1. The CS Curriculum in the Age of Coding Agents

More questions than answers

Bill Pugh, University of Maryland

Professor Emeritus of Computer Science

2. Questions and some answers

- How is software development changing, and which changes will last?
- What skills should CS students have when they graduate?
- What do we know about how AI affects learning?
- How should the CS curriculum change, from the first programming course to the capstone?

And the two most urgent questions:

- What advice can we give current CS students?
- How should we change our intro CS courses?

3. How much change should we prepare for?

¯_(ツ)_/¯

Very few knowledgeable people try to make detailed predictions further than 2 years into the future.

I'm going to pick a rate of change that I think is just a little bit conservative:

- Should be good preparation for all but the rapid-change scenarios
- If things change very rapidly, we will have moved in the right direction and can adjust

4. How is software development changing, and which changes will last?

5. Source code will be *typed* by AI

- "Typed," not "written." I mean the text of source files. Writing software is more than that.
- **Anthropic**, June 2026: more than 80% of code merged into their codebase was authored by Claude
- **Google**, April 2026: 75% of new code AI-generated and approved by engineers (25% in Oct 2024, 50% fall 2025)
- **Thorsten Ball**, co-creator of the Amp coding agent and author of *Writing a Compiler in Go*, July 2026: "Nobody in our team really writes code by hand anymore. I certainly don't." His team poll: 99, 95, 90 percent

Two companies and one engineer who used to write compilers by hand. One direction.

6. This becomes near-universal even if the models stop improving

Why not yet:

1. **Adoption lag.** Even at Google: "that's how we did it six months ago"
2. **Harness headroom.** The agent scaffolding can improve independently of the model, some of it company-specific
3. **QA infrastructure.** Rerun every affected test, static analysis, style guides, checks that the code is not getting harder to maintain, before code is ready for review or use

None trivial. Most companies get there in about two years.

7. What happens to code review

A 12-year developer emails the creator of Claude Code, Sept 10, 2026: "What to do about slop?"

> "It costs more time to determine if their approach is wrong than it did for them to generate it."

Cherny's reply: production code written by Claude "should have a higher bar than if it was written by a human." The bar: lint, tests, Claude-driven end-to-end tests, daily fuzzers, automated code and security review, automated refactoring.

> "Your job is to hold the bar on code quality."

He never says "read the code."

Simon Willison, May 2026: "I'm not reviewing every line of code that they write anymore, even for my production level stuff."

8. The reviewer is ending. The review is relocating.

Marc Brooker, Distinguished Engineer, AWS, Sept 2026: "I believe that, long-term, humans have no role in routinely reviewing code."

Martin Monperrus (KTH), arguing for the end of review, June 2026: reviews of agent code "become rubber-stamps: the human approves because the code looks correct, because the tests pass, and because the cognitive cost of genuine scrutiny is prohibitive. The assurance that review is supposed to provide is illusory."

Yet even he keeps human approval for "high-risk changes, novel architecture choices, and regulated code paths," and concedes that tacit knowledge will need "pair programming and structured mentorship" to survive.

The reviewer as diff inspector is ending. The review, judgment about whether the software is right for its purpose, is moving to where the agent cannot follow.

New engineers used to acquire that judgment by being reviewed for years. Where does it come from now?

9. Agents get very good at anything with a clear pass/fail test

- Anything with a clear pass/fail test can be used for **reinforcement learning with verifiable rewards.** The models get exceptionally good at exactly that
- And an agent can iterate in a loop until every test passes

So: for a well-specified problem with an adequate validation harness, functional correctness is mostly no longer the worry.

The current issues are all the **other** things we expect of software.

10. What code review was always about

- Three studies over fifteen years: about **75% of review findings are maintainability**, 25% functional (Mäntylä & Lassenius 2009; Bacchelli & Bird 2013; Beller et al. 2014). Microsoft's own title: "Code reviews do not find bugs" (Czerwonka, Greiler, Tilford, 2015)
- Much of that 75% was not style. "When an experienced engineer reads a diff and says 'I don't understand this,' their confusion is the finding" (Allspaw, 2026). Comprehensibility is a property of a person meeting the code, not of the code alone
- Google's stated reason for code review: "code must act as a teacher for future developers." Readability, "maintaining norms": *that isn't how we do it here*
- Google tried **117 code and log metrics** to predict where its engineers said working code was slowing them down. **None worked.** Their explanation: that is a judgment comparing the code as it is to the code as it should be, and no metric can see the "should" (Jaspan & Green, 2023)

11. What injury might we incur by just using AI code that passes every test?

Suppose the tests are adequate and the code passes them. What might we lose?

