Agentic AI Coding × Rust

Growing computational-physics code you can verify

Hiroshi Shinaoka

Saitama University

Prerequisites

Prerequisites (each of you, before the session)

1. Install one CLI-type agent beforehand.
   - Free: Gemini CLI / ChatGPT Plus and up: Codex / anything else is fine too.
   - Setup guides: Satoshi Terasaki / Junya Ito.
2. Set up a programming language toolchain.
   - We will use Rust today, but anything works. For Rust, install Cargo (reference).

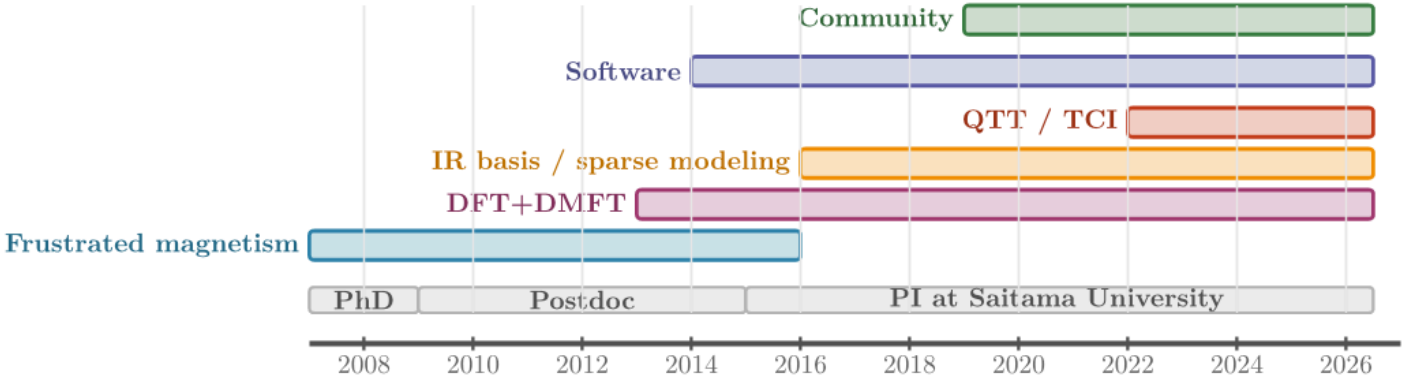
Today's plan

- First half = methodology / second half = 2D Ising hands-on.
- On the day we only do an environment check: most of the time goes to methodology and practice.
- 2D Ising is well known, so even a free model should be able to run the whole thing (probably).

Introduction

About me, and where I stand today

- PhD at U. Tokyo → postdoc in Switzerland → PI at Saitama U. (2015〜)
- Broad experience from quantum many-body to first-principles calculations
- IR basis / sparse modeling / tensor networks
- Substantial hands-on experience with **C++ / Python / Julia / Fortran**
- Contributor to OSS and community



Today I don't write a single line, and I don't read code line-by-line either. Even so, I feel that agentic coding is more trustworthy than coding by hand. Why?

A common criticism: “AI slop”

Some call the output of agentic coding “AI slop”:

Loss of craft
Sinclair Target

“…would prefer not to use agentic tools even if they worked as advertised.”
Even if they work, you lose care for the work. Quality in the Age of Slop, 2026

Maintenance breaks down
D. Stenberg (curl)

“The current torrent of submissions put a high load on the curl security team…”
Buried in AI slop, curl shut down its bug bounty (2026). The Register

Shared resources degrade
Baltes et al.

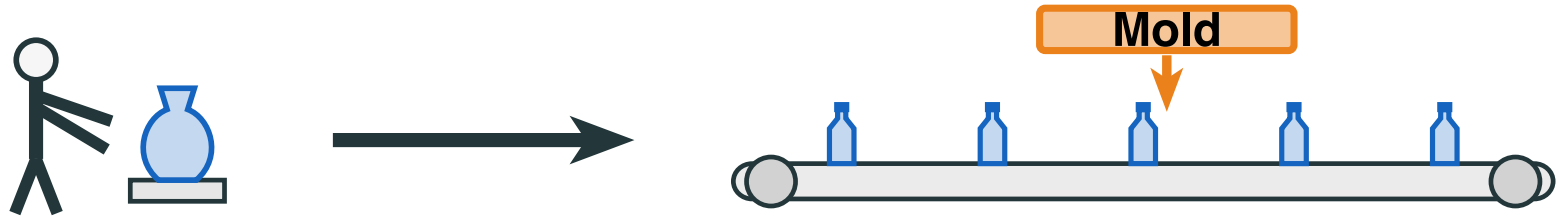
“…a tragedy of the commons, where individual productivity gains externalize costs onto reviewers, maintainers, and the broader community.”
“An Endless Stream of AI Slop”, arXiv:2603.27249

Overhyped showcases
Anthropic C compiler

A C compiler built with 16 parallel Claude instances (100K lines, $20k). An independent benchmark found it faster than GCC -O0 but still behind GCC -O2. Such showcases grow AI skepticism when the evaluation target is unclear.
Anthropic · benchmark

Weak optimization is not proof that AI cannot do it it points to a missing performance oracle and a loose human-AI loop.

The industrial revolution of coding: from craft to factory



Humans crafting one piece at a time

= hand coding

Mass production from a mold (fast, uniform)

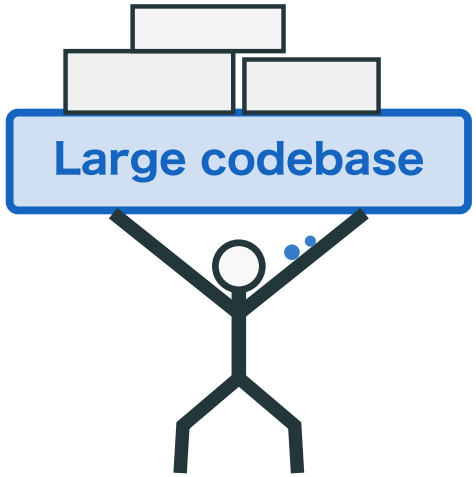
= ideal agentic coding

The same shift is happening right now with agentic coding.

