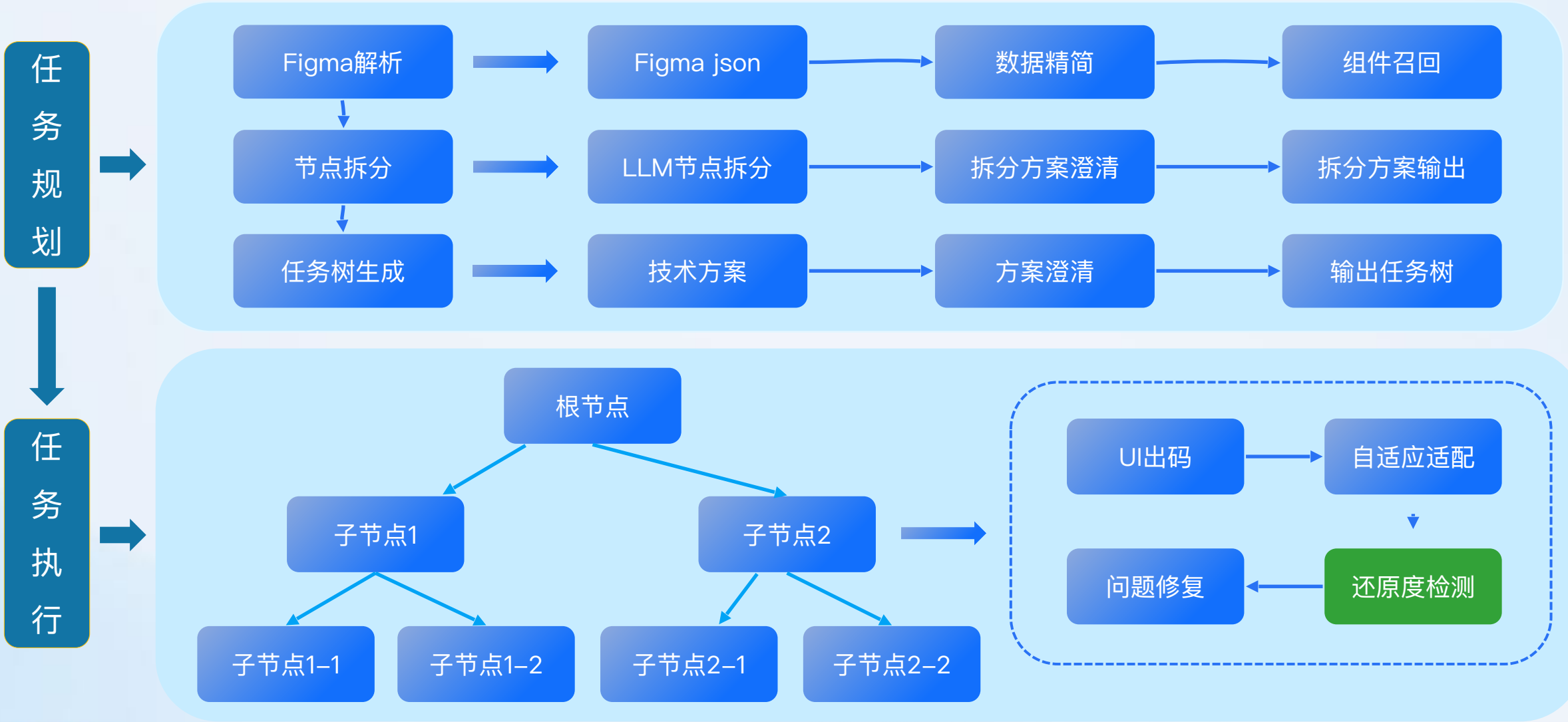
大模型存在幻觉

- Figma设计稿布局复杂，存在推理瓶颈
- 任务复杂，上下文遗忘
- 多生成、漏生成view

还原度低

- 相比设计稿还原度低
- 人工代码审核成本高

还原度检测流程


任务规划
Figma解析
Figma json
数据精简
组件召回
节点拆分
LLM节点拆分
拆分方案澄清
拆分方案输出
任务树生成
技术方案
方案澄清
输出任务树
任务执行
根节点
子节点1
子节点2
子节点1–1
子节点1–2
子节点2–1
子节点2–2
UI出码
自适应适配
还原度检测
问题修复

