🐘 Ship It With PHP



A feature-first builder's guide to Laravel, Symfony, WordPress, TYPO3, API Platform, Nextcloud, and the rest of the modern PHP ecosystem

Someone has handed you a feature, a deadline, and a blank terminal. You already know how to code. You have no wish to learn a new language to prove it. **This book tells you which command to type first**, feature by feature, and what each choice will cost you later.

Foreword: You Don’t Need to Love the Engine to Drive the Car

Nobody asks a delivery driver to rebuild the truck’s engine before their first shift. They get the keys, a route, and a deadline. If the truck breaks down, someone else gets called. Most software work is closer to that than programming books admit.

You have been handed a feature. Maybe a client site that needs a blog and a contact form by Friday. Maybe an internal tool that needs a login screen and a table nobody wants to build by hand. Maybe a proof of concept that has to look real enough to get funded. You did not open this book to fall in love with a programming language. You opened it because something needs to exist, and it happens to run on a language you may not even want named out loud.

That’s fine. This book will barely say its name either. **It shows you, feature by feature, the fastest honest route from “nothing exists yet” to “this works and I can hand it over.”** Sometimes that route runs through a full framework like Laravel or Symfony. Sometimes it runs through a CMS like WordPress or TYPO3 that solved your problem a decade ago. Sometimes the right answer is one small library that does one thing and gets out of your way. And sometimes, when the budget allows, the right answer is to pay someone else to run the hard part.

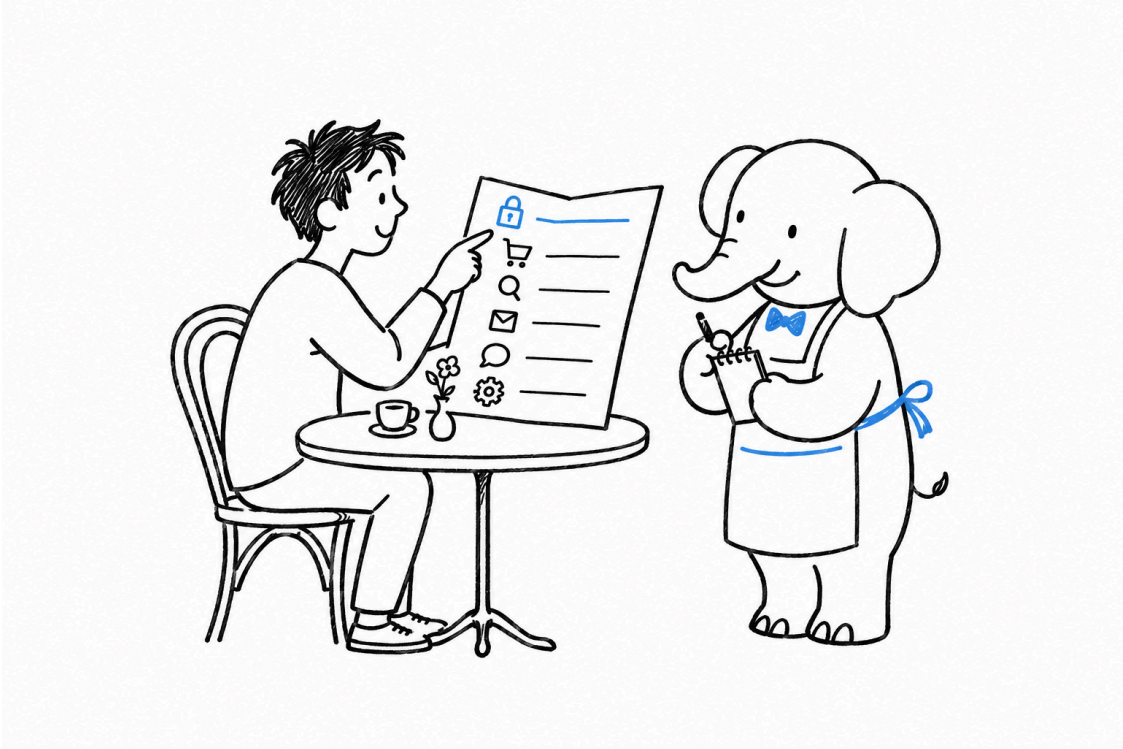
None of that is a compromise. The skill here is picking the tool that ships fastest, holds up under a real client, and doesn’t leave you debugging someone else’s cleverness at 2 a.m. This book teaches that skill as honestly as it can: what each option is good at, what it quietly costs you later, and how to tell the two apart before you commit.

A small aside closes most sections, marked “Under the hood.” It is there for the moment, and it will come, when something these frameworks do for free stops feeling like magic and starts feeling like a question. You can skip every one of them and still ship everything in this book. Or you can read one, get curious, and end up somewhere you didn’t expect: caring, a little, about the engine after all.

The thing you were asked to build is waiting. Turn the page.

How This Book Works

“Add a login screen.” “We need a blog.” “Make search feel instant.” **Every chapter after the first is a request like that, something a client or a product manager would say**, never a piece of syntax. Open the chapter you were asked for and skip the rest. Nobody expects you to read this cover to cover.



The shape of a chapter

A feature chapter opens with a short page that lays out the decision: what the feature really involves, and the two to four ecosystems that can ship it well. Then each ecosystem gets a page of its own, with the same three parts:

- What the tool or platform does for you, in a paragraph.
- A minimal, working example: the command you would run or the code you would write.
- When to reach for this option, and when it is the wrong fit.

Ecosystems come roughly in order, from “fastest to a working demo” to “best fit for something that needs to last.” Treat that order as a guideline, not a ranking. The right pick depends on your client’s budget, on the stack your team already runs, and on how long the thing has to survive after you ship it.

The $ marker

Most of this book is free, open-source software. A few names carry a `$` in their title: paid products or software-as-a-service, not open source. They are here because they are often the fastest or the most reliable path to a specific feature, not because anyone paid to be in this book. Every `$` option sits next to at least one open-source alternative in the same chapter, so you can weigh a subscription against your own time.

“Under the hood” boxes

Most ecosystem pages end with a short aside like this one:

> **Under the hood:** A one- or two-paragraph note on the language feature or engine behavior that makes the convenience above possible. Always optional. Always skippable.

You will ship everything in this book without opening one of these. They exist for the moment your curiosity gets the better of you, and for that moment Appendix C indexes every one of them by chapter.