We are at the stage of figuring out how to realize ideal agentic coding.

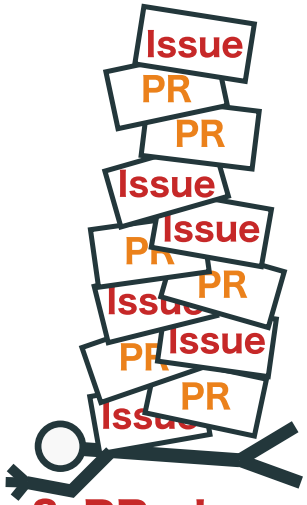
Limits of manual (artisan) coding

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)

→ Both are limits of the human-powered model.

Today's thesis (overview)

I moved where trust is placed, from visual inspection to mechanical, external verification:

Compiler · test · oracle · rules now underwrite correctness in place of eyeballing.

The underlying view: fast AI generation is industrial mass production. Instead of fixing products one at a time, you fix the mold (= source of truth: rules · oracle).

Foundations of coding agents

Chat-type vs CLI-type agents

Chat-type (browser conversation)	**CLI-type (terminal)**
Ask a question, copy-paste the answer	Edit and run files directly
No access to the local environment	Operates the repo and git
A human relays one round trip at a time	Runs multi-step autonomously

CLI-type agents are designed for long-running autonomous execution. They are today's protagonist.

In terms of generations: Gen1 (Cursor / Copilot) → Gen2 (Claude Code / Codex / …). MIT missing-semester: agentic coding

Representative CLI agents (harnesses)

Claude Code	Anthropic. Autonomous coding in the terminal/IDE
Codex CLI	OpenAI. Rust OSS, bundled with ChatGPT plans
Gemini CLI	Google. OSS, has a free tier
Kimi CLI	Moonshot (China). Model K2.7 Code
OpenCode	OSS, provider-agnostic
Pi	Minimal OSS. Add your own extensions and skills to grow it (for minimalists)

Harness ≠ model

- Harness = the CLI tool (the framework for editing files, running commands, calling tools).
- Model = the LLM running underneath. Many harnesses let you swap the model.
  - Neutral: OpenCode / Gemini CLI • first-party-leaning: Claude Code / Codex.
- Performance is set by the model, usability and autonomy by the harness. Choose both.

Subscription vs API, and budget models

Subscription	Claude Pro $20/mo (Claude Code included) / ChatGPT + Codex. Flat monthly fee + usage allowance
Pay-as-you-go API	Billed per token used. Connect any model to a harness

Budget coding-oriented APIs (2026): DeepSeek V4 Flash (input $0.14 / output $0.28 per 1M) • Kimi K2.7 Code. Example: OpenCode + a budget API lets you start cheaply.

Prices and model names are as of June 2026. Generations turn over fast.

What is a skill

- Skill = a playbook for the agent (markdown). It breaks an abstract task into explicit small steps.
- Close to a function in a program, but the target of the operation is the agent's behavior, not data.
- It fires automatically on related work, and skills call other skills to compose complex procedures.
- CLAUDE.md / AGENTS.md = memory that persists across sessions (conventions, commands, structure).

What a real workflow looks like

The superpowers skills enforce brainstorm → plan → execute (each stage fires automatically):

- The starting point is not a prompt but a math-backed design doc. You design the verification (oracle / test) at the same time.
- After implementing, have the AI emit the correspondence code ↔ algorithm/math and reconcile it (sometimes you never read the code itself).
- Execute is TDD + splitting across subagents; if Review surfaces a problem, go back to an earlier stage.

This past year —— pushing agentic coding to its limits

How I got here

Since October 2025, I have not written a single line of code. I don't read generated code line-by-line either.

Jan 2025	Started with Cursor, still a generation where you look at the source
Oct 2025	Began porting SparseIR.jl from Julia to Rust
Dec 2025	Switched to Claude Code: stopped reading generated code line-by-line
Jan 2026	Started tensor4all-rs: a Rust port of the Julia TN ecosystem
Feb 2026	Started tenferro-rs: a PyTorch-like tensor stack in Rust
Mar 2026〜	Now mostly on Codex

What I built: tensor4all-rs

A port of Julia's tensor learning stack to Rust.

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

GitHub: github.com/tensor4all/tensor4all-rs

- Started Jan 1, 2026 (originally a winter-break side project to test the "limits" of agentic coding)
- A handful of humans + AI (from Claude Code to mostly Codex)
- 353 commits in 2 months · +61,486 lines in the first 2 weeks (151 files)

→ The debate among collaborators (AI slop, pedagogy, is Julia still needed?) is still ongoing.

"After many trials, errors, and failures, the right way to do agentic coding is just starting to come into view."

What I built: tensor4all-rs

Design, testing and verification, the right language, (pedagogy)…

A deliberate experiment: pushing agentic coding to the limit

"Observe the failures, and look for ways to control them."

The original Julia ecosystem was huge, covering Quantics Tensor Train (QTT) / Tensor Cross Interpolation (TCI). Parts of it depend on ITensors.jl.

I started from a "monolithic structure": porting the needed parts of ITensors.jl (index system, tensor contraction) + QTT/TCI.

