Business Central in the Age of AI: From Manual to Multiplied

The ultimate competitive advantage is combining deep BC domain expertise with AI orchestration.

3-5x Faster Delivery

Developers write code 55% faster

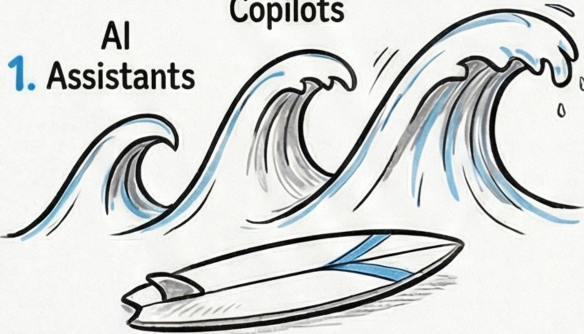
Consultants generate complex functional specifications in minutes rather than hours.

Rule #1: Context is Everything

Treat AI as a brilliant junior developer—it needs project context and your expert review to be effective.

Ride the Three Waves

1. AI Assistants
2. Embedded Copilots
3. Autonomous Agents



Move from using Assistants to Copilots and eventually Autonomous Agents.

10x Faster Data Migration

3 WEEKS	3 DAYS

Based on 1,500+ customers

NotebookLM

Business Central in the Age of AI

A Field Guide for Developers and Consultants

Miloš Mikulášek

2026 Edition · Version 1.0

This Is a Sample

You're reading a sample from the book **Business Central in the Age of AI — A Field Guide for Developers and Consultants**.

The complete book contains **47 chapters in 6 modules** covering everything from prompt engineering through Azure OpenAI integration in AL code to RAG, AI agents, and MCP servers.

This sample contains 6 selected chapters — one from each part of the book — to give you a sense of the style, depth, and practical focus of the content.

Complete book contents:

- **Module 0:** Introduction and Orientation (3 chapters)
- **Module 1:** AI as Your Development Partner (11 chapters)
- **Module 2:** AI for the Consultant's Daily Work (9 chapters)
- **Module 3:** AI Integration in Business Central (10 chapters)
- **Module 4:** Advanced Techniques (6 chapters)
- **Module 5:** Action Plan (4 chapters)
- **Appendices:** AL code reference, prompt templates, exercises

Get the complete book at **ai4bc.dev**.

Preface

This book was born out of a simple frustration: too many talented Business Central specialists spend too much time on things that AI could handle for them.

I've watched BC developers spend hours hunting for the right AL syntax for something Claude Code would write in thirty seconds. Consultants manually rewriting meeting notes into functional specifications, even though AI could produce a first draft in two minutes. Freelancers turning down projects because they don't have time for documentation — even though AI would give that time back.

At the same time, I've seen the other group. People who embraced AI and wove it into their workflow. They write code 3–5× faster. Specifications that used to take a day now take an hour. They offer their customers things the competition hasn't even seen yet.

This book is for the first group that wants to become the second.

About the Author

Miloš Mikulášek has worked with NAV and Business Central for over twenty years — as a developer, trainer, and someone who has been through every phase of the ecosystem, from the earliest versions of Navision to today's cloud-based BC. Beyond AL, he has SQL skills that let him solve problems where standard BC tools fall short.

He adopted artificial intelligence as a natural extension of his toolkit. He wasn't interested in the hype — he was interested in what AI can actually do in an environment he knows inside and out. The result is this book — *Business Central in the Age of AI* — and the community around it.

His approach is straightforward: no promises AI can't keep, no shortcuts where real system knowledge is needed. Just concrete tools and real-world examples from practice.

Find him at ai4bc.dev or at **milos.mikulasek@ai4bc.dev**.

Why AI Is Changing the ERP World (And Why It's Not Just Hype)

Two years ago, I was standing at a BC conference, coffee in hand, chatting with a fellow developer. He was experienced — 12 years in BC, knew NAV going back to version 3.60. He said: "AI? It's just hype. Nobody will be talking about it a year from now."

Today that same person works at a company that told him to "implement AI in BC by end of year." He's standing in front of a whiteboard covered in sticky notes, with no idea where to start.

I don't want that to happen to you.

This isn't a chapter about how AI will change everything and how you need to immediately overhaul the way you work. This chapter gives you context. Realistic, practical context. What's actually changing, why it affects BC developers and consultants specifically, and why now is the right time to start. Not because marketing says so — but because the data points to concrete things.

Let's be honest about what AI content on the internet usually looks like.

On one side, the enthusiasts: "AI will replace developers!" "AI will change everything in five years!" "Prompt engineering is a new superpower!"

On the other side, the skeptics: "Just hype." "Hallucinations are a problem." "My customers don't want AI."

And then you go back to work, open VS Code, stare at the AL code — at a codeunit you need to finish by Friday — and think: "So what now? How does any of this actually help me?"

None of those articles told you how to write a codeunit with AI assistance. How to describe BC architecture to an AI. How to work with CLAUDE.md on a real BC project. How to integrate Azure OpenAI into AL code in a way that's production-ready.

That's the problem this book solves. And this chapter is the starting point — I'll give you a framework to understand where we are, where we're headed, and what it means for you specifically.

What's Actually Changing — Data, Not Hype

Let's start with numbers. GitHub ran a developer productivity study with GitHub Copilot — they tested over 95 developers on specific tasks. The finding: an average 55% speedup on routine programming tasks. Fifty-five percent. That's not a negligible number.

McKinsey's 2023 study reports that AI can automate 30 to 50 percent of the time developers spend writing code. We're talking about tasks like boilerplate code, documentation, testing, and code review.

But here's the catch — these studies are based on general-purpose code. JavaScript, Python, Java. What about AL and Business Central?

Why BC Developers Have Both an Advantage and a Disadvantage

Let's get specific. Business Central occupies a unique position in the AI landscape.

The disadvantage: AL isn't represented in AI training data as well as mainstream programming languages. If you ask an AI to "write me an AL codeunit for X" without context, you'll get something that looks syntactically plausible but lacks knowledge of BC conventions, event architecture, and table relationships. AI doesn't know BC specifics like DataPerCompany, how extensions vs. the base app work, or how to properly subscribe to events.

The advantage: And here's the good news. Because you know what AL and BC mean, you have a huge edge over anyone using AI without that context. You know what the AI got right and what it got wrong. You understand the architecture. You're the domain expert — AI is your tool.