还原检测方案和挑战

还原度检测挑战

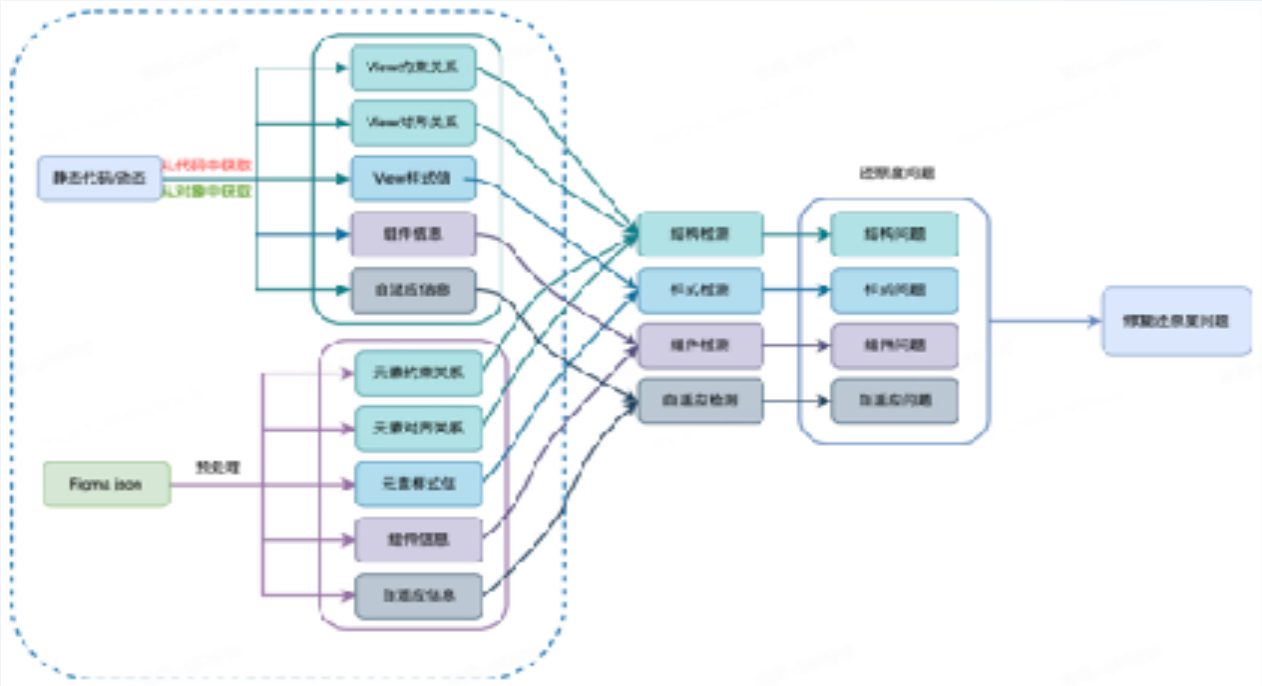
1. 客户端编译慢：检测→问题修复链路不畅

2. 还原度结构检测困难

➢ Figma json中无任何约束关系和对齐关系，结构检测难度大

➢ 出码后的UI存在层级拍平，无法直接对比Figma json 树和运行时的View渲染树

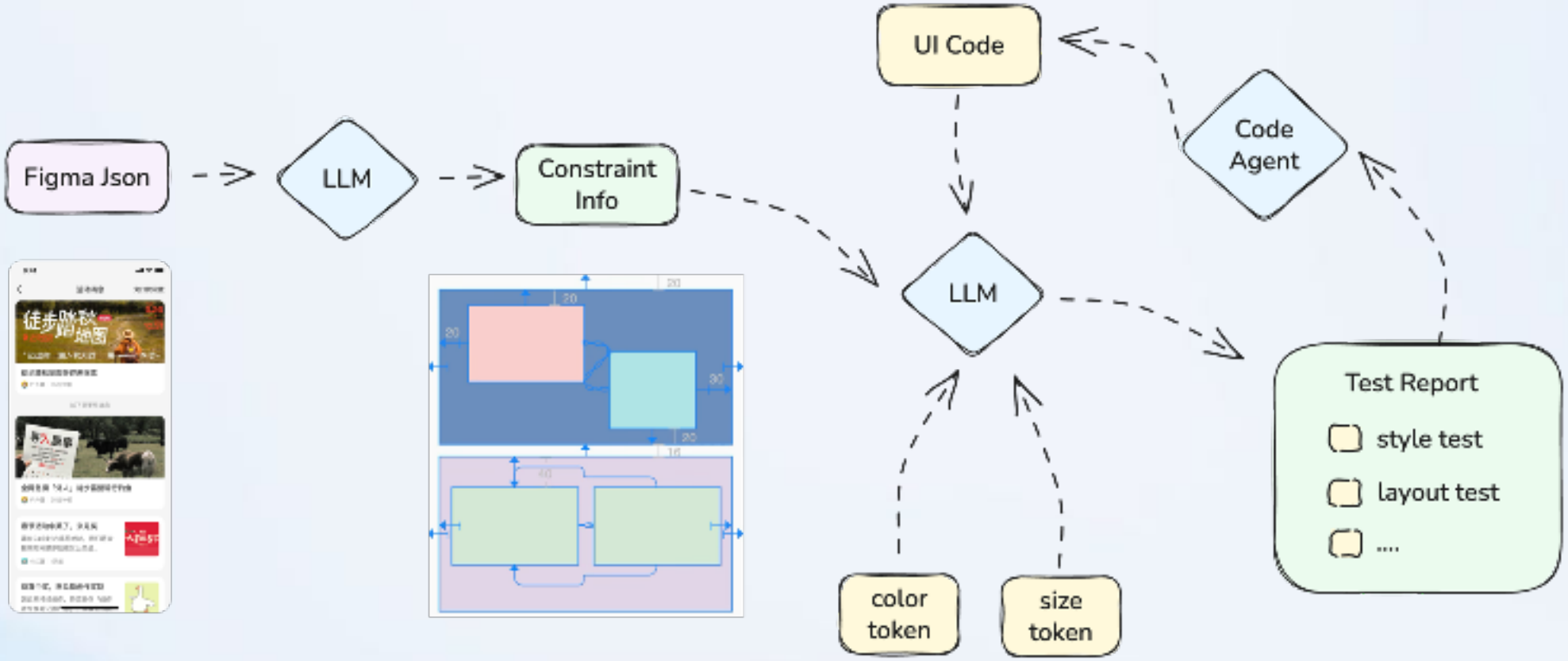
还原度检测方案



还原度检测方案对比(静态代码VS运行时)

	静态代码	运行时
检测输入	• 静态代码 • 原始Figma json	• 运行时View渲染树 • 原始Figma json
结构检测	• 从静态代码中提取view的位置信息和约束对齐信息 • 对比静态代码View的宽高和Figma json中的宽高 • 对比静态代码的约束对齐关系和Figma元素间的约束对齐关系	• 运行时获取view的位置信息和约束对齐信息 • 对比运行时View渲染宽高和Figma json中的宽高 • 对比运行时View渲染树的约束对齐关系和Figma元素间的约束对齐关系
样式检测	• 对比静态代码中View的颜色值、圆角、字体等和Figma json中元素的颜色值是否一致	• 对比运行时View渲染的颜色值、圆角、字体等和Figma json中元素的颜色token是否一致
检测效率	无需编译和运行	需要编译和运行，一次完整的还原度检测需要 N 分钟
链路集成	可以丝滑的集成到AI 出码环节中	因为存在编译和运行环节，无法很好的集成到AI出码环节
检测效果	检测率：88.5%	检测率：55.4%

