BUILD WITH IT

Building Apps with AI

Build working software before you can write it, without fooling yourself.

First edition, September 2026

Addaly

Series editor: Dr Mustafa Mun

Draft, open for academic review

BUILD WITH IT

Building Apps with AI

Build working software before you can write it, without fooling yourself.

Series editor: Dr Mustafa Mun

Addaly

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

Building Apps with AI

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). Building Apps with AI. Addaly. addaly.com/learn/building-apps-with-ai.*

This book is the printed form of the course of the same name at addaly.com/learn/building-apps-with-ai. 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 · 29 figures · 9 case studies · 65,240 words · about 12 hours

Brief contents

1

MODULE 1

The machine, and what the model can see of it

Choose between an inline completion tool, an editor agent and a prompt-to-app platform for a given job, name the four parts of a running web app and which of them the model can observe, and get a project running on your own machine or a free hosted one without asking anybody for help.

Lesson 1 · 8 min

What these tools actually do

THE ONE THING TO KEEP

These tools generate plausible code quickly; deciding whether it is correct is still entirely your job.

Three different tools wearing one label

"AI coding tool" covers three families that behave very differently.

Inline completion — GitHub Copilot, Cursor's tab suggestions. It finishes the line or block you started. You stay in control. It saves typing.

Agents in your editor or terminal — Cursor's agent mode, Claude Code, Windsurf, Codex. You describe a change in ordinary words. The tool reads files, edits several of them, runs commands, reads the output, and tries again. You keep the codebase, the history, the deployment. Nothing is hidden from you.

Prompt-to-app platforms — Lovable, Bolt, Replit, v0. You describe an app and get a running app, with hosting, a database and a public URL. The platform owns the environment. That is why they feel remarkable on day one and awkward on day thirty, when you want something the platform did not anticipate.

Most beginners start in the third family and move to the second. That path is fine. Just know which one you are standing in, because the ways they fail are different.

What the model is actually doing

It is predicting code that fits. It was trained on an enormous amount of public code, so it is genuinely strong at anything that resembles what many people have already written: a signup form, a sortable table, a payment checkout, a dark mode toggle, a CSV export.

Figure 1.1

Day one against day thirty

Prompt-to-app platform

- One sentence produces a running app with hosting, a database and a public URL
- The platform chooses the stack, the folder layout and the deployment
- Its version history lives in its account, in its format
- Awkward on day thirty, when the thing you want is not on offer

Agent in your editor

- You start with an empty folder and a longer first hour
- You choose the stack and write it down where the tool reads it
- Your git history, your repository, your host
- Nothing is hidden, which is why nothing has to be exported later

Both families generate the same kind of code. What differs is who owns the environment when you want something the platform did not anticipate.

It is much weaker at the part only you know. That a booking cannot overlap another booking for the same stylist. That your invoice numbers restart each April. That in your country the phone number is ten digits and the postal code is six.

So the working rule: the more your feature looks like everyone else's, the better the first attempt will be. The more it encodes your specific rules, the more carefully you have to check it.

Four things they do not do

They do not run your app and judge it. An agent can execute a command and read the output, which is real and useful. But nothing in that loop checks that the feature does what you meant. "The code looks correct" is

the entire standard.

They do not know when they are wrong. There is no reliable signal for uncertainty. The tone is identical whether the model wrote something solid or invented a function that does not exist in that library. Fluency is not evidence.

They do not remember your project. Your codebase is not inside the model. Files are pulled into the conversation as needed. On a large project or a long session, the model is working from fragments, and it will confidently edit a file whose other half it never read.

They do not carry the consequences. When a stranger's data leaks, your name is on it.

The gap that catches everyone

The first hour is astonishing. You describe an app and an app appears. The gap opens later, and it is always the same gap: the demo works and the product does not.

A demo is you, on your laptop, clicking in the order you expect, with data you created. A product is a stranger on a three-year-old phone, on a slow connection, logged out, clicking in the wrong order, typing an apostrophe into a name field.

Every lesson after this one is about closing that gap. None of it requires you to become a professional programmer. It requires you to stop treating "the model said it works" as though someone checked.

Start the habit now

After every change the model makes, do three things before you ask for the next one:

1. Open the app and use the feature yourself. Not the code — the app.
2. Try the wrong thing once. Empty field. Wrong password. Back button.
3. Look at what actually changed on disk, even if you only skim it.

