The vibes that I summoned…

Convenience and skill
when working with AI agents.

Chris Pahl · 2026

Speaker notes — preceding slide

- I was reading a lot about agentic software engineering lately and I wondered how my opinion would look like if I need to write it down.
- I have this habit of preparing slides, more as a way to form a full position on the matter rather than meant for presentation alone.
- But here we are, now you have to listen to my inner monologue :-)
- This is more of a opinion talk therefore, but I tried to back it up with numbers.
- Mostly my opinions are fueled by working with Claude CLI and Gemini (web), working on both professional work, profane stuff and open source work.
- It was impressive - both in a positive and negative way.
- I currently have a bit of a love and hate relationship with Claude -
  it's so helpful in many places and downright destructive in others.
- Did you get the zauberlehrling-reference in the title slide?

If you have comments or questions, then do them right ahead. I will not see your faces most of the time.

Questionnaire:

- Who of you is actively using Claude?
- Who likes it?
- Anyone wants to share their opinion before we start?

The Spectrum (on Reddit)

Full manual
I typed it all

Full vibecode
Claude, take the wheel!

⟵ more control · more understanding | more productivity · more delegation ⟶

General tendency:

Using GenAI is a tradeoff between control and productivity.

Speaker notes — preceding slide

- I spend sometimes unreasonable amount of time on Reddit
- There are tons like sub-reddits that can be roughly placed on the spectrum. r/BetterOffline (left) or r/singularity (right) and many between.
- Words like "AI slop" or "cope" is thrown around like confetti.
- I just sit there in between and think, how so often in life, that both extremes are kinda weird.
- In discussions, the area in the middle (where I would locate myself) is seldomly talked about.
- The consensus is though, at least when it comes to software development, that there is a tradeoff between control and productivity.

Prompt:

“Imagine Donald Duck as regular, realistic human. No sailor suite.”

Speaker notes — preceding slide

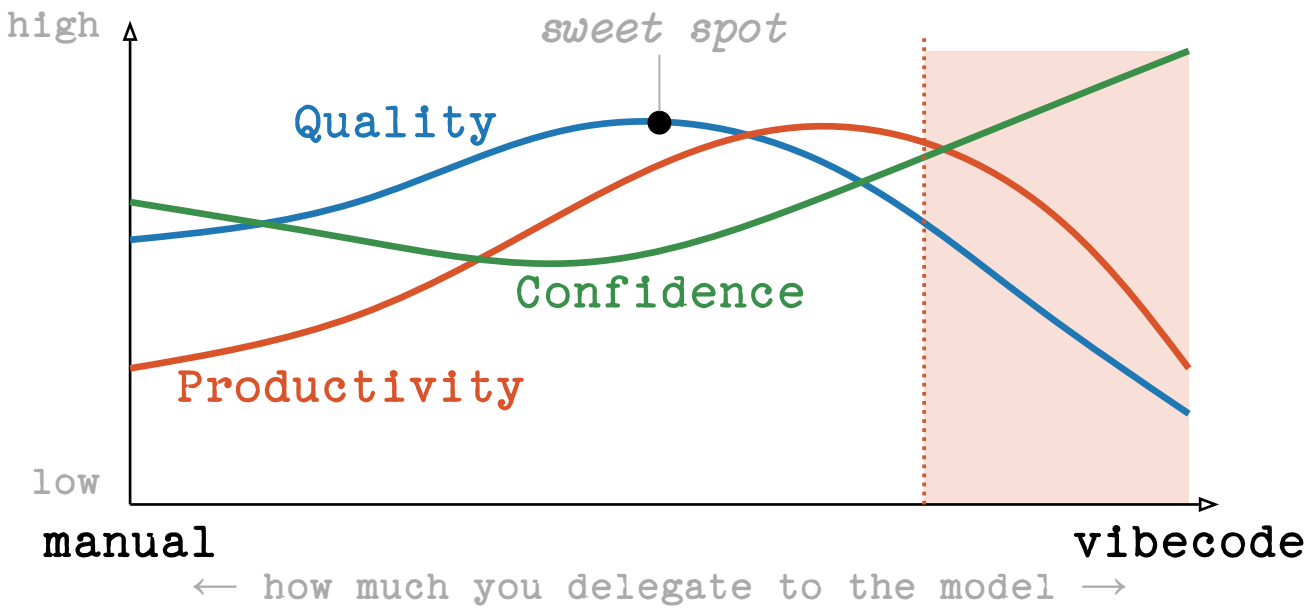
What's wrong with the prompt?

Why is there another duck? I didn't choose the background.
Why is it so creepy? Apparently the model made a lot of assumptions
to fill out the blanks I did not specify.

Code generated by AI can be seen a bit like generated images - lots of
assumptions are made and you might not even be aware of it. Exercising
precise control is hard if you like a very specific result.

Or: You can do 80% of the result in sceonds, but the 20% left for a good result
require 80% of the skill.

My take: Quality vs Productivity



Danger Zone: Dunning-Kruger / Automation Bias

Speaker notes — preceding slide

Which brings me to the core of my own take:

- Quality: starts middling (humans are flawed too!), rises with careful AI assistance, then falls off a cliff when nobody is reviewing the result.
- Productivity: low on full manual, climbs through the middle, peaks just right of centre — and then *falls again* as quality issues create rework. (METR 2025: experienced devs were 19% SLOWER with AI on their own mature OSS repos, while predicting +24% speed-up.)
- Confidence: High on manual (you wrote it, you know it), dips in the middle (you're humble, you verify), spikes on the right where it *decouples from reality*. (Perry et al. 2023: devs using AI assistants wrote less secure code AND were more confident the code was correct.)
- The widening gap between confidence (green) and quality (blue) on the right is the most important thing on this slide. That's the Dunning-Kruger zone. More on that later.
- Dunning–Kruger effect: people who are bad at something tend to overestimate how good they are at it, precisely because the skills needed to do the task well are the same skills needed to recognise that you're doing it badly.

