前端 Markdown 渲染库调研（WebMCP Sidecar）— Deep Research（2026-02-14）

Executive Summary

针对浏览器扩展（MV3 / Side Panel / Options）里渲染 **LLM 输出 Markdown** 的场景，核心矛盾是：**美观** vs **安全（XSS）** vs **体积/集成成本**。综合可靠性、精简程度与“无打包器也能落地”的现实约束，最推荐的两条路线是：`markdown-it（禁用 HTML）` 与 `marked + DOMPurify（渲染后净化）`。[1][2][3]

Key Findings

- `markdown-it` **在“默认不启用 HTML”的模式下可做到相对安全且不强依赖 sanitizer**，并且其安全策略（禁用危险协议等）在官方 security 文档中有明确说明。[1]
- `marked` **明确声明“不做输出 HTML 的 sanitize”**，若用于不可信内容渲染（LLM 输出、用户输入），必须配合 HTML sanitizer（例如 DOMPurify）。[2][3]
- `micromark` **主打“safe by default + CommonMark 100%”**，并在文档中明确指出启用危险 HTML/协议会带来 XSS 风险；但其“精简 + 生态组合”通常更偏工程化（更适合 unified/remark 体系），在“无 bundler 的扩展里”集成门槛更高。[4]
- **React 体系的** `react-markdown` **默认会逃逸/忽略 HTML**，并提示只有在可信内容下才使用 `rehype-raw`；该路线在你当前项目（原生 DOM 渲染）里不一定合适，但其安全策略值得借鉴。[5]
- **Sanitizer 选择上，DOMPurify 是最主流的前端 HTML 净化方案之一**，并提供明确的 threat model；同时也需要关注历史安全公告与版本下限（例如曾有 ❤️.2.4 的修复点）。[3][6][7]

Detailed Analysis

1) 你的真实需求拆分（决定选型）

你的 UI 里消息分两类：

1. **人类可读对话（user / assistant）**：最适合 Markdown 渲染（标题/列表/代码块/链接），提升可读性。
2. **工具事件（tool.use / tool.result / system.notice 等）**：更适合保持等宽文本 + 可折叠/截断策略（你已做了 tool.result 截断），避免被 Markdown“排版美化”导致信息丢失。

因此建议策略是：

- 仅对 `assistant.message` / `user.message` 的 `text` 做 Markdown 渲染；
- 其它事件继续使用纯文本（或仅做轻量语法高亮/折叠）。

2) 候选库对比（按“可靠 + 精简 + 可落地”排序）

方案 A：`markdown-it`（推荐：默认禁用 HTML）

适用：你想要“开箱即用 + 安全策略清晰 + 可做少量扩展”的方案。

优点：

- 官方明确给出安全建议：**不要启用 HTML** 是最优策略，并且默认会阻止一些可用于 XSS 的链接协议（如 `javascript:` 等）。[1]
- 工程上成熟、生态丰富、bug 风险相对可控。

缺点：

- 相对 `marked` / `micromark`，体积通常更大（尤其你还想要一些 GFM 功能时）。

落地建议（你这个扩展）：

- `markdown-it({ html: false })`（坚持不启用 HTML）
- 加 `linkify`（可选）但注意链接策略：给所有外链加 `rel="nofollow noopener noreferrer"` 与 `target="_blank"`（你可以在渲染后的 DOM 上二次处理）。

方案 B：`marked + DOMPurify`（推荐：渲染后净化）

适用：你想要“最少依赖、集成快、渲染效果够用”的方案。

关键事实：

- `marked` 官方文档显式警告：**不 sanitize 输出 HTML**。[2]
- DOMPurify 的目标是“移除可导致 XSS 的内容并返回可安全注入 DOM 的字符串”，并在 wiki 里定义了威胁模型/非目标等细节。[3][6]

优点：

- 组合简单直观：`html = marked(md)` → `safeHtml = DOMPurify.sanitize(html)` → `el.innerHTML = safeHtml`。
- 在“无 bundler、纯前端”场景可以很容易用 UMD/单文件构建落地。

缺点：

- 你必须把“安全”责任扛在 sanitizer 上：配置、升级、以及任何“sanitize 后再被二次加工”的链路都需要谨慎（DOMPurify 也提示 sanitize 后再修改可能破坏安全性）。[3]
- Sanitizer 本身也有历史漏洞修复点，意味着你要对版本与升级保持敏感度。[7]

方案 C：`micromark`（安全与规范性最强，但更像“底层组件”）

关键事实：

- `micromark` 自述“safe by default”，并说明开启 `allowDangerousHtml/Protocol` 会引入 XSS 风险，同时建议对输入大小做限制以避免 DoS/崩溃等问题。[4]

