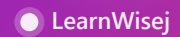


AI-Assisted Development, MCP-Ready Documentation, and Capstone Delivery

Lesson Guide · Lab / Exam Guide



Lesson Guide

AI-assisted coding discipline, documentation access, review prompts, final enterprise defense. What this means for an advanced Wisej.NET team — and how to read this module.



Production focus

AI-assisted coding discipline, documentation access, review prompts, final enterprise defense.

1. The production problem

Advanced teams will use AI tools, but enterprise Wisej.NET work still requires architecture, source-grounded documentation, code review, security review, and operational ownership.

2. Key concepts

Concept	Why it matters
AI-assisted development	Using AI tools to draft code, tests, and documentation while the team stays responsible for reviewing, grounding, and owning the result.
source-grounded prompts	Prompts that point the assistant at the actual project code, documentation, and conventions, so generated output matches the real architecture.
MCP documentation endpoint	A Model Context Protocol server that exposes the team's documentation to AI tools, so answers come from the project's own sources.
review checklist	A fixed list of architecture, security, and convention checks applied to every AI-generated change before it is accepted.
architecture defense	Presenting and justifying the design decisions of the capstone, with evidence, so the team can challenge them and record the outcome.
capstone demo	The end-to-end demonstration of the finished application, proving the module patterns work together under realistic conditions.

3. Lesson narrative

AI helps the team move faster

AI can draft screens, services, tests, prompts, documentation, and migration inventories. It cannot own production risk. The advanced developer must provide architecture context, verify APIs, review generated code, and test failure paths.

Use documentation as a tool source

Wisej.NET documentation can be treated as a source of truth for current platform behavior. AI prompts should ask for code that follows project conventions and should require citations or references when using platform-specific APIs.

Speed without unreviewed code

AI assistance is most valuable at the edges of the work: drafting a service skeleton from a command definition, generating test cases for a failure path matrix, writing a migration inventory from a solution file, or turning a runbook into a checklist. In each case the developer already knows what correct looks like and can verify the output quickly. It is least valuable where verification is hard, such as security logic, concurrency, or platform APIs the developer does not know well.

The rule the lab enforces is that generated code is a draft from an unknown contributor. It goes through the same review as any pull request, with the same questions: where does state live, what can fail, what is secure, and what changes in production. The lab's failure path shows why. A generated service invents a Wisej.NET API that does not exist and stores per-user data in a static field. Both would compile in a quick demo and both would fail under real sessions.

Accept a suggestion only when someone on the team can explain it without the tool present.

Prompts and checklists grounded in the project

The quality of generated code follows the quality of the context it was given. Every AI session in the lab starts with a prompt header that states the project rules: the solution structure, where business logic lives, that statics never hold session data, that services enforce permissions, and that only documented Wisej.NET APIs may be used, with a reference for each. The prompt library collects these headers and task-specific prompts so the whole team starts from the same baseline.

The generated-code review checklist is the other half. It is short enough to apply to every change and specific enough to catch the failures the course has taught: a static field holding user or tenant state, a UI handler containing persistence logic, an API that cannot be found in the documentation, a permission check missing from a service, an exception surfaced to the user, and a resource that is never disposed.

- Does the prompt name the conventions the output must follow.
- Does every platform-specific claim in the output cite documentation.
- Has the checklist been applied and signed by a reviewer, not by the author.

Documentation that tools and maintainers can read

A project that will be maintained with AI assistance needs documentation that a tool can retrieve, not only a person can browse. An MCP documentation endpoint exposes the team's own architecture notes,

decision records, conventions, and runbooks to the assistant, so answers about this project come from this project's sources rather than from general training data. The same endpoint serves a new team member asking the same questions.

That only works if the documentation is structured. The lab's documentation index lists every document with a one-line purpose, the decisions it records, and the date it was last verified. Architecture decision records from Module 1 are the most valuable entries, because they explain why a design exists and a tool can cite them when asked to change it. Runbooks, the permission matrix, the failure path matrix, and the configuration table follow.

Write for the reader who has never met the team. If a note relies on knowledge that exists only in someone's head, it will be wrong in a year.

Defending the capstone as an architecture review

The capstone is delivered as a production architecture review rather than a feature demo. The demo script walks a reviewer through the EnterpriseOps Command Center in a fixed order: the solution structure and decision records, a tenant-aware session and a rejected cross-tenant request, a command flowing through a service to the database and back with its audit entry, a background import with progress and cancellation, a workflow with a compensated failure, the diagnostics page, and the deployment package with its rollback.

Each stop pairs a working screen with a failure path, because the review is about what happens when things go wrong. The reviewer will ask where state lives, who can call what, how a bad release is reversed, and what the team would do at two in the morning. The production readiness statement answers those in writing, lists what is not yet done, and names the owner of each open item.

A defense succeeds when every decision can be traced to a reason, every risk has a named mitigation, and the team can say plainly what they would change next.



Advanced production habit

Do not stop when the screen looks correct. Explain state ownership, security implications, failure paths, and production behavior before considering the module complete.



Learning objectives

By the end of this module you'll be able to:

- ✓ Use AI assistance to accelerate Wisej.NET development without accepting unreviewed code.
- ✓ Create prompts and review checklists grounded in project standards and official documentation.
- ✓ Prepare MCP/documentation-ready project notes for future maintainers.
- ✓ Deliver and defend the Advanced capstone as a production architecture review.

Lab / Exam Guide

What you'll build in the hands-on lab, the suggested approach, and the required deliverables.



Hands-on lab

Finalize the capstone package. Add AI usage notes, prompt library, documentation links, code review checklist, final demo script, architecture defense deck outline, and production readiness statement.

Suggested approach

- ✓ Read the production problem and identify the architecture risk before touching the UI.
- ✓ Create or update the relevant service, command, model, or component boundary first.
- ✓ Build the Wisej.NET screen or component using the established project standards.
- ✓ Add a failure-path demonstration, not only the happy path.
- ✓ Record what changed in the learning log and update the relevant checklist.

Required deliverables

- ✓ AI prompt library
- ✓ Generated-code review checklist
- ✓ Project documentation index
- ✓ Capstone demo script
- ✓ Production readiness statement

Student review questions

- ✓ Which generated code was accepted and why?
- ✓ Was every platform-specific claim checked against documentation?
- ✓ Can the student defend deployment, security, and state decisions?



Advanced production habit

Do not stop when the screen looks correct. Explain state ownership, security implications, failure paths, and production behavior before considering the module complete.