静态还原度检测方案



通过大模型推理约束关系来确定View布局，当静态代码约束正确时，运行时页面即可准确还原UI结构。基于静态约束关系的还原度检测准确率可以达到 88%+的水平

02

AI Coding 端侧关键解法

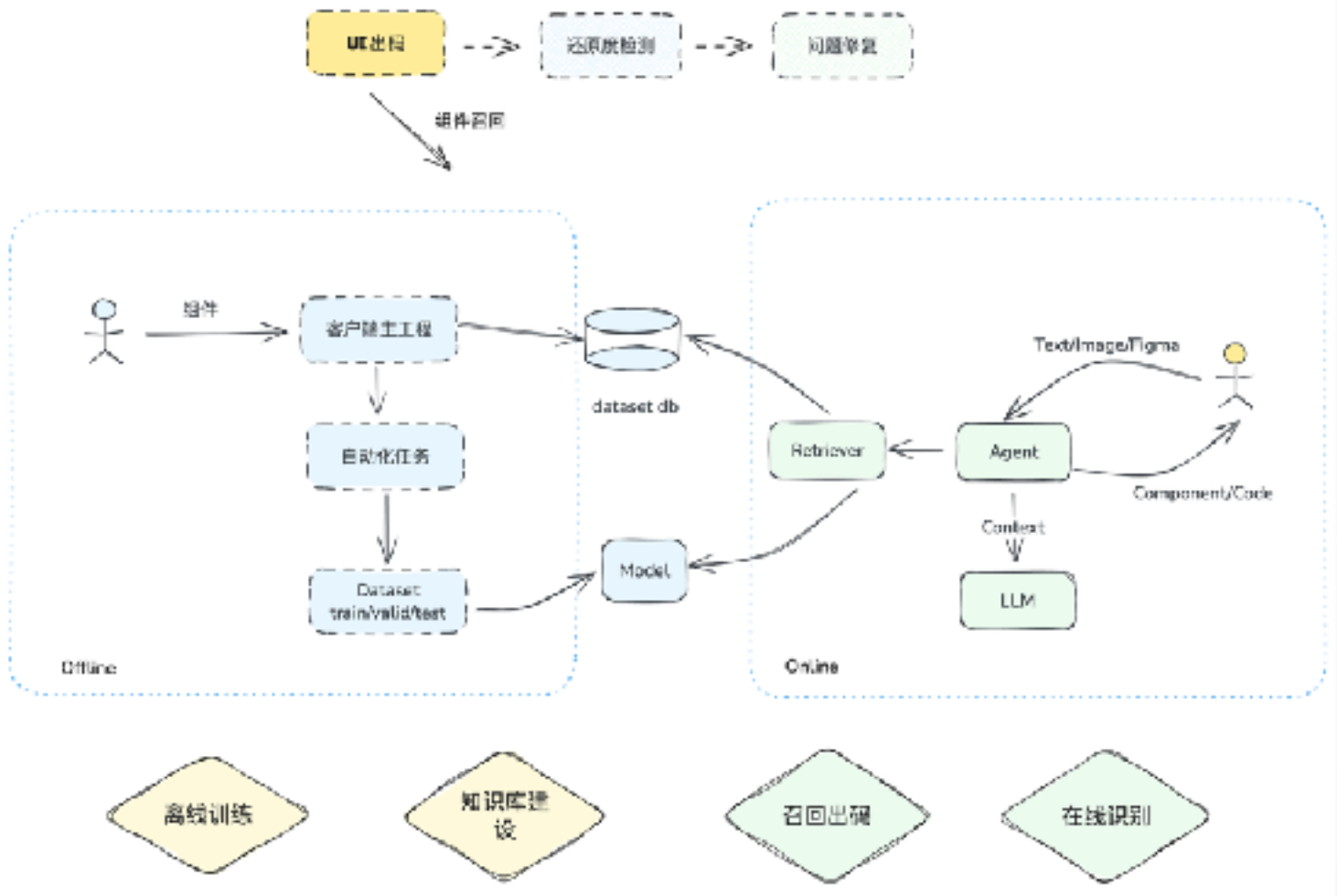
#4 UI 组件召回

UI 组件召回

问题：功能实现 OK，但是代码采纳率低。

解法：基于代码上下文和开发意图，智能召回组件库中的最佳匹配组件，避免重复造轮子，降低长期维护成本。

