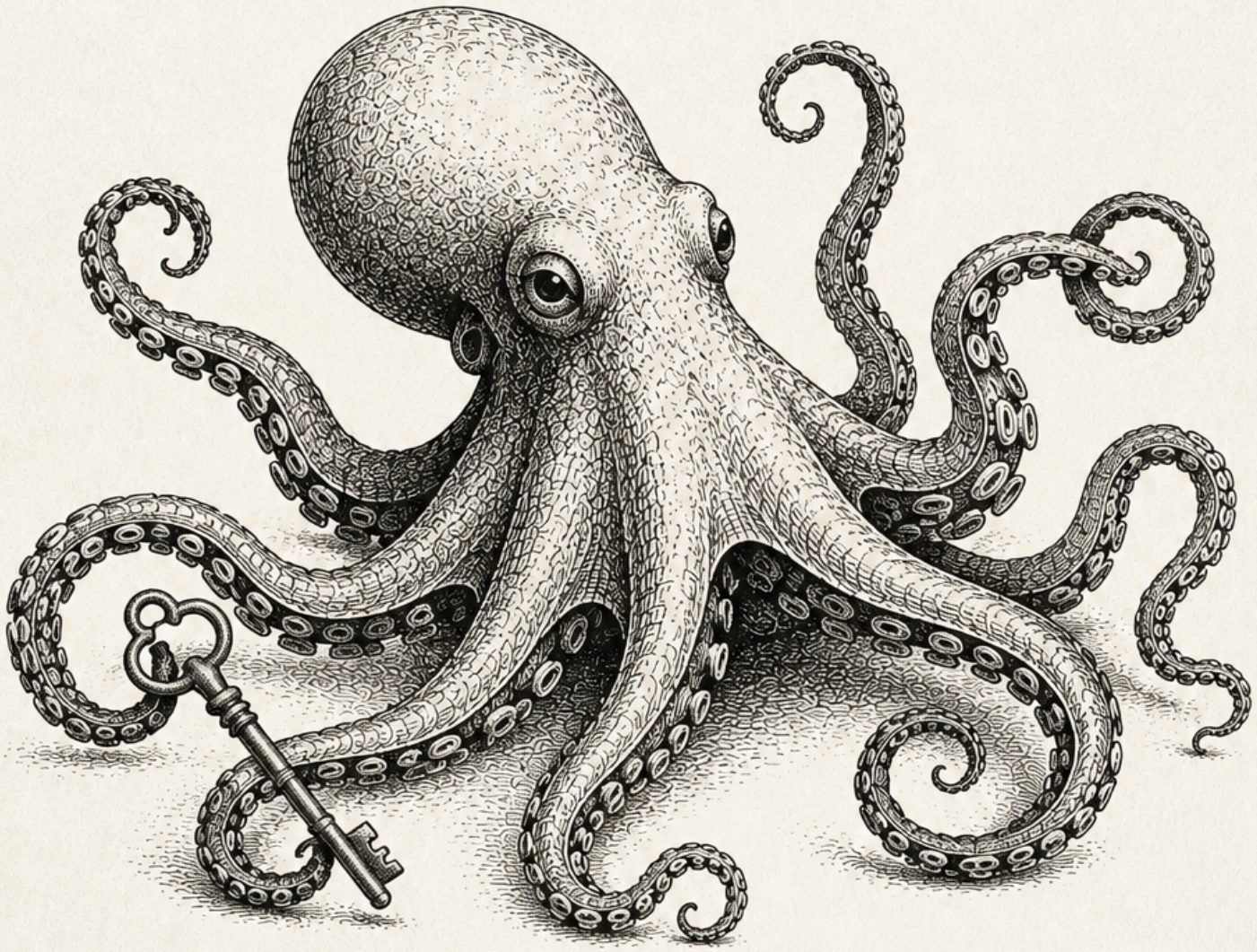
Laravel,
thinking fast



Julian Beaujardin

Table of Contents

Preface **10**

What This Book Is About 10

Who This Book Is For 10

How This Book Is Structured 11

Conventions Used In This Book 11

The Four Principles 11

What Happens When You Ignore Them 12

How I Learned This 13

What "Thinking Fast" Actually Means 13

Chapter 1: Your First AI Endpoint **14**

Start From the Wrong Place, Briefly 14

The Shape That Survives 15

One Driver, All The Provider's Weirdness 16

The Manager, Where the Decisions Live 17

Fail Loudly on a Missing Credential 18

Credentials Belong to a Tenant, Not to the Application 19

Retry What Is Worth Retrying 19

Reaching It From the Application 20

What This Bought 21

What This Does Not Solve Yet 21

Key Principles 21

Chapter 2: Getting an Answer You Can Use **22**

Ask For JSON, and Mean It 22

The Fence It Adds Anyway 23

Decode or Throw, at the Boundary 23
Valid JSON That Ruins Your Afternoon 24
One Schema, Read by Everyone 25
Required Paths Are a Contract — Keep Them Small 26
Constrain the Model Away From What It Cannot Know 27
Keep the Prompt and the Validator in One Room 27
When Validation Fails 28
Key Principles 29

Chapter 3: Prompts Are Code 29
Where a Prompt Lives 29
Compose, Do Not Fork 30
Separate the Copy From the Assembly 31
Bind an Instruction to the Value It Is About 32
Write Rules a Model Can Actually Follow 33
Teach It to Refuse, and Give Refusal a Shape 34
The Prompts You Did Not Write 34
Put the Stable Part First 35
Changing a Prompt Is a Deploy 35
Testing Prose 36
What This Chapter Argued 36

Chapter 4: What a Token Costs You 37
The Bill Has More Lines Than You Think 37
The Cap Is a Server Decision, Made Per Purpose 37
The Cap That Truncates Thinking Too 39
Two Limits, Not One 39
Increment First, Ask Afterwards 40
Key the Ceiling to the Thing That Pays 40
Say No Properly 41

Read the Usage Block 41
Prove the Cache Is Actually Working 42
The Dial Between Cost and Latency 43
Multiply Everything by Iterations 43
What This Chapter Argued 44

Chapter 5: Never Let the Browser Talk to the Model 44
What You Hand Over 45
The Failure That Taught Us 45
Put the Boundary Where the Facts Are 46
