Your AI's First Day

A practical guide to getting going

Onboard AI into your dev team
like you'd onboard your best new hire

PETER ROSSI

Your AI’s First Day: A practical guide to getting going
First edition, 2026

Peter Rossi

Feedback welcome at peter@zpventures.com

Contents

Foreword

The world is evolving fast. There is no single way of doing anything with AI, and no known best practice yet. Just guides. This book is my way of showing you what I've tried and what's worked, across a number of teams and projects over the last year. The general direction of travel is: understand, stabilise, build, accelerate. Exactly the same as if you were taking over a development team in a business. The tools and methods will improve over time. Here's how I'm doing it today.

Most of the teams I've worked with are good at their craft and quietly drowning in context. The code is there, but nobody's documented it properly. The patterns exist, but they're in people's heads. The architectural decisions were sensible at the time, but the reasons got lost somewhere between a Slack thread from 2021 and a developer who left the company.

Then AI coding tools arrived, and a lot of people's first reaction was to put them in front of developers and say: go on then, write some code. That sort of works. But it misses the bigger opportunity.

The bigger opportunity is using AI to build a shared understanding of your codebase, your standards, and your patterns. Then using that understanding to make everything downstream better: the reviews, the requirements, the code, the onboarding.

I keep coming back to one analogy: treat AI like a smart new team member. If you onboard them properly, they get

good fast. If you give them no context, they struggle for reasons that have nothing to do with intelligence. **Iteration** is a theme throughout. These tools reward repeated effort, not a one-off setup.

I’ve written this as plainly as I can. No framework, no methodology, no jargon. Just a sequence of things worth doing, explained the way I’d explain them over a coffee.

Let’s get into it.

Introduction: The context problem

Here’s the thing about AI coding tools out of the box. They’re smart, but they don’t know anything about you.

They don’t know that your team prefers composition over inheritance. They don’t know you made a deliberate decision two years ago to keep your service boundaries small. They don’t know you’ve got a naming convention that came out of a painful debugging session and that everyone silently agreed to follow.

So when you point an AI tool at your code and ask it to help, it gives you answers based on what it’s learned from the whole internet. Broad and sometimes useful, but not tuned to your world. Developers end up correcting the AI, explaining the same preferences over and over, accepting suggestions that technically work but don’t fit. I’ve seen developers get frustrated enough to give up on the tools entirely. The bit they’re missing is the shared understanding: putting iterative effort into building the context so the AI stops making the same mistakes. That’s the key.

The fix is context. Give the AI enough context about how your team works, what your codebase looks like, what your standards are, and it starts giving much better answers. The gap between “AI that’s vaguely useful” and “AI that feels like a helpful team member” is mostly a **context gap**. And you can close it quickly.

If you hire someone brilliant and give them zero context about your business, codebase, and conventions, they'll look average. That's not on them. It's on you.

The good news is that gathering context doesn't require a big manual exercise. Historically, you'd have to sit down, gather up all the documentation, read it, write it up, compare it with the existing code, interview the developers, and produce a big onboarding pack. A huge amount of work. Sounds horrific. And I'm pretty confident nobody ever did it well.

So I tried a different approach: **get the AI to do the reading itself and create its own context.** Point it at the codebase. Point it at the commit history. Point it at whatever documentation already exists, even if it's patchy and out of date. Let it figure out what's going on. Then use what it finds to build better context.

That's Phase 1. And it's where most of the value is.

The rest of the book covers what happens after: wiring AI into your review process, using it in your product management workflow, using it to write code, and building a feedback loop so the system improves over time. But Phase 1 is the foundation. If you do nothing else in this book, do Phase 1.

A note on time. I've seen teams work through this in about three weeks when they're focused. Two people spending a day or two on Phase 1, then a week setting up and tuning Phase 2, then gradually building Phase 3 habits from there. Not a six-month programme. Not a weekend project either. Although, if you're a small team on a single project, you can probably do it in a weekend.

One more thing. You can use AI to help you implement everything in this book. Save this ebook into your project, open Claude Code in the terminal, and say "Help me implement this in my repo over the next two weeks." Let it guide you through the steps. That's not cheating. That's the point.

Phase 1: Get AI to understand your world

Phase 1 at a glance

- Use a **directory-attached tool** (Claude Code, Codex, Cursor)
- Point AI at your codebase and let it read everything
- Get it to generate architecture diagrams and documentation
- Interview your developers with AI-generated questions, and use transcription, not notes
- Capture everything in a **CLAUDE.md** and **AGENTS.md**, and build a docs folder with INDEX.md for easy reference

Gathering context and documentation used to be slow and manual. Now it's fast, easy, and genuinely insightful. Here's how.

The goal of Phase 1 is simple: by the end of it, your AI tools should know your codebase well enough to give useful, specific answers instead of generic ones. You do that by letting AI do the reading, then capturing what it learns in a way that's easy to reference.

I'd break it into four parts.

Why directory-attached tools

Directory-attached tools give AI direct access to your repo. The three I use most are **Claude Code** (a terminal-based coding agent from Anthropic), **Codex** (OpenAI's coding agent), and **Cursor** (a fork of VS Code with AI deeply integrated). All three attach directly to your project folder. They can read any file in the repo, run commands, see your git history, write new files, and edit existing ones. When you ask Claude Code to implement something, it reads the relevant parts of your codebase first, checks how similar things are already done, then writes code that fits the actual project, not a generic version of the problem.