UI 组件召回 —— 自我进化迭代



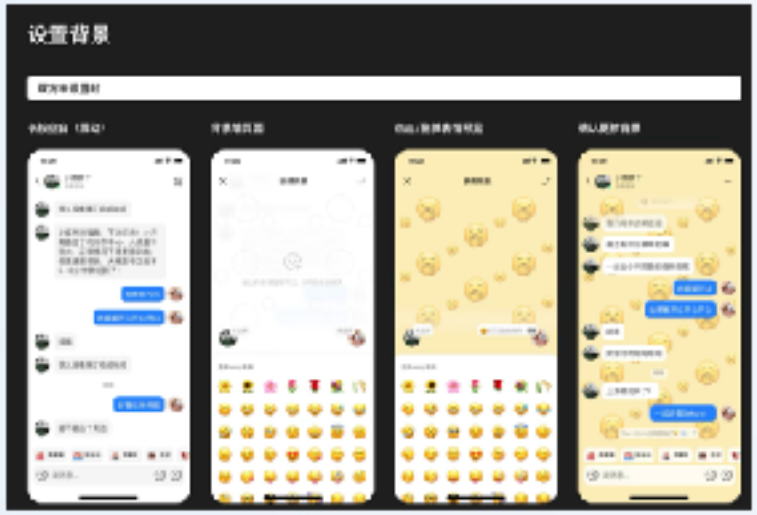
如何能够不断自主收集数据、不断的自我进化迭代?

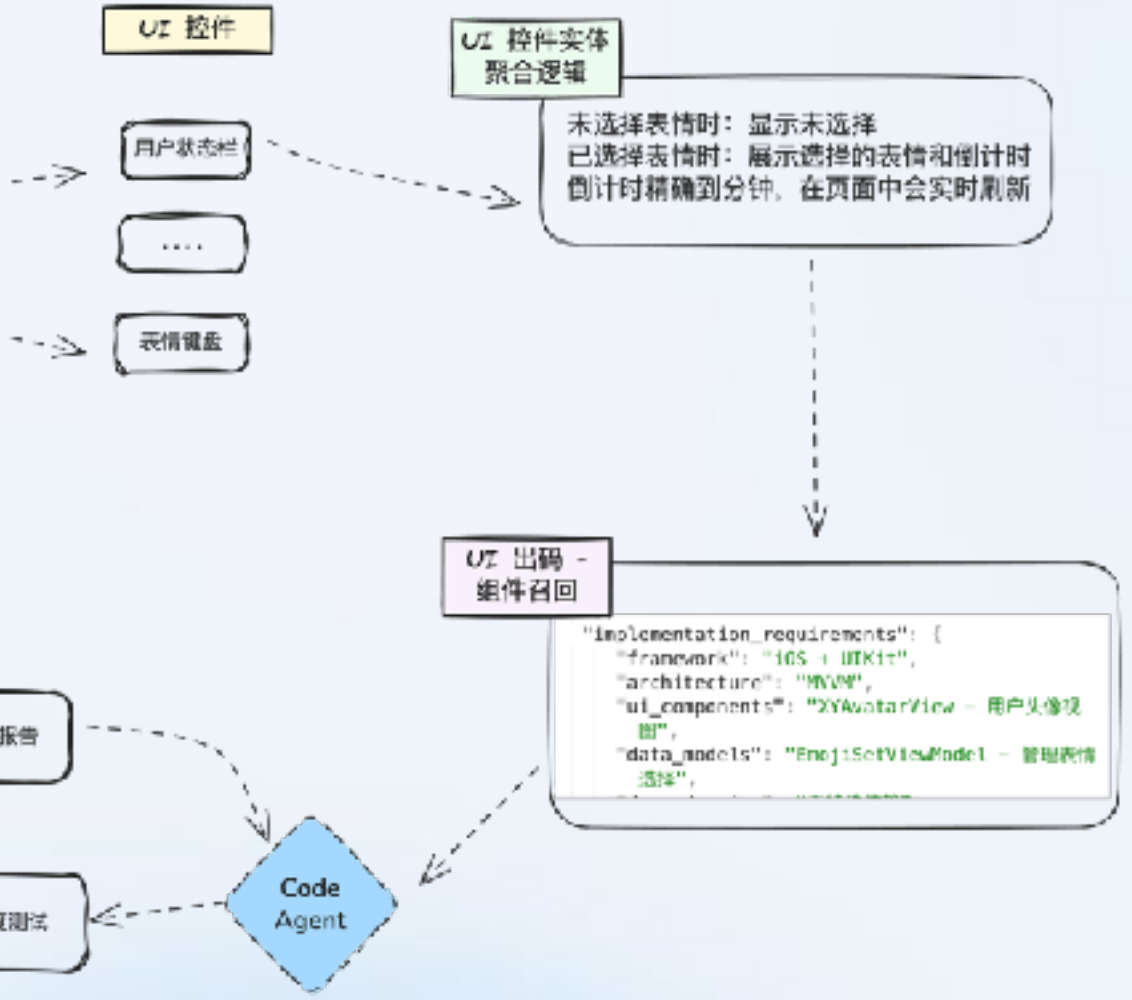
组件召回依赖人工标注和训练，每次新组件上线都需重复此流程，缺乏自学习能力。通过结合UI自动化，可批量采集组件的实际使用截图和运行时属性，动态生成高质量数据集。

