

AI Quick Start

The generate → verify → integrate → commit loop for AI-assisted development

Emmanuel BRUNO
Université de Toulon, LIS UMR CNRS 7020
emmanuel.bruno@univ-tln.fr

2026-10-14

Abstract A hands-on introduction to AI-assisted full-stack development. You will set up an AI assistant — any editor-based or chat tool you prefer — learn the CRISP prompting framework, and run every change through one uniform loop — Concept → AI generates → You verify → You integrate (checkpoint + git commit) — demonstrated on a review-card example built beside the real fullstack-minimal-app. The session also covers agentic AI and the harnesses that run it (Claude Code, OpenCode, GitHub Copilot agents), and states our stance: S0–S2 fundamentals are built by hand, and agents are deferred to an optional accelerator from S3 onward.

Table of contents

1 Why AI-first	2	6.2 2. AI generates (CRISP prompt shown verbatim)	5
2 What this session covers	2	6.3 3. You verify — common-AI-mistakes checklist	5
3 What LLMs can / can't do	3	6.4 4. You integrate (checkpoint + <code>git commit</code>)	6
3.1 Key terms	3	7 Worked example: a review card	6
3.2 What they do well	3	7.1 Step 1 — Concept	7
3.3 Where they fail (the honest list)	3	7.2 Step 2 — AI generates (CRISP, verbatim)	7
4 Setting up an AI assistant	3	7.3 Representative generated result	8
4.1 If you use an editor assistant	4	7.4 Step 3 — You verify	8
4.2 Talking to any assistant	4	7.5 Step 4 — You integrate (checkpoint + <code>git commit</code>)	9
5 The CRISP prompt framework	4	8 Advanced prompting — RECALL / if needed	9
5.1 The scaffold (use verbatim)	5	8.1 Multi-turn prompting	9
5.2 Short examples	5		
6 The uniform loop: generate → verify → integrate → commit	5		
6.1 1. Concept	5		

8.2	Single-prompt techniques	9
8.3	Refactoring prompts	9
8.4	Temperature (a prompt parameter) .	10
8.5	AI is confidently wrong about your stack	10
8.6	Security review — the single XSS rule	10
8.7	When to reach for the deeper internals	10
9	Key takeaways	11
10	Quick reference	11
11	Agentic AI & harnesses	11
11.1	What “agentic” means	11
11.2	The harness (the agentic CLI)	11
11.3	Chat prompting vs. an agent	12
11.4	Capabilities — and risks	12
12	Our stance: we defer the agents	12
12.1	Why we defer them	13

1 Why AI-first

! Important

Why does this course start with AI tools?

Every professional full-stack team now works with AI assistants inside the editor. Starting here means **every later session reuses the same loop** — you are not learning a one-off trick, you are learning the working method.

- **They are productivity tools, not a substitute for understanding.** An AI assistant can draft a component in seconds, but you are still responsible for what ships. Reading, verifying, and understanding the code is *your* job, not the model’s.
- **They mirror how real teams work.** In industry the assistant proposes, the developer disposes: you review the diff, you run the

app, you write the commit message. This session builds that habit from minute one.

- **One loop, reused everywhere.** The step you learn here — **Concept → AI generates → You verify → You integrate (checkpoint + git commit)** — is the exact pattern every later lecture re-applies, whether the task is a form, an API route, or a refactor.

Learning objectives — by the end of this session you will be able to:

- Explain what LLMs can and cannot do, and why they need supervision.
- Set up an AI assistant in your workflow — an editor assistant (for example GitHub Copilot in VS Code) or a plain chat assistant (for example ChatGPT); any is fine.
- Write a well-formed prompt using the **CRISP** framework.
- Run any AI-generated change through the **generate → verify → integrate → commit** loop.
- Spot the most common AI mistakes before they reach a commit.

2 What this session covers

This session walks through five beats. The first three are the working method you will reuse in every later session; the last two frame where autonomous agents fit — and where they do not.

1. **AI as a conversational assistant** — what LLMs can and cannot do, and how to set one up.
2. **Prompting with CRISP** — the framework that makes your prompts verifiable.
3. **Good practices (you are the quality gate)** — the generate → verify → integrate → commit loop.
4. **Agentic AI & harnesses** — autonomous, multi-step agents and the tools that run them.

