USE IT IN YOUR WORK

How Software Development Changed

Writing code got cheap. Knowing what to build did not.

First edition, September 2026
Addaly

Series editor: Dr Mustafa Mun
Draft, open for academic review

USE IT IN YOUR WORK

How Software Development Changed

Writing code got cheap. Knowing what to build did not.

Series editor: Dr Mustafa Mun

Addaly

First edition, September 2026 · Draft, open for academic review

How Software Development Changed

First edition, September 2026, build 62a26075.

Written by AI models to a written standard, with Dr Mustafa Mun as series editor. Exam keys checked blind by a second model: 3,665 of 3,666 agreed. Academic review invited.

The manuscript was drafted with the assistance of a large language model and was edited and published under his name. That is stated plainly here rather than left to be discovered, because a reader is entitled to know how a book they are trusting was made.

How to cite: *Mun, M. (2026). How Software Development Changed. Addaly. addaly.com/learn/how-software-changed.*

This book is the printed form of the course of the same name at addaly.com/learn/how-software-changed. The website is the living version and is corrected first. Where the two differ, the website is right.

Free to read, free to download, free to print, and free to teach from. There is no paid edition and there never will be.

Every diagram in it is drawn from data by code, not generated as a picture, so the numbers on the page are the numbers in the argument. The answers to every check, test and examination question are at the back.

9 modules · 87 lessons · 30 figures · 9 case studies · 68,190 words · about 13 hours

Brief contents

1

MODULE 1

Where the cost actually went

Account for a productivity claim in numbers — name which stage of the delivery pipeline got faster, compute what that does to end-to-end throughput when the constraint sits elsewhere, and read a study or a vendor figure well enough to say precisely what it does and does not establish.

The productivity claim, taken apart. What got cheaper, by how much, measured how, and why a faster writing stage does not make a faster team when the queue is somewhere else.

Lesson 1 · 8 min

The cost curve moved, but only part of it

THE ONE THING TO KEEP

Generating code got cheap; deciding what to build and confirming it is right did not.

What actually got cheaper

A working first draft of a well-specified function is now close to free. Parsing a CSV with awkward quoting. A paginated list endpoint. Converting a callback-based class to promises. Writing fixtures for a test. Adding a migration. That work used to cost somewhere between twenty minutes and a day. It now costs roughly as long as it takes you to describe it and read the result.

That is a real change, and not a small one. But notice how narrow the claim is. It holds where you already know exactly what you want, the shape of the solution is common, and correctness is easy to check.

What did not get cheaper

None of this moved much:

- Deciding what to build, and for whom
- Knowing whether the thing you got back is right
- Integrating it with everything that already exists
- Operating it at 3am when it fails
- Getting five people to agree on one interface
- Understanding a system well enough to change it safely

These were always the expensive parts. They were just harder to see, because typing was expensive too, and typing was the visible activity.

The arithmetic that people skip

Suppose your engineers spend about a third of their working week actually writing code. Estimates vary a lot by team and by study, so treat that as a rough figure, not a fact about you. Now make that third three times faster.

You have removed about 22% of the week. Not 60%. Not 10x. And you have removed it from the stage of the pipeline that was never the queue.

Figure 1.1

This is the old lesson about optimising the wrong part of the system. If code review takes four days because reviewers are busy, and you now produce three times as many diffs, review takes longer, not shorter. You did not speed up delivery. You built a bigger pile in front of the same door.

The evidence is genuinely mixed, and you should know that

Two things are both true and they get quoted selectively.

On greenfield work, unfamiliar languages, boilerplate-heavy tasks, and one-off scripts, the gains are large and easy to feel. People who write a lot of glue code, or who work across many stacks, are not imagining it.

On mature codebases that you know well, the picture is murkier. A widely-discussed 2025 randomised trial had experienced open-source maintainers work on issues in their own repositories, with and without AI assistance. They finished roughly 19% slower with it. They believed they had been about 20% faster. The sample was small, the participants were unusually expert, and the tooling has changed since. Do not treat it as the final word. But the gap between felt speed and measured speed is the finding that keeps replicating in less formal settings, and it should make you cautious about your own impressions.