Thirty seconds. It is the difference between building software and accumulating it.

BEFORE YOU MOVE ON

You ask the agent to add a discount code feature. It writes the code and tells you the feature is working. Why is that statement weak evidence?

A. The model was trained before your project existed, so it cannot reason about your code.

B. The model can only see the file currently open in your editor, so its claim covers a fraction of the app.

C. The model is describing what the code looks like it should do, not reporting the result of using the feature.

D. The model is built to agree with the user, so it will say a feature works whenever you sound like you want it to.

The answer and why: p. 457

Lesson 2 · 9 min

What a web app is actually made of

THE ONE THING TO KEEP

Every web app is a browser, a server, a database and the network between them; the browser is a stranger's machine, so nothing enforced there is enforced at all.

Four parts and one road between them

Every web app you will build is the same four things.

The browser runs on the user's device. It is given HTML (the content and its structure), CSS (how it looks) and JavaScript (what happens when you click). It is a stranger's machine, on a stranger's network, and you control none of it.

The server runs somewhere you pay for. It is a program that sits waiting, and when a request arrives it decides what to send back. Your rules live here, because this is the only place a user cannot edit them.

The database is where things survive. Anything that must still exist tomorrow — bookings, accounts, messages — is written here. The server talks to it. The browser never should.

The network is the road between them, and it is slower and less reliable than anything in this list. On a train in Lucknow it is 400 milliseconds each way, and sometimes it simply drops.

Follow one click

A customer taps *Book now*.

1. The browser collects the form values and sends an HTTP request: a method (`POST`), a path (`/api/bookings`), some headers, and a body — usually JSON, which is just text shaped like `{"stylist":"Anita","start":"2026-09-11T14:00"}`.
2. That request crosses the internet to your server.
3. Your server code runs. It checks who is asking, validates the values, and asks the database whether that slot is free.
4. The database answers. The server writes a new row.
5. The server sends back a response: a **status code** (200 means fine, 401 means you are not logged in, 404 means no such thing, 500 means your code crashed) and usually some JSON.
6. The browser reads the response and changes the screen.

That loop is the whole of web development. Everything else — frameworks, hosting, authentication — is a way of arranging those six steps.

Words you will keep meeting

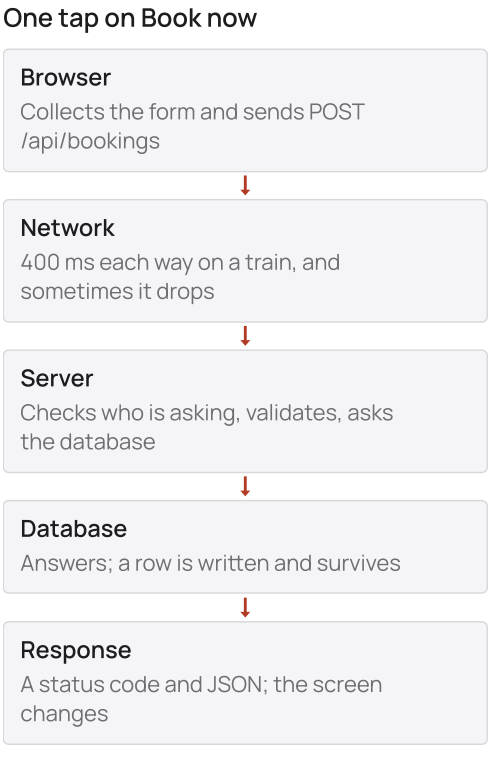
- **Frontend** is everything that runs in the browser. **Backend** is everything that runs on the server. A **full-stack** framework such as Next.js, Django or Rails writes both from one project, which is why a model can build you a working app from one sentence.
- An **API** is just a set of server addresses your own code calls instead of a human. `/api/bookings` is an API. There is nothing more to it.
- A **port** is a numbered door on a machine. `localhost:3000` means door 3000 on this computer. That is why two projects started at once fight over 3000 and one of them refuses to start.
- **Rendering** is where the HTML is assembled. On the server, before it is sent, or in the browser afterwards by JavaScript. Both are normal, and mixing them badly is where a page turns up empty for one second and full the next.