> � **Note:** A good BC developer with AI will be more productive than an average BC developer without AI. And significantly more productive than an excellent general developer with AI but no BC knowledge. Your domain expertise is the key — AI amplifies it, it doesn't replace it.

Three Waves of AI in the ERP World

I like to think about where we are through a metaphor of three waves. It helps you understand our current position and what's coming next.

Wave 1 — AI assistants for developers. These are tools like ChatGPT, Claude, and GitHub Copilot. They help you write, explain, and debug code, generate documentation, and answer technical questions. This is where we are now. This wave is real, accessible, and usable today. Modules 1 and 2 of this book cover it in detail.

Wave 2 — AI built directly into the product. Microsoft Copilot in Business Central, AI-powered search, automatic suggestions for users, and intelligent workflows. We're beginning to enter this phase. Microsoft is actively developing it, and recent BC releases include the first features. Module 3 focuses on this.

Wave 3 — Autonomous AI agents. Systems that monitor processes on their own, detect anomalies, and propose or execute actions within BC. This wave is on a two- to three-year horizon. We

discuss it in Module 4 because we want you to have the context — even though we can't fully implement it yet.

Why am I telling you this? Because understanding where we sit on the timeline helps you set the right expectations. Waves 1 and 2 are here and working. Wave 3 is coming — and those who prepare today will be ready tomorrow.

What This Means for Your Career

Let me tell you what I honestly think. The good news: BC and AL expertise remains essential. AI won't replace it anytime soon. Customers still need someone who understands their business, who designs solutions, and who takes responsibility for the outcome.

> ⚠ **Warning:** A developer who doesn't use AI will spend all day on work that an AI-assisted developer finishes by lunch. That doesn't mean AI will replace you — but you could be replaced by a developer who uses AI better.

And that developer doesn't have to be as experienced as you — because AI helps bridge part of the experience gap.

So: the question isn't whether to use AI. It's how to use it well. And that's exactly what this book will teach you.

Summary

Let's recap what we've covered.

First: AI in BC development isn't hype. There are real numbers behind it — a 55% productivity gain in the GitHub Copilot study, 30–50% of automatable time according to McKinsey.

Second: BC developers have a unique advantage. We understand the context that AI doesn't know on its own. We're domain experts — AI is a tool that amplifies that expertise.

Third: We're in Wave 1 — AI assistants are here today, accessible and usable. Wave 2 (AI in the BC product) is starting. Wave 3 (autonomous agents) is on the horizon.

Fourth, and most importantly: It's not a question of "whether." It's a question of "how well." And that's exactly what the rest of this book focuses on.

Real-World Examples: AI in BC Practice Today

To move beyond numbers and studies, let's look at concrete situations from BC specialists who already use AI.

Example 1: A Developer and Boilerplate Code

A developer gets this assignment: "We need a table for tracking service contracts with fields for customer, start date, end date, monthly fee, status, and notes. Plus a page for management and a codeunit for automatic expiration checks."

Without AI: Open VS Code, create the table, field by field. Then the page, field by field. Then the codeunit. Total: 45–60 minutes of routine work with no real decisions — just writing standard structure.

With AI: Submit a prompt with the requirements and conventions (prefix, comments), and in 5 minutes you have a complete table, page, and codeunit skeleton. Another 15 minutes on review and customization. Total: 20 minutes. Savings: 30–40 minutes on a single task.

Example 2: A Consultant and a Customer Workshop

A consultant has a discovery workshop tomorrow at a customer's site. She has the topic (changes to the shipping process) but no agenda. Previously, she'd spend an hour preparing.

With AI, in 15 minutes she has: a structured workshop agenda, key questions ranked by priority, a list of documents the customer should bring, and an overview of common BC issues in this area. She arrives at the workshop prepared — and the customer notices.

Example 3: A Freelancer and a Proposal

A freelancer receives an inquiry about a BC extension for lease management. He needs to write a proposal with estimates, scope, and pricing.

With AI: he inputs the requirements and the AI breaks the project into components, estimates hours for each, identifies risks, and proposes implementation phases. The freelancer edits the proposal, adds his rate, and in 30 minutes has a professional document — instead of 2 hours of thinking and formatting.

What These Examples Have in Common

In all three cases, AI doesn't make the decisions — the person does. AI handles the routine work: structuring text, generating boilerplate code, formatting documents. The person adds expert knowledge: BC architecture, customer specifics, pricing strategy.

This is the model that works. And it's exactly the model the rest of this book will teach you.

Try It Yourself

Exercise 1: Measure your baseline. This week, write down 3 tasks you do and how many minutes each one takes. By the end of the week, you'll have a benchmark to compare against when you start using AI.

Exercise 2: First contact with AI. If you haven't tried AI yet, open claude.ai or chatgpt.com and enter: "I'm a BC developer. Write me a simple AL procedure that checks whether a customer has an email address filled in." Does it work? What would you change?

This is a sample. The complete book contains 8 more chapters covering prompt engineering, Claude Code, GitHub Copilot, and a complete hands-on workshop. Get the full book at ai4bc.dev.

Prompt Engineering — Fundamentals for Developers

Two developers, same tool, same task. One gets usable code in 2 minutes. The other gets a generic result, spends an hour fixing it, and concludes "AI doesn't work."

The difference isn't the tool. It's how they described the task.

I've seen it dozens of times. A developer tries Claude or ChatGPT, types "write me a codeunit for invoices," gets a generic result, goes back to StackOverflow, and decides AI is all hype. Meanwhile, their colleague in the next office — with the same tool, the same BC version — gets exactly what they need on the first try.

Prompt engineering sounds academic, but it boils down to a simple thing: learning how to talk to AI so it understands your work. And for BC developers, there are specific patterns that work repeatedly.

The biggest mistake I see among developers who try AI and then stop using it: they describe tasks like a Google search. Short, no context. "Write me AL code for an invoice." "How do I fix this error?" "Make a report for customers."

AI is a different tool than Google. Google searches existing content — the exact wording of your query helps it find the right page. AI generates new content based on what you tell it. The less context you provide, the more it fills in the gaps — and it doesn't fill in your specific situation, but the average situation from its training data.