Both results can hold. Speed comes from not having to know things. In a codebase you already know, there is less not-knowing to buy your way out of, and more cost in checking output against context the model does not have.

Where the constraint went

The useful mental model is a factory with two stations: production and inspection. Production got much faster. Inspection did not. Every unit still has to pass through a human who understands the system, and that human's throughput is roughly what it was in 2022.

So the constraint moved from *making the thing* to *confirming the thing*. Almost every other lesson in this course is a consequence of that one sentence.

This is why teams that adopt these tools and change nothing else often see more activity and the same delivery. Commits are up, pull requests are up, everybody feels productive, and lead time is flat. The measurement that matters is not how much code you produced. It is how much verified, integrated, working change reached users, and how often it had to be rolled back.

Try this

For one week, tag your working hours into four buckets: figuring out what to build, writing, verifying and reviewing, and everything else (meetings, operations, waiting). Do it roughly. Half-hour granularity is fine.

Most people are surprised. If writing is 20% of your week, then the tool that makes writing free has a ceiling of 20% on you, and the honest question is what to do about the other 80%.

BEFORE YOU MOVE ON

Your team's engineers spend about 30% of their week writing code. A new tool makes that part roughly three times faster, and everyone adopts it. Six months on, commits per week have nearly doubled and delivery time for a typical feature is unchanged. What is the most likely explanation?

A. Writing was not the constraint. The queue is in review, decisions and integration, and those got no faster — arguably slower, with more diffs arriving.

B. The team has not learned to use the tool well. With better prompting and more practice, the delivery gains will show up.

C. Delivery time is a lagging measure. The real gain is visible in output, and doubled commits is the result you were looking for.

D. The generated code is lower quality, so rework is consuming the time that was saved.

The answer and why: p. 487

Lesson 2 · 9 min

The fifth time code got cheap

THE ONE THING TO KEEP

Every wave of cheaper code has attacked the difficulty of expressing intent and left the difficulty of deciding what should happen untouched, and the current wave differs mainly in that it fails confidently rather than loudly.

This has happened four times before

The claim that writing code has become cheap is true. It is also the fifth time it has been true, and the previous four are worth knowing, because each one came with the same prediction and the prediction was wrong in the same way.

Assemblers, early 1950s. Before them you wrote numeric opcodes. An assembler let you write `ADD` instead of an octal number and let the machine keep track of addresses. This was a genuine order-of-magnitude change in typing and errors. It was also resisted, on the grounds that real programmers knew the addresses.

FORTRAN, 1957. John Backus's team at IBM spent about three years on the first compiler, and the pitch was explicit: programs would be written by the people who had the problem, not by programming specialists. The compiler produced code within striking distance of hand-written assembly, which was the technical achievement nobody expected. The social prediction — that programming as an occupation would shrink — did not happen.

COBOL, 1959. Designed so that the statements read like English and a manager could check them. `SUBTRACT DISCOUNT FROM GROSS-PAY GIVING NET-PAY.` Managers did not, in the event, read the programs. COBOL instead created a profession that is still being paid, sixty-five years later, to maintain systems written in it.

Fourth-generation languages and CASE tools, 1980s. Report generators, screen painters, diagram-to-code translators, and a marketing category literally called *application development without programmers*. This wave mostly failed, and it failed on a specific rock: the tools removed the typing and left the deciding, and the deciding was the expensive part.

Frameworks, package managers and public Q&A, 2000s onward. Rails claimed a blog in fifteen minutes and roughly delivered it. npm made installing more attractive than writing. Stack Overflow made the answer to the awkward question a search away. This was an enormous change, and it did what the earlier waves did: it changed the job rather than removing it. A large part of modern engineering is integrating and debugging other people's code, and that is what this wave produced.

Figure 1.2

Five times code got cheap

- **Early 1950s**
  Assemblers replace numeric opcodes. Resisted on the grounds that real programmers knew the addresses.
- **1957**
  FORTRAN. The pitch was that people with the problem would write the programs. The occupation grew instead.
- **1959**
  COBOL is written to read like English so a manager can check it. Managers did not read the programs.
- **1980s**
  Fourth-generation languages and CASE tools, marketed as application development without programmers. They removed the typing and left the deciding.
- **2000s onward**
  Frameworks, package managers and public question sites. Much of the job became integrating and debugging other people's code.
- **Now**
  Generation. No new notation to learn, it reaches the long tail, and it fails confidently rather than loudly.