- **The code may still be slop:** duplication, not the way this codebase does it, structure that resists the next change, a design nobody would have chosen
- When humans built the code, building it had benefits beyond the code:
  - the people who wrote it **understood** it, and could change it safely
  - the **reasons** for its shape were in someone's head, and often in the names and structure
  - review taught newcomers **how we do things here**, and the conversation produced "a shared understanding that neither party had prior to the discussion"
  - accept generated code and you get the artifact without the side effects. The tests cannot see what is missing

12. Three kinds of debt (Storey, 2026)

- **Technical debt** makes systems harder to change
- **Cognitive debt** makes systems harder to understand
- **Intent debt** makes it hard to know what the system is for

> "Generative AI does not remove the challenges of software engineering; it redistributes them."

Her own students: a senior team using AI to ship fast hit a wall in week eight, because "no one on the team could explain why certain design decisions had been made."

13. What skills should CS students have when they graduate?

14. From programmers to engineers

Ciera Jaspan, Engineering Productivity Research, Google, May 2026:

> "I kind of like that this is maybe going to be the forcing function that's going to take us from being programmers to being engineers. The code is still there. We still have to make sure it's high quality. But part of quality is not just the individual lines of code. It's about how the whole system is put together. It's about, does this system even solve the problem my users have in the first place?"

And earlier in the same session:

> "we're now going to need to make sure everybody is a senior engineer, including our most junior people graduating from college today."

15. What the people who hire say

Kam et al., Google + Education Development Center, FSE 2025 (job analysis with expert developers, May 2024, before agents):

- Four skill domains: using AI; core software engineering; adjacent engineering (security, regulation); **adjacent non-engineering** (users, customers, business, market)
- "a regular AI-enhanced developer of tomorrow will need to have the skills & knowledge of today's senior developer"

16. Skills that used to be learned late are becoming entry requirements

Specification. Testing and quality assurance. Version control. Design. Reading and explaining code you did not write. Working with people who are not software developers.

Students used to pick these up in senior courses, internships, side projects, or the first year on the job.

We could have been teaching many of these fifteen years ago. Some CS programs were.

17. Depth and *detectors*

Shriram Krishnamurthi, "Advice to the New College Student in the AI Era," summer 2026:

> "the purpose of going to college is to emerge as close to middle-aged as possible … cognitively. The way you do that is to acquire depth."

> "the scars of battle help you develop a finely-tuned sense of what can go wrong. This translates into high-quality detectors that identify various kinds of weaknesses: bullshit, threats to validity, ground not covered, and so on."

Detectors = a person's internal alarms. Not a tool.

> "Long, manual slogs helped us build up those detectors, and now we have tools that amplify the need for them. Without those detectors, modern AI tools set you up for epic failure — and the worst part is, without the detectors, you won't even know you've failed."

18. Software is more than source code and unit tests

Software includes decisions and thinking on:

1. **What** the software must do, and what properties it must have
2. **How** it accomplishes that: strategies, algorithms, architecture, interfaces
3. **Gaining confidence** that what was built serves its intended purpose

These concerns are spread across many documents and artifacts: requirements, design documents, diagrams, API and schema definitions, tests, conventions files, review discussions, and the code itself. Often jumbled.

19. Confidence is bigger than tests: exposure notification

At Apple, on COVID exposure notification:

- Nothing in clear logs on Apple servers. Not even the number of users in a region
- Privacy-preserving analytics: you could only learn what you had decided **in advance** to ask, and devices added noise
- We needed two significant revisions. We had missed the most important question: **are people who get a notification more likely to contract COVID?**
- After vaccines, base rates fell and the noise swamped the signal. We had to argue for new analytics
- The analytics also found bugs in Apple's and Google's code that we would never have known about

20. Design documentation is where humans matter most right now

Sequence diagrams. State machines. Externally exposed APIs you cannot refactor after release. Complex types and schemas.

This used to live in developers' heads, or in a document somewhere that did not track revisions with the code and quietly went stale. Now: **humans author and review the design; models keep it in sync with the code and tests.**

> "the responsibilities of software developers are rapidly becoming more about designing and less about programming." Shaw, Hilton, Fairbanks, *IEEE Software*, Mar-Apr 2026

Caution (Storey): "Resist the automation of understanding."