Credentials Go in Headers, Never in URLs 46
One Guard, One Answer 47
The Browser Sends Data, Never Instructions 49
Resolve the Paying Credential Server-Side 49
What the Browser Is Still Allowed to Do 50
What This Chapter Argued 50

Chapter 6: Slow Is the Default 51
The Web Request Is the Wrong Place 51
Move the Turn to a Queue 51
Three Timeouts, and Their Order Matters 52
Publish as You Go 53
Checkpoint, Then Recover 54
Let the User Stop 55
Telling the User What Is Happening 56
What This Chapter Argued 57

Chapter 7: Tools — Giving the Model Hands 57
The Description Is the API 58
Type the Input, Then Validate It Anyway 59

Flat Arguments Beat Nested Ones 60
Shape the Payload at the Boundary 60
Normalise the Output, Name the Keys Defensively 61
The Result Is Also a Prompt 62
Order the Toolset 63
What This Chapter Argued 63

Chapter 8: The Agent Loop 64
The Shape 64
stop_reason Is Your Control Flow 65
pause_turn, and Why Order Matters 66
Truncation Must Not Render as an Answer 67
Refusal Is Not an Error 67
The History Is the State 68
Read the Blocks by Type 68
Two Caps, Not One 69
What Happens After the Loop 69
What This Chapter Argued 70

Chapter 9: Tool Results Are Not Answers 70
Branch on Keys, Not on Tool Names 71
Status Messages Are Not Chat Messages 71
Give Every Card an Identity 72
One Card That Updates, Not Five That Stack 72
Rendering Model Text Without an Injection 73
Let the Result Drive the Application, Not Just the View 75
What This Chapter Argued 76

Chapter 10: Side Effects You Cannot Take Back 76
The Instruction Is Not the Guard 76

Persist First, So the Database Is the Guard 77
Single-Use Tokens Must Survive a Rejection 78
Never Hold a Transaction Across a Network Call 79
Roll Back Before You Report 79
Only Commit When It Is Real 80
Idempotency Is a Per-Tool Decision 80
Tell the User What Happened 80
What This Chapter Argued 81

Chapter 11: The Model Is an Untrusted Caller 81
Count Attempts Against the Identity, Not the Guess 82
Two Windows, or You Have Paced Abuse Rather Than Stopped It 82
Do Not Build an Oracle 83
Bindings Made at Issue Time Are Authoritative 84
Assume a Leaked Identifier 85
Some Tools Should Exist and Never Be Offered 86
Reject Usefully 86
What This Chapter Argued 86