Every wave attacked the difficulty of expressing intent. None of them touched the difficulty of deciding what should happen, which is why the same prediction has now been falsified five times.

Brooks's distinction is still the one that works

In 1986 Fred Brooks wrote *No Silver Bullet*, and its central split explains all five waves. He separated **accidental** complexity — the difficulty of expressing your intent in the available medium — from **essential** complexity — the difficulty of the thing itself: working out what should happen, in every case, consistently, for people who disagree with each other.

Every wave above attacked accidental complexity. None touched the essential kind. Brooks's specific prediction was that no single development in a decade would give an order-of-magnitude improvement in productivity, reliability and simplicity together, and for the four decades since he has been broadly right.

Whether he is right about this wave is genuinely contested and you should treat it as open. The honest statement is that the current wave attacks accidental complexity harder and more broadly than any previous one, and that if it turns out to reduce essential complexity as well, it will be the first thing that ever has.

Three things that really are different

Be fair to the present. Three differences are structural, not marketing.

1. **There is no new notation to learn.** Every previous wave asked you to adopt a language. This one accepts the description you would give a colleague. That removes a barrier that kept the earlier waves inside the profession.
2. **It covers the long tail.** A framework helps when your problem matches its shape. Outside that shape it is worse than nothing. A model will attempt anything, which is why it reaches work no framework ever did.
3. **It fails differently.** A compiler that could not do something said so, precisely, with a line number. This wave produces a confident, well-formatted answer whether or not it is right.

The third difference is not a detail. It is the reason the rest of this course exists, because it breaks the review habits that four decades of the previous waves taught everybody.

What to take from the history

The prediction *programmers will not be needed* has now been made and falsified five times. That is not proof it is wrong a sixth time — an argument from history is weak evidence about a genuinely new mechanism — but it does mean the burden of proof sits with the person making it.

The more useful pattern is this: each wave removed a layer of work, demand expanded to fill the new capacity, and the job moved up a level of abstraction. The people who had a bad decade were not the ones displaced by the tool. They were the ones who defined their skill as the layer that got automated. Assembly programmers who learned FORTRAN were fine. Assembly programmers who insisted that compilers produced sloppy code were fine for about eight years.

That is the risk worth planning around, and it is a career question rather than an existential one.

BEFORE YOU MOVE ON

Why did 1980s CASE tools and 4GLs, which really did generate working applications from diagrams and forms, fail to remove the need for programmers?

A. The generated code was too slow to run in production on the hardware of the time, so teams rewrote it by hand.

B. They removed the cost of expressing a decision and left the cost of making it, so the expensive part of the work was untouched.

C. They were proprietary and expensive, so adoption never reached the scale needed for the tools to improve.

D. Programmers resisted them for professional reasons, in the same way early assemblers were resisted, and blocked adoption inside firms.

The answer and why: p. 488

CASE STUDY · MODULE 1

The quarter the pull requests doubled

An illustrative case, built from documented patterns.

A 34-person product team at a logistics company in Pune, six months after every engineer got an assistant licence

Meera runs delivery for a team that builds routing and proof-of-delivery software for a fleet operator. In January the company bought assistant licences for all 34 engineers. By April the dashboards looked excellent.

Commits were up 78 per cent on the same quarter last year. Pull requests opened had gone from an average of 21 a week to 41. In the all-hands, the CTO used the phrase "step change in output" and nobody in the room disagreed, because everybody felt it.

Then the head of the fleet operator's operations team called to ask why a feature he had signed off in February was still not live.

Meera pulled four numbers out of the git host and the deploy log, by hand, into a spreadsheet. It took an afternoon.

- Deployments per week: 9 in the previous year, 9 now.
- Median time from first commit to production: 1.3 weeks then, 2.9 weeks now.
- Open pull requests at any moment: 25 then, 46 now.
- Changes that came back within thirty days as a revert, a hotfix or an incident: 11 per cent then, 19 per cent now.

