Agentic AI Coding × Rust

Growing computational–physics code you can verify

Hiroshi Shinaoka

Saitama University

事前准备

事前准备 (每人, 课程前完成)

1. 安装一个 CLI 型代理.
   - 免费: Gemini CLI / ChatGPT Plus 及以上: Codex / 其他工具也可以.
   - 安装指南: Satoshi Terasaki / Junya Ito.
2. 准备一个编程语言工具链.
   - 今天使用 Rust , 但其他语言也可以. 若使用 Rust, 安装 Cargo (参考).

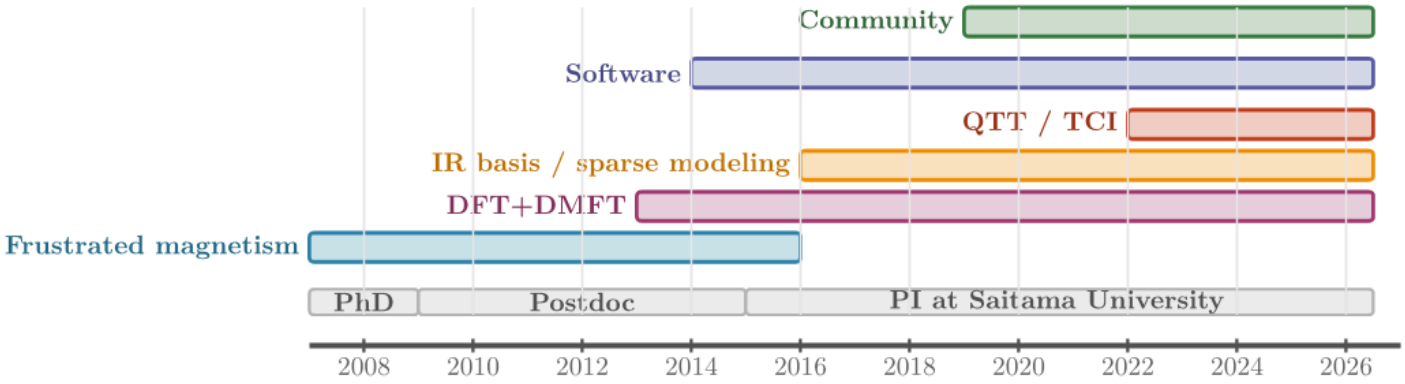
今天的流程

- 前半 = 方法论 / 后半 = 二维 Ising 实操.
- 当天只做 环境检查: 主要时间用于方法论和实操.
- 二维 Ising 是经典问题, 因此即使免费模型也应能跑完整流程 (大概).

导入

自我介绍与今天的立场

- 东京大学 PhD → 瑞士博士后 → 埼玉大学 PI (2015∼)
- 从量子多体计算到第一性原理计算的广泛经验
- IR basis / sparse modeling / tensor networks
- **C++ / Python / Julia / Fortran** 的实际开发经验
- 参与 OSS 与社区贡献



现在我一行代码也不手写, 也不逐行阅读生成代码. 即便如此, 我仍然认为 代理式编码比手动编码更可信. 为什么?

常见批评: “AI slop”

有人把代理式编码的产物称为 “AI 垃圾产物”:

工匠性的丧失
Sinclair Target

“…would prefer not to use agentic tools even if they worked as advertised.”
即便工具能工作, 人也会失去对工作的关心. Quality in the Age of Slop, 2026

维护崩溃
D. Stenberg (curl)

“The current torrent of submissions put a high load on the curl security team…”
curl 被 AI slop 淹没, 关闭了 bug bounty (2026). The Register

共享资源劣化
Baltes et al.

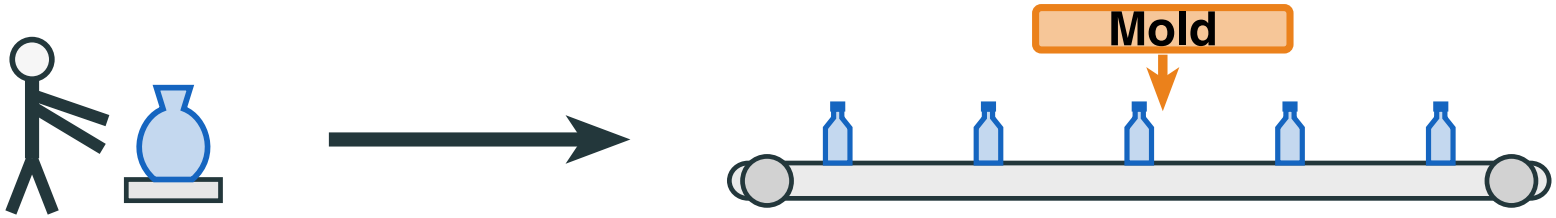
“…a tragedy of the commons, where individual productivity gains externalize costs onto reviewers, maintainers, and the broader community.”
“An Endless Stream of AI Slop”, arXiv:2603.27249

过度宣传的反作用
Anthropic C compiler

用 16 个并行 Claude 实例构建 C 编译器 (100K 行, $20k). 独立基准测试显示它 快于 GCC −O0, 但仍落后于 GCC −O2. 评价目标不清楚时, 这类展示会 扩大对 AI 的怀疑.
Anthropic · benchmark

优化不足 并不是 AI 做不到的证据 它说明缺少性能 oracle, 也说明 human–AI loop 不够紧密.

