

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.

● LearnWisej



Writing the code is the easy half now.

Getting working code for "Memory, Allocations and Session Leaks" out of a modern assistant takes one good prompt. Judging it, correcting it and extending it takes everything this module covered. That is the split this exercise is built around — you supply the specification and the review, the assistant supplies the typing.

What you are asking for

Attack both halves of the memory problem in WisejPerfLab. Run the ticket search scenario under .NET Object Allocation and find the row models and strings allocated per search, then reduce them by projecting once instead of rebuilding objects per redraw. Next take a Memory Usage snapshot after warm-up, open and close the ticket detail form fifty times, take a second snapshot and compare: the retained forms are rooted by a static GlobalTicketBus.TicketChanged subscription and a refresh timer that was never stopped. Unsubscribe and stop both in an overridden Dispose, clear the BindingSource data source, then rerun the open/close scenario and show the retained heap returning close to snapshot A.

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

- ✓ Allocation trace naming the top allocating type and call path for the search scenario, before and after
- ✓ Snapshot A/B comparison showing the retained detail form instance count before the fix
- ✓ Path-to-root screenshot identifying the static event subscription and the live timer
- ✓ Dispose override that unsubscribes the static event, stops and disposes the timer and clears the binding
- ✓ Per-session memory budget and a disposal checklist, with the after-snapshot proving the retention path is gone

Why it is shaped this way

The WisejPerfLab dashboard now clears its CPU budget, and the app still falls over after four hours in production. Memory problems behave differently from CPU problems: they are invisible in a two-minute demo, they never show up as one slow click, and they are the reason a server that comfortably holds eighty sessions in the morning is refusing connections by mid-afternoon. There are two of them in the lab, and they are not the same problem. The ticket search allocates far more than it needs to, which costs GC time and CPU while it runs. The ticket detail form is never actually released after the user closes it, which

costs memory forever. One is found with an allocation trace, the other only with two snapshots and a path to root.

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
allocation pressure	How much memory a scenario creates while running; high pressure raises GC frequency and steals CPU even when nothing leaks.
retained memory	What is still reachable after the scenario should be finished; this is what multiplies by session count and decides capacity.
.NET Object Allocation	The tool that records every managed allocation during a scenario and groups it by type and by allocating call path.
Memory Usage snapshots	Heap captures taken at chosen moments; comparing snapshot A and snapshot B shows which types gained instances that were never released.
path to root	The chain of references keeping an object alive; it is the actual evidence of a leak, and the only way to prove the leak is gone.
retention root	The thing at the top of that chain — a static field, a static event, a running timer, a background task or a long-lived cache.
per-session memory budget	A number, in megabytes retained per idle session, that turns "the app uses a lot of memory" into a pass or fail and later into a health-check threshold.

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 4 is about. You can open [module-04.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 4 – "Memory, Allocations and Session Leaks" – 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-04.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: Attack both halves of the memory problem in `WisejPerfLab`. Run the ticket search scenario under `.NET Object Allocation` and find the row models and strings allocated per search, then reduce them by projecting once instead of rebuilding objects per redraw. Next take a Memory Usage snapshot after warm-up, open and close the ticket detail form fifty times, take a second snapshot and compare: the retained forms are rooted by a static `GlobalTicketBus.TicketChanged` subscription and a refresh timer that was never stopped. Unsubscribe and stop both in an overridden `Dispose`, clear the `BindingSource` data source, then rerun the open/close scenario and show the retained heap returning close to snapshot A.
3. Produce these deliverables, and tell me where each one lives in the solution:
 - Allocation trace naming the top allocating type and call path for the search scenario, before and after
 - Snapshot A/B comparison showing the retained detail form instance count before the fix
 - Path-to-root screenshot identifying the static event subscription and the live timer
 - Dispose override that unsubscribes the static event, stops and disposes the timer and clears the binding
 - Per-session memory budget and a disposal checklist, with the after-snapshot proving the retention path is gone
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 allocation trace improved dramatically but the snapshot comparison is unchanged. What kind of problem did you fix, what kind did you not, and how would a user notice the difference?
- Why does the lab open and close the form fifty times rather than once, and what would you conclude from a single instance appearing in snapshot B?
- The timer callback is one line long and only sets a label. Explain precisely how it keeps an entire form alive, and what an `IsDisposed` check does and does not solve.

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 allocation trace improved dramatically but the snapshot comparison is unchanged. What kind of problem did you fix, what kind did you not, and how would a user notice the difference?
- ✓ Why does the lab open and close the form fifty times rather than once, and what would you conclude from a single instance appearing in snapshot B?
- ✓ The timer callback is one line long and only sets a label. Explain precisely how it keeps an entire form alive, and what an IsDisposed check does and does not solve.

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



[Open the Module 4 sample on GitHub](#)

Or clone the whole set with `git clone https://github.com/iceteagroup/LearnWisej-Samples` and open `Performance and Profiling Course/Module 4`.



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.

- ✓ **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."*
- ✓ **Add the second case.** *"Extend this OpsMonitor project with a second case that follows the same pattern as the first — another entity, screen or control of the same shape, wired the same way. Factor out what is genuinely shared into a base class or service, but only where it earns its place; two similar things do not always want an abstraction. Show me the diff and justify each thing you extracted."*



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.