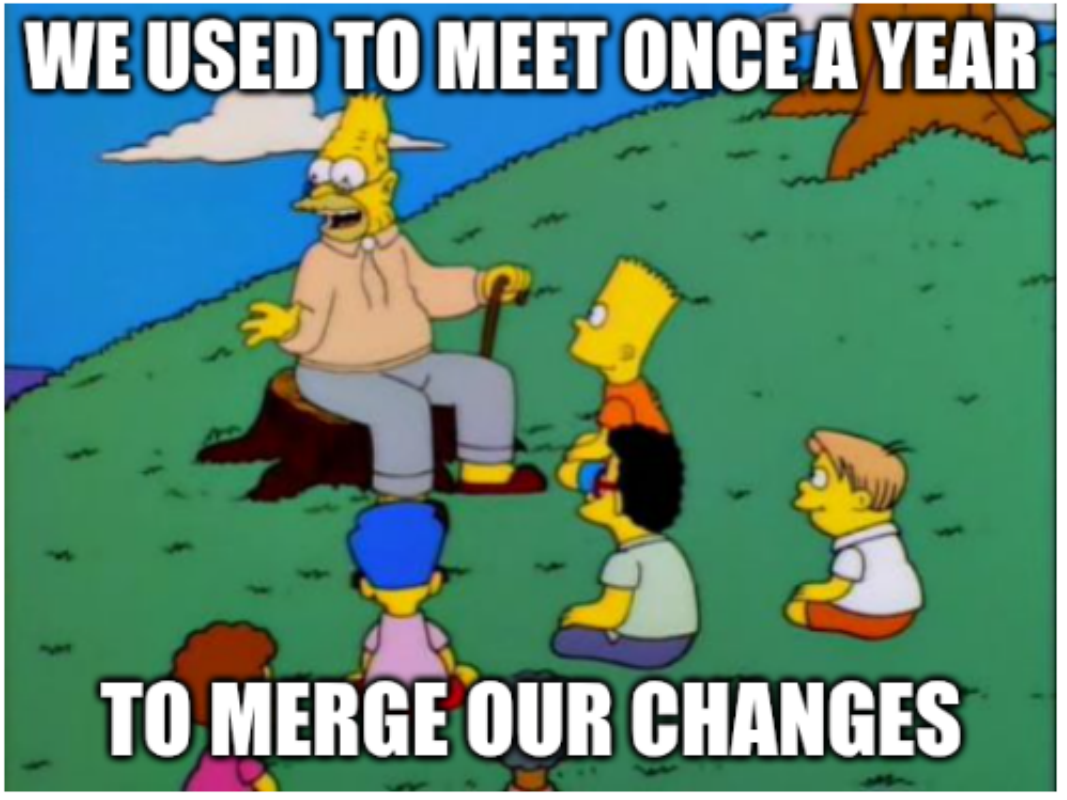
Software engineering with a research mindset

Lessons from both sides of the academy-industry divide

Radovan Bast, 2026 Nordic-RSE Conference

About me

- Chemistry 1999 — 2004
- PhD thesis in theoretical chemistry 2004 — 2008
- Postdoc 2008 — 2011
- Research 2011 — 2015
  - Started teaching the precursor for CodeRefinery
- Computing, programming, and teaching 2015 — 2025
  - Co-founded CodeRefinery, Nordic-RSE, and Research Software Engineering at UiT
- Oceanbox (oceanography startup) 2025 —