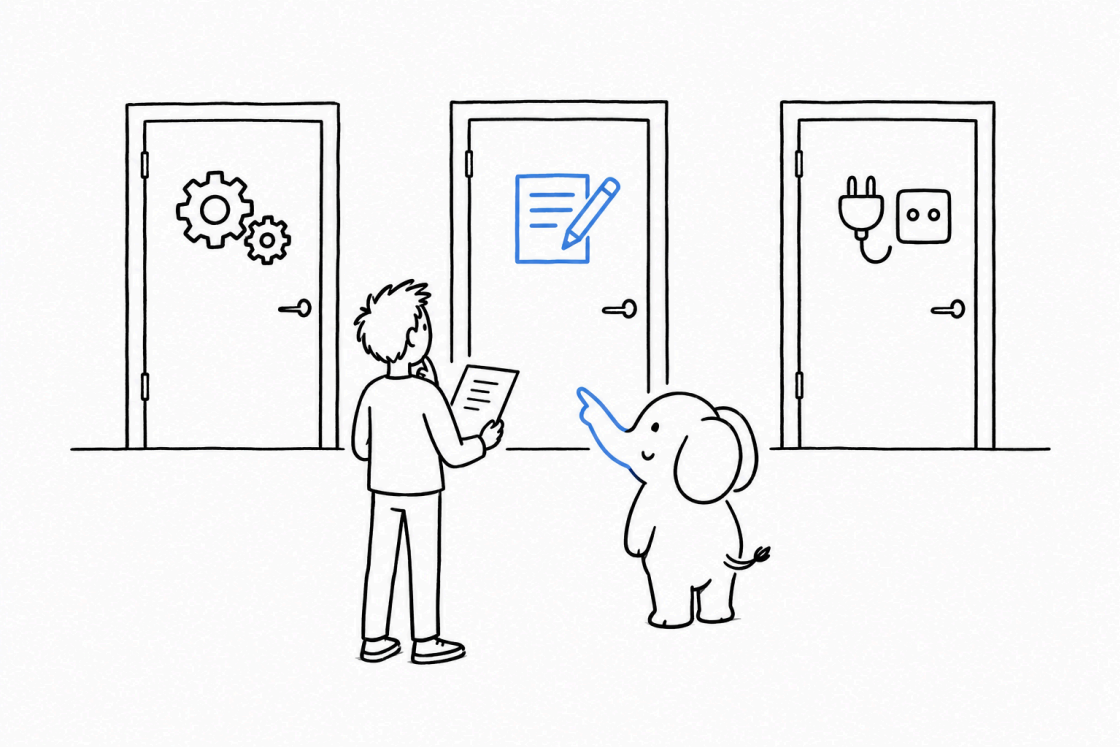
If you get stuck on vocabulary

Appendix B defines the recurring ecosystem terms (ORM, service container, hook, bundle, resource) by what they do for you, not by their textbook definition. A term you don’t recognize is a reason to look there, not to stop reading.

The first decision comes before any feature: what kind of thing are you building?

Choosing Your Stack in Five Minutes

Before any feature ships, one decision shapes everything downstream: what kind of thing you are building, and what already exists that gets you most of the way there. **Get this right and every later chapter becomes a menu you order from.** Get it wrong and you spend six months fighting a tool that was never meant to build what you are building.



This chapter is short on purpose. It gives you a decision guide, not a religion, and a way to have something running in under a minute, so you can feel whether the choice was right before you have spent an afternoon on it.

- Framework, CMS, or Headless: What Are You Actually Building?: the three shapes a project takes, and which one your brief describes.
- Scaffolding a First Project in Under a Minute: the command to run for each main option in this book, so you compare them hands-on instead of from a chart.

Framework, CMS, or Headless: What Are You Actually Building?

Most briefs describe one of three shapes, whether or not the person who wrote the brief knows it.

“We need an application”

A booking system, an internal dashboard, a marketplace: something with custom logic, where the behavior is the product. This is framework territory. Laravel or Symfony gives you routing, a database layer, and a place to put your own logic, and no opinion about what your app does. You write that part.

Reach for a framework when the value of the product is the custom behavior itself, and no off-the-shelf tool already does what the brief describes.

“We need a site people can edit”

A marketing site, a blog, a documentation portal, most small business websites: the value is the content, and non-technical people will add, edit, and reorganize it after launch. This is CMS territory. WordPress or TYPO3 gives you an editing interface, a content structure, and a plugin ecosystem on day one, and you never write an admin panel from scratch.

Reach for a CMS when the client’s real request is “and then our marketing team should be able to update this themselves.” That is most content briefs, whether they say it out loud or not.

“We need an API for something else to consume”

A mobile app, a separate front end, another team’s service: something that feeds a consumer, with no server-rendered page at all. This is headless territory. A tool like API Platform exists to turn a data model into a documented, versioned API without hand-writing every endpoint.

Reach for a headless approach when you already know the consumer is a JavaScript front end, a mobile app, or another backend team, and any page you rendered would be thrown away.

When the brief is both

Plenty of real projects are two of these at once: a marketing site that also needs a booking system, an app that also needs an editable content section. Start from the shape closest to the primary value of the project, and bolt the other capability on afterward. A WordPress site with a custom plugin for the booking logic ships faster than a Laravel app that reimplements a content editor. A Laravel app with a `posts` table and a simple admin screen ships faster than custom booking logic wedged into WordPress hooks.

> **Under the hood:** None of this is about PHP itself. Frameworks, CMSes, and API tools are built from the same language features (a router that matches a URL to code, an ORM that turns rows into objects), packaged around a different assumption about who will use the result and how often the content changes without a deploy.

Scaffolding a First Project in Under a Minute

Reading about a stack tells you what it claims. **Running its scaffolding command for sixty seconds tells you what it feels like.** Here is the fastest path into each of the main options this book covers.

Laravel

```
composer create-project laravel/laravel my-app
cd my-app
php artisan serve
```

A running application at `http://localhost:8000`, with routing, a database connection, and an ORM already wired together.

Symfony

```
composer create-project symfony/skeleton my-app
cd my-app
composer require webapp
symfony server:start
```

The skeleton starts nearly bare. `composer require webapp` pulls in Twig, routing, and the form component, which most sites end up needing anyway.

WordPress

```
wp core download --path=my-site
cd my-site
wp config create --dbname=my_site --dbuser=root --dbpass=
wp core install --url=localhost:8080 --title="My Site" --
admin_user=admin --admin_password=admin --admin_email=you@example.com
php -S localhost:8080
```

Or skip the command line and use Local, which does the same four steps behind a GUI in about the same time.

API Platform

```
composer create-project api-platform/api-platform my-api
cd my-api
docker compose up -d
```

A running API skeleton with an interactive OpenAPI docs page, ready for the moment a single PHP class becomes a full CRUD API.