The team was producing twice as much and delivering the same amount, more slowly, with worse outcomes. Nothing in the pipeline had changed except the rate at which work arrived at the review stage. Reviewers were the same eleven people with the same days, and they had started skimming, which is what a queue does when it cannot grow the service rate: it lowers the service quality until arrival and service match again.

Meera worked out the arithmetic she would need. Work in progress equals arrival rate times time in system, so with 46 open and about 20 merging a week, an item waited about 2.3 weeks. Capping the team at 12 in flight would bring that to about 0.6 weeks at the same throughput.

That left her with a decision, and both sides of it cost something real.

Cap work in progress. Each engineer limited to two open pull requests. On hitting the cap you do not open a third; you go and review, or you go and get one of yours merged. The mechanism is automatic and it moves effort to the constraint without anybody having to be persuaded each time.

The cost: every activity number on the executive dashboard falls. Commits down, pull requests down, and Meera would have to stand in a quarterly review and say that her team had deliberately reduced its output in the same quarter the company paid for tooling meant to increase it. Two of her engineers were up for promotion and their cases had been written around shipped volume. The finance director had already asked whether the licences were paying for themselves, and a chart going down is not an answer that survives that question.

Leave it alone and push on review speed. Ask reviewers to turn work around within a day, add a second approver on critical paths, and run a review rota. This is the response everyone expects and it costs nothing politically.

The cost: review capacity is eleven people's attention and no exhortation creates more of it. Faster review at a fixed capacity means shallower review, which is exactly the failure this kind of change is most vulnerable to, because a generated diff passes a shallow read trivially. The 19 per cent change failure rate would keep climbing and the cause would stay invisible, because no single incident would be traceable to the decision not to cap.

She also considered hiring two more senior engineers to review, and priced it: a five-month lead time at best, and the two most likely candidates would need three months in the codebase before their reviews meant anything. That is next year's fix for this quarter's problem.

There was one more complication. The fleet operator's contract had six features named in it for the quarter. Capping work in progress would not reduce throughput in theory. In practice, nobody had ever run this team that way, and Meera could not promise the board that it would not.

YOUR ANSWERS

1. Commits and pull requests both rose sharply while deployments per week stayed flat. What does that pattern tell you about where the constraint sits, and why does adding more writing capacity make it worse?

2. Meera could have hired two senior reviewers instead of capping. Using the Theory of Constraints sequence, why is that the wrong first move rather than simply a slower one?

3. Why did Meera present a prediction with a date rather than an argument about queueing theory, and what would have made her case unfalsifiable?

STOP HERE

Write your answers before you turn the page. How it played out starts on p. 52.

CASE STUDY · MODULE 1

How it played out

Meera capped it, at 15 rather than 12, and took the four numbers to the executive review instead of the activity chart. She framed it as a prediction that could be checked: same throughput, half the latency, in one quarter. By the end of the next quarter deployments per week had gone from 9 to 13, median lead time from 2.9 weeks to 1.1, and change failure rate from 19 to 12 per cent. Pull requests opened per week fell from 41 to 26, and the finance director asked about that number first. The contract's six features shipped, five of them earlier than the quarter before. The thing Meera did not anticipate was that two engineers found the cap genuinely uncomfortable for the first month, because sitting with two open changes and nothing new to start felt like idleness, and she had to write down the rule for what you do when blocked: review, help finish something, or work on the constraint. Never start something else.

The questions, discussed

1. Commits and pull requests both rose sharply while deployments per week stayed flat. What does that pattern tell you about where the constraint sits, and why does adding more writing capacity make it worse?

It tells you the constraint is downstream of writing. Throughput is set by the slowest stage, and deployments per week is measured at the end of the pipeline, so a flat number with rising input means the extra work is accumulating in front of some stage rather than passing through it. Here that stage is review: eleven people with unchanged hours. Adding writing capacity raises the arrival rate at a queue whose service rate cannot move, so the queue grows without bound until the service quality drops to match — which is what a shallow review is. Delivery gets slower and less reliable while every activity number improves.

