Module 1: The AI-Augmented Environment, Lesson 4: Context is King

Hands-On: The Curation Discipline

Overview

This demo proves three claims from the lecture in real time:

- A polluted chat thread drags assumptions from one domain into every answer that follows
- The Fresh Chat + @ tagging workflow produces dramatically sharper output on the same task
- A 10-second context audit before every prompt raises quality while shrinking token cost

You will debug the same bug — a user avatar that fails to render on the profile page — three times, using three different context strategies. By the end, you will feel the quality gap between "dump everything" and "curate surgically" in your own eyes, on your own screen.

Before You Start

- Open Cursor (preferred — it has native @ tagging) or any AI chat: Claude.ai, ChatGPT, or GitHub Copilot Chat
- Clone the sample repo (https://GitHub.com/AI-NativeEngineer-Course/avatar-demo) and run: npm install && npm run dev
- Open the profile page in your browser — you should see a broken avatar image, which is the bug you will debug
- Keep the RICE structure from Lesson 3 visible: Role → Instructions → Context → Expectations
- Know your shortcut: Ctrl+L on Windows, Cmd+L on Mac, opens a fresh chat

> **Remember**
> The context window is the AI's short-term memory — not free storage. Every token you attach is a token the model must process, and every irrelevant token dilutes the signal. In this exercise we prove, by direct comparison, that curating context is the single highest-leverage habit you will build in this module.

During Hands-On Keep Visible if Possible

The 4-question curation checklist — run it before every prompt:

1. `WHAT` — what specific behavior is broken?
2. `WHERE` — which ONE file owns that behavior?
3. `DEPENDS` — what types, APIs, or utilities does that file import?
4. `NOISE` — what am I NOT attaching? (CSS, migrations, tests, unrelated features)

Prompt 1 — The Polluted Chat (Do It Wrong First)

> **What to Learn**
> You need to feel context pollution before you will respect the Fresh Chat rule. We deliberately start in a chat thread with a stale backend mental model, then ask a pure frontend question. Watch the AI drag assumptions from the earlier domain into a completely unrelated answer.

Step 1 — Build the polluted context (paste this first, let the AI respond):

```
I've been debugging a slow PostgreSQL query on our users table.
The JOIN between users and user_profiles is using a sequential scan
instead of an index. We use SQLAlchemy 2.0 with asyncpg. The schema
has a GIN index on metadata.jsonb but a btree on profile_id. The
query plan shows 450ms on production. What's likely causing the
planner to skip the index?
```

Let the AI respond with its backend advice. You are not looking for a correct answer here — you are intentionally loading the context window with database, ORM, and query-planner vocabulary. That residue is the whole point.

Step 2 — In the SAME chat, now paste the unrelated frontend question:

```
Different question now: on the user profile page, the avatar image
isn't showing up — it's a broken image icon. Can you help me figure
out why?
```

> **What you should get**
> An answer tinged with backend assumptions. The AI may suggest checking whether the avatar URL is returning from the database correctly, inspecting the API response payload, or verifying the ORM relationship. Even though the bug is a broken image URL in a React component, the first half of the context window is still weighted toward your earlier SQL discussion. This is context pollution in the wild.

You should ask:

"Why did the AI suggest database fixes when I only asked about the frontend?"

Answer: The first half of the context window carried rich backend signal — table names, index types, ORM patterns. The AI cannot "un-know" that context. It reuses it as prior assumption on every follow-up, even across a topic change. This is exactly why the Fresh Chat rule exists.

Prompt 2 — Fresh Chat + Surgical @ Tagging

> **Remember**
> Ctrl+L (Windows) or Cmd+L (Mac) opens a fresh chat with zero prior context. Hit it every time you switch domains. Your future self will thank you — and your output quality will jump immediately.

Step 1 — Press Ctrl+L (or Cmd+L) now. In the fresh chat, use @ to attach ONLY these three files — nothing else:

```
@src/components/ProfileAvatar.tsx  —  owns the broken rendering behavior
@src/api/userApi.ts                —  defines how the user is fetched
@src/types/User.ts                 —  defines the User type and avatar field
```

If you are not using Cursor, paste the contents of these three files into the chat instead — but paste only these three, and label each one clearly.