Context loading is the other key piece. When you open Claude Code in a project directory, it reads your **CLAUDE.md** automatically. When Codex starts, it reads **AGENTS.md**. Those files contain everything you've documented about how your team works, what patterns to follow, what to avoid. You don't have to re-explain any of it. It's loaded at the start of every session.

This is the foundation everything else in the book builds on. The directory structure in the next section, the input pipeline, the review setup in Phase 2: all of it assumes you're working with a tool that can actually see your files. Pick one and start there.

Your project directory

One of the things that makes this concrete is seeing the actual folder structure. Here's the shape I'd aim for, which we'll build out across the rest of Phase 1.

For most teams, a single repo setup looks like this:

```
my-project/
├── CLAUDE.md                        <- Workflow config: how you use AI tools,
│                                       input pipeline, processing rules
├── docs/
│   ├── INDEX.md                     <- AI-maintained TOC with summaries
│   ├── architecture.md
│   ├── coding-standards.md
│   ├── team-knowledge.md
│   └── integrations.md
├── inputs/
│   ├── transcripts/
│   └── documents/
└── app/                             <- Your code repo (git submodule or clone)
    ├── CLAUDE.md                    <- Code-specific AI context for this repo
    ├── AGENTS.md                    <- OpenAI Codex context
    ├── src/
    ├── tests/
    └── ...
```

Notice there are two CLAUDE.md files, and they serve different purposes. The one at the root is your workflow config: it tells Claude Code how you're using AI across this project, how to handle files that land in inputs/, what the docs folder contains, and what the processing rules are. The one inside the app repo is code context: it tells AI developers how to write code in this specific codebase, what patterns to follow, what to avoid.

Each repo's CLAUDE.md should be self-contained. If a developer just clones the app repo and opens it in Claude Code, the CLAUDE.md in that repo gives them everything they need. The shared docs add depth, but they're not required for someone working on a single repo.

If you're working across multiple repos, the shape scales up naturally:

```
ai-workspace/
├── CLAUDE.md
├── docs/                          <- Shared team/company docs
│   ├── INDEX.md
│   ├── company-standards.md
│   └── architecture-overview.md
├── inputs/
├── api/                           <- Git repo
│   ├── CLAUDE.md
│   └── AGENTS.md
├── frontend/                      <- Git repo
│   ├── CLAUDE.md
│   └── AGENTS.md
└── mobile/                        <- Git repo
    ├── CLAUDE.md
    └── AGENTS.md
```

How you manage the sub-repos is up to you. Some teams use git submodules, some just clone each repo into the workspace folder and manage them separately. Either works. The point is that there's one place where your AI context lives, and each repo has its own context file that works standalone.

Don't try to solve every edge case up front. Get the shape right and adapt from there.

The input pipeline

The inputs/ folder is one of the most useful parts of this setup. It's a drop zone for anything you want the AI to process against your existing docs.

The workflow is user-initiated, not automatic. You drop files into inputs/, open Claude Code in the terminal, and say what you want processed. For example: "I've dropped some new transcripts in the inputs folder, process them and come back to me." Claude Code reads the files alongside docs/, summarises what's new, and proposes updates for review.

The human-in-the-loop part is deliberate. CLAUDE.md tells the agent **how** to process, but it does not auto-trigger anything. You decide when to run it, and you decide what gets written into your docs.

The root CLAUDE.md would include something like this:

> "When I ask you to process files in inputs/, read them alongside the existing docs. Summarise what's new. Identify whether existing docs need updates. Present findings first. Make no document changes until I confirm."

This is particularly useful for meeting transcripts. You run a session with a developer or stakeholder, the transcript lands in inputs/transcripts/, and the AI picks out anything new that should feed into your architecture docs, coding standards, or team knowledge files. No manual summarising. Just a proposed update for you to review and approve.

INDEX.md: your AI's table of contents

Once your docs/ folder has a few files in it, add one more: docs/INDEX.md. This is a file the AI maintains for you, and it changes how efficiently Claude Code navigates your documentation.

The INDEX.md is a brief summary of every document in the docs folder: what each one covers, what kind of questions it answers, when it was last updated. When Claude Code needs to find something, it reads INDEX.md first, then follows the pointer to the right file. It loads the map, then reads only what it needs.

It looks something like this:

```
# docs/INDEX.md

## architecture.md
System architecture overview. Main components, data flow,
service boundaries. Updated 2025-09-14.
Questions answered: How is the system structured? What calls what?
Where do requests flow?

## coding-standards.md
Naming conventions, error handling, async patterns, test structure.
Updated 2025-09-10.
Questions answered: How should I name this? How do we handle errors?
What does a good test look like here?

## team-knowledge.md
Institutional knowledge from developer interviews. Decisions,
reasons, things to be careful about. Updated 2025-09-12.
Questions answered: Why is this built this way? What should I not touch?
Who owns this module?
```

Ask Claude Code to keep this updated whenever it creates or updates a document. It takes about two seconds and it means the whole docs folder stays navigable as it grows.