21. What do we know about how AI affects learning?

22. What counts as learning?

- Learning is not the same as getting the task done or the score on the test
- Acquiring a skill requires mental effort and, usually, struggle and failure along the way
- The research question is not "do students perform better with AI" but "do they still know it when the AI is gone"
- Almost everything measured so far used tools that could not run code. Agents can do more of the student's work, so what follows is a lower bound

23. Performance is not learning

Bastani et al., PNAS 2025 (about 1,000 high-school math students, randomized):

- With unguarded GPT-4 during practice: **+48%** on practice problems
- On the unassisted exam: **–17%** versus students who never had access
- With a tutor scaffold that withheld answers: harm removed, no gain

Anthropic, Jan 2026 (52 junior engineers learning a Python library): AI-assisted group **50%** on the follow-up quiz vs **67%** hand-coding. Two minutes saved.

"Cheating," redefined: completing the task without the learning the task was for. Nobody broke a rule.

24. Struggle has to land on the right thing

- Bjork: a difficulty is desirable only if it exercises the process you are trying to build. A logic error in a loop exercises how programs run. A missing colon does not
- Kapur, "productive failure": students who first attempt problems they have not yet been taught how to solve, and then get instruction, learn more than students who get the instruction first and then practice, **provided the failure is safe and is followed by consolidation.** A meta-analysis of 53 studies puts the effect at about a third of a standard deviation, larger when done well
- Krishnamurthi's test for the student: **"Am I eliminating failure?"**

25. Students who skip the learning strategies do not find out until it is too late

- Prather et al. 2024: struggling students using GenAI "finished with an illusion of competence"; students with a plan used it to write code they already intended to write
- Denny et al. 2025, 1,028 intro students at the University of Auckland working on underspecified problems: about **95%** of A students probed the oracle before coding; **more than half** of D students never did
- Bastani's students who practiced with AI felt fine right up to the exam

If the course only *recommends* the strategies that lead to learning, the students who most need them will not adopt them. The course has to push everyone into them.

26. What advice can we give current CS students?

27. Advice to current CS students

If your CS department offers a course in modern software development, take it.

You are in one now. Good choice.

Specification, testing, review, design: these are the skills that are becoming entry requirements, and they change far more slowly than the tools.

28. Advice to current CS students

Don't race to keep up with the bleeding edge of AI coding tools.

It is exhausting, and whatever you get up to is out of date in six months.

You do have to experience it for yourself, once, properly. Jaspan learns one new tool a month: "It keeps me sane." That is a full-time Google engineer's pace. With a full course load, something new about three times a year, learned properly, is enough.

29. Advice to current CS students

Have something software-related going on outside your regular courses, with people who are not all CS students.

Don't depend on the CS courses offered in the next two years to be the only thing you need to get you there. Find a group helping some organization solve a need in which software is a key component: a club such as Hack4Impact, a startup idea, a field outside CS you come to understand deeply.

I've observed over decades as a student and a faculty member that the students best prepared to have impact all had something like this. Sometimes research with a faculty member. More often, something built with other students. True for decades. More important now.

30. What that builds: taste

Calibrated judgment on the dimensions nobody has written a test for:
what is missing, what would be simpler, what will be hard to change, what the user actually wanted.

- Scarce because it cannot be measured (Google's 117 metrics found none that worked) and because the places it was learned, code review and asking a senior, are eroding
- Built the way every design discipline builds it: many examples with feedback, and comparison. The studio critique
- Agents make comparison cheap: five designs in a minute

31. How should the CS curriculum change, from the first programming course to the capstone?

32. Changing a curriculum

- A CS department cannot reinvent its courses every year or two
- A course has to reuse **60 to 90% of its material** from one year to the next, or it cannot be staffed, approved, or taught well
- So the question is not "what should the course look like this fall" but "what can we build that stays mostly right for several years"
- Fundamentals change over decades. Effective use of AI tools changes in months. Design the course so the perishable part is small and swappable

33. Every layer changes

Layer	What changes
Intro	Concepts without the syntax tax; specification; verification
Data structures, systems	Read and audit generated code; invariants; performance, memory, behavior under stress
Software engineering	Design documentation, API design, types and schemas, review as norm-setting, the three debts
Capstone	An intent-debt laboratory

Experiments are wanted throughout. **The intro course is where they are urgently needed.**

34. Two obvious ways to patch an existing programming course

1. **Ban AI** on programming assignments
2. **Allow AI at the student's discretion**, but leave the content unchanged from before AI, and what students are responsible for on exams unchanged

A coherent case can be made for 1, and some courses may do it until we have better approaches. But it cannot be enforced.

2 could end badly. Some students will avoid the challenges needed to learn skills that are still being tested, and will not find out until the exam. Unfortunately, 2 is what many courses have drifted into, because 1 cannot be enforced.

Anything beyond patching means moving topics around and re-imagining the course.

