信息·趋势·感悟
THE POWER OF COMMUNITY STARTS WITH YOU
美国2026RSAC热点研讨
暨第十八届信息安全高级论坛
INFORMATION SECURITY FORUM 2026

# 关于AI驱动自动代码审计的实践与思考

漏洞研究院 冯骁韬

# AI辅助漏洞检测方法发展演进

## 统计学时代（早期）

核心机制：
大规模特征匹配

能力缺陷：无法理解代码的上下文逻辑

## 机器学习时代（中期）

核心机制：
表征提取

能力进化：在复杂结构中识别非线性的漏洞模式

## 大模型时代（当下）

核心机制：业务逻辑理解以及语义审计

能力进化：通过语义推理的形式完成类似于人类代码审计的工作

# 目录

CONTENTS

PART 01

# Case Study
# “逻辑自洽”的幻觉

# Case Study - 一次由AI引导的漏洞审计



## Claude Code + Opus-4-6

**安全漏洞分析报告**

🔴 **存在高危路径遍历漏洞** (Path Traversal / Arbitrary File Write)

---
**漏洞位置**

文件: [redacted]
行号: **第 158-162 行**

```
158 |         if write_to is True:
159 |             write_to = self.model_cls.model_config.get("yaml_file")
160 |         if write_to:
161 |             with open(write_to, "w", encoding="utf-8") as fp:
162 |                 fp.write(template)
```

---
**漏洞详情**

**1. 缺少路径验证**

- 第 161 行直接使用 `write_to` 参数作为文件路径，**没有任何路径验证或清理**
- `write_to` 参数可以来自：
  - 方法调用时传入的外部参数 (`create_yaml_template(write_to="...")`)
  - 配置文件中的 `model_config["yaml_file"]` 值

**2. 攻击场景**

**场景一：直接调用攻击**

```
template = YamlTemplate(model_obj)
# 攻击者控制 write_to 参数
template.create_yaml_template(write_to="/etc/passwd")  # 覆盖系统文件
template.create_yaml_template(write_to="../../sensitive/file.txt")  # 路径遍历
```

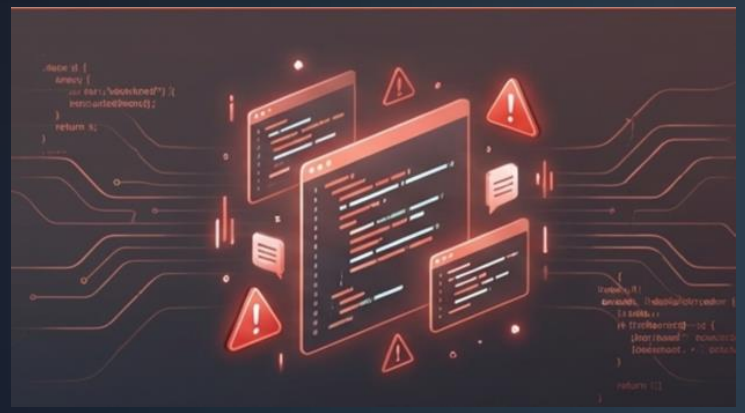
**场景二：配置注入攻击**

如果攻击者能控制 `model_config`（例如通过配置文件注入），可以设置：

```
model_config = {
    "yaml_file": "../../../etc/cron.d/malicious"  # 写入定时任务
}
```