2. Meera could have hired two senior reviewers instead of capping. Using the Theory of Constraints sequence, why is that the wrong first move rather than simply a slower one?

The sequence is identify, exploit, subordinate, elevate, repeat, and hiring is elevation — the last step, not the first. Before adding capacity you make sure the constraint never idles and you subordinate the other stages to its pace. Capping work in progress is subordination: it moves effort to review automatically. Hiring skips two cheaper steps, takes five months plus three months of ramp-up, and adds reviewers whose reviews do not yet mean much in an unfamiliar codebase. It also leaves the arrival rate uncapped in the meantime, so the pile keeps growing while you wait.

3. Why did Meera present a prediction with a date rather than an argument about queueing theory, and what would have made her case unfalsifiable?

A prediction invites verification, which is what makes it persuasive to someone who is sceptical and who controls the budget: she named the four numbers, said which direction each would move and by when, and used data from the company's own systems rather than an outside benchmark. An argument from principle cannot be checked, so it is heard as preference. Her case would have been unfalsifiable if she had promised better quality, higher morale or cleaner code — none of which has a number attached — or if she had left the definitions of incident and deployment free to be adjusted later, because then any outcome could be read as success.

MODULE 1 TEST

Check what stuck

6 questions, one right answer each. This one is for you, not for a record. The answers, and the reason behind each, are in the key on p. 486. If two go wrong, re-read the module before the exam.

1 A team's writing stage gets faster, so pull requests now arrive at 40 a week. The same eleven reviewers can service about 20 a week. What happens to the review queue?

A. It settles at a longer but stable wait, because every queue eventually finds an equilibrium length

B. It stays roughly as it is, because reviewers speed up once the pile in front of them becomes visible

C. It grows until review gets shallower and the two rates match again

D. It shrinks, because the changes arriving are individually smaller and quicker to read

2 In the 2025 trial, experienced maintainers were about 19 per cent slower with assistance and estimated afterwards that they had been about 20 per cent faster. Which mechanism best explains the direction of that error?

A. Waiting is not remembered as effort, so the clock and the memory diverge

B. The participants had an incentive to report favourably on the tools they were given

C. The tasks were unrepresentative of the work these developers normally do

D. The tools really were faster and the measurement instrument was at fault

3 A randomised trial gives every participant the same task — implement an HTTP server in JavaScript — and the assisted group finishes about 55 per cent faster. What is the most precise claim that result supports?

A. That teams adopting the tool will deliver more working software
B. That developer productivity rises by roughly half
C. That the code submitted by the assisted group was as correct as the rest
D. That completion time falls on a well-specified greenfield task

4 Suppose cheap generation triples the amount of software an organisation runs over a decade. What does that imply about the amount of work available?

A. Less work, because the same output now requires fewer people to produce it
B. More work, and mostly of the kind that did not get cheaper
C. More work, spread evenly across the same roles in the same proportions
D. No change, because maintenance cost tracks the number of users rather than the amount of software

5 You are asked to generate a discount engine for a scheme nobody has written down. The code involved is short and simple. Why is this among the weakest cases for generation?

A. Because pricing code is longer and more intricate than it first appears
B. Because models are trained mostly on web code rather than on financial logic
C. Because the difficulty was deciding, and a confident answer forecloses it
D. Because the generated implementation will be measurably slower than a hand-written one

6 In a two-week self-experiment, why does the allocation have to be decided per task rather than per day or per week?

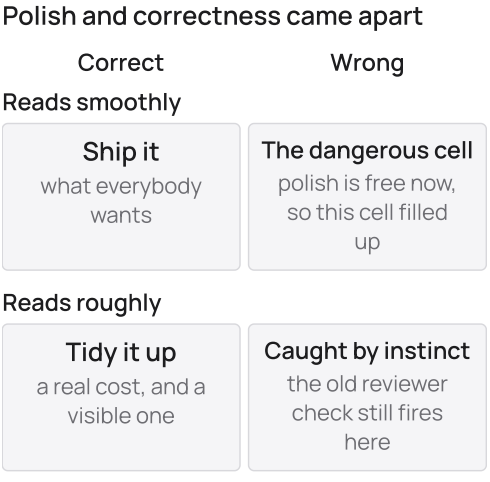
A. Because allocation by day measures your Mondays rather than the tool

B. Because a fortnight is too few data points for any statistical test to run

C. Because assisted and unassisted work should never be mixed within a single week

D. Because a coin flip is the only allocation method a sceptical manager will accept as fair

Figure 2.1



For thirty years, code that looked careful was decent evidence that somebody had been careful, because producing that appearance took care. It is now free, and the proxy broke.