03

典型业务场景与需求分析

典型业务场景与需求分析





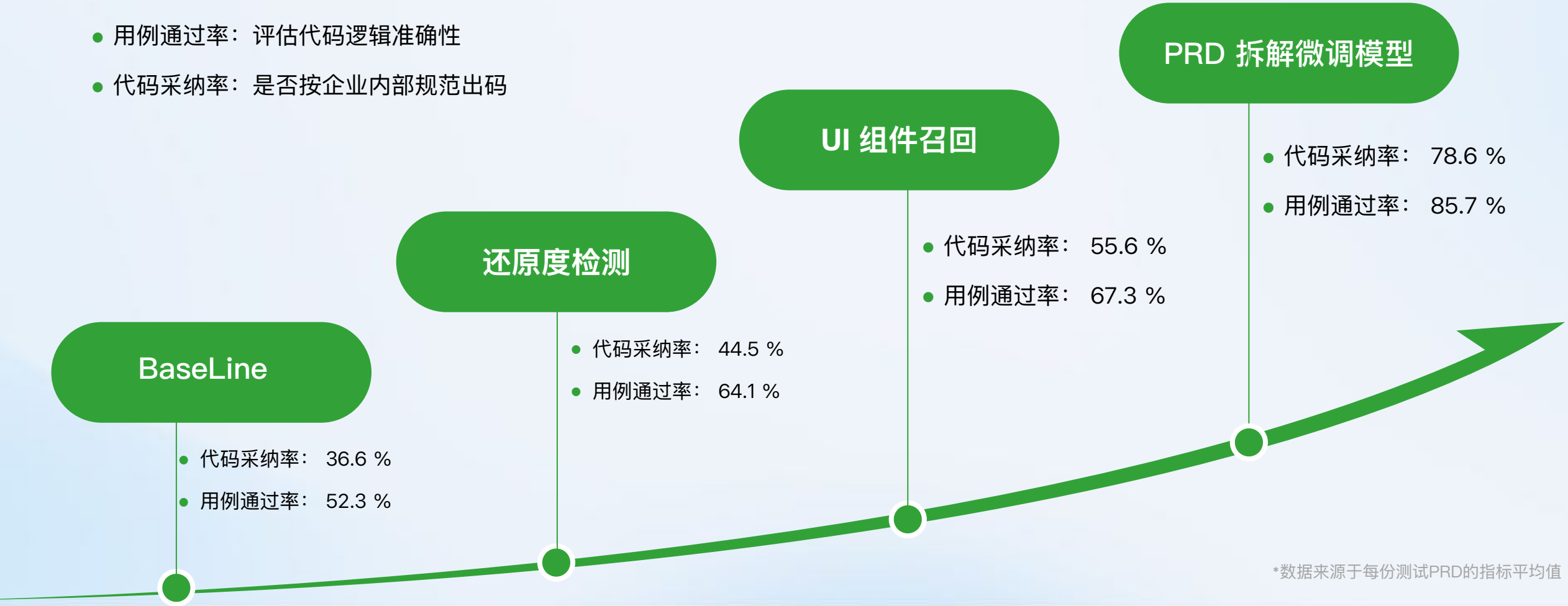
典型业务场景与需求分析

```
{
  "task_id": "贴表情交互_control_2",
  "module_name": "贴表情交互",
  "source_content": "点击与拖拽表情到背景的交互逻辑，包括预览、生效确认、提示等",
  "task_type": "control",
  "category": "功能按钮类控件",
  "control_name": "确认按钮「√」",
  "interactions": [
    "点击页面右上角的「√」后设置生效，退出当前设置状态，返回到对话",
    "进入页面无操作时「√」disable有操作(包括操作移除 or 点选任意表情)「√」able可点击"
  ],
  "implementation_requirements": {
    "framework": "iOS + UIKit",
    "architecture": "MVVM",
    "ui_components": "UIBarButtonItem, UIButton",
    "data_models": "ConfirmationModel, StateTrackingModel",
    "dependencies": "SnapKit",
    "layout_method": "SnapKit自动布局"
  },
  "acceptance_criteria": [
    "功能实现完整性 - 所有interactions中的交互逻辑都要实现",
    "UI还原度 - 符合iOS设计规范和用户体验",
    "代码质量 - 遵循MVVM架构，含完整注释",
    "兼容性 - 适配不同屏幕尺寸的iPhone机型"
  ]
},
```