For BC, this creates a specific problem. The average situation in the training data is Python, JavaScript, web APIs. Not AL, not Business Central, not your specific project with its particular tables and naming conventions. That's why the result without context looks generic — because the AI didn't know it was working in a BC context and generated what it would for an average developer.

The solution is to add context.

Rule #1: Context Is Everything

Let's start with a basic experiment. I'll take the same task and describe it two ways.

Bad prompt: "Write me a codeunit for item validation."

Good prompt: "I'm a BC developer working in BC v27. We have an extension for e-shop integration. I need a codeunit that checks item availability in the Item table (Available Inventory field) for a list of Item No. values from a JSON array. If availability is below the requested quantity, return a JSON response with the error items. Naming prefix ZZ_, comments in English."

Did you see the difference in results? The second prompt gets you a result you can use directly in the project. You'll spend twenty minutes rewriting the first result — renaming variables, changing naming, adding BC-specific things that the AI doesn't know because you didn't mention them.

> 💡 **Tip:** Always start with four elements of context: (1) role and environment — "I'm a BC developer in BC v27," (2) what exactly you want — "I need a codeunit that…," (3) relevant tables and objects, (4) project conventions — prefix, comment language, error handling style. These four elements give the AI the foundation for an accurate result.

Rule #2: Break Big Tasks into Small Ones

AI is better at focused, bounded tasks than at large monoliths. Instead of "write me an entire module for project management with tables, codeunits, and pages," give the AI one codeunit, one report, one page extension. One task per prompt.

Why does this work better? With a small, bounded task, the AI better understands what you want and where the boundaries of the result lie. With a large task, it has to fill in many gaps at once — the relationships between tables, the interfaces between codeunits, how pages connect to logic. Each assumption is a potential source of error or deviation from what you want.

In practice, this means: First, break the task into parts. Maybe five minutes with pen and paper or in a notepad. What are the independent units? What can stand on its own? Then give each part to the AI separately. The result will be more accurate and easier to review than one giant prompt.

Rule #3: Iterate, Don't Start Over

When the AI generates code that isn't quite right, don't fire off a new prompt from scratch. Instead, make corrections in the same conversation. The AI remembers the context of the entire conversation — every message you sent and every response it gave.

Example of a corrective prompt I use: "This code looks good, but the mail sending uses an older approach. Modify it to use the Email codeunit from the standard BC library instead of direct SMTP."

Or: "The approval logic is correct, but I'm missing a permission check. Add verification that the calling user has 'Purchase Manager' authorization."

Or even more specific: "I'm adding a new field ZZ_CustomField to the customer object. How would you update the UpdateCustomer procedure to work with this field?"

This is how you iterate quickly and efficiently. Each iteration builds on previous work — you're not switching context, not rewriting the entire prompt, not re-explaining what the project does. The average conversation for a new codeunit takes three to five messages, not one. And that's normal — that's how it's supposed to work.

Rule #4: Show an Example

One of the most effective techniques is called few-shot prompting — showing the AI an example of what you want to get. Instead of describing what you want, show it a sample.

Instead of "write a comment for this procedure," say: "Write a comment for this procedure in the style of this example: [paste an existing comment from your project that you like]."

The AI immediately grasps the style, format, and level of detail you want. You don't have to describe it in words — just show it. Words like "brief but informative" are subjective. An example is objective.

This works great for naming conventions, error message style, documentation format — anything where "style" matters and is hard to describe in words. Take existing code from your project

that you like, paste it into the prompt as a sample, and the AI will draw inspiration from its style.

Rule #5: Ask for Reviews, Not Just Generation

This is perhaps the most underrated way to use AI, and yet one of the most valuable. Developers have gotten used to treating AI as a code generator. But AI is equally good as a reviewer.

Instead of "write me code," try: "Here's my code. Find potential problems and suggest improvements."

Or before starting implementation: "Here's the specification from the customer. What should I watch out for during implementation? What edge cases might I have missed?" This gives you a perspective you don't have on your own — the AI has seen an enormous amount of code and knows what typically goes wrong in similar situations.

Or after writing a codeunit: "Check whether I've made unnecessary database queries in loops. Where could there be a performance problem?" The AI will identify N+1 query problems, missing SetLoadFields calls, and places where the same data is read twice.

AI as a reviewer is a different role than AI as a generator. Both are extremely valuable. Developers who only use AI as a code generator are missing half the value AI offers them.

Summary

Five rules of prompt engineering for BC developers.

Rule one: Context is everything — always specify the role, environment, tables, and conventions. Without context, the AI generates an average result, not your situation.

Rule two: Break big tasks into small parts. One task per prompt — results are more accurate and easier to review.

Rule three: Iterate in the same conversation instead of starting over. The AI remembers the context — use it.

Rule four: Show an example — few-shot prompting works better than verbal descriptions for things like style and conventions.

Rule five: Use AI as a reviewer, not just a code generator. Half the value lies in getting a different perspective on your code.

These principles apply to all tools — Claude, ChatGPT, Copilot. They're principles of how to communicate with AI, not specifics of any one tool.

Try It Yourself

Exercise 1: Context experiment. Give the AI the task "Write AL code for customer validation" — first without context, then with full context (BC version, tables, conventions). Compare the results. How many lines differ?

Exercise 2: Iteration vs. new prompt. Have the AI write a codeunit. Find a problem in the response (there always is one). Try fixing it two ways: (a) a new prompt from scratch, (b) a corrective prompt in the same conversation. Which is faster and more accurate?

Exercise 3: Few-shot prompting. Take an existing comment from your project that you like. Paste it into the prompt as a sample and ask the AI for comments on three other procedures. Are they consistent with the sample?

This is a sample. The complete book contains 9 more chapters covering AI specifications, testing, documentation, troubleshooting, and SQL analysis. Get the full book at ai4bc.dev.

AI for Functional Specifications — From Notes to Document

Jana is a BC consultant who writes an average of three functional specifications every week.

Each one takes two to three hours. Meeting notes from a customer session, a few email threads, a couple of phone calls — and from all of that, she has to produce a structured document that developers can understand and the customer will sign off on. That's hours spent organizing text, crafting sentences, and maintaining consistent terminology — work that clearly has value, but requires zero creativity.

Last January, Jana tried using AI for one of those specifications. She took rough notes from a meeting — not even clean ones, just bullet points — and fed them to Claude with a simple prompt. The customer was a manufacturing company, the request involved adjusting invoicing for framework agreements, and Jana's notes were no longer than three paragraphs.

