ZEN.SOLUTIONS

The AI Practitioner's *Field Manual*

Build with intention. Scale with clarity.

Hans Havlik

Founder, Zen Solutions
Writer, Developer, AI/ML Systems Architect

zen-solutions.dev

Version 3.0 / August 2026 / Free to share

CONTENTS

What is in here

PREFACE

Why this exists

I have spent years helping teams design, build, and deploy enterprise agentic AI systems: multi-agent orchestration, recommendation and prediction pipelines, and the operational scaffolding that keeps them running inside real client environments. The through-line of that work has been plain. Automate the large majority of the entry and mid-level operational load so the people on the team move up to the problems that actually need judgment. I also do this at home. My own knowledge base, my own agents, my own pipelines running in the background while I get on with the rest of my life.

I learned this the way working practitioners learn anything. Years of doing it, building it, getting things wrong, and writing down what held up. This is the reference I wish someone had handed me three years ago. The principles here are the ones I have tested in environments where being wrong has consequences.

What this guide is: a calm, grounded, honest walk through the AI landscape in 2026. What is actually possible right now, what is overhyped, what is dangerous, how to get a real workbench set up, and a 90 day path to becoming a practitioner.

What this guide is not: a prompt dump. It is not a list of ten ChatGPT hacks or copy-paste templates that promise to change your business overnight. The shortcut economy has saturated this space. You are reading this because you suspect there is something deeper, and there is.

What is new in version 3.0

The first two versions told you what to think about. Readers came back with the same question: fine, but what do I actually install, and how do I get an agent working on a real project without breaking something. So this version adds two full parts on the workbench. How to set up GitHub, Claude Code, Cursor or VS Code, and Codex. How to connect an agent to a repository so it can branch, commit, and open a pull request you review before anything lands. How to give an agent written procedures instead of hoping it improvises well, using my open source skills library as the worked example.

I have also updated the evidence throughout. Some of what was true in June is no longer true in August, and I would rather say so than quietly leave it.

This is free. There is no upsell at the end. If it lands, the best thing you can do is go to zen-solutions.dev and subscribe to the work. The Foundations series and the 26 week Field Manual series are where this goes deeper, week by week.

A NOTE ON THE FIELD

The AI practitioner space is still small. Where another writer or builder has done work that shaped mine, I name them. Dan Koe writes clearly about what one person can build with modern tools, and that work informs how I think about a one-person practice. Anthropic's engineering team writes publicly about how Claude Code and context engineering actually work, and that writing is worth reading. Most of what follows is from my own years in the work.

PART 1

Where we actually are

Most people are using a 2026 AI model like it is a 2023 AI model. They open a chat window, type one line, get a mediocre response, and conclude that AI is overrated. They are not wrong about their experience. They are wrong about the cause.

The frontier has moved on three axes since 2023, and almost nobody has updated their mental model.

Capability

Today's frontier models reason across long, complex problems that would have broken a 2023 model in the first paragraph. They hold multi-step problems in working memory, plan, revise, and check their own work when you ask them to. The systems from the major labs now run autonomously for hours on a single task.

One thing changed in 2026 that you should know before you read another benchmark chart. The standard software engineering benchmark stopped being trustworthy. In February, OpenAI announced it would no longer report scores on it, citing memorization and contaminated training data. Weeks later, an adversarial re-test found that roughly a fifth of the problems marked as solved were not actually solved: the patches passed only because the tests checking them were weak. Strengthening those tests dropped the leading agent from 78.8 percent to 62.2 percent, and from first place to fifth. A separate research group found that training a model on a benchmark makes it better at that benchmark and frequently worse at the next three benchmarks over.

The ceiling really has lifted by an order of magnitude. The published numbers describing that ceiling are now much softer than they look. Judge tools on whether they work on your problem, because in 2026 that is the only test left standing.

Context

Context windows went from a few thousand tokens to hundreds of thousands, and in the largest models past a million. That sounds like the end of the context problem. It is not. In 2026 the research settled a question practitioners had felt for a while: every frontier model degrades as its context fills, at every length tested. The industry named it context rot. A bigger window does not remove the problem. It moves it. The model still answers, it just answers worse, and it does so silently.

A working figure to hold: plan around 40 to 50 percent of the advertised window as the part you can rely on for precise retrieval and reasoning. The underlying effect, first measured in 2023 and replicated across six model families, is that accuracy is highest at the beginning and end of a long input and sags in the middle.

The skill that matters is no longer feeding the model everything you have. It is curating the smallest set of high signal material the task actually needs, and evicting the rest. That discipline has a name now, context engineering, and it has quietly become the load-bearing skill of the whole field. Gartner named it the breakout AI capability of 2026.

Agency

The model is no longer just a text generator. It reads and writes files, runs code, navigates the web, calls APIs, and coordinates other models. The same intelligence that drafts an email can spend four hours researching a topic, writing the report, and leaving it in your inbox while you sleep.

That capability is real. So is its failure mode, and most of the market is not honest about the gap. In controlled benchmarks of real office work, the best agent systems complete only about a third of the tasks end to end. The demo runs on clean inputs and a cooperative path. Production does not. Holding both facts at once, the genuine power and the genuine fragility, is the beginning of practitioner judgment.