WE USED TO MEET ONCE A YEAR
TO MERGE OUR CHANGES

CodeRefinery

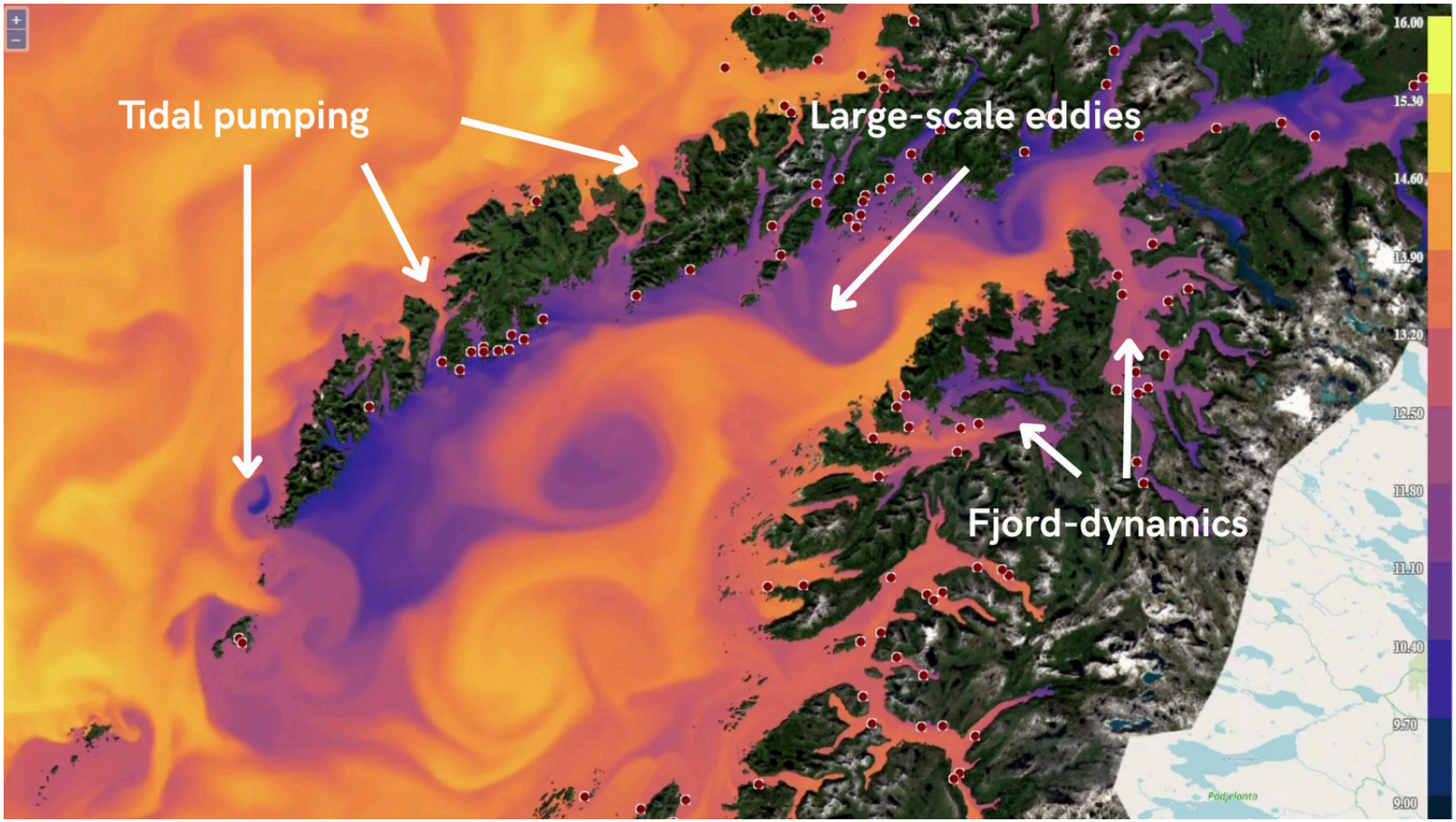
- Grew out of a week-long course taught at KTH Stockholm in 2014 and 2015.
- It was clear to us that we are filling a gap and that the demand goes beyond Stockholm and beyond Sweden.
- We were fortunate to have staff who were really motivated to share their knowledge.
- We were fortunate that we got support and trust "from above" (NeIC and partners).
- It started because there were people who encouraged me to.

How we started Research Software Engineering at UiT

1. Pitch the idea to the boss
2. Create a website
3. Talk about it and repeat yourself
4. Send an email to all department with "cake" in the subject
5. Invite yourself to other department seminars
6. Infoscreens
7. Show up with coffee, cookies, pizza around the campus

Most difficult part: marketing

Oceanbox: ocean current hindcast and forecast



Oceanbox: what I work on

Topics

- Computational geometry
- Interpolation
- Geospatial processing
- Ocean model
- Optimizing speed and memory with algorithms and data structures
- **Norwegian coastline**: "final boss" of computational geometry 👾