The result surprised her. The document was 80% done in 12 minutes. Structured, in clean Markdown, with every section in the right place. The remaining 20% she filled in herself — the business context, the customer's specifics, things the AI couldn't know because they weren't in the notes.

Twenty-five minutes total instead of two hours. Three specifications a week times 90 minutes saved on each — that's 4.5 hours back every week. Hours that Jana now spends on customers, strategic conversations, and project preparation — not documentation.

Writing functional specifications has one fundamental problem: it's routine work with high value but low creativity.

The structure of a specification repeats over and over — header, current state description, desired state description, business rules, edge cases, and test scenarios. AI handles this perfectly because it's structured text generation from given inputs.

What AI can't do without you: understand the customer's business culture. Know why the customer has these business rules. Catch unspoken assumptions that are considered obvious within the organization. That's you.

AI handles structure and phrasing. You bring the business context and expert judgment.

Workflow: From Customer Notes to Specification

The entire process has three steps. First, prepare the input material. Second, use a prompt template. Third, iterate on the weak spots in the output.

Step 1: Prepare the input material

You don't have to write cleanly. AI works well even with rough notes — sentence fragments, bullet points, even imprecise wording. What matters is that the input contains everything important. Write down what the customer said, what the current process looks like, what the customer wants to change, and any specific conditions or exceptions they mentioned. The more input material you provide, the better the output. AI can't conjure context you didn't give it.

In practice, right after a customer meeting, set aside five minutes and jot down the key points in a text file or notes app. Don't format, don't rewrite — just dump what you remember. This is the input for AI. It's better to have a lot of raw information than a few polished sentences. Experience shows that even 150 words of rough notes are enough for AI to generate a solid specification draft of 500 to 700 words.

Step 2: Use a prompt template for the specification

A basic template that works consistently well looks like this:

Start by telling the AI who you are and what you do: "I'm a BC consultant. The customer described this requirement for a change in Business Central." Then paste your notes or the customer's email. Then describe exactly what you want in the output — the structure of the specification, the document format, and the language.

Specifically, you want a header with the project name, customer, author, date, and version. Then a description of the current state — how it works in BC today. Then a description of the desired state. Business rules with conditions and logic. Edge cases and exceptions. Impact on other parts of the system. And finally, open questions — things you still need to clarify with the customer.

You want the format in Markdown.

Why this template works: AI gets a clear framework. It knows what you want, in what order, and in what format. The result is then consistent and predictable. Without a template, you'll get nicely written text, but the sections won't always be in the order that works for the developer or the customer.

Step 3: Iterate with AI on the weak spots

The AI output is never the final document. It's a solid foundation that you reshape into the final document. The most common weak spot is the edge cases section — AI generates generic examples but doesn't know the customer's specifics.

How to iterate: After the first output, write a follow-up prompt with specifics: "The edge cases section in this specification is too generic. The customer has these specifics: framework agreements can have multiple billing addresses, the customer requires separate invoices for each branch, and they occasionally send summary invoices on request. Expand the edge cases section to include these scenarios."

Result: detailed, customer-tailored edge cases in two minutes.

Tip: Header Template in CLAUDE.md

If you regularly work with the same customer or have a standardized header for your whole company, add the header template to your CLAUDE.md file. Simply note the customer name, BC version, your name, and preferred format. AI will then automatically fill in the header with the correct details on every subsequent prompt, without you having to enter this data repeatedly.

What If AI Makes a Mistake?

AI occasionally invents a business rule the customer never mentioned. Or phrases a requirement differently than you intended. You'll catch these errors when reading the output — and that's fine. That's why you read the entire specification, rather than blindly accepting it.

Always review the AI output. Verify the business rules. Fill in customer specifics that weren't in the input material. Remove sections that don't make sense. From an 80% finished document, you produce a 100% finished document — and that's still 75% less work than writing the specification from scratch.

With practice, you'll learn where AI is systematically weak — for one customer it might always be edge cases, for another it might be integration with a third-party system. You'll start checking those areas first, and iterations will shrink to minutes.

Summary

The workflow for functional specifications with AI is clear and repeatable.

First: prepare the input material. It doesn't have to be clean, but it has to be complete. AI works with what it gets.

Second: use a prompt template with the specification structure and output format. The result is 80% done.

Third: iterate on the weak spots — edge cases, customer specifics, things the AI didn't know.

The time savings are real: a typical specification that used to take two hours is done with AI in 25 to 30 minutes.

The key takeaway: AI doesn't write the specification instead of you. You're still the author and responsible for the content. AI handles the structure, phrasing, and consistency — freeing up your capacity for what you truly excel at as a consultant: understanding the customer's business, uncovering unspoken assumptions, and making expert judgments about what to implement and how.

Practical tip: right after a meeting, open Claude, paste your notes, and let it generate in the background while you write a follow-up email to the customer. The specification will be waiting for you when you hit send.

Case Study: A Consultant and a Specification for a Manufacturing Company

Jana, a BC consultant, handles requirements for a customer — a manufacturing company with 200 employees. She used to write specifications in 2–3 hours; now, with AI, it averages 35 minutes.

One specific case: the customer wanted a change to the shipping process — automatic generation of delivery notes with QR codes for a mobile warehouse. Jana's meeting notes were 12 bullet points. AI generated a 4-page specification in 8 minutes. Jana added the customer's specifics (QR code types, delivery note layout) in another 15 minutes.

The developer who implemented the specification said: "This is the clearest brief I've ever received from a consultant." Because AI doesn't improvise the structure — it follows a format that developers understand.

Try It Yourself

Exercise 1: Take notes from your last customer meeting and generate a specification using AI. Measure the time. Compare it with your usual time.

Exercise 2: Have AI generate an "Open Questions" section from a specification. How many of the questions are genuinely relevant? How many would you have missed on your own?

This is a sample. The complete book contains 16 more chapters covering Azure OpenAI integration, REST APIs in AL, Power Platform, GDPR, and more demo projects. Get the full book at ai4bc.dev.

Demo: Automatic Item Categorization with AI

This scenario comes up with many customers. A company migrates to a new BC tenant or runs a data migration from a legacy system. The result is always the same: the Item table in BC ends up with five hundred or two thousand items with no categories. The consultant gets the assignment: "Please categorize these." And then spends a week manually going through items, reading descriptions, and assigning categories. One consultant told me it took three days of pure effort.