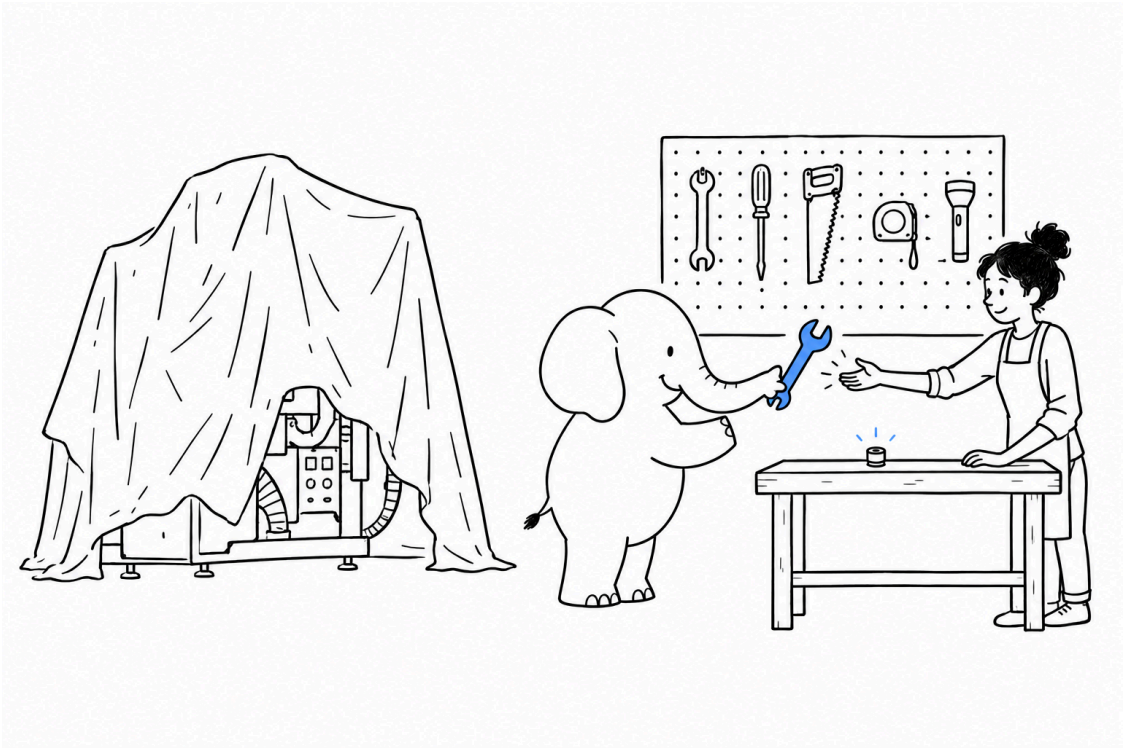
What to compare

Not install time: they are all under a minute. Compare what is already decided for you when the install finishes. Laravel and Symfony hand you an empty application and a lot of freedom. WordPress hands you a working, editable site with nothing custom in it yet. API Platform hands you a fully documented, empty API. The one that feels closest to “done” for your brief is usually the one to keep building on.

> **Under the hood:** All four are Composer projects: PHP’s package manager resolves a dependency tree and generates an autoloader. WordPress is the outlier. It predates Composer’s popularity and can run without it, though WordPress development today almost always uses Composer for plugins and dependencies too.

Shipping Without a Framework: Standalone Components

A one-off import script doesn't need Laravel. A cron job that pings three APIs and writes a report doesn't need Symfony. **Sometimes the fastest way to ship is one `composer require` for the library built for this job, and forty lines around it.**



Every package in this chapter stands on its own: no framework underneath, nothing to configure beyond the one thing it does. Several of them are also the engines inside the frameworks covered later in this book. Once you have used them alone, the "Under the hood" boxes of later chapters read like old news.

- Command-Line Tools: Symfony/Console
- Talking to Other APIs: Guzzle
- Templating Without a Framework: League/Plates
- Logging That Just Works: Monolog
- Validating Input: Respect/Validation
- File Storage Without Buying Into a Framework: League/Flysystem

Command-Line Tools: Symfony/Console

The next time you run `composer require` or `php artisan migrate`, you are using Symfony's Console component. Composer and Laravel's Artisan are both built on it. None of that needs the rest of Symfony. **Console is a standalone package that turns a PHP class into a proper command-line tool**, with arguments, options, colored output, and progress bars, and nothing else attached.

```
composer require symfony/console
```

```php
<?php
// bin/console
require __DIR__.'/../vendor/autoload.php';

use Symfony\Component\Console\Application;
use Symfony\Component\Console\Attribute\AsCommand;
use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputArgument;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

#[AsCommand(name: 'app:import-users', description: 'Import users from a CSV file')]
class ImportUsersCommand extends Command
{
    protected function configure(): void
    {
        $this->addArgument('file', InputArgument::REQUIRED, 'Path to the CSV file');
    }

    protected function execute(InputInterface $input, OutputInterface $output): int
    {
        $rows = array_map('str_getcsv', file($input->getArgument('file')));
        $output->writeln(sprintf('<info>Imported %d rows.</info>', count($rows)));

        return Command::SUCCESS;
    }
}

$app = new Application('My Tool', '1.0.0');
$app->add(new ImportUsersCommand());
$app->run();
```

```
php bin/console app:import-users users.csv
```

When to reach for this

Any job that someone runs by hand or from a cron entry: a data import, a cleanup task, a deploy step. It beats a raw PHP script the moment you need arguments, flags, or output a human has to read and trust.

When it's the wrong fit

If the "script" is the beginning of a web application, or if you are already inside a Laravel or Symfony project, use `artisan` or the app's own `bin/console`. A second command-line entry point next to the first one helps nobody.

> **Under the hood:** Console commands are ordinary PHP classes with a `configure()` and an `execute()` method. There is no special runtime and no compiled binary, just the PHP interpreter you already have, pointed at a script instead of a browser request.

Talking to Other APIs: Guzzle

Almost every feature that "talks to another service" comes down to sending HTTP requests and handling the responses. **Guzzle has been the default way to do that in PHP for over a decade**, and it behaves the same inside a framework and in a forty-line script.

```
composer require guzzlehttp/guzzle
```

```
<?php
require 'vendor/autoload.php';

use GuzzleHttp\Client;

$client = new Client(['base_uri' => 'https://api.example.com']);

$response = $client->get('/users/42', [
    'headers' => ['Authorization' => 'Bearer ' . getenv('API_TOKEN')],
]);

$user = json_decode($response->getBody()->getContents(), true);
echo $user['name'];
```

Guzzle also handles the parts that get tedious by hand: retries, timeouts, streaming large responses, and concurrent requests that don't block on each other in turn.

```
use GuzzleHttp\Promise\Utils;

$promises = [
    'users' => $client->getAsync('/users'),
    'orders' => $client->getAsync('/orders'),
];

$results = Utils::unwrap($promises);
```

When to reach for this

Any script or service that calls a third-party API and isn't already inside a framework with its own HTTP client. Laravel's `Http` facade and Symfony's `HttpClient` component wrap the same ideas, and either is a fine choice if you are already there. Guzzle is the safe default when you don't know yet what the rest of the project will look like.