Tools

- Nix and NixOS
- Rust
- Programming for the command line and for **the browser** (Rust -> WebAssembly)
- Python
- Containers
- Fortran
- F#

... for the browser

Pros

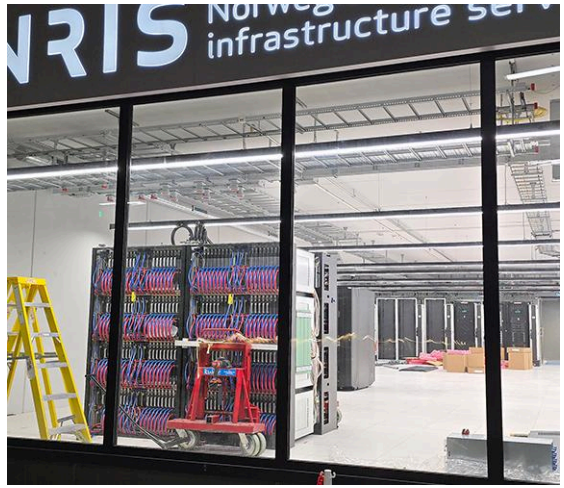
- Cross-platform
- Nothing to install
- Easy to ship updates
- Typically single-threaded: makes me think about good algorithms
- I have to be careful to not use too much memory
- Code must never crash
- Graphical: I can *see* whether something does not work

Cons

- File I/O a bit trickier than on the command line
- Debugging a bit trickier
- Typically single-threaded: makes me think about good algorithms
- I have to be careful to not use too much memory
- Code must never crash

~~Top 8 skills~~ 8 underappreciated skills that I wish I had known earlier

[https://uit.no/nyheter/artikkel?p_document_id=850450]



[https://www.sigma2.no/meet-olivia-norways-next-supercomputer]

MRI calibration "phantom":

[https://www.sunnuclear.com/products/acr-mri-phantom]

Calibration sample (**1**):

?

Why does nobody seem to ~~care~~ calibrate?

- "I am not a software engineer ..."
- "I thought this is already calibrated for me"
- Almost everybody has a computer in the backpack or the pocket.
- Programs produce numbers with decimals and numbers with decimals look correct.
- Fortunately we have now Research Software Engineers who are really good at this and can help.

Research mindset

Apply the scientific method to programming and debugging. In particular: hypothesis testing and experimental validation.

Approach the computational experiment the same way you would approach an experiment in your research area.

- Change one variable at a time.
- Bugs are phenomenons. Try to make the problem reproducible.
- Debugging is a series of experiments trying to falsify hypotheses by **simplifying the problem (2)**.
- **Prototype when scaling up (3)**: Test whether an idea is likely to work before investing lots of time into it.

Example for debugging as a series of experiments

"Our Monte Carlo simulation gives different means on my laptop and the cluster. The cluster result keeps changing too."

- are the code versions identical?
- are they using the same versions of libraries?
- check random seed
- force both to run on the same number of cores
- check compiler versions
- try to simplify and turn off features (**results will change** but now we want to know whether they change consistently)
- ...

Recipe for a systematic reduction (2)

1. **Make it reproducible**: Find the smallest input, shortest run, or simplest configuration that still shows the problem.
2. **Bisect in time**: When does it go wrong? Narrow the forecast hour, time step, iteration, or epoch until you have a tight window.
3. **Shrink the domain**: Reduce grid size, dataset size, number of steps, number of parameters. The goal is the smallest version of the problem that still fails.
4. **Turn off features**: Disable components one at a time.
5. **Bisect in change history**.