This is the same idea behind skill systems and plugin indexes: describe what's available, load only what's needed. It keeps context lean and searches fast.

One habit worth building early: ask the AI to summarise key discussions into Markdown files as you go. Write things down, save them in docs/, review later with fresh eyes, and share with the team when needed.

Part 1: Let AI read your codebase

Start by giving an AI tool access to your code and asking it to describe what it sees. Not formally. Just: "I'm going to show you our codebase. Tell me what you find."

What you're looking for:

- What kind of application is this? What does it do?
- What's the main architecture? How are things structured?
- What patterns does it use consistently?
- What's the tech stack?
- Are there any obvious inconsistencies?

You'll get a mix of the accurate and the slightly off. That's fine. The point isn't a perfect answer on the first pass. It's a starting point you can correct and build on.

The goal is to end up with a docs folder full of structured Markdown files that describe your codebase, architecture, testing approach, naming conventions, deployment guidelines, and team norms. AI tools work brilliantly with Markdown, and humans can read and edit it quickly too.

Claude Code works well for this. Cursor does too if you prefer an IDE. Any tool that can hold a codebase in context and read local files will work.

If your codebase is large, you don't have to feed all of it at once. Start with the core: the main application code, the key services, the models. Get an understanding of the centre of gravity first, then expand outward.

Try asking the AI to list the things it's uncertain about. "What parts of this codebase are unclear to you? Where would you want more information?" This is a quick way to identify gaps in your documentation and the places where the code is most opaque. Those are often exactly the places where developers also get confused.

One thing worth doing early: ask the AI to generate a **Mermaid** diagram.

Mermaid is a simple text-based diagramming language. You've probably seen the diagrams it produces, even if you didn't know the name. Boxes and arrows, sequence diagrams, flowcharts. The AI can write it directly, and then you can render it.

Ask something like: "Based on what you've seen, draw me an architecture diagram in Mermaid format. Show the main components and how they connect."

It won't be perfect. But it'll be close enough to be useful, and correcting it is faster than drawing it from scratch. There are plenty of Mermaid editors and visualisers available online. Just search for one, paste the output in, and you've got a visual you can share with the team.

Then do the same for your data model, your service boundaries, your API surface, whatever matters most.

Here's a prompt that's worked for me:

> "You've now read the main application code. I'd like you to draw a Mermaid diagram showing the key modules and how data flows between them. Don't worry about being exhaustive, I want to see the main paths. If there are things you're unsure about, mark them with a comment."

That last line is useful. It forces the AI to flag its uncertainty rather than hide it behind a confident diagram that's partially wrong.

I've found this is often the first time a team has actually seen a proper diagram of their system. That alone tends to generate useful conversations. Save the diagrams. They

become part of your documentation and get referenced in the CLAUDE.md file later.

Part 2: Read the commit history

The commit history is a record of every decision the team has made. Most of it is noise, but the signal is valuable.

Ask the AI to look at the git log and tell you what it finds. Claude Code can run these git commands itself in your repo, so you don't need to shuttle output between tools.

```
git log --oneline --since="1 year ago" | head -200
```

Then ask Claude Code: "What can you tell me about this team's development patterns from these commit messages? What kinds of things are they working on? Are there patterns in how they name things or describe their work?"

You'll get useful observations. Things like: "This team makes a lot of small, incremental changes rather than large feature drops." Or: "There are a lot of commits that mention hotfixes to a specific module, which might be worth investigating." Or: "The commit messages vary a lot in style, which could make it harder to understand the history."

You can also ask more specific questions. "Which parts of the codebase have changed most frequently in the last year?" High churn in a particular module usually means one of three things: it's genuinely complex, it's not designed well, or it's the main area of active development. Worth knowing which.

For a more structured view:

```
git log --since="6 months ago" --name-only --pretty=format: | sort | uniq -c |
sort -rn | head -20
```

That gives you the files changed most often. Ask Claude Code to comment on what it sees. None of this requires special tooling. You're just using the AI to read something that's already there.

Part 3: Interview your developers, with AI's help

This step is one of the highest-leverage parts of the process. **The goal is getting developers talking openly about how things really work**: gotchas, annoyances, trade-offs, decisions they've made, and where reality differs from the docs.

I use a mix of AI-generated questions and open conversation. Don't read a script at people for 30 minutes. Use prepared questions to get started, then follow the thread. Ask them to live demo things while they explain.

After AI has read the codebase and commit history, ask it for interview prompts like: "Based on this codebase, what should I ask developers to understand architecture decisions, team conventions, and hidden risks?" Pick the best ten or so and use them as anchors.

Always use AI transcription. Stay fully present in the conversation. Don't break flow to take notes. Let the transcription tool capture everything, then process the raw transcript files afterwards. **Granola** is good for this, and most meeting tools now have usable transcripts too.

AI can pull out context and nuance you'd otherwise miss, then correlate across multiple interviews. It spots patterns between what different people said, where they agree, and where they conflict.

You can combine that with commit logs and PR comments to build team context: who tends to work in which areas, coding patterns people lean on, and where specific strengths sit.