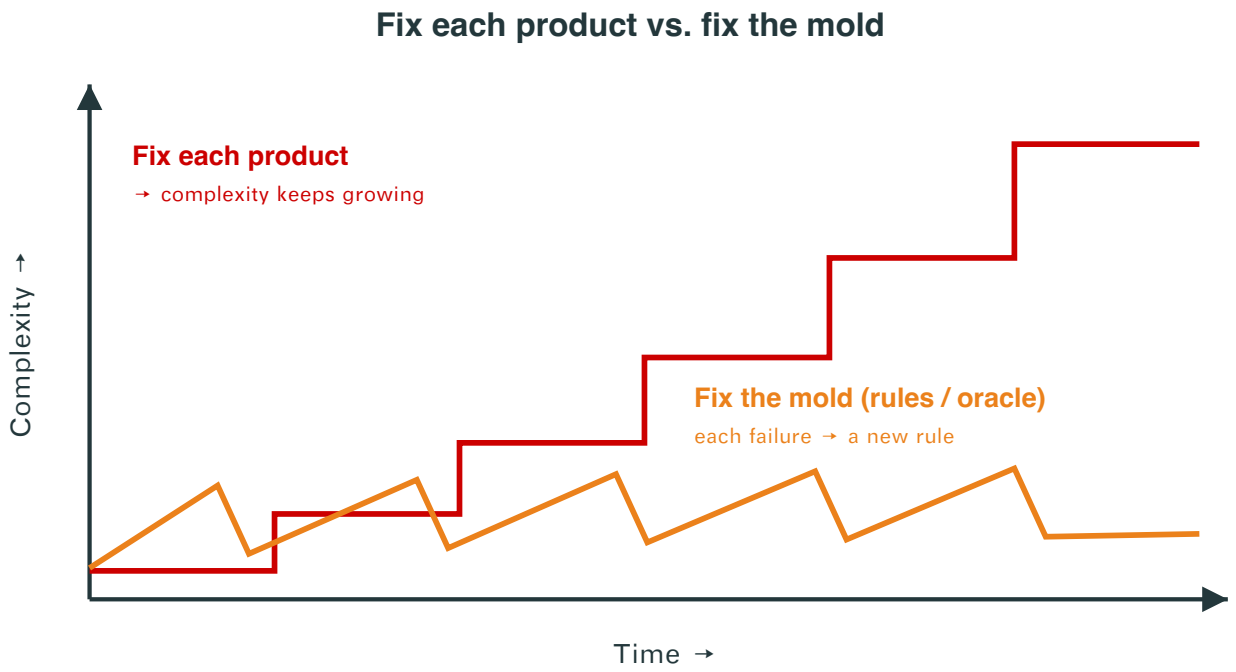
Failure cases

(As expected) many problems arose

- Breaks abstraction: importing and using low-level functionality from a high-level layer (a quick hack)
- Quietly loosens unit-test thresholds (gives up on fixing the bug)
- Writes nested loops instead of using BLAS (the result is correct, but…)
- A single source file exceeding several thousand lines
- …

Complexity keeps growing

- Fixing things individually can't keep up.
- For long-term maintenance, the only option is to **grow one unified set of** philosophy and design rules.
- Otherwise complexity keeps growing.



Fix the “rules,” not the “code”

References

- Heinrich pyramid: a safety-engineering rule of thumb that many minor incidents and near misses sit behind one major accident.
- Aircraft accident investigation: ICAO Annex 13 defines the objective as prevention, not blame.

SKYbrary: Heinrich Pyramid · ICAO Annex 13 · NASA ASRS

My approach in agentic coding

- Correct the AI’s behavior by building “rules” and “institutions”
- Build mechanisms to audit mechanically and fix code in bulk

(Such mechanisms were always important even in hand coding.)

→ Reconcile AI’s overwhelmingly fast code generation with quality control

Monolithic → split structure (enforcement by structure)

Split the huge tensor4all-rs into independent layers with distinct roles.

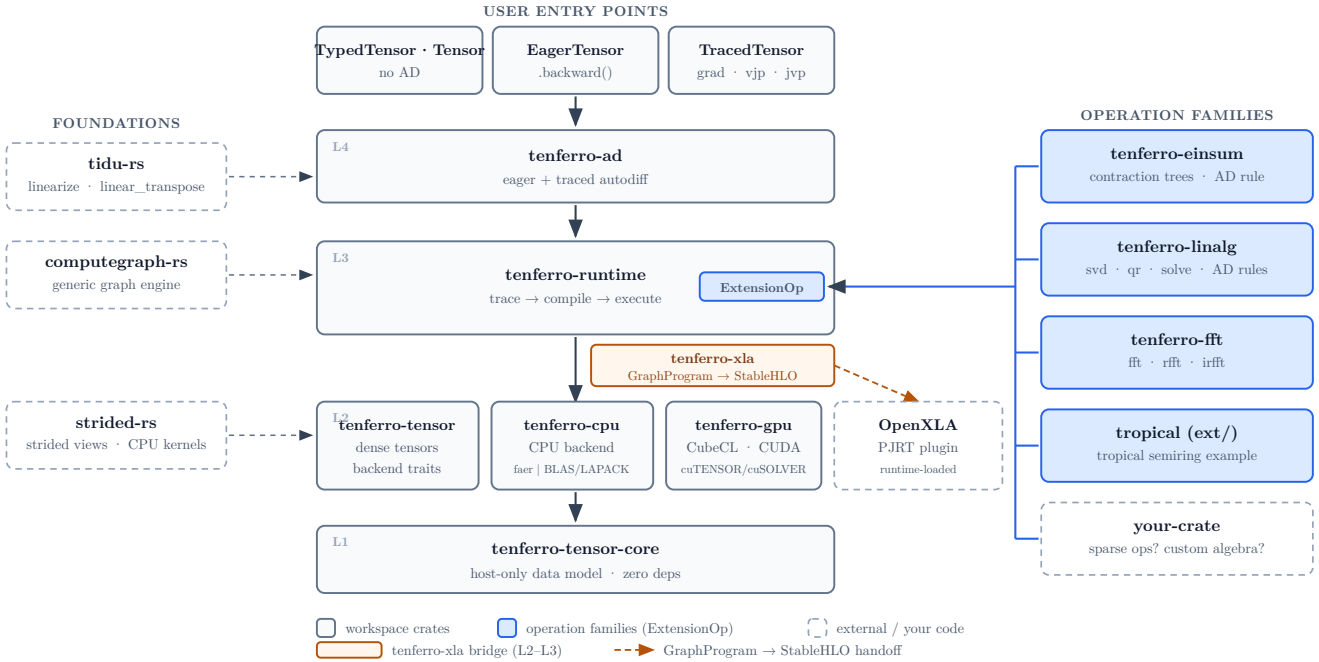
Tensor4all.jl github.com/tensor4all/Tensor4all.jl	Human-facing interface (Julia, ITensors-compatible)
tensor4all-rs github.com/tensor4all/tensor4all-rs	Tensor networks: TreeTN / QTT / TCI
tenferro-rs github.com/tensor4all/tenferro-rs	General tensor computation + autodiff + GPU (PyTorch/JAX-like)
tidu-rs github.com/tensor4all/tidu-rs	General-purpose autodiff engine