Step 2 — Paste this RICE-structured prompt into the fresh chat:

```
ROLE: You are a senior React + TypeScript engineer. You debug UI
rendering bugs by reading the component and its data contract
first, without assuming backend or database problems.
```

```
INSTRUCTIONS: The <ProfileAvatar /> component in the attached
ProfileAvatar.tsx shows a broken image icon on the /profile route.
The user object is confirmed populated in the network tab — the
API response matches the User type exactly. Identify the most
likely cause and propose a minimal fix.

CONTEXT: Only the three attached files are in scope. Do not
suggest backend, API, or database changes. The API response
is verified correct and matches the User type contract.

EXPECTATIONS:
  - Name the exact file and line most likely responsible
  - Propose a one-to-three-line fix
  - Explain in one sentence why this is the fix, referencing
    the relevant User type field by name
  - Do not refactor unrelated code
```

What you should get

A precise answer. The AI identifies the exact line inside ProfileAvatar.tsx where `(user as any).profile?.avatar` is being read — a property path that does not exist on the User type. It flags the `as any` cast as the reason TypeScript missed the bug, notes that optional chaining short-circuits to undefined at runtime (hence the broken image), and proposes replacing that line with `user.avatarUrl` — the canonical field defined in types/User.ts. No suggestions to check the database, no unrelated refactors, no speculation about the backend. Same bug, dramatically cleaner output.

Tip:

Notice how the ROLE at the top of the prompt anchors the high-attention zone with “do not assume backend.” The EXPECTATIONS at the bottom close with the same guardrail. That is the U-shaped attention curve from the lecture working for you instead of against you — critical constraints live where the model is looking hardest.

Prompt 3 — The 10-Second Audit Habit

What to Learn

> Curation is a habit, not a one-time act. Before you ever hit Send, walk through a ruthless audit of every file you attached. Every file is a vote for where the model's attention goes. Files that do not earn their vote get removed — no exceptions.

Step 1 — Imagine a teammate attaches these six files to the same avatar bug:

```
@src/components/ProfileAvatar.tsx
@src/api/userApi.ts
@src/types/User.ts
@src/App.tsx
@src/styles.css
@src/pages/ProfilePage.tsx
```

Step 2 — Run the audit out loud on each file. Answer the single question: "Does the AI actually need this to solve this specific problem?"

```
ProfileAvatar.tsx  →  YES. Owns the broken render.            KEEP.
userApi.ts         →  YES. Defines how the user is fetched.   KEEP.
types/User.ts      →  YES. Defines the User contract.         KEEP.
App.tsx            →  NO.  Layout shell, unrelated.          REMOVE.
styles.css         →  NO.  Styling, not a rendering bug.      REMOVE.
ProfilePage.tsx    →  NO.  Passes the User through, fine.     REMOVE.
```

Step 3 — Now send the audited prompt (same RICE prompt from Prompt 2, but with only the three earned files in context).

> **What you should get**
> An answer equal in quality to Prompt 2, but with a smaller context window, lower token cost, and faster response. More importantly, you just practiced the habit that separates AI-native engineers from AI tourists: every prompt passes an audit before it leaves your keyboard.

Tip:

If you are ever unsure whether a file belongs, default to removing it. You can always add it back in the next turn. You can never un-pollute a context that has already been sent.

Wrap-Up — The Pattern, The Habit, The Handoff

> **Remember**
> Three prompts. One bug. Three dramatically different outputs. The takeaway: context is engineered, not dumped. The AI is only as smart as the signal you give it — and curation is how you keep the signal high.

Key takeaways:

- Fresh Chat (Ctrl/Cmd+L) resets the U-shaped attention curve whenever you switch domains
- @ tag only the files that directly own the broken behavior, its types, and its immediate dependencies
- Audit every prompt in 10 seconds — every attached file must earn its place
- Pair curated context with a RICE prompt: structure amplifies signal, and curation keeps that signal clean
- Relevant context beats more context, every single time