$ whoami
> junior_developer

The Agentic Junior

How a junior developer actually works with AI agents

```
# CLAUDE.md
## How I work
- I'm still levelling up. When you
  explain something, show the
  reasoning, not just the answer
  — I want to understand the why.
```

second brains · hooks · hive-minds · guardrails

Kevin Demeulenaere

The Agentic Junior

Free sample: Introduction and Chapter 1

Kevin Demeulenaere

Introduction

Everyone has an opinion about AI and junior developers, and most of those opinions end the same way: badly, for us. The entry-level work is drying up. The agents write the boilerplate now. Why hire a junior when a model does the junior's job in forty seconds?

I think that argument is backwards. It has the facts right (agents really do write the boilerplate now) but draws the wrong conclusion from them. Treat the agent as a rival and you lose: it types faster than you, even if you copy paste. But learn to run them instead, to brief them properly, box them in, check their work and make them explain everything, and you learn stupidly fast. Faster than the seniors, in some ways, because we have more to learn and every agent-driven task now comes with its reasoning attached, if you insist on it.

That's the whole premise of this book. AI already changed the job, so I won't pretend otherwise. I'm also not going to feed you "prompt harder and you'll be fine"; that's LinkedIn filler, and there will be none of it here. The premise is narrower and, I think, truer: driving agents well is a learnable skill, and if you pick it up early, the thing that supposedly threatens you becomes the best learning setup you're likely to find as a junior, better than any mentoring arrangement I could realistically get.

Who is telling you this

I'm a junior developer on a three-person team, at a company that builds software for local governments. My weeks split between maintaining legacy .NET applications and greenfield work in Vue 3, TypeScript, and

modern .NET. The legacy stuff is old in the unglamorous way: original authors long gone, comments that lie. I'm Belgian; my personal notes are partly in Dutch, my team notes in English. I work with Claude Code every day. Not an experiment, just how the work gets done now.

I am not a guru, and this is junior-to-junior advice: everything I recommend is something I do myself, and every warning comes from something that bit me first. I have no course to sell you and no company to promote. What I have is about a year of running agents on real work: legacy bug fixes, greenfield features, a personal-finance pipeline that has to reconcile to the bank to the cent. That, and a stack of notes on what worked, what broke, and what broke in ways I only noticed weeks later. The broken parts are in this book too. An agent of mine once wrote a chart caption whose arithmetic was simply wrong, with total confidence, and I shipped it into my own notes. More on that later.

What you'll be able to do by the end

This is a practical book, not a manifesto, so I'll be concrete. By the last page you will be able to:

- **Teach an agent your world.** Write the instruction file (in Claude Code, a `CLAUDE.md`) that tells an agent how your project is structured, what your conventions are, and how you work. I get more value out of that one file than anything else in my setup, and hardly anyone bothers to write one.
- Build a second brain the agent maintains: a plain-markdown vault (I use Obsidian) that acts as external memory for you *and* the agent, with daily logs, per-project notes and a learning folder that compounds. Sessions come and go; the notes are what you keep. It's also how

corrections become permanent: one-off feedback ("never do X like that again") turns into standing rules in a durable memory file, so the agent behaves tomorrow the way you corrected it today.

- Automate the deterministic parts with hooks: small scripts that run at session boundaries (mine regenerate finance charts whenever the underlying notes change), plus a feel for what belongs in a hook and what belongs in an instruction.
- **Run more than one agent when it pays.** Parallel read-only auditors for sweeping and finding, a single writer for changes, and verification passes that catch the agent's own mistakes, including the confident ones.
- **Scale it to a team.** A shared vault as the team's source of truth, routing rules for writing to both team and personal notes, and the provenance rule: work is always attributed to a human, never to an agent.
- Hold the guardrails: the short list of things you never let an agent do, and why those rules can't bend even when bending would be convenient.
- **Keep learning while delegating.** The habits that stop agent use from hollowing out your own growth; if you let the agent do everything and never follow along, you'll ship a lot in a year and learn nothing.