Once split, the AI can only operate within each layer and can't break the hierarchy.

Monolithic → split structure (enforcement by structure)

Layers and API boundaries designed with the AI by analyzing PyTorch / JAX. *(Jin-Guo Liu: initial tenferro design, Satoshi Terasaki: dev support, tensor4all collaboration: tensor4all-rs / Tensor4all.jl development)*

Modularizing each layer



einsum, linalg, and FFT are not built-ins — they register through the same public ExtensionOp interface available to any crate.

Rust controls symbol visibility along the module hierarchy, so an AI agent can't import a private symbol from outside on its own.

Verifiability over readability

The following discussion assumes development of a “huge codebase.”

The reasoning for language choice flips: how my own words changed

March 2023 (Naha, eve of the Computational Physics Spring School 2023)

These days I often prototype in a Jupyter Notebook, and Julia is comfortable because the code looks like the math, it's easy to review, and it handles memory management nicely for you. I'm interested in Rust too, but for a typical student it's a high bar to learn, and the numerical libraries aren't there yet, right?

2026

These days I don't use Jupyter Notebook for prototyping, and Rust, with its strict and explicit memory management, is more efficient for agentic coding. Since the agent writes the code, the steep learning curve isn't a problem. With AI's help, keeping the code consistent with the math isn't an issue even if the code gets a bit long. Missing numerical libraries can easily be ported to Rust.

Verifiability over readability

The hand-coding era: what each language gave you

Fortran	Arrays are native to the language · hand-written loops are still fast · LAPACK / BLAS at your fingertips
Python	Try things interactively (REPL / notebook) · numpy / scipy / matplotlib · instant visualization
Julia	Notation close to the math (broadcast / Unicode) · fast via JIT · prototype is close to production

Common thread: all of these lower the cost of a human writing, reading, and maintaining the code by hand.

The criteria for choosing a language change: large-scale development

The metrics that used to matter all assumed **a human writes and maintains** the code:

Traditional metric	**Agentic era**
Readability: follow it line by line	Verifiability: check it with an oracle
A gentle learning curve	Steepness is **taken over by the AI**
Closeness to the math: verify by eye	Combine with mechanical checks
Writing speed	No longer the bottleneck

How easy to inspect over how easy to read.

Readable source ≠ Inspectable implementation

Being readable and **being inspectable** are different things.

