Web AI Agent

渲染容器与动态化方案探索

Exploration of Rendering Containers and Dynamic Architectures for Web-based AI agents

About Me

饶海

Hai Rao

团队：蚂蚁集团 —支付宝体验技术部 —智能体验

高级前端专家

Staff Front-end Engineer

支付宝体验技术部

引领体验科技，驱动数字生活

https://afx-team.github.io

AI /Full Stack /Data Visualization /DevOps

https://github.com/raoHai/

目录

/01

AI Agent 的趋势

AI Agent 元年，AI 应用正沿着几个维度发生跃迁

/01 AI Agent 的趋势

任务场景的跃迁

AI Agent 的任务场景正从最早的补全与对话，发展到信息整合与深度报告生成，最后通过模型上下文协议及工具调用，开始具备与物理世界交互，并执行复杂任务的能力。

自主性的跃迁

AI Agent 的智能化程度同时在增加。从早期 AI 应用严格遵循开发者预设的编码路径，到由工作流驱动，在流程节点上做出有限判断，到能够自主的将高级目标分解成具体任务、规划执行路径、动态调用工具等。

表达和交互形态的跃迁

AI Agent 的交互范式正在逐渐突破「对话框」的限制，给用户带来更原生高效的 AI 体验。AI 从生成文本，逐渐开始编排 UI，最后可以根据用户需求，既时的生成 UI。

用户与 AI Agent 的关系范式转变：从「消费者」到「协作者」

技术演进亦催生了用户与 AI Agent 关系的重塑：
从单向的「消费者」模式转变为双向的「Human-in-the-Loop」协作模式：

/01 AI Agent 的趋势 —给 Web 带来的需求

Web 作为渲染容器

Web 作为 AI Agent 生成内容的基础渲染平台

承载从静态文本到动态交互界面的所有视觉呈现。

Web 作为信息源

互联网绝大部分信息基于 Web 构建。

AI Agent 通过抓取和解析网页内容获取信息，以支持其决策与规划。

Web 作为执行环境

Web 成为 AI Agent 执行任务的场域，涵盖程序化的 API 调用和模拟人类的 GUI 操作。

本次将讨论 Web 作为 AI Agent 的渲染容器面对的挑战。

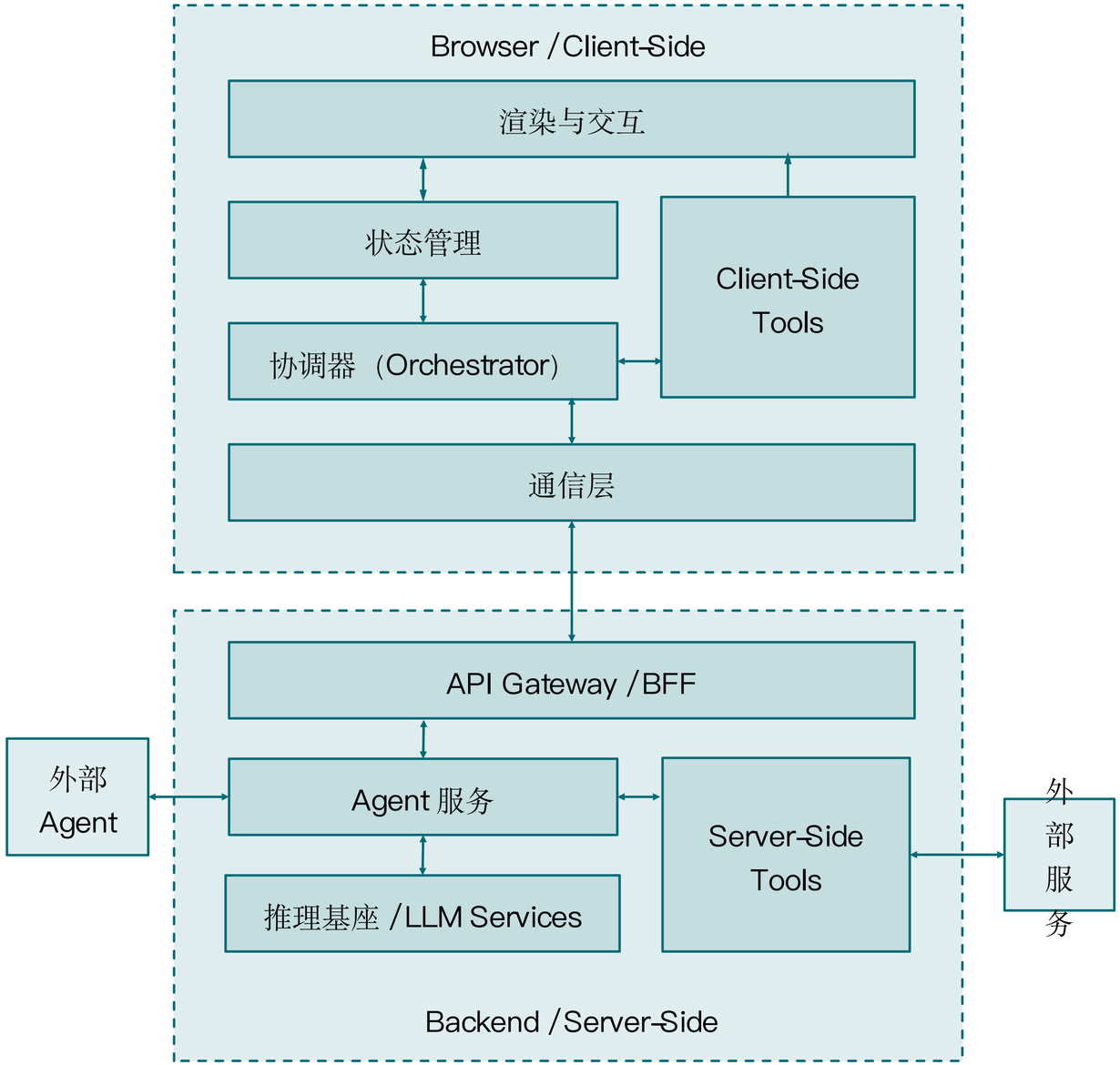
视觉仍然是人类获取信息最高效的器官。GUI 的信息密度远超于其他模态；
GUI 仍将在人机交互中扮演重要角色，会与新的交互范式融合，成为更强大的人机交互体系的一部分。这个转变对 Web 技术也产生了挑战