How to read this book

The chapters build on each other, roughly in the order I learned things: first the instruction file, then the vault, then memory, hooks, multi-agent work, the team layer, and guardrails. Chapter 9 is three end-to-end walkthroughs with real prompts and real friction. Chapter 10 is about protecting your own learning, and it might be the most important one.

If you're impatient, you can read chapters 2 and 3 and start working; everything else layers on top of those two. But read chapter 8 before you give an agent anything resembling production access. Guardrails feel like pure overhead, right up until an agent tries something you never wanted it to do, and then they're all that matters.

Tool examples throughout use Claude Code, because that's what I use and I refuse to write about tools I don't run daily. None of the ideas are tied to it, though. Every chapter ends with a short box (*the principle, tool-agnostic*) that states the underlying idea in a form you can carry to whatever tool you're using by then, even after half of today's tooling has been renamed or abandoned. The tooling will keep churning, but I'd bet on the ideas in those boxes still holding up in three years.

Configs and snippets in this book are real in shape but sanitized in content: invented example numbers, generic names, nothing traceable to my employer or my bank account. The failures, though, I left as they happened; those aren't cleaned up.

One warning before we start. None of this is passive; running agents well takes work, and the work is briefing, constraining, verifying, and asking *why* until you understand the answer. Juniors who put in that work aren't the ones getting replaced. If anything, they're the ones teams end up fighting over.

01. The junior's edge

There's a line you hear at every meetup and under every thinkpiece: AI is coming for the junior developers first. The reasoning is always identical. Juniors do the simple work. Agents do the simple work now. Therefore, no more juniors.

I want to make the opposite argument. Not from theory, but from a year of running agents every working day on exactly the kind of code juniors get handed, and watching what that did to my learning curve.

The junior who learns to drive agents well is the best-positioned person in the industry right now, and being junior is a big part of why.

That's the claim this book defends. This chapter explains why I believe it, what changes about the job in practice, and where the whole thing goes wrong, because credibility matters more than optimism.

The part of the fear that's true

Let's not pretend the fear is baseless. Part of it is dead right, and you should stare at it rather than around it.

The traditional junior deal worked like this: you were cheap implementation labour. A senior sketched the solution, you typed it out, and in exchange for your typing, the company tolerated your questions. The typing was your ticket into the room.

An agent types faster than you. It types faster than your senior. It knows more syntax, more standard library, more framework trivia than anyone on

your team. If your entire value proposition is "I convert a clear description into working code", then yes, that proposition is in trouble, and no amount of motivational posting changes it.

So the honest version of the fear: *the ticket into the room is being devalued.* What the thinkpieces miss is that the typing was only ever the way in. The value was in what you picked up once you were there. You learned. You absorbed how systems fit together, why decisions were made, what breaks and how. That used to be slow, rationed, and dependent on how much patience the seniors around you had left on a Thursday afternoon. With an agent, most of that rationing disappears.

What compounds

A senior developer's advantage is accumulated experience: ten thousand small lessons about what works, stacked over years. Nobody hands you that. You accumulate it or you don't.

The old way of accumulating it was grinding tickets. You'd fix a bug, half-understand the fix, move on, and maybe three months later the pattern would click. Explanation was scarce. The senior who could explain *why* the fix worked had eleven other things to do, and you learned to budget your questions.

Working with an agent inverts the scarcity. Every task can now explain itself, if you insist on it. The agent that just found the bug can walk you through exactly why the bug existed, what class of bug it belongs to, and how you'd spot the next one. It never sighs or checks its watch, and it never makes you feel like question number four was one too many.

That's the edge. A senior gains marginal value from this, because they already know most of what the agent would explain. For you, every explained task compounds, and you have thirty-plus years ahead of you for it to compound over. The earlier you start and the less you already know, the more this pays off, which is exactly the spot a junior is standing in.

But the compounding is not automatic, and this is the hinge of the whole book. An agent will happily do the work *without* explaining anything, and you will happily let it, because the ticket gets closed either way. Let it, and you become the junior the thinkpieces are describing. Most of this book is about how to stop that happening.

Ask for the why, then write it down

The most useful habit I have costs two lines of configuration.