When it's the wrong fit

Deep inside Laravel, `Http::get(...)` is Guzzle under a friendlier syntax. There is rarely a reason to bypass it and reach for raw Guzzle.

> **Under the hood:** Guzzle and most PHP HTTP clients speak PSR-7 and PSR-18, standard interfaces for HTTP messages and clients. That is why Laravel's `Http` facade, Symfony's `HttpClient`, and Guzzle can hand requests and responses to each other without any of them knowing the others exist.

Templating Without a Framework: League/Plates

The moment a script has to produce HTML instead of a JSON blob or a log line, `echo` statements mixed into the logic become unreadable fast. **Plates is a templating engine written in plain PHP: no new syntax**, just PHP files with layouts, sections, and automatic escaping, usable from any script.

```
composer require league/plates
```

```
<?php
require 'vendor/autoload.php';

use League\Plates\Engine;

$templates = new Engine(__DIR__ . '/templates');

echo $templates->render('profile', ['name' => 'Ada']);
```

```
<?php // templates/layout.php ?>
<!doctype html>
<html>
<head><title><?= $this->e($title ?? 'My Site') ?></title></head>
<body>
    <?= $this->section('content') ?>
</body>
</html>
```

```php
<?php // templates/profile.php ?>
<?php $this->layout('layout', ['title' => 'Profile']) ?>

<?php $this->start('content') ?>
    <h1>Hello, <?= $this->e($name) ?></h1>
<?php $this->stop() ?>
```

`$this->e()` escapes output. It is the same protection Twig or Blade gives you, spelled out instead of hidden behind a different syntax.

When to reach for this

A small internal tool, a report generator, or a script that renders a handful of HTML pages and doesn't justify a full framework's templating stack.

When it's the wrong fit

Once a project grows components, includes, and more than a handful of pages. A framework's own engine (Blade in Laravel, Twig in Symfony) then earns its keep with component reuse and asset compilation, features Plates leaves out on purpose to stay small.

> **Under the hood:** Plates templates are PHP files, so there is no compile step and no template cache to reason about. That simplicity is the whole point of the library, and it is what a framework's own templating engine trades away in exchange for more features.

Logging That Just Works: Monolog

When a script fails outside a browser, with nobody watching, a log line is the only trace it leaves. **Monolog is the logger nearly every PHP framework wraps**: Laravel's `Log` facade and Symfony's `logger` service are both Monolog with a friendlier front end. Use it directly and every project, framework or not, ends up with logs shaped the same way.

```
composer require monolog/monolog
```

```
<?php
require 'vendor/autoload.php';

use Monolog\Level;
use Monolog\Logger;
use Monolog\Handler\StreamHandler;
use Monolog\Handler\RotatingFileHandler;

$log = new Logger('import-job');
$log->pushHandler(new StreamHandler('php://stdout', Level::Info));
$log->pushHandler(new RotatingFileHandler(__DIR__ . '/logs/import.log',
14, Level::Debug));

$log->info('Import started', ['file' => 'users.csv']);

try {
    // ... do the work ...
} catch (\Throwable $e) {
    $log->error('Import failed', ['exception' => $e]);
}
```

One logger writes to several places at once: the console, a rotating file, a service like Sentry (see Catching It in Production). Each destination is one more handler.

When to reach for this

Any script or service that runs outside a request and response cycle: a queue worker, a cron job, a CLI import. If nobody is staring at the terminal when it fails, it needs a log.

When it's the wrong fit

Inside Laravel or Symfony, use the framework's own `Log` facade or `logger` service. They are Monolog underneath, already configured with sensible handlers, and consistent with how the rest of the app logs.

> **Under the hood:** Monolog implements PSR-3, the standard logging interface most PHP frameworks and libraries expect. That is why a library you `composer require` can accept "a logger" in its constructor without caring whether it gets Monolog itself or a framework's wrapper around it.

Validating Input: Respect/Validation

Every form, every API payload, every CSV upload asks the same question: is this data usable? **Respect/Validation answers it with readable, chainable rules**, and needs no framework request lifecycle wrapped around it.

```
composer require respect/validation
```

```php
<?php
require 'vendor/autoload.php';

use Respect\Validation\Validator as v;

$validator = v::key('email', v::email())
    ->key('age', v::intVal()->between(18, 120))
    ->key('username', v::alnum()->length(3, 20));

try {
    $validator->assert([
        'email' => 'ada@example.com',
        'age' => 34,
        'username' => 'ada_l',
    ]);
    echo "Valid.\n";
} catch (\Respect\Validation\Exceptions\NestedValidationException $e) {
    foreach ($e->getMessages() as $message) {
        echo "- {$message}\n";
    }
}
```

Rules read close to plain English and compose freely. `v::stringType()->notEmpty()->length(1, 255)` or `v::arrayType()->each(v::stringType())` cover most of what a form or an API payload needs, without a single custom rule.

When to reach for this

A script or a small service that accepts outside input: a CSV import, a webhook receiver, a lightweight API. Pulling in a whole framework for its validation layer alone would be overkill.

When it's the wrong fit

Once you are inside Laravel or Symfony. `$request->validate()` and the Symfony Validator component are wired into their framework's forms and error display, and are the better default there.

> **Under the hood:** Validation libraries lean on PHP's type system more than they used to. Underneath the fluent `v::intVal()` calls, recent versions describe what "valid" means with PHP's own union types and enums rather than reinventing type checks from scratch.

