BEYOND CODE GENERATION

Agentic Framework for Reliable OpenBMC Platform Porting

超越代码生成：可靠 OpenBMC 平台移植的 Agent 框架

演讲人：郑春辉 | 字节跳动固件架构师

2026 OCP China Day | 北京

议程 / Agenda

01
痛点剖析

02
理念重塑

03
技术深挖

04
闭环演进

痛点剖析：固件移植的瓶颈与幻觉

平台移植的瓶颈

“OpenBMC的Yocto recipe可以支持特性裁剪，理论上可以解决问题。”

然而，现实真的如此吗？

项目需要的知识维护

1

人脑记忆

知识全锁在专家的头脑中。一旦发生人员流动或岗位变动，积累的调试经验流失。

2

IM 消息流

最真实的调试逻辑，由于沟通压力，完全散落在零碎的聊天记录和信息流中，彻底无法被检索归档。

3

静态指南

维护了厚达大几百页的静态指南，由于过于冗长，实践中工程师漏掉配置的概率极高，容易出错。

4

代码注释

代码注释和实际代码不吻合，随着项目分支演变甚至出现不正确的描述。

5

文档腐烂

特性迭代频繁，文档同步往往极度滞后，文档持续发生腐烂，在 review 代码阶段才痛苦地暴露问题。

配置爆炸

配置爆炸

固件移植工作量的构成

指数级组合爆炸的乘数效应

在云数据中心环境下，硬件形态呈现爆炸式增长：

通用服务器　BOX定制机、　整机柜　多主机共享平台　…

01 传感器配置（Sensor SDR）

收集并解析 SensorSDR、设置各种高低阀值及 Entity manager 映射

02 Kernel DTS配置（DTS config）

配置复杂的内核配置

平台数 × 配置维度 × 多供应源器件

03 FRU / Inventory 数据（FRU&Assets）

指数级剧增的复杂配置矩阵

初始化及编排FRU资产信息和底板Assets映射

04 散热控制策略（ThermalPID）

调试风扇PID阶梯配置（Stepwise），

以适配不同冷态环境

05 Redfish 适配（Redfish Interface）

- 项目量激增：以前我们一年可能只需支撑两三款新平台。现在面对海量异构和定制化需求，一年要支撑几十款不同的组合实体。
- 形态差异最狭：通用计算服务器、各种复杂的BOX定制机、整机柜大功耗电源管理，多主机共享平台，其配置要求有时，其散热求时规则有巨大异构
- AI时代周期更短：大模型算力迭代迫使交付周期较之前压缩50%以上

传统纯人工调试已经走到效率红线。

对接Redfish标准管理接口，适配异构服务器底层资源模型

06 胶水脚本与 Recipes（Bitbake Recipes）

编写 Shell/Python 胶水代码和编译 recipe 编译依赖

⚠ 项目规模激增 × 异构复杂度提升 × 周期压缩 → **传统纯人工调试已达效率极限**

为啥不用AI呢?

为啥不用AI呢?

“既然固件移植的工作说到底都是文本和配置文件上的修改，而AI大模型天然又最擅长处理文本，为什么不直接使用大模型来做这些事情呢?”

“使用AI Agent进行代码与配置生成，在理论上似乎可以非常完美地一键搞定全新项目移植，让我们彻底告别枯燥的手动排错。”

我们的期待：

深刻的底层本质探究!

我们一开始也是这么想的。第一次尝试用大语言模型去自动生成传感器配置时，输出结果看起来像模像样——

```
{
  "sensor": "Device1",
  "i2c_addr": "0x68",
  "freq": "100Hz"
}
```

