The Artisan and the Agent

A field manual for building a Laravel SaaS with AI agents

The Artisan and the Agent

A field manual for building a Laravel SaaS with AI agents

Written by maudev, @maudevme

Cover illustration by Mary Amato (Notioly), @maryamato88.

This book is self-published.

In this free preview

This is a free preview of the Artisan and the Agent. Pass it on to anyone you think would want it.

You get four things here. The contents of the complete book, so you can see what it covers. The first chapter in full, which is the map. The tagged history of the app that comes with the book, one tag per chapter-sized change. And three of the book's prompts, word for word as they appear in it.

The method I've laid out includes five ordered phases. Contract, Analyze, Build, Test, Deploy. Write the contract, attack it, build against it, test against it, and let a pipeline gate the deploy on all of it. A change goes spec first, docs second, code last.

Let's get started.

What is in the complete book

Five parts, one per phase, and a short closing chapter. Six appendices at the back. Every chapter opens with the tag or tags it builds, and every section ends with tips and with prompts you can paste into the coding agent you already run.

Introduction

An AI agent can write production code. That question is closed. The open question is what you have to do so the code it writes is code you'd run a business on. This book is my answer, and this first chapter is the map: what is in the book, how each chapter is laid out, then why I wrote it, and who it is for.

> ***Follow along:*** `git checkout step-00` *through* `step-06` *is the base the template is built on. From fresh installer output it adds Laravel Boost and* `.ai/guidelines/`*, Pint, the three Rector configs, the composer scripts (* `fix`*,* `lint`*,* `test`*,* `ci`*), the model conventions every later line inherits (strict Eloquent,* `public_id` *UUIDs,* `final` *controllers), and the first project rules in* `.ai/rules/`*. Run* `git diff step-00 step-06` *once now. It reads in minutes.*

What is in this book, and how to read it

The book has one part per phase, plus a short closing chapter on what to do next.

- **Contract** writes down what the product is, in steering docs, an OpenAPI spec, and an infrastructure contract, before any code exists.
- **Analyze** attacks that contract the way an attacker would, runs the scanners over it, and hardens the infrastructure the findings point at.
- **Build** has the agent write the code inside your conventions, in small slices, with your diff review as the main work.

- **Test** builds a four-ring suite read from the outside in: browser journeys, then feature, architecture, and unit tests.
- **Deploy** makes the CI pipeline the gate, so nothing ships that quality, security and the tests did not clear.

The template is the spine: a Laravel app carrying a contract-first JSON:API surface, OAuth via Passport, Standard Webhooks both ways, an MCP server so an agent can run what the REST API exposes, hardened HTTP defaults, the four-ring test suite, and a CI pipeline that gates deploys. It has no product domain on purpose. Your product is what you build on it. Its git history is tagged `step-00` through `step-30`, one chapter-sized change per tag, so every decision is a diff you can read.

```
git tag                    # step-00 … step-30
git checkout step-09       # the app exactly as the
JSON:API chapter leaves it
git diff step-08 step-09   # precisely what that chapter
changed
git checkout main          # back to the finished template
```

Getting it running takes four commands.

```
composer run setup        # install, .env, app key,
migrate, build assets
php artisan boost:install # pick YOUR coding agent
(recorded in boost.json)
php artisan passport:keys # OAuth signing keys for this
machine
composer test             # 166 tests green before you
write a line
```

`boost:install` asks which coding agent you run (Claude Code, Codex, OpenCode, Cursor), records the answer in `boost.json`, and generates that agent's instruction files from `.ai/guidelines/`.[1] After any guideline edit or Boost upgrade, `php artisan boost:update` makes them again. A second agent, or a switch, is `boost:install` again.

Two files are off limits. `CLAUDE.md` and `AGENTS.md` are generated, so a hand edit lasts until the next update and then goes. Files under `.ai/rules/` are off limits too: you record a project rule by asking the agent to remember it, which keeps Boost's rules index true, and a rule file you wrote yourself is invisible to every agent that reads that index.

If `boost:update` refuses with "Please set up Boost with [php artisan boost:install] first", `boost.json` has no agents in it. An install that asked no questions leaves it that way. Run `boost:install` once and answer it.

Six appendices sit at the back.