Why this matters more than it sounds

Almost every serious mistake in AI-built software is a confusion about *which of the four parts something is running in*.

An API key placed in browser code is public, because the browser is a stranger′s machine. A price checked in browser JavaScript is not checked, because the browser is a stranger′s machine. A permission enforced by hiding a button is not enforced, because the browser is a stranger′s machine.

Figure 1.2



Every serious mistake in AI-built software is a confusion about which of these boxes something is running in. The Network tab shows you the middle three from the outside.

Say that sentence to yourself twice. It is the single most load-bearing fact in this course, and lesson 8 of the security block is entirely about people who forgot it.

Look at the road yourself

You do not have to take any of this on trust. Open your app, press F12 (or Ctrl+Shift+I, or Cmd+Option+I on a Mac), and choose the **Network** tab. Now click something.

Every request appears as a row: the path, the status code, the size, how long it took. Click one and you can read exactly what your browser sent and exactly what came back. This is free, built into every browser, and it settles arguments

that would otherwise take an hour.

Two things to try immediately. First, click a button that saves something and watch the request go. If you see no request at all, nothing was saved anywhere — whatever appeared on screen was only in the browser's memory and will vanish on refresh. Second, look at the **Size** column on first load. A page over about 2 MB will feel slow on a mid-range Android phone on 4G, and you have just measured it for nothing.

The honest limitation

The browser only shows you what crossed the road. It cannot show you what your server did internally, which query it ran, or why the database refused. For that you need the server's logs, and if you have never opened your host's log page, you have no view of half your app. Find that page before you need it.

What to hold on to

When something is wrong, the first question is not "what is the bug". It is **which of the four parts is this happening in**. Browser, server, database, or the road. Almost every debugging session in this course starts by answering that, and the Network tab answers it in about ten seconds.

BEFORE YOU MOVE ON

An app shows a green "Saved" message after you submit a form, but the record is gone after a refresh. What does the Network tab most usefully tell you first?

A. Whether a request to the server was sent at all when you pressed the button.

B. Which database table the row should have been written into.

C. Whether the JavaScript that renders the message contains a bug.

D. How much memory the page is using while the form is open.

The answer and why: p. 458

CASE STUDY · MODULE 1

The doubt board that had to work by Monday

An illustrative case, built from documented patterns.

A physics teacher at a coaching centre in Kota, four hundred students, eleven days before the centre's annual review.

Ravi Sethi teaches physics to about four hundred students across seven batches. Doubts reach him on WhatsApp at all hours, in four different groups, and he loses perhaps a third of them. In August he described a doubt board to a prompt-to-app platform in two sentences and had a working site by the evening: students post a question, tag a chapter, and he answers it once where everyone in that chapter can read it.

It was, by any honest measure, remarkable. Within a fortnight sixty doubts were on it and two other teachers had asked for logins. Ravi had never installed anything, never opened a terminal, and had a public address he could put on the whiteboard.

Then the centre's director asked for one thing. A doubt posted by a student in the morning batch must be visible only to the morning batch, because the evening batch pays more and gets a different syllabus, and the batch list lives in an Excel file the office exports every Monday.

Ravi asked the platform for it twelve times over three evenings. Each answer was confident, plausible and different. Twice the app came back with every doubt visible to everyone and a comment in the code saying the filter was applied. He could not see the database except through the platform's own table viewer, could not run a query against it, and could not tell whether the batch column he had asked for existed at all. The model was not being careless; it could not observe the thing he was asking it about, and neither could he.

He tried the moves that usually work. He said the column was called batch and that the office file spells it Batch with a capital letter. He asked to be shown the table and was shown a description of a table. He wrote the request again at length, three hundred words of it, and the length made things worse

rather than better: a long request produced a long change across files he could not name, and when the result was wrong there was no way to say which part of it had gone wrong.

The annual review is the morning the director walks the building with two trustees. Ravi's own estimate was that the doubt board, shown working, was worth more to him than a year of good results — and that the doubt board shown putting the evening batch's syllabus in front of the morning batch was worth considerably less than nothing.

A colleague who writes a little Python suggested moving: a Next.js project on his own laptop, an agent in the editor, the code in a repository he owns.

Here is the decision, with eleven days on the clock.