What the year settled

In March 2026 the field finally named the thing that had been true for a while. **Agent equals model plus harness.** If you are not the model, you are the harness. The harness is every piece of code, configuration, and execution logic that is not the model itself: what goes into context, which tools exist, how results are checked, when to stop.

The clearest evidence is a team that changed only the infrastructure around a fixed model, weights untouched, and moved from outside the top thirty to fifth place on a major agent benchmark. Same intelligence. Different harness. Twenty five places.

This is why two products running the identical model can feel completely different, and it is why the leverage available to you personally is much larger than the leverage available to the labs. You cannot make the model smarter. You can absolutely build a better harness around it, and that is where the next two years of advantage sit.

Stop using AI. Start building your harness.

The model is the engine. The harness is everything around it: context, tools, structure, discipline, governance, and judgment. The harness is what decides whether your AI feels like a slot machine or a colleague. This guide is about the harness.

PART 2

The four practitioner levels

A practitioner moves through four levels. Each one is a real change in what you can do, what limits you, and what kind of leverage compounds. Most users never leave Level 1. Most enthusiastic builders plateau at Level 3. Level 4 is where the operating leverage lives, and it is the level most of the discourse pretends does not exist.

LVL	IDENTITY	WHAT YOU DO	WHAT UNLOCKS
1	**User**	Chat	Ideation, drafting, research
2	**Builder**	Harness	Context engineering, projects, skills
3	**Operator**	Systems	Agents, pipelines, automations
4	**Strategist**	Doctrine	Restraint, security, leadership

Level 1: the user

You use a chat interface for ideation, drafting, summarizing, and basic research. This is where most users live. Your output quality is directly proportional to the quality of your prompts, and you spend most of your time fighting generic results.

What limits you here: the assumption that AI is a thing you talk to, not a thing you build with. You are missing the structural moves that compound.

The single highest-leverage upgrade: stop writing prompts, start writing context. A 200 word context block (here is who I am, here is what we are working on, here is the standard I want, here is the audience) at the start of a project beats any prompt hack you will ever learn.

Level 2: the builder

You have stopped treating each conversation as fresh. You use Projects in the chat app. You write context files for your serious work. You install a terminal agent and start working from the command line or your editor. You connect your AI to your filesystem, your notes, your calendar. You are building a personal harness around the model.

What limits you here: you are still operating one session at a time. Your AI helps you when you are at the keyboard. When you stop, it stops.

The single highest-leverage upgrade: treat your context as a versioned artifact, not as something you re-type each session. Context files, system prompts, skills, project READMEs. Commit them to git. Improve them the way you would improve any other piece of code. Part 8 of this guide is the practical version of this step.

Level 3: the operator

You have moved from sessions to pipelines. You run agents that spend hours doing research while you do something else. You have a content pipeline, a research pipeline, a capture pipeline. You have a personal knowledge base your agents can read. You think in terms of inputs, processes, and outputs, not turns of conversation.

What limits you here: complexity. Every new agent and pipeline is another thing to maintain. You will eventually have an over-engineered, fragile system you do not trust. This is where most enthusiastic operators wash out.

The single highest-leverage upgrade: delete things. The mature operator runs a small number of agents and pipelines extremely well, with strong observability and clear failure modes. Most of what you built in your first six months should be retired or simplified by month nine.

Level 4: the strategist

At Level 4, the question is no longer how do I use AI, but what does this organization or this life look like with AI integrated correctly, including the decision not to use AI in certain places. Most of the public discourse stops at Level 3. This is the level where the actual value gets made or lost.

You lead agents the way you would lead a team of junior staff. You hold a doctrine. You enforce security boundaries because you understand the threat surface. You know which problems deserve AI, which deserve traditional automation, and which deserve a human, full stop. You can argue against AI being deployed somewhere it does not belong, and that is more valuable than another agent.

The currency at Level 4 is judgment.
The market is flooded with operators. Strategists are rare.

PART 3

Two techniques that compound

Most users treat the model as an answer machine. They ask a question, get a response, and judge the result. This is the lowest-leverage use of frontier AI. The practitioner uses the model differently. The two techniques below are the ones I rely on every day. Both share a principle: do not push at the model, invite the model into the work.

Analogical translation

When you are learning a new domain, the largest cognitive cost is not the concepts. It is the wall of foreign vocabulary that arrives before the concepts do. Every unfamiliar term is one more thing your brain has to hold while it is still trying to grasp the underlying idea. By the third unfamiliar word in a paragraph, you have stopped learning and started surviving.

The model can dissolve this barrier. Tell it which domain you already know well, and ask it to translate the new domain into analogies from your existing expertise. Then, after the analogy lands, ask it to introduce the actual vocabulary tied to the analogy you now understand.

Consider a nurse learning to be a software developer. Networking, databases, APIs, system architecture, these all sound abstract. They are not. They are the same kinds of patterns the nurse already understands from human anatomy, hospital workflows, and clinical handoffs. A database is a chart. An API is the handoff at change of shift. A network outage is a code blue without staff. A microservice architecture is a hospital with specialized departments that have to communicate cleanly or patient care breaks down.

The concept lands before the jargon does. When the jargon arrives, it attaches to a concept that is already there. The learning sticks because it has somewhere to live.

I have used this on every domain crossover I have made: medic to software developer, developer to AI solutions architect, individual contributor to team lead. The model is exceptionally good at producing these translations on demand. Most users never ask for them because they assume the model is for getting answers. The model is also a translator between domains, and that is one of its highest-leverage uses.