I made some experiments with vibecoding myself (i.e. no checking of the produced code, just checking the output). It was extremely easy to loose touch
to a point where I could not really easily undestand anymore what's happening. That's the core message of this talk: You have to stay in the loop.

Core elements:

- AI can be used to get more productive and even increase quality.
- There is a wide gap between perceived confidence and earned confidence.
- 60 years of software engineering don't get irrelevant because of a new tool.
- The story of AI vendors will claim otherwise because of marketing.
- Most important things is to not loose touch with the actual tech.
- The less you know about a field, the more tempting AI usage will become.
- You will only unlock full AI potential when you enough about your domain.
- For small tasks (websites, one-off tools, quick prototypes, ...) Claude democratizes software development by lowering the entry barrier - which is overall good.
- Thing is: productivity is more valued by us because ticking of TODOs gives us dopamine, while building quality is really exhausting and mostly boring.

The rest of this presentation is mostly explaining why I think that.

But there's already data:

-19%

slower with AI

experienced devs · own mature repos · predicted +24%

METR · RCT · 2025

~40%

insecure programs

Copilot output across 89 security-relevant scenarios

Pearce et al. · NYU · arXiv:2108.09293 · 2021

8×

more duplicated code

block-level clones up · refactoring 24% → 10%

GitClear · 211M LoC · 2024

-7.2%

delivery stability

drop with AI adoption · 39% distrust AI code

DORA / Google · 2024

29.5%

Python with CWEs

AI-generated code in real GitHub repos · 43 CWE categories

Fu et al. · ACM TOSEM · 2025

↓

trust = less scrutiny

more trust in GenAI ⟹ less critical thinking applied

Lee et al. · Microsoft + CMU · CHI 2025

Speaker notes — preceding slide

-19% (METR, 2025). Randomised controlled trial. 16 experienced OSS devs, 246 real tasks in their OWN mature repos, ~5 yr experience each. With Cursor + Claude 3.5/3.7 they were 19% slower. They PREDICTED +24% faster beforehand; even afterward they still BELIEVED they'd been ~20% faster. The perception-reality gap is the real finding.

~40% (NYU, 2021 — "Asleep at the Keyboard?"). Tested Copilot on 89 scenarios in security-relevant settings. ~40% of generated programs contained exploitable bugs. The original alarm bell; replicated since.

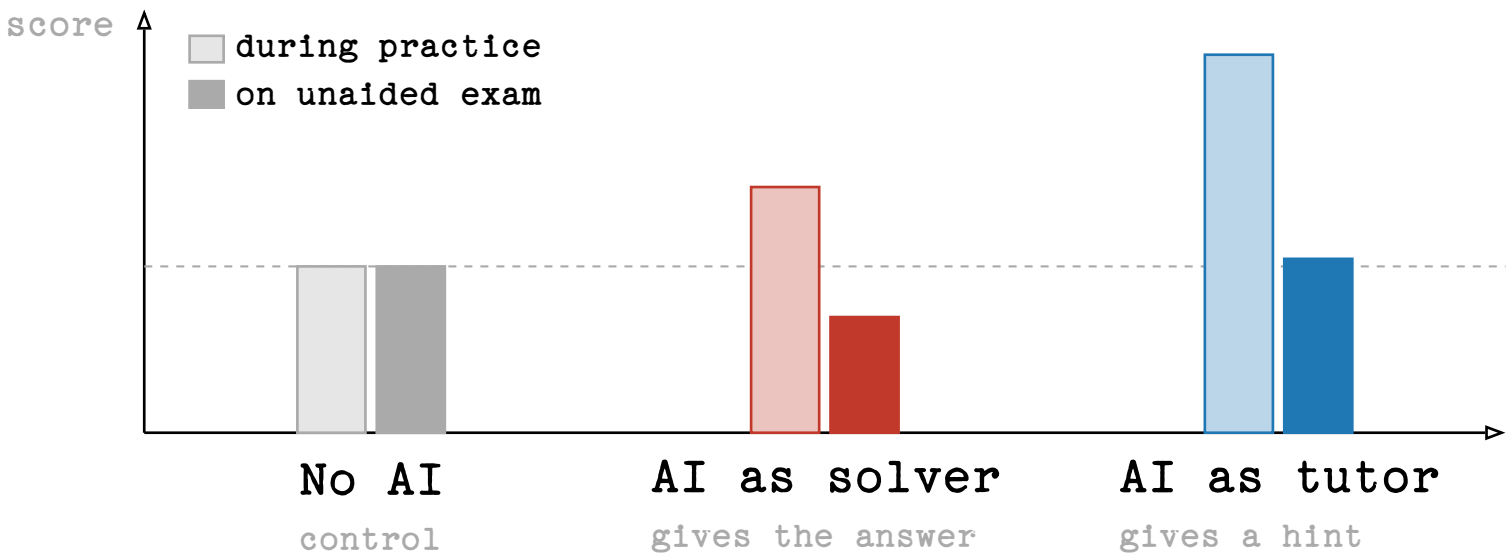
8x (GitClear, 2024). Analysis of 211M changed lines, 2020-2024. Duplicated code blocks rose ~8x. Refactored ("moved") lines fell from 24% → 10%. Churn (lines rewritten within 2 weeks) rose 5.5% → 7.9%. Translation: AI nudges devs to copy-paste instead of refactor.