With AI, you can do this in twenty minutes.

Problem Definition

We have items in BC. Each item has `No.`, `Description`, maybe `Description 2`, and an empty `Item Category Code` field. We have categories defined in the `Item Category` table — things like ELECTRONICS, FURNITURE, OFFICE, TOOLS, etc.

Manual categorization fails for three reasons. First: it's tedious work, and people start making mistakes after a while. Second: different people categorize differently — one says a printer is ELECTRONICS, another says OFFICE. Third: during migration there's pressure to move fast, categorization gets pushed to the end, and then it never gets done properly.

AI solves this. The model receives the item description and a list of allowed categories, and returns the best match. Fast, consistent, and tireless. It works well for 80–90% of cases — you review the remaining 10% manually, but that's still a massive time saving.

Extension Architecture

There are three parts:

Part one: Setup. We need somewhere to store the Azure endpoint URL and API key. We'll create a setup table `AI Categorization Setup` with these fields and a page for the administrator to configure them. The key should ideally be stored via Isolated Storage, not directly in the table.

Part two: Call logic. A codeunit `Item AI Categorizer` that: 1. Loads the list of categories from the `Item Category` table 2. For each uncategorized item, builds a prompt 3. Calls Azure OpenAI via the `OpenAIHelper` codeunit 4. Parses the response and writes the category back to the item

Part three: Execution. A report or page action that kicks off the whole process. Ideally with a progress bar, since 500 items can take a few minutes.

Code Walkthrough

We start with `OpenAIHelper.al`, which handles the low-level API calls. There's no business logic in it — just clean communication with Azure OpenAI.

Now let's look at the core of the application — `ItemAICategorizer.al`:

```
namespace BCDemoAI;

codeunit 50102 "Item AI Categorizer"
{
    procedure CategorizeAllItems()
    var
        Item: Record Item;
        OpenAIHelper: Codeunit "OpenAI Helper";
        CategoryList: Text;
        Prompt: Text;
        SuggestedCategory: Text;
        ProcessedCount: Integer;
        TotalItems: Integer;
        ProgressDialog: Dialog;
    begin
        // Load the list of existing categories for the prompt
        CategoryList := GetCategoryList();

        if CategoryList = '' then
```

```
        Error('No item categories are defined. Create categories in BC and try again.');

    // Filter only uncategorized items
    Item.SetFilter("Item Category Code", '%1', '');
    Item.SetFilter(Description, '<>%1', '');

    if Item.IsEmpty() then begin
        Message('All items are already categorized.');
        exit;
    end;

    TotalItems := Item.Count();  // Count once before the loop — calling Count() in every iteration

    ProgressDialog.Open('Categorizing items with AI...\Processing: #1####\Done: #2#### / #3####');

    Item.SetLoadFields(Description, "Item Category Code");
    if Item.FindSet() then
        repeat
            ProcessedCount += 1;
            ProgressDialog.Update(1, Item.Description);
            ProgressDialog.Update(2, ProcessedCount);
            ProgressDialog.Update(3, TotalItems);

            // Build a prompt with the item description and list of categories
            Prompt := BuildCategorizationPrompt(Item.Description, CategoryList);

            // Call the AI
            SuggestedCategory := OpenAIHelper.CallChatCompletionSimple(Prompt);

            // Validate that the returned category exists in BC
            SuggestedCategory := CleanAndValidateCategory(SuggestedCategory, CategoryList);

            // Write the result
            if SuggestedCategory <> '' then begin
                Item."Item Category Code" := SuggestedCategory;
                Item.Modify(true);
            end;
        until Item.Next() = 0;

    ProgressDialog.Close();
    Message('Done! Processed %1 items.', ProcessedCount);
end;
```

The key part is prompt engineering. Let's look at the `BuildCategorizationPrompt` function:

```
local procedure BuildCategorizationPrompt(ItemDescription: Text; CategoryList: Text): Text
var
    PromptBuilder: TextBuilder;
begin
    PromptBuilder.Append('You are an assistant for categorizing inventory items in Business Central.');
    PromptBuilder.Append(' Your task is to assign ONE category to the following item.');
    PromptBuilder.Append(' Reply ONLY with the category code, nothing else. No extra text.');
    PromptBuilder.AppendLine();
    PromptBuilder.Append('Available categories: ');
    PromptBuilder.Append(CategoryList);
    PromptBuilder.AppendLine();
    PromptBuilder.Append('Item to categorize: ');
    PromptBuilder.Append(ItemDescription);
    PromptBuilder.AppendLine();
    PromptBuilder.Append('Category:');

    exit(PromptBuilder.ToText());
end;
```

Let's walk through why this prompt is written the way it is.

"Reply ONLY with the category code, nothing else." This instruction is absolutely critical. Without it, the AI will return sentences like "Based on the item name, I recommend the category ELECTRONICS because…" You'd then have to parse that. With this instruction, it returns just "ELECTRONICS." Clean and simple.

"Category:" at the end with no value. This is a classic completion pattern. The AI "completes" the text after the colon. It works consistently.

We provide the list of categories. This is important! If the AI doesn't know your specific category codes, it will invent its own. Always include the available options in the prompt.

The second critical part is response validation:

```
local procedure CleanAndValidateCategory(AIResponse: Text; ValidCategories: Text): Text
var
    CleanedResponse: Text;
    ItemCategory: Record "Item Category";
begin
    // Remove extra whitespace and newlines
    CleanedResponse := AIResponse.Trim();
    CleanedResponse := CleanedResponse.ToUpper();

    // Verify that the category exists in BC
    if ItemCategory.Get(CleanedResponse) then
```

```
        exit(CleanedResponse);

    // If not, try to find a similar one (AI sometimes returns a slightly different format)
    ItemCategory.SetFilter(Code, '@' + CleanedResponse);
    if ItemCategory.FindFirst() then
        exit(ItemCategory.Code);

    // If nothing matches, return an empty string (the item stays uncategorized)
    exit('');
end;
```

> ⚠ **Warning:** Never blindly trust an AI response. Always validate against real data in BC. If the AI returns a category that doesn't exist, it's better to leave the item uncategorized than to write nonsense.