So the code that most deserves your suspicion is often the code that reads most smoothly — the confident, idiomatic, well-structured block that matches your house style exactly, because it was pattern-matched from a million examples of code that was right in a slightly different situation.

What to do instead

Verify every external call against the actual documentation. Not from memory, and not by asking the same model. Open the docs. This costs about thirty seconds per unfamiliar call and catches the entire first category.

Compute one case by hand. Take one realistic input, work out the expected output on paper, run it. This is old-fashioned and it catches semantic errors that no amount of reading catches, because reading is exactly the channel that has been compromised.

Ask the four boundary questions. What does this do when the input is empty? When there are duplicates? When it arrives out of order? When it is a thousand times bigger than expected? Most silent semantic bugs live in one of those four.

Be most careful where the stakes are asymmetric. Money, authentication, permissions, deletion, anything that sends a message to a human, anything that writes to another company's system. In those places, insist on understanding the code line by line, or do not ship it.

One reframe

Stop thinking of the output as a draft from a colleague and start thinking of it as a draft from a very well-read stranger who has never seen your system, cannot run it, will not be paged when it breaks, and will agree with you enthusiastically if you suggest something wrong. That framing gets your review posture roughly right.

BEFORE YOU MOVE ON

You are reviewing a change and you see two blocks. Block A has an odd variable name, an inconsistent indent, and a comment that trails off. Block B is clean, idiomatic, well named, and matches the house style exactly. Both are new. Where should your careful attention go first?

A. Block A. Sloppiness in presentation usually indicates sloppiness in thinking, and that has not stopped being true.

B. Block B, because smooth, idiomatic code is now cheap to produce and carries no information about whether the logic suits your system.

C. Neither block specifically. Review the diff strictly in order so nothing is skipped and no bias creeps in.

D. Block A, since its rough edges suggest a human wrote it by hand and human bugs are the ones you know how to find.

The answer and why: p. 494

EXAMINATION

How Software Development Changed — final examination

This paper covers the whole course:

- where the cost of building software actually moved
- how to review a change whose author may not have read it closely
- the oracles that stay ahead of cheap generation
- writing a specification that doubles as a review criterion
- running a working loop where being wrong stays cheap
- changing a team's process to match the new constraint
- skill formation and hiring
- provenance and security and the unsettled law
- what did not change at all

63 questions, the whole pool. The site draws 25 for a sitting, pass mark 70%. Work any 25 under your own conditions. Answers from page 546.

1 Module 1 · Where the cost actually went

Assemblers, FORTRAN, COBOL, fourth-generation tools and frameworks all came with the prediction that fewer programmers would be needed. What does Brooks's distinction say they had in common?

A. Each attacked the difficulty of expressing intent and left the deciding untouched

B. Each was oversold by vendors, and the technology itself was less capable than claimed

C. Each required a new notation, which limited who could adopt it and slowed the effect down

D. Each was adopted by large organisations first, where productivity is hardest to measure

2 Module 1 · Where the cost actually went

A reviewer is about 80 per cent booked and takes on enough extra work to reach 90 per cent. What happens to the average wait in front of them?

A. It rises by about an eighth, in proportion to how much busier they now are

B. It roughly doubles, because waiting scales with utilisation over one minus utilisation

C. It falls slightly, because a busier reviewer batches work more efficiently

D. It is unchanged until utilisation reaches 100 per cent, at which point it becomes unbounded

3 Module 1 · Where the cost actually went

A company reports that 46 per cent of its code is now written by AI. Why does that number not belong in a decision about whether the tools are working?