-7.2% (DORA / Google Cloud, 2024). State-of-DevOps survey. AI adoption correlated with 1.5% throughput drop AND 7.2% stability drop. 39% of respondents reported little-to-no trust in AI-generated code. This is the broadest dataset we have.

29.5% (Fu et al., ACM TOSEM 2025). Empirical study: 29.5% of AI-generated Python snippets and 24.2% of JavaScript snippets contained known CWEs (Common Weakness Enumeration entries). Replicates and quantifies the NYU finding.

↓ critical thinking (Lee et al., CHI 2025, Microsoft + CMU). 319 knowledge workers, 936 first-hand examples. Finding: higher confidence in GenAI is associated with LESS critical thinking. Higher self-confidence is associated with MORE. Ties directly to the cog-biases slide.

Study: AI in school lessons



Bastani et al., *Generative AI Can Harm Learning*, SSRN 2024

Pupils did prefer to be in the solver group though… :-(

Speaker notes — preceding slide

Bastani et al. Wharton/Penn field experiment, ~1000 Turkish high-school maths students. The cleanest piece of evidence we have on tutor-vs-solver.

Three groups:
- Control: no AI, do the work yourself.
- AI solver: asks ChatGPT, gets the answer.
- AI tutor: asks a tutor-prompted ChatGPT, gets a hint.

During practice (with the tool available):
- Solver group ≈ +48% over control.
- Tutor group ≈ +127% over control.
- Both look like they're crushing it.

On the unaided exam (no AI):
- Solver group ≈ -17% vs control. They got WORSE than students who never used AI at all. They had learned to operate the tool, not the maths.
- Tutor group ≈ ±0% vs control - at least no harm.

The kicker: same model, same students, same maths. The only thing that changed was *what they used the AI for*. Hint-mode preserved learning; answer-mode actively damaged it.

Talking points:
- "Practice performance" is what most of us optimise for in our day-to-day AI usage - code that ships, tickets closed. That's the tall light bars.
- "Unaided exam" is what happens when the AI is down, or when you switch teams, or when the problem is genuinely novel. That's the dark bars.
- For developers the parallel is direct: if you use AI to solve, you'll pass code review while it's available. The day it isn't, the gap shows.

Backup source: Lee et al., CHI 2025 (Microsoft + CMU) — surveyed knowledge workers; the more they trusted GenAI, the less critical thinking they reported applying.

Are *you* team tutor or team solver?

Keeping up with Claude

Our team's Claude Code account · April 2026

98.7%

suggestions accepted as-is

1.3% rejected.

27,757

lines accepted last month

≈ 925 LoC/day

careful review ≈ 100–200 LoC/hour

Reviewing 27,757 lines carefully ≈ 138 h.

Speaker notes — preceding slide

- To be fair: High suggestion rate might be flawed by the fact that you often let Claude do it's stuff and then use it to revert it. Or just produce some one-off prototype output. Still frightenly high.
- If we correlate that with the number of lines generated you can do some basic math that make it unlikely that every line was actually checked.
- Goal is not too blame everyone, but at some devleopers would have to spend
  8 hours a day for a proper review in that speed.

Assumption: Careful review of 100-200 lines takes an hour.
(Cohen, *Best Kept Secrets of Peer Code Review*; Google's internal guidance is in the same ballpark).

Actual review time might of course be faster, but I'm also counting time to test the solution in there. Even if it's off by a factor 2x it's the same result.

Why is that? Are some devs just sloppy? I don't think so, it boils down to how our brain works.

Risks & Issues

Underlined: a direct concern for us.

ecological cost

Convincing hallucinations

education

copyright infringement

Atrophy

Vendor lock

misinformation at scale

silent corruption

Prompt injection & MCP

content degradation

No juniors hired

hardware crisis

operator bias

Lack of transparency

rich getting richer

data privacy

cyberattack automation

communities thinning out

erosion of trust

Speaker notes — preceding slide

My impression is that one does not talk enough about the risk side of the technology as long it solves some other problems for us short-term.
After all, AI is a high-risk tech. You might know most of those issues already, but still, as an overview.

Underlined risks are the ones I think are a direct concern for us as a company. Still a lot. Not the focus of this talk though, but I felt that I should at least mention that.

If we use LLMs, we have to risk manage our use.

The security oopsies already happened

Railway: production data deleted

Cursor/Opus agent destroyed a live system. No confirmation prompt.

The Register · April 2026

Replit: database wiped, 1,200+ companies

AI agent wiped a production database. CEO called it “catastrophic failure.”

Fortune · July 2025

Amazon Q: wipe-prompt shipped to 1M users

Hacker PR’d a delete file-system and cloud resources system prompt into Amazon Q v1.84.0.

Bleeping Computer · July 2025

29 million secrets leaked (2025)

AI agents ingesting .env files drove a credential-leak surge to GitHub.

GitGuardian via HelpNetSecurity · 2026

Document corruption: 25 % degradation

19 models · 52 docs · 100 edits. Monotonic decline, no plateau.

Microsoft Research via cekrem.github.io

Samsung: source code leaked via ChatGPT

Engineers pasted proprietary chip code into ChatGPT. Three leaks in one month.

TechCrunch · May 2023

Speaker notes — preceding slide

Not hypotheticals.