Chapter 12: Testing Something That Never Answers Twice 87
Fake at the Boundary You Own 87
Test the Loop, Not the Model 88
Tool Tests Are Feature Tests 88
Test the Compensations, Not the Happy Path 88
Contract Tests Beat Output Tests 89
Record Real Cases as Fixtures 89
What Not to Test 90
What This Chapter Argued 90

Chapter 13: When the Model Is Wrong 91

The Failure: Shipping Position Zero 91
Coverage, Not Cleverness 91
The Experiment That Got Deleted 92
Ties Go to the Provider, and No Match Changes Nothing 92
Deterministic Means Testable 93
Where Else This Shape Applies 93
What This Chapter Argued 94

Chapter 14: Observability for Things You Cannot Reproduce 94
Names and Sizes, Never Inputs 94
One Structured Line Per Round Trip 95
Measure the Hop You Are About to Argue About 95
Log the Question You Will Be Asked 97
What This Chapter Argued 97

Chapter 15: Failure Is a Feature 98
Name Your Dependencies 98
Name What the User Loses 98
Refuse to Start Work You Cannot Finish 99
Stop Asking a Service That Is Down 100
Say What Is Wrong Without Naming the Vendor 100
What This Chapter Argued 100

Chapter 16: Long Content 101
Chunk on Structure, Not on Characters 101
Leave Room for the Instructions 102
Fail the Whole Thing Rather Than Reassemble a Lie 102
Prompt With Names, Not Codes 103
When the Browser Has to Call Directly 103
Clear What Goes Stale Afterwards 104

What This Chapter Argued 104

Chapter 17: Where the AI Belongs **104**

What a Leaf Service Buys 105

What It Costs 105

The Hop Nobody Measured 105

Own the Boundary, Rent the Model 106

Prefer a Star to a Mesh 106

What This Chapter Argued 106

Chapter 18: Shipping It **107**

Pin the Model Version 107

Smoke the Path, Not the Model 107

Alert on Symptoms, Not on the Model 108

Keep a Human Escape Hatch 108

The Six-Month Test 108

What This Chapter Argued 109

Afterword: Knowing the Good **109**

Plausible Is the Failure Mode 109

The Scarce Skill Is Discrimination, Not Production 110

What the Eighteen Chapters Were Actually About 110

The Part I Am Not Going to Pretend About 110

What I Actually Think 111

Appendices **111**

Appendix A: Pre-Launch Checklist 112

Appendix B: Prompt Review Checklist 113

Appendix C: Cost Checklist 113

Appendix D: The stop_reason Table 113

Appendix E: Where to Go Next 114

Preface

Every AI feature demos beautifully.

You wire a model into a controller, type a sentence, and something remarkable comes back. The room goes quiet. Somebody says ship it.

Then it meets production, and you discover that the remarkable part was never the hard part. The hard part is the request that took ninety seconds. The one that came back as prose when you asked for JSON. The one where a user pressed the button eleven times and each press cost you money. The one where the model, asked politely, created the same paid resource twice.

None of those are AI problems. They are the ordinary engineering problems you already know how to solve — timeouts, idempotency, rate limits, trust boundaries, retries — wearing an unfamiliar hat.

That is the whole argument of this book.

The model is the least interesting part of your AI feature.

What This Book Is About

This book is about building AI features in Laravel that survive real users: features that cost what you expected, fail in ways you planned for, and can be handed to another developer without a séance.

It uses a production system as its running example — a platform where a conversational agent gathers what somebody wants, checks it, verifies their email, and provisions a real, live, billable website while they watch. That agent calls tools. Those tools have side effects that cannot be taken back. It has been wrong, it has been slow, and it has been talked into things. Everything in this book was learned by fixing that.

You will not find prompt-engineering folklore here, and you will not find a benchmark of six models. You will find the code around the model: the manager that isolates the provider, the queue job that survives a turn that runs too long, the tool that refuses to create the same thing twice, the logging that tells you what happened when you cannot reproduce it.

Who This Book Is For

This book is for Laravel developers who are comfortable with HTTP, queues, and testing, and who are about to put a model in front of their users — or who already did and are now living with the consequences.