优点：

- CommonMark 合规性强、默认安全策略更“硬”。

缺点：

- 实际使用时，很多团队会走 unified/remark 生态，组合上更工程化；在你当前“扩展内原生 JS + 尽量不引入构建链”的前提下，集成与维护成本可能更高。

方案 D：`commonmark.js`（参考实现，适合“最朴素 CommonMark”）

关键事实：

- `commonmark.js` 明确说明：默认不 sanitize raw HTML / link attributes；用于不可信输入时需要启用 `safe` 或额外 sanitizer。[8]

优点：

- 规范、稳定、提供单文件 dist 的可用性（适合“无 bundler”）。[8]

缺点：

- 功能与生态相对没有 `markdown-it` / `marked` 灵活。

3) 安全要点（无论你选哪个库）

1. **禁用/限制 raw HTML**
   最简单的策略就是：不让 Markdown 里的 HTML 生效（或渲染后 sanitize）。`markdown-it` 官方也把“禁用 HTML”作为首选策略。[1]

2. **限制链接协议**
   避免 `javascript:` / `data:` 等协议形成可点击攻击面； `markdown-it` 默认会拦截多类危险协议。[1]
3. **Sanitize 的“最后一步原则”**
   如果你选择 sanitize：应尽量保证 sanitizer 在“最后一个不可信转换步骤之后”执行； `rehype-sanitize` 也强调“sanitize 后面的处理可能重新引入不安全”。[9]
4. **CSP（扩展页也能用）**
   你在扩展页面里渲染 HTML 时，仍建议保持扩展页面 CSP 严格（减少脚本注入面），并避免 `unsafe-eval` 等宽松策略（此处属于架构层面建议）。

Areas of Consensus

- **Markdown 渲染不等于安全**：如果内容不可信（LLM 输出、外部输入），要么禁用 HTML，要么做 sanitizer。[1][2][8][9]
- **DOMPurify 是主流 HTML sanitizer 方案**，并且其威胁模型/约束需要遵守（不要 sanitize 后再做危险的 HTML 操作）。[3][6]

Areas of Debate

- **“默认禁用 HTML 是否足够”**： `markdown-it` 认为禁用 HTML + 默认链接过滤在多数场景足够安全；但如果你还要自定义插件/自定义渲染（例如自动生成 id/name），仍可能引入 DOM clobbering 等风险，需要额外规则与前缀策略。[1]
- **“是否应依赖 sanitizer”**： `marked` 路线几乎必然需要 sanitizer；有些团队更倾向“彻底不生成 raw HTML”，从根上减小 XSS 面，但工程复杂度会上升（例如走 AST 渲染）。[2][4]

Recommended Choice（针对 WebMCP Sidecar）

若你优先“可靠 + 精简 + 快速落地”：

- **首选**： `markdown-it` （坚持 `html: false` ）[1]
- **备选**： `marked + DOMPurify` （渲染后 sanitize，严格限制配置与升级流程）[2][3]

若你优先“规范/安全默认 + 可扩展生态（偏工程化）”：

- `micromark` （并保持默认安全选项，不开启危险 HTML/协议）[4]

Sources

[1] markdown-it — Security documentation (`docs/security.md` via GitHub Security overview). https://github.com/markdown-it/markdown-it/security (官方；高可信)

[2] Marked Documentation — “Warning: Marked does not sanitize the output HTML…”. https://marked.js.org/ (官方；高可信)

[3] DOMPurify — GitHub repository README. https://github.com/cure53/DOMPurify (官方；高可信)

[4] micromark — README “safe by default” + security section. https://github.com/micromark/micromark (官方；高可信)

[5] react-markdown — Appendix A: HTML in markdown (escapes/ignores by default; rehype-raw only for trusted content). https://github.com/remarkjs/react-markdown (官方；高可信)

[6] DOMPurify Wiki — Security Goals & Threat Model. https://github.com/cure53/DOMPurify/wiki/Security-Goals-%26-Threat-Model (官方；高可信)

[7] GitHub Advisory Database — DOMPurify CVE-2025-26791（受影响 ❤️ .2.4）. https://github.com/advisories/GHSA-vhxf-7vqr-mrjg (权威安全数据库；高可信)

[8] commonmark.js — “A note on security” (no sanitize by default; suggest safe option/sanitizer). https://github.com/commonmark/commonmark.js (官方；高可信)

[9] rehype-sanitize — Security notes (“sanitize after last unsafe thing”). https://github.com/rehypejs/rehype-sanitize (官方；高可信)

Gaps and Further Research

- 你的扩展是否要支持 GFM（表格/任务列表）与代码高亮（highlight.js / shiki）？这些会显著影响体积与实现复杂度。
- 若要支持“复制时保留 Markdown/纯文本/富文本”三种模式，需要再单独设计剪贴板策略与 UX。