Then feed the interview transcripts back in and ask for synthesis plus follow-ups: "What do you now understand that wasn't obvious from code alone? What should we clarify next?" That's how you capture the knowledge that usually takes months for a new engineer to absorb.

Part 4: Write your CLAUDE.md and AGENTS.md

This is where everything from the first three parts comes together.

CLAUDE.md is a file you put in your repository that tells AI tools how to work in your codebase. Claude Code reads it automatically when it starts in a project. Cursor reads it if you configure it to. Other tools have equivalent files. The name matters less than the fact that it exists and that it's accurate.

The AI can write this file for you. Based on everything it's learned, ask it to draft a CLAUDE.md that captures the key context a developer would need.

Here's the kind of thing I ask for:

> "Based on everything we've discussed, write a CLAUDE.md file for this repository. It should cover: a brief description of what the application does; the main architecture and how the key components fit together; the tech stack and important dependencies; coding conventions and naming patterns the team uses; things to be careful about or avoid; how to run tests and deploy; and any important context that a developer new to this codebase would need to know."

What comes back won't be final. Review it carefully. Some things will be wrong or missing. Fix those, add what's missing, remove anything inaccurate.

But you're editing a draft rather than writing from scratch. That's a lot faster. And reviewing it is itself useful, because you'll often spot things you'd never thought to write down.

What goes in CLAUDE.md

The official docs are at `https://docs.anthropic.com/en/docs/claude-code/overview` and worth a read. You can generate a starter file by running `/init` inside Claude Code in your project directory. It'll read your codebase and produce a first draft automatically.

Keep it under 300 lines. Ideally shorter. **A dense file that nobody reads carefully is worse than a lean file people actually trust.**

Here's what a good CLAUDE.md tends to cover:

What this is. One paragraph on the application, what it does, and who uses it. An AI that thinks it's working on a consumer mobile app will approach things differently than one that knows it's a B2B admin panel.

Architecture overview. A short description of the main components, how they're structured, and how they fit together. Link to the Mermaid diagram you generated earlier.

Tech stack. Language, frameworks, key libraries, database, infrastructure. The things that shape every technical decision.

Conventions. Naming patterns, error handling approach, how tests are structured, how async code is written, anything the team has strong preferences about. The more specific the better. "We use snake_case for database fields and camelCase for JavaScript" is more useful than "we have consistent naming."

Things to avoid. Often the most valuable section. The things developers know not to do but never wrote down. "Don't modify the legacy payment module without talking to Sarah." "The import order in config files matters and the linter won't catch it if you get it wrong." "Don't use raw SQL queries in this service, we have an ORM wrapper."

How to run and test things. A new developer should be able to read this and know how to get started. Commands for running locally, running tests, deploying.

For larger projects, use **progressive disclosure**. Keep the main CLAUDE.md lean and point to referenced files for detail. You can also add subdirectory CLAUDE.md files for domain-specific context. A CLAUDE.md in src/api/ can cover API-specific conventions without cluttering the root file. Claude Code picks these up automatically when it's working in that directory. This file should evolve over time. More on that in Phase 4.

What goes in AGENTS.md

AGENTS.md is CLAUDE.md's counterpart for OpenAI Codex. By mid-2025 it was present in over 60,000 public repos. It's becoming a cross-tool standard, not just a Codex-specific thing.

The official Codex docs are at `https://developers.openai.com/codex/learn/best-practices/` and worth reading alongside the Claude Code docs.

The hierarchy works like this: a global file at `~/.codex/AGENTS.md` applies to everything on your machine. A repo-level AGENTS.md applies to that repo. An AGENTS.override.md in a subdirectory overrides for that area, which is useful for monorepos.

The content is similar to CLAUDE.md: project structure, coding conventions, testing commands, how to handle errors, what patterns to follow. Both files serve the same purpose for different tools. They can coexist in the same repo without conflict. Codex reads AGENTS.md. Claude Code reads CLAUDE.md. Developers get consistent AI behaviour regardless of which tool they reach for.

Automated PR reviews

Once your CLAUDE.md and AGENTS.md are in place, you can set up automated PR reviews that use them as the policy baseline. I cover this in detail in Phase 2, but it's worth knowing it exists before you finish Phase 1, because it's what makes the context files pay off day to day.

The short version: Claude Code can review every PR automatically via a GitHub Action, and Codex can do the same with its built-in review feature. Both read your context

files and apply your specific standards, not generic ones. The full setup is in the next chapter.

You'll hear people say you need a RAG database or vector store over your codebase. They're not wrong, but it's not where you start. CLAUDE.md gets you moving now. If you outgrow it, look into RAG later. Don't let perfect be the enemy of getting started.

The more you lean in, the more you get out. The context is in place. Now let's put it to work.

Phase 2: Wire AI into your review process

Phase 2 at a glance

- Automate code review on every PR so humans don't have to remember to trigger it
- Use CLAUDE.md as your review policy, not generic rules
- Human reviewers focus on judgment. AI handles the mechanical pass.
- Expect noise at first. Tune the config. It gets better.
- The goal isn't compliance with an algorithm. It's better code.

Here's where the work from Phase 1 starts paying off every day. Not in theory, but in the actual flow of how code moves from a developer's machine into production.