典型业务场景与需求分析

评估需求出码质量

- 用例通过率：评估代码逻辑准确性
- 代码采纳率：是否按企业内部规范出码

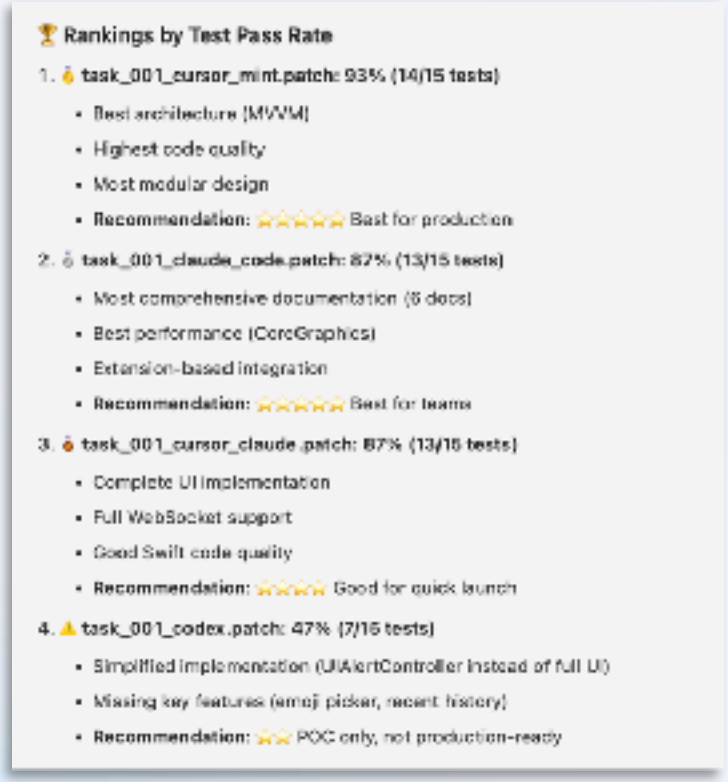


*数据来源于每份测试PRD的指标平均值

典型业务场景与需求分析

根据实际需求的复杂度，将需求分层 Easy、Medium、Hard 3 种类别的测试集，能够看到适配完的 CodeAgent 有不错的表现，整体端到端提升 10% 左右，在某些垂类测评中效果显著。

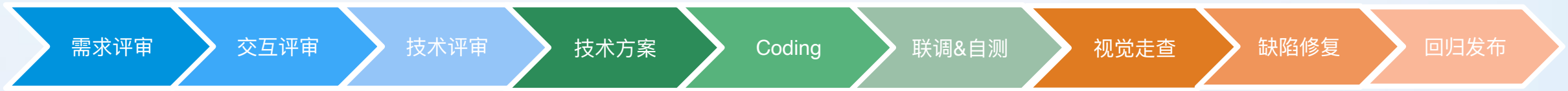
Agent	Easy	Medium	Hard
GPT-5-Codex (GPT-5 High)	**38%**	**35%**	**26%**
Claude Code (Sonnet 4.5)	**35%**	**33%**	**26%**
Cursor (Claude Sonnet 4.5)	**36%**	**34%**	**30%**
Custom Code Agent	**40%**	**38%**	**34%**



04

总结与展望

总结与展望



Topic 1：如何构建科学的端到端评测体系，长期指导和度量 Agent 动作?

Topic 2：面对需求理解这个环节，PRD 如何拆解，拆解到什么粒度?

Topic 3：如何保证客户端UI 的高还原度出码?

Topic 4：面对长期 UI 组件的新增和迭代，如何更高效的做组件识别召回?