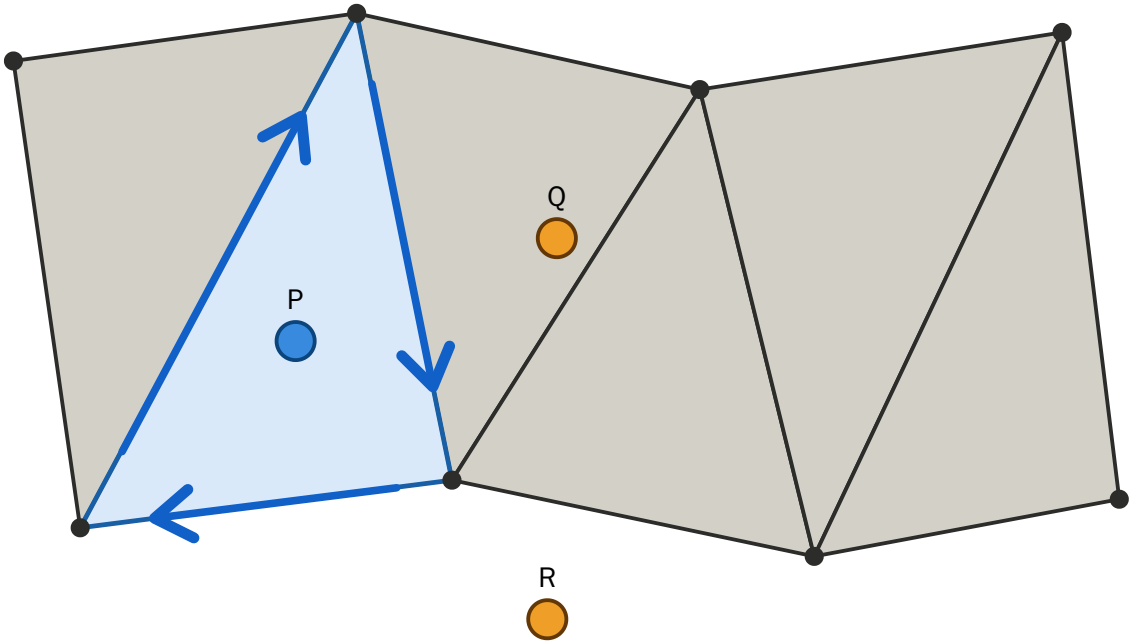
Rob Pike's 5 Rules of Programming

[emphasis mine]

1. You can't tell where a program is going to spend its time. Bottlenecks occur in surprising places, so don't try to second guess and put in a speed hack until you've proven that's where the bottleneck is.
2. Measure. Don't tune for speed until you've measured, and even then don't unless one part of the code overwhelms the rest.
3. Fancy algorithms are slow when n is small, and n is usually small. Fancy algorithms have big constants. Until you know that n is frequently going to be big, don't get fancy. (Even if n does get big, use Rule 2 first.)
4. Fancy algorithms are buggier than simple ones, and they're much harder to implement. Use simple algorithms as well as simple data structures.
5. Data dominates. If you've chosen the right data structures and organized things well, the algorithms will almost always be self-evident. Data structures, not algorithms, are central to programming.

[https://www.cs.unc.edu/~stotts/COMP590-059-f24/robsrules.html]

Example: point in triangle test



The test triangle is highlighted. Point P is inside. Points Q and R are outside.

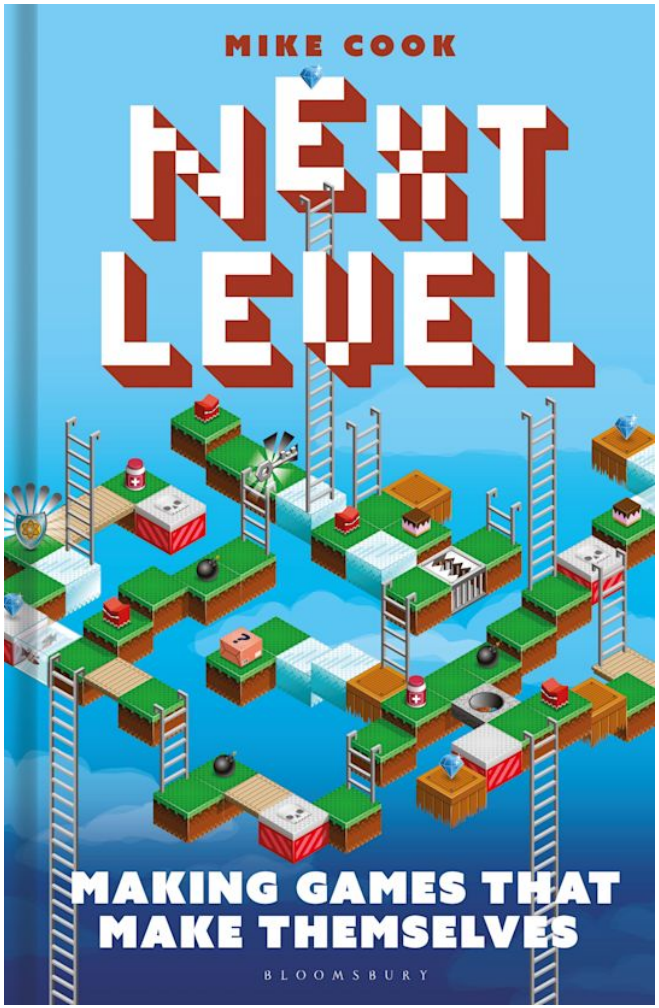
How I measure

```
// start the timer
let start = Instant::now();


// compute something that takes time ...
// ...


// print the result to the output (console log)
info!("this section took {:?}", start.elapsed());
```