- **A** puts every AI Prompts section from the chapters in one place, in reading order.
- **B** is a steering-doc template you can copy into your own repo.
- **C** reproduces the template's `.ai/guidelines/` and `.ai/rules/` files in full, with a note on why each part is there.
- **D** walks through creating your own AI skill.
- **E** builds Standard Webhooks by hand, both directions.
- **F** covers running several agents at once in git worktrees with Herd.

Tips

- Keep the app at `main` while you read. `git checkout step-NN` shows the code as a chapter leaves it, and `git checkout main` brings you back.

- Don't fork the template into your product on the first read. Walk the ladder once with the diffs open, so you can understand the method first.

AI Prompts

```
This repo is a Laravel template with a tagged git history,
step-00 through step-30, where each tag is one chapter-
sized change. Read the README and steering/_index.yaml,
then give me a reading plan: for each tag, one line on
what lands and which phase it belongs to (Contract,
Analyze, Build, Test, Deploy). Flag any tag whose purpose
you can't infer from its diff alone, so I know where to
slow down and read the chapter first.
```

The sections you will find in every chapter

Every chapter opens with a **Follow along** section naming the tag or tags it builds, the files to read at that tag, and the diff to run against the step before. Not every chapter involves writing code; some are purely strategic. When this happens, the chapter will tell you and guide you on where to use the concept in your own build.

Every section ends with **Tips**, short do-this-today advice, and **AI Prompts**, prompts to paste into any coding agent you run. They say "my OpenAPI spec" and "my steering docs" because you run them against your own work.

Edit the prompts. Each carries an order of operations, and some carry a point where the agent must stop and refuse to go on. Keep those and rewrite the rest in your project's words and file names.

Keep the checkpoints. Many prompts stop and ask the agent to show a diff or a list before going on. Every expensive agent mistake I've cleaned up so far came from a session where I skipped that checkpoint.

Work in slices. One endpoint, one webhook event, one gate. If you ask for the whole product in one prompt, you will get more code than anyone can check.

Extend Boost shows how to keep a rule after the prompt that found it is gone. An app-wide rule goes into `.ai/guidelines/`, then `php artisan boost:update`. For location-specific rules, just ask the agent "Remember that..." — it will save your instruction in `.ai/rules/` as a project rule.

Take the slicing rule above. A rule you only say in a prompt dies with the session, so it belongs where the agent reads it every time.

> ***Extend Boost:*** *add "Build one slice at a time: one endpoint, one webhook event, one gate. Stop and show me the diff before you start the next one." to* `.ai/guidelines/conventions.md`*, then run* `php artisan boost:update`*. From then on the agent comes back for the next slice instead of writing five.*

When you'd choose differently highlights the limits of each choice, so you know when to take a different approach for your own project. Every decision in this book was made for one kind of product: a solo-built, API-first SaaS. Whenever a choice is presented, this section explains its boundaries and what to do in a different context, preventing you from blindly applying a rule to the wrong project.

One example. The template authorizes with a policy per model and denies as not found, so a request that isn't allowed looks like a missing record. That decision gets its box here.

> ***When you'd choose differently:*** *if your product is a pure single-owner API, with every row in one account and no role sharing, scoping every query to the logged-in account can carry the whole authorization story with less code than policy classes. The moment a second role appears, an admin, a team, a support person, the policies in Part 3 become the map of who may do what.*

Homework is where your product gets built. Some chapters end with one: giving you the prompts that turn that chapter into your own code.

Tips

- Start a prompts file in your repo on day one. Each time a prompt from this book works after you edited it, save your version. In six weeks that file will speak your project's language better than the book does.
- When a prompt gives a bad result, fix the prompt or the steering doc it points at rather than the output, so you will have covered every similar case too.

AI Prompts

```
I'm adapting a library of reusable prompts to my project.
Read my repo's steering docs and conventions, then take
the following generic prompt and rewrite it to reference
my actual file paths, doc names, and conventions,
preserving its structure and every verification checkpoint
it contains: [paste prompt]. Show me the adapted version
and list what you changed.
```

Why I wrote this book, and who it is for

The build behind this book is MiniMailer: email infrastructure for people who build AI agents, with MCP as a first-class client, on Laravel. MiniMailer includes a REST API, a customer dashboard, signed webhooks, and an agent surface. Where an AI agent wrote most of the code, I wrote the specs, read the diffs, argued about design, and sent much of the work back.