编码的工业革命: 从手工业到工厂



人类逐件精心制作

= 手动编码

用模具大规模生产 (高速 · 均质)

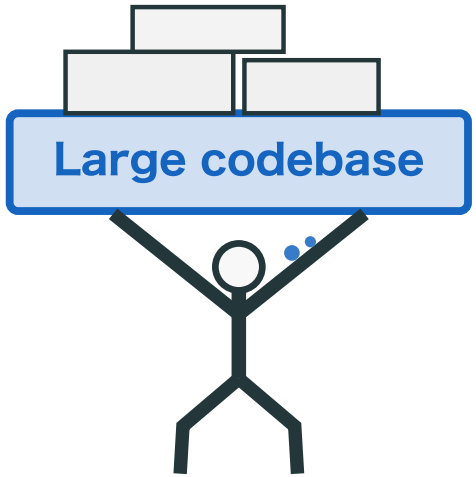
= 理想的代理式编码

同样的转变正在 代理式编码.

我们正处在思考如何实现 理想的代理式编码.

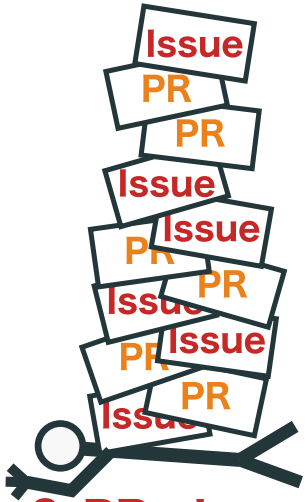
手动（工匠模式）编码的局限

Scientific computing



Held up by a few artisans
(PD / staff leave → unmaintainable)

Large project



Issues & PRs keep piling up
(manual review can't keep up)

→ 两者都是 依靠人力支撑 模式的局限.

今天的主张 (概要)

我把信任的放置点从 目视检查 转移到 机械化 · 外部验证:

Compiler · test · oracle · rules 取代目视, 支撑正确性.

底层观点: AI 的高速生成是工业化大规模生产. 与其逐个修产品, 不如修 模具 (= 正本: rules · oracle).

编码代理的基础

聊天型代理 vs CLI 型代理

聊天型 (浏览器对话)	CLI 型 (终端)
提问后复制粘贴答案	直接编辑并运行文件
不能访问本地环境	操作 repo 与 git
每一轮都由人搬运	自主执行多步任务

CLI 型代理是 面向长时间自主执行的设计. 这是今天的主角.

按代际来说: Gen1 (Cursor / Copilot) → Gen2 (Claude Code / Codex / …). MIT missing-semester: 代理式编码

代表性的 CLI 代理（执行框架）

Claude Code	Anthropic. 在终端/IDE 中自主编码
Codex CLI	OpenAI. Rust 制 OSS, 随 ChatGPT 计划提供
Gemini CLI	Google. OSS, 有免费额度
Kimi CLI	Moonshot（中国）. 模型 K2.7 Code
OpenCode	OSS, 不绑定模型提供方
Pi	极简 OSS. 可自行添加扩展和技能来培养它（适合极简主义者）

执行框架 ≠ 模型

- 执行框架 = CLI 工具 (编辑文件, 执行命令, 调用工具的框架).
- 模型 = 底层运行的 LLM. 许多执行框架允许 替换模型.
  - 中立: OpenCode / Gemini CLI · 偏自家集成: Claude Code / Codex.
- 性能由 模型 决定, 可用性和自主度由 执行框架 决定. 两者都要选择.

订阅 vs API, 以及低价模型

订阅	Claude Pro $20/月 (含 Claude Code) / ChatGPT + Codex. 固定月费 + 使用额度
API 按量付费	按使用 token 计费. 可把任意模型接到执行框架

低价编码向 API (2026): DeepSeek V4 Flash (输入 $0.14 / 输出 $0.28 每 1M token) · Kimi K2.7 Code. 例如: OpenCode + 低价 API 可以低成本开始.

价格和模型名截至 2026 年 6 月. 世代更新很快.

什么是技能

- Skill = 代理的 操作手册 (markdown). 它把抽象任务拆成明确的小步骤.
- 它接近程序中的函数, 但操作对象是 代理的行为, 而不是数据.
- 它会在相关工作中 自动触发, 技能也会调用其他技能来组合复杂流程.
- CLAUDE.md / AGENTS.md = 跨会话保存的记忆 (规范, 命令, 结构).

实际工作流的样子

superpowers 技能强制 brainstorm → plan → execute (各阶段自动触发):

- 起点不是 prompt, 而是 带数学依据的设计文档. 同时设计验证 (oracle / test).
- 实现后让 AI 输出 code ↔ algorithm/math 的对应关系并核对 (有时完全不读代码本身).
- Execute 阶段是 TDD + 子代理分工; Review 发现问题则回到前一阶段.

过去一年: 把代理式编码推到极限

我是如何走到这里的

自 2025 年 10 月以来, 我没有手写过一行代码. 也不逐行阅读生成代码.

2025 年 1 月	开始使用 Cursor, 仍是看源代码的世代
2025 年 10 月	开始将 SparseIR.jl 从 Julia 移植到 Rust
2025 年 12 月	切换到 Claude Code: 停止逐行阅读生成代码
2026 年 1 月	开始 tensor4all-rs: Julia TN 生态的 Rust 移植
2026 年 2 月	开始 tenferro-rs: Rust 制 PyTorch 式张量栈
2026 年 3 月～	现在主要使用 Codex

做出的东西: tensor4all-rs

将 Julia 的 tensor learning stack 移植到 Rust.

TensorCrossInterpolation.jl / QuanticsTCI.jl etc. / SciPost Phys. 18, 104 (2025) / tensor4all

GitHub: github.com/tensor4all/tensor4all-rs

- 2026 年 1 月 1 日开始 (原本是寒假中测试代理式编码“极限”的自由研究)
- 少数人类 + AI (从 Claude Code 到主要使用 Codex)
- 2 个月 353 commits · 最初 2 周 +61,486 行 (151 个文件)

→ 合作者之间关于 AI slop, 教育法, Julia 是否仍然必要的讨论仍在继续.