/02

Web 作为 AI Agent 渲染容器

渲染层是所有 AI Agent 渲染容器技术挑战的核心所在，它决定了 Agent 输出内容的丰富度和交互性。

/02 Web 作为 AI Agent 渲染容器 —应用架构划分



AI Agent 应用架构划分

当下业界更侧重在 AI Agent 架构的讨论，
对应用和客户端的架构讨论甚少。

为了方便讨论，我们不妨把 AI Agent 客户端的架构做拆分

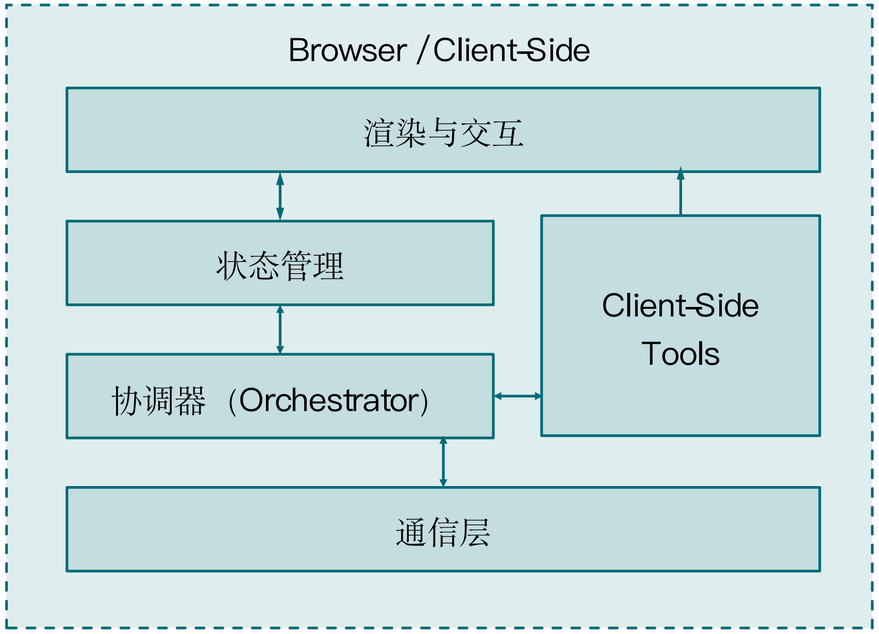
通信层：由 ChatGPT 普及的「打字机」效果已经成为用户对 AI 应用的实时性预期。背后是一套完整的全栈式架构。LLM 逐个生成 token，通过 SSE 协议和网关发送到前端。最终渐进式的渲染到屏幕。

由于客户端和网络环境的复杂性，并不是所有环境都完美支持 SSE，因此在工程上需要增加一层通信层封装，屏蔽具体通信细节的实现：无论是通过 SSE，WebSocket，还是 gRPC 等其他流式方案。

协调器：在传输的数据流之上，业务需要定义自己的应用层数据协议。协调器负责解析在网络传输层之上定义的业务协议。例如流式管控、会话状态、工具调用等；更新 UI 状态。

/02 Web 作为 AI Agent 渲染容器 —应用架构划分

AI Agent Web 端架构



Client-Side Tools：一些场景需要让 AI Agent 调用客户端侧的一些能力，例如 AI Coding 场景的读写文件，C 类 Agent 的读取地理位置、请求权限并开启摄像头等；自动读写表单等；这类客户端工具需要在容器初始化时，以 tools 的形式给到 AI Agent，在协调器解析识别后发起调用。

状态管理：以一种可预测的方式存储和更新应用的所有状态，并让 UI 能够响应这些状态的变化。通常遵循「单向数据流」的模式。

渲染与交互：所有生成式 AI 都不会满足于只输出纯文本。LLM 在不断支持更多的「模态」。渲染和交互层的范式是随着前面提到的「任务场景」、「自主性」而同步产生变化的。

渲染层是所有 AI Agent 渲染容器技术挑战的核心所在，它决定了 Agent 输出内容的丰富度和交互性。

接下来着重讨论这一部分。

/02 Web 作为 AI Agent 渲染容器 —渲染层的三个阶段

生成式范式从文本拓展到 UI

随着 LLM 的能力和 AI Agent 应用场景复杂度的提升
一方面，用户不再满足于与 LLM 进行纯文本的沟通
另一方面，产品也希望把生成式的范式，
从「生成文本和内容」逐渐拓展到「生成 UI 和应用」。