5. **Our stance: we defer the agents** — why S0–S2 is built by hand, and when agents re-enter.

3 What LLMs can / can't do

A **Large Language Model (LLM)** is a probabilistic model that predicts the next likely token given what it has seen. It is *often* helpful and *not* always correct — that single fact drives everything in this course.

3.1 Key terms

Term	Definition	Why it matters
Prompt	Your input to the model	Better prompts = better, more verifiable output
Hallucination	Confidently generated false information	Always verify against the real stack and docs
Context	What the model “remembers” in the conversation	Earlier messages and pasted files shape the answer
Temperature	Randomness of sampling (0.0–1.0)	Lower = more deterministic; higher = more varied

3.2 What they do well

- **Boilerplate & scaffolding** — forms, components, CRUD routes, config files.
- **Explanation** — “what does this do?”, “why is it doing it this way?”.
- **Refactoring** — renaming, extracting functions, adding error handling, JSDoc.
- **Test generation** — happy paths and edge cases for a given function.

3.3 Where they fail (the honest list)

- **They hallucinate APIs.** They will happily invent a hook, a function, or a library that does not exist.
- **They get your stack's specifics wrong.** The right database, the right version, the right package name — the model does not know *your* project, so it guesses.
- **They do not know your project's conventions.** Your naming, your styling conventions, your folder layout — none of that is in the model unless you tell it.
- **They cannot run or test your code.** They never see your browser, your console, or your database. Only you can confirm it works.

⚠ Treat the model like a fast junior developer

- ☑ Good at: boilerplate, syntax, common patterns.
- ✗ Bad at: your specifics, security, edge cases, business logic.
- 🔍 **Your job:** supervise, verify, understand — then commit.

4 Setting up an AI assistant

Keep it practical — the course does **not** mandate a particular assistant. Pick one tool you are comfortable with and use it for the whole session:

- **An editor assistant** — GitHub Copilot inside VS Code is the most common choice in the course devcontainer (students get Copilot free via the GitHub Student Developer Pack); any comparable editor extension works just as well.
- **A chat assistant** — ChatGPT (<https://chat.openai.com>), Claude (<https://claude.ai>), or

any equivalent tool in a browser tab next to your editor.

Whatever you pick, S0 stays **conversational**: you paste context, the assistant proposes text or a diff, and *you* decide — line by line — what to apply. No autonomous mode, no agent running your terminal (we explain why later in this session).

4.1 If you use an editor assistant

1. **Enable the assistant** in your editor and sign in with whatever account the tool needs (for GitHub Copilot, a GitHub account).
2. **Verify it works.** Open any `.jsx` file and type a comment such as `// a button that submits the search form and shows a loading state`, then watch for the inline suggestion. Accept or dismiss it with your tool's shortcut.
3. **Learn its two entry points:** inline completion (suggestions as you type) and a chat panel (a conversation about the selected code).

Shortcuts vary by tool, version, and personal configuration — check your assistant's documentation (in VS Code: `File` → `Preferences` → `Keyboard Shortcuts`).

4.2 Talking to any assistant

Most assistants separate two kinds of interaction. Use these conversational ones in S0:

- **ask / chat** — questions, explanations, small snippets, debugging help.
- **edit** — targeted edits to one file or selection (bug fix, style tweak, accessibility fix).

Workspace-aware prompting: where the tool allows it, reference a file (for example `#filename` in Copilot Chat) or paste the snippet so the assistant uses *lo-*

cal context instead of generic knowledge. Keep snippets focused and short.

The **agent** mode some of these tools also offer — multi-step, autonomous work across the repo — is deliberately out of scope for S0. It is described in *Agentic AI & harnesses* below, and its use is deferred by course policy.

Note

A pure browser workflow is fine. Chat-GPT and Claude work the same way for prompting — you just paste files instead of referencing them by name.

Warning

Privacy: never paste API keys, passwords, production credentials, or real user PII into a public or third-party assistant. Use mock or redacted data in prompts.