File Storage Without Buying Into a Framework: League/Flysystem

Reading and writing files sounds simple until "files" means the local disk in development, S3 in production, and FTP for one stubborn client. **Flysystem is a filesystem abstraction: you code against one interface and choose where the files live in configuration.** It is also the library behind Laravel's `Storage` facade (see Laravel: Filesystem Abstraction and S3-Compatible Storage), available here with no framework attached.

```
composer require league/flysystem-aws-s3-v3
```

```
<?php
require 'vendor/autoload.php';

use League\Flysystem\Filesystem;
use League\Flysystem\AwsS3V3\AwsS3V3Adapter;
use Aws\S3\S3Client;

$client = new S3Client([
    'region' => 'us-east-1',
    'version' => 'latest',
]);

$adapter = new AwsS3V3Adapter($client, 'my-bucket');
$filesystem = new Filesystem($adapter);

$filesystem->write('reports/2026-09.csv', $csvContents);
$exists = $filesystem->fileExists('reports/2026-09.csv');
$filesystem->delete('reports/2025-01.csv');
```

Swapping the adapter for a local disk in development is a one-line change:

```php
use League\Flysystem\Local\LocalFilesystemAdapter;

$adapter = new LocalFilesystemAdapter(__DIR__ . '/storage');
$filesystem = new Filesystem($adapter);
```

The rest of the code, `write()`, `fileExists()`, `delete()`, does not change.

When to reach for this

A script, a worker, or a small service that reads or writes files and has to behave the same locally and in production, without “local disk” or “S3” hard-coded into the business logic.

When it’s the wrong fit

Inside Laravel, use the `Storage` facade instead of installing Flysystem separately. It is already there, already configured per environment, and it is the same library with a friendlier API on top.

> **Under the hood:** Flysystem defines a small interface (`write`, `read`, `delete`, `fileExists`, and a handful of others) and lets each adapter implement it however the underlying storage works. That is ordinary interface-based polymorphism, the same idea behind PHP’s built-in interfaces, applied to a problem developers hit often enough to deserve a library.

Shipping a Content Site

Marketing pages, blogs, documentation, portfolios: the value is the content, and someone non-technical will update it after launch. Writing a custom application for this is almost always the wrong call. **Every option in this chapter exists so that you never build an editor, a content model, or a publishing workflow from scratch.**



The four options trade “fastest to a first page” against “best fit for a large, structured, or long-lived site” in different ways.

- WordPress: Block Themes and the Site Editor: the fastest path to an editable site, and the one that runs the largest share of the content web.
- TYPO3: Structured Content at Enterprise Scale: for when “content” means thousands of structured pages across departments, not one blog.
- Statamic ($ for commercial use): A Flat-File CMS Built on Laravel: WordPress-like editing without a database, for Laravel developers.
- Craft CMS ($): A Licensed CMS Built for Editorial Control: a bespoke content model, for clients willing to pay a commercial license to get it.

WordPress: Block Themes and the Site Editor

WordPress still runs a large share of the content web, and the WordPress of today is a different tool from the one its reputation is stuck on. **With block themes and the Site Editor, a client builds and rearranges whole page layouts, headers, and footers visually**, in the same block editor they already use for posts. Common layout changes need no PHP template file at all.

```
wp core download --path=my-site
cd my-site
wp config create --dbname=my_site --dbuser=root --dbpass=
wp core install --url=localhost:8080 --title="My Site" --
admin_user=admin --admin_password=admin --admin_email=you@example.com
wp theme install twentytwentyfive --activate
```

Your job in a block theme is mostly `theme.json`: the color palette, the spacing scale, and the typography the client may choose from. Whatever they rearrange, the site stays on brand.

```
{
  "version": 2,
  "settings": {
    "color": {
      "palette": [
        { "slug": "brand-primary", "color": "#1d4ed8", "name": "Brand
Primary" },
        { "slug": "brand-ink", "color": "#111827", "name": "Ink" }
      ]
    },
    "typography": {
      "fontSizes": [
        { "slug": "small", "size": "0.875rem", "name": "Small" },
        { "slug": "large", "size": "1.5rem", "name": "Large" }
      ]
    }
  }
}
```

When a client needs something the default blocks don't cover, a custom block is registered in PHP and can be as small as a single function.

When to reach for this

Marketing sites, blogs, and small business sites where the client wants to edit the site themselves after launch, and where the plugin ecosystem (forms, SEO, caching) solves problems faster than you could build them.

When it's the wrong fit

A site that is mostly custom application logic wearing a thin content layer. Forcing that into WordPress's content model usually costs more time than it saves.

> **Under the hood:** Block themes store their layout as structured HTML with block comments (`<!-- wp:heading -->`), not PHP template tags. WordPress's own PHP renders those blocks at request time, but the authoring format stays deliberately close to plain markup, which is what lets the visual editor read it back reliably.

TYPO3: Structured Content at Enterprise Scale

Some content sites are not a blog with a few pages. They are a few thousand pages across a dozen departments, in six languages, with an approval workflow before anything goes live. **TYPO3 is a CMS built for that scale**: a content tree, fine-grained permissions, and multilingual support (see Shipping Multi-Language, Multi-Site) designed in from the start, not bolted on.

```
composer create-project typo3/cms-base-distribution my-site
cd my-site
./vendor/bin/typo3 setup
```

Content lives in a page tree. Each page is built from "content elements," structured blocks (text, images, a form, an embedded plugin) that editors arrange without touching code. When the built-in set doesn't cover a client's need, you extend the vocabulary with a custom element:

```
<?php
// Configuration/TCA/Overrides/tt_content.php
use TYPO3\CMS\Core\Utility\ExtensionManagementUtility;

ExtensionManagementUtility::addTcaSelectItem(
    'tt_content',
    'CType',
    ['label' => 'Team Member Card', 'value' => 'teammember_card'],
);
```

Permissions are set per page and per user group. A regional editor manages their department's branch of the tree and cannot touch anyone else's.

When to reach for this

Large institutional or enterprise sites: universities, government agencies, multinational companies. Content ownership is spread across teams, and the editorial workflow is a requirement rather than a nice-to-have.

When it’s the wrong fit

A small business site or a single-editor blog. TYPO3’s structure is the whole value at scale and pure overhead below it. WordPress ships the same result faster for a simpler brief.

> **Under the hood:** TYPO3’s TCA (Table Configuration Array) is a large PHP array that describes how each content type behaves: its fields, their validation, how they render in the backend. It is metadata-driven UI generation, a pattern that shows up again in API Platform’s admin panels, applied to editorial content instead of API resources.

Statamic ($ for commercial use): A Flat-File CMS Built on Laravel

Statamic is what a content editor looks like when a Laravel developer builds it. **Content lives in flat Markdown and YAML files instead of a database**, so version control gives you content history for free. Underneath, it is still a Laravel application, and everything you know about Laravel (see Shipping User Accounts, Shipping Background Work) still applies.

```
composer create-project statamic/statamic my-site
cd my-site
php please make:user
php artisan serve
```

Content types ("collections" and "blueprints" in Statamic's vocabulary) are defined in YAML, and editors get a clean admin interface generated from that definition:

```
# resources/blueprints/collections/posts/post.yaml
title: Post
sections:
  main:
    fields:
      - handle: title
        field: { type: text, required: true }
      - handle: content
        field: { type: markdown }
      - handle: featured_image
        field: { type: assets, container: images }
```

Templates use Statamic's own Antlers syntax or standard Blade. Your choice.