- You can maintain several "stopwatches" in your code and at the end print a summary.

Examples for scaling prototypes (3)

- Random but valid DNA/protein sequences
- Simulated telescope time-series (light curves with injected noise)
- Random social graphs with realistic degree distributions
- "Lorem ipsum" variants with controlled vocabulary size and sentence length distributions
- Synthetic GPS traces following road-network-like constraints
- Random point clouds with known density profiles (for spatial indexing benchmarks)
- Generated or multiplicated polygons
- ...



[https://www.bloomsbury.com/us/next-level-9781399423427/]

Abstraction is a powerful tool to manage complexity

- **Vinaigrette**: Whisk together 1 part red wine vinegar and 3 parts olive oil. Add a spoonful of mustard, salt, and pepper and whisk until combined.

- **Caesar dressing**: Whisk together 1 part lemon juice and 2 parts mayonnaise. Add garlic, Parmesan, and a dash of Worcestershire and whisk until smooth.

- **Blue cheese dressing**: Whisk together 1 part white wine vinegar and 4 parts sour cream. Crumble in some blue cheese, add garlic and salt, and stir until combined.

```
let vinaigrette = make_dressing(
    RedWineVinegar,
    OliveOil,
    1,
    3,
    vec![Mustard, Salt, Pepper]);

let ceasar_dressing = make_dressing(
    LemonJuice,
    Mayonnaise,
    1,
    2,
    vec![Garlic, Parmesan, Worcestershire],
);

let blue_cheese_dressing = make_dressing(
    WhiteWineVinegar,
    SourCream,
    1,
    4,
    vec![BlueCheese, Garlic, Salt],
);

fn make_dressing(
    acid: Acid,
    fat: Fat,
    acid_parts: u32,
    fat_parts: u32,
    seasoning: Vec<Seasoning>,
) -> Dressing {
    let proportions = mix(acid, fat, acid_parts, fat_parts);

    combine(proportions, seasoning)
}
```

Functions (4)

- Rule of three: "Three strikes and you refactor"
- When teaching functions, IMO we put too much emphasis on reuse and duplication avoidance.
- I use and teach the "**rule of one**": almost all code should be inside some function even if the function is called only once.
- Functions help to collect related code and give it a name.
- It helps the human reading it to **minimize the context window**.
- It helps to **overcome memory bottlenecks** since it allows the memory management or garbage collector to release large arrays earlier (when leaving the function).

How do I recognize "good" functions?

```rust
/// combines an acid and a fat in a given ratio with seasoning to make a dressing.
///
/// # arguments
/// - `acid` - the acidic component, e.g. lemon juice or vinegar
/// - `fat` - the fat component, e.g. olive oil or mayonnaise
/// - `acid_parts` - relative amount of acid
/// - `fat_parts` - relative amount of fat
/// - `seasoning` - list of seasonings to add
fn make_dressing(
    acid: Acid,
    fat: Fat,
    acid_parts: u32,
    fat_parts: u32,
    seasoning: Vec<Seasoning>,
) -> Dressing {
    let proportions = mix(acid, fat, acid_parts, fat_parts);

    combine(proportions, seasoning)
}
```

- Good: reusable, understandable
- Does it fit in the screen? If not, it may not fit into my "context window".
- Pure functions are deterministic and more likely fit into my "context window".
- Types help both the human and the compiler.