AI code review, and what it's actually good at

AI code review tools read your code and flag things before a human reviewer looks at it. The good ones flag things with context, not just generic warnings.

The most useful thing you can do here is make reviews automatic. They should fire on every PR without anyone having to remember to trigger them. Add your AI review tool as a required check in your CI pipeline, the same way you'd add tests. That way no PR moves forward without a

first-pass review. Here's how to set that up with the two tools I'd recommend.

Claude Code via GitHub Actions. Use the official `anthropics/claude-code-action` GitHub Action. Create a workflow file at `.github/workflows/claude-review.yml` in your repo. It triggers on PR opened and synchronize, reads the diff, uses your CLAUDE.md as the review policy, and posts inline comments plus a summary directly on the PR.

```yaml
name: Claude Code Review
on:
  pull_request:
    types: [opened, synchronize]
jobs:
  review:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v4
    - uses: anthropics/claude-code-action@v1
      with:
        anthropic_api_key: ${{ secrets.ANTHROPIC_API_KEY }}
        prompt: "Review this PR for bugs, security issues, and code quality
against our CLAUDE.md standards. Be specific and actionable."
```

Add your `ANTHROPIC_API_KEY` to your repo secrets and you're done. Every PR gets a review. No one has to remember to ask for it. If the YAML looks unfamiliar, just ask Claude Code to set it up for you. Tell it what you want and it'll step you through it for your specific repo.

Codex. Enable automatic reviews in Codex settings for your repo. Every PR gets reviewed automatically from that point on. You can also trigger a review on demand by commenting `\@codex review` on any PR. Codex analyses the changes for correctness, performance, security, and maintainability, then gives a verdict with a confidence score.

Both tools use your CLAUDE.md and AGENTS.md as the policy baseline. The reviews fire automatically. No human has to remember to request them.

Point the tool at your CLAUDE.md. Most of these tools let you add a custom instruction file or a repository-level configuration. That configuration should tell the AI what your standards are, what patterns to expect, and what a good PR looks like for your specific codebase. Without that, you'll get generic feedback. With it, you'll get feedback that references your actual conventions.

Be clear about what you're asking the tool to check. I tend to ask AI reviewers to focus on correctness, security, and consistency with existing patterns. Style enforcement belongs in a linter, not a reviewer.

Treat AI review comments as a first pass, not a verdict. Some comments will be wrong. Some will be technically correct but not relevant. Developers should feel able to say "yes, I know, but here's why I did it this way" and override an AI comment. The AI is not always right. The goal is better code, not compliance with an algorithm.

In practice, human reviewers spend less time on the boring stuff. The AI catches the missing null checks, the unvalidated inputs, the function that handles nine of ten cases but not the tenth. Humans focus on the things that need human judgment: architectural fit, product logic, whether this is solving the right problem.

This makes reviews faster and more thorough. Not because the AI is smarter than the developers, but because it reads everything and doesn't get tired. It doesn't skim a large diff because it's the end of the day. It doesn't miss something because it's reviewed eight PRs already this week.

AI review tools can be noisy at first. They flag a lot. That's okay. After a few weeks, you'll notice patterns in what's useful and what isn't, and you can tune the configuration accordingly. Expect to iterate on it, just like everything else in this process.

Give your developers a session where you walk through what the tool does and why. If it shows up unexpectedly and starts commenting on their code, the reaction is often defensive. If they understand what it is, what it's checking for, and how it helps them, adoption is much smoother.

AI review tools don't replace human code review. Human review still matters. What changes is what human review focuses on. The mechanical stuff gets filtered out. The judgment calls still need a human.

Once the review process is running, the question becomes: what else can AI actually produce? Phase 3 is where things get interesting.

Phase 3: AI starts contributing code

Phase 3 at a glance

- Good context in means good code out. Phases 1 and 2 are what make this work.
- Be specific: reference existing patterns, ask for tests alongside code
- Ship only code you understand and can defend
- The product management loop is often where the biggest time savings show up
- AI quality is mostly a context function. Rich context in, strong output out.

This is the part everyone talks about first. I've put it third deliberately, because it works much better when you've done Phases 1 and 2 first.

If an AI tool knows your codebase, your patterns, and your standards, and if the review process is there to catch mistakes, then code generation becomes a lot more useful. The AI produces code that fits. The review process catches what it gets wrong.

Making the most of your tools

Claude Code is the one I reach for most on substantial tasks. You give it a task, it reads your CLAUDE.md, explores the relevant parts of your codebase, writes the code, and

runs your tests to check its own work. Tasks that would take a developer most of a day often produce a solid first draft in an hour or two. It integrates naturally with Git, so you can ask it to work on a branch, commit as it goes, and prepare a PR when it's done.

Codex is fast and precise on well-scoped work. Where Claude Code tends to excel at larger, more exploratory tasks, Codex is the one I reach for when the scope is clear and bounded. Both tools support project context and both respond well to a good CLAUDE.md.

Cursor is my day-to-day IDE. You can ask it to write code in the context of your whole codebase, not just the current file. It reads your CLAUDE.md, understands existing patterns, and generates code that fits. Composer mode is particularly useful for tasks that span multiple files.