Licensing

Statamic is free for personal, non-commercial projects. A commercial site needs a paid license, priced per site. You can evaluate it for free, but shipping it for a client is a line item in the budget.

When to reach for this

A Laravel-comfortable team building a content-heavy site, one that wants Git-based content history and no separate database for pages and posts.

When it's the wrong fit

A client who needs the WordPress plugin ecosystem specifically, or a project too small for the license cost to make sense.

> **Under the hood:** Storing content as flat files instead of database rows means Statamic leans on PHP's filesystem functions and its caching layer for performance instead of SQL queries. For most content sites, reading a cached, pre-parsed file is at least as fast as a database round trip, and `git diff` comes along as a free audit log.

Craft CMS ($): A Licensed CMS Built for Editorial Control

Craft starts from a different assumption than WordPress: there is no default content model at all. **Every project defines its own fields, entry types, and structure from scratch**, with Craft's editor as the visual layer. The first hour is slower than with WordPress. The following year is faster, for clients whose content does not fit a generic "post."

```
composer create-project craftcms/craft my-site
cd my-site
php craft setup
php craft serve
```

You define the content structure in Craft's admin UI (fields, sections, entry types), then query it in Twig templates:

```
{% for entry in craft.entries()
    .section('caseStudies')
    .client(currentClient)
    .all() %}
    <article>
        <h2>{{ entry.title }}</h2>
        {{ entry.summary }}
    </article>
{% endfor %}
```

Licensing

Craft's core is source-available: you can read and modify the code. Running it commercially requires a paid license per project. A free tier exists for evaluating it on a single-user, non-commercial site.

When to reach for this

Agencies building custom, editorial-heavy sites for clients with specific content needs and a budget for licensing, who would rather have a content model that fits them than one stretched to fit.

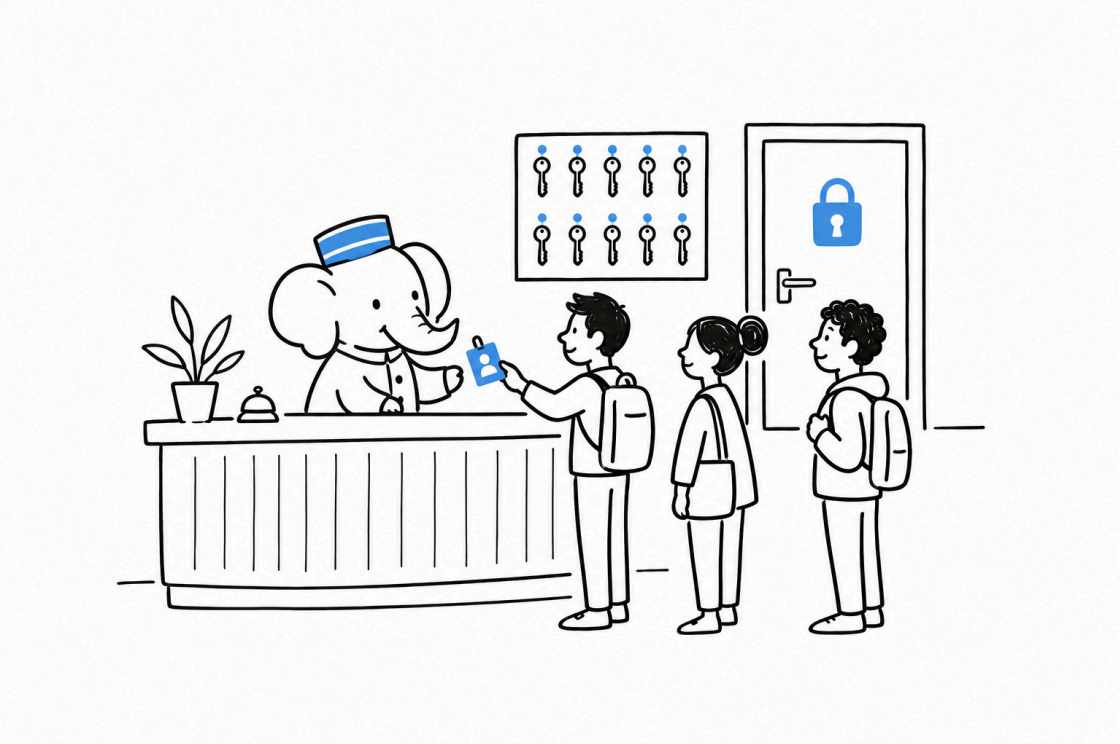
When it’s the wrong fit

A tight budget, a generic blog-shaped brief, or a client who wants WordPress’s plugin marketplace. Craft’s plugin ecosystem is real but far smaller.

> **Under the hood:** Craft’s flexible field system sits on a fairly conventional relational schema. PHP’s dynamic property access and Craft’s own query builder make wildly different content structures feel like first-class, typed data in Twig templates rather than loosely typed arrays.

Shipping User Accounts

Sign-up, login, “forgot password,” and knowing who is looking at the screen right now: nearly every application needs this, and almost none should write it from scratch. **Password hashing, sessions, and the dozen small security details around them are code you want to inherit from a well-maintained package**, not reinvent under a deadline. Timing attacks, session fixation, and rate limiting have all been solved before you started.



- Laravel: Breeze, Fortify, and Jetstream: three official starter kits, from “just the views” to a full team-based SaaS scaffold.
- Symfony: The Security Bundle: authentication and authorization driven by configuration.
- WordPress: Roles, Capabilities, and Application Passwords: what WordPress core already ships, and when it is enough.

Laravel: Breeze, Fortify, and Jetstream

Laravel ships three official answers to “add login,” layered by how much you want handed to you.

Breeze installs view files and controllers into your project. You get working registration, login, password reset, and email verification, as code you own and can edit right away.

```
composer require laravel/breeze --dev
php artisan breeze:install blade
npm install && npm run dev
php artisan migrate
```

Fortify implements the same authentication logic as a backend-only package, with no views. It is meant for a custom frontend or an API.

```
composer require laravel/fortify
php artisan vendor:publish --
provider="Laravel\Fortify\FortifyServiceProvider"
php artisan migrate
```

Jetstream builds on Fortify and adds a full application shell: team management, API tokens, two-factor authentication, and a choice of Livewire or Inertia frontend. It is aimed at SaaS products that need accounts and teams from day one.

```
composer require laravel/jetstream
php artisan jetstream:install livewire
npm install && npm run build
php artisan migrate
```

When to reach for each

Breeze for a straightforward app where you want to see and customize the auth code immediately. Fortify when the frontend is Inertia, a mobile app, or something custom that has no use for Breeze’s bundled views. Jetstream when the product is multi-tenant or team-based and you would otherwise build that structure yourself.