阶段一：流式文本和 Markdown

阶段二：Workflow 范式下的卡片定制

阶段三：「生成式 UI」

/02 Web 作为 AI Agent 渲染容器 —渲染层的三个阶段

阶段1:流式文本和 Markdown

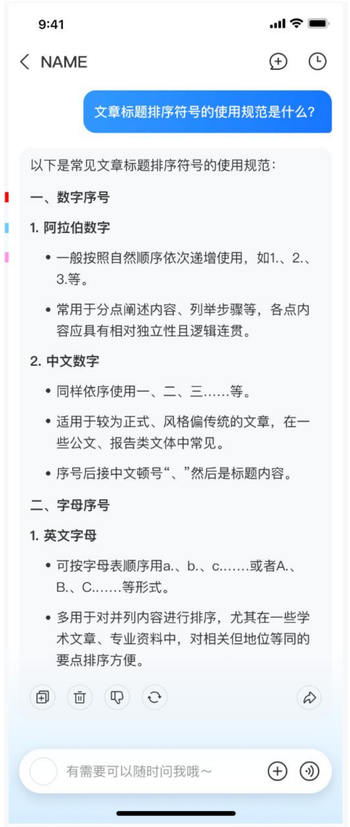
几乎所有的 LLM 都选择支持 Markdown 格式作为富文本的输出方案；

相比 HTML 的复杂嵌套结构，Markdown 的语法更清晰，歧义更少。在 LLM 的能力水位较低的早期，有些模型可能无法稳定生成 HTML，但基本都可以正确的生成 Markdown

经济上：Markdown 带来更少的 token 消耗。以开源项目 ant-design 的 README 计算，要达到相似的渲染效果，HTML 需要 5 ~ 6 倍的 token；