Results and Production Considerations

From testing on 50 items: the AI correctly categorized 44 — that's 88% accuracy without any prompt tuning. We leave the remaining 6 for manual review — but those 44 didn't require any manual work.

Before deploying this to a customer, a few things to consider:

Rate limiting. If you have 2,000 items, 2,000 back-to-back API calls will likely hit the Azure OpenAI rate limit. The fix: add a `Sleep(200)` between calls, or use a batch approach — send 10 items in a single prompt and ask for results as a JSON array.

Batch prompting. It's more efficient. One prompt saying "Categorize these 10 items, return a JSON array of codes" is cheaper than 10 individual calls. It requires slightly more complex parsing, but it's worth it.

Audit records. Log what the AI suggested and what was written. The customer may want to go back and review decisions. A simple logging table with the date, item number, and suggested category is enough.

Human review. I don't recommend deploying without a review step. A better pattern is: AI writes categories to a staging field or staging table, the consultant reviews the results, and confirms them in bulk. The customer then knows the results went through human review.

Complete Extension Code

For completeness, here are the full files you need for a working extension. These files aren't in the previous sections, but without them the extension won't run.

app.json

```json
{
  "id": "a1b2c3d4-e5f6-7890-abcd-ef1234567890",
  "name": "AI Item Categorizer",
  "publisher": "BCDemoAI",
  "version": "1.0.0.0",
  "brief": "Automatic item categorization using Azure OpenAI",
  "description": "Extension for bulk item categorization using AI.",
  "privacyStatement": "",
  "EULA": "",
  "help": "",
  "url": "",
  "contextSensitiveHelpUrl": "",
  "logo": "",
  "dependencies": [],
  "screenshots": [],
  "platform": "27.0.0.0",
  "application": "27.0.0.0",
  "idRanges": [
    { "from": 50100, "to": 50199 }
  ],
  "resourceExposurePolicy": {
    "allowDebugging": true,
    "allowDownloadingSource": true,
    "includeSourceInSymbolFile": true
  },
  "runtime": "14.0",
  "allowedExternalUrls": [
    "https://*.openai.azure.com"
  ]
}
```

Setup Table — AI Categorization Setup

```
namespace BCDemoAI;

table 50101 "AI Categorization Setup"
{
    Caption = 'AI Categorization Setup';
    DataClassification = SystemMetadata;

    fields
```

```
    {
        field(1; "Primary Key"; Code[10])
        {
            Caption = 'Primary Key';
            DataClassification = SystemMetadata;
        }
        field(10; "Endpoint URL"; Text[250])
        {
            Caption = 'Azure OpenAI Endpoint URL';
            DataClassification = SystemMetadata;

            trigger OnValidate()
            begin
                // Validate the URL format
                if ("Endpoint URL" <> '') and (not "Endpoint URL".StartsWith('https://')) then
                    Error('Endpoint URL must start with https://');
            end;
        }
        field(20; "Deployment Name"; Text[100])
        {
            Caption = 'Model Deployment Name';
            DataClassification = SystemMetadata;
        }
        field(30; "API Version"; Text[50])
        {
            Caption = 'API Version';
            DataClassification = SystemMetadata;
            InitValue = '2025-01-01-preview';
        }
        field(40; "Max Tokens"; Integer)
        {
            Caption = 'Maximum Response Tokens';
            DataClassification = SystemMetadata;
            InitValue = 100;
            MinValue = 10;
            MaxValue = 4000;
        }
        field(50; "Temperature"; Decimal)
        {
            Caption = 'Temperature (0.0-2.0)';
            DataClassification = SystemMetadata;
            InitValue = 0.1;
```

```
        MinValue = 0;
        MaxValue = 2;
    }
    field(60; "Batch Size"; Integer)
    {
        Caption = 'Items Per API Call';
        DataClassification = SystemMetadata;
        InitValue = 1;
        MinValue = 1;
        MaxValue = 20;
    }
    field(70; "Delay Between Calls Ms"; Integer)
    {
        Caption = 'Delay Between Calls (ms)';
        DataClassification = SystemMetadata;
        InitValue = 200;
        MinValue = 0;
        MaxValue = 5000;
    }
}

keys
{
    key(PK; "Primary Key")
    {
        Clustered = true;
    }
}

procedure GetSetup()
begin
    if not Get() then begin
        Init();
        if not Insert() then
            Get();
    end;
end;

procedure StoreApiKey(ApiKey: Text)
begin
    IsolatedStorage.Set('AzureOpenAIKey', ApiKey, DataScope::Company);
end;
```

```
    procedure GetApiKey(): Text
    var
        ApiKey: Text;
    begin
        if not IsolatedStorage.Get('AzureOpenAIKey', DataScope::Company, ApiKey) then
            Error('API key is not set. Set it on the AI Categorization Setup page.');
        exit(ApiKey);
    end;

    procedure HasApiKey(): Boolean
    begin
        exit(IsolatedStorage.Contains('AzureOpenAIKey', DataScope::Company));
    end;

    procedure GetFullEndpointUrl(): Text
    begin
        GetSetup();
        TestField("Endpoint URL");
        TestField("Deployment Name");
        exit("Endpoint URL" + '/openai/deployments/' + "Deployment Name" +
             '/chat/completions?api-version=' + "API Version");
    end;
}
```

