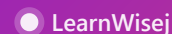


AI Coding Exercise

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.



The assistant is the fast pair of hands. You are still the architect.

An assistant can produce this module's sample in minutes. What it cannot do is decide whether the result is the right design for your application — that call stays with you. You supply the judgement: brief it properly, then check what comes back against what Module 2 just taught you.

What you are asking for

Profile the slow WisejPerfLab dashboard refresh properly. Switch to Release, warm the app up once, then collect a CPU Usage trace of that one scenario with Alt+F2, narrow the summary timeline to the click itself and use Show Hot Path to find the top path below the Wisej.NET dispatch frames. Collect a second Instrumentation trace of the same scenario to get call counts and wall-clock time, decide from the two traces whether the cost is CPU inside your code or time spent waiting, and save both .diagsession files with a name that carries the module, scenario, build, dataset and timestamp.

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

- ✓ Release CPU Usage trace of the dashboard refresh scenario, timeline narrowed to the click
- ✓ Hot path screenshot naming the top application function and its self versus total CPU
- ✓ Instrumentation trace of the same scenario with call counts and wall-clock time
- ✓ Trace note stating course, module, scenario, build, hosting, tool, browser and expected budget
- ✓ One-paragraph suspected root cause that says which of the five buckets the evidence points at

Why it is shaped this way

Module 1 left WisejPerfLab with a number and no explanation: the dashboard refresh takes 1,840 ms. The obvious next step is to profile it, and the obvious next step is where most investigations go wrong. A trace collected under the debugger measures the debugger. A trace started before the app is warm measures JIT. A trace that covers two minutes of clicking measures everything and proves nothing. A trace collected with four tools enabled at once measures the profiler's own overhead. This module is the mechanical discipline that makes a trace admissible evidence: the right build, the right tool for the bucket you suspect, the right process when the app is not started by Visual Studio, one scenario per collection, and a report you know how to read.

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
Performance Profiler (Alt+F2)	The post-mortem collection path: it can start the project without the debugger, collect a whole scenario and produce a saved report, which is where real evidence comes from.
Diagnostic Tools window	The live, debugger-attached view; useful for a quick look while stepping through code, misleading as a source of timings because the debugger changes what the runtime does.
tool selection matrix	CPU Usage for computation, Instrumentation for call counts and wall-clock, Object Allocation for churn, Memory Usage for retention, Database for queries, File I/O for files,.NET Async for await chains,.NET Counters for live health.
.diagsession	The saved trace file; naming it with module, scenario, build, dataset and timestamp is what makes a before-and-after comparison possible weeks later.
summary timeline	The CPU or memory graph across the whole collection; narrowing the selection to the exact user action is the step that removes startup and warm-up from the call tree.
hot path	The single most expensive chain of calls in the selected range; Show Hot Path expands it so you can find your own code under the framework frames.
self vs total CPU	Total includes children, self is the function body alone: high self means the method itself is expensive, high total with low self means the cost is in what it calls.

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 2 is about. You can open [module-02.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 2 – "Visual Studio Profiling Workflow for Wisej.NET" – 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-02.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: Profile the slow WisejPerfLab dashboard refresh properly. Switch to Release, warm the app up once, then collect a CPU Usage trace of that one scenario with Alt+F2, narrow the summary timeline to the click itself and use Show Hot Path to find the top path below the Wisej.NET dispatch frames. Collect a second Instrumentation trace of the same scenario to get call counts and wall-clock time, decide from the two traces whether the cost is CPU inside your code or time spent waiting, and save both `.diagsession` files with a name that carries the module, scenario, build, dataset and timestamp.
3. Produce these deliverables, and tell me where each one lives in the solution:
 - Release CPU Usage trace of the dashboard refresh scenario, timeline narrowed to the click
 - Hot path screenshot naming the top application function and its self versus total CPU
 - Instrumentation trace of the same scenario with call counts and wall-clock time
 - Trace note stating course, module, scenario, build, hosting, tool, browser and expected budget
 - One-paragraph suspected root cause that says which of the five buckets the evidence points at
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:

- Your CPU Usage trace accounts for only 300 ms of the 1,840 ms the probe measured. What does that gap mean, and which tool would you collect next?
- A framework method sits at the top of the hot path with a very high total CPU and almost no self CPU. Explain what that means and what you would look at instead of filing a framework bug.
- Why does a second person need the trace note to reproduce your result, when `the.diagsession` already contains the whole trace?

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:

- ✓ Your CPU Usage trace accounts for only 300 ms of the 1,840 ms the probe measured. What does that gap mean, and which tool would you collect next?
- ✓ A framework method sits at the top of the hot path with a very high total CPU and almost no self CPU. Explain what that means and what you would look at instead of filing a framework bug.
- ✓ Why does a second person need the trace note to reproduce your result, when the diagsession already contains the whole trace?

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



[Open the Module 2 sample on GitHub](#)

Or clone the whole set with `git clone https://github.com/iceteagroup/LearnWisej-Samples` and open `Performance and Profiling Course/Module 2`.



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.

- ✓ **Prove it actually works.** *"Go through this project deliverable by deliverable against the module specification at <https://learnwisej.com/assets/courses/performance-and-profiling/specs/module-02.md> and show me the evidence that each one is genuinely done — the code path, what to click, and what I should see. Where a deliverable is only partly met, say so plainly instead of arguing it through. Then add whatever is missing."*
- ✓ **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."*



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.