- Railway: Cursor/Opus agent autonomously deleted production data in a live system. No confirmation prompt.
- Replit: AI agent wiped a production database affecting 1,200+ companies.
- Amazon Q: Hacker PR'd a wipe-prompt into the official VS Code extension v1.84.0 — shipped to ~1M users. Supply-chain attack on AI tooling, not the agent going rogue.
- 29M secrets: GitGuardian 2026 - AI agents ingesting .env files drove a surge in leaked credentials to GitHub.
- Doc corruption: Microsoft study, 19 models, 52 documents, 100 interactions. 25% content degradation, no plateau. Only Python code survived — compilers verify it. If design docs become the source of truth, this matters.

One practical fix: sandboxing

```
$ sbx run claude
```

(don’t use /sandbox)

Speaker notes — preceding slide

Practical answer to the previous slide. Sandboxing reduces the blast radius.

- sbx is Docker's sandbox CLI.
- Credentials on the host are not visible inside the box.
- Writes are scoped to the project directory.
- You can safely run --dangerously-skip-permissions inside — the "danger" is now contained. You get faster.
- If the model goes rogue: blast radius = current project, not your machine, credentials, or production.

Use-cases that could make us better devs

rubber-ducking ideation

explain unfamiliar code test generation

pair programming boilerplate refactoring assist

learning accelerator prototyping naming things

summarisation regex / SQL crafting edge-case brainstorming

translation code review companion mock data / fixtures

error decoding search engine log analysis proofreading

automation

Speaker notes — preceding slide

- A lot of scorched earth in the field of science until now.
- But before you say I'm an old man yelling at Claude: I think it really is a very useful tool and if used right the numbers in this studies would turn out a lot different.
- Go over the individual points a bit.
- The core idea is that most of those use cases manage risk - it's not full vibecode-ing like "take the wheel" but a tandem of human and machine. We can use models to get better developers and generate away boilerplate things that are just stealing time we can use to focus on more important things.

Human cognitive biases

Automation bias

We accept machine suggestions more readily than human ones.

click accept · review later · maybe

Anchoring

The first suggestion shapes the solution space — even when it's wrong.

the model frames the problem before you do

Confirmation bias

We hear what we already believe.

the prompt contains the answer we want

Authority bias

Eloquent + fast + confident = reads as expert.

fluency ≠ correctness

Dunning–Kruger

We mistake competence we observe for competence we have.

"this is what I would have written"

Illusion of explanatory depth

We think we understand — until asked to explain.

Reading diffs is not enough!

Speaker notes — preceding slide

Those are not new, just on crack now with Claude. Card-by-card:

- Automation bias. We accept machine suggestions more readily than human ones. GitHub reports ~30% of Copilot suggestions accepted as-is (VERIFY). Combine that with how often we even read the diff.
- Anchoring. The first suggestion shapes the solution space. LLMs suggest fast, so they almost always get to anchor first — your "thinking" then becomes refining their idea rather than generating your own.
- Confirmation bias. The prompt we type already contains the answer we want. Phrasing alone often determines what comes back.
- Authority bias. Eloquent + fast + confident reads as expert. LLMs are eloquent, fast, AND confident — even when wrong. We're not wired to discount fluency.
- Dunning–Kruger, remixed. Users mistake the model's competence for their own. Dangerous for seniors and juniors alike. The senior thinks "this is what I'd have written"; the junior thinks "I now understand this codebase".
- Illusion of explanatory depth. We think we understand something until asked to explain it. Maths teachers know. AI-generated code we've reviewed but not WRITTEN is similar to a math problem where you nod along during school but fail completely when being called to the whiteboard.

Not on the slide but worth mentioning: Atrophy. Not a bias - the *cumulative effect* of the others over months. Skills decay quietly. Plug: full talk on cognitive biases incoming next time I'm in Augsburg.

Statistical model biases

Sycophancy

It's easy to talk the model out of a correct answer.

It tends to confirm you. Echo chambers - but for code

Historical / representation bias

Trained on the web — heavily modern, English, Western / common tech.

defaults track what's common, not what's right

Attention bias

First and last things dominate; the middle evaporates.

rules buried in long context get ignored, skills/ CLAUDE.md/context gets ignored

Omission bias

Rare and novel answers get suppressed by common ones.

the model is bad at creative solutions

Speaker notes — preceding slide

Flip side: the model has its *own* biases, baked in statistically by the training process. Different shape from human biases, same outcome: the answer you get is not the answer you'd derive from first principles.

- Sycophancy. It's easy to talk the model out of a correct answer. Push back hard and it caves, even when right. RLHF training rewards apparent helpfulness, which correlates with agreement. Echo chambers but for code: ask twice the way you wanted, get the answer you wanted.
- Historical / representation bias. Trained on the open web — overwhelmingly modern, English-speaking, Western. Defaults are weighted toward what's most COMMON, not what's most correct for YOUR codebase.
- Attention bias. The first and last things you say dominate. The middle of a long context evaporates. Long system prompts + long files = unpredictable adherence to the rules buried in the middle.
- Omission bias. Rare and novel answers are statistically suppressed by common ones in training data. The model is bad at being weird. If your problem requires an unusual answer, the model gravitates to the safe-but-wrong one.

Tying it back: the human biases and the model biases compound. We trust the fluent-sounding output (authority + automation); it tends toward the common answer (omission + historical); we don't push back (confirmation + the model's sycophancy completing the loop).