A. It is not comparable between companies, which use different definitions of a line

B. It cannot be measured accurately, because no detector for generated code is reliable

C. It counts accepted characters, which says nothing about necessity or correctness

D. It excludes the configuration and infrastructure work where most of the effort actually goes

4 Module 1 · Where the cost actually went

A team merges 20 changes a week and has 25 open at any moment. What can you say about time from opening to merged, and what does the calculation require?

A. Nothing useful, because the figure depends on how long each review takes individually

B. About 1.25 weeks, but only if every change is roughly the same size as the others

C. About 20 days, since the arrival rate has to be converted into working days first

D. About 1.25 weeks, and it requires only that the queue be stable over the period

5 Module 1 · Where the cost actually went

Why did a compiler, a type checker, a linter and a property-testing library become more valuable at the moment producing plausible code became free?

A. They reject wrong code without anyone having to read it, and reading is the constraint
B. They are free, so they are the only option for teams that cannot afford a subscription
C. They have improved considerably over the same period, particularly in their error messages
D. They produce a record that can be shown to auditors when a defect reaches production

6 Module 1 · Where the cost actually went

An agentic session re-sends its context on every turn and keeps re-reading a 2,000-line file. What is the practical control on the cost?

A. Choose a cheaper model for the routine turns and a stronger one for the difficult ones
B. Give it three files rather than the repository
C. Lower the response length limit so that each turn produces fewer output tokens
D. Run the session at a quieter time of day when provider rates and latency are both lower

Module 1 · Where the cost actually went

The module starts on p. 11.

MODULE 1 TEST

Module 1 test, Q1, p. 54
A team's writing stage gets faster, so pull requests now arrive at 40 a week. The same eleven reviewers can service about 20 a week…

Answer C: It grows until review gets shallower and the two rates match again

Why: With arrivals at 40 and service at 20, twenty items accumulate every week and there is no stable length for the queue to settle at. The system reaches equilibrium the only way it can, by raising the service rate, and the only variable available is depth of reading. A shallow review is that adjustment, and it is exactly the kind of review that generated code passes without difficulty.

Module 1 test, Q2, p. 54
In the 2025 trial, experienced maintainers were about 19 per cent slower with assistance and estimated afterwards that they had been about 20 per cent…

Answer A: Waiting is not remembered as effort, so the clock and the memory diverge

Why: Forty minutes of typing registers as forty minutes of effort. Forty minutes of prompt, wait, skim, reject and re-prompt registers as much less, because the exertion is intermittent and each cycle is short, and people estimate duration from remembered exertion. Add debugging time being filed under a different task and the memorable wins outweighing the boring losses, and the estimate drifts in one direction — which is why an impression of speed is not a basis for a decision about how a team works.

Module 1 test, Q3, p. 55
A randomised trial gives every participant the same task — implement an HTTP server in JavaScript — and the assisted group finishes about 55 per…

Answer D: That completion time falls on a well-specified greenfield task

Why: A fixed-task trial can speak only about fixed tasks: this population, this language, this tool, and work with no existing system around it. It establishes nothing about a four-year-old codebase, nothing about the defect rate of what was submitted, and nothing about correctness beyond whatever check the study itself applied. Delivery and productivity are claims about the whole pipeline, and the pipeline was not measured.

Module 1 test, Q4, p. 55
Suppose cheap generation triples the amount of software an organisation runs over a decade. What does that imply about the amount of work available?

Answer B: More work, and mostly of the kind that did not get cheaper

Why: Every system that exists has to be patched, upgraded when its dependencies move, migrated, secured, monitored and understood by somebody at 3am, and that cost is roughly proportional to how much software exists. Maintenance is made almost entirely of understanding and verifying, which are the activities that did not get cheaper. So tripling the software triples a bill denominated in the expensive currency, and the extra work is not necessarily distributed to the same people.

Module 1 test, Q5, p. 55

You are asked to generate a discount engine for a scheme nobody has written down. The code involved is short and simple. Why is this…

Answer C: Because the difficulty was deciding, and a confident answer forecloses it

Why: The payoff tracks how cheap verification is, not how easy the code is. Where the hard part was deciding what should happen — a rounding rule, an order of application, a tie-break — a confident answer arrives dressed as a decision and terminates the thinking, and you cannot check it because there is nothing to check it against. The question to ask before generating is how you will know it is correct and how long that will take.