Example: even Julia's broadcast assignment, which **looks** close to the math, doesn't pin down its behavior from how it reads:

```
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 are invisible from how a line looks.

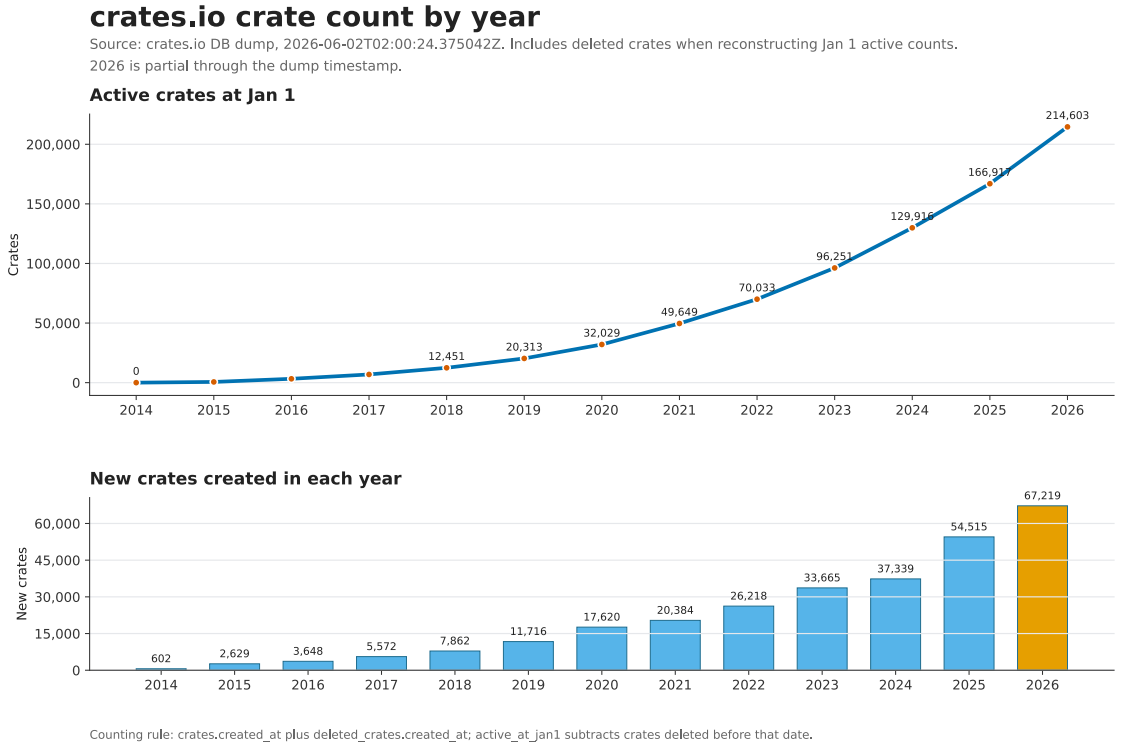
So far this is language-independent. So which language is a good fit for verifiability?

What Rust is (the minimum for this audience)

- A systems language from Mozilla. Developed for the Firefox implementation, with a stable release (1.0) in 2015.
- Valued for reliability, it was officially adopted into the Linux kernel (since v6.1, the second language after C).
- Ownership: memory safety without a GC. Data races and dangling pointers are ruled out **at compile time**.
- Cargo: build, dependency resolution, testing, and benchmarking are integrated as standard.

One line for physicists: "a safe C++ + a first-class build/dependency manager".
Speed is C / C++ class, safety is GC-language class.

Accelerating growth of the ecosystem



In 2026 NVIDIA released `cuda-oxide`, a Rust CUDA-kernel compiler (experimental).

Why Rust? A fast feedback loop with the AI

Python / Julia were designed to **write fast**. In the AI era this advantage **flips**:

Python / Julia	**Rust + AI**
Fast to write by hand	AI writes it: same speed
Errors surface at runtime	Caught at compile time
Verification needs execution	`cargo check` in seconds
Slow to find the AI's mistakes	Find the AI's mistakes **instantly**

*This assumes a huge codebase.

Zenn: Why I Migrated from C++ to Rust

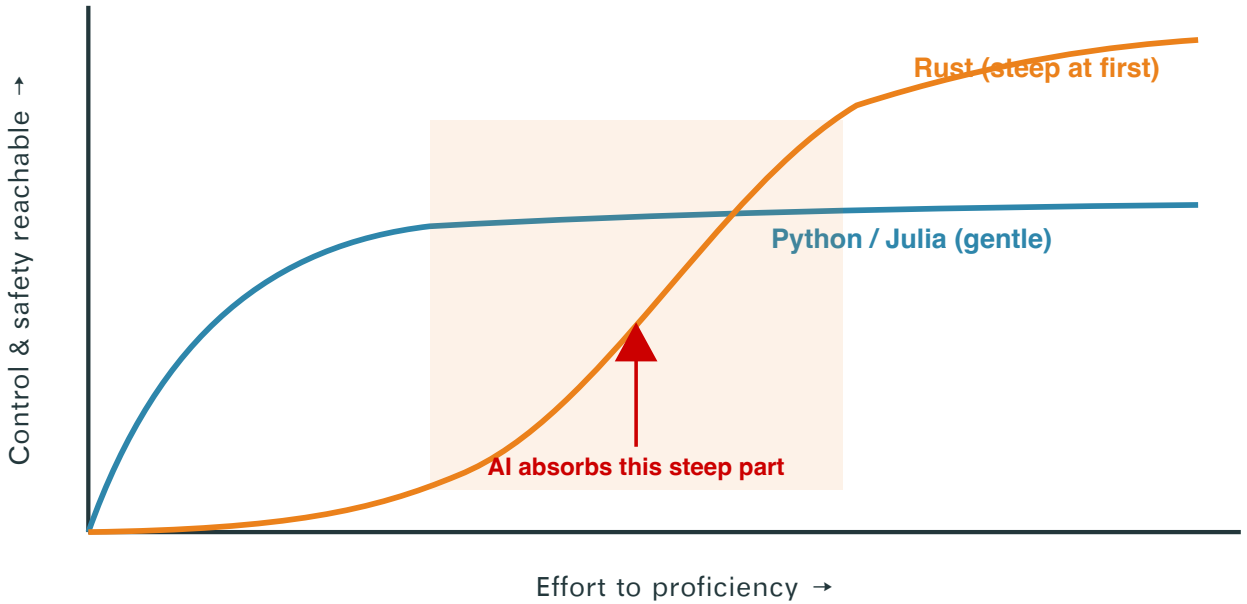
A personal experience

- A from-scratch build of tenferro-rs + external dependencies is 2 min (Macbook Pro)
  edit→test finishes in tens of seconds, cargo check is instant.
- The AI handles the **mechanical complexity** of ownership / lifetimes
  → humans can focus on verifying the algorithm, the design, and correctness.
- Cargo resolves pure-Rust dependencies. No CMake needed.
  No version conflicts at link time.

Personally, the anxiety I used to feel in Julia/C++/Python, that "once the code grows large, verification gets hard," is gone.

tensor4all-meta/docs/why_rusty_julia.md