Setup Page

```
namespace BCDemoAI;

page 50101 "AI Categorization Setup"
{
    PageType = Card;
    SourceTable = "AI Categorization Setup";
    Caption = 'AI Categorization Setup';
    ApplicationArea = All;
    UsageCategory = Administration;
    InsertAllowed = false;
    DeleteAllowed = false;

    layout
    {
        area(Content)
        {
```

```
        group(AzureOpenAI)
        {
            Caption = 'Azure OpenAI';

            field("Endpoint URL"; Rec."Endpoint URL")
            {
                ApplicationArea = All;
                ToolTip = 'URL of your Azure OpenAI resource, e.g. https://my-ai.openai.azure.com/'
            }
            field("Deployment Name"; Rec."Deployment Name")
            {
                ApplicationArea = All;
                ToolTip = 'Name of the model deployment in Azure, e.g. gpt4o-mini-v1';
            }
            field("API Version"; Rec."API Version")
            {
                ApplicationArea = All;
                ToolTip = 'Azure OpenAI API version';
            }
            field(ApiKeyStatus; ApiKeyStatusTxt)
            {
                ApplicationArea = All;
                Caption = 'API Key';
                Editable = false;
                StyleExpr = ApiKeyStyle;
            }
        }
        group(Parameters)
        {
            Caption = 'AI Parameters';

            field("Max Tokens"; Rec."Max Tokens")
            {
                ApplicationArea = All;
            }
            field(Temperature; Rec.Temperature)
            {
                ApplicationArea = All;
            }
            field("Batch Size"; Rec."Batch Size")
            {
                ApplicationArea = All;
```

```
                ToolTip = 'How many items to send in one API call (1 = individually)';
            }
            field("Delay Between Calls Ms"; Rec."Delay Between Calls Ms")
            {
                ApplicationArea = All;
                ToolTip = 'Delay between calls in milliseconds (rate limiting prevention)';
            }
        }
    }
}

actions
{
    area(Processing)
    {
        action(SetApiKey)
        {
            Caption = 'Set API Key';
            Image = EncryptionKeys;
            ApplicationArea = All;

            trigger OnAction()
            var
                ApiKeyInput: Text;
            begin
                if Page.RunModal(Page::"AI API Key Input", ApiKeyInput) = Action::OK then begin
                    Rec.StoreApiKey(ApiKeyInput);
                    UpdateApiKeyStatus();
                    CurrPage.Update(false);
                end;
            end;
        }
        action(TestConnection)
        {
            Caption = 'Test Connection';
            Image = TestReport;
            ApplicationArea = All;

            trigger OnAction()
            var
                OpenAIHelper: Codeunit "OpenAI Helper";
                Response: Text;
```

```
                begin
                    Response := OpenAIHelper.CallChatCompletionSimple('Say "OK".');
                    Message('Connection works. AI responded: %1', Response);
                end;
            }
        }
    }

    trigger OnOpenPage()
    begin
        Rec.GetSetup();
        UpdateApiKeyStatus();
    end;

    var
        ApiKeyStatusTxt: Text;
        ApiKeyStyle: Text;

    local procedure UpdateApiKeyStatus()
    begin
        if Rec.HasApiKey() then begin
            ApiKeyStatusTxt := '⍰ Set';
            ApiKeyStyle := 'Favorable';
        end else begin
            ApiKeyStatusTxt := '⍰ Not set';
            ApiKeyStyle := 'Unfavorable';
        end;
    end;
}
```

Logging Table for Audit

```
namespace BCDemoAI;

table 50102 "AI Categorization Log"
{
    Caption = 'AI Categorization Log';
    DataClassification = CustomerContent;

    fields
    {
        field(1; "Entry No."; Integer)
```

```
        {
            Caption = 'Entry No.';
            AutoIncrement = true;
        }
        field(10; "Item No."; Code[20])
        {
            Caption = 'Item No.';
            TableRelation = Item."No.";
        }
        field(20; "Item Description"; Text[100])
        {
            Caption = 'Item Description';
        }
        field(30; "Suggested Category"; Code[20])
        {
            Caption = 'Suggested Category';
            DataClassification = CustomerContent;
        }
        field(40; "Applied Category"; Code[20])
        {
            Caption = 'Applied Category';
            DataClassification = CustomerContent;
        }
        field(50; "Was Valid"; Boolean)
        {
            Caption = 'Was Valid';
        }
        field(60; "Processing DateTime"; DateTime)
        {
            Caption = 'Processing Date/Time';
        }
        field(70; "Tokens Used"; Integer)
        {
            Caption = 'Tokens Used';
        }
    }

    keys
    {
        key(PK; "Entry No.")
        {
            Clustered = true;
```

```
        }
        key(K1; "Item No.")
        {
        }
        key(K2; "Processing DateTime")
        {
        }
    }
}
```

Try It Yourself

Exercise 1: Run the categorization. Install the extension on a dev sandbox, create 10 test items without categories (e.g., "HP LaserJet Printer," "Ergonomic Office Chair," "Phillips Head Screwdriver PH2"). Set the Azure OpenAI endpoint on the setup page and run the categorization. How many items did the AI classify correctly?

Exercise 2: Modify the prompt. In the `BuildCategorizationPrompt` procedure, change the instruction: instead of "Reply ONLY with the category code," try "Reply with the category code and a one-sentence justification in the format CATEGORY: reason." How does the output change? What do you need to adjust in the parsing?

Exercise 3: Batch approach. Modify the codeunit to send 5 items in a single prompt and expect a JSON array of responses. Hint: prompt "Categorize these 5 items, return a JSON array: [{'item': 'description', 'category': 'CODE'}]." How does it affect speed and cost?

Summary

First: Prompt engineering is decisive. The instruction "reply only with the category code" and the list of allowed values are essential for usable results.

Second: Always validate the AI response against BC data. Never blindly write what the AI returned without checking that it makes sense in the context of your system.

Third: Consider a batch approach for large volumes — instead of one call per item, send multiple items at once. Cheaper and faster.

Fourth: A complete extension needs more than just business logic — it also requires a setup table with Isolated Storage for the API key, an admin configuration page, and a logging table for audit. These "boring" parts are just as important in production as the AI code itself.

> **This is a sample.** The complete book contains 12 more chapters covering RAG, AI agents, MCP servers, fine-tuning, and a career roadmap. Get the full book at ai4bc.dev.

A 30-Day Plan for Bringing AI into Your Work

The most common outcome of an online course isn't a bad review or disappointment. The most common outcome is nothing. Someone finishes the course, feels great, thinks "awesome, I'll start using this" — and three weeks later they're back to old habits. AI feels complicated, there's no time, projects are running, and that feeling of "I should get started" slowly turns into guilt.

That's exactly why this chapter is different — it's not theory, it's a 30-day action plan. Concrete, realistic, and tailored to who you are.

Why People Don't Change After a Course

Three patterns I see over and over again among developers and consultants.

The first is analysis paralysis. You learn so many tools, so many possibilities, that you don't know where to start. Copilot or Claude? Should I go through all the modules again first? Should I learn more about prompting? The result: you do nothing.

The second is "when I have time." Projects are running at full capacity, the customer is waiting, the boss is pushing. AI is a bonus, not a necessity, so it always gets sidelined. But "when I have time" never comes. You either make time for it, or AI has to earn its keep by saving you time.

The third is perfectionism. "I'll wait until I understand RAG better." "I still need to study MCP servers." "Once I really know this stuff, then I'll start using it." But the only way to learn AI is to start using it. You can't study your way there in advance.