Module 1 test, Q6, p. 56

In a two-week self-experiment, why does the allocation have to be decided per task rather than per day or per week?

Answer A: Because allocation by day measures your Mondays rather than the tool

Why: Allocating by day or by week ties the condition to everything else that differs between those periods: which tasks were queued, how interrupted you were, how tired you were, what else was happening. Flipping a coin per task breaks that link, which is the whole of the method. It feels absurd, it costs nothing, and it is the difference between measuring the tool and measuring a stretch of your calendar.

THE LESSON CHECKS

Lesson 1, p. 15

Your team's engineers spend about 30% of their week writing code. A new tool makes that part roughly three times faster, and everyone adopts it…

Answer A: Writing was not the constraint. The queue is in review, decisions and integration, and those got no faster — arguably slower, with more diffs arriving.

Why: Generating code got cheap; deciding what to build and confirming it is right did not.

B The team has not learned to…: Assumes the gap is skill with the tool, when the arithmetic caps the possible gain at about a fifth of the week before skill enters the picture.

C Delivery time is a lagging measure…: Treats volume as the goal, which is exactly the substitution that hides a throughput problem behind a busy-looking team.

D The generated code is lower quality…: Often partly true, but even with flawless code the 30% ceiling means feature delivery could improve at most about 20%, not double.

Lesson 2, p. 20

Why did 1980s CASE tools and 4GLs, which really did generate working applications from diagrams and forms, fail to remove the need for programmers?

Answer B: They removed the cost of expressing a decision and left the cost of making it, so the expensive part of the work was untouched.

Why: Every wave of cheaper code has attacked the difficulty of expressing intent and left the difficulty of deciding what should happen untouched, and the current wave differs mainly in that it fails confidently rather than loudly.

A The generated code was too slow…: Performance was a complaint but not the structural failure; FORTRAN faced the same objection and survived it because the underlying economics still worked.

C They were proprietary and expensive, so…: Mistakes a commercial condition for a technical limit; free and widely adopted successors hit the same wall.

D Programmers resisted them for professional reasons…: Attributes to politics what the mechanism explains: the tools were adopted in many places and still did not remove the specification work.

Lesson 3, p. 24

A consultancy that builds bespoke CRUD applications for small firms sees its build time drop by two thirds. Its owner assumes revenue will hold because…

Answer D: Its clients can now build the simple application themselves, so its price falls without its volume rising.

Why: Making software cheap to produce expands what gets built rather than shrinking the work, but it expands the maintenance surface at the same rate, and maintenance is made of the activities that did not get cheaper.

A Induced demand only applies to physical…: Invents a boundary that does not exist; the mechanism is about price and demand, and applies to any good whose cost falls.

B Backlogs are usually imaginary, because firms…: Denies the unmet demand outright, when the observable pattern in most organisations is a real and long list of wanted, unaffordable tools.

C The backlog will absorb the capacity…: Treats a structural exposure as a timing problem, which leads to waiting out a decline that will not reverse.

USE IT IN YOUR WORK

How Software Development Changed

Writing code got cheap. Knowing what to build did not.

WHAT IT TEACHES	A plain account of what actually shifted in how software gets built, for people who write code or manage people who do. Where the bottleneck moved, why plausible-but-wrong code is a new kind of failure, how review and testing have to change, why specifications got more valuable, and which parts of the job got harder rather than easier.
WHAT IS INSIDE	Nine modules and 87 lessons, 30 figures drawn from data, nine case studies, nine module tests, a 63-question examination, and every answer with the reason behind it.
LICENCE	CC BY-NC-SA 4.0. Copy, print, share and adapt it for any non-commercial purpose, with credit, under the same licence. There is no paid edition.
THE COURSE	Free to read, with the lessons, the films and the examination, at **addaly.com/learn/how-software-changed**

Scan to open the course

Addaly

First edition, September 2026 · build 62a26075