“经过大量试错和失败, 正确的代理式编码方法才刚开始显现.”

设计, 测试与验证, 合适的语言, (教育法)…

有意的实验: 把代理式编码推到极限

“观察失败, 并寻找控制失败的方法.”

原来的 Julia 生态很大, 覆盖 Quantics Tensor Train (QTT) / Tensor Cross Interpolation (TCI). 其中一部分依赖 ITensors.jl.

最初从“单体结构”开始: 移植 ITensors.jl 中需要的部分 (index system, tensor contraction) + QTT/TCI.

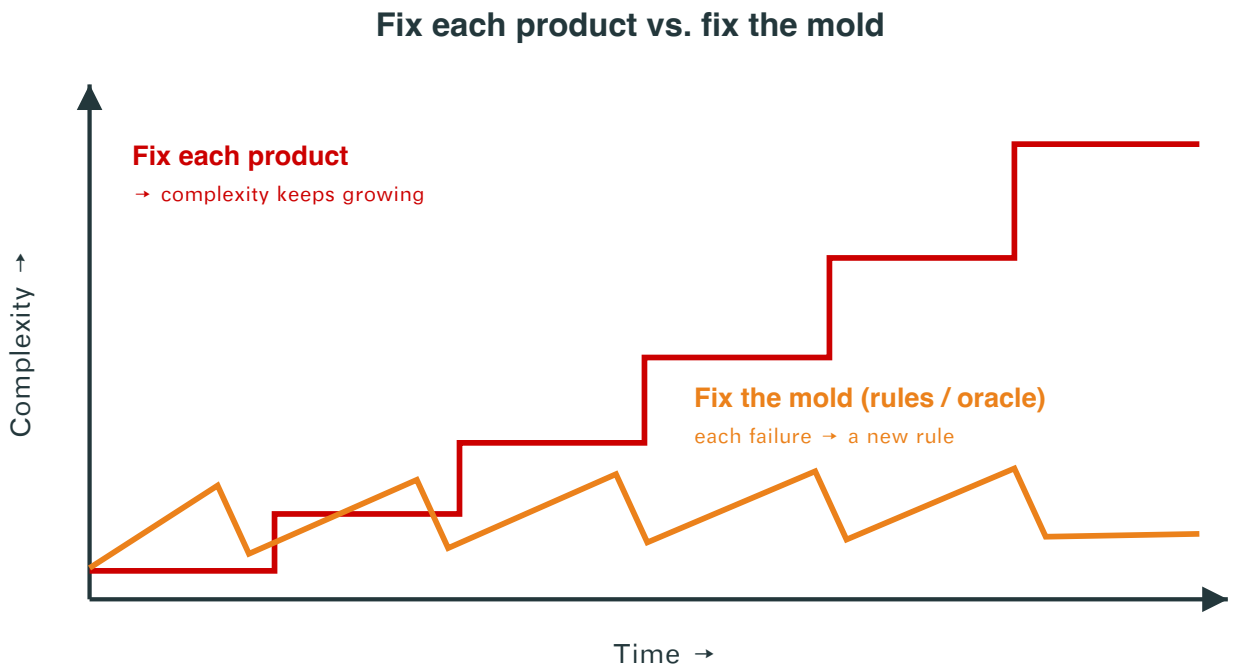
失败例

(如预期) 出现了大量问题

- 破坏抽象: 在高层导入并使用低层功能 (临时修补)
- 悄悄放宽单元测试阈值 (放弃修 bug)
- 不用 BLAS, 而是写 nested loops (结果正确, 但是...)
- 单个源文件超过数千行
- ...

复杂度持续增长

- 逐个修补跟不上.
- 为了长期维护, 只能 **培养一套统一的** 思想和设计规则.
- 否则复杂度会持续增长.



修正“规则”, 而不是“代码”

参考

- Heinrich pyramid: 一个安全工程经验法则, 即一个重大事故背后有许多轻微事故和 near miss.
- 航空事故调查: ICAO Annex 13 将调查目的定义为预防, 而不是追责.

SKYbrary: Heinrich Pyramid · ICAO Annex 13 · NASA ASRS

我在代理式编码中的对策

- 通过建立“规则”和“制度”来矫正 AI 的行为
- 构建机械化审计和批量修正代码的机制

(这种机制在手动编码时代也一直很重要.)

→ 兼顾 AI 压倒性的高速代码生成与质量控制

单体 → 分离结构 (用结构强制约束)

把巨大的 tensor4all-rs 拆成职责不同的独立层.

Tensor4all.jl
github.com/tensor4all/Tensor4all.jl
面向人类的接口 (Julia, 兼容 ITensors)

tensor4all-rs
github.com/tensor4all/tensor4all-rs
张量网络: TreeTN / QTT / TCI

tenferro-rs
github.com/tensor4all/tenferro-rs
通用张量计算 + 自动微分 + GPU (类似 PyTorch/JAX)

tidu-rs
github.com/tensor4all/tidu-rs
通用自动微分引擎

一旦分层完成, AI 只能在各层内部活动, 不能破坏层次结构.

单体 → 分离结构 (用结构强制约束)

层和 API 边界由 AI 一起分析 PyTorch / JAX 后设计. (Jin-Guo Liu: tenferro 初期设计, Satoshi Terasaki: 开发支持, tensor4all collaboration: tensor4all-rs / Tensor4all.jl 开发)

各层内部的模块化

tenferro 内部也被模块化.

No root facade. Each family is a first-class crate.

