

AI Coding Exercise

Build this module's OpsMonitor sample with ChatGPT or Claude from a ready-made prompt that points the model at this module's specification, then review, run and extend what it gives you — including against our own reference build on GitHub.



Brief it, read it, then break it.

This exercise has three parts, and only the first is about prompting. You brief an assistant with the module specification and get an OpsMonitor solution back; you read what it wrote and decide whether it really does what the module asked; then you push it further than the module asked. The second and third parts are where the learning is.

What you are asking for

Fix the WisejPerfLab Refresh button. Profile the refresh scenario with CPU Usage, open the flame graph and Show Hot Path, and identify the two offenders: a formatter that runs per cell on every refresh and a control rebuild that recreates the KPI panel each time. Replace them with a DashboardSnapshot view model whose display strings are computed once in the service, mutate the labels and the chart data source once inside a probe scope instead of row by row, and keep the button disabled in a try/finally while the work runs. Rerun the identical scenario and capture the new hot path.

The module asks for these deliverables, and the prompt below asks the assistant for exactly them:

- ✓ Before CPU Usage trace naming the repeated formatter and the control rebuild with their total and self CPU
- ✓ DashboardSnapshot view model with display strings precomputed in the service
- ✓ Batched refresh handler that mutates controls once inside a probe scope with try/finally re-enable
- ✓ After CPU Usage trace of the identical scenario with the new hot path
- ✓ Before/after table of total CPU, top function and call count, plus the remaining risk

Why it is shaped this way

Module 2 ended with a name: TicketFormatter.FormatRow, 68.9% self CPU, called fifty thousand times in one click. That is the shape of almost every CPU problem in a server-side UI. Nobody wrote a slow method; somebody wrote a reasonable method and then arranged for it to run once per cell, per refresh, forever. The second offender in the WisejPerfLab dashboard is the mirror image: RebuildKpiPanel is called once, but it disposes and recreates a panel full of controls that did not need to change. One is a call-count problem and one is a work-per-call problem, and this module is about recognising which you have and fixing it without guessing.

These are the terms the assistant has to get right. If what comes back uses them loosely, it has not understood the module:

Term	What it means here
event-handler cost	A click, selection change or timer tick runs on the server and blocks that session's next event, so the amount of logic the click triggers is the amount of time the user is frozen.
flame graph	The same call-tree data drawn as width-proportional bars, which makes a single wide frame — one function eating the scenario — obvious at a glance.
self CPU vs call count	High self CPU with few calls means the body is expensive; low per-call cost with an enormous call count means the fix is to call it less, not to micro-optimize it.
precomputed view model	A small object carrying display-ready strings and values, built once by a service, so the UI layer assigns values instead of formatting them.
batched UI mutation	Setting the properties the user must see in one pass, with prepared values, instead of mutating controls inside a loop that also does work.
control rebuild	Disposing and recreating controls to refresh them; it re-runs construction, layout and theming, and it is almost always more expensive than updating the controls that exist.
immutable lookup cache	Reference data shared application-wide and never mutated; safe to cache globally, unlike per-user state which must stay in the session.

Build it — pick your assistant

Both buttons open a new tab with the **same ready-made prompt** already filled in. The prompt points the model at this module's specification — a single Markdown file holding the lesson, the lab brief and the deliverables — so it works from the real material instead of guessing what Module 3 is about. You can open [module-03.md](#) yourself to see exactly what it will read.



Use ChatGPT

Opens [chatgpt.com](#) with this module's prompt loaded. Works with GPT-5 or Codex; any plan that can browse the web will fetch the spec.



Use Claude

Opens [claude.ai](#) with the same prompt loaded. Claude Code users can paste it into a terminal in an empty folder instead.



Send it as-is and you get a project, not a snippet

The prompt ends by asking for the whole solution as a single [.zip](#) with build-and-run steps. Download it, open the solution in Visual Studio, press F5 and OpsMonitor comes up on [localhost](#). You need the .NET SDK and the [Wisej-4](#) NuGet package; everything else the prompt asks the assistant to write itself.

The prompt, in full — read it, then edit it if you want a different sample

I am working through Module 3 – "CPU Hot Paths, Event Handlers and Server-Side UI Work" – of the LearnWisej Performance & Profiling course, and I want to build the module sample with you.