格式正确、字段齐全。

直到我们把这些完美的代码，真正拿去跑 Regression 回归测试阶段——同一个输入，跑三次，三个不同的结果。

一次，生成了一个根本不存在的传感器配置。一次数值上产生幻觉，同源数据在不同特性配置上没有匹配。

纯大模型代码生成的致命缺陷

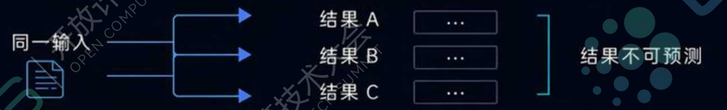
1. 严重幻觉：

凭空捏造不符合物理逻辑的硬件引脚和寄存器地址配置

- 实际不存在的寄存器地址
- 不存在的引脚复用

2. 输出失控：

大模型自身的不确定性导致输出不收敛，难以通过回归模型验证



3. 不可审计：生成过程是黑盒，在硬件强相关的固件领域，

“为什么大模型选择匹配这个时序或值”

这并不是大模型自身的 Bug，而是“代码生成”(Code Generation)这一思想路线本身的局限性：

生成本身是一个开放式问题，而在高确定性底层固件领域，开放往往就代表着不可控(缺乏约束)。

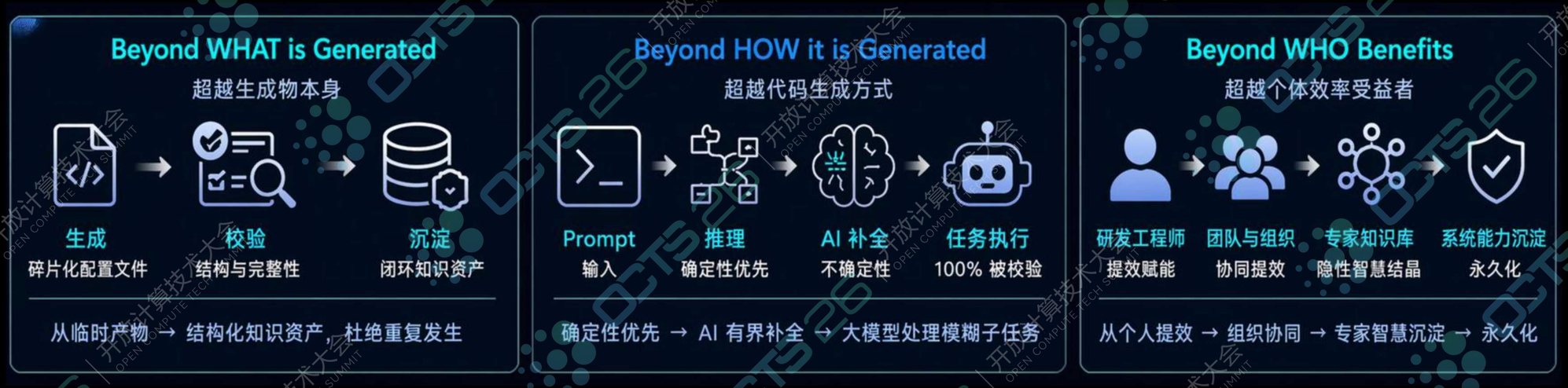
“我们需要的不是更好的代码生成，我们需要超越代码生成。”

AI 可以生成，但不能凭空理解硬件的真实世界规律。

我们选择：让 AI 在 人类经验 + 物理约束 的框架内，做有界的生成与推理。

理念重塑：超越代码生成与知识结晶

超越代码生成



🎯核心范式总结

大模型代码生成只能极其表象地解决“如何书写”；而我们的混合框架解决的是“如何获知、如何校验、以及如何永久留存”。

严格约束

🧭核心工程设计哲学 (Design Philosophy)

我们的思路是 “Determinism-First”，目标是让 AI 有可能产生幻觉的场景越来越少。

考量维度	行业主流：AI-First 路线	我们：Determinism-First 路线
发展目标	让 AI 承担得越来越多，AI 生成覆盖率无限扩大	让 AI 覆盖率越来越小（每次 AI 猜对结果自动固化为规则）
AI 的底层角色	绝对的主力。代码生成的首选入口与主要生产力	探索未知领域的‘桥梁/侦察兵’。用完自动退居二线
知识沉淀走向	沉淀在大模型的 Prompt 或者私有化微调模型参数里	析出并彻底结晶为独立于 AI 之外、可白盒审计的可执行代码
长期演进依赖	项目交付越来越依赖 AI 引擎的演进与参数调试	对 AI 的依赖呈断崖式递减。相同场景下次零 AI 介入
生成审计性	纯黑盒。无法反查生成这行配置的硬性硬件依据	强可信白盒。每行配置与注释均强行反查并对齐特定规则 ID