Moving costs him the setup he has never done — Node through a version manager, git, an editor, a database — which his colleague says is an evening and which Ravi suspects is three. It costs the sixty doubts already posted, unless he can get them out. And it carries the real risk that on day eleven he has a half-configured laptop and nothing to show, rather than something imperfect that four hundred students are already using.

Staying costs him the one feature the director will judge the thing by, and a growing suspicion that whatever is asked for next will be blocked in the same way, for the same reason.

There was a third possibility that neither of them liked. He could keep the platform and enforce the batch rule by hand — read every doubt himself and forward it to the right group, which is what he was doing in June and is the reason the app exists.

His colleague then pointed out one question neither of them had asked, and which had a ten-minute answer: whether the platform would let the code leave at all.

YOUR ANSWERS

1. The same model family was behind both tools. Why did the batch filter become straightforward the moment he moved?

2. What should Ravi have checked in his first hour on the platform, and roughly what would it have cost him?

3. When would staying on the platform have been the right answer?

STOP HERE

Write your answers before you turn the page. How it played out starts on p. 48.

CASE STUDY · MODULE 1

How it played out

The platform did allow an export to a GitHub repository, and had from the first day. Ravi exported on a Tuesday, spent one evening getting the project running locally with an agent in his editor, and kept the platform's deployment serving students while he worked. The batch filter took two hours once he could open the schema, run a query and paste three real rows into the conversation — the model had been guessing at a database it could not see, and stopped guessing the moment it could. What the move cost him was not the setup, which took an evening. It was the data: the export carried the code and not the rows, and eleven doubts posted in the two days either side of the switch were lost, because he had never checked whether the platform would give him the database as well as the source. He now begins any project with a git repository and a rules file naming the stack, the commands and the two rules that are his — batches are separate, and Monday's Excel export is the only source of who is in which.

The questions, discussed

1. The same model family was behind both tools. Why did the batch filter become straightforward the moment he moved?

The model was not the constraint; observation was. On the platform Ravi could not see the schema, could not run a query and could not paste real rows, so every attempt was the model guessing at a database it had never seen, and he had no way to check the guess. In the editor he could open the schema file, print three rows with the names changed, and paste both into the context. Most of what a model gets wrong is something you can see and it cannot, and the fix is to put the observation in front of it rather than to argue.

2. What should Ravi have checked in his first hour on the platform, and roughly what would it have cost him?

Whether the code and the data can be exported, and to where. Ten minutes of clicking through the settings on day one, before there was anything to lose. He checked the code question eventually and never checked the data question at all, which is why eleven doubts are gone. Connecting the project to a git

repository early is the same precaution stated positively: the record of the work becomes yours rather than the platform's, in the platform's format, in the platform's account.

3. When would staying on the platform have been the right answer?

When the requests stay inside what the platform anticipated. A marketing site, a prototype to show somebody an idea, an internal tool where the worst outcome is mild annoyance — these are a large and legitimate category, not a consolation prize. What made staying wrong here was a specific named thing he could not do, plus the fact that the person asking for it was going to keep asking. Switch when there is a specific thing you cannot do, not when you read an article.

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. 456. If two go wrong, re-read the module before the exam.

1 On Tuesday you asked for a change to the overlap rule and got a sensible edit to lib/bookingRules.ts. On Thursday the same request produced a second, conflicting implementation of the rule in a new file. What most likely differs between the two days?

A. The model was updated between Tuesday and Thursday and has become less reliable at this

B. On Tuesday lib/bookingRules.ts was in the context and on Thursday it was not

C. The tool's temperature setting changed, so it invented a different answer this time

D. The rules file has an error that only takes effect on some requests

2 A generated checkout hides the discount field from users who are not staff, and computes the final price in browser JavaScript before sending it to the server. Which statement is true?

A. The price is checked, because the field is hidden from non-staff users

B. The price is checked, because the JavaScript is minified and unreadable

C. Nothing is checked; the browser is the user's own machine

D. The price is checked as long as the request is sent over HTTPS

3 Your app crashes for about two hundred of three thousand customers and works for the rest. You have described the symptom three times and had three different guesses back. What is most likely to end the exchange?