5 The CRISP prompt framework

Vague prompts produce vague, un-verifiable output. **CRISP** is the checklist that turns “make it better” into a prompt you can hold the model accountable to.

- **C — Context.** What are you building? Stack, project, target file.
- **R — Role.** Who should the model act as? (e.g. “senior React developer on this code-base”).
- **I — Input.** What does it have to work with? Pasted file(s), data shape, ports.
- **S — Steps.** Ordered, numbered actions it should take.
- **P — Parameters.** Constraints: styling system, versions, dependencies, what **not** to touch.

5.1 The scaffold (use verbatim)

```
Context:    [What are you building?
stack, project, target file]
Role:       [Who should the AI act
as? e.g. senior React developer on
this codebase]
Input:      [What does it have
to work with? pasted file(s), data
shape, ports]
Steps:      [Ordered, numbered actions
it should take]
Parameters: [Constraints: styling
system, versions, dependencies, what
NOT to touch]
```

5.2 Short examples

Example 1 — generate a component:

```
Context:    I'm adding a product-
review section to a React app. Reviews
arrive as mock
            JSON exported from a
            scraper-service prototype.
Role:       You are a senior front-
end developer.
Input:      A JSON array of review
objects { review_id, source, author,
rating, title,
            body, created_at,
verified_purchase }.
Steps:      1) Render each review as
a card. 2) Show author, a 1-5 star
rating, title,
            body, and created_at; mark
verified_purchase when true.
Parameters: Reuse the project's
existing CSS classes; accessible
labels; no inline
            styles; no new libraries.
```

Example 2 — explain code:

```
Context:    I'm learning React in
fullstack-minimal-app.
Role:       You are a patient
instructor for a student who knows
vanilla JavaScript.
Input:      [paste the function from
ProductDetail.jsx]
Steps:      1) Explain what it does.
2) Point out any bugs. 3) Suggest one
improvement.
Parameters: Short; use the app's
actual variable names; no jargon.
```

Why it works: each letter removes one class of failure. *Context* and *Role* stop generic answers; *Input* stops the model inventing your data; *Steps* make the output checkable line by line; *Parameters* stop it drifting off your stack.

6 The uniform loop: generate → verify → integrate → commit

This is the step of the whole course. Run **every** AI change through these four steps.

6.1 1. Concept

Before prompting, state in one sentence what the change should do and where it lives. If you cannot state it, you are not ready to prompt.

6.2 2. AI generates (CRISP prompt shown verbatim)

Write the CRISP prompt, paste it, and let the model generate. **Do not edit the suggestion in your head** — take what it returns, as-is, to the verify step.

6.3 3. You verify — common-AI- mistakes checklist

Never commit AI-generated code without running this checklist. Each item is a real, frequent failure mode:

- ☐ **Invented APIs.** Does every imported function, hook, or library actually exist? (The model loves to invent a `useSomething` that is not in React.)
- ☐ **Wrong version / stack.** Does it use *your* framework version, the right database, the right package name? (The model does not know your `package.json`.)
- ☐ **Unescaped / unsafe HTML.** React escapes output **by default**, so plain JSX text is safe. The real risk is `dangerouslySetInnerHTML` or injecting unsanitized HTML — is either present, and is it justified?
- ☐ **Ignored project conventions.** Does it match your naming, your styling conventions, your folder layout? (The model has no idea these exist unless you told it.)
- ☐ **Over-verbose code.** Is it a focused change, or did it rewrite the file, add dead code, and pull in a dependency you do not need?

Then confirm it *runs*: open the browser (or run the test), check the console for errors, and inspect the rendered output (DevTools → Console / Elements / Network).

6.4 4. You integrate (checkpoint + git commit)

Once it passes, integrate it and **stop at a green checkpoint**:

```
git add <the file(s) you changed>
git commit -m "<type>: <short, specific summary of the change>"
```

The commit is your checkpoint. If the next step breaks, `git` gives you a clean place to return. Small, focused changes → small, focused commits.

💡 Tip

The loop is the lesson. Generate → verify → integrate → commit. Every later session in this course reuses these exact four steps; only the task changes.