GitHub Copilot is the one most people have already tried. Good at autocomplete and inline suggestions, and lower friction to adopt because it slots into existing habits. Less capable for complex multi-file work, but a solid starting point.

A few practical notes:

Be specific about what you want. "Write a function that does X" works better than "help me with the database layer." The more context you give, the better the output.

Reference existing patterns. "Write this in the same style as the UserService in services/user.ts" gets you code that fits much better than a generic request.

Ask it to write tests alongside the implementation. "Write the implementation and the tests for it" produces better results than asking for code first and tests later.

Don't just accept the first output. Ask follow-up questions. "Is there an edge case you haven't handled?" "How would this perform with a large dataset?" The AI will often find issues in its own code if you ask.

Testing as a hard standard

This isn't optional. If your AI tools are writing code without tests, you're building on sand.

Make testing a non-negotiable part of your development standards. Add it explicitly to your CLAUDE.md:

> "Always write tests for new code. Run the full test suite before committing. If tests fail, fix them before proceeding."

Claude Code and Codex can both run tests as part of their workflow. They write the code, write the tests, run them, and only then create the commit or PR. That's the sequence you want. Not "write code, then write tests later." Write code, write tests, run them, ship.

AI-generated code can be confidently wrong. Tests are how you catch it. They're also how the AI catches itself. When Claude Code runs the test suite and something fails, it investigates and fixes before handing the work to you.

Build the expectation in from the start. When a developer or an AI tool hands over a PR, tests should already be passing.

The product management loop

Most of what I've described so far has been engineering-led. But some of the biggest time savings come from earlier in the process, before a developer has touched a keyboard.

AI is good at the product management part of software development. Writing requirements, breaking features into tickets, structuring work so that implementation can follow cleanly.

It starts with a rough idea. "We need to let users export their data." Take that to Claude Code and ask it to write a proper product requirements document. Give it context: who the users are, what the product already does, what the edge cases might be. It'll think through the user journey, the error states, the permissions model, the success criteria.

The PRD it produces won't be final. This part should be conversational. Talk it through with the AI. Push back on assumptions. Add business context. Explain why something won't work in your environment. Ask for alternatives. You iterate live in the same thread until the PRD reflects reality.

Then ask it to break that PRD into implementation tickets: specific, scoped work items with clear dependencies and enough context that a developer can pick one up and ship.

The tickets need to end up somewhere real. If your team uses Linear, have the AI draft tickets in Linear's format. If you use Jira, same idea. Tickets that land in your project management tool are tickets that actually get done. Ones that stay in a document tend to drift.

Some teams are going further: using **MCP** connectors to link AI tools directly to Jira or Linear. MCP (Model Context Protocol) is an open standard that lets AI tools connect to external systems. There are MCP connectors for Jira, Linear, GitHub, and others. Claude Code supports MCP natively.

With the right setup, you can have Claude Code read a Jira ticket, understand the requirements, implement the code, run the tests, and flag the ticket as ready for review. It can also create sub-tasks automatically when a piece of work is larger than expected, or update ticket descriptions as it learns more about what's needed.

The setup is simpler than it sounds, and you can get the AI to help. You can be up and running in a few minutes. Even a partial version is valuable. AI that can read your existing backlog gives better suggestions, and AI that can write tickets from requirements saves product and engineering managers a lot of time.

The full pipeline, from rough idea to implementation-ready tickets, can happen in a couple of hours for a reasonably sized feature. That's a meaningful compression of what used to be a multi-day process of meetings, documents, and back-and-forth. Vague requirements produce vague code. This is as true for AI as it is for humans.

Someone still needs to review the output and decide if it's right. But it gets you to a first draft faster, and developer time goes into reviewing and refining rather than writing boilerplate.

What AI handles well

With enough context, AI handles most coding tasks well. I’ve seen it produce a well-structured service layer for a feature I could barely spec in plain English. I’ve also seen it miss an obvious edge case and present the code with complete confidence. The difference was context, not capability. Same as a strong new hire: performance tracks what you give them to work with.

When output is weak, it’s usually a **context gap**, not a capability gap. Fill in product intent, domain rules, codebase conventions, and recent decisions, and quality jumps.

Experienced developers still add massive value through deep business and domain knowledge. They know why things are the way they are, where the edge cases live, and which trade-offs are deliberate.

AI can still be confidently wrong, so keep one hard rule: **never ship AI-generated code you can’t explain.** If you can’t explain it, don’t ship it.

Phase 4: The feedback loop

Phase 4 at a glance

- The first version of anything here won't be perfect. That's fine. Iterate.
- Feed production signals back in: error logs, support tickets, usage patterns
- Update CLAUDE.md like it's code: PRs, changelogs, reviews
- Everyone on the team can contribute context, not just developers
- The compounding effect is real. Month three looks very different from month one.

AI tools don't improve on their own. They improve because someone keeps feeding them better information.

Iteration: the backbone of this whole approach

The CLAUDE.md you wrote in Phase 1 was a starting point. It won't have been right in every detail. Some things will have been missing. Some things will have been slightly off. That's fine. The second version will be better. The third version better still. After a few iterations, you'll have something that accurately reflects how your team works.