Markdown 其实是一种「关注点分离」。LLM 在输出内容的时候只需要考虑「加粗」，「引用」等逻辑，无须考虑这里适合几号字，或者引用的样式怎么写。

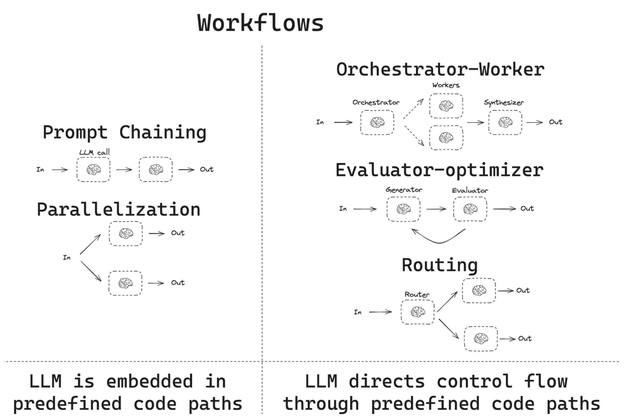
Markdown 的 Web 解决方案有特别多，这里不做赘述。



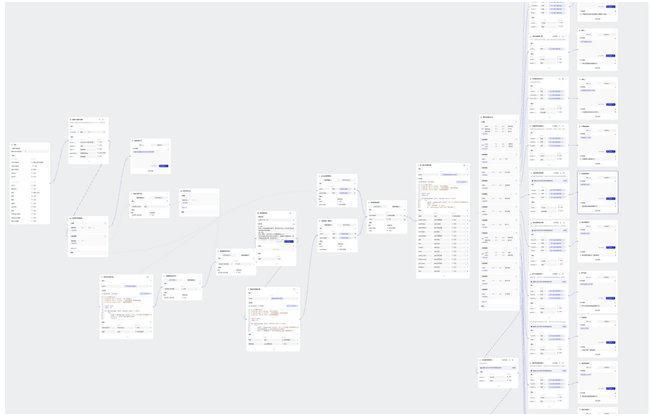
基本的 Markdown 渲染

/02 Web 作为 AI Agent 渲染容器 —渲染层的四个阶段

阶段2:Workflow 范式下的定制卡片



使用「Workflow」的范式构建 AI Agent



一个商用的巨大的 Workflow

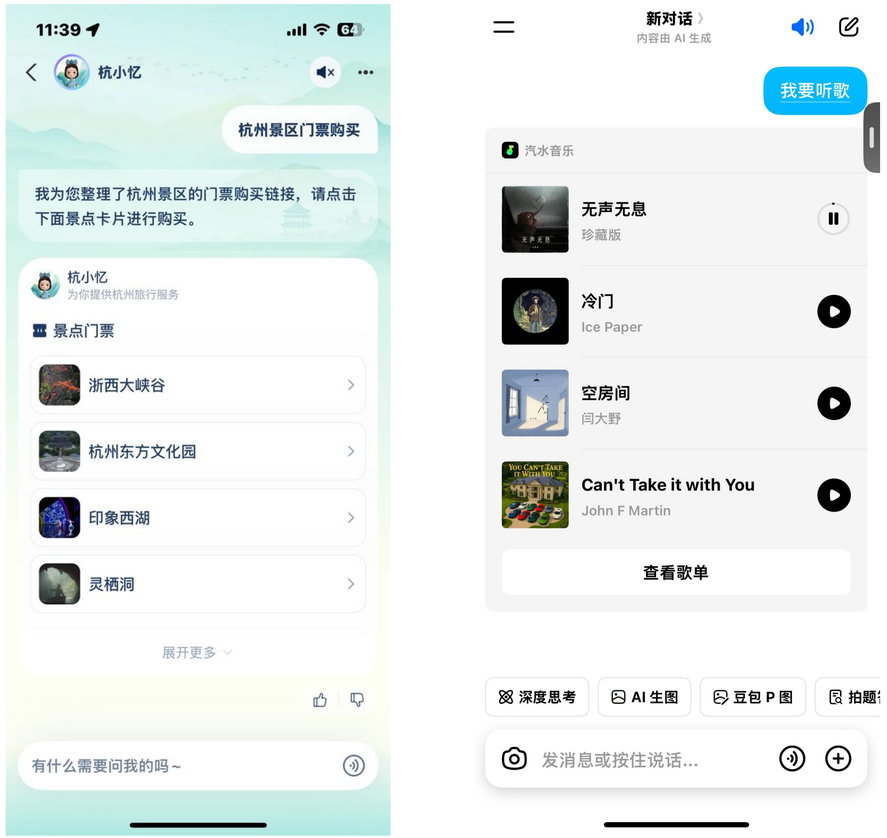
在 Workflow 中通过「卡片消息」串联分支和预置卡片

早期 AI 应用很快发展到「Workflow」的范式：根据用户的意图派发到不同的分支处理，最后输出结果。这个范式的挑战主要在：

1. **模块化拆分**：根据需求预开发一系列的业务卡片或卡片模板；主对话容器与各业务卡片应当分离。卡片需要定义标准接口；
2. **动态化加载**：需要有卡片的发现机制和动态加载机制。Code Splitting 是不够的，需要 umd 或者 importmap；
3. **通信机制**：卡片不是纯静态，会有自己的业务逻辑和状态。需要设计卡片与主对话的消息与状态同步方案；

/02 Web 作为 AI Agent 渲染容器 —渲染层的三个阶段

Workflow 范式下的定制卡片的局限与批评



一些 Workflow 范式的 AI 应用

局限与批评：

基于 Workflow 的定制卡片范式虽然提供了一种非常清晰可行的方案。
但是局限性也非常明显。

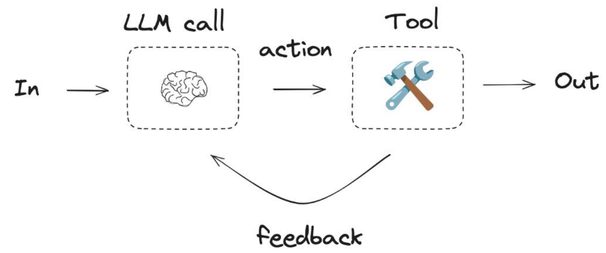
智能化低：意图和分支派发没有多少 AI Agent 的自主性可言；这是当前商业化落地的一种妥协：企业需要确保核心业务流程（如交易、预订）是 100% 可靠的，不能允许 AI 「自由发挥」导致失败。但代价就是牺牲了真正的智能。

表达受限：只有预设的部分意图可以使用预开发卡片渲染，一旦超出，会降级到大模型纯文本或 Markdown 输出；用户的体验会在“高度结构化”和“完全无结构”之间反复横跳。这种「降级处理」是该范式脆弱性的直接体现。

组合性低：现实世界的需求往往是复合的、多任务的。用户不会像操作菜单一样，一次只执行一个精确定义的任务。Workflow 的范式很难做到用户「我先点一杯咖啡再听歌」的需求，在 UI 层也无法把两张业务卡片有机结合渲染。

/02 Web 作为 AI Agent 渲染容器 —渲染层的三个阶段

阶段 3: Agentic AI 与「生成式 UI 」范式



LLM directs its own actions
based on environmental feedback

Agentic AI 强调的是 AI 的自主性和代理性。AI Agent 能在人类尽量减少干预的情况下自主决策的完成任务。

这样的范式下，无法进行「意图」和「卡片」的绑定。渲染层往往直接降级到 Markdown 输出。

生成式 UI (Generative UI) 是一种可行的方案：

放弃「意图」与「卡片」之间的刚性绑定。转而将前端 UI 拆解成一系列标准化的、可复用的组件；

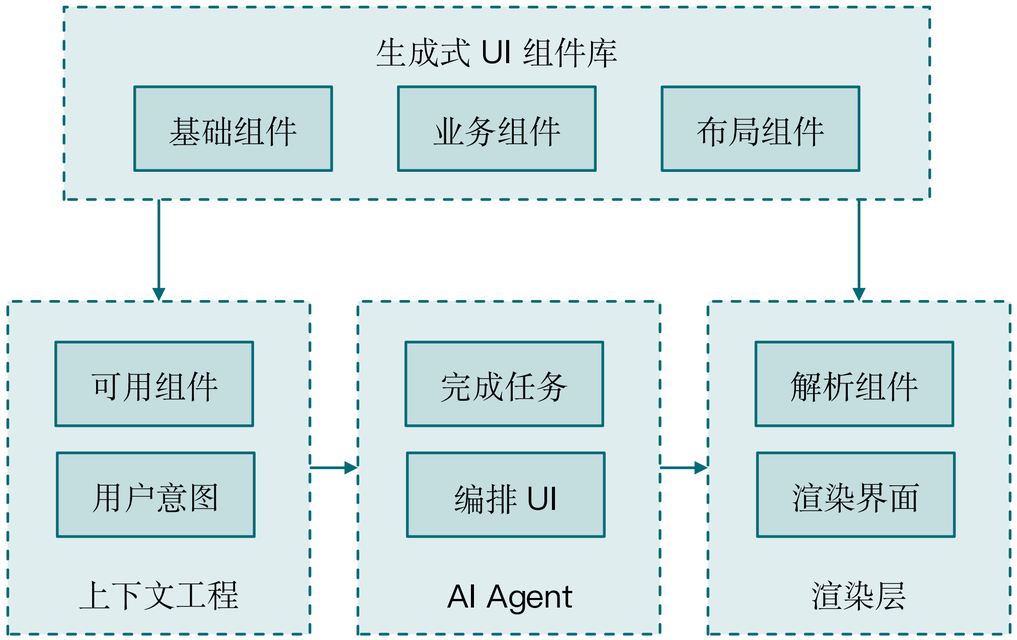
AI Agent 的任务不再是选择一个「套餐」，而是根据任务的实时需要，自主选择如何使用这些组件「搭建」出当下最合适的交互界面