能确定的绝不猜，必须猜的严格约束，所有产物必须验证

可执行知识库

1

Stage 1: 口头传授与聊天流

隐性知识完全锁定在个人聊天和口头中。【痛点：人走知识丢】

2

Stage 2: 几百页 **Porting Guide**

指南繁复，没人看完，极易漏过。【痛点：没人看、内容易过期腐烂】

3

Stage 3: LLM 盲目直接生成

大模型盲目拼凑配置，速度快但常犯严重硬件错误。【痛点：幻觉多】

4

Stage 4: 可执行知识库 **(我们)**

规则即代码，约束不可绕过，AI 补位，新模式半自动回流沉淀。

❓为什么这不是“又一个规则引擎”？

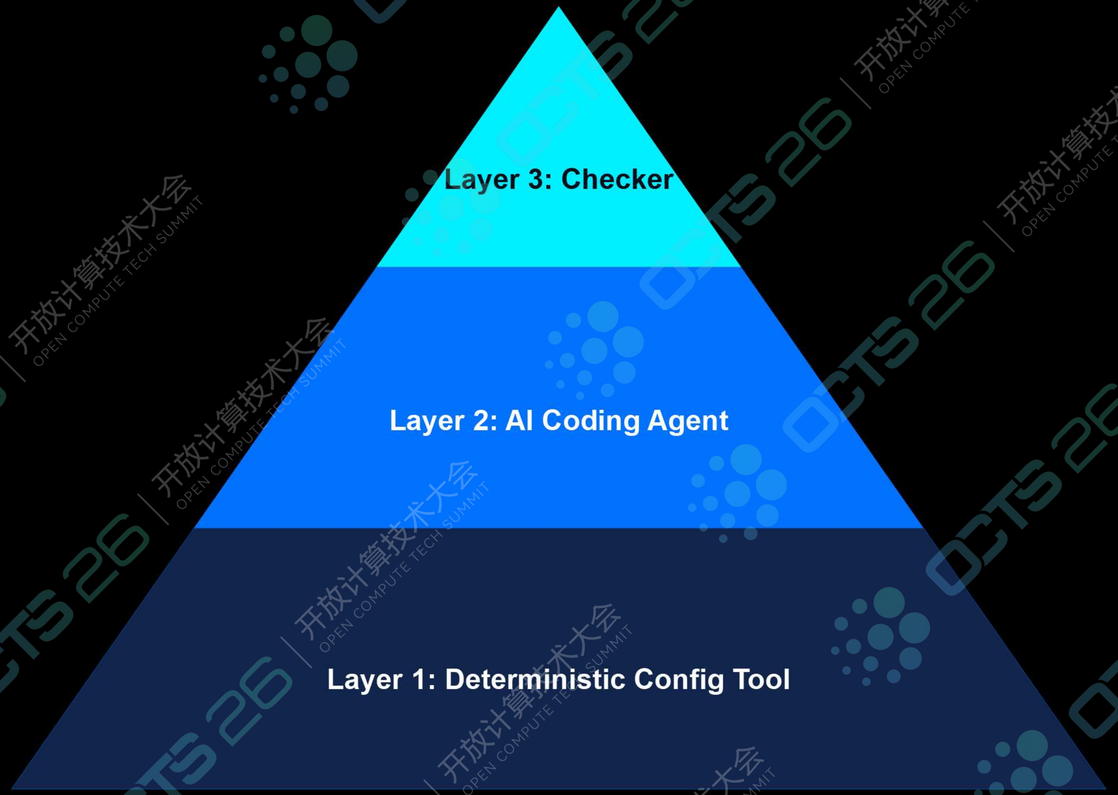
传统的规则引擎是**静态**的——需要依靠人工持续更新维护，往往极易腐烂、过期，沦为和 Porting Guide 文档一样的陈旧摆设。