在 Rust 中可以沿模块层次细粒度控制符号可见性. AI 编码代理不能从外部擅自 import 私有符号.

可验证性优先于可读性

以下讨论以“大型代码库”的开发为前提.

语言选择的理由发生反转: 我自己的说法变化

2023 年 3 月 (那霸, 计算物理春季学校 2023 前夜祭)

最近我常用 Jupyter Notebook 做原型. Julia 的代码看起来接近数学, 易于检查, 内存管理也处理得很好, 因此很舒服. 我也对 Rust 感兴趣, 但对一般学生来说学习门槛高, 数值计算库也还不够成熟.

2026

现在我不再用 Jupyter Notebook 做原型. Rust 的内存管理严格而显式, 对代理式编码更高效. 代码由代理来写, 因此陡峭的学习曲线不是问题. 在 AI 帮助下, 即使代码稍长, 保持代码与数学一致也不是问题. 缺失的数值库可以移植到 Rust.

可验证性优先于可读性

手动编码时代: 各语言带来的“好处”

Fortran	数组是语言原生能力 · 手写循环也快 · LAPACK / BLAS 触手可及
Python	交互式试验 (REPL / notebook) · numpy / scipy / matplotlib · 立即可视化
Julia	记法接近数学 (broadcast / Unicode) · JIT 带来速度 · 原型接近生产代码

共同点: 它们都降低了 人类手写, 阅读, 维护 代码的成本.

选择语言的指标变化: 大规模开发

过去重要的指标都默认 **人类写代码并维护代码**:

传统指标	**代理式时代**
可读性: 逐行跟踪	可验证性: 用 oracle 检查
平缓的学习曲线	陡峭部分由 **AI 接管**
接近数学: 目视验证	与机械化检查结合
书写速度	不再是瓶颈

容易检查 优先于 容易阅读.

可读源码 ≠ 可检查实现

可读 和 **可检查** 是不同的事情.

例: 即使 Julia 的 broadcast 赋值 **看起来** 接近数学, 也无法仅凭外观确定行为:

```
b = a          # b and a point to the same array (alias)
a .+= c        # if updated in-place, b changes too ← side effect
```

aliasing / mutation / allocation 无法从一行代码的外观中看出.

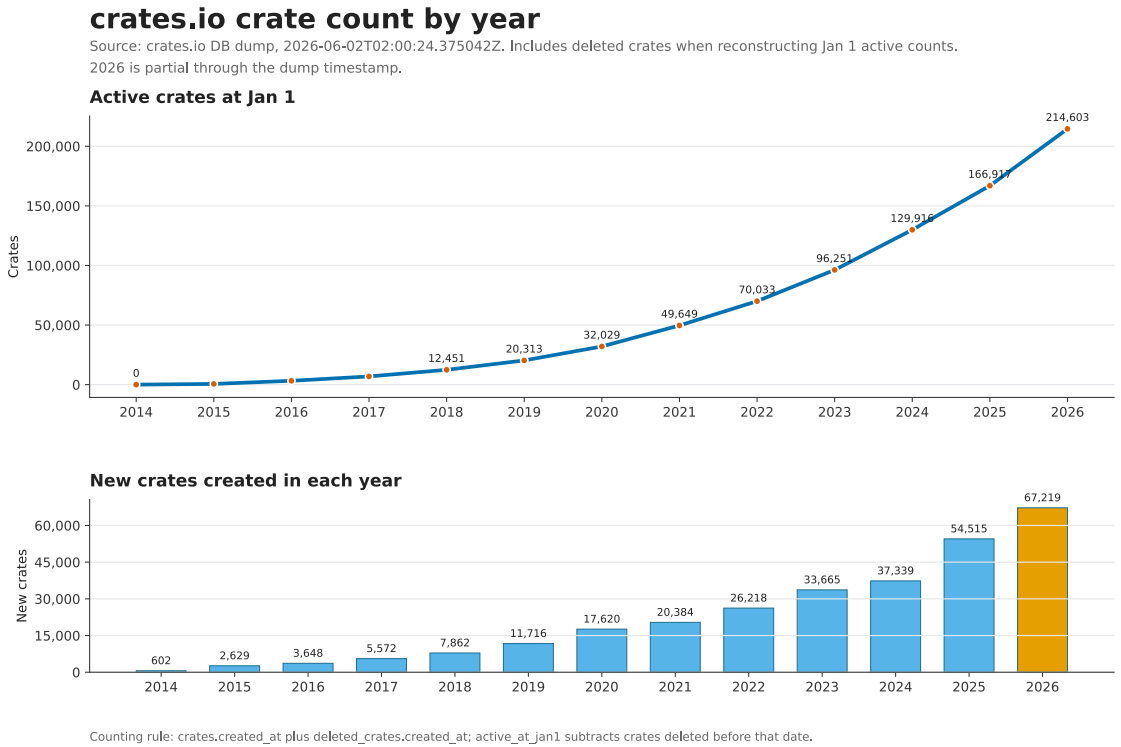
到这里为止与语言无关. 那么哪种语言适合可验证性?

Rust 是什么 (面向本听众的最低限度)

- 来自 Mozilla 的系统语言. 为 Firefox 实现而开发, 2015 年发布稳定版 (1.0).
- 因可靠性受到重视, 已被 Linux kernel 正式采用 (v6.1 以后, C 之后的第二种语言).
- 所有权: 没有 GC 的内存安全. 数据竞争和悬垂指针在 **编译时** 被排除.
- Cargo: build, 依赖解析, testing, benchmarking 被标准化整合.

给物理学家的一句话: "安全的 C++ + 一流的 build/dependency manager".
速度是 C / C++ 级, 安全性是 GC 语言级.