Claude Code (the agent I use; substitute your own) reads a project instruction file at the start of every session. Mine contains, among other things, this:

```
## How I work
- I'm still levelling up. When you explain
 something, show the
 reasoning, not just the answer -- I want to
   understand the why.
```

That's it. A standing instruction, written once, applied to every session forever. The agent stops handing me bare diffs and starts handing me diffs with reasoning attached. (Chapter 2 goes deeper on the instruction file: what belongs in it, what doesn't. For now, know that it exists and that this line is in mine.)

The second half of the habit: when the why is non-obvious, it goes into a note. I keep a `learning/` folder in a plain-markdown vault for exactly this. Here's a real-shaped example from legacy .NET maintenance, a bug class that cost me an afternoon before I understood it:

```
# TIL -- decimal.Parse is culture-dependent
date: 2026-03-11
tags: [dotnet, legacy, gotcha]

Bug: an import job parsed "1.250,50" fine on my
 machine and threw
on the server. My machine runs nl-BE (comma decimal
 separator),
the server runs en-US.

Why: decimal.Parse(string) silently uses
 CultureInfo.CurrentCulture
unless you pass one. Same binary, different machine
 settings,
different behaviour. Nothing in the code hints at
 this.

Rule: parsing numbers from files or the wire ->
 always pass the
culture explicitly. Use the source's actual culture,
 or
InvariantCulture if the format is fixed. Never rely
 on the machine.

Related: [[datetime-utc-gotcha]]
```

Note what that note is *not*. The fix isn't in it; the fix lives in a commit. What it holds is the general rule extracted from the specific incident, plus the why and a link to a related gotcha. Six months from now, when a different import job misbehaves on a different server, I don't re-derive this from scratch. I search my own notes and find my own past self explaining

it to me.

The agent found that bug in minutes. Left alone, it would have fixed it in minutes too, and I'd have learned nothing. The standing instruction got me the explanation, and the note is why I still have it. Without either, the ticket closes just the same and I'm none the wiser.

Do that a few hundred times a year and you're running an apprenticeship at a pace no human mentor could sustain.

The job now: judgment, verification, taste

If the agent does the typing, what exactly do you do? Judgment, verification, taste. They were always the real job; the typing used to obscure that.

Judgment is deciding what should happen. Which of three approaches fits this codebase. Whether this bug is worth fixing now or logging for later. Whether the agent's proposed refactor is simpler or just complicated in a different direction. Agents are quite bad at this, because judgment depends on context they don't have: your team's history, your deployment realities, the fact that module X is scheduled for deletion next quarter. You have that context, or you can get it. Guard it; it's your half of the partnership.

Verification is the boring-sounding one: checking the result yourself. Did the agent do what you meant? Does the diff touch things it shouldn't? This is the skill nobody warned me would become the core of the job, so let me warn you: *reading code you didn't write, sceptically, is now the job.* And not just code.

A story to make it concrete. I have the agent generate chart summaries from my notes (numbers, totals, captions). One caption said, roughly: "€180 + €95 + €60 across the three categories, €310 total." Read it again. It's €335. The agent wrote that caption, presented it with complete confidence, and it was simply wrong; arithmetic a ten-year-old could check. I only caught it because a later verification pass (a fresh agent instance reading everything with clean eyes; more on that pattern in chapter 6) flagged it.

The point isn't that agents are rubbish. It's that agent output always looks right, whether or not it is. Well-written and correct used to arrive together, with an agent they don't. If you don't check the numbers, nobody has. Everything that ships under your name (and it does ship under your name) is yours to verify.

Taste is the slow one: recognising that code is correct but wrong. It works, but it doesn't fit the codebase. The abstraction is clever but nobody will understand it in a year. Taste is exactly what juniors classically lack, and the good news is that reviewing large volumes of agent output builds it faster than writing ever did. You see ten solutions where you'd have written one. You start noticing which ones smell.

What doesn't change

It's worth being explicit about what the agent does *not* take off your plate, because this list is more or less the job description now.