Type safety: expect the unexpected

```
fn make_dressing(
    acid: Acid,
    fat: Fat,
    acid_parts: u32,
    fat_parts: u32,
    seasoning: Vec<Seasoning>,
    garnish: Option<Garnish>,
) -> Result<Dressing> {
    if acid_parts == 0 || fat_parts == 0 {
        return Err(Error::InvalidRatio);
    }

    let proportions = mix(acid, fat, acid_parts, fat_parts);

    Ok(combine(proportions, seasoning, garnish))
}
```

- Option/Maybe: helps to deal with values which can be missing. Much more robust than to agree that e.g. -9999.0 means "missing".
- Result: communicates that we anticipate failures and the type checker will make sure that we don't forget to deal with them.

Move uncertainties to the command-line interface (5)

😬❌

```
# ... lots of code

df = pd.read_csv("/home/radovan/myproject/data.csv")

# ... lots of code

# DRY_RUN = False
DRY_RUN = True

# ... lots of code

NUM_RIVERS = 1000
# NUM_RIVERS = 2000

# ... lots of code
```

😊✅

```
$ myscript --data /home/radovan/myproject/data.csv --num-rivers 1000 --dry-run
```

```
$ myscript --help
```

Containerizing (6)

An exercise in **decoupling/isolating** an application from your file system and structure with the hope that it will run on a different computer (**portability**) and/or in future (**reproducibility**).

- The result is a definition file ("recipe").
- Reproducibility and portability is not easy.
- Even an imperfect definition file can serve as super useful documentation.

The two hardest problems in computer science are ... (7)

The two hardest problems in computer science are ... (7)

Hazel Weakly
@hazelweakly.me

I've posted it before, but it feels evergreen

The two hardest problems in Computer Science are

1. Human communication
2. Getting people in tech to believe that human communication is important

> **Jake Anbinder** @jakeanbinder.bsky.social · 2mo
> I think we need an inverse thread where we talk about things that normal intelligent people universally agree upon that are for some reason hot takes within your profession

8:46 AM · Mar 31, 2026 Everybody can reply

275 reposts **16** quotes **1K** likes **68** saves

[https://bsky.app/profile/hazelweakly.me/post/3midldzhcdc26]

The XY problem

A — B — **Y** — ? — ? — **X**

- The XY problem is asking about your attempted solution or current obstacle without also explaining your actual bigger picture problem/goal

The XY problem

A — B — **Y** — ? — ? — **X**

- The XY problem is asking about your attempted solution or current obstacle without also explaining your actual bigger picture problem/goal

The "100 X problem"

A — B — **X** — ? — **100 X**

- We agree to implement X
  - "Can you help me implementing X?"
  - "No problem! This should take me 1-2 days."
- Two weeks later:
  - "X is ready, please test it! Here is an example ..."
  - "Thanks! Your example works. But when I try my example it fails with ..."
  - "Let me look at your example ... Oh but it's 100 times larger!"
- We forgot to discuss the expected scale

What can academia learn from industry (8)?

- Maximizing value: "What would create most value?"
  - Value does not have to mean money or shareholder value. It can mean freeing up time for my colleagues.
  - We use this question to prioritize tasks for the day, for the next week, for the next month.
- User focus: Optimizing for user experience
  - Usability
  - Asking users for feedback and using this feedback
  - Documentation
  - Support