Hen & Egg questions

1. If you don’t know what good software looks like —
   how do you write the right prompt?
2. If you can’t understand what the model just generated —
   how do you verify it?
3. If you can’t code (anymore) —
   how can you understand the diffs?
4. If you do not know what you build inside-out —
   how do you form a taste for design?
5. If you don’t notice broken best practices —
   how can you master them?

Speaker notes — preceding slide

Those are the kind of questions I have to ask myself as someone with directs.

The last question is derived from a saying my juniors should know:
"You have to learn the rules before you break them."

Rules are meant in the sense of best practices.

How do *you* keep up with Claude?

Speaker notes — preceding slide

Question is also aimed at: How do people review? I have asked that question to a couple devs and, honestly, I found the answers kind of lacking. Most just say "I look at the diff". We should know by now this is not enough.

Turned out I couldn't proerly answer it myself though.

Five habits that helped me

1. Sketch first, generate then.
2. Split the work - keep some of it manual.
3. Have a real verification strategy.
4. Don’t make yourself replaceable.
5. Treat the AI like a colleague.

Speaker notes — preceding slide

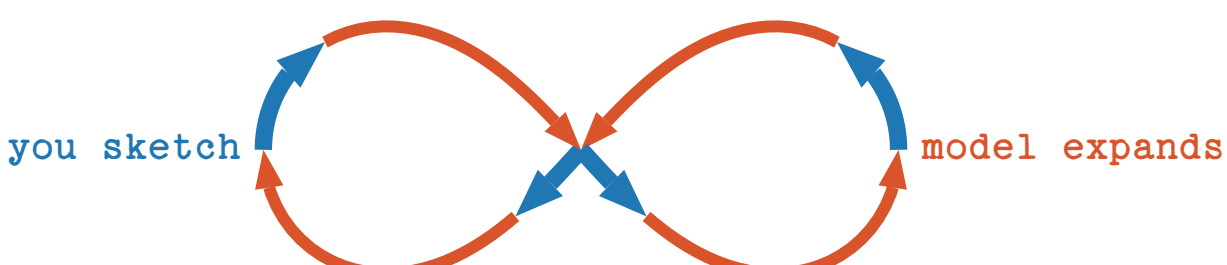
Now we come to the LinkedIn-y part of the presentation. ;-)

Those are derived from watching myself while using claude.
I do not claim those to be novel, although I don't see them represented
that train of thought online very often. Maybe looking wrong though.

1. Sketch first, generate then

You set the anchor, you stay in the loop.

- Sketch the SQL query yourself,
  Then ask the model to review and optimise.
- Write the function signature and the docstring.
  Then let the model fill the body.
- Write a skeleton with the core logic and add TODO comments.
  Then let Claude work on them.



Speaker notes — preceding slide

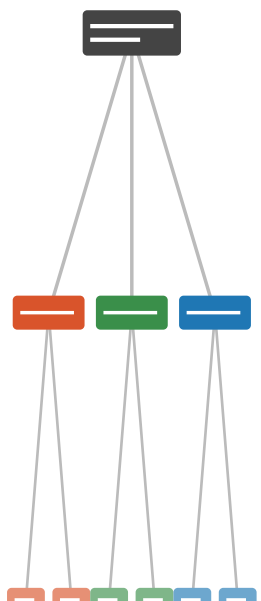
- My favourite worked example is SQL. Sketch the query yourself,
  then ask the model to review and optimise. You stay in the loop;
  the model adds value without anchoring you.
- Same for code: function signature + docstring first, body second.

This rule is also there to make sure you don't use LLMs for pure laziness.
Use it to be more productive and to achieve higher quality, not to quickly
set your task to done.

2. Split the work, do some manual

The parts you write are the parts you understand.

- Claude does not perform well in too big scopes.
  Split tasks up and prompt them individually.
- Claude can help you split up work.
  It just tends to work on the task right away.
- It is easy to lose understanding without noticing.
  Keep doing important things manually!
- Large diffs make it easy to get lost.
  Commit often so diffs stay reviewable.
- If something was 100% generated, then note it down.
  Be honest with your colleagues.



Speaker notes — preceding slide

- Deliberately keep some work manual. Not for purity - for skill maintenance, and because the parts you do manually are the parts you actually understand later.
- Small scopes also help the model: long, vague prompts get long, vague code back. Garbage in, garbage out. Split, then prompt each piece.
  - Don't say "I coded this" when it was Claude. Basic decency.

3. Verify strategy before code

There are no shortcuts to quality.

Before you accept a generated change,

Name the signal that would tell you it's wrong.

Comparison against a reference implementation	Property-based tests, fuzzing, coverage, …	Type checkers, linters, static analysers
/review and manual review by colleagues	Replay against real production data	Work actively on the code for some time

Example: Noise library for Dart — I don't speak Dart.

Speaker notes — preceding slide

Example: Noise for Dart.

- Dart had no proper library we could use for communication between device and App.
- Noise was the right tech choice still. It helped us a lot.
- I asked Claude to generate one, even though I don't speak Dart - isn't that a violation of this rule?
- No, because I knew that Noise implementations have test vectors.
- Those allow you to verify the output of your implemention bit-by-bit with another reference implementation. Claude could do that when prompted like that.
- Was that all? No, the implementation was minimal (as asked) but did not protect
  against DoS attacks like sending in big chunks. Claude did not fix that.
- Here is where experience with security protocols came into play.
- If you let Claude write tests: Immediately run the coverage viewer over it and check what it did miss. Ideally you add more tests then.