7 Worked example: a review card

We will apply the full loop to one tiny change connected to the course's data example: a **review card**. The reference repo is `fullstack-minimal-app` (the `develop` branch — referenced by URL, not a submodule).

What the app actually is, in one line: a React + Vite catalog storefront — frontend under `frontend/src/`, backend on `:4000`, frontend on `:5173`. **The shipped app is catalog-only: it has no review routes, no review database table, and no review UI.** Review data lives, for now, only in the standalone mock `scraper-service/` that sits next to the app in the same repo.

The one tiny change: generate a *standalone* review card component that renders one review in the scraper-service's data shape — as a new file, on a scratch branch, **not wired into any page or route**. Nothing in the catalog changes.

The review data shape (canonical, as produced by the scraper-service mocks):

Field	Type	Example
<code>review_id</code>	string	"AMZ_0001"
<code>source</code>	one of	"Amazon"
	Amazon	/
	BestBuy	/
	Walmart	

Field	Type	Example
author	string	"A. Student"
rating	integer 1-5	4
title	string	"Solid, but the manual is thin"
body	string	the review text
created_at	ISO date string	"2026-03-14"
verified_purchase	boolean	true

Note: `product_id` exists only as a **foreign-key column in the scraper-service's database design** — it is not part of the review card's props. Do not let the model invent UI fields your data does not actually have.

7.1 Step 1 — Concept

A presentational `ReviewCard` component that renders one review object — `author`, a 1-5 star rating, `title`, `body`, `created_at`, and a verified-purchase badge — as a new standalone file. No route, no fetch, no change to existing components.

7.2 Step 2 — AI generates (CRISP, verbatim)

Context: `fullstack-minimal-app` — a React (Vite) catalog storefront. The shipped

`app` is catalog-only; a separate mock `scraper-service` produces product reviews as JSON. I want a

NEW standalone review card component, not wired

into any page or route yet.

Role: You are a senior front-end developer working on this codebase.

Input: One review object with exactly these fields: { `review_id`, `source`, `author`,

`rating`, `title`, `body`, `created_at`, `verified_purchase` } — `review_id` is a

string like "AMZ_0001", `source` is one of Amazon / BestBuy / Walmart,

`rating` is an integer 1-5, `created_at` is an ISO date string, `verified_purchase` is a boolean. (I will paste a sample object.)

Steps: 1) Render `author`, a 1-5 star rating, `title`, `body`, and `created_at`.

2) Show an accessible "Verified purchase" badge when `verified_purchase` is true.

3) Use `review_id` as the React key when a list renders these cards.

4) Do NOT invent fields (no `product_id` prop, no image) and do NOT modify

any existing file.

Parameters: Plain JSX plus the project's existing CSS conventions; accessible,

semantic markup; no new dependencies; no inline styles; mock data only,

no fetching.

7.3 Representative generated result

No review component exists in the app today — this is a **new, representative file**, showing **what to expect and what to check**, not a diff to copy blindly.

```
// mock/review-sample.json — shaped
like the scraper-service output
{
  "review_id": "AMZ_0001",
  "source": "Amazon",
  "author": "A. Student",
  "rating": 4,
  "title": "Solid, but the manual is
thin",
  "body": "Works as advertised. Setup
took ten minutes; the documentation
is sparse.",
  "created_at": "2026-03-14",
  "verified_purchase": true
}

// frontend/src/components/
ReviewCard.jsx — NEW and standalone
(representative)
export default function
ReviewCard({ review }) {
  return (
    <article
      className="review-card"
      aria-label={`Review ${review.review_id}`}
    >
      <header>
        <h3>{review.title}</h3>
        <p>{review.author} &middot;
{review.created_at}</p>
        <p role="img" aria-label={`
${review.rating} out of 5
stars`} >
          {"★".repeat(review.rating)}
```

```
          {"★".repeat(5 -
review.rating)}}
        </p>
      </header>
      <p>{review.body}</p>
      {review.verified_purchase && (
        <span
          className="badge">Verified
          purchase</span>
        )}
    </article>
  );
}
```

7.4 Step 3 — You verify

Run the common-AI-mistakes checklist against the generated file:

- ☐ **Invented APIs?** No new imports, no new libraries — plain JSX. ☒
- ☐ **Wrong version / stack?** Plain React plus the project's CSS conventions — not a UI kit or styling library the app does not use. ☒
- ☐ **Unsafe HTML?** Review titles and bodies are *scraped, untrusted text*. They must render as escaped JSX text — any `dangerouslySetInnerHTML` here is an instant fail. ☒
- ☐ **Ignored conventions / invented data?** Does it use exactly the schema fields — `review_id`, `source`, `author`, `rating`, `title`, `body`, `created_at`, `verified_purchase` — with no invented `product_id` prop or image field, and a rating kept within 1–5? **△ check**
- ☐ **Over-verbose code?** It is one new component plus a mock file — did it also rewrite the catalog components or the app shell? It must not have. ☒

Then confirm it **runs**: render the card in a scratch preview (a local test page is enough — this is not a feature of the app), start the frontend on :5173, and check DevTools → Console

for errors. The card shows author, the star rating, title, body, and date, and the verified badge appears only when `verified_purchase` is true.

7.5 Step 4 — You integrate (checkpoint + git commit)

```
git      add      frontend/src/
components/ReviewCard.jsx      mock/
review-sample.json
git commit -m "feat(review-card): add
a standalone review card for mock
scraper data"
```

One new component, one mock file, one commit — still not wired into the catalog. You now have a clean checkpoint to build the next change on, which is exactly the loop the rest of the course repeats. (Presenting reviews *inside* the app would be a design decision for a later session — the S0 claim is only that the shipped app is catalog-only and the card here is standalone and representative.)

8 Advanced prompting — RECALL / if needed

RECALL / if needed — Advanced prompting (multi-turn, refactoring, verifying your stack) The techniques below deepen the same loop. Use them when a single CRISP prompt is not enough — but the **verify** step never changes.

8.1 Multi-turn prompting

Do not ask for a whole feature in one prompt. **Chain smaller prompts** and let each answer feed the next:

```
1) "Design the data shape for a
product list (fields only)."
```

```
2) "Given this shape [paste step 1],
write the React component skeleton."
```

```
3) "Now wire the component to a fetch
```

```
from :4000 [paste step 2]."
```

```
4) "Integrate the component into
App.jsx [paste step 3]."
```

The **debugging conversation** is multi-turn by nature — iterate, do not restart:

```
Turn 1: "I expect [X] but get [Y].
Here is the code and the error. What's
wrong?"
```

```
Turn 2: "I tried [suggestion] and
now [new issue]. Here is the updated
code."
```

```
Turn 3: "It works — explain why the
first attempt failed."
```

Keeping context means referencing what already works instead of re-explaining.

8.2 Single-prompt techniques

- **Chain-of-Thought.** Ask it to reason before it codes: Think step-by-step: 1) what are the rules? 2) edge cases? 3) approach? 4) how to test? Then implement with comments.
- **Few-shot.** Show 2–3 examples of the exact pattern you want (e.g. two existing components) and ask for a third in the same style.
- **Role prompting.** “You are a security expert auditing this code” yields a security-focused review; “You are a tech writer” yields documentation.
- **System vs user prompt.** Where a tool supports a system prompt, put your project conventions there (stack, async/await, parameterized SQL, accessible HTML) so every user prompt inherits them. In Copilot, `#filename` references and a short conventions note in the chat play the same role.

8.3 Refactoring prompts

Refactor in **small, testable steps**, never “refactor the whole file”:

```
1) "Replace magic numbers with named constants."
2) "Extract this block into a well-named function."
3) "Suggest more descriptive variable names."
4) "Add error handling with try/catch."
```

Each step is small enough to verify on its own — which keeps the loop intact.

8.4 Temperature (a prompt parameter)

Lower temperature (≈ 0.2 – 0.4) for deterministic code completion; higher (≈ 0.6 – 0.8) when brainstorming alternatives. When in doubt for code, keep it low.