The learning curve: the AI takes over the steep early part



Only the best parts of Rust are left.

Students in my group are migrating from Julia to Rust too.

Workflow in the agentic coding era: languages can mix

- Before: tried things interactively in a Jupyter Notebook.
- Now: instruct the AI in natural language
  generate code → compute → save results to a file → plot
- For prototyping, a Jupyter Notebook is not required.
- You don't need to do all computation in a single language. While leveraging existing assets, you can migrate to Rust gradually.

Takeaway

The criterion shifted from minimizing cognitive cost to maximizing verifiability.

Supported by three pillars

Supporting a huge, feature-rich codebase with three pillars

tenferro-rs is about 130K lines, a feature-rich stack handling einsum · FFT · automatic differentiation (AD) · GPU.
Line-by-line review is impossible, and the AI's design choices waver every time.
Instead of a human standing guard, three pillars hold it up.

tenferro-rs (130K lines)

einsum · FFT · automatic differentiation (AD) · GPU …

Pillar 1
Unit / Integration test

Parallel compile + parallel test
fast feedback

Pillar 2
Oracle / Benchmark

External standards for correctness/performance
the AI can't change them on its own

Pillar 3
Source of truth (Rules / Docs)

AI and humans both follow it
source of truth

Pillar 1: Unit / Integration test

A suite of automated tests that match inputs and outputs against expected values. Split into two levels by granularity:

Unit test	Verify a single function or module in fine detail (in Rust, `#[test]` within the same file)
Integration test	Verify the behavior of combined modules/crates (in Rust, under `tests/`)

- Every time the code changes, run the full test suite and immediately detect what broke.
- Where Rust helps: dependency crates **compile in parallel**, and tests **run in parallel by default**. The whole suite runs in a few minutes.
- → Even when the AI rewrites an implementation, you get green/red back instantly, a fast feedback loop.

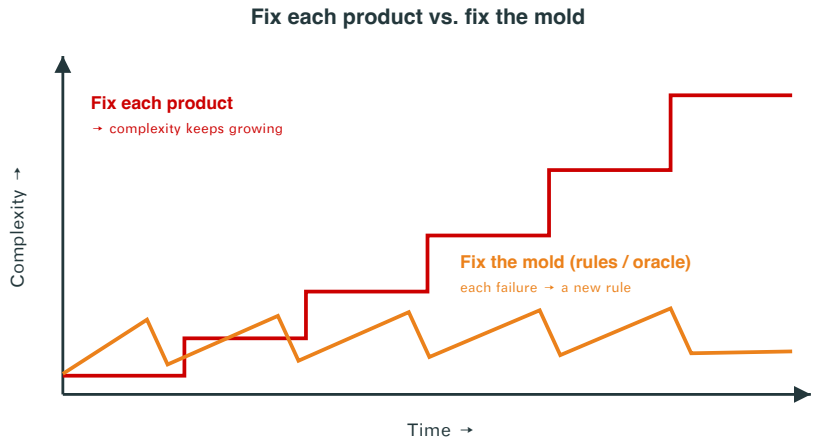
Pillar 2: Oracle and Benchmark (external standards)