> **FIELD-TESTED RULE**
>
> Anchor the unfamiliar to the familiar, then let the unfamiliar earn its own ground. A working prompt: *I am trying to understand a new concept. I have deep expertise in my own domain. Before you introduce any jargon, translate the concept into analogies and mental models from that domain. Then introduce the real terminology, and tie each new term back to the analogy.*

Reverse prompting

Pull, do not push. Most users push instructions at the model and get generic results. I invert the flow: state the goal, then ask the model to ask me questions until it has the context to produce what I actually want. I call this reverse prompting, and it has been part of how I work for years.

The shape, in its simplest form: *I need your help with a task. Before you help me, ask me three to five questions to make sure you have the context you need. Ask one at a time.*

That is the floor. The version I actually use loads stable context first, pulls only the marginal context this task needs, and forces a critique at the end: *I need your help with a task. Here is what you should know about me and this work, read it first. Then ask me three to five questions to fill in only what is missing for this specific task, one at a time. Once I have answered, give me your best work, then critique it as if a senior practitioner in this field were reviewing it.*

The structure does three things at once. It loads stable context, it pulls only the marginal context needed, and it forces self-critique. Most practitioners stop after the first move. The second and third are where the leverage compounds.

Both techniques come from the same instinct: stop trying to extract output from the model, and start working with it. The model is not a vending machine. It is a collaborator. Treat it accordingly and the work changes.

PART 4

The five disciplines

If the four levels are where you are, the five disciplines are how you operate at any level. These are not skills you graduate from. They are practices you return to.

1. Context discipline

Your AI is only as good as the context you give it, and in 2026 that sentence needs a second half: only as good as the context you give it and the context you keep out. Context rot is real. Past a certain fill, every model starts favoring the wrong tokens and losing the thread, even when the answer is still technically in the window. So the senior practitioner does two jobs at once. Load the right material, and actively remove the material that has stopped earning its place. Curate what goes in. Evict what is stale. Compress long history into summaries. Delegate heavy search to a separate agent that hands back only its findings, not its entire trail.

In practice: every project has a context file, every recurring workflow has a system prompt that has been refined and committed to git, and a session that has run long and started repeating itself gets cleared and reloaded rather than pushed harder. You version your context like code, and you treat the context window as a workspace to be kept clean, not a bucket to be filled.

> FIELD-TESTED RULE
>
> If you find yourself writing the same instruction more than twice, move it into context and make it permanent. And when a long session starts repeating itself or dropping your constraints, do not push harder. Clear it and reload only what the next step needs.

2. Restraint discipline

Knowing when not to use AI is the rarest discipline and the most valuable. Most of the industry is AI-washing every problem it sees. Some problems do not want AI. They want a database query, a deterministic script, a phone call to the right person, or nothing at all.

In practice: before deploying AI anywhere, ask what is the simplest thing that would work. Often it is a regex. Sometimes it is a meeting. AI is the right tool when the problem requires reasoning across unstructured inputs, judgment, or generation. It is the wrong tool when the problem is solved by exact-match logic, when the consequences of failure are severe and reversibility is low, or when a human is already doing this in thirty seconds.

> FIELD-TESTED RULE
>
> If you can write the deterministic version in less than an hour, write the deterministic version. AI is for the problems where deterministic logic does not fit.

3. Verification discipline

The model can be wrong. The model can also be confidently, fluently, hallucinated-citation wrong. The practitioner builds verification into the workflow, not as an afterthought. There is a real cost to this, a verification tax, and it never goes down if the system does not learn from its corrections. Designing that tax down is part of the work.

How wrong depends heavily on framing, which is worth internalizing. A 2026 study measured hallucination rates across twenty six leading models and found they ranged from twenty two percent to ninety four percent depending on how the question was posed. One leading model's accuracy dropped from 98.2 percent to 64.4 percent purely from a reframing of the prompt. The model does not know when it has crossed from knowing into guessing, and neither does the writing.

In practice: the model produces, then critiques its own work in a separate pass. Important claims get a skeptical-reviewer check. Cited facts get a web fetch. Code gets executed. Anything headed for an external destination crosses a human review gate.

> **FIELD-TESTED RULE**
>
> Never let a model's output reach a high-stakes destination (a client, a public post, a deployed system) without at least one verification pass. Speed without verification is just a fast way to be wrong.

4. Security discipline

The agentic threat surface is real and growing. If you give an agent access to your email, your filesystem, your credentials, or your bank, you have made it a target. Almost nobody covers this honestly because it is not what sells.

In practice: least privilege for every agent. Read-only where possible. Sandboxed environments for autonomous work. Audit logging on every action. Prompt-injection awareness, because an agent reading a webpage can be hijacked by content in that webpage, and an agent reading your email can be instructed by an email. Treat all untrusted content as potentially adversarial.

> **FIELD-TESTED RULE**
>
> For any new agent capability, ask: if this agent were fully compromised, what is the worst it could do? Design the answer to be tolerable.

5. Leadership discipline

You lead an AI agent the way you lead a junior team member. You set context, you set standards, you give feedback, you correct course. You do not assume the agent will figure it out. You write down what good looks like. You do an after-action review.