The solution to all three? Small steps, consistently, on real projects. Not on practice exercises. Not on demo data. On your actual tasks — the ones sitting on your desk today.

Week 1 — Foundations in Practice

The goal of the first week is simple: get your tools up and running and use them on one real problem.

Specifically — install Claude Code if you haven't already. Not download it, not read the docs — install it and open it. Then create a CLAUDE.md file for your current BC project. It doesn't have to be perfect — 10 lines is enough: what the project is, which BC version, what the main objects are, what the tables you work with are called. That's it.

> � **Tip:** Find one real bug or task from your backlog and try solving it with AI. Not an exercise — an actual ticket. If AI helps, great. If it doesn't, you'll find out why — and that's valuable information too.

Petr — would take one bug from the last sprint: a faulty validation in a codeunit, a broken report, or an incorrectly calculated field. He describes the problem in Claude Code and watches how AI suggests a fix.

Jana — doesn't develop, but has a specification or test scenario ahead of her. She takes one real process from a customer project and lets AI write a draft specification. She doesn't know AL? Doesn't matter — this doesn't require AL.

Martin — as a freelancer, he probably has an in-progress project and a pile of correspondence. He takes one email to a customer that he's been putting off because he doesn't have the energy for it — and lets AI draft the structure of a response.

Week 2 — Building the Routine

The goal of the second week isn't revolution. It's routine.

Every working day — every day, not every other day — use AI on at least one work task. One email. One specification. One piece of code. One test case. Anything from your actual work.

And keep a short journal. Really short — two sentences. "Today I used AI on X. It took Y minutes instead of Z minutes." Or: "AI suggested a solution that didn't work because…" This isn't overkill — it's how you start seeing patterns. What AI saves you time on, where it slows you down, what it's good at, and what it's not.

Why a journal? Because in three months, when you're convincing your boss or a colleague, you won't be saying "I feel like it helps me." You'll be saying "I wrote this specification in 40 minutes instead of three hours" or "I solved this bug in 20 minutes — normally I'd spend an hour and a half searching." Those are the arguments that work.

Week 3 — Going Deeper

By now you've been using AI for two weeks. You're starting to feel where it costs you time. Week three is about taking one specific recurring task — something you do at least once every two weeks that takes 30 minutes or more — and creating a prompt template for it.

What's a prompt template? It's a pre-built prompt containing context, instructions, and output structure. You save it to a file, open it next time, fill in the specifics, and go. You don't have to reinvent how to ask AI for help every time — you have a template.

Petr would create a template for code review of an AL object: what the rules are, what to look for, what the output should look like. **Jana** would create a template for a functional specification of a BC process: what the required fields are, how to describe the workflow, what the exceptions section must include. **Martin** would create a template for a customer proposal: how to estimate effort, how to write an executive summary, how to scope the work.

One template. By the end of week three. That's it.

Week 4 — Sharing

Week four is about going from "I'm doing this" to "we're doing this." And for three reasons.

First — when you explain something to a colleague, that's when you truly understand it. Show one colleague how you use AI. It doesn't have to be a presentation — five minutes is enough: "Hey, look at how I solved this." That's it.

Second — write one LinkedIn post or an internal tip for your team. It doesn't have to be long, it doesn't have to be perfect. But by writing it publicly or sharing it with the team, you create commitment. And you force yourself to articulate what you've learned.

Third — propose one specific AI project to your boss or team lead. Not "we should implement AI." Something concrete: "I propose we try automating test scenario generation for releases. I estimate it'll save us X hours per release." Small, measurable, specific. How to prepare a proposal like this is covered in the next chapter.

How to Measure Progress

Measuring progress is key. It's not enough to say "I feel more productive" after 30 days. That's useless — for you and for making your case to the boss.

Instead, keep a simple spreadsheet. Three columns: task, time without AI, time with AI. Add three to five rows every week. After 30 days, you'll have concrete data. You'll know that a specification takes 40 minutes instead of three hours, that code review takes 15 minutes instead of an hour, that you write a customer email in 5 minutes instead of 25.

Those are numbers you can do something with. Those are numbers that convince the boss. Those are numbers that show you where the biggest value is.

Summary

The 30-day plan isn't complicated. Week one: install, set up, use on one real problem. Week two: one task a day with AI, short journal. Week three: one prompt template for recurring work. Week four: show a colleague, write a post, propose a project.

The key rule: real projects, not exercises. Small steps, consistently. Measure time, not feelings.

The biggest mistake you can make is waiting until you have more time or more knowledge. Time and knowledge come from starting. Not any other way.

Try It Yourself

Exercise 1: Week 1 — today. Don't put it off. Right now: open an AI tool and give it one real task from your to-do list today. Measure the time. Write down the result.

Exercise 2: Tracking spreadsheet. Create a simple spreadsheet (Excel or paper): Task | Time without AI | Time with AI. Fill in at least 3 rows this week.

Exercise 3: Prompt template. Identify one recurring task you do at least once every 2 weeks. Write a prompt template for it and save it.

Detailed Weekly Checklist

Week 1: Checklist

- ☐ Claude Code installed and working
- ☐ CLAUDE.md file created for your current project
- ☐ One real task completed with AI assistance
- ☐ Note: how many minutes it took vs. your estimate without AI
- ☐ GitHub Copilot installed in VS Code (if you're a developer)

Week 2: Checklist

- ☐ Monday: first task with AI
- ☐ Tuesday: second task with AI
- ☐ Wednesday: third task with AI
- ☐ Thursday: fourth task with AI
- ☐ Friday: fifth task with AI + journal entry
- ☐ Journal: 5 rows in the format "task | time without AI | time with AI"

Week 3: Checklist

- ☐ Identify one recurring task (at least once every 2 weeks)
- ☐ Write a prompt template for this task
- ☐ Save the template somewhere easily accessible (Notion/VS Code/file)
- ☐ Test the template on a real task
- ☐ Refine the template based on the result

Week 4: Checklist

- ☐ Show one colleague how you use AI (5 minutes)
- ☐ Write one LinkedIn post or internal tip for the team
- ☐ Propose one specific AI project to your boss/colleagues
- ☐ Calculate monthly savings from your journal data (week 2)
- ☐ Decide: do I keep going? What do I add in month 2?