Start by reading the module specification. It is a single Markdown file holding the full lesson, the lab brief and the deliverables:

<https://learnwisej.com/assets/courses/performance-and-profiling/specs/module-03.md>

Then build a complete Wisej.NET 4 project I can open in Visual Studio:

1. Project "OpsMonitor", C#, net8.0 or later, NuGet package Wisej-4 4.1.x, Kestrel host calling `app.UseWisej()`.
2. Build what the specification's lab asks for: Fix the WisejPerfLab Refresh button. Profile the refresh scenario with CPU Usage, open the flame graph and Show Hot Path, and identify the two offenders: a formatter that runs per cell on every refresh and a control rebuild that recreates the KPI panel each time. Replace them with a DashboardSnapshot view model whose display strings are computed once in the service, mutate the labels and the chart data source once inside a probe scope instead of row by row, and keep the button disabled in a try/finally while the work runs. Rerun the identical scenario and capture the new hot path.
3. Produce these deliverables, and tell me where each one lives in the solution:
 - Before CPU Usage trace naming the repeated formatter and the control rebuild with their total and self CPU
 - DashboardSnapshot view model with display strings precomputed in the service
 - Batched refresh handler that mutates controls once inside a probe scope with try/finally re-enable
 - After CPU Usage trace of the identical scenario with the new hot path
 - Before/after table of total CPU, top function and call count, plus the remaining risk
4. Follow the specification wherever it names a Wisej.NET class, property or event – use exactly that one. Where it leaves a choice open, tell me which API you picked and why.
5. Keep the server authoritative: state and validation live in .NET, and the browser renders and reports. Never move a decision into JavaScript because it is easier there.
6. Make the result demonstrable: I should be able to run it and see the behaviour this module is about, including what happens when something goes wrong.

Before you write any code, restate in your own words what you are going to build and where each piece will live, so I can correct the design while it is still cheap.

When the code is done, package the whole solution as a single .zip I can download, and give me the exact steps to build and run it.

Finally, answer these questions from the specification, using your own implementation as the evidence:

- Both offenders were slow, but for opposite reasons. Explain which was a call-count problem and which was a work-per-call problem, and why the profiler evidence for each looks different.
- A colleague proposes marking the refresh handler async to make it faster. What would actually change, what would not, and how would the trace show it?
- Your snapshot is built once per refresh. What staleness or correctness risk does that introduce, and what would you have to change if the same snapshot were cached across sessions?

Ask me about anything that is ambiguous before you start.

Then review what came back

Code that compiles is not yet code that is right. The interesting questions are design questions: is this the Wisej.NET control the job actually calls for, does it follow the lifecycle the framework gives you, is the authoritative state on the server rather than parked in the browser because that was convenient? Read the result as a reviewer, not as a customer.

This module gives you the checklist. Answer each one against the code you just received:

- ✓ Both offenders were slow, but for opposite reasons. Explain which was a call-count problem and which was a work-per-call problem, and why the profiler evidence for each looks different.
- ✓ A colleague proposes marking the refresh handler async to make it faster. What would actually change, what would not, and how would the trace show it?
- ✓ Your snapshot is built once per refresh. What staleness or correctness risk does that introduce, and what would you have to change if the same snapshot were cached across sessions?

Anything you had to correct is worth writing down. A short note of the conventions and decisions you settled on, pasted into the first message of your next session, is worth more than any single clever prompt.



We already built this one — compare yours with ours

The Module 3 reference implementation lives in the public samples repository. Build yours first, then open ours: the differences are the interesting part.



[Open the Module 3 sample on GitHub](#)

Or clone the whole set with `git clone https://github.com/iceteagroup/LearnWisej-Samples` and open `Performance and Profiling Course/Module 3`.



Our sample is a starting point, not a finished product

It was built the same way you are building yours — by prompt, then corrected — and it deliberately stops where the module stops. Expect stand-ins where production code would go, thin error handling and no tests. Treat that as an invitation: use what you learned in this module, plus the assistant, to push it further. Improving code you did not write is the skill this course is actually teaching.

Three prompts to take it further

Point your assistant at the project you just built — or at ours and try these. Each one is a review-and-extend exercise, not a rewrite.

- ✓ **Review it like a pull request.** *"Walk me through this OpsMonitor code as if you were reviewing a colleague's pull request and did not write it. Where does state live, what happens end to end on one user action, what would you send back for changes, and which parts would you not be comfortable putting in front of a customer? Be specific and quote the lines you mean — do not summarise the design back to me."*
- ✓ **Swap a piece without moving the seams.** *"Pick the part of this OpsMonitor project that is most obviously a stand-in for something real — a fake data source, an in-memory store, a simplified service — and replace it with a production-shaped implementation. The contract either side of it must not change: same types, same calls, same behaviour under failure. Tell me what the real implementation forced you to handle that the stand-in did not."*
- ✓ **Make it survive a bad day.** *"Review this OpsMonitor project for the failure modes it does not handle yet — bad or missing input, a dependency that is slow or unavailable, work arriving after the user navigated away, and two actions racing each other. For each one, tell me first whether it is a real risk in Wisej.NET 4 and why, and only then write the fix. Where you decide something needs no handling, say so and explain the reasoning rather than adding defensive code."*



Carry it into the lab

The hands-on lab that closes this module asks for the same deliverables you just had the assistant produce. Use what you have here as the shape of your answer and the evidence behind it — not as the answer itself. The lab is graded on your reasoning, and that part has to be yours.