You still have to understand the system. An agent can explain any file, but the map of how the pieces fit (which service owns which data, where the bodies are buried in the legacy code) lives in your head or nowhere. (And

in your notes.)

Responsibility doesn't move either. When the deployment breaks, "the agent did it" is not a sentence you get to say. Not to your team, not to yourself.

Fundamentals still matter, maybe more than before. If you can't follow the code, your review isn't a review; you're scrolling and nodding. And you're now reviewing more code per day than juniors of any previous generation.

Asking good questions stays too. That one, at least, juniors were always naturally good at.

The honest limitations

Some things daily use has taught me that the landing pages don't mention.

The agent will let you learn nothing. This is the failure mode that matters most and I've felt its pull myself. There are days when I'm tired, the ticket is boring, the agent's diff looks fine, and I merge without reading it closely. Every one of those merges is a small loan against my own development, and the interest rate is brutal: do it for a year and you can operate an agent but can't operate without one. At that point you're less a junior developer than a very slow API client. Chapter 10 is entirely about this trap, because it deserves a chapter.

The agent is confidently wrong, regularly. The caption story above wasn't a one-off. It has been wrong about its own arithmetic, about what a regex matches, about whether its script worked. It is never *hesitantly* wrong, which is what makes it dangerous. Plan for this the way you plan for rain in Belgium: not as a possibility.

The agent remembers nothing. Every session starts blank. Whatever it learned yesterday about your codebase, your conventions and your corrections is gone, unless you've built somewhere durable for it to live. This sounds like a footnote. It's the foundation of everything: instruction files, memory files, a notes vault the agent reads and maintains. Chapters 2 through 4 exist because of this one limitation, and the setup they describe is the difference between an agent that resets every day and one that gets better at working with you.

None of this is a reason to avoid agents. It's what the job looks like now, and you plan around it the same way you plan around any other constraint.

Why juniors, specifically

Everything above applies to anyone. Here's why I think it applies to juniors *more*.

You have the least to unlearn. Seniors have a decade of muscle memory built around being the fastest typist in the room, and some of them are finding it hard to put that down. You have no such attachment. The agentic workflow can just *be* your workflow.

The payoff is lopsided too. An explained task is marginal for someone with fifteen years of experience; for a senior the agent mostly saves time. For you it's the whole curriculum, delivered at whatever pace you can absorb.

You were already the question-asker. The core skills of agent work (asking precisely, checking sceptically, admitting what you don't know) are junior instincts. Seniors have to consciously practise the humility part. You get it

for free.

And the experiment is absurdly cheap. My entire setup is a subscription that costs less per month than a decent office chair costs once, plus plain-text files. No committee approval, no platform, no budget line. A junior with curiosity and a few euros a month can run the same setup I'll describe in this book, tonight.

The pessimists are right that the old junior job is dying. They're wrong about what that means. The old job was typing with a learning problem bolted on. What replaces it is judgment work plus a tool that will explain anything, at any hour, and nobody is better placed to take that deal than the person who just arrived and has everything still to learn.

The rest of this book gets into the how.

> ***The principle, tool-agnostic:*** *A junior's real asset is the capacity to compound, and agents turn every task into a lesson for whoever insists on the explanation. The job that remains is judgment, verification, and taste: deciding what should happen, proving that it did, and knowing when correct is still wrong. Whatever agent you use, demand the why with the work, write it down somewhere durable,* ***and never ship anything under your name that you haven't read properly.***

The rest of the book

This is where the free sample ends. If the argument in these two chapters landed, the full book carries it into practice, with everything drawn from real work and the failures left in.

What the other chapters cover:

- Teaching the agent your world: the instruction file it reads before every session
- A second brain in plain markdown that the agent reads and keeps current
- Durable memory for an agent that forgets everything between sessions
- Hooks that automate the dull parts without rotting a week later
- Running several agents at once for audits and reviews
- The team hive-mind: sharing agent knowledge across a small team
- Guardrails that stop an agent doing something you can't undo
- A full week in the life, with real prompts and the mistakes left in

The Agentic Junior, full book as PDF and EPUB:

`https://demeulenaere.gumroad.com/l/xslyom`