我们的可执行知识库是 **“自生长、自结晶”** 的：

- AI 在围栏安全模糊地带不断探索，审核验证通过的新主板模式被系统自动捕捉，半自动化结晶、写回规则引擎库中。
- 知识库越用越大，规则越沉淀越充沛。大模型下次移植同类硬件时的 **“盲目猜测空间”便越缩越小**，彻底实现从液态（散乱隐性知识）向固态（代码硬规则）的不可逆转变。

这是 Knowledge Crystallization——知识结晶的真正奥义。

效率与确定性之间实现完美平衡



Layer 3 — RAG Checker | 封闭式检查 (100% 必经)

作为整个体系的绝对门闸。将生成产物进行封闭式的规则级 RAG 比对，完全杜绝大模型在时序和底层参数上的任何幻觉。

Layer 2 — AI Coding Agent | 模糊补位 (~20% 场景)

在精心构造的 Prompt Schema、Few-shot 历史注入等严格温室围栏约束下，处理长尾和非结构化胶水代码的辅助子任务生成。

Layer 1 — Config Tool | 可执行知识库 (~70% 场景)

固化所有无可置疑的物理因果和硬约束规则，提供零幻觉的 100% 确定性底层配置编译。

关键创新：这不是静态的三层架构，Layer 1 的地盘正随着结晶闭环而在动态、吞噬性增长！

技术原理：三层深挖

Agent 框架


Layer 3
RAG Checker
100% 必经验证
Layer 2
AI Coding Agent
~20% 模糊补位
Layer 1
Deterministic Config Tool
~70% 确定性规则
动态增长
Layer 3 — RAG 强制门闩
所有生成件 100% 闭环比对验证
幻觉无处逃逸
Layer 2 — AI 围栏补位
严格 Schema + Few-shot + 约束下
处理长尾子任务
Layer 1 — 可执行知识库
固化无可置疑的物理因果和硬约束规则
零幻觉确定性编译
Layer 1 地盘随结晶闭环不断吞噬性增长！

Layer 1 确保 稳定输出

⚙ 领域知识代码化

- **因果关系强制：** “如果采用主板电源砖，而非电源模块 PSU，则供电冗余策略无需配置。”
- **级联依赖校验：** “NVMe sensor 依赖对应 的I2C通道DTS配置上需要添加mctp相关属性。”
- **硬性物理防冲突：** “同一条 I2C 主线及其所有分支级联段下，硬件器件物理地址绝对无碰撞。”

🌐规则解耦与知识图谱 (Graph)

- **规则与模板高度解耦：** 规则引擎 + 模板机制，更新物理引脚或主板规范只需修改模板配置，内核机制不动。
- **拓扑依赖拓扑图 (Knowledge Graph)：** **规则并非零散平行，而是存在紧密的图依赖网络。** 这极具深度地构成了 “Porting Knowledge Graph” 白盒引擎，能自动追溯由于单点时序变动而带来的全局级联变动。

🎯与“代码生成”的本质界河

- **传统 AI 代码生成器：**
输入 → 大模型神经网络 (完全黑盒子) → 输出。
- **Layer 1 Config Tool 引擎：**
输入 → 确定性规则和 Knowledge Graph (完全白盒子) → 输出。结果每次可重现，过程绝对可解释、可合规审计、可直接行级 Git 修改。

Layer 2 聚焦局部代码生成

🛠️ 规则未覆盖的长尾场景

- **非结构化胶水脚本：**如特殊的初始化 init / 状态健康巡检 monitor 脚本，其逻辑因机型而异、千差万别。
- **自然语言 Spec 提取：**将 Vendor spec 等非结构化自然语言说明书直接提炼翻译，生成临时 config fragment 配置文件片段。
- **参数预测：**在部分信息缺失时，自主、合理地补全缺失引脚与时序默认值。

🛡️ 约束手段

- **结构化 Prompt Template：**固定严密的上下文指令框架与输出模式，禁止无边界的自由生成与对话。
- **输出格式强约束：**强制限定 JSON Schema 的硬结构要求，防止生成无法解析、超出预期的杂乱文字。
- **精准 RAG 注入：**选取真实、完全合规的主板历史配置实例做 Few-Shot 锚定方向。
- **Agent Skill 封装：**提供对工具手册和硬件映射策略的强读取权限。
- **Layer 3 闭环强制验证：**100% 封闭审查，编译/执行通过才放行，否则驳回。