生态系统的加速增长



2026 年 NVIDIA 也发布了 Rust 的 CUDA 内核编译器 `cuda-oxide` (实验阶段).

Why Rust? 与 AI 的快速反馈循环

Python / Julia 是为了 **快速书写** 而设计的. 在 AI 时代, 这个优势会 **反转**:

Python / Julia	Rust + AI
手写速度快	AI 来写: 速度相同
错误在运行时暴露	编译时捕捉
验证需要执行	`cargo check` 数秒完成
AI 的错误发现较慢	AI 的错误被 **立即** 发现

*这里假设大型代码库.

Zenn: Why I Migrated from C++ to Rust

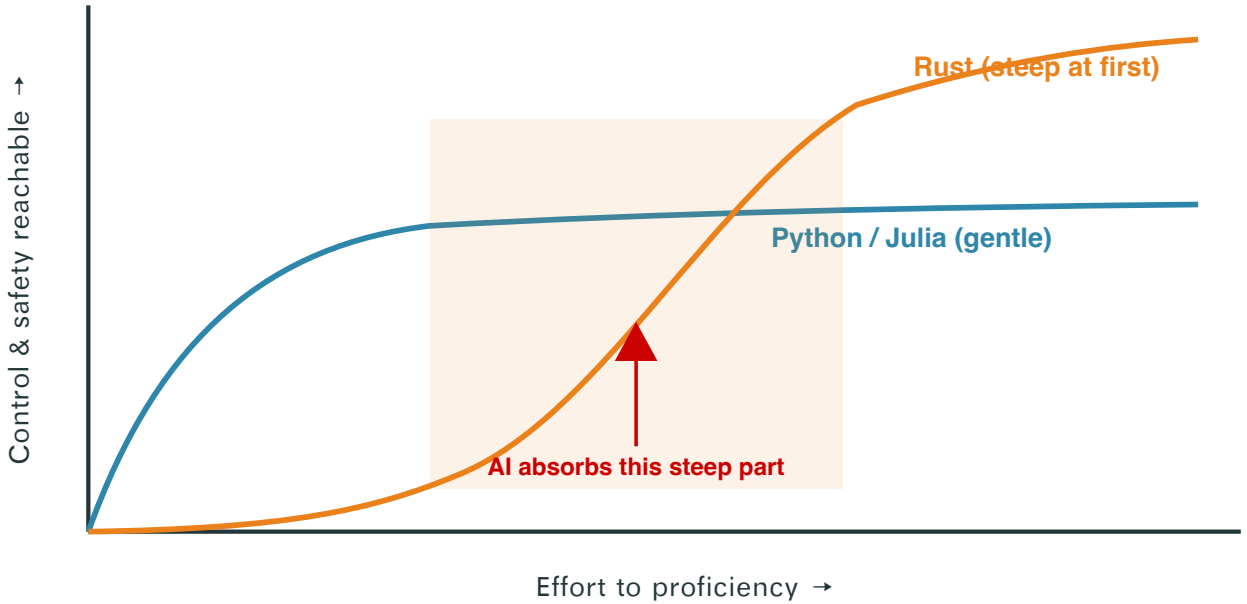
个人经验

- tenferro-rs + 外部依赖从零构建约 2 分钟 (Macbook Pro)
  edit→test 数十秒完成, cargo check 几乎瞬时.
- AI 处理所有权 / 生命周期的 **机械复杂性**
  → 人类可以集中验证 算法, 设计, 正确性.
- Cargo 解析 pure-Rust 依赖. 不需要 CMake.
  链接时没有版本冲突.

个人而言, 我在 Julia/C++/Python 中感到的“代码变大后验证变难”的不安消失了.

tensor4all-meta/docs/why_rusty_julia.md

学习曲线: AI 接管陡峭的初期部分



只剩下 Rust 最好吃的部分.

我组里的学生也在从 Julia 迁移到 Rust.

代理式编码时代的工作流: 语言可以混用

- 以前: 在 Jupyter Notebook 中交互式尝试.
- 现在: 用自然语言指示 AI
  生成代码 → 计算 → 保存结果到文件 → 绘图
- 为了原型开发, Jupyter Notebook 不是必需品.
- 不需要把所有计算放在单一语言里. 可以利用既有资产, 同时 逐步迁移到 Rust.

小结

选择基准从 最小化认知成本 转向 最大化可验证性.

用三根支柱支撑

用三根支柱支撑巨大且高功能的代码库

tenferro-rs 约 13 万行, 是处理 einsum · FFT · 自动微分 (AD) · GPU 的高功能栈. 逐行 review 不可能, AI 的设计选择也会每次摇摆. 不是让人类站岗, 而是用三根支柱支撑.

tenferro-rs (130K lines)

einsum · FFT · 自动微分 (AD) · GPU …

支柱 1
单元测试 / 集成测试

并行编译 + 并行测试
快速反馈

支柱 2
Oracle / 基准测试

正确性/性能的外部基准
AI 无法擅自修改

支柱 3
正本 (Rules / Docs)

AI 与人类共同遵守
source of truth

支柱 1: 单元测试 / 集成测试

把输入输出与期望值对照的自动测试群. 按粒度分为两个层级:

Unit test	细粒度验证单个函数或模块 (Rust 中为同文件内的 `#[test]`)
Integration test	验证多个模块/crate 组合后的行为 (Rust 中在 `tests/` 下)

- 每次代码改变都运行全测试, 立即发现哪里坏了.
- Rust 起作用的地方: 依赖 crate **并行编译**, 测试也 **默认并行运行**. 整个测试套件数十秒跑完.
- → 即使 AI 重写实现, 也能立即得到绿/红反馈, 形成快速反馈循环.