35. How should we change our intro CS courses?

36. Why the intro courses are the hard case

- The typical assignments are so easy that even non-frontier models produce solutions to all of them
- The problems are too simple and unrealistic to engage students
- And yet these courses may be where students acquire the fundamental conceptual building blocks: iteration, lists, conditionals, functions, and how programs run
- Without that conceptual foundation, students might struggle to become good software developers, even in a world where they never type source code

37. Additional complication: many students will skip CS1

- We want our new CS1 to be approachable by students with no previous programming experience
- Many students will likely skip CS1, based on work in high school or community college, in courses that have not been updated for generative AI
- Any topic we consider required for using generative AI effectively has to be covered in CS2, the first UMD CMSC course some students will take

38. For consideration: an experimental CS1 course

Students learn the **core concepts of programming** while spending a **minimum of time, friction, and struggle on source code syntax.**

- Stick to basic constructs that look much the same in many languages: loops, conditionals, arrays, maps, sets, functions, recursion
- Let students see the same code in Java, Python, C, and Swift, and notice what each language makes you decide
- Program reading and tracing. Faded Parsons problems. Writing tests that distinguish 4 correct implementations from 8 incorrect ones. Probeable problems
- Struggle on the right thing

39. The Rainfall problem

> Write a function that takes a list of daily rainfall amounts, stopping if it sees the value 99999, and returns the average rainfall, or `None` if the average can't be determined. Ignore negative values.

40. Rainfall: what forty years of studies found

For those of us who have written code for decades it is hard to see how a student struggles with this. Some do.

- Soloway, Yale, 1983: **14%** of CS1 students solved it
- Forty years of studies: the failures are the empty-input and negative-value corner cases, not the loop
- Any model solves it now. In 2023 Copilot failed it the way students do: it forgot the empty list

The source code that solves this is not interesting. **Thinking about the corner cases is the skill students need to develop.**

> "Most bugs won't be 'coding' bugs. They'll be 'you asked for the wrong thing' bugs." Thorsten Ball, Sept 2026

41. Rainfall as a Parsons problem

The lines of a working program, scrambled. The student puts them in order and indents them. Nothing to type, no syntax errors.

> Write a function that takes a list of daily rainfall amounts, stopping if it sees the value 99999, and returns the average rainfall, or `None` if the average can't be determined. Ignore negative values.

Where two blocks sit side by side, only one belongs.

```
def average_rainfall(amounts):
return total / count
if amount > 0:
```
or
```
if amount >= 0:
```
```
count = count + 1
for amount in amounts:
return None
total = total + amount
if amount == 99999:
if count == 0:
continue
```
or
```
break
```
```
total = 0
count = 0
```

42. Rainfall as a Parsons problem: the answer

```
def average_rainfall(amounts):
    total = 0
    count = 0
    for amount in amounts:
        if amount == 99999:
            break
        if amount >= 0:
            total = total + amount
            count = count + 1
    if count == 0:
        return None
    return total / count
```

- `break`, not `continue`: nothing after 99999 is read
- The 99999 test comes first: otherwise 99999 is averaged in
- `>= 0`, not `> 0`: a day with no rain is still a day
- `total` and `count` change together, under the same `if`: ignored values stay out of the denominator
- `if count == 0`: nothing to average. `None` says "no data"; 0 would say "no rain"

The student typed nothing, and still had to decide every corner case.

43. Faded Parsons problems

Real code, visible. Lines to arrange. **Blanks for the names and types the student must choose.** Paired distractors that encode the classic bugs (`if count == 0` next to `if total == 0`).

- Weinman, Fox, Hearst (Berkeley, 2021): transfer to code writing at least as good as writing, in about a third less time, preferred three to one
- Never run as the primary format for a whole semester. That is part of the experiment
- Any syntax question, the AI explains on demand. Brown's rule: don't ask it anything you wouldn't ask a TA
- A language toggle: Python by default; later, the same problem in Java, C, Swift, or pseudocode, with one question: what did this language make you decide?

44. The verification unit

How do you know the answer is right when you did not compute it?

- Properties, not expected values
- Equivalent answers, not equal answers: a simple equality test on outputs fails when several different outputs are all correct
- A slow, obviously correct second implementation, and compare them
- Invariants checked after every operation
- Metamorphic relations: `avg(2·xs) = 2·avg(xs)`
- Coverage is a floor. Mutation testing: test the tests
- **An exercise:** 12 implementations, pairwise distinguishable; 4 correct, 8 wrong. Write tests that tell them apart

The one principle: the oracle must be **independent** of the implementation. Tests written after seeing the code catch 14% of faults instead of 25%.