🎯AI 的有限使命

“AI 在这里的定位不是绝对主力，而是探索未知边缘地带的**侦察兵**。它的真正任务是负责发现并把结果带回，固化沉淀为 Layer 1 规则。”

同时对Layer1 不方便支持的场合进行补位

“AI 的核心 KPI 不是生成了多少代码，而是消灭了多少个需要 AI 补位介入的模糊场景！”

Layer 3 RAG-as-Verifier 确保闭环可信

🔒所有生成件的最终门闸 Gate

任何由大模型参与生成、或是从长尾脚本补全出来的代码和配置文件，其输出物**绝不被直接采信**。

• 每一个生成产物，都必须无条件通过 Layer 3 这一绝对的封闭式闭环校验门控，否则一律驳回并重新纠偏，牢牢把控住交付代码的确定性底线。

📄100% 规则覆盖的静态检查

采用基于对齐规则的静态比对框架：

• 系统会调用精准的基础规则，对大模型生成的 JSON/YAML/DTS 配置文本进行语义差异分析，快速阻断有问题的配置

Innovation #1: Agent Tool

❌传统 Agent 集成方式的沉重包袱

- **适配成本昂贵：**需要对每个工具写大堆 API wrapper 封装和复杂 json schema schema 描述文件。
- **接口分裂严重：**必须分别维护供人类工程师使用的界面/CLI，和专供 Agent 调用的两套平行接口。
- **调试复杂度极高：**出错时根本无法辨析是 Agent 框架底层、接口层还是配置生成器内部的逻辑冲突。

⚡Agent-Ready CLI

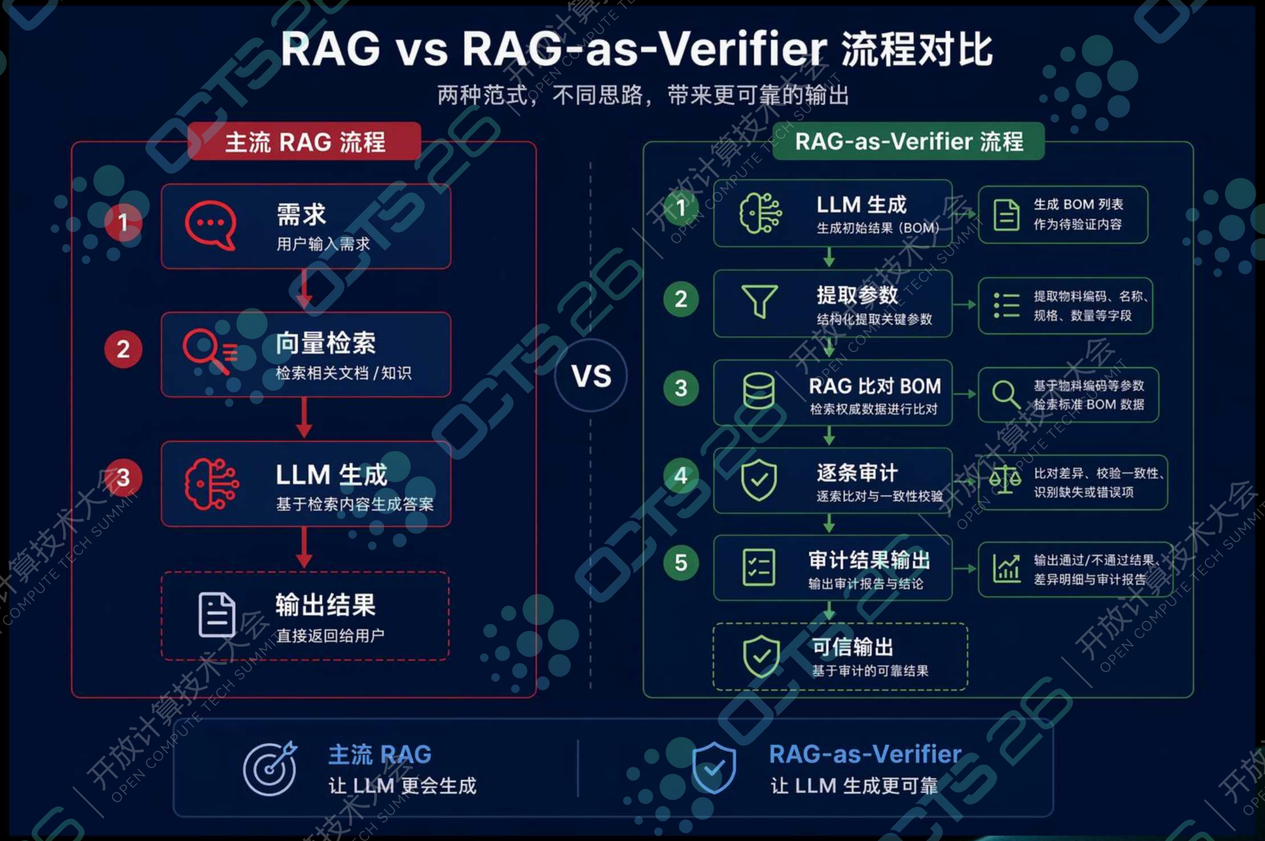
- **零适配接入成本：**配置工具本身就是标准的 CLI 命令行，Agent 直接调用命令行即可使用，适配工作量彻底清零。
- **接口天然高度一致：**人与 AI 智能体共用同一套命令行接口，不仅测试简单，更能保持长期的一致性。
- **原生支持结构化解析：**命令行天然原生输出整齐的 JSON/YAML 数据流，大模型智能解析提取畅通无阻。
- **极简的白盒调试：**只需直接调试命令行：**凡是人能够在终端跑通的命令，Agent 就 100% 能够顺利执行！**