You do not need to know anything about machine learning. Nothing in this book requires it. If you can write a FormRequest, dispatch a job, and fake an HTTP client in a test, you have every skill this book asks for.

You should be suspicious of AI features. That is a qualification, not a flaw.

If you have shipped a chat widget and then watched your provider bill, this book is for you. If you are being asked to add "AI" to a product and you would like the result to still be maintainable next quarter, this book is for you. If your AI feature works and you cannot explain why, this book is very much for you.

How This Book Is Structured

The book moves outward from the single call.

The early chapters build the boring foundation: one endpoint, one provider, structured output you can trust, prompts that live in version control, and caps on what any of it can cost you. The middle chapters give the model hands — tools, agent loops, side effects, and the trust boundaries that keep an agreeable model from becoming an expensive one. The later chapters deal with everything that happens after it works: testing something that never answers twice, deterministic rescue when the model is wrong, observability, degradation, and shipping.

Each chapter is built on code that runs in production today. Where a pattern exists because something broke, the book says what broke.

Conventions Used In This Book

Code follows the same conventions as the rest of my work and the previous book: strict types, explicit return types, FormRequests for validation, Response Wrappers for JSON, DTOs for typed data, and the Manager-Facade-Driver pattern at integration boundaries. Examples target Laravel 13 and PHP 8.4.

The term "provider" means whoever runs the model — the company you send a request to and get tokens back from. "Driver" means your class that talks to one provider. "Tool" means a function you have exposed to the model, and "turn" means everything that happens between a user pressing enter and the feature going quiet again.

Provider APIs change faster than framework APIs. Where this book shows a request shape, treat the shape as the lesson and the field names as a snapshot: check the current documentation before you copy it.

The Four Principles

Everything in this book follows from four ideas. They are not opinions about architecture. They are what is left after the outages.

1. A model is an unreliable network call wearing a costume.

It is a third-party HTTP endpoint with high latency, variable output, occasional 429s, and a bill

attached. Every instinct you have about integrating a flaky payment gateway is correct here. Timeout it. Cap it. Retry the failures worth retrying and none of the others. Assume it fails, and decide now what the user sees when it does.

2. Prompts are code.

A prompt determines behaviour. That makes it code, and code belongs in version control, in review, and in a deploy — not in a database row somebody edits on a Friday, and not in a config file nobody diffs. When a prompt changes, behaviour changes, and you should be able to point at the commit.

3. Cap everything: tokens, wall-clock, iterations, money.

An uncapped AI feature is an uncapped invoice. Every loop needs a maximum number of turns. Every job needs a deadline shorter than its timeout. Every tenant needs a ceiling. Every response needs a token limit chosen by you on the server, not implied by the client. You will not notice the missing cap until the month you do.

4. The model is an untrusted caller.

It can be persuaded. It will be persuaded. Anything you enforce by asking the model nicely is not enforced. Verification, ownership, quotas and permissions live in your database and your middleware, behind the tools — never in the instructions.

What Happens When You Ignore Them

I can describe the failure precisely, because I have watched each part of it.

The provider SDK goes straight into a controller, because that is what the tutorial did. Six weeks later the provider changes a field name and the failure surfaces in four unrelated features at once, because there is no single place where the provider is described.

Nobody caps output tokens, so a response is truncated mid-sentence. There is no check on why it stopped, so the half-sentence renders as though it were a complete answer, and a user reads a confident, wrong, unfinished reply.

The request runs in the web process, because it worked in development. In production it takes ninety seconds, holds a PHP worker the whole time, and dies at the load balancer with a 504 — after the model has been paid for.

The browser calls the AI service directly, because that was one fewer hop. The credential is now in the client, the endpoint accepts anything, and your provider spend is limited only by the patience of whoever finds it.

And somewhere in the middle of all this, a model that has been told "only create one site per user" creates two — because it was asked twice, politely, and nothing but the instruction stood in the way.

Not one of those is an exotic failure. Every one is a Tuesday.

How I Learned This

The first version of our agent worked on the first try, which was the problem.

It was a chat box. You told it what your business was, and it built you a website. In the demo it was genuinely delightful. It asked good questions, it made reasonable choices, and about a minute later there was a real site at a real URL. We showed it to everyone.