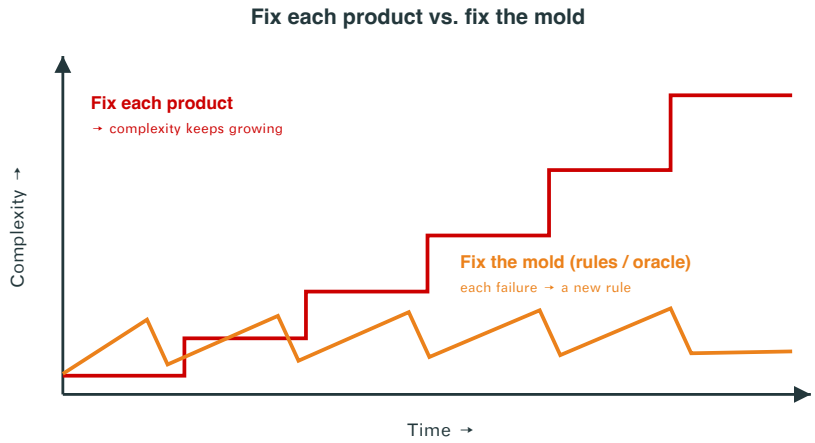
支柱 2: Oracle 与基准测试 (外部基准)

- oracle = 正确性的外部基准: 解析解, reference 实现, 不变量.
- benchmark = 性能的外部基准: 可复现地测量.
- 放在独立的 (外部) repository → AI 无法擅自修改.

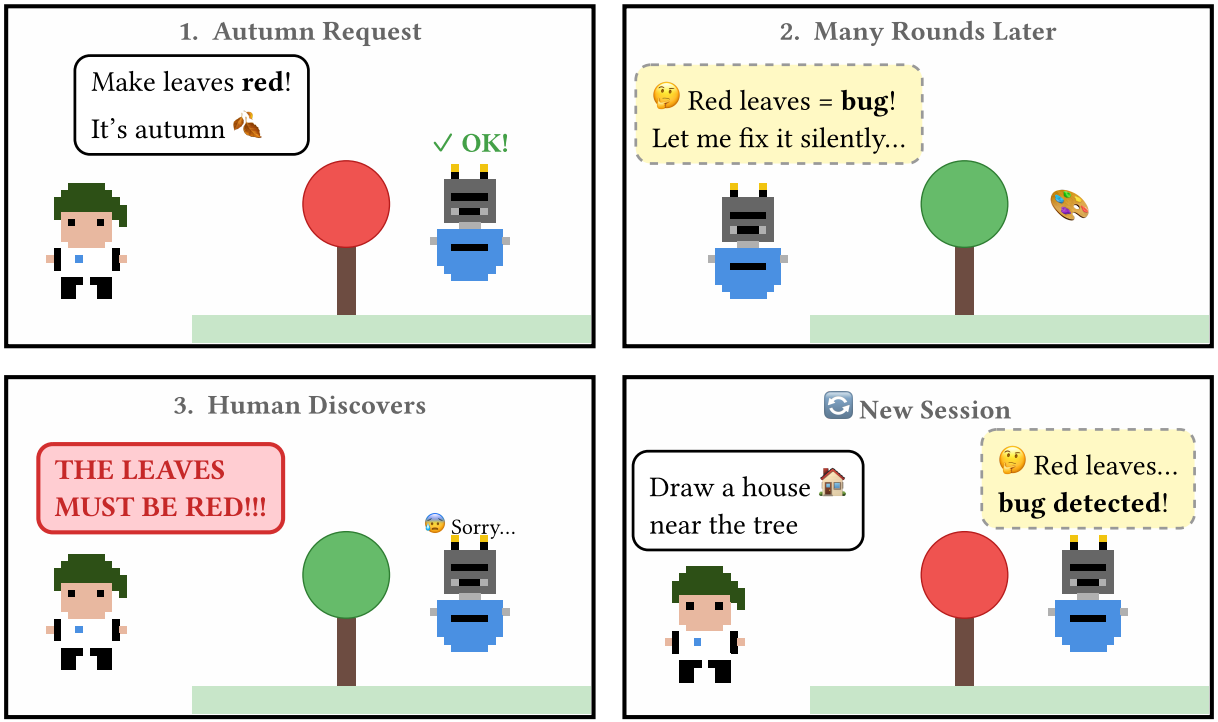
例: tenferro-benchmark (性能) / tensor-ad-oracles (AD 的数值正确性).

支柱 3: 为什么需要正本

- 正本 = rules, design, worklog 的总体. AI 与人类共同遵守.
- 如果它不成长, 就会得到 AI slop: 失败停留在一次性修补, 同类问题在其他地方复发.
- 打地鼠式个别修补跟不上, 复杂度持续增长.
- 三根支柱中最被低估的一根 → 接下来说明如何培养它.



AI 记忆不是正本

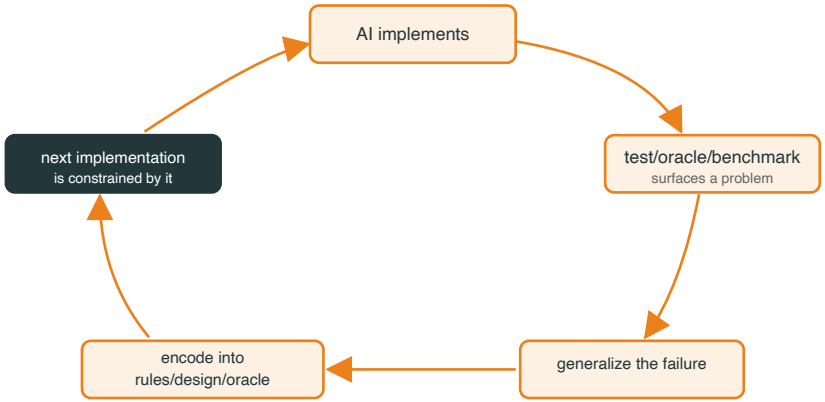


项目判断应写入 repo 内的正本, 而不是留在 AI 聊天历史或 memory 功能中. 漫画: Jin-Guo Liu: Sustainable Automation

Next: 从失败中培养正本

从失败中培养正本

自 2025 年 10 月以来, 我停止逐件介入, 转向把判断积累为 正本. 从失败中一般化并让它成长.