8.5 AI is confidently wrong about your stack

This is the single most important verify habit. The model answers from *average* training data, not from *your* `package.json` or `schema.sql`:

```
You: "What's the best database for this app?"
AI: "PostgreSQL is definitely the best choice."
Reality: This app uses MySQL — check the actual stack before accepting anything.
```

The AI suggested PostgreSQL but this app uses MySQL — verify against the real stack. The same trap appears with package names, versions, and framework APIs. When the model states a fact about your project, cross-check it against the actual file, the official docs, or a second source before you trust it.

8.6 Security review — the single XSS rule

Run a security pass on any generated code, and hold the **one** consistent rule:

React escapes output by default. Plain JSX text is safe. The real XSS risk comes from `dangerouslySetInnerHTML` or injecting unsanitized HTML.

Security audit of this code:
[paste code]

Check for:

- SQL injection — are queries parameterized (no string interpolation of user input)?
- XSS — per the single rule above, is there any `dangerouslySetInnerHTML` or unsanitized HTML being injected?
- Authentication / authorization issues
- Sensitive data exposure (secrets in code or logs)
- Input validation gaps

8.7 When to reach for the deeper internals

If you want the mechanics — tokens and context windows, the transformer / self-attention model, the training pipeline, embeddings and semantic search, or RAG (retrieval augmented generation, how Copilot Chat searches *your* repo) — those are the “how it works” topics behind the prompting techniques above. They are useful context, but for day-to-day development the loop and the verify step are what you actually use.

9 Key takeaways

1. **AI is a productivity tool, not a substitute for understanding** — you still read, verify, and commit.
2. **One loop everywhere** — Concept → AI generates → You verify → You integrate (checkpoint + `git commit`).
3. **CRISP makes prompts verifiable** — Context, Role, Input, Steps, Parameters.
4. **The verify step is non-negotiable** — invented APIs, wrong stack, unsafe HTML, ignored conventions, over-verbose code.
5. **Small change, small commit** — a clean checkpoint before every next step.

10 Quick reference

CRISP prompt template:

```
Context:    [What are you building?
stack, project, target file]
Role:      [Who should the AI act as?]
Input:     [Pasted file(s), data
shape, ports]
Steps:     [Ordered, numbered
actions]
Parameters: [Constraints: styling
system, versions, dependencies, what
NOT to touch]
```

Verify checklist (common AI mistakes):

- ☐ No invented APIs
- ☐ Right version / stack (your `package.json`, your database)
- ☐ No unsafe HTML (React escapes by default; watch `dangerouslySetInnerHTML`)
- ☐ Matches project conventions
- ☐ Focused, not over-verbose

The loop: Concept → AI generates → You verify → You integrate (checkpoint + `git commit`)

11 Agentic AI & harnesses

Everything so far has been **chat prompting**: you type a prompt, the model returns text, you decide what to do with it. **Agentic AI** changes who is in the driver's seat — and that change is the single biggest shift in how the tool can help or hurt you.

11.1 What “agentic” means

An **agent** is an AI system that, given a goal, works *autonomously* over *multiple steps* and *uses tools* to get there — reading files, running commands, editing code, checking the result, and deciding the next move — without you issuing each individual instruction.

Three properties separate it from the chat assistant you have been using:

- **Autonomous.** You give it an outcome (“make the tests pass”), not a sequence of edits.
- **Multi-step.** It plans, acts, observes the result, and iterates until it thinks it is done.
- **Tool-using.** It reads the repo, runs the terminal, edits files, re-runs — a real loop.

11.2 The harness (the agentic CLI)

The model itself is just a text engine — it produces tokens, it does not touch your machine. What lets it *do* things is the **harness**: the scaffolding wrapped around the model that gives it a working directory, a terminal, file access, and a permission system. An **agentic CLI** is this harness run from the command line. You launch it inside your repo, describe the goal in plain language, and it drives the loop for you.

Because the harness controls what the model can touch, it also carries a **permission model**: what the agent may read, edit, and run is gated by what you allow. That gate is part of the tool, not a suggestion — and it is where you keep your hands on the wheel.