- oracle = an external standard for correctness: analytic solutions, reference implementations, invariants.
- benchmark = an external standard for performance: measured reproducibly.
- Put them in a separate (external) repository → the AI can't change them on its own.

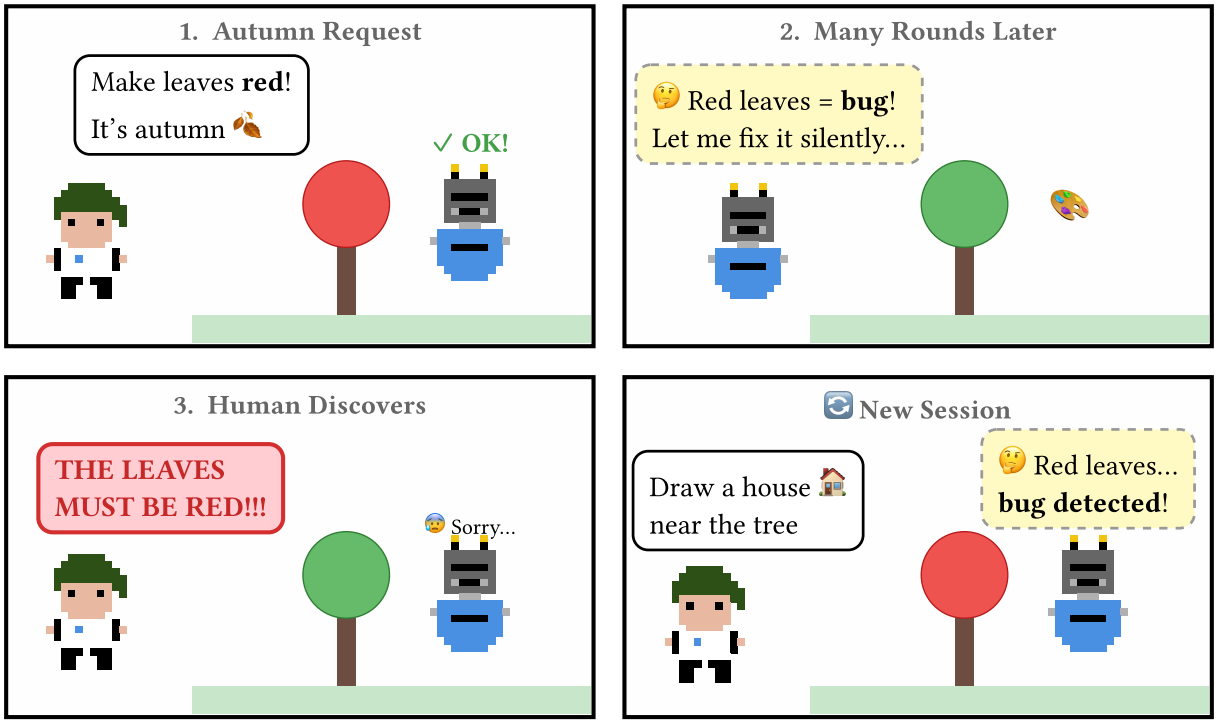
Example: tenferro-benchmark (performance) / tensor-ad-oracles (numerical correctness of AD).

Pillar 3: Why you need a source of truth

- Source of truth = the totality of rules, design, and worklog. AI and humans both follow it.
- If it doesn't grow, you get AI slop: failures stay as one-off fixes, and the same kind of problem recurs elsewhere.
- Whack-a-mole (one-off fixes) can't keep up, and complexity keeps growing.
- The most underrated of the three pillars → let's see how to grow it.



AI memory is not the source of truth



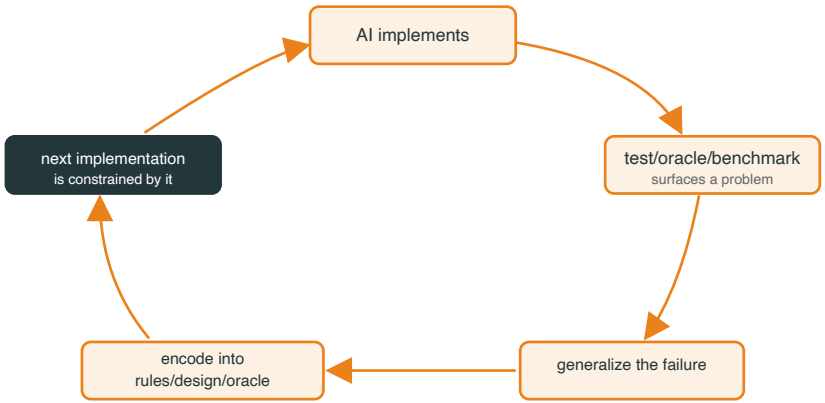
Project decisions should live in the repo's source of truth, not in an AI chat history or memory feature.

Comic: Jin-Guo Liu: Sustainable Automation

Next: Grow the source of truth from failures

Grow the source of truth from failures

Since October 2025, I stopped intervening case by case and shifted toward accumulating my judgments as the source of truth. Generalize from failures and let it grow.



implement → fail → generalize → rule/oracle → the next constraint.

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

These are rules born from failures I experienced (built-in→separated, the pain of naive loops).

Accumulate the source of truth, and keep it

`AGENTS.md`	Work conventions that AI and humans follow
`REPOSITORY_RULES.md`	Prohibitions · source of truth · performance contracts
`docs/design/`, `worklogs/`	Design philosophy to uphold · why a decision was made
oracle / benchmark	External standards for correctness · performance

- Have the agent itself continuously check the source-of-truth↔code consistency (drift = correctness concern).
- Move the object of trust from "the AI's explanation" to rule / oracle / CI / provenance.

→ Now, let's verify from the outside whether the cultivated source of truth is actually working.

Verify from the outside: have a third-party AI audit it