The first thing that broke was time. A turn where the model used several tools took well over a minute, and the whole thing ran inside the HTTP request. Users watched a spinner until something upstream gave up, and then saw nothing at all — no error, no partial answer, no site, despite the fact that we had already paid for every token. The work had been done. We just had nowhere to put it.

So we moved it to the queue, and the second thing broke: the job had a timeout, and a long turn hit it. The job was killed mid-turn, having written nothing, while the widget politely polled a cache key that would never appear. The user sat in front of a chat that had simply stopped talking.

Then a turn came back truncated. We had capped output tokens, sensibly, but we never looked at *why* the model stopped. A cut-off reply fell through the same branch as a finished one and rendered as an answer. It read fine. It was wrong.

Then a user pressed the button twice.

The tool that creates a site had been told, in its own description, to only ever do this once. And that instruction was honoured perfectly, right up until two requests arrived close enough together that "once" was ambiguous. We got two provisioning runs for one person. The fix was not a better instruction. The fix was a unique index.

None of these were problems with the model. The model was fine. Every single failure was in the code around it, and every fix was something I already knew how to do — I had just never thought to do it *here*, because this part had a costume on.

That is why the chapters in this book look the way they do. They are not about prompting. They are about queues, deadlines, unique indexes, trust boundaries and structured logs, applied to a component that happens to be non-deterministic.

What "Thinking Fast" Actually Means

The title is not about latency, and it is not about how quickly the model responds.

It means the same thing "shipping fast" meant in the last book: making a decision once, at the right level, so you never have to make it again. Where the provider lives. Where prompts live. What a turn is allowed to cost. What happens when the model is wrong. Decide those once, apply them everywhere, and adding your fifth AI feature costs a fraction of what your first one did.

The alternative is a codebase where every AI feature is a special case with its own retry policy, its

own idea of a timeout, its own credential handling and its own failure mode. That is not fast. That is four features and four outages.

Boring scales. Clever decays. That was true before there was a model in the loop, and the model did not repeal it.

Chapter 1: Your First AI Endpoint

The most valuable thing in this chapter is how unremarkable it is.

We are going to build an endpoint that sends a prompt to a provider and returns what comes back. There is no cleverness in it. What there is instead is a boundary — one place where the provider is described, one contract the rest of the application talks to, and one set of decisions about credentials, timeouts and retries that every future AI feature inherits for free.

Get this wrong and you will pay for it in every chapter that follows. Get it right and most of the remaining chapters become small.

Start From the Wrong Place, Briefly

Here is what almost everybody writes first, and it is worth looking at honestly, because it is not stupid — it is just unfinished.

```
public function generate(Request $request): JsonResponse
{
    $response = Http::withToken(config('services.openai.token'))
        ->post('https://api.provider.example/v1/chat/completions', [
            'model' => 'some-model',
            'messages' => [['role' => 'user', 'content' =>
$request->input('prompt')]],
        ]);

    return response()->json([
        'content' => $response->json('choices.0.message.content'),
    ]);
}
```

This works. On a good day it works every time.

What it lacks is every decision you have not made yet. There is no timeout, so a slow provider holds this worker until PHP gives up. There is no retry, so a single 503 is a user-visible failure. There is no check that the credential exists, so an empty one is sent as an empty bearer and comes back as an

authentication error that blames you for something you did not do. The provider's URL and payload shape are welded to a controller, so the day you add a second provider — or the day this one changes — you will be editing controllers.

And critically: there is nowhere to put the next decision. When you learn something about calling this provider, this shape gives you no place to record it.

The Shape That Survives

The pattern is the one Laravel already uses for every pluggable subsystem it owns — cache, queue, filesystem, mail. A contract that says what the capability is, a driver per provider that implements it, and a manager that builds drivers from configuration.

Three files.

```
app/Services/Ai/AiContract.php      what an AI provider can do
app/Services/Ai/AiManager.php       builds and configures drivers
app/Http/Integrations/Ai/OpenaiAPI.php   one provider
```

The contract comes first, and it should describe *your application's* needs, not the provider's API surface.

```
interface AiContract
{
    /**
     * Generate structured site content as a decoded array.
     *
     * @return array<string, mixed>
     */
    public function getSiteContent(string $model, string $prompt): array;

    /**
     * Generate a single block of text, capped at maxTokens.
     */
    public function getContent(string $model, string $prompt, int $maxTokens):
string;
}
```

Notice what is not in there. There is no `messages` array, no `response_format`, no `temperature`. Those are one provider's vocabulary. If they leak into the contract, the contract is not an abstraction — it is a rename.