「生成式 UI」≠自底向上生成每一个 UI 要素

不是做不到，而是没必要

许多 Web 组件和业务组件是复杂的。没必要重新发明一遍。
C 类 UI 需要注重产品设计规范、性能与一致性。复用是最经济的做法

阶段 3: Agentic AI 与「生成式 UI 」范式



一个「生成式 UI」的流程

构造「生成式 UI」组件库

描述：定义组件的用途和场景。类似 MCP Tools 的 Description
入参：定义组件的配置项。类似 MCP Tools 的 Parameters
事件：定义组件的交互输出。
示例：组件的调用示例。让 LLM 能正确参考。
静态资源：组件的实现物料。由渲染容器加载；

在 Agent 中使用「生成式 UI」范式

上下文工程：通过上下文工程，把当前业务可用的组件和用户意图传入 LLM 上下文。结合组件的描述、入参和示例做 few shot；
Agentic Agent：在完成任务后，增加「编排输出」任务，根据上下文编排 UI
渲染层：接收并解析 AI Agent 的流式返回，根据协议动态加载资源，完成布局渲染

/02 Web 作为 AI Agent 渲染容器 —渲染层的三个阶段

阶段 3:「生成式 UI 」协议

基于 Markdown 增强语法的「生成式 UI」协议

一种切实可行的「生成式 UI」协议方案就是 Markdown 增强。仍然基于 LLM 的文本流，在原生 Markdown 语法之上，定义额外的组件语法，实现「图文混排」式的「生成式 UI」：

例如：

- 使用 <Chart /> 标签来混入；
- 使用 <MediaPlayer src=… /> 来混入音频播放器；
- 使用 <FlightInfo id=…/> 直接调用航班信息卡片；

```python
async def generate_report(state: AgentState) -> AgentState:
    """
        生成报告
    """
    generate_role_define_prompt = """
    ### Role Definition
        You are a stock market analyst expert.
    ### Task
        You are given a stock ticker and a date.
        You need to analyze the stock market and provide a report.
        The report should be in markdown with enhanced symbols and components.
        - use <highlight> to highlight the important parts of the text.
        - use <CandlestickChart> to draw the candlestick chart.
    ### Example
        <CandlestickChart title="AAPL" />
    """

    model = get_llm_client(state)
```

```
## 📈 Overview & Market Sentiment

Tesla Inc. (TSLA) demonstrated significant strength during the trading session on Friday,
August 29, 2025, largely outperforming the broader market indexes. The primary driver for the
stock was a **major analyst upgrade from Morgan Stanley**, which revised its price target to
$310, citing stronger-than-expected Full Self-Driving (FSD) adoption rates in the second
quarter. <highlight>This news injected substantial optimism, shifting investor focus back to
Tesla's long-term technology and AI growth narrative.</highlight> The overall market
sentiment was cautiously positive, but the EV and tech sectors saw the most aggressive buying
activity, with Tesla leading the charge.

---

## 📊 Performance Chart

The candlestick chart below illustrates the intraday price action for TSLA on August 29, 2025.

<CandlestickChart title="TSLA" />

---
```

从 Prompt 到 Markdown 到最后渲染的示意

阶段 3: 「生成式 UI 」的协议

基于 UI Schema 的渲染协议

Markdown 仍然基于文本流和文本布局，表达能力和灵活性受限。

我们可以进一步的分拆组件，引入「布局和结构组件」，定义一个功能更强的 DSL 来支持更灵活的布局。

右侧是我们定义的一个 UIDSL、实际生成的 UI Schema 和实际渲染结果。

```ts
export interface UIDSL {
  /** 当前节点的类型，是组件还是原子html标签,当为VirtualDom类型，交由消费测
*/
  type: 'Component' | 'Tag' | 'VirtualDom';
  /** 组件或者html的tag名称 */
  name: string;
  /** 重复渲染配置，默认为该节点整个完全重复 */
  repeat: RepeatConfig;
  /** 组件版本号，type为Component时必填 */
  packageVersion?: string;
  /** 组件包名，typNe为Component时必填 */
  packageName?: string;
  /** 组件的tailwind类名 */
  className?: string;
  /** 样式 token，就近原则 */
  tokens?: string[];
  /** 属性参数，key为参数名，值为参数结构 */
  params: Record<string, ParamConfig>;
  children: UIDSL[];
}
```


```
"root" : { 4 items
  "type" : string "Tag"
  "name" : string "div"
  "className" : string "flex flex-col bg-white rounded-lg overflow-hidden shadow-lg max-w-4xl mx-auto font-sans"
  "children" : [ 2 items
    0 : { 4 items
      "type" : string "Tag"
      "name" : string "div"
      "className" : string "bg-[#FFC900] p-6 relative overflow-hidden"
      "children" : [ 2 items
        0 : { 4 items
          "type" : string "Tag"
          "name" : string "div"
          "className" : string "relative z-10"
          "children" : [ 3 items
            0 : { 4 items
              "type" : string "Tag"
              "name" : string "div"
              "className" : string "text-sm text-black/80 font-medium mb-1"
              "params" : { 1 item
                "textContent" : { 2 items
                  "bindType" : string "Static"
                  "value" : string "Stock Market Analysis"
                }
              }
            }
            1 : { 4 items
              "type" : string "Tag"
```