In practice: every recurring agent has a job description, a system prompt. Every output gets reviewed against the standard. When the output is wrong, you do not just rerun, you update the context so the next run does not make the same mistake. The agent improves the way a person would, through coaching written into context.

> **FIELD-TESTED RULE**
>
> If your agent makes the same mistake twice, it is your fault, not the agent's. Fix the context.

PART 5

When not to use AI

The AI-washing pattern is everywhere. Founders pitch AI-powered everything because the term raises funding. Teams add a model to a workflow that was working fine, increase the operating cost, decrease the reliability, and call it innovation.

A practitioner respects the problem more than the tool.

USE THIS APPROACH	WHEN THE PROBLEM HAS
AI / LLM	Unstructured inputs that need parsing; generation requirements; reasoning across ambiguous context; high variation across instances; a human reviewing the output before consequence
Traditional automation	Structured inputs; deterministic logic (if X then Y); high volume with low variation; low tolerance for non-determinism
Classical ML (not LLMs)	Large labeled datasets; a well-defined prediction target; fast inference at scale; latency or cost constraints that rule out LLMs
A human	High consequence and low reversibility; trust or relationship dynamics; genuine novelty with no analog in training data; ethical, legal, or fiduciary weight

These categories overlap. Most real systems blend several. The job of the strategist is to draw the boundaries with care, not to default to AI because AI is the trend.

A short list of places where the right move was to remove AI and replace it with something simpler: email triage where a sender-and-subject rule covered most cases; document classification where the documents had unique IDs you could match on; smart routing where a lookup table did the job in a millisecond; a research summarizer where the user was reading the summary and then reading the full document anyway.

Restraint is a senior practice.
The market does not reward it. The work does.

PART 6

Why most AI projects fail

There is a set of numbers that should change how you think about all of this, and almost nobody building AI content will say them plainly. The published research on enterprise AI is brutal and consistent.

MIT's Project NANDA, studying more than three hundred initiatives, found that ninety five percent of organizations saw zero measurable return from generative AI. RAND, looking across more than two thousand four hundred projects, put the failure rate near eighty percent, twice the failure rate of ordinary IT projects, and noted it has barely moved in three years. Gartner expects more than forty percent of agentic AI projects to be canceled by the end of 2027, and most projects without production-ready data to be abandoned before then. Independent benchmarks of agents doing real office work show the best systems finishing about a third of tasks end to end. Industry surveys put the share of pilots that never reach production near ninety percent.

Read those numbers and the obvious conclusion is that AI does not work. That is the wrong conclusion. Look at what the same research says about the cause.

The failure is almost never the model. The organizations that succeed define the business outcome before they write a line of code. Most do the reverse: they start with the tool because the tool is exciting, and hope the value shows up later. The projects die from unclear success metrics, from sponsorship that evaporates after the first impressive demo, from data that was never ready for production, and from workflows the new system was bolted onto rather than built into. The model was the one part that worked.

The failure is almost never the model.

This is the restraint discipline seen from the altitude of a whole organization. A demo runs on clean inputs and a cooperative path. Production runs on messy data, real consequences, and a verification cost that never goes away if the system does not learn from its corrections. The distance between those two is where the budget dies.

The practitioner's job, and the strategist's especially, is to see that distance before the money is committed. Most of what kills an AI project is an organizational decision wearing a technical costume. Name it as what it is, early, and you either save the project or you save the spend.

The personal version of the same failure

The organizational data has an individual mirror, and it is the finding I would most want a new practitioner to sit with.

In February 2026 Anthropic published a controlled trial of fifty two developers learning an unfamiliar library. Half used AI assistance, half did not. The AI-assisted group produced better code during the exercise. Then both groups took a comprehension quiz with no AI available, and the AI-assisted group scored seventeen points lower, fifty percent against sixty seven. The loss concentrated in questions about *why* something was built a certain way, not questions about where the code does a particular thing. People kept the map and lost the reasoning.

Here is the part that matters. The tool was not the variable. The interaction pattern was. Sorted by how people worked with the model, comprehension ranged from twenty four percent to eighty six percent. Full delegation, meaning ask for code and accept it, scored twenty four. Generating code and then interrogating it line by line scored eighty six, beating the group that used no AI at all.

There is a companion finding worth holding next to it. A 2025 trial of sixteen experienced developers working in codebases they knew well found they expected AI to make them twenty four percent faster, estimated afterward that it had made them twenty percent faster, and measurably took nineteen percent longer. A thirty nine point gap between the feeling and the fact.

> **FIELD-TESTED RULE**
>
> Before you approve an AI project, write the one sentence that says what business outcome it will change and how you will measure it. If you cannot write that sentence, you do not have a project. You have a demo.
>
> And at your own desk: never accept output you could not explain to someone else. Generate, then interrogate. That single habit is the difference between an operator who compounds and one who plateaus.

PART 7

Setting up your workbench

Everything up to here is orientation. This part is the workbench. By the end of it you will have a place for code to live, an agent that can act on it, and a review gate between the agent and anything that matters.

The setup has four pieces and they are worth naming before you install anything.

PIECE	WHAT IT DOES
GitHub and git	Where the work lives. Your undo button, your audit log, and your review gate.
An editor	Where you read what the agent did. VS Code or Cursor.
An agent	The thing that reads the project, writes files, and runs commands. Claude Code, Codex, or the agent built into your editor.
Context files	The written instructions the agent reads on every run. This is your harness in its simplest form.