客户端实现 PRD 到代码的直出，当下是做不到的，不影响我们的发展路径选择：**以评测驱动子 Agent能力提升**。也许后面随着模型能力的飞跃，我们可以无缝切换到代码直出的新范式

总结与展望

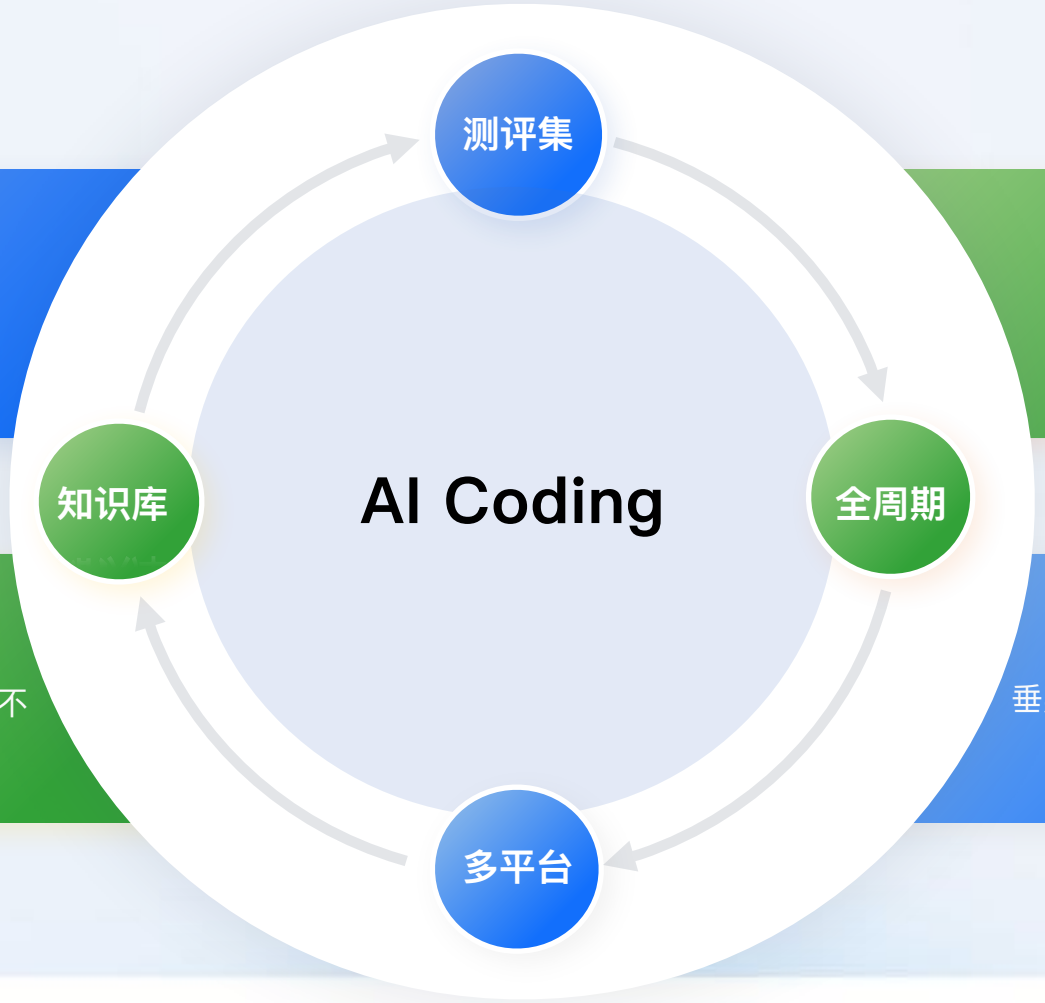
长期的客户端提效怎么走？

生产级别的测评集

包含 PRD、Figma、commit、测试用例等集合的以往真实生产环境的测评集

覆盖需求全生命周期

贯穿编码、测试、CR、Bug fix、甚至发布等整个软件开发生命周期



流动闭环的企业知识库

融入企业知识库后，结合企业自动化流程不断自主收集数据、不断的自我进化迭代。

打破平台的垂类 Agent

垂类 Agent 的融合，可以完成更复杂的、跨系统的任务。**实现了“任务的端到端闭环”**

05

Q & A

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