基于 UI Schema 定义的渲染 DSL

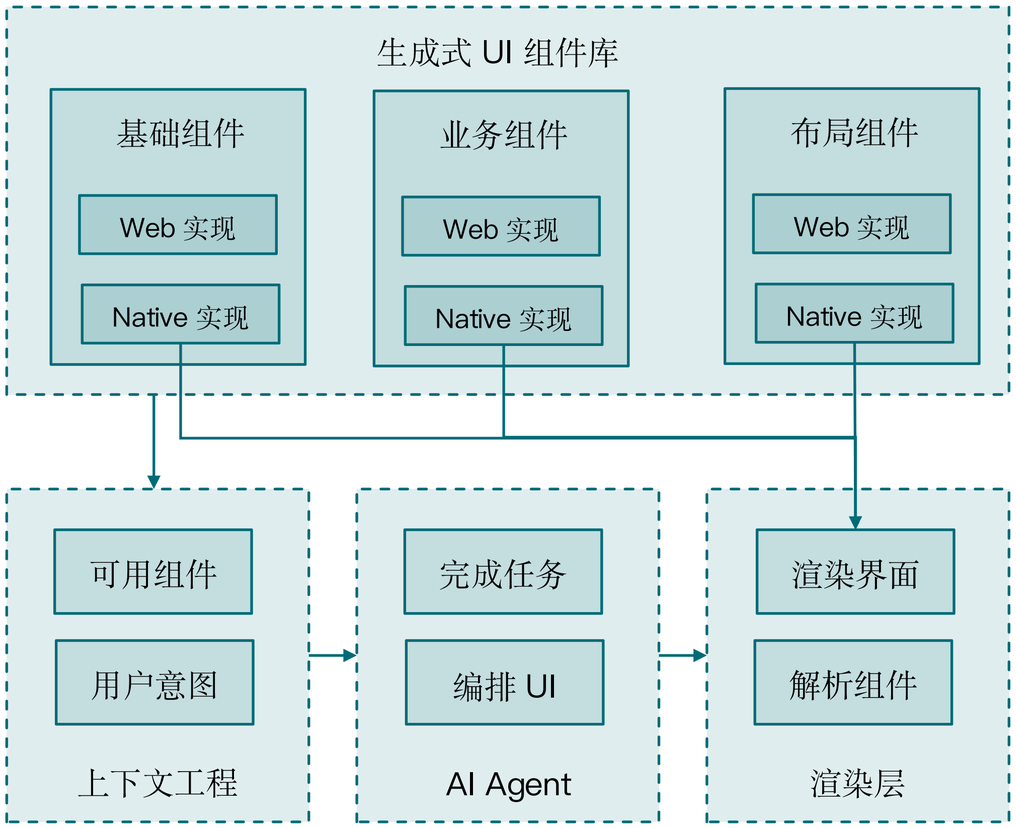
阶段 3:「生成式 UI 」对 Web 技术的挑战

更复杂的流式处理：应对“不完整”的挑战： LLM 的输出是线性的、逐个 token 的，而 UI 渲染需要的是结构完整的、有明确状态的指令。需要在渲染层前置一个健壮的「流式处理与解析缓冲区」。实现识别、解析、容错与恢复；

复杂的模块化与动态化：本质是「微前端架构」： AI Agent 的 Web 容器，需要从一个单体应用（Monolithic App）演变为一个「对话式微前端（Conversational Micro-Frontends）」的宿主容器。包括运行时动态加载（importMap、Module Federation）、研发与发布分离、会话历史与多版本共存等。

样式和运行时隔离： 组件加载到主会话中后，需要保证不出现样式、脚本上的冲突。包括不同组件之间的冲突、同一组件不同版本的冲突、组件对主会话的副作用等。需要一个真正可用的「沙箱」。