When it’s the wrong fit

A single-page marketing site with no user accounts, or a project where WordPress’s built-in roles (see WordPress: Roles, Capabilities, and Application Passwords) already cover the need without adding a framework.

> **Under the hood:** All three packages lean on Laravel’s `Hash` facade, which defaults to bcrypt or argon2id. Both algorithms are slow on purpose, so that brute-forcing stolen password hashes stays impractical. You never call a hashing function yourself; the authentication guard does it on every login attempt.

Symfony: The Security Bundle

Symfony's approach to authentication is configuration first. **You describe in YAML how users are loaded, how passwords are checked, and which routes need which role, and the framework enforces it on every request.** You never write the enforcement logic yourself.

```
composer require symfony/security-bundle
```

```yaml
# config/packages/security.yaml
security:
  password_hashers:
    App\Entity\User: 'auto'

  providers:
    app_user_provider:
      entity:
        class: App\Entity\User
        property: email

  firewalls:
    main:
      lazy: true
      provider: app_user_provider
      form_login:
        login_path: app_login
        check_path: app_login
      logout:
        path: app_logout

  access_control:
    - { path: ^/admin, roles: ROLE_ADMIN }
    - { path: ^/account, roles: ROLE_USER }
```

The Maker Bundle generates the matching entity, form, and controller in one command:

```
composer require symfony/maker-bundle --dev
php bin/console make:user
php bin/console make:auth
php bin/console make:migration
php bin/console doctrine:migrations:migrate
```

Inside a controller, checking a permission is one line, not a manual session check:

```
class AdminController extends AbstractController
{
    #[IsGranted('ROLE_ADMIN')]
    public function dashboard(): Response
    {
        // only reachable by ROLE_ADMIN users; Symfony enforces it
before this runs
    }
}
```

When to reach for this

Any Symfony application, with no real exception. The Security Bundle is the standard, audited path. Hand-rolling session authentication next to it would only bring back bugs it already fixed.

When it's the wrong fit

An API-only Symfony app that authenticates with tokens instead of sessions still uses the Security Bundle, with a different authenticator (an API token authenticator instead of `form_login`). Less a wrong fit than a different configuration of the same component.

> **Under the hood:** `#[IsGranted]` is a PHP 8 attribute, metadata attached to the method and read by Symfony at runtime through reflection. The same mechanism, attributes plus reflection, lets a single class drive routing, validation, and serialization elsewhere in the framework without repetitive configuration files.

WordPress: Roles, Capabilities, and Application Passwords

WordPress has shipped a complete user and permissions system since long before “auth as a service” was a category. **If the site is built on WordPress, accounts are already solved**, not a feature to add.

Out of the box, WordPress ships five roles: Subscriber, Contributor, Author, Editor, and Administrator. Each carries a set of capabilities (`edit_posts`, `publish_posts`, `manage_options`, and dozens more). Registration, login, and password reset screens already exist at `/wp-login.php`.

When the defaults don’t match a client’s team, custom roles and capabilities take a few lines:

```
add_role('reviewer', 'Reviewer', [
    'read' => true,
    'edit_posts' => true,
    'publish_posts' => false,
]);

if (current_user_can('publish_posts')) {
    // show the publish button
}
```

For headless or API-driven use (a mobile app, a decoupled frontend calling the WordPress REST API), Application Passwords let a user generate a scoped credential without exposing their real login password:

```
curl -u "admin:xxxx xxxx xxxx xxxx xxxx xxxx" \
  https://example.com/wp-json/wp/v2/posts
```

When to reach for this

Any WordPress project. Reimplementing authentication next to WordPress core, instead of extending its role system, is wasted effort and a security liability.

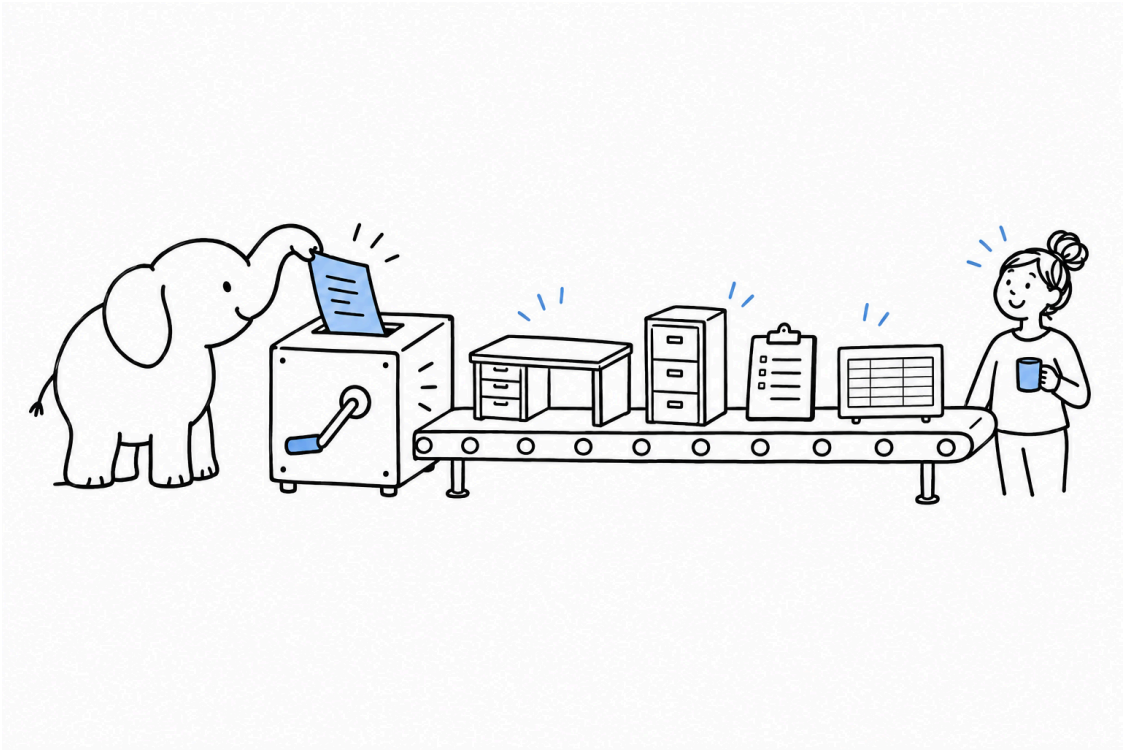
When it’s the wrong fit

A headless setup that wants token-based auth (JWT, OAuth) instead of Application Passwords. That flow is not part of WordPress core, so it needs a dedicated plugin, such as the JWT Authentication plugin.