45. Ambiguity, on purpose

Probeable Problems (Denny et al., 2025): the spec omits details. "Find the smallest even value." An oracle answers any input you try. Probes are free; failed submissions cost.

Top students probe first. More than half of the weakest never do.

Brown's version: a testing plan with rows that just say **"decide."**

46. When do we cover agentic AI?

- Honestly, I don't know yet. Where agentic AI belongs in CS1 needs a much more detailed course design before anyone can say
- Perhaps this is best covered more in CS2, which fewer students will skip
- Under the covers, the course will likely use agentic AI for generating and analyzing Parsons problems, and more

47. Questions?

48. Backup

49. Backup: Parsons problems, the evidence

- Ericson, Margulieux, Rick 2017: Parsons 473 s vs fix 679 s vs write 714 s per session; no difference in learning or one-week retention
- Denny, Luxton-Reilly, Simon 2008: Parsons marks correlate with writing marks, not tracing; lower-quartile scores: tracing 0%, writing 48%, Parsons 78%
- Weinman 2021: faded Parsons beat writing on pattern acquisition; on code-writing transfer, 85% vs 75% in one study, 28% vs 29% in the other
- 13-institution study 2023: Parsons practice erased the prior-familiarity advantage on the final code-writing task
- Limits: near transfer only; expertise reversal for experienced students; GPT-4V solved 96.7% of Parsons screenshots, so not an assessment format

50. Backup: the verification techniques, with references

- Coverage is a floor: Inozemtseva & Holmes, ICSE 2014
- Mutation testing: Jia & Harman, TSE 2011; Just et al., FSE 2014; Code Defenders (Fraser et al.)
- Property-based testing: Claessen & Hughes, QuickCheck, ICFP 2000; Hypothesis; Wrenn, Nelson, Krishnamurthi 2021 on relational problems
- Differential testing: McKeeman 1998; caveat Knight & Leveson 1986
- Metamorphic testing: Chen et al., ACM CSUR 2018
- The oracle problem: Barr et al., TSE 2015; Weyuker 1982
- Examplar: Wrenn & Krishnamurthi, ICER 2019; conceptual mutation testing: Prasad et al. 2024

51. Backup: Kam et al., method and the T

- DACUM job analysis: 8 expert developers (screened from 3,000), 13 Google advisors, 2.5-day workshop May 31 to June 2, 2024; 12 goals, 75 tasks
- Six-step task workflow: Identify, Engage, Evaluate, Calibrate, Tweak, Finalize. "The use of GenAI tools is perhaps often understood to entail prompting an LLM only. Unfortunately, this corresponds to only a subset of the Engage step"
- The profile has a seventh step the paper dropped: Socialize. Getting the artifact accepted by the team
- Limits: 8 people, 7 men, self-report, pre-agent

52. Backup: industry numbers and their caveats

- Cherny, June 2026: "I haven't written a line of code by hand in, I think, eight months now"; Anthropic spokesperson, Jan 2026: 70 to 90% company-wide
- METR RCT (early-2025 tools): experienced developers **19% slower** with AI; they predicted +24% and believed afterward +20%. METR has since said its design broke because developers refused to work without the tools
- LinearB (8.1M PRs, 2025): AI PRs wait 4.6x longer for first review; accepted 32.7% vs 84.4%
- DORA 2025: 90% use AI daily; 30% trust its output little or not at all; throughput up, stability down
- Stack Overflow 2025: 84% use AI, 29% trust it, 3% highly
- GitClear (623M changed lines, vendor): duplication +81%, refactored code 21% to 3.8%, edits to year-old code −74%

53. Backup: design rules versus tests (Yu et al., 2026)

- 1,787 design rules mined from the code-review threads of six real projects (Django, SymPy, Ansible, Teleport, Flipt, Open Library); 495 issues
- Best agent: passed 77% of tests, fully satisfied the design rules on 35%; 42% of its patches passed while violating a rule
- Passing the tests predicted design compliance about as well as a coin flip (Cramér's V at most 0.12)
- Handing agents the relevant rules cut violations by about six points; pass rates mostly got worse
- Example: Django #13410, the rule was catch BlockingIOError only; the agent caught OSError

54. Backup: Cornell, September 2025

Students (2,029): 56% use GenAI several times a week or daily; 68% never use it to complete entire assignments, 7% often; 58% say it will be essential to their future work.

Faculty (737): 61% worried students gain an unfair advantage; 57% say it has challenged their ability to assess learning; 28% encourage students to use it; 16% think it has improved teaching and learning.