阶段 3：跨端的 Agent



跨端的 UI 编排流程

同样功能的 AI Agent 不应因为渲染端的差异而实现多遍。

例如：一个机票预订 Agent 的核心逻辑（如何查询航班、如何处理订单），不应该因为目标是 Web 还是小程序而重写。

这里需要设计一层跨端的组件库；通过上下文工程传入 LLM 的描述是渲染目标无关的抽象描述和表示；

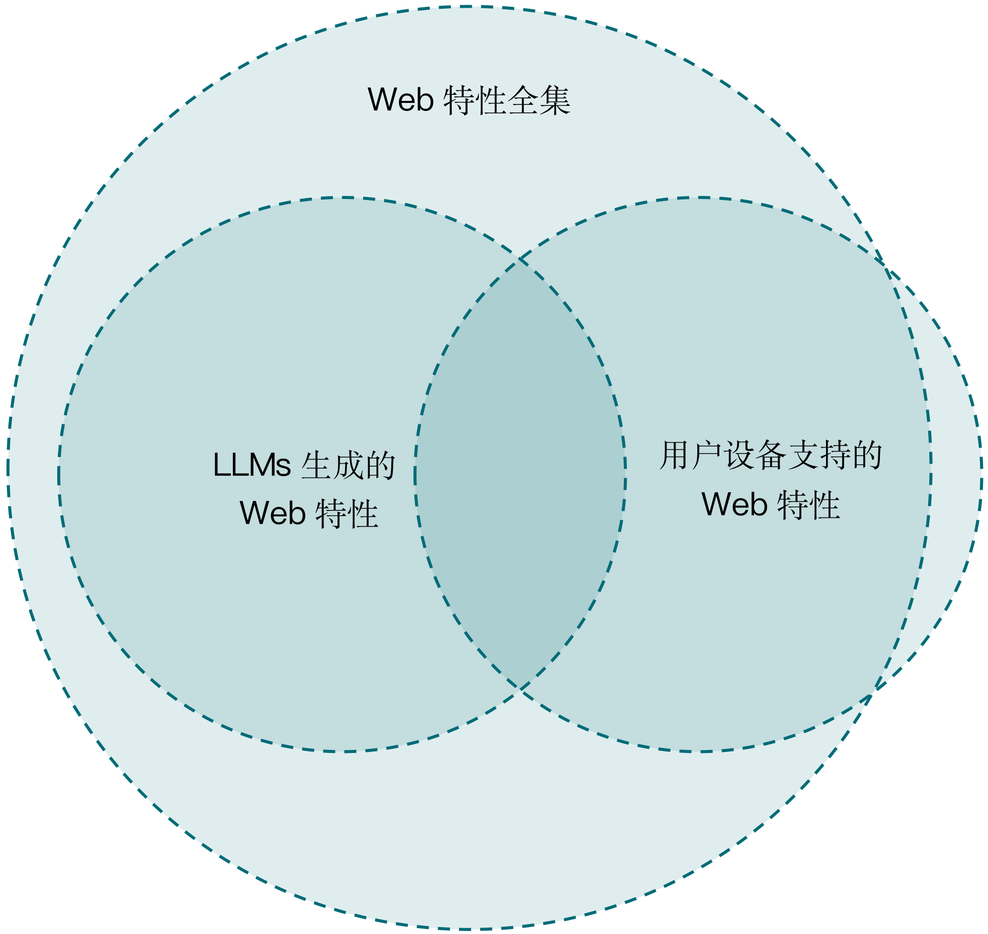
在渲染时，由渲染层根据实际渲染端而拉取不同的组件进行渲染。

批评：UI Schema /DSL 真的有必要吗?

翻译：为什么我们不生成 Raw Web，而是需要一个 DSL，或者一个子集

/02 Web 作为 AI Agent 渲染容器 —生成式 UI 的未来

为什么「生成式 UI」需要一个子集



「无限画布」vs.「有限画框」：

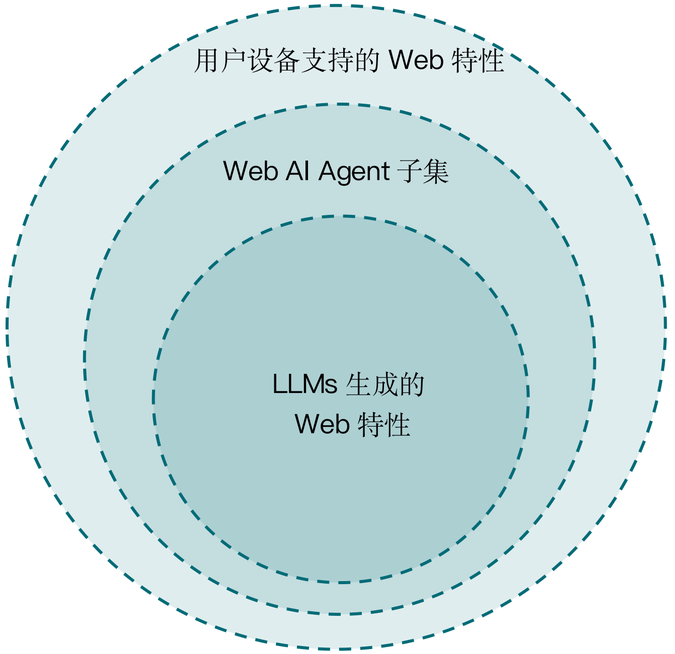
- Web 的规范是一个「无限画布」，包含了过去数十年所有的技术规范、特性和标准；
- 用户的设备是一个「有限画框」，用户的设备、网络和浏览器永远是具体的、受限的，充满差异的。
- 让 AI Agent 不受限的生成 Web 内容，意味着需要在工程上解决「受限」问题。

/02 Web 作为 AI Agent 渲染容器 —生成式 UI 的未来

为什么「生成式 UI」需要一个子集

无论是不是 Raw Web，对 Web 渲染核心挑战依然不变：

- 生成 Raw Web 并不代表抛弃模块化、组件化和样式隔离；
- 因此我们在上一部分讨论的渲染容器的挑战：**流处理**、**模块化**和**隔离**，在生成 Raw Web 的路线中依然存在，并且挑战更复杂。