The test is simple: could you implement this interface against a completely different provider without changing a single caller? If yes, the boundary is in the right place.

One Driver, All The Provider's Weirdness

The driver is where the provider's actual API lives, and it is the only place that is allowed to know about it.

```
final class OpenaiAPI implements AiContract
{
    use SendsRequests;

    public function __construct(
        protected PendingRequest $request
    ) {}

    /**
     * @return array<string, mixed>
     */
    public function getSiteContent(string $model, string $prompt): array
    {
        $data = $this->send('POST', '/chat/completions', [
            'model' => $model,
            'messages' => [
                ['role' => 'user', 'content' => $prompt],
            ],
            'response_format' => ['type' => 'json_object'],
        ]);

        $content = (string) $data->json('choices.0.message.content');
        $result = json_decode($content, true);

        if (! is_array($result) || $result === []) {
            throw new \RuntimeException(
                'AI site content response could not be decoded into a non-empty
array.'
            );
        }

        /** @var array<string, mixed> $result */
        return $result;
    }
}
```

Two things here are worth more than they look.

The driver receives a configured `PendingRequest`, not a token. It does not know where its

credentials came from, how long its timeout is, or what its retry policy is. That is the manager's job, and it means the driver stays a thin translation layer that is trivial to fake in a test.

The driver's guarantee is transport-level, and it stops there. It promises you decodable, non-empty JSON. It deliberately does *not* validate that the JSON has the keys your feature needs, because the driver has no idea what your feature needs — that check belongs where the caller's context is known, and it gets a chapter of its own. Mixing the two produces a driver that has to be edited every time a feature changes its mind.

That throw matters more than it looks, too. The alternative — returning `null` or `[]` on garbage — pushes an unfalsifiable "did it work?" question into every caller. Fail here, once, loudly.

The Manager, Where the Decisions Live

The manager extends Laravel's own `Manager`, which means driver resolution, caching and custom creators come for free. What you add is configuration and policy.

```
final class AiManager extends Manager
{
    public function getDefaultDriver(): string
    {
        return (string) Config::get('services.ai.default');
    }

    public function createOpenaiDriver(): AiContract
    {
        return $this->buildDriver(
            driverClass: OpenaiAPI::class,
            config: $this->getConfig('openai'),
        );
    }
}
```

Adding a second provider is now genuinely additive: write a driver, add a `createXDriver()` method, add a config block. No caller changes. No controller changes.

And `buildDriver` is where every AI call in the application inherits its manners:

```
protected function buildDriver(string $driverClass, array $config = []):
AiContract
{
    $this->ensureValidDriver($driverClass);

    $token = $config['token'] ?? '';

    if (! is_string($token) || $token === '') {
        throw new InvalidArgumentException(
            message: 'AI driver token is not configured; expected a per-tenant
credential or an environment fallback.',
            code: Response::HTTP_INTERNAL_SERVER_ERROR,
        );
    }

    return new $driverClass(
        request: Http::baseUrl($config['url'] ?? '')
            ->withToken($token)
            ->timeout(Config::integer('services.ai.config.timeout', default:
60))
            ->retry(
                times: max(1, Config::integer('services.ai.config.retry_times',
default: 3)),
                sleepMilliseconds: max(0,
Config::integer('services.ai.config.retry_sleep_ms', default: 500)),
                when: fn (Throwable $e): bool => self::shouldRetry($e),
            )
            ->throw()
    );
}
```

Every AI call in the application now has a timeout, a retry policy and a validated credential, and none of it is repeated anywhere.

Fail Loudly on a Missing Credential

That token check deserves its own section, because skipping it costs a genuinely miserable afternoon.

If the credential is missing and you do nothing, Laravel cheerfully sends `Authorization: Bearer` — the header exists, its value is empty. The provider rejects it with an authentication error whose message says, in effect, *you did not provide an API key*.

Which is true, and completely misleading, because the code that reads the key ran fine and the

config value exists. You will go looking for a revoked key, a wrong environment, a clock skew, a proxy stripping headers. The actual answer is that a per-tenant credential resolved to an empty string and nobody checked.