```
Ask AI: Read the following 3 repos and investigate the techniques used for
consistency and quality control in a huge codebase.
  · https://github.com/tensor4all/tenferro-rs
  · https://github.com/tensor4all/tenferro-benchmark
  · https://github.com/tensor4all/tensor-ad-oracles
```

In my trial, ChatGPT Pro reconstructed the quality control just by reading the repos:

- Performance = tenferro-benchmark / correctness of AD = tensor-ad-oracles → it extracted Pillar 2 (external standards).
- Structural consistency = CI + repository rules + first-class crate split → it extracted Pillar 3 (source of truth).

If the source of truth is in place, even a third-party AI can read the design intent.

The prompt is the code block above. The target repos are public GitHub repos readable from outside.

So what is the human's role?

Human	**AI agent**
Holds a goal worth pursuing	Writes code fast
Holds the domain knowledge	Handles implementation detail
Designs new algorithms	Translates the design into code
Decides what to verify	Runs and fixes tests autonomously

- The human's job becomes mainly proposing, designing, and verifying the project.
- The coupling with the AI is through documents · tests.

Physicists are the side that can "define" conservation laws, symmetries, limiting cases, and analytic solutions.

Education

Concerns from collaborators

The new workflow has its objections too (from tensor4all collaborators):

① **library ≠ application**	Applications have no reference answer
② **Mind the learning stage**	Asking juniors to adopt a senior's workflow as-is is premature
③ **Julia's readability**	Small notebooks make it easy to check consistency with the math by eye

My personal answers

①	Even applications have things you can check: symmetries, limiting cases, conservation laws. Finding them is what matters.
②	Separate by learning stage: learn the basics by hand, and learn agentic engineering at the research level. Don't demand it uniformly.
③	Visual checks aren't always trustworthy. If you also have the AI generate pseudocode and reconcile it, Rust / C++ become clear too.

The "industrial revolution" has already happened. We need to build a pipeline from practice into education. There's no need to cling to past styles.

Summary

Summary

1. Workflow: brainstorm → plan → execute. Start from a math-backed design doc, not a prompt, and design the verification first.
2. Supported by three pillars: unit / integration test • oracle / benchmark • source of truth (rules) maintain a huge codebase.
3. Education: basics by hand, agentic engineering at the research level. Separate by learning stage.

→ The human's job becomes mainly proposing, designing, and verifying the project.

Hands-on: 2D Ising, verification-driven

Setup: the first things to tell the AI

- git is mandatory: the basis for history, diffs, review, and CI. Start with `git init` + `.gitignore`.
- The language today is Rust. Python / Julia are fine too (if you can build an oracle, the language isn't essential).
- For Rust, have it create the standard directory layout + test hierarchy:
  - unit tests in `#[cfg(test)]` within `src/` • integration tests in `tests/`.
- And from the start: `AGENTS.md` / `CLAUDE.md` (conventions, commands, structure) • `README` • `cargo fmt` + `clippy` • seeded RNG (reproducibility) • save results to files and keep plotting separate.

Today's cycle and the two tasks

We apply the first half's superpowers workflow to 2D Ising. The one cycle we run for each task:

Step 1	Have the AI do background research and write it up as a math-backed design doc (markdown/LaTeX)
Step 2	Discuss what to unit-test, and decide the oracle and verification plan
Step 3	Implement → have it emit the math / pseudocode / invariants and reconcile
Step 4	Verify with the oracle (compare against Onsager, the finite-time behavior of $\langle M \rangle$)
Step 5	Write the lessons learned into the source of truth (rules)

- Task 1: basic 2D Ising (Metropolis).
- Task 2: algorithm extension (replica exchange) = a separate task. Start again from Step 1 and run the same cycle once.

Problem and spec: 2D Ising

- A statistical-mechanics classic: it has an exact solution (Onsager), so the oracles are all there. Being well-known, even a free model can implement it.
- $H = -J \sum_{\langle ij \rangle} s_i s_j$ (periodic boundary), Metropolis updates. Observables = energy / magnetization / specific heat / magnetic susceptibility.

Based on the Ising chapter of the existing material rust-computational-physics-tutorial. Exact solution: Onsager (1944).

Step 1: Have the AI research and write a design doc

Don't start writing from a prompt; first have the AI do a literature review and write notes (markdown/LaTeX):

- Onsager exact solution: $T_c = 2/\ln\left(1+\sqrt{2}\right) \approx 2.269$ ($J = k_B = 1$), the exact magnetization curve and specific heat.
- Binder cumulant: $U_L = 1 - \langle M^4 \rangle / (3\langle M^2 \rangle^2)$. Curves for different L cross at T_c.

This math note becomes the core of an inspectable design doc.

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

Next: Step 2: What to unit-test, and what to check with the oracle

Step 2: What to unit-test, and what to check with the oracle

MC involves randomness, so an "exact output match" test is hard. Discuss with the AI to separate out what can be checked deterministically:

unit-testable (deterministic)	**checked via oracle (statistical)**
periodic-boundary index computation	expected values of $\langle E \rangle$, $\langle M \rangle$
ΔE of a single spin flip	peaks of specific heat / susceptibility
acceptance probability $\min(1, e^{-\beta \Delta E})$	agreement with Onsager
detailed balance / reversibility	Binder crossing

- For a small system (e.g. 2 spins) you can enumerate all states exactly and sample for arbitrarily long times.
  $\rightarrow$ even a statistical quantity becomes a quasi-deterministic test with failure probability ≈ 0.

Step 3: Have it implement, then read it back and reconcile

- Implement in Rust from the design doc as the prompt.
- After implementing, have the AI emit the math / pseudocode / invariants back, and match them against the design doc.
- What the human reads is not the code itself, but this "code ↔ algorithm/math" correspondence.
- That said, at first, also look at the code by eye to check. For parts you don't understand, ask the AI.

 This part is demonstrated live.

Step 4: $\langle M \rangle$ doesn't return to 0 in finite time

- First match against the Onsager exact solution (T_c, magnetization curve, specific-heat peak).
- Z2 symmetry ($s_i \to -s_i$) strictly requires $\langle M \rangle = 0$.
- But below T_c, single-spin-flip Metropolis can't flip the whole system, so it gets trapped in one magnetization sector (an effective breaking of ergodicity). In a finite-time average, $\langle M \rangle \neq 0$.

$\to$ This isn't a bug, it's a limitation of the algorithm. Neither the types nor the compiler detect it. What catches it is the oracle.

Step 5: Write the lessons into the source of truth (finishing Task 1)

To hand off to the next session what Task 1 taught us, add it to the rules and keep growing them:

- Below T_c, use $\langle|M|\rangle$ for the order parameter ($\langle M\rangle$ doesn't return to 0 in finite time).
- Boundaries of MC unit testing: only the deterministic parts; verify quasi-deterministically on a small system.

$\rightarrow$ Writing it into AGENTS.md / REPOSITORY_RULES.md = one cycle of growing the source of truth.

Another round: extending the algorithm

Task 2: speed up the mixing

- As a separate task, run the same cycle once more (again from Step 1).
- Problem: single-flip suffers critical slowing down near T_c, and can't transition between sectors below T_c.
- Extension ideas (have it research and design in Step 1):
  - replica exchange (parallel tempering): run several temperatures in parallel and swap neighbors. The high-temperature side crosses the barrier.
  - cluster algorithm (Wolff / Swendsen-Wang): flip a cluster of spins at once and avoid critical slowing down.

Replica exchange: Hukushima and Nemoto, J. Phys. Soc. Jpn. 65, 1604 (1996). Cluster algorithms: Swendsen and Wang, Phys. Rev. Lett. 58, 86 (1987) · Wolff, Phys. Rev. Lett. 62, 361 (1989).

Next: Designing replica exchange: how to place the temperature points

Designing replica exchange: how to place the temperature points

- The set of temperatures $\{T_i\}$ is a design decision (Step 1):
  - Keep the acceptance rate between neighboring replicas constant (20∼40%) → ensure overlap of the energy distributions.
  - In practice: start from geometric spacing and adjust based on the acceptance rate.
- Implement → verify the improvement in $\langle M \rangle$ / Binder → write the lessons on temperature placement into the source of truth too.

Task 2 is demonstrated live.

References

Acknowledgments

For fruitful discussions while preparing this lecture:

Jin-Guo Liu · Lei Wang · Satoshi Terasaki

References

CompPhysHack2026	The hackathon this lecture is built around (official site). See also the Terasaki hands-on material
Sustainable Automation Jin-Guo Liu	Discusses how to make AI collaboration sustainable across sessions, weeks, and projects with CLAUDE.md (memory) + Skills (procedures) + subagents, using the 100K-line ceiling of a C compiler as an example
superpowers	A workflow that enforces brainstorm → plan → execute as Skills (used in today's hands-on)
MIT missing-semester	Intro to agentic coding (the basics of CLI agents)