# Case Study - “误报”

## AI的“假设”

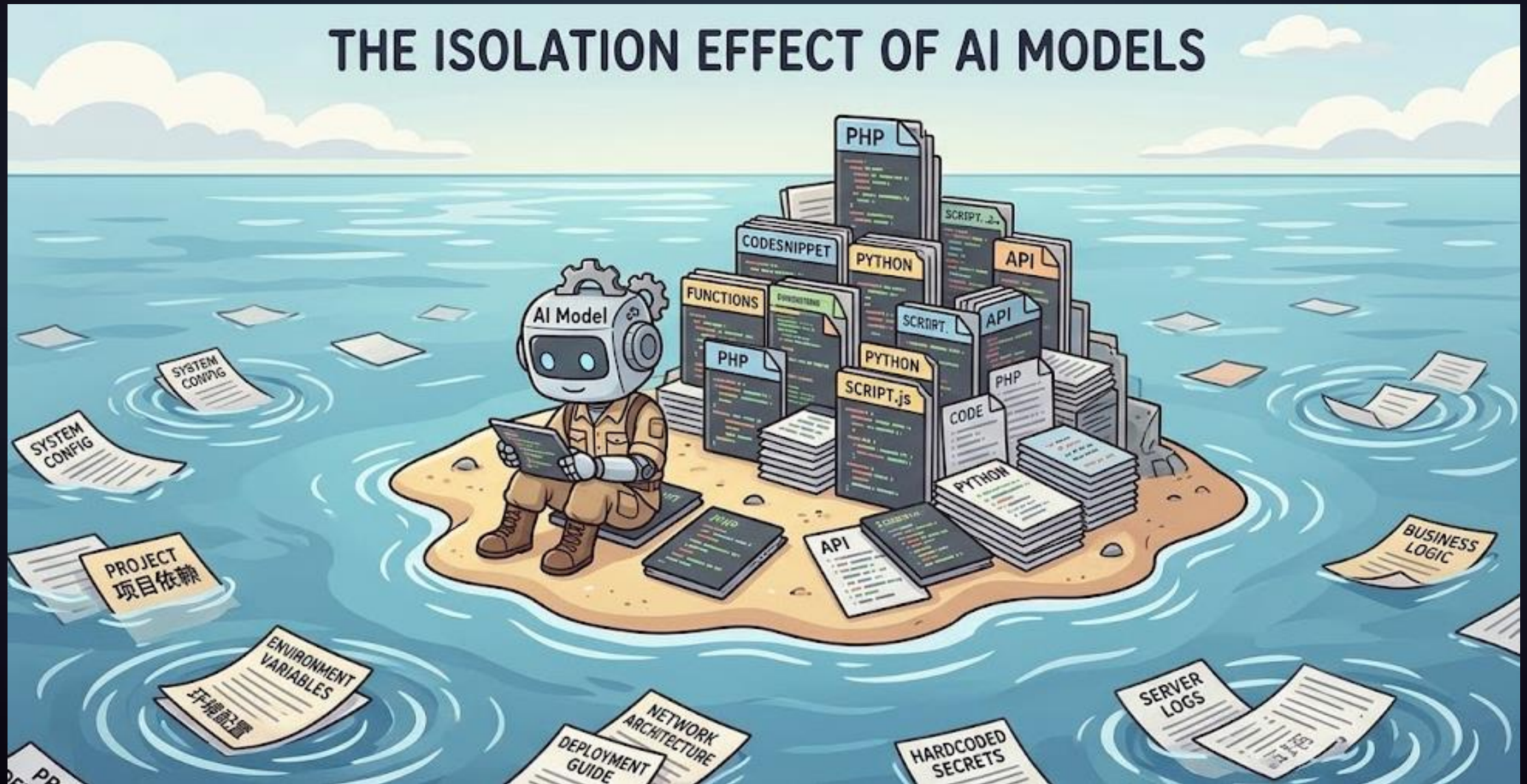


**视角**：纯代码级视野（语义推理）
**判断逻辑**：只要是外部输入，一律视为高危
**结果**：为了无“漏报”，产生“误报”

## 真实的部署环境

**视角**：架构与环境视野
**真实环境**：系统运行在受控的隔离环境
**事实**：该环境变量由项目静态注入，攻击者在网络层面上无法进行干预

# Case Study - “盲区”的本质 - AI模型被困于代码孤岛上



PART 02

# AI代码审计落地的难题

# 环境语义的结构性缺失

**现状**：AI无法识别到跨模态的输入是否受控（如只读环境变量或者内网透传参数等）

**判断逻辑**：安全评估结论严重脱离业务现实，审计报告失去可信度

# 落地的难题（二） - 通过提示词修复问题

通过修改提示词来提醒Agent在判断漏洞时需要注意部署环境

“在漏洞研判中,请考虑项目的部署情况”

Agent能够通过扫描项目的文档以及环境配置来思考外部攻击是否可以实现

---

🟢 **实际安全状态：基本安全**

**关键事实：**

1. **方法调用场景**
   - ✅ 只在 **CLI 初始化命令** (`python cli.py init`) 中调用
   - ✅ 只在 **脚本直接执行** 时调用
   - ❌ **没有任何 API 端点**暴露这些方法
2. **参数来源**
   - ✅ `write_to` 参数完全**硬编码**为 `True`
   - ✅ 文件路径来自内部配置 (`model_config["yaml_file"]`)
   - ❌ **无法从用户输入控制**
3. **部署场景**