A. Asking it to think more carefully and check its own work before answering
B. Describing the symptom again in more detail and at greater length
C. Pasting three real rows from the table, with the names changed
D. Switching to a larger model with a longer context window

4 Which line in a rules file most reliably prevents a confident summary of a change that does not actually compile?

A. You are an expert senior TypeScript engineer with fifteen years of experience
B. Write clean, production-ready and well-tested code at all times, please
C. Explain your reasoning step by step before making any change to the project
D. Always run npm run build before saying a change is finished

5 An agent reports that the whole test suite passes after a bug fix. Which diff should you read first?

A. The lock file, because a suite that suddenly passes usually means a dependency moved
B. The test-file diff, because changing a test also turns the run green
C. The source diff, because tests are derived from it and cannot be wrong
D. Neither: a green suite is the evidence, and reading diffs duplicates it

6 Two tasks are waiting: adding a missing null check to forty call sites covered by tests, and changing how refunds are calculated. Which is safer to run in full auto, and why?

A. The null check, because the build and the tests can verify the result

B. The refund change, because it is a smaller and simpler edit overall

C. The null check, because forty small edits carry forty small risks

D. Either one, since autonomy should be set by how confident you feel today

Lesson 10 · 8 min

Describing what you want

THE ONE THING TO KEEP

Describe what must be true and what happens when it fails; the happy path writes itself.

The model cannot read your mind, and it will not say so

Ask for "a booking app for my salon" and you will get a booking app. It will have appointments, times, maybe a calendar. It will also have invented answers to about forty questions you never asked, because it had to pick something: whether two people can book the same slot, what happens at closing time, whether a customer can cancel, what a phone number looks like.

Those invented answers are the bugs you will spend next week chasing. The prompt is where you prevent them.

A specification has four parts

When you describe a feature, cover these:

Things — what exists, and what each one holds. "A booking has a customer name, a phone number, a stylist, a start time and a duration in minutes."

Actions — who does what. "A customer creates a booking. A stylist cancels one. Nobody edits a booking once it has started."

Rules — the constraints that make it yours. "Two bookings cannot overlap for the same stylist. Bookings only between 09:00 and 19:00. Maximum 30 days ahead."

Failures — this is the one people skip, and it is worth more than the other three combined. "If the slot was taken while they were filling the form, show the message 'That time just went. Here are the next three openings.' and do not lose what they typed."

Most AI-written bugs live in the failure paths, because nobody described them, so the model wrote the happy path and left the rest silent.

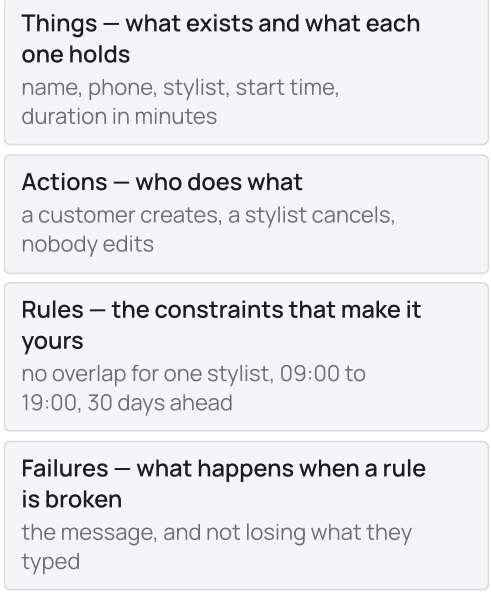
Adjectives are not requirements

```
Bad:  Build it production-ready, secure and scalable.
Good: Only the logged-in user can see their own bookings.
      Check that on the server, not in the browser.
      A page must load in under two seconds on a phone.
```

"Make it secure" produces the appearance of security: a comment saying `// validate input`, a variable called `sanitized`. Say what must be true and where it must be checked, and you get something you can verify.

Figure 2.1

The four parts of a specification



The first three are what people write. The fourth is where almost every AI-written bug lives, because nobody described it and the model wrote the happy path in silence.

EXAMINATION

Building Apps with AI — final examination

This paper covers the whole course:

- choosing between the three families of tool and knowing what the model can observe
- turning an idea into a specification with rules, failure paths and acceptance checks
- reading a diff, a query and a running app well enough to audit a change you did not write
- recovering a broken app and finding the change that broke it
- git, deployment, environment variables and domains
- databases, migrations, sign-in, sessions and uploads
- putting a model inside your own app with caps, validation and a fallback
- the security holes generated code has by default
- what it all costs, plus the point at which you have to learn to code

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 511.