RFC：是否应该建立一套 Web AI Agent 的子集标准：

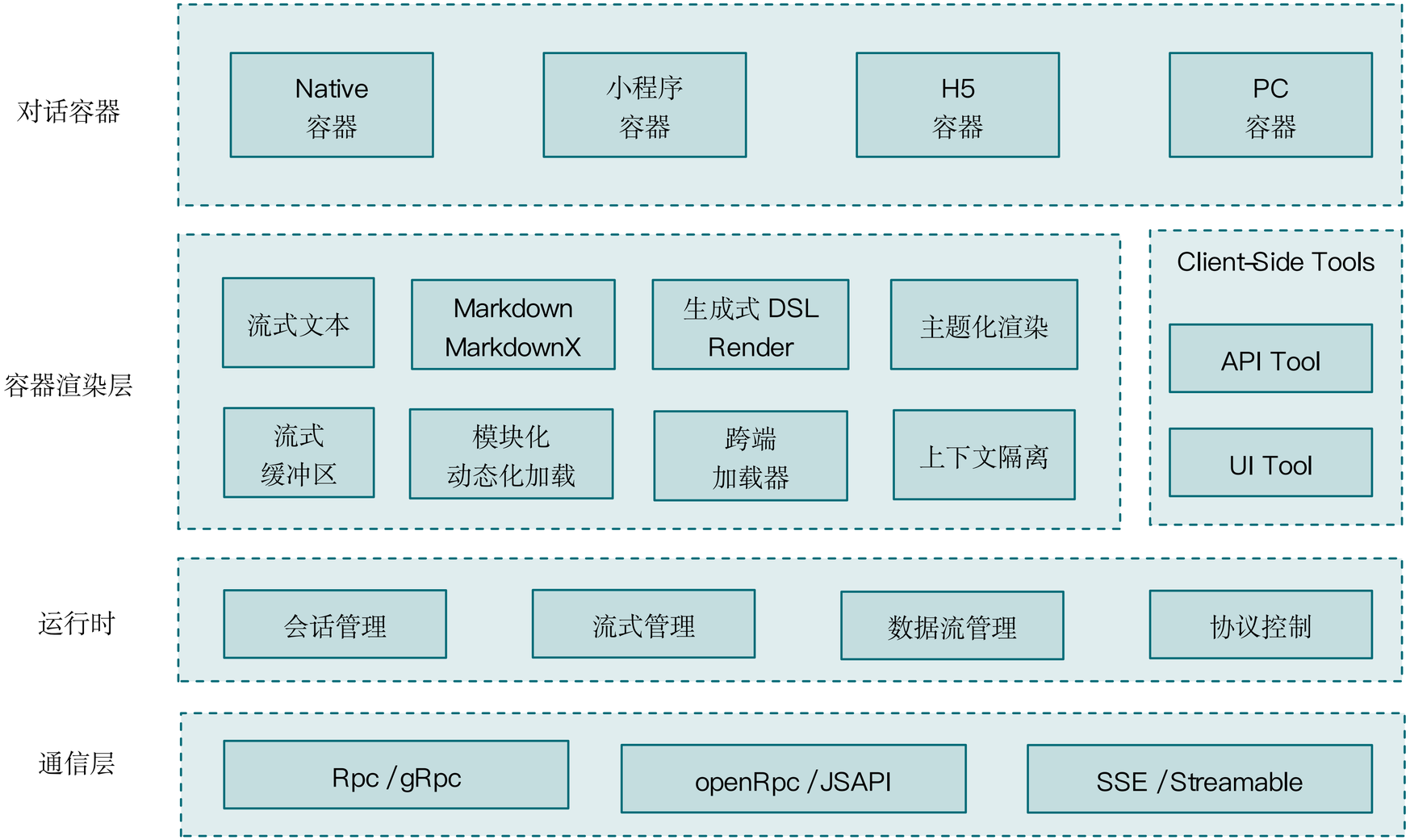
- DSL 是解决摩擦的临时方案。由于缺少规范，各组织重复发明 DSL 导致生态的割裂。
- 希望定义一套精简的、跨平台的 Web AI Agent 标准；
- LLM 未来不再生成 Raw Web，也不是自定义的 DSL；而是直接生成子集产物；

/03

总结

为 AI Agents 构建 Web 容器

/03 为 AI Agent 构建 Web 容器



总结

「生成式 AI」的范式正在从文本和内容生成拓展到界面和应用生成： 与 AI Coding 不同，它是一个在运行时由 AI Agent 根据需求、上下文和用户输入，动态生成和组装的产物。

「生成式 AI 」的范式转移给 Web 带来了系统性摩擦： 当前的 Web 架构在性能、可靠性和安全性方面与 AI Agent 的生成式范式需求存在系统性摩擦，这要求我们对 Web 标准和架构最佳实践进行思考。

通过构建 AI Agent 的 Web 容器来缓解摩擦： Web 开发者应该主动理解并拥抱这种范式转移，给用户带来更智能的体验。

标准化组织应该制定 AI Agent 渲染相关标准： 制定一套开放、可扩展、安全且与平台无关的中间标准，用于 AI Agent 生成动态、可交互的用户界面。赋能开发者构建可互操作、跨平台的下一代智能应用，并为最终用户提供一致、可靠且丰富的体验。

Thanks

附录 知识产权说明

一、字体版权

本演示文档的制作使用了“OPPO Sans 4.0”字体。

本使用行为遵循《OPPO Sans 字体许可协议》的条款，特此鸣谢。

二、图片版权

文档内所引用的所有截图、图像均来源于互联网公共资源，其著作权归原作者所有。

本作品对相关素材的引用仅限于个人学习、研究或教学演示之目的，属于《著作权法》规定的合理使用范畴。

若任何内容涉及侵权，请及时联系我们予以删除。