Software industry is in an existential crisis

A — **?** — X

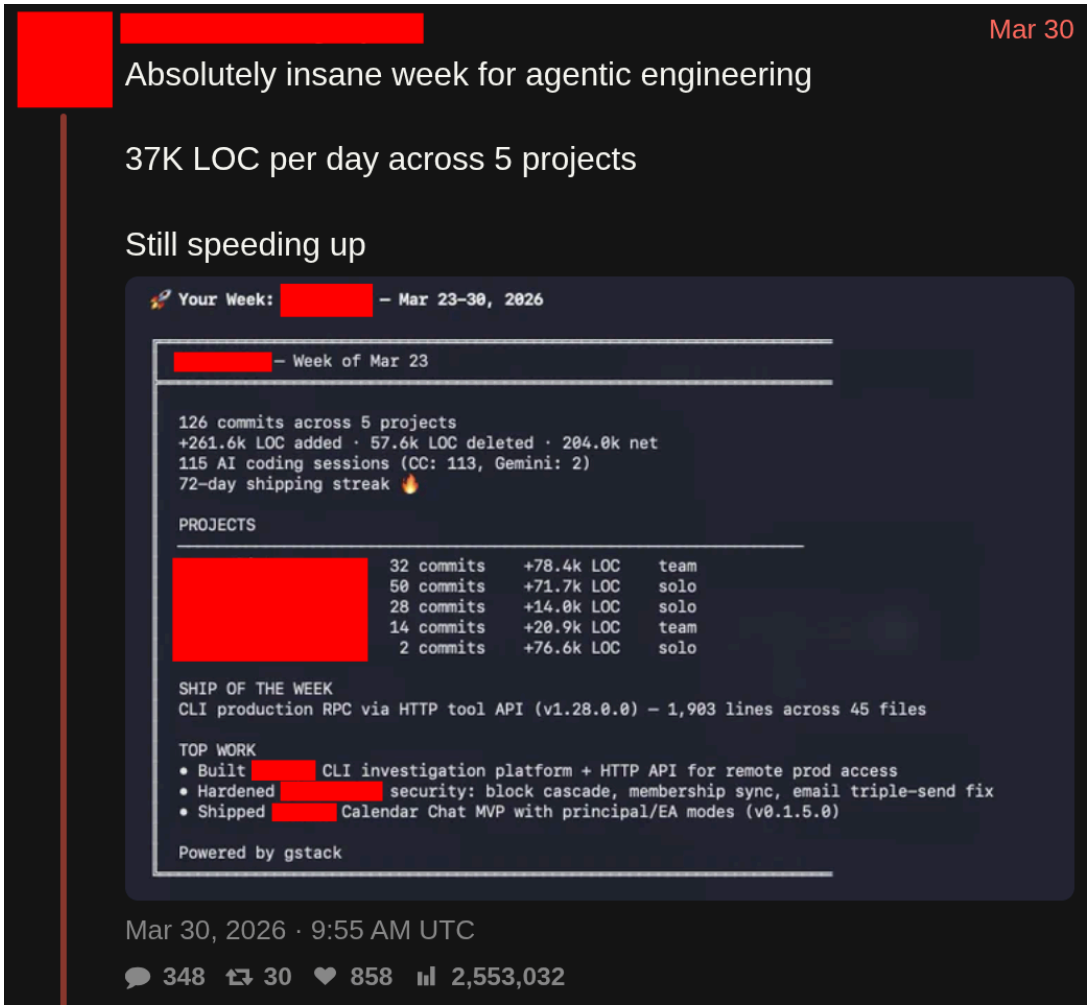
- Majority of code might be written by LLM agents
- We are not only delegating tasks but **delegating judgement** without delegating accountability
- How will this affect **education**?
- How will this affect **code complexity**? "LLMs do not feel a need to optimize for their own (or anyone's) future time"

  [https://bcantrill.dtrace.org/2026/04/12/the-peril-of-laziness-lost/]

Software industry is in an existential crisis

- Will software libraries go away?
  - "[...] libraries are not just code, they're time invested in edge cases, they're hundreds of hours of human eyes. I don't want to replace that."

    [https://bsky.app/profile/vickiboykis.com/post/3mkpmimhnls2q]
- Will my job go away?
  - Is my education still needed?
  - Maybe we don't need 10-20 years of "scar tissue" to write good code?
- What if we forget how to program and ...
  - ... we are no longer able to make an educated Yes or No?
  - ... nobody can call the bluff on hallucinations?
- What will we do if LLM agents become prohibitively expensive?



Excellent blog post: "**The peril of laziness lost**" (Bryan Cantrill):
https://bcantrill.dtrace.org/2026/04/12/the-peril-of-laziness-lost/

Summary

- Humans have a context window, too.
- Wrap almost all code into functions, unless it is a 20-line shell script. It will help computers and humans to save memory.
- Let's spend some time thinking about good calibration samples. This process will help during the next debugging session. Start creating end-to-end tests.
- Thanks to AI, testing and documentation are experiencing a comeback.
- If in doubt, make it a command-line argument.
- The ability to depart from a real system towards simpler or towards more complex is a powerful skill.
- Reproducibility is not only for academia.