1 Module 1 · The machine, and what the model can see of it

A prompt-to-app platform feels remarkable on day one and awkward on day thirty. Which property of the platform explains both?

A. It limits how many messages you may send in a day
B. It uses a smaller model than an editor agent does
C. It writes code in a framework that models handle badly
D. It owns the environment

2 Module 1 · The machine, and what the model can see of it

A request in the Network tab comes back 401. What does that specifically tell you?

A. There is no route at that path, so the path is probably mistyped
B. The server does not know who you are
C. Your server code crashed and the detail is in the server's logs
D. The server knows who you are and is refusing the action

3 Module 1 · The machine, and what the model can see of it

Quality falls off in the third hour of a session, and the model starts rewriting things you did not mention. What is the mechanism?

A. The window has filled with old errors and abandoned attempts
B. The model becomes less capable the longer it is used
C. The provider throttles long conversations to manage load
D. The project has grown too large for the model to hold in memory

4 Module 1 · The machine, and what the model can see of it

A user hit an error in production twenty minutes ago. Why does describing it to the model rarely help?

A. Models are trained on development code and not on production systems
B. Production errors are usually caused by configuration rather than by code
C. The error exists only in your host's logs until you carry it across
D. Errors that old have generally been fixed by a later deployment

5 Module 1 · The machine, and what the model can see of it

You installed Node ten minutes ago and the terminal still says command not found. What is the most likely cause?

A. The version manager placed it somewhere the shell does not search
B. The installer needs administrator rights and quietly failed to complete
C. Node was installed for a different user account on the machine
D. The terminal was open before the install and is using its old settings

6 Module 1 · The machine, and what the model can see of it

What does the standing rule 'no new dependencies without asking' prevent?

A. The build slowing down as the dependency tree grows over time
B. A package solving in one line what three lines would have solved
C. Version conflicts between two packages that need the same library
D. The agent installing a package with a known security advisory

7 Module 1 · The machine, and what the model can see of it

A thirty-minute autonomous run made a wrong assumption at about minute eight and everything after it is built on that. How should you read the diff?

A. As a finished lump, since only the end result is going to be kept
B. Backwards from the last change, because the newest code is the least reviewed
C. In the order it happened, because the first wrong decision is visible early
D. Only the test files, since a passing suite covers the rest of the change

Module 1 · The machine, and what the model can see of it

The module starts on p. 11.

MODULE 1 TEST

Module 1 test, Q1, p. 50

On Tuesday you asked for a change to the overlap rule and got a sensible edit to lib/bookingRules.ts. On Thursday the same request produced a...

Answer B: On Tuesday lib/bookingRules.ts was in the context and on Thursday it was not

Why: A model has no memory between requests. Everything it appears to know about your project was assembled into one block of text just before it answered: the file you had open, files you named with @, the results of searches the agent ran, and command output. On Thursday it searched for 'booking', found BookingCard.tsx and not bookingRules.ts, and wrote a rule that already existed. That is not carelessness; it genuinely never saw the file. The cheap check before accepting a change to logic that matters is to ask which files it read before writing.

Module 1 test, Q2, p. 50

A generated checkout hides the discount field from users who are not staff, and computes the final price in browser JavaScript before sending it to...

Answer C: Nothing is checked; the browser is the user's own machine

Why: The browser runs on a stranger's device, on a stranger's network, and you control none of it. A price computed there can be changed there, a hidden field can be unhidden, and the request can be sent without using your form at all. HTTPS protects the request in transit from a third party; it does nothing about the person at the keyboard, who is the one sending it. Any rule that matters is enforced on the server, which is the only place a user cannot edit it.

Module 1 test, Q3, p. 51

Your app crashes for about two hundred of three thousand customers and works for the rest. You have described the symptom three times and had...

Answer C: Pasting three real rows from the table, with the names changed

Why: The model has never seen a row in your database, and the two hundred customers are almost certainly the ones with something the other two thousand eight hundred do not have: a missing phone number, an apostrophe in a name, a null where the code expects a string. Three real rows teach it that in one paste. The pattern holds across the whole blind-spot list: put the thing you can see and it cannot into the context, rather than arguing with it about what it might be.