Set them up in that order. People who start with the agent and add version control later spend their first month unable to answer the question *what did it change*, which is the only question that matters when something breaks.

> BEFORE YOU START
>
> Commands and install steps in this section were correct in August 2026 and will drift. Where a command does not work, go to the tool's own documentation rather than an article. The shape of the workflow is stable; the exact flags are not.

Step 1: GitHub, before anything else

An agent makes a lot of changes quickly. Without version control you cannot see what it touched, cannot roll back a bad run, and cannot review before it lands. Git gives you all three, and it is the reason agentic coding is workable at all.

Create a free account at github.com. Then install git and the GitHub command line tool.

```
# macOS (Homebrew)
brew install git gh

# Windows: install Git for Windows from git-scm.com,
# then GitHub CLI from cli.github.com

# Debian / Ubuntu
sudo apt install git gh
```

Set your identity once, so every commit is attributable:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
```

Then authenticate the command line tool with your GitHub account. This step is the one most guides skip, and it is what later lets your agent open pull requests on your behalf without you handling tokens by hand.

```
gh auth login
```

Choose GitHub.com, HTTPS, and authenticate in the browser. Verify it worked:

```
gh auth status
```

Now make your first repository and clone it locally:

```
gh repo create my-first-project --private --clone
cd my-first-project
```

Step 2: the git you actually need

You do not need to learn git properly to start. You need six commands and one rule.

COMMAND	WHAT IT DOES
`git status`	What has changed since the last save point. Run this constantly.
`git diff`	Shows the actual changed lines. This is how you review an agent's work.
`git add .`	Stages changes for the next save point.
`git commit -m "..."`	Creates the save point, with a message saying what changed and why.
`git switch -c name`	Starts a new branch, a parallel line of work that cannot damage the main one.
`git push`	Sends your commits up to GitHub.

The mental model: a commit is a save point you can return to. A branch is a parallel copy where you can make a mess safely. A pull request is a proposal to merge that mess back in, with a diff you read first.

> **FIELD-TESTED RULE**
>
> Never let an agent work directly on `main`. One branch per task, one pull request per branch, and you read the diff before you merge. This single rule converts an agent from a liability into a colleague, because every change becomes reviewable and reversible.

Step 3: an editor

Two reasonable choices, and the difference is a matter of how you like to work.

VS Code is free, ubiquitous, and does everything. Install it from code.visualstudio.com. Add the GitHub Copilot extension if you want inline suggestions and an in-editor agent. Copilot has a free tier and a paid tier around ten dollars a month.

Cursor is a fork of VS Code with an agent built into the core of the editing experience rather than bolted alongside it. Your VS Code extensions and keybindings carry over. Free tier available, paid tiers start around twenty dollars a month and climb with usage.

My recommendation for someone starting: install VS Code and use a terminal agent alongside it. Keeping the two roles separate, the editor for reading and the terminal agent for doing, teaches you what the agent is actually doing. Move to Cursor once that is second nature and you want the loop tighter.

Step 4: a terminal agent

This is the piece that does the work. Install one, not three.

Claude Code

Anthropic's terminal-native agent. It reads your repository, edits files, runs commands, and manages git through natural language. It is the strongest option for work that spans many files and needs a plan before edits. Requires Node.js.

```
npm install -g @anthropic-ai/claude-code

cd my-first-project
claude
```

On first run it will walk you through authentication. It works against a Claude subscription (Pro from around twenty dollars a month, Max tiers above that) or an API key with usage billing. Inside a session, `/init` generates a starting context file for the project, and plan mode makes it lay out an approach for your approval before it touches anything. Use plan mode until you trust it.

OpenAI Codex

OpenAI's agent, available as a command line tool, an editor extension, and a web surface. Its particular strength is delegated work: hand it a defined task, let it run in the cloud, review the result later. Included with ChatGPT paid plans on a credit system.

```
npm install -g @openai/codex

cd my-first-project
codex
```

OpenCode

Open source, MIT licensed, and model-agnostic, meaning you supply whichever provider key you want. Worth knowing about if you would rather not be tied to one vendor, or if you want to run against a local model.

> ON COST
>
> Agentic coding consumes far more tokens than chat, because the agent reads files, runs commands, and reads the output of those commands, over and over. A flat subscription is usually cheaper and far more predictable than pay-per-token while you are learning. Set a spending limit before your first long session, not after.

Step 5: the permissions conversation, before the first run

An agent in a terminal can do anything you can do in that terminal. Decide the boundaries before you find them by accident.

- Run the agent from inside the project directory, never from your home directory or the root of your drive.
- Keep every project in git before the agent touches it, so `git diff` and `git restore` are always available.
- Start with approval required for commands. Both major agents ask before running things by default. Leave that on until you have a feel for what it wants to run.
- Never put real credentials, API keys, or customer data in a repository an agent works in. Use a `.env` file, and put `.env` in `.gitignore` before you write anything into it.
- Autonomous or unattended modes exist and are genuinely useful. Earn them. Use them in a container or a throwaway machine, not on the laptop with your tax returns on it.

Step 6: connecting the agent to GitHub

This is the step that turns a code generator into something that participates in a real workflow. Because you ran `gh auth login` in step one, the agent inherits your GitHub access when it runs commands. Nothing further to configure.

The loop looks like this, and you can hand it to the agent almost verbatim:

1. Create a branch for the task. `git switch -c fix-contact-form`

2. Do the work. The agent edits files and runs whatever checks exist.
3. Commit with a message that says what changed and why.
4. Push the branch. `git push -u origin fix-contact-form`
5. Open a pull request. `gh pr create --fill`
6. **You** read the diff on GitHub, in the browser, away from the conversation that produced it.
7. Merge if it is right. If it is not, comment and send it back.

Step six is not a formality. Reading the diff in a different place from where the work happened breaks the spell of a persuasive explanation and puts you in front of what actually changed.

Give the agent a signal it can read

Add a GitHub Actions workflow so tests run automatically on every pull request. This matters more than it sounds: it gives the agent an objective pass or fail signal it can fetch and respond to, instead of relying on its own judgment about whether the work is done.