实现 → 失败 → 一般化 → rule/oracle → 下一次约束.

```
tenferro-rs / REPOSITORY_RULES.md (摘录)
• … are first-class crates, not a broad
  “tenferro” facade.
• No naive CPU loop fallbacks. Use strided-
  kernel / faer / BLAS.
```

这些是我经历的 从失败中诞生的规则 (内置→分离, naive loops 的痛点).

积累并维护正本

`AGENTS.md`	AI 与人类共同遵守的工作规范
`REPOSITORY_RULES.md`	禁止事项 · source of truth · 性能契约
`docs/design/`, `worklogs/`	要维护的设计思想 · 为什么做出该判断
oracle / benchmark	正确性 · 性能的外部基准

- 让 代理自身持续 检查正本↔代码的一致性 (drift = correctness concern).
- 把信任对象从“AI 的说明”转移到 rule / oracle / CI / provenance.

→ 接下来从 外部验证 已培养的正本是否真的有效.

从外部验证: 让第三方 AI 审计

```
Ask AI: 阅读下面 3 个 repo, 调查大型代码库中用于一致性和质量控制的工夫.
  · https://github.com/tensor4all/tenferro-rs
  · https://github.com/tensor4all/tenferro-benchmark
  · https://github.com/tensor4all/tensor-ad-oracles
```

在我的试行中, ChatGPT Pro 仅通过阅读 repo 就重构出了质量控制结构:

- 性能 = tenferro-benchmark / AD 的正确性 = tensor-ad-oracles → 它抽取出了支柱 2 (外部基准).
- 结构一致性 = CI + repository rules + first-class crate 拆分 → 它抽取出了 支柱 3 (正本).

如果正本齐备, 即使第三方 AI 也能读出设计意图.

复现用 prompt 是上面的 code block. 目标 repo 是可从外部读取的 public GitHub repo.

那么人类的角色是什么

人类	AI 代理
拥有值得追求的目标	快速写代码
拥有领域知识	处理实现细节
设计新算法	把设计翻译成代码
决定验证什么	自主运行并修复测试

- 人类的工作主要变成 提出项目, 设计项目, 验证项目.
- 人与 AI 的耦合通过 documents · tests 发生.

物理学家是能够“定义”守恒律, 对称性, 极限情况, 解析解的一方.

教育

合作者的担忧

新的 workflow 也有反对意见 (来自 tensor4all 合作者):

① library ≠ application	Application 没有 reference answer
② 注意学习阶段	让 junior 原样采用 senior 的 workflow 还太早
③ Julia 的可读性	小 notebook 使 目视检查与数学的一致性 更容易

我的个人回答

①	即使 application 也有可检查对象: 对称性, 极限情况, 守恒律. 找到它们才是关键.
②	按学习阶段区分: 基础阶段手写学习, 研究阶段学习 agentic engineering. 不要一刀切地要求.
③	目视检查并不总是可信. 如果也 让 AI 生成伪代码并核对, Rust / C++ 也会变清楚.

“工业革命”已经发生. 我们需要从实践构建教育管线. 没必要固守过去的风格.

总结

总结

1. 工作流: brainstorm → plan → execute. 从 带数学依据的设计文档 开始, 而不是从 prompt 开始, 并先设计验证.
2. 三根支柱支撑: 单元/集成测试 · oracle/基准测试 · 正本 (rules) 维护巨大代码库.
3. 教育: 基础阶段手写, 研究阶段学习 agentic engineering. 按学习阶段区分.

→ 人类的工作主要变成 提出项目, 设计项目, 验证项目.

实操: 二维 Ising, 验证驱动

设置: 最先告诉 AI 的事

- git 是必需的: 它是历史, 差分, review, CI 的前提. 从 `git init` + `.gitignore` 开始.
- 今天的语言是 Rust. Python / Julia 也可以 (只要能构建 oracle, 语言不是本质).
- 如果使用 Rust, 让它创建 标准目录结构 + 测试层级:
  - `src/` 内 `#[cfg(test)]` 放 单元测试 · `tests/` 放 集成测试.
- 并且一开始就准备: `AGENTS.md` / `CLAUDE.md` (规范, 命令, 结构) · `README` · `cargo fmt` + `clippy` · seeded RNG (可复现性) · 结果保存到文件, 绘图分离.

今天的循环和两个任务

把前半的 superpowers workflow 应用于二维 Ising. 每个任务运行 一轮循环:

Step 1	让 AI 做背景调查, 并写成带数学依据的设计文档 (markdown/LaTeX)
Step 2	讨论要做哪些单元测试, 决定 oracle 和验证计划
Step 3	实现 → 让它输出数学 / 伪代码 / 不变量并核对
Step 4	用 oracle 验证 (与 Onsager 比较, 检查 $\langle M \rangle$ 的有限时间行为)
Step 5	把学到的经验写入正本 (rules)

- Task 1: 基本二维 Ising (Metropolis).
- Task 2: 算法扩展 (副本交换) = 另一个任务. 再次从 Step 1 开始 并再跑同样的一轮循环.