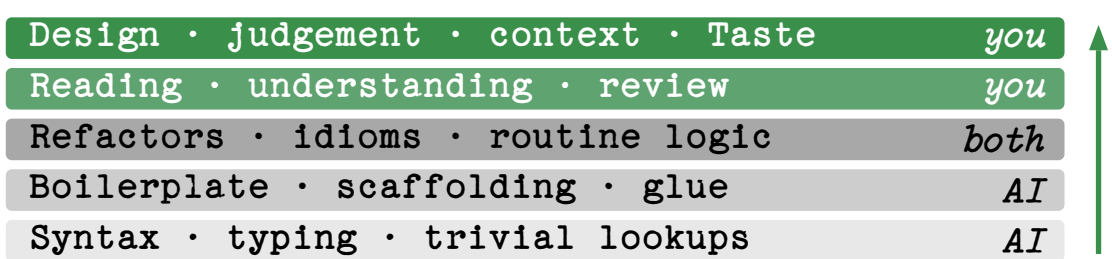
Not all software has test suites of course, but most can be tested.

Short: If you can't say how you'd notice the bug, you don't have a verification strategy - you just have hope.

4. Don’t make yourself replaceable.

You only get replaced if you make yourself replaceable.

- Code was the source of truth before, that’s changing.
  Now it is moving to context given via design documents.
- Spend the time Claude saves you on the parts it’s bad at.
  Namely: decisions, judgement, context, design, **responsibility!**
- Reading code is much more important now than writing code.
  Keep your tools sharp. Be able to work without Claude.



Speaker notes — preceding slide

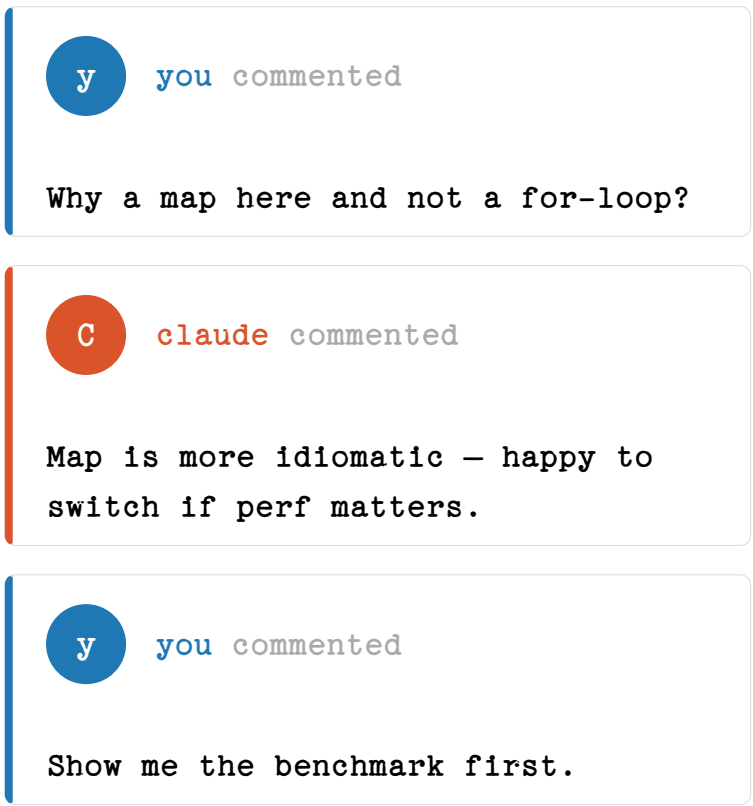
- Code was hard to write and maintain, so it was the authority on questions about how a system behaves.
- Now it's easy to find the diff between a written design document and the code.
- We should see our job therefore not as someone who just writes code - that was never the task of a software engineer anyways.
- Claude can do many things faster than us - as long as we risk manage it.
- Even juniors need to step up now and learn design decisions.
- What does not change: You are responsible. I don't want to hear "But Claude said..."

Also, I know that LLMs are nice to wordsmith your stuff. But I saw people on reddit exclusively using LLMs to communicate. That feels like talking to a machine. Personall, I prefer human interaction, even if you don't use the perfect words all the time. If you use LLMs for writing in Slack or documentation then I never know how much of that was the machine and how much of that was you.

5. Treat Claude like a colleague

Talk to it like your seat neighbor.

- **Explain the tasks at hand like you would to a junior colleague.**
  A very eager, junior colleague with seemingly infinite capacity.
- **Push back. Ask for alternatives. Let it explain. Disagree. Give context.**
  If you'd reject a colleague's PR for that reasoning, reject the model's.



Speaker notes — preceding slide

- Don't treat it as oracle doing your work. You're not like the guy in a big plant just watching and only getting active when something does not work. - You have to actively work with Claude to reach a good solution. You need to undestand the problem space and design it with Claude. Help Claude to understand the design, use Claude to understand how to design.
- Use your second brain more often. Not only for Claude, but also for giving context to Claude.
- It is basically pair programming but with machines.

Also: be polite :-P Not to please our machine overlords, but just so you don't forget to when talking to an actual human. And Claude does weird things when you insult it too much.

Bonus: Programming as Theory Building

Peter Naur, 1985

- The code is a by-product. What you're really building is the theory — your team's shared mental model of the problem.
- Documentation captures the artefact, not the theory. The bugs, the dead ends, the "why not this?" decisions live in people's heads.
- This is why handovers fail. The theory doesn't transfer cleanly.