用户 → HTTP请求 → FastAPI路由 → [业务逻辑]
↓
[不会调用create_yaml_template]

管理员 → SSH/本地 → python cli.py init → [可以调用，但这是正常使用]

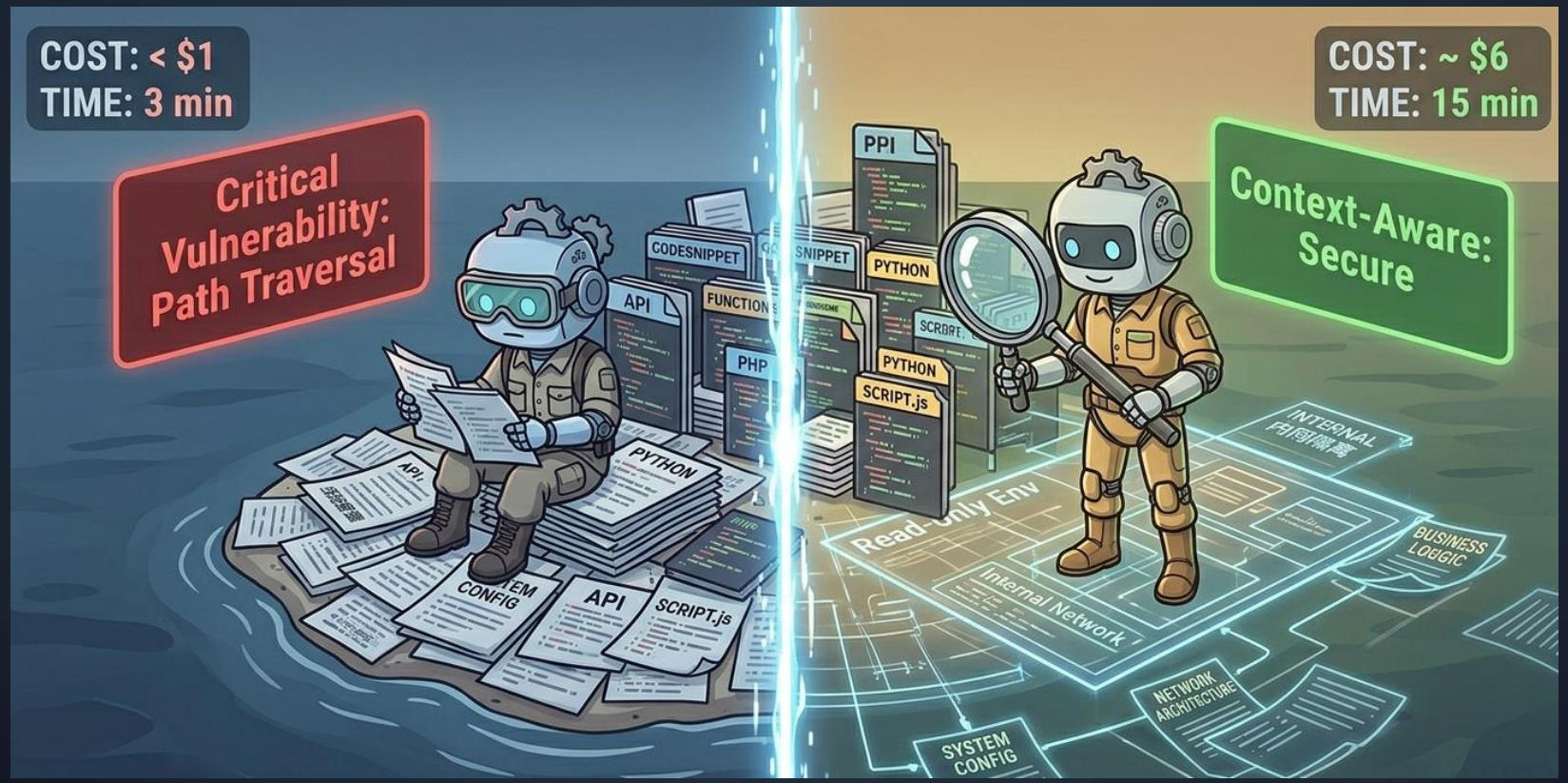
---

**修正后的风险评估**

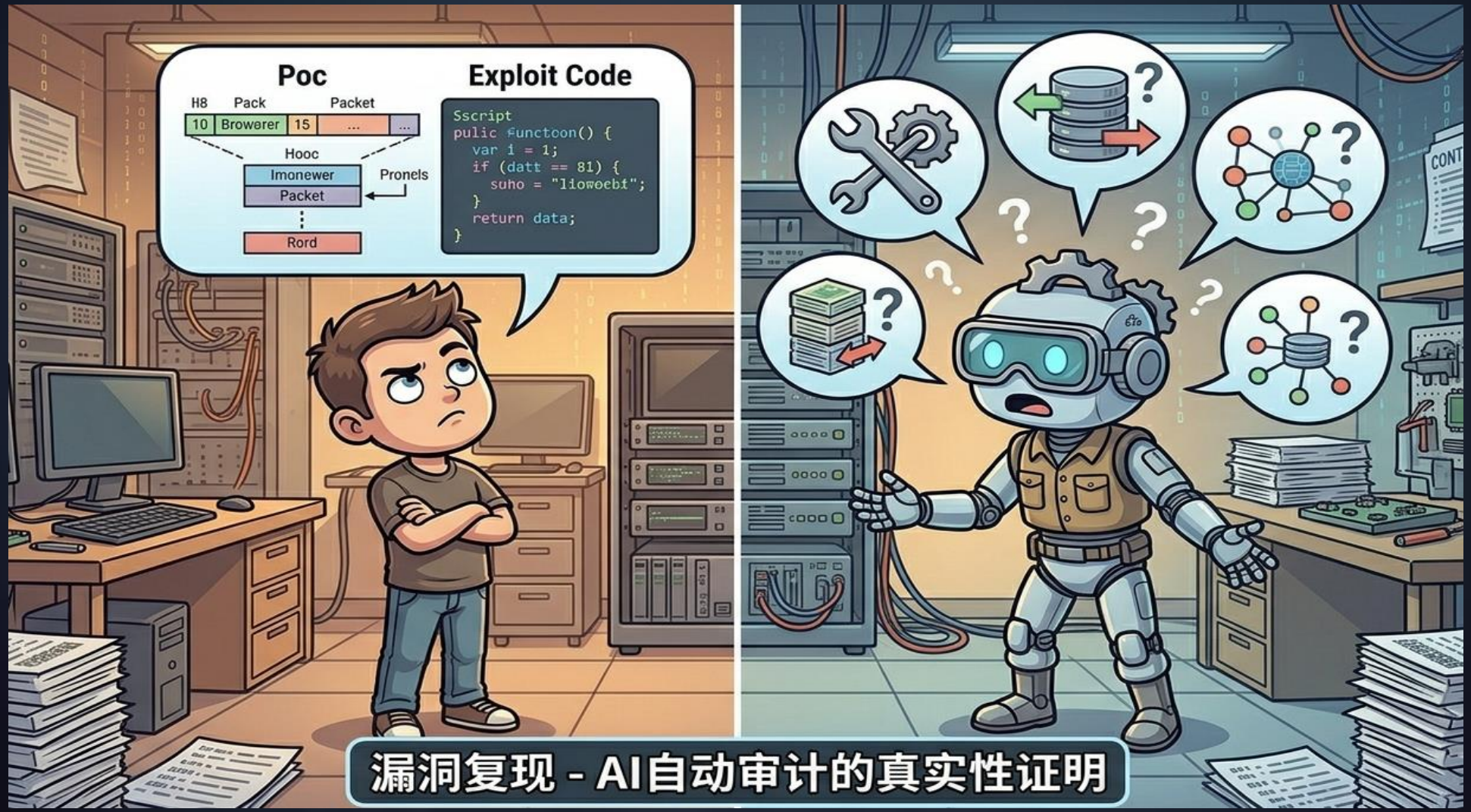
| 攻击向量 | 可行性 | 实际风险 |
|---|---|---|
| **远程攻击** | ❌ 不可能 | 🟢 无风险 |
| **API注入** | ❌ 方法未暴露 | 🟢 无风险 |
| **配置文件篡改** | ⚠️ 需要服务器访问权限 | 🟡 低风险 |
| **环境变量污染** | ⚠️ 需要容器/系统权限 | 🟡 低风险 |