💡重要工程启示：

如果开发一款全新的基础设施、固件或系统级排错工具，**务必设计成 CLI-first（命令行优先）**。

——**对于 AI Agent 而言，CLI 是最高效、最没有噪音的工具接口**。

Innovation #2: 从生成器到验证器



🔥从生成器到验证器：将 RAG 的主轴定位彻底反转

实践落地、效能飞跃与未来展望

双回路协同

“人类专家的反馈不再只是临时修补 Bug 的 Patch，而是我们整个系统运行的源泉。”

效果对比

指标	传统纯人工模式	业界常见：直接 LLM 生成	我们：Beyond Code Gen (Agent) 闭环
单次配置生成	数天（肉眼排错编写）	数分钟（但后续需要大量排错修正）	⏰ < 1 分钟（极速生成，90% 零人工配置错漏）
完整移植周期	数周（繁重的手工上电调试）	不可直接用于物理硬件生产阶段	⚡ < 2 人周（大幅缩短产品线上市时间）
配置正确率	~95%（由于配置零散难免遗漏）	~60% - 70%（受制于强物理幻觉）	🎯 >98%（Layer 3 强制防错与动态拦截）
输出稳定性	90%（依赖人类开发状态）	极度不稳定（单输入多次输出不同）	95%+（通过规则级白盒对齐引擎收敛）
可审计性	✅ 强（人类代码行行可见）	❌ 无（神经网络概率黑盒）	✅ 强（项目时序配置行级追溯至硬性 Rule ID）
知识留存率	随人员离职断崖式折损下跌	N/A（不沉淀组织知识）	100% 固留（规则即硬代码，沉淀在可执行知识库中）
跨项目知识复用	仅限少部分极易腐烂过期的静态 Wiki 文档	仅能复用已有的模型参数和静态提示词	100% 滚动自生长（自动析出，跨项目随时 0 成本调用）

🎯我们对“Beyond（超越）”的最终工程践行落点

“历史项目的所有探索与试错中积累的高价值工程经验，都已经彻底析出、并且写回的底层规则中。这就是知识复利的胜利。”

联结与共建

在固件领域探索高可靠的 AI Agent应用方法，共筑固件的智能未来。

THANKS

THANKS