If you're the sort of person who gets frustrated when something isn't right first time, this is worth sitting with. These tools reward persistence. **The more you lean in,**

the more you get out. Each time you feed something back in, you get a better result next time. The investment is real and the returns compound.

Iteration applies to everything: your review prompts, your testing standards, your PR workflow configuration, your interview questions. When the automated review keeps flagging the same pattern, improve your prompt. When tests keep needing the same kind of fix, update your testing standards. Every part of the system has a feedback loop.

Treat your CLAUDE.md and AGENTS.md like code. Put changes through a PR. Review them. Keep a changelog. They're living policy documents, not one-off setup files. If a change to CLAUDE.md breaks the way AI behaves on your project, you want to be able to trace it back and understand why.

What's worth feeding back constantly:

- Coding styles and naming conventions your developers actually use, as they evolve
- Testing frameworks and patterns you've settled on
- Architectural decisions and the reasons behind them
- Integrations and dependencies that matter
- Anything that keeps coming up in code reviews as a repeated issue

You only need to tell it once. Once it's in the CLAUDE.md, it stays in. The AI doesn't forget. Every future session starts with that context already loaded.

Feeding production back in

Once your software is running in production, there are new sources of information that didn't exist before. Feed those back into your AI context to improve quality upstream.

Production data. What are users actually doing? Which features get used, which get ignored, where do people drop off? Take anonymised usage patterns, feed them into your AI context, and ask questions. "Users are abandoning this checkout flow at step 3. Based on the code for that step, what might be causing friction?"

Support tickets. A direct line to what's confusing or broken. Feed a batch of recent tickets to your AI tool and ask: "What patterns do you see? Are there recurring issues that might point to a systematic problem in the codebase?" I've done this and it's surfaced things I hadn't noticed: a cluster of tickets all pointing to the same confused user journey, or a recurring error that looked different in each ticket but had the same root cause.

Error logs. Give Claude Code a sample of recent errors and ask it to categorise them. "Are these one-offs or are there patterns? Are any of them clearly related?" A fast way to spot things slipping through your monitoring.

Code review feedback. The comments from your PR reviews, human and AI combined, are a signal about where the codebase is generating confusion. **Patterns in review comments often point to gaps in the CLAUDE.md.** "We keep seeing comments about how async errors are handled. Let's add a clearer note to the context file about our approach."

None of this is automatic. It requires someone to sit down periodically and do the analysis. But it doesn’t have to be a big exercise. An hour a month of feeding new information back into the system compounds over time.

The AI’s understanding of your codebase in month three should be meaningfully better than it was in month one, not because the AI changed, but because you gave it better context.

Making it stick

The hardest part of this isn't the technical setup. The hardest part is getting your team to actually use the tools in a consistent way, without either ignoring them or trusting them blindly.

Start with the skeptics

Bring the most skeptical developers in early, not last. Ask them to help evaluate the tools. Ask for their honest feedback on whether the AI review comments are useful, whether the code suggestions fit the codebase, whether the CLAUDE.md captures things accurately.

Two things tend to happen. Either they find it useful and become advocates, or their feedback reveals real gaps you can fix before the wider rollout. Either outcome is good.

The worst thing you can do is mandate a tool without involving the people who have to use it. You'll get compliance without buy-in, which means people will technically use the tools but dismiss their outputs.

Don't promise speed immediately

There's a ramp-up period where using AI tools actually slows people down a bit, because they're learning to use them well. The payoff comes later, usually a few weeks in, when they've got the hang of how to prompt well and what the tools are actually good at.

Set expectations accurately. "This will probably feel awkward for the first few weeks, and then it'll start to click." If you overpromise and the tools feel slow or noisy at first, people will write them off. If you set realistic expectations, they'll stick with it through the initial friction.

The developers who get the most out of these tools are the ones who treated the first few weeks as a learning period, tried different approaches, and adjusted based on what worked.

Make the wins visible

When the AI review catches a real bug before it ships, point it out. "This one would have cost us two hours to debug in production." When someone uses an AI tool to implement something faster than usual, mention it. When the CLAUDE.md gets updated because of something that came up in a review, explain why.

Concrete wins build the case better than any pitch. People believe what they see working, not what they're told should work.

Keep the human judgment visible

AI is effectively a co-author now. That's just reality.

Experienced developers add value through review because they carry business context, domain knowledge, and the history of why the system looks the way it does. They review what ships and own it. Same as they always have.

That's the discipline to protect. Keep humans engaged in the judgment calls.

What’s next

I don’t have a confident view of where this all goes in three years. The pace of change is fast enough that confident predictions age badly within months.

What I do think is directionally true:

The tools will get better at holding more context. The constraint right now is that AI tools can only see so much of your codebase at once. That constraint is loosening, and as it does, the quality of suggestions will improve.

The skills that will matter most for developers won’t be typing speed or remembering syntax. They’ll be: understanding what the code is supposed to do, knowing how to evaluate whether it does it, and being able to spot problems that aren’t obvious on a first read. Those skills matter more, not less, when AI is writing more of the code.

The teams that will do well are the ones that have been deliberate about this. That have set up context, built review processes, and maintained human judgment throughout. The teams that have just pointed AI at their codebase and hoped for the best will probably find the results inconsistent and eventually frustrating.