```
mkdir -p .github/workflows
```

Create `.github/workflows/ci.yml` with whatever check suits your stack. Then the agent can run `gh pr checks` and see the truth rather than assert it.

Close issues automatically

If you track work as GitHub issues, put a closing keyword in the pull request *description*, not the title and not a comment:

```
Closes #12
```

The rules here fail quietly in several ways. Keywords are ignored in a pull request title and in comments. They are ignored entirely unless the pull request targets the default branch. One keyword followed by a list closes only the first issue. Each of those produces a pull request that looks correct, merges cleanly, and leaves your tracker wrong.

Step 7: context files

Your agent has no memory between sessions. Context files are how you stop re-explaining your project every morning.

Two names matter. `AGENTS.md` is the emerging cross-tool convention that most agents now read. `CLAUDE.md` is Claude Code's own. Many projects keep both, with one a pointer to the other.

What belongs in it:

- What this project is, in two sentences.

- The stack, and any version constraints that will bite.
- The exact commands to install, run, test, and build. Write them out. An agent that has to guess how to run your tests will guess wrong.
- Conventions you care about: file layout, naming, formatting, commit message style.
- What never to touch. Generated files, vendor directories, anything with credentials.
- How you want it to behave: ask before large refactors, keep changes small, always run tests before saying it is done.

Commit the file. Improve it every time the agent gets something wrong in a way that a written instruction would have prevented. That habit, more than any prompt technique, is what separates a Level 2 builder from a Level 1 user.

> **FIELD-TESTED RULE**
>
> Your context file is a living document, and the correction loop is the point. When an agent makes a mistake, do not just fix the output. Add the line to the context file that would have prevented it. The system gets better while you sleep, which is the entire idea.

PART 8

Giving your agents real tools

A context file tells an agent about your project. A **skill** tells it how to perform a specific piece of work: what the procedure is, when not to use it, and how to tell whether it succeeded. It is the difference between briefing a new hire on the company and handing them the standard operating procedure for the task in front of them.

This is the highest-leverage thing available to you as an individual, and it follows directly from the harness argument in Part 1. You cannot make the model smarter. You can absolutely write down, once, the procedure you keep re-explaining, and have every future session start from it.

The portability problem

Write a procedure for Claude Code and it lives in Claude Code's format. Write it for Cursor and it lives in Cursor's. Switch tools, or work with someone who uses a different one, and the work does not travel. Most people solve this by maintaining several copies, which is the same as maintaining none, because they drift.

The fix is to keep one harness-agnostic source file per procedure and generate the tool-specific versions from it. Write once, use in every harness.

A worked example: Zen Agent Skills

This is my open source implementation of that idea, and it is free under the MIT license. It is a portable library of reusable agent skills plus the tooling to distribute them across coding harnesses.

github.com/hams-ollo/zen-agent-skills

Each skill is one source file, `SKILL.md`. The installer places that source where a supported tool discovers it, or generates a thin native adapter for the target project. The scripts use only the Python standard library, so there is nothing to install beyond Python 3.11 or newer.

```
git clone https://github.com/hams-ollo/zen-agent-skills.git
cd zen-agent-skills