# 落地的难题（二） - 资源陷阱

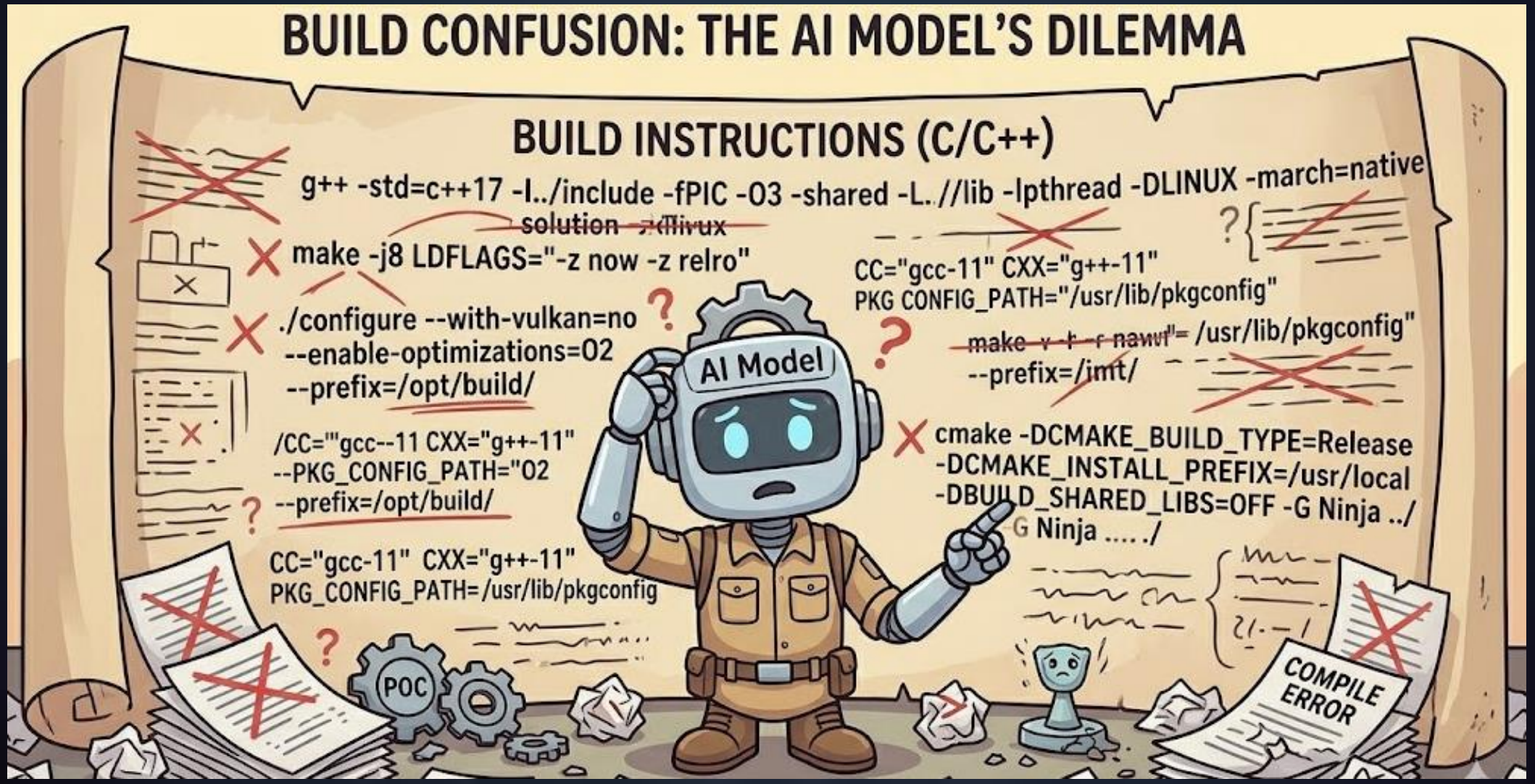
## 让Agent对项目进行全面分析的代价是高昂的资源消耗



# 落地的难题（三） - PoC生成需求



# 落地的难题（三） - PoC生成难题



PART 03

# 方法论转型
# 与
# Harness Engineering

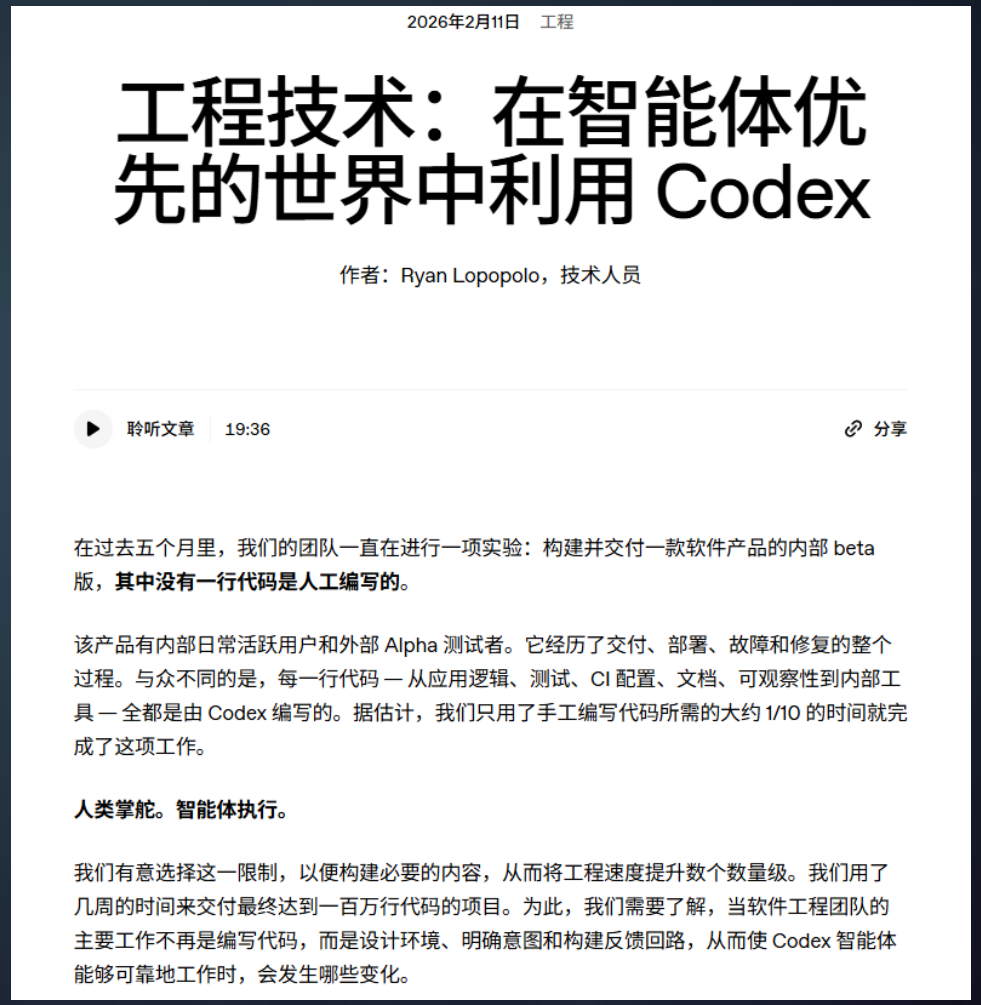
# Harness Engineering - Openai团队工作的启示



Openai团队将强大的模型比作拥有无限潜能但是天性狂野难以控制的“烈马”

而Harness则是为这匹烈马量身定制的“马具”

Harness Engineering的本质就是旨在将模型的能力转化为受治理的可靠的行动



2026年2月11日 工程

## 工程技术：在智能体优先的世界中利用 Codex

作者：Ryan Lopopolo，技术人员

聆听文章 19:36 分享

在过去五个月里，我们的团队一直在进行一项实验：构建并交付一款软件产品的内部 beta 版，**其中没有一行代码是人工编写的。**

该产品有内部日常活跃用户和外部 Alpha 测试者。它经历了交付、部署、故障和修复的整个过程。与众不同的是，每一行代码 — 从应用逻辑、测试、CI 配置、文档、可观察性到内部工具 — 全都是由 Codex 编写的。据估计，我们只用了手工编写代码所需的大约 1/10 的时间就完成了这项工作。

**人类掌舵。智能体执行。**

我们有意选择这一限制，以便构建必要的内容，从而将工程速度提升数个数量级。我们用了几周的时间来交付最终达到一百万行代码的项目。为此，我们需要了解，当软件工程团队的主要工作不再是编写代码，而是设计环境、明确意图和构建反馈回路，从而使 Codex 智能体能够可靠地工作时，会发生哪些变化。