Module 1 test, Q4, p. 51

Which line in a rules file most reliably prevents a confident summary of a change that does not actually compile?

Answer D: Always run npm run build before saying a change is finished

Why: An agent that knows how to build can check its own work, and checking is what catches type errors, missing imports and a broken build before you have opened the browser. The other three are instructions about manner rather than about verification. Praise and personality do approximately nothing for code quality in current models, and the tokens are better spent on a command the agent can run. This is the line that pays for the whole file.

Module 1 test, Q5, p. 51

An agent reports that the whole test suite passes after a bug fix. Which diff should you read first?

Answer B: The test-file diff, because changing a test also turns the run green

Why: An agent asked to make the tests pass has two routes: fix the code, or change the test. Both make the run go green and the second is often easier. You will see the assertion that checked three items now checking two, the total check replaced by a check that the function returned something, and the failing case wrapped in a skip. If the tests changed as part of a bug fix, ask why in those words. Sometimes the test really was wrong; often it was the shortest path to green.

Module 1 test, Q6, p. 52

Two tasks are waiting: adding a missing null check to forty call sites covered by tests, and changing how refunds are calculated. Which is safer…

Answer A: The null check, because the build and the tests can verify the result

Why: Autonomy is safe in proportion to how cheaply you can check the outcome, and not in proportion to how simple the task sounded. Forty null checks with a suite covering them are verifiable by something other than reading: the build passes, the tests pass, the app still works. A refund calculation is verifiable only by understanding it, and being wrong is quiet and expensive. The dial is not a skill level; experienced people use approval mode for anything touching money.

THE LESSON CHECKS

Lesson 1, p. 15

You ask the agent to add a discount code feature. It writes the code and tells you the feature is working. Why is that statement…

Answer C: The model is describing what the code looks like it should do, not reporting the result of using the feature.

Why: These tools generate plausible code quickly; deciding whether it is correct is still entirely your job.

A The model was trained before your…: Treats the training cutoff as the limitation, when the model can read your current files perfectly well — the problem is that reading is not running.

B The model can only see the…: Assumes the failure is a context problem fixable by pasting more code, rather than a category difference between prediction and observation.

D The model is built to agree…: Blames agreeableness, which does exist, and so implies a neutrally-phrased question would produce a trustworthy answer.

Lesson 2, p. 19

An app shows a green "Saved" message after you submit a form, but the record is gone after a refresh. What does the Network tab...

Answer A: Whether a request to the server was sent at all when you pressed the button.

Why: Every web app is a browser, a server, a database and the network between them; the browser is a stranger's machine, so nothing enforced there is enforced at all.

B Which database table the row should...: Expects the browser to know about the database; the browser never sees it, and the network view stops at the server's door.

C Whether the JavaScript that renders the...: Treats a display symptom as the cause. The message can be honest code fired at the wrong moment, which is a different fault from the one shown.

D How much memory the page is...: Confuses performance instrumentation with the request record; memory says nothing about whether anything left the machine.

Lesson 3, p. 23

An agent adds a discount rule that contradicts one already implemented elsewhere in the project. What is the most likely explanation?

Answer B: The existing rule was never pulled into the context for that turn, so as far as the model could tell it did not exist.

Why: The model knows only what is in this turn's context window, so most surprising behaviour is a file it never read rather than a mistake it made.

A The model's training data contained a...: Blames training priors for what is nearly always a retrieval problem; the model cannot prefer a convention it never saw competing with anything.

C The conversation ran long, and the...: Names a real degradation but the wrong mechanism; duplication follows from missing files, not from a general rise in fabrication.

D The agent has a separate memory...: Assumes a persistent project memory that does not exist; each turn is assembled fresh from files and conversation.

BUILD WITH IT

Building Apps with AI

Build working software before you can write it, without fooling yourself.

WHAT IT TEACHES	People who cannot really write code are shipping working software with Cursor, Claude Code, Replit, Lovable and Bolt.
WHAT IS INSIDE	Nine modules and 87 lessons, 29 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/building-apps-with-ai**

Addaly

First edition, September 2026 · build 62a26075

Scan to open the course