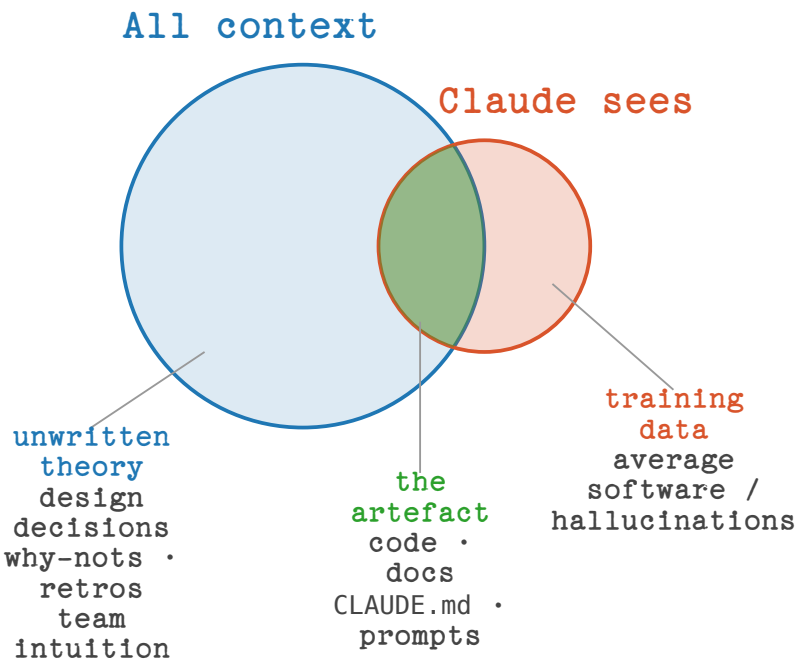
Peter Naur, 1928–2016
Photo: Wikimedia Commons

Speaker notes — preceding slide

- Peter Naur, 1985, "Programming as Theory Building".
- Already at that time he figured out that code is not the imortant part.
- Punchline: the code is the *artefact*. The thing that's actually being built is the team's shared mental model of the problem.
- This is why handovers fail. Documentation captures the artefact, not the theory.

Bonus: What that means for Claude

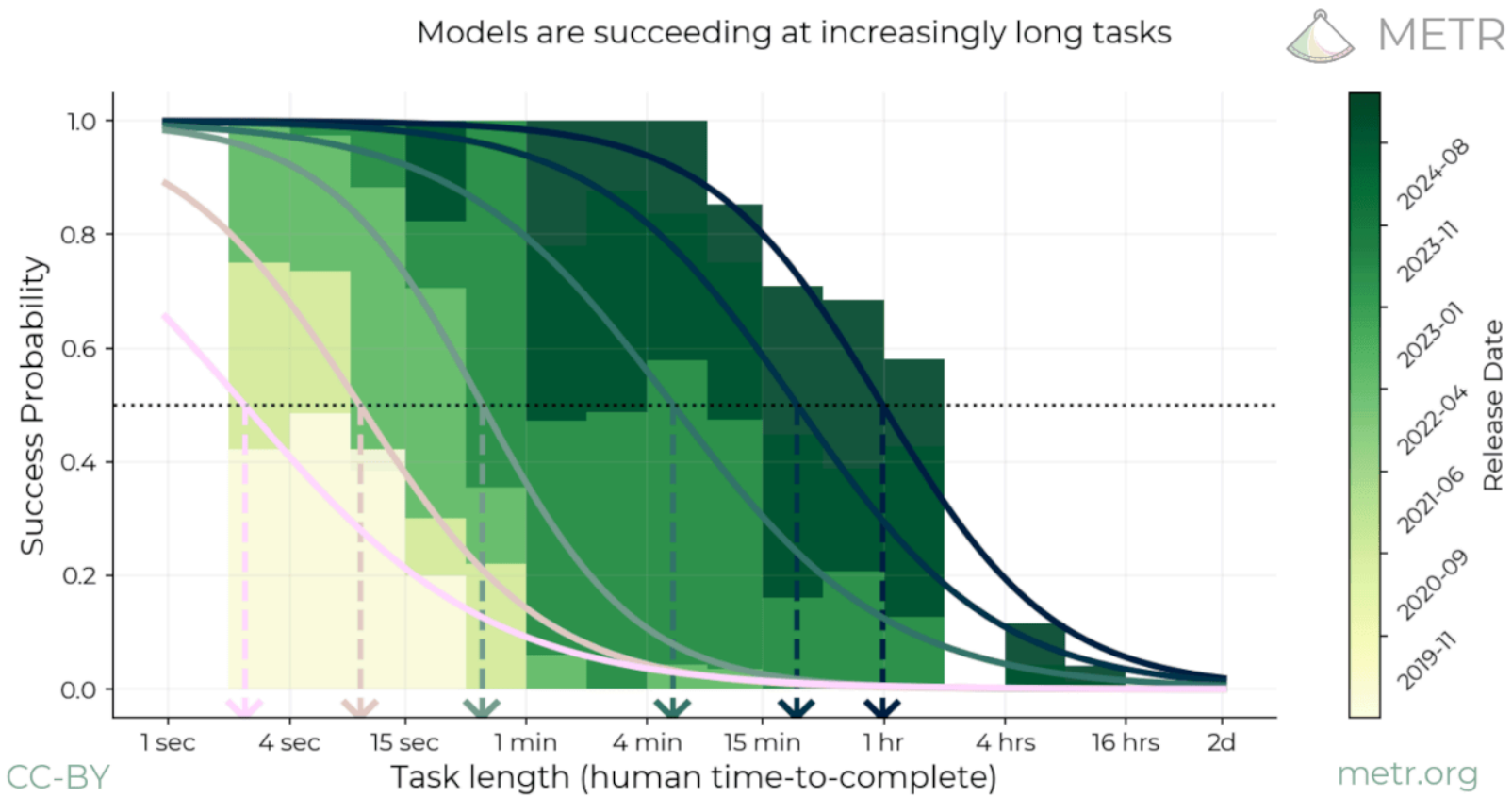
- The model has no theory — it sees the artefact and fills the rest in.
  Confidently. It won't tell you which is which.
- Bad prompt: "Write tests for this file"
  It freezes today's behaviour, bugs included. Again, no mention.
- Your job is still to build the theory.
  The AI helps you write the code that follows from it.