Three you will meet in industry (and, later, in this course):

- **Claude Code** — Anthropic’s agentic coding CLI; runs in your terminal, edits files, runs commands.
- **OpenCode** — an open-source, terminal-based agentic coding assistant with the same shape.
- **GitHub Copilot agents** — the agent modes of assistants such as Copilot Chat (multi-step, workspace-aware work: create files, scaffold, refactor across the repo).

11.3 Chat prompting vs. an agent

The difference is *who drives the loop* — it is the sharpest contrast in this session:

	Chat prompting (this session so far)	Agentic AI
Who acts	You	The agent
Granularity	You drive every step	It drives the loop
Per turn	One prompt → one answer	Goal → plan → many actions
Tools	It only suggests text	It runs commands, edits files
You stop it	You never run anything it suggests	You review after the fact (or interrupt)

In chat, you type, you paste the result, you run the app, you commit. With an agent, you describe the goal and it *runs the loop for you* — and you come back to a set of changes.

11.4 Capabilities — and risks

Agents are powerful, and the power and the risk are the same thing. Because the agent **can run commands, edit many files, and keep going without you**, it can also **make mistakes at scale**: a wrong edit propagated across ten files, a command run it should not have, a refactor that “works” but drops behavior you never noticed.

```
You:      "Sort the product list by
           rating, highest first."
Agent: plans → edits the list page →
         edits the API route → runs the app →
           "Done." (3 files changed, one
           test now failing, a query you didn't
           ask for)
```

That is why the rule from the rest of this session does not weaken — it gets *more* important with agents: **you remain the quality gate**. An agent can drive the car, but you still decide whether to get out of it, and you inspect the road it took before you accept where it stopped. The verify step, the git checkpoint, and your own understanding are the difference between an accelerator and a liability.

⚠ Warning

An agent is a fast, confident junior that can run your terminal. Give it a narrow goal, review every diff it produces, keep small commits, and never let it touch a secret or a destructive command without watching. Capabilities and risks rise together — the gate is you.

12 Our stance: we defer the agents

We introduced agents on purpose — so you can see exactly what they can do. Now the stance

for this course, stated plainly, because it will shape every session after this one:

- **S0–S2 — fundamentals, by hand.** The HTML, CSS, and JavaScript fundamentals are built *by you, by hand*, with **no autonomous agents**. You write the markup, the styles, and the logic yourself so that, line by line, you can explain what you shipped. In these sessions no agent writes the loop, the style, or the markup — that work is deliberately yours.
- **S3 onward — agents as an optional accelerator.** Once the fundamentals hold, agents become a *choice*, not a requirement. Use them to go faster — but always with the same rule from this session: **you remain the quality gate**. You review the diff, you run the app, you write the commit. An agent may drive the loop; you still decide what ships.

In practice: in S0–S2 you may still use a chat assistant to *explain* something or *check* an idea — that is conversational, not autonomous. The line we draw is against the agent *writing and running* the fundamentals for you.

12.1 Why we defer them

The reason is pedagogical, not technical. An agent that writes the code *for* you can hide the very thing you are here to learn — and it hides it exactly when the learning is happening.

- **Fundamentals first.** If an agent writes the loop, the style, and the markup, you never learn the loop, the style, or the markup. The shortcut and the lesson occupy the same moment; take one and you lose the other.
- **Agents can hide the learning.** “It worked” is not the same as “I can explain why it worked.” When the agent does the thinking, the learning is exactly what slips away — and you cannot verify what you do not understand. A result you did not produce is a result you cannot defend.

- **The gate has to be you, not a tool.** A quality gate only works if *you* can read the code it produces. Agents do not lower that bar; they raise the cost of not having it, because the faster the agent moves, the more expensive it is to have to catch up.

Tip

Deferring is not refusing. We are not anti-agent. We are building the fundamentals by hand first so that, from S3 onward, you become the kind of developer an agent *cannot* replace — the one who checks the diff, runs the app, and knows, without asking the model, whether the change is right.

The through-line. Every session after this one repeats the same test: *can you explain what you just built, without the model?* If the answer is yes, the agent — used or not — accelerated you. If the answer is no, the agent did not accelerate your learning; it only produced code you could not stand behind. That test is the course, and it is the same test the quality gate runs on every commit.