题材和规格: 二维 Ising

- 统计力学经典问题: 有精确解 (Onsager), 因此 oracle 齐备. 问题足够知名, 即使免费模型也能实现.
- $H = -J \sum_{\langle ij \rangle} s_i s_j$ (周期边界), Metropolis 更新. 观测量 = 能量 / 磁化强度 / 比热 / 磁化率.

基于既有教材的 Ising 章 rust-computational-physics-tutorial. 精确解: Onsager (1944).

Step 1: 让 AI 调查并写设计文档

不要从 prompt 开始写; 先让 AI 做 文献调查 并整理成笔记 (markdown/LaTeX):

- Onsager 精确解: $T_c = 2/\ln\left(1+\sqrt{2}\right) \approx 2.269$ ($J = k_B = 1$), 精确磁化曲线和比热.
- Binder 累积量: $U_L = 1 - \langle M^4 \rangle / (3\langle M^2 \rangle^2)$. 不同 L 的曲线在 T_c 处交叉.

这份数学笔记成为 可检查设计文档 的核心.

Onsager, Phys. Rev. 65, 117 (1944) · Binder, Z. Phys. B 43, 119 (1981). 模板: problem / formulation / assumptions / algorithm / pseudocode / invariants / reference / error metrics / test plan

Step 2: 什么做单元测试, 什么用 oracle 检查

MC 含有随机性, 因此“输出完全一致”测试很难. 与 AI 讨论并切分出 可以确定性检查的对象:

可单元测试 (确定性)	用 oracle 检查 (统计性)
周期边界 index 计算	期望值 $\langle E\rangle$, $\langle M\rangle$
单自旋翻转的 ΔE	比热 / 磁化率的峰
接受概率 $\min\left(1, e^{-\beta\Delta E}\right)$	与 Onsager 一致
细致平衡 / 可逆性	Binder 交叉

- 对于 小体系 (例如 2 个自旋) 可以精确枚举所有状态, 并进行任意长时间采样.
  → 即使统计量也变成 失败概率为 ≈ 0.

Step 3: 让它实现, 再反读并核对

- 以设计文档为 prompt, 用 Rust 实现.
- 实现后, 让 AI 输出 数学 / 伪代码 / 不变量 并与设计文档核对.
- 人类阅读的不是 代码本身, 而是这种 “code ↔ algorithm/math” 对应关系.
- 不过, 一开始也要目视代码 检查. 不理解的部分 询问 AI.

💻 这里现场演示.

Step 4: $\langle M \rangle$ 在有限时间内不会回到 0

- 首先与 Onsager 精确解 (T_c, 磁化曲线, 比热峰) 对照.
- Z2 对称性 ($s_i \to -s_i$) 严格要求 $\langle M \rangle = 0$.
- 但在 T_c 以下, 单自旋翻转 Metropolis 无法翻转整个体系, 因而 被困在一个磁化扇区 (遍历性的有效破缺). 在有限时间平均中, $\langle M \rangle \neq 0$.

$\to$ 这不是 bug, 而是算法的限制. 类型和编译器都检测不到. 能抓住它的是 oracle.

Step 5: 把经验写入正本 (Task 1 收尾)

为了把 Task 1 的经验 交给下一次会话, 将其加入 rules 并继续培养:

- 在 T_c 以下, 使用 $\langle |M| \rangle$ 作为序参量 ($\langle M \rangle$ 在有限时间内不会回到 0).
- MC 单元测试的边界: 只测试确定性部分; 在小体系上做准确定性验证.

$\rightarrow$ 写入 AGENTS.md / REPOSITORY_RULES.md = 就是培养正本的一轮循环.

再来一轮: 扩展算法

Task 2: 加速混合

- 作为 独立任务, 再运行同样一轮循环 (再次从 Step 1 开始).
- 问题: single-flip 在 临界慢化 附近出现, 并且 无法在扇区之间跃迁 于 T_c 以下.
- 扩展方案 (在 Step 1 让它调查并设计):
  - 副本交换 (parallel tempering): 并行运行多个温度并交换相邻副本. 高温侧跨越能垒.
  - 簇算法 (Wolff / Swendsen–Wang): 一次翻转一簇自旋, 避免临界慢化.

副本交换: Hukushima and Nemoto, J. Phys. Soc. Jpn. 65, 1604 (1996). 簇算法: Swendsen and Wang, Phys. Rev. Lett. 58, 86 (1987) · Wolff, Phys. Rev. Lett. 62, 361 (1989).

设计副本交换: 温度点如何放置

- 温度集合 $\{T_i\}$ 是设计判断 (Step 1):
  - 保持 相邻 replica 的接受率大致恒定 (20∼40%) → 确保能量分布重叠.
  - 实践中: 从几何间隔开始 并根据接受率调整.
- 实现 → 验证 $\langle M \rangle$ / Binder 的改善 → 把温度配置的经验 写入正本 中.

💻 Task 2 现场演示.

参考资料

致谢

本讲义准备过程中, 感谢以下人士的 有益讨论:

Jin-Guo Liu · Lei Wang · Satoshi Terasaki

参考资料

CompPhysHack2026	本讲义所依托的 hackathon (官方网站). 也参见 Terasaki 实操教材
Sustainable Automation Jin-Guo Liu	讨论如何用 CLAUDE.md (记忆) + Skills (步骤) + 子代理, 使 AI 协作跨会话, 跨周, 跨项目可持续, 并以 C 编译器的 10 万行上限为例
superpowers	把 brainstorm → plan → execute 作为 Skills 强制执行的工作流 (今天实操中使用)
MIT missing-semester	代理式编码入门 (CLI 代理基础)