That build changed what slows me down. For most of my working life, the bottleneck was simply typing. An agent takes the typing and leaves you the part behind it: what the controller must enforce, which rule the migration must never break, what "done" means for this feature. If you don't write anything of that down, the agent will pick its own answer, which you'll probably meet three features later in the form of a bug.

So the method fits in one line. Contract → Analyze → Build → Test → Deploy. Write the contract, attack it, build against it, test against it, and let a pipeline gate the deploy on all of it. A change goes spec first, docs second, code last.

None of that work is new. Lead developers have always written specs and read other people's code. What changed is the ratio: when the code is nearly free, the spec and the checking are nearly the whole job.

You also stop holding the whole codebase in your head, and that took me a while to accept. I can't walk you through every private method on that build. What I can show you is the spec each behavior traces to, the test that pins it, and the gate that would catch it changing. That is a different kind of ownership, and it is the kind that still works after a codebase grows past what one person can remember.

Every phase in that line is there because skipping it has cost me something on that build. The steering docs were mine to keep current. The OpenAPI spec was hand-written before any endpoint existed, and a code change

with no spec change was not finished. The agent got the contract, one narrow slice, and the acceptance criteria. I read the boilerplate quickly and slowed down where mistakes are expensive: authorization checks, data boundaries, error paths, anything touching money.

This book is for a Laravel developer who wants to ship a real SaaS with an agent doing the typing, without letting quality drop.

I assume you already work with AI: you know what an LLM is, you have run a coding agent, and you know Laravel and its ecosystem. So I don't explain what a migration is, what HTTP status codes mean, how an LLM works, or how to install PHP. What I don't assume is that you have run an agent against a codebase with paying customers on it. That is where the easy demos stop working and the habits in this book start to matter.

Tips

- Track how you spend a build day for one week. If most of it is still code rather than spec and review, you're using the agent as autocomplete.
- Write the acceptance criteria before the prompt. If you can't say what "done" looks like in a form you can check, the agent can't hit it.
- Keep a list of every mistake the agent repeats. Each entry is a rule missing from your written conventions.
- When something goes wrong, name the phase that failed before you fix the file. A bug that traces to a missing spec line is a Contract failure, and a code patch alone will not prevent the next bug of that kind.

AI Prompts

```
I work in five ordered phases: Contract (specs and
steering docs), Analyze (adversarial review of the
design), Build (implementation against the contract), Test
(outside-in: browser, feature, architecture, unit), Deploy
(a CI gate that blocks on quality, security, and tests).
For the feature I'm about to describe, tell me which phase
we're in, what artifact this phase must produce, and what
you need from the previous phase's output before you can
start. Refuse to skip ahead if an input is missing.
```

SOURCES

1. Laravel Boost documentation: installation and agent set-up, AI guidelines, project rules and skills. https://laravel.com/docs/13.x/boost

The ladder

The book ships with a Laravel app that has no product domain in it. What it has is a git history cut into chapter-sized pieces, one tag each. You read a chapter, then you read the diff that chapter made.

```
git tag                    # step-00 … step-30
git checkout step-09       # the app exactly as the
JSON:API chapter leaves it
git diff step-08 step-09   # precisely what that chapter
changed
git checkout main          # back to the finished template
```

A decision you can read as a diff is a decision you can argue with. Every tag, in order:

- `step-00` — Switch the JavaScript toolchain to bun
- `step-01` — Install Laravel Boost and make guidelines the contract
- `step-02` — Tighten Pint beyond the Laravel preset
- `step-03` — Add three Rector configurations: app, migrations, tests
- `step-04` — Split composer scripts into fix, lint, test and ci
- `step-05` — Lock the application conventions: strict Eloquent, public identifiers, final controllers
- `step-06` — Record the project rules: path-scoped law in .ai/rules, one index for agents