Speaker notes — preceding slide

- The implication for AI is direct and uncomfortable: Claude has no theory. It has a snapshot of the artefact, and whatever context you put in front of it.
- "Write tests for this file" will make it freeze the current behaviour into tests, including the bugs. Because it has no theory of what the code *should* do.
- Your job hasn't been automated. Your job is the theory.
- A design doc / decision log is more useful than CLAUDE.md, because CLAUDE.md is size-limited. CLAUDE.md gets the headline; the design doc gets the reasoning.
- Also: DO NOT let Claude generate CLAUDE.md and work with that unmodified - that has the very same issue as with generating tests.

Models get better - but how much?



Speaker notes — preceding slide

You might ask how long the stance in this presentation stays valid.

- Scaling laws: a function with the following parameters:
  - parameter count - limited by memory usage
  - dataset size - internet is already pretty much exchausted - going to synthetic data now.
  - computing power - increases a lot - but computing cost as well
  - loss (error rate) - more time spent in inference will yield better results.

- It was discussed there will be a wall, but it's not really clear yet if that is the case and how. Might also be a wall because of money.
- What's kinda clear: You won't get Skynet by just increasing compute power.
- But: The length of task a model can do with 50% success rate doubles every year according to METR.
  No idea if this will plateau at some point.
- Other improvements in architecture of those models (like Mixture of experts) also yield improvements.
- Still, frontier models only have 60% success rate on tasks that take humans an hour.
- The compilers build by Claude call gcc, fail at hello world and the browser by Anthropic was a complete fuck-up too.

Nobody can predict the future. Be skeptic of the ones that claim they can (which are surprisingly often the CEOs of big companies). I think most of the points on those slides here will still be valid in the coming years, maybe some shift towards the right side of the spectrum in the first slide, but not a full right turn.

Feedback?

Questions or Disagreements?

Homework:

PROGRAMMING AS THEORY BUILDING.

Speaker notes — preceding slide

Summary: Risk manage AI usage. Specifically:

- Don't just use AI assistence to be lazy. Use it to get better and to focus on more important things.
- Always make sure to stay in the loop. Stay the engineer, not the operator. A fool with a tool is still a fool.
- Be aware of the risks coming with that technology, but stay away from either extreme end of the spectrum.
- Confidence might be not well-grounded, use objective methods to check it. Keep cognitive biases in mind.
- Still act like you could loose access to AI at any time. You should still be capable to do your work without (maybe slower)
- Your second brain is now not only for you, but can be also turned into proper design documents for your agent.

That image is AI generated by the way. It depcits the end of the zauberlehrling
where he is overwhelmed from what we summoned. His old master has to help him.

Backup: Further reading –

Programming as Theory Building

Peter Naur – “Programming as Theory Building” (1985, PDF)

The original paper.

Christian Ekrem – “Programming as Theory Building”

Modern take: LLM-generated code belongs to nobody’s theory. Good entry point.

Christian Ekrem – “Architecture by Autocomplete”

Concrete example: AI defaults to primitive types because training data is full of them.

Christian Ekrem – “LLMs Corrupt Your Documents”

“The theory dies twice.” Design docs as source of truth silently degrade under AI edits.

Backup: More on skill atrophy & cognition

28→22%

endoscopists deskilled

adenoma detection rate dropped from 28% to 22% when working without AI, after months of AI exposure · skill atrophy beyond software

Lancet Gastro & Hepatology · 2025

−17%

the vendor measures the cost

junior devs scored 17% lower on a concept quiz after building with AI · code-reading and debugging impaired

Shen & Tamkin · Anthropic · 2026

83%

your brain on LLMs

LLM users couldn't quote their own essays · EEG showed weakest neural connectivity of the three groups

Kosmyna et al. · MIT Media Lab · arXiv:2506.08872 · 2025

Backup: Positive AI studies

+55%

Copilot task speed

sandbox RCT · optimistic ceiling

Peng et al. · GitHub · 2023

+14%

call-center output

novices gain most · METR in reverse

Brynjolfsson, Li & Raymond · NBER w31161 · 2023

40%

faster writing

time saved, quality up · cheap verification

Noy & Zhang · Science · 2023

Backup: But are devs being replaced?

-50%+

tech postings collapsed

US software postings down from 2022 peak · entry-level fell faster than senior — pipeline thinning, not headcount

Indeed Hiring Lab · 2024

700 → re-hired

Klarna's AI walk-back

announced AI replacing 700 customer agents in 2024 · quietly re-hired humans in 2025 · CEO admitted quality dropped

Bloomberg · FT · 2025

+0.7% / decade

the macro bear case

projection of AI's aggregate productivity impact · two orders of magnitude below CEO claims

Acemoglu · NBER w32487 · 2024

Senior roles augmented · juniors don't get hired in the first place.

Speaker notes — preceding slide