I think the setup I’ve described in this book is a reasonable starting point. It’s not the only way to do it. It’s the way that made sense for the teams I’ve worked with, on the scale I’ve worked at.

If you try it and find things that work better differently, I’d genuinely like to hear about it. The whole field is learning in

public right now, and the best way to make progress is to share what’s working.

You can find me on LinkedIn, search for Peter Rossi, or reach out at peter@zpventures.com.

Further ideas

Once you've got the basics working and you're confident with the workflow, there's more you can build on top of it. None of these are essential from day one. They're natural extensions once the foundations are in place, and the right one to tackle next will depend on where your team is feeling the most friction.

Automated error triage. A scheduled job that reads your error logs, categorises issues, writes tickets to resolve them, and then passes the tickets to Claude Code or Codex to create the fix and issue a pull request. Fully automated bug detection to fix to PR. You wake up in the morning and the obvious stuff is already handled.

User behaviour analytics agent. An agent that analyses your web and access logs, identifies where users stall, drop off, or get stuck. Looks at session patterns, time-of-day usage, repeated retries. Surfaces insights about how people actually use your product versus how you think they do.

Live product shaping with stakeholders. Use a tool like Lovable.dev or Replit in a meeting with stakeholders when speccing new features. Build it live while they watch. They spot edge cases immediately when they can see and feel the tool. Transcribe the meeting with Granola. Then take the working prototype source code plus the raw transcript and give them to Claude Code to work up a proper PRD and spec based on your documented ways of working.

Migration assistant. An agent that scans legacy modules and proposes stepwise refactor plans with risk scoring.

Points out what's safe to move first and what to leave alone for now.

Dependency guardian. Reviews dependency updates weekly, opens safe PRs with test evidence. Keeps your dependencies current without someone having to manually track it.

Docs drift detector. Compares code changes versus documentation and flags when docs need updating. Catches the slow decay that happens when documentation falls behind reality.

Incident companion. During an outage, summarises logs and timeline, drafts a first-pass postmortem. Means you're not staring at a blank doc at 2am while also trying to fix the problem.

Onboarding copilot. New developers ask questions grounded in CLAUDE.md, recent PRs, and codebase context. Dramatically cuts onboarding time.

Product analytics copilot. By this stage, AI already understands your code, your users, your features, and your data schema. You can ask for quick reports or SQL-style queries to analyse behaviour, uptake, and feature usage, then feed those findings straight back into product decisions.

None of these need to be perfect. A rough version of any of them that runs reliably is better than a polished version that never gets built.

Tool reference

A quick reference of the tools mentioned. This isn't a comprehensive market survey, and things change fast. Prices, features, and what tools even exist will look different by the time you read this. Do your own due diligence before committing.

Tool	Category	What it does	Notes
Claude Code (Anthropic)	CLI agent	Terminal-based agent for codebase exploration, implementation, and PR review	Reads CLAUDE.md natively; integrates with Git; supports MCP connectors
claude-code-action	GitHub Action	Automated PR review via GitHub Actions	Official Anthropic action; posts inline comments; uses CLAUDE.md as policy
OpenAI Codex	AI coding agent	Code generation and review; works well for well-scoped implementation tasks	Supports automatic PR reviews; trigger on-demand with @codex review
Cursor	IDE	AI-native VS Code fork with codebase-aware suggestions	Good day-to-day coding IDE; Composer mode spans multiple files
GitHub Copilot	IDE plugin	Autocomplete and inline suggestions in VS Code	Lower friction to adopt; less capable for complex multi-file work
Granola	Transcription	AI meeting notes and transcription	Good for in-person and hybrid calls; produces raw transcripts
Linear	Project management	Ticket tracking and project management	Works well as a destination for AI-generated tickets

Jira	Project management	Ticket tracking at scale; MCP connectors available	MCP integration lets AI tools read and update tickets directly
Lovable.dev	Prototyping	AI-assisted rapid app prototyping	Useful for live product shaping sessions with stakeholders
Mermaid	Diagramming	Text-based diagram syntax rendered as visuals	Not AI itself. AI writes the Mermaid; you render it
Mermaid Live Editor	Rendering	Browser-based Mermaid editor	Good for quick diagrams with no install needed

The tools I use most are Claude Code for Phases 1 and 3, Codex for reviews and implementation checks, and Cursor as my day-to-day IDE. But that's my setup, not a prescription.

About the author

Peter Rossi is a technology leader who’s spent his career building and scaling software platforms and teams. He started out running trackside IT in Formula 1, co-founded and sold a SaaS business, and went on to lead technology across a PE-backed group, integrating over twenty acquisitions and aligning teams across multiple countries. He also advises PE and VC investors on technology due diligence during M&A.

More recently, he’s been working with development teams in PE-backed businesses, helping them move faster without adding process overhead. That’s where AI has really started to accelerate things. He found it genuinely useful, started documenting what worked, and this book grew out of those notes.

He’s learning at the same pace as the rest of the industry. This book is written by someone doing the work, not watching from the sidelines.

You can find him on LinkedIn or reach him at peter@zpventures.com.

First published 2026. Feedback welcome.