- `step-07` — Add the steering layer: design docs, a machine-readable index, and the agent contract
- `step-08` — Author the API contract and serve it raw, host shape decided by config
- `step-09` — Lay the JSON:API foundation: negotiation, strict queries, one error producer, schema registry
- `step-10` — Wire Spectator so the contract is enforced in tests, not just published
- `step-11` — Enforce the change hierarchy with a route-to-contract parity test
- `step-12` — Own the security headers and publish the two well-known files
- `step-13` — Guard outbound requests against SSRF and make API 404s carry zero information
- `step-14` — Install Passport: personal access tokens for confidential clients, PKCE for public ones
- `step-15` — Authorize the Laravel way: a policy per model, denials that read as not found
- `step-16` — Send Standard Webhooks: signed deliveries, rotating secrets, spec retries, a breaker
- `step-17` — Receive Standard Webhooks: raw-body verification, replay window, idempotent redelivery
- `step-18` — Serve MCP over Passport: agents are API clients, tools gated by token scope
- `step-19` — Pin the dashboard to the free tier of Flux
- `step-20` — Take package guidance from Boost's skills rather than a hand-written copy
- `step-21` — Complete the testing onion: arch presets, layering rules, the opt-in browser ring

- `step-22` — Test the tests: mutation runs with covers() targets and one composer script
- `step-23` — Discipline the test harness: frozen clocks and a fail-closed network guard
- `step-24` — Build the docs site: guides and changelog from Markdown, the API reference from the contract
- `step-25` — Make the pipeline the gate: quality, security and tests before any deploy
- `step-26` — Gate promotion on shape: ops:verify-env refuses a misconfigured production boot
- `step-27` — Close the supply chain: verified gitleaks in CI and Dependabot on every ecosystem
- `step-28` — Let an agent with no browser claim a token: the device grant under auth.md names
- `step-29` — Answer each failure by what it would reveal: 401, 403 and one 404
- `step-30` — Deploy production from main, with a preview environment per pull request

Three prompts

Every section of the book ends with prompts, and they all sit together in Appendix A. They are written for any coding agent, and the parts written like `[this]` are yours to fill in.

Three of them are below, covering the places where a build goes wrong most commonly: the spec nobody attacked, the test that proves nothing, and the rule that lives only in prose. Paste them into your agent today, against your own repo.

Attack your own spec

Every protection your docs say you have must live somewhere in the code. This one goes looking for each of them, and tells you which ones it cannot find.

```
Act as an adversarial reviewer of my specification, not my
code. Read my steering docs in [directory]. For each
protection, guarantee, or compliance requirement a doc
claims, do three things: (1) state where in the codebase
it would have to be implemented, (2) tell me whether you
can find that implementation, and (3) flag any requirement
that lives only in a planning or design doc and never made
it into a doc the implementer reads at build time. Rank
findings by launch impact. Do not fix anything yet. Give
me the list.
```

Feature tests that do not lie

A test that only proves a 200 came back proves nothing. This one sets the rules first, then makes the agent say what each test covers before you read the diff.

```
Write feature tests for [endpoint/behavior] under these
rules: exercise the real route through the full middleware
stack; fake only at the framework seams and the named
external-service seams (use the existing helpers); never
mock a class defined in this application; every test
asserts response shape and at least one piece of persisted
state; freeze time if any assertion involves a timestamp.
After writing, list each assertion and the behavior it
proves; any test whose list is just "returns 200" must be
strengthened before you show me the diff.
```

Turn your conventions into tests

A rule that lives only in a doc is a rule the agent will get wrong at some point. This one reads your own docs, picks out the rules a test could hold, and writes those tests.

```
Read my steering docs and .ai/guidelines/, and list every
rule stated in prose that could be enforced as a Pest
architecture test (structural facts: finality, strict
types, namespace dependencies, banned classes or
functions, required traits). For each, write the arch()
test in tests/Feature/Conventions/, name it with the
rule's plain-language statement, and run the suite. Where
a rule fails against the current codebase, don't weaken
the rule; show me the violating files and propose fixes.
```

The complete book

What you've just read is the map with three of the prompts. The rest of the book walks the same road with the app open beside it.

25 chapters and 6 appendices, about 103,000 words. 130 prompts with 373 tips and 71 notes. The complete package adds the app itself, with all 31 tags in its history, the prompt library as one file you can work from, both instruction packs and both field guides.

The complete book is at
https://maudev.me/works/the-artisan-and-the-agent/.