# 方法论的改变（一）

## 方法论的改变

- 从“模型驱动”转向“工程约束“：
  - 核心公示：智能体 = 模型 + 驾驭系统
  - 底层模型是通用的，而决定审计质量的“护城河”是为其量身定制的约束系统。我们应该通过设计精密的CAR架构（Control-Agency-Runtime）来引导AI

- 破除”代码孤岛“ - 仓库即现实（Repo-as-truth）：
  - “凡是模型在运行时无法访问的东西，对它而言就是不存在的”
  - 必须将所有的“隐性语义”显性化。除了代码，应将项目文档、设计文件、部署架构、环境变量约束以及历史补丁等全部写入知识库并进行索引

# 方法论的改变（二）

## 方法论的改变

- **建立确立性的反馈闭环：**
  - 在方法论的各个环节中AI都会出现幻觉从而引起错误。
  - 利用Runtime（运行时层）的监控机制建立自愈循环（self-healing loop）。例如，当AI生成PoC因编译报错或者断言不符合事实而失败时，系统应自动捕获错误日志并回传给系统，要求模型基于“错误经验”重新推理修正认知。

- **状态外部化与成本控制：**
  - 在对一个项目的审计过程中，一些背景以及逻辑意图的推理可能会在不同的case中复用，重复消耗推理token
  - 需要建立项目知识库实现状态外部化（State Externalization）。例如将AI对项目环境和逻辑意图的前期总结持久化存入知识库，避免每一轮审计重复进行高昂的初始化推理，从而降低模型的消耗。

## 基于harness engineering思想的CAR架构模型

- CAR模型的核心目的是为了实现从“模型驱动”到“系统可靠”的范式跃迁。

- CAR模型的构成:
  - **控制层（Control - C）**：包含在任务开始前塑造行为的持久化制品。者包括项目指令（如AGENTS.md）、架构规则、代码检查器（Linter）、权限政策和验收标准。本质上是将“人类判断经验”转化成“机器刻度约束”的地方。
  - **代理层（Agency - A）**：定义模型被允许如何行动，即行动基质。这包括受控的工具集（API）、执行环境（如代码沙盒、浏览器）、以及分工结构（如多智能体协作模式）。
  - **运行时层（Runtime - R）**：管理任务随实践推移的执行稳定性。这包括上下文压缩与外部状态化（如建立知识库）、重试与回滚策略、审计轨迹追踪以及预算控制。

# Control（控制层）：建立“仓库即现实”的契约

## 过去（隐性知识）

部署假设、网络架构规则散落在开发者大脑或零散文档中，AI无从知晓。

## 现在（显性契约）


将非形式化的“部署假设”转化为机器可读的 Control Artifacts。

## 核心动作拆解：

1. 扩大输入面：从“仅读取代码”扩大到“摄入架构设计文件、历史补丁与网络讨论”。

2. 定义信任边界：强制配置控制层组件，明确告知AI：哪些变量是天然安全的？系统运行在什么隔离环境中？

3. 结果：彻底消除由于“环境语义缺失”导致的低级幻觉与误报。

# Runtime（运行时层）：状态外部化与反馈自愈

## 状态外部化（State Externalization）

- 建立持久化的“项目知识库”。
- 动态记录 Agent 对环境和逻辑意图的阶段性总结。
- 目的：避免AI每一轮审计都“从零重新推理”，彻底根治Token“吞金兽”问题。

## 自愈闭环（Feedback Loop）


- 承认幻觉：从工程上接受模型首次输出的 PoC 大概率无法直接编译。
- Runtime 监控：精准捕获 PoC 运行时的编译报错、依赖缺失或逻辑冲突。
- 反馈驱动：将真实的错误日志作为高价值反馈信号回传，强行驱动 AI 引擎进行自我修正与迭代。

PART 04

# 总结

# 总结

- 核心观点：Agent = Model + Harness
- 视角转变：不再把Agent当作独立的“专家”，而是将其作为整个“安全审计系统”中的一个特定引擎
- 安全审计系统的CAR架构：
  - Control： 定义边界，引入文档、Patch 及设计文件，划定环境信任边界
  - Agency： 赋予能力，引擎化分工，多工具（SAST、Linter、沙盒）联动
  - Runtime：确保持续化运行，状态外部化（知识库）解决成本问题，自愈闭环（日志回传）解决幻觉问题。
- 未来的核心竞争力不再是如何写好Prompt，而是构建严谨、可审计、自愈的Harness Engineering体系

# THANKS!