Fail at the boundary, with a message that names both places the value could have come from. That error should tell the next person exactly where to look, because the next person is you in four months.

This generalises. **A misconfiguration should fail where it is configured, not where it is used.** The further those two points drift apart, the more expensive the bug.

Credentials Belong to a Tenant, Not to the Application

Here is the decision that most first implementations get wrong, and it is very hard to retrofit.

A single application-wide API key means every tenant's usage lands in one bucket. You cannot attribute spend. You cannot cap one noisy tenant without capping all of them. You cannot let a customer bring their own key. And if that key is rotated or rate-limited, every tenant fails at once.

So the config for a driver resolves per tenant, with an environment fallback for the ones who have not brought their own:

```
protected function getConfig(string $driver): array
{
    return (array) Config::get("services.ai.drivers.{$driver}", default: []);
}
```

The important part is not the method — it is that the values behind it are resolved from the authenticated caller's settings before the driver is built, and only fall back to a shared environment value when there is nothing tenant-specific.

Do this on day one. Retrofitting per-tenant credentials into a system that assumed one global key means touching every call site, every test and every cache key, usually under pressure, usually because a bill arrived.

Retry What Is Worth Retrying

Retries are where good intentions produce bad systems. The default instinct — "retry three times on failure" — quietly turns one malformed request into three, and one billing error into three.

The policy that survives contact with a real provider distinguishes between failures that might resolve themselves and failures that definitely will not.

```php
private static function shouldRetry(Throwable $e): bool
{
    if ($e instanceof ConnectionException) {
        return true;
    }

    if ($e instanceof RequestException) {
        $status = $e->response->status();

        return $status === 429 || ($status >= 500 && $status < 600);
    }

    return false;
}
```

Connection failures and provider 5xx are transient — the same request may well succeed a moment later. A 429 is explicit backpressure, and is the one case where the provider has told you it is worth waiting.

Everything else is not retried, and the important member of that set is the 4xx. A malformed request will be malformed on the second attempt too. Retrying it wastes latency, wastes rate limit, and — depending on what you sent and how the provider counts — can waste money. Worse, it hides the bug: an error you retry three times is an error you notice three times more slowly.

The final clause matters as much as the first two. Anything that is not an HTTP failure — a JSON decode error, a type error, a bug in your own mapping code — must not be retried. Those are your problems, and running them again does nothing but delay the stack trace.

Reaching It From the Application

Bind the manager as a singleton and put a facade in front of it, so calling code reads like the rest of Laravel:

```php
final class AiServiceProvider extends ServiceProvider
{
    public function register(): void
    {
        $this->app->singleton(AiManager::class);
    }
}
```

```
$content = AiFacade::getSiteContent(
    model: config('services.ai.model'),
    prompt: $prompt,
);
```

A caller now cannot see which provider answered, cannot construct an unconfigured client, and cannot accidentally skip the retry policy. There is exactly one road, and it is paved.

What This Bought

Take stock, because it is easy to look at three small files and conclude you have written a wrapper for the sake of it.

You now have one place where the provider's API shape lives — one file to edit when they rename a field. One place where credentials are validated, with an error that says where to look. One place where timeouts and retries are decided, applied to every AI call in the application including the ones you have not written yet. One interface for tests to fake, at a boundary that does not move.

And you have somewhere to put the next lesson. That is the part that compounds. Everything the rest of this book teaches — cost ceilings, structured output validation, cancellation, observability — lands cleanly because there is a boundary to land on. Without it, each of those becomes a change to every call site.

The wrapper is not the point. The place to put decisions is the point.

What This Does Not Solve Yet

Be clear about what is still missing, because at this stage the endpoint is honest but naive.

It has no idea what a call costs, and no ceiling on how many a tenant can make. It trusts that the response has the structure the feature needs, which is not yet true. It runs inside the HTTP request, which will not survive a long turn. And its prompt is still a string that somebody assembled somewhere, which is the subject of the next two chapters.

Every one of those is a chapter, and every one of those is a change *inside this boundary* rather than a change across your application. That is the return on the three files.

Key Principles

- **Put the provider behind a contract that describes your needs, not theirs.** If the interface uses the provider's vocabulary, it is a rename, not an abstraction.
- **The driver is a translation layer, and nothing else.** It receives a configured client; it does not build one.
- **Guarantee transport, validate meaning elsewhere.** The driver promises decodable JSON.