# See what would happen, change nothing
python scripts/install.py --dry-run

# Install for Claude Code and OpenCode
python scripts/install.py

# Generate Cursor and VS Code adapters into your project
python scripts/build-adapters.py --target cursor,vscode --out ../my-project
```

The adapters land as `.cursor/rules/` entries for Cursor and `.github/prompts/` entries for VS Code or Copilot, with the supporting material they reference copied alongside so nothing dangles.

What the skills actually do

They compose into one development spine, while remaining useful individually:

STAGE	WHAT HAPPENS
Setup	`project-bootstrap` scaffolds the project, `init-worktracking` stands up a local work tracker.
Contract	`spec-author` turns a rough idea into a written specification. `spec-plan-readiness` gates it before a line of code is written.
Build	`new-task` decomposes the approved spec into atomic tasks. `fix-batch` handles grouped corrections.
Verify	`test-author` derives tests from the spec's scenarios. `spec-conformance` audits the implementation against the contract. `verifier-agent` runs the declared commands and returns pass, fail, or blocked, with evidence.
Land	`reconcile-worktrees` merges parallel work. `doc-sync` finds which documents the change invalidated. `pr-describe` writes the pull request and gets the issue-closing reference right.

The shape is the point, more than any individual skill. A rough idea becomes a written contract. The contract is checked for readiness before implementation. Implementation is decomposed into small pieces. Tests come from the contract rather than from the code, so they can catch the code being wrong. An independent verification runs real commands and reports evidence rather than an opinion. Only then does anything land.

That sequence is not novel. It is how careful engineering teams have always worked. What is new is that you can now hand it to an agent as written procedure and have it followed on every task, by a solo practitioner, without a team.

If you build your own

You do not need my library. You need the discipline it encodes. A skill worth writing has four parts: what it does, when *not* to use it, the procedure, and how to tell it worked. That last one is where most attempts fall down. A procedure with no acceptance criteria produces an agent that declares victory.

> **FIELD-TESTED RULE**
>
> No skill ships cold. A written procedure for a workflow you have never actually performed is fiction. Do the work manually at least once, notice where it went wrong, and write the skill from what you learned. This is the bar for contributions to my library and it is the right bar for your own.

A SECURITY NOTE THAT APPLIES TO EVERY SKILL LIBRARY, MINE INCLUDED

A skill is not a library your code calls in a sandbox. It is prose an agent reads and acts on, in your repository, usually with permission to write files and run commands. Review one before installing it, with the same standard you would apply to a script you were about to run, and especially for skills from outside a source you trust.

PART 9

Security and governance

Every cheat sheet on the internet tells you how to start using AI. Almost none tell you how not to get hurt using it. This section is short because the topic is large, but the points below will protect you from most of the damage I have seen in production.

The threat model in plain English

When you give an agent access to your data and tools, three things can go wrong.

Prompt injection. Hostile content inside something the agent reads (a webpage, an email, a PDF, a tool description) contains instructions the agent treats as commands, and it does something you never asked for.

Data exfiltration. The agent is tricked into sending sensitive information to an attacker, often inside a URL or an API call to a malicious endpoint.

Unintended action. The agent acts on a misunderstood instruction and does something irreversible: deleting files, sending messages, making purchases.

None of these are hypothetical. In July 2025 an AI agent deleted a production database during an explicit code freeze at a well-known platform, and the company's chief executive called it unacceptable. Independent scans through late 2025 and 2026 found thousands of applications built quickly with AI tools leaking user data, exposed credentials, and personal information at scale. One engineer extracted debt amounts, home addresses, and administrator keys from live applications with fifteen lines of code, in his words in less time than it took to finish lunch.

The connector problem

Model Context Protocol, the standard for plugging tools into agents, was handed to the Linux Foundation in December 2025 and is now genuinely ubiquitous. It is also young, and security researchers have found real structural issues. Malicious instructions can hide inside a tool's description, visible to the model and not to you. Injection can land at the moment tools are listed, before any human approval step, which turns your review into a rubber stamp. A critical remote code execution flaw was found in the official inspector tool in mid-2025. A popular package was quietly modified to copy every email it sent.

This does not make the standard bad. It makes it new. Connect tools you have a reason to trust, prefer official servers over convenient ones, and treat a tool description as untrusted input, because that is exactly what it is.

The practitioner's defaults

- **Least privilege.** Every agent gets only the access it needs for the job, nothing more. A research agent does not need your email. A draft-writing agent does not need your bank.

- **Read-only by default.** Write access is granted explicitly and minimally. Destructive actions (delete, send, purchase) require human confirmation.
- **Sandboxed execution.** Code-running agents work in isolated environments. They cannot reach your real filesystem or network without explicit permission.
- **Version control as a safety net.** Everything an agent touches lives in git. If you cannot diff it, you cannot review it, and if you cannot roll it back you should not have let the agent near it.
- **Logging and observability.** Every agent action is logged. You can replay what an agent did, in order, after the fact. If you cannot audit it, you cannot trust it.
- **Treat all external content as untrusted.** Webpages, emails, PDFs, tool descriptions, even shared documents from colleagues can carry prompt injection. The agent's job is to use the content, not to obey it.
- **Human in the loop where consequences matter.** Approval gates before any high-stakes action. The agent prepares, the human approves.

A reasonable governance baseline

For a small team or a solo operator: a written list of what each agent is allowed to do and what it is not; audit logs you actually look at, weekly; a documented incident response that says what you do if an agent does something it should not; and a periodic review, monthly is fine, of which agents are still in use and which can be retired.

This is not the exciting part. It is the part that lets the exciting part survive contact with reality.

PART 10

The 90 day practitioner's path

Most guides hand you frameworks and leave you to find the path. Here is the path I would walk if I were starting again.

Days 1 to 30: master the chat box

1. Pick one model and commit for the month. Not all three.
2. Start using Projects (or the equivalent). Treat every recurring kind of work as a project.
3. Write your first context block. Two hundred to five hundred words about who you are, what you do, how you write, what you value, what your standards are. Use it in every project.
4. Practice reverse prompting on every task for two weeks. Force yourself to let the model pull context from you.
5. Practice generate-then-interrogate on anything technical. Never accept output you could not explain to someone else.
6. Pick one weekly workflow. Use AI on it every week. Refine. Notice what context made the biggest difference.

Outcome at day 30. You have stopped using AI as a search engine. You have a stable context artifact. You have one workflow meaningfully better than it was a month ago.

Days 31 to 60: build the harness

1. Work through Part 7. GitHub account, git installed, `gh auth login` done, first repository created.
2. Install one terminal agent and one editor. Use them weekly, not daily. Get past the awkward phase.
3. Write an `AGENTS.md` for your first real project. Commit it.
4. Run the full loop once, deliberately: branch, work, commit, push, pull request, read the diff, merge. Do it on something small enough that you can hold the whole change in your head.
5. Set up a personal knowledge base. Keep it simple: daily notes, project notes, reference notes. Let the agent read it when relevant.
6. Initialize a private repository for your context files, system prompts, and any skills or templates you build. This is the start of your harness as a versioned artifact.

Outcome at day 60. The AI knows what you have written, what you have decided, and what you are building. Your context is versioned. You have shipped at least one change through a real review gate.

Days 61 to 90: first agent, first pipeline

1. Install a skills library, mine or your own, and use one workflow skill end to end on a real task.

2. Write your first skill from scratch. Pick a procedure you have performed manually at least twice and keep re-explaining. Include acceptance criteria.
3. Identify one autonomous task: something you would happily hand to a junior employee if you had one. Research a topic. Draft weekly content. Process incoming messages.
4. Write a job description for that agent: what it does, what good output looks like, what it must never do.
5. Set it up with the smallest privilege footprint that works. Read-only if at all possible.
6. Run it for two weeks. Review every output. Refine the job description after each session.
7. After two weeks, decide: keep, simplify, or retire.

Outcome at day 90. You have a working agent you trust to do one thing well, running inside a workflow with a review gate. You have learned, in your own hands, the difference between *AI helped me* and *AI did the work for me*. That difference is the leverage everything else is built on.

Beyond day 90

This guide is the foundation. The work goes deeper in two seasons at zen-solutions.dev. The Foundations series builds the literacy and judgment for people who are not ready to build yet, including the founders and decision-makers who need to evaluate AI without being fooled by it. The 26 week Field Manual series then builds the practice one layer at a time, from your first context file through your own agents and pipelines. Each piece ships as a video, a written companion, and a commit to an open source Starter Kit. It is free.

CLOSING

The operator's mindset

There is a verse in the Bhagavad-gita, *karmany evadhikaras te* (BG 2.47), that translates roughly as: you have a right to the work, never to its fruits. It is one of the oldest pieces of operating doctrine in human history, and it is unreasonably relevant to working with AI.

The work is yours. Showing up, choosing well, being honest about what is yours to do and what is not. The fruits are not. Whether the model works today, whether the project lands, whether the audience comes, whether the business pays off, all of that is downstream of forces you do not control.

The operator who internalizes this gets quieter and more effective. They do not chase the latest tool. They do not panic when a model release breaks something. They do not over-engineer in pursuit of the perfect setup. They show up to the work, they build their harness, they lead their agents, they exercise restraint where it is called for, and they ship.

It also sets the standard for what is worth building. Give someone a solution and you help them once. Teach them to build solutions and you change what they are capable of. The systems worth making are the ones that leave the people who use them more capable, not more dependent. That is the difference between *AI helped me* and *AI did the work for me*, written at the scale of a tool you put in front of other people.

Methods are many. Principles are few.
The one who grasps the principles selects their own methods.

That is the line I keep coming back to in this work, and it is the thread that runs through every section of this guide.

This is the work I do every day, and it is the work I want to share. If it lands, come find the rest of it at zen-solutions.dev. The Foundations series and the 26 week Field Manual series are where the real journey begins.

I am glad you read this far. That alone tells me something about you. Welcome aboard.

Hans Havlik

APPENDIX

About the author

Hans Havlik (Hamsa) is an AI solutions architect with ten years as a developer and five in enterprise AI delivery. His work centers on multi-agent orchestration, agentic harness design, and the retrieval, recommendation, and prediction pipelines that go with them, built and shipped for enterprise clients across aerospace, global logistics, financial services, healthcare, and IT.

He came to the work by an unusual road. EMT at sixteen. After leaving college, he enlisted in the US Army and trained as a 68W Healthcare Specialist attached to a Cavalry unit, then served in the Army Reserves while building a parallel life as a student, father, and working medical professional, through hospital and VA medicine and the COVID period, before moving to full-time software and AI in 2022. He writes at Zen Solutions about AI engineering, contemplative practice, and the discipline of building well. He is a father and a Gaudiya Vaishnava practitioner in Alachua, Florida.

Follow the work

- Website: zen-solutions.dev
- Substack: substack.com/@zensolutions
- YouTube: youtube.com/@zen-solutions-dev
- GitHub: github.com/hams-ollo
- Skills library: github.com/hams-ollo/zen-agent-skills

Writers and practitioners worth your time

- **Dan Koe**, for the steady work he does in public on what one person can build with modern tools.
- **The Anthropic engineering team**, for writing publicly about how Claude Code and context engineering are meant to work.

On the figures in this guide

Research findings cited here come from primary sources where possible: controlled trials from Anthropic and METR, benchmark analyses from adversarial re-tests published on arXiv, the Stanford HAI AI Index, MIT Project NANDA, RAND, and named independent security firms. Vendor claims are identified as such. Tool commands, prices, and version numbers were correct in August 2026 and will drift. The principles will not.

This is version 3.0 of the AI Practitioner's Field Manual. It is revised as the field moves. The current version is always at zen-solutions.dev.