> **Under the hood:** WordPress capabilities are plain strings checked with `current_user_can()`, stored as serialized PHP data against each role in the database. There is no formal permissions engine, which is why adding a custom capability is so easy: a string in an array, not a schema migration.

Shipping an Admin Back Office

“Can we get a screen to edit the products ourselves?” Sooner or later every application needs one: a place where someone who is not a developer edits records, approves orders, or fixes a typo in a settings table. Building it form by form is one of the least valuable ways to spend a sprint. **The data model you already have describes that screen well enough for a tool to generate it.**



The pick depends on what already describes the data: an Eloquent model, a Doctrine entity, an API resource, or a WordPress post type.

- Laravel: Filament in an Afternoon generates a full admin panel from Eloquent models, with almost no boilerplate.
- Symfony: EasyAdmin and Sonata is the light and the heavy option for a Doctrine project.
- API Platform: An Admin Generated From Your API turns the API you already expose into a panel, for free.
- WordPress: Custom Post Types and ACF as a CRUD Engine shows how far the WordPress admin stretches before you need anything else.
- Laravel Nova ($): The Official, Supported Alternative to Filament trades an open-source panel for first-party support and a roadmap.

Laravel: Filament in an Afternoon

Point Filament at an Eloquent model and it builds the admin resource around it: a list with search and filters, a create form, an edit form, delete actions. **You configure a single PHP class; you do not write the screens.**

```
composer require filament/filament
php artisan filament:install --panels
php artisan make:filament-resource Product --generate
```

The last command inspects the `products` table and pre-fills a resource class. What is left is usually a short list of fields:

```php
use Filament\Forms\Components\TextInput;
use Filament\Forms\Components\Select;
use Filament\Tables\Columns\TextColumn;
use Filament\Tables\Columns\IconColumn;

class ProductResource extends Resource
{
    public static function form(Form $form): Form
    {
        return $form->schema([
            TextInput::make('name')->required(),
            TextInput::make('price')->numeric()->prefix('$'),
            Select::make('category_id')->relationship('category',
'name'),
        ]);
    }

    public static function table(Table $table): Table
    {
        return $table->columns([
            TextColumn::make('name')->searchable(),
            TextColumn::make('price')->money('usd'),
            IconColumn::make('is_active')->boolean(),
        ]);
    }
}
```

That is a searchable, sortable, paginated screen with validated create and edit forms. No HTML, no controller.

When to reach for this

Any Laravel project that needs an internal admin screen and can live with what a schema-driven form and table express. That covers most back-office requests.

When it’s the wrong fit

A customer-facing dashboard with a custom design. A generated panel fights that kind of design instead of speeding it up. Filament is built for internal tools, not product surfaces.

> **Under the hood:** Filament’s form and table builders are fluent, chainable PHP objects (`TextInput::make(...)->required()->numeric()`). Named arguments and first-class enums make that style pleasant to write: one method call configures behavior that used to need several array keys or config options.

Symfony: EasyAdmin and Sonata

Symfony has two established answers, sized for different jobs. **EasyAdmin gets you an admin screen this week; Sonata is for the admin panel that is itself a product.**

EasyAdmin generates a clean CRUD interface from a Doctrine entity with minimal configuration.

```
composer require easycorp/easyadmin-bundle
php bin/console make:admin:dashboard
php bin/console make:admin:crud
```

```
use EasyCorp\Bundle\EasyAdminBundle\Config\Crud;
use EasyCorp\Bundle\EasyAdminBundle\Field\TextField;
use EasyCorp\Bundle\EasyAdminBundle\Field\MoneyField;

class ProductCrudController extends AbstractCrudController
{
    public static function getEntityFqcn(): string
    {
        return Product::class;
    }

    public function configureFields(string $pageName): iterable
    {
        yield TextField::new('name');
        yield MoneyField::new('price')->setCurrency('USD');
        yield AssociationField::new('category');
    }
}
```

Sonata Admin is heavier, older, and more extensible. It asks for more configuration up front, and in exchange it handles complex permission rules, nested admin relationships, and workflows that EasyAdmin was never designed to stretch to.

When to reach for each

EasyAdmin for most internal tools: quick to set up, easy to read, enough for CRUD-shaped needs. Sonata when the admin panel carries complex permissions or deeply nested relationships, and needs more structure than EasyAdmin's simpler model gives you.

When it's the wrong fit

A public, customer-facing dashboard, the same caveat as Filament. Both tools are built for trusted internal users.

> **Under the hood:** Both bundles read Doctrine's entity metadata (the same attributes that define your database schema) to infer field types. A `price` column typed as `decimal` in your entity becomes a numeric field in the admin with no extra declaration, because the framework already knows the type.

API Platform: An Admin Generated From Your API

If you build with API Platform (see Shipping an API Other Teams Can Use), your data is already described well enough for an admin panel. **The panel reads the API's own schema, so there is no admin-specific code to write.**

Take a resource already exposed through the API:

```
#[ApiResource]
class Product
{
    public ?int $id = null;

    #[ApiProperty(description: 'The product name')]
    public string $name;

    public float $price;

    #[ApiProperty]
    public ?Category $category = null;
}
```

API Platform's admin package reads the OpenAPI schema and renders a React-based interface (list, create, edit, delete) against it:

```
npm install @api-platform/admin
```

```
// src/App.js
import { HydraAdmin } from '@api-platform/admin';

export default () => <HydraAdmin entrypoint="https://api.example.com"
/>;
```

No field-by-field configuration is needed for a working panel. You add configuration only where a default needs overriding, such as a custom widget for one field.

When to reach for this

Any project already built on an API Platform backend. The panel is close to free at that point: it reads metadata you produced for the API documentation, instead of asking you to describe the data model a second time.

When it's the wrong fit

A project with no API layer. Standing up an API just to get this generator is a detour, when Filament or EasyAdmin would generate the same panel straight from the database.

> **Under the hood:** API Platform generates a machine-readable OpenAPI (and Hydra/JSON-LD) description of your API from the PHP attributes on your classes. The admin panel is a generic client that renders a UI from *any* API described that way, not something written for your project.

WordPress: Custom Post Types and ACF as a CRUD Engine

Team members, case studies, product listings: a client on WordPress often needs to manage something that is not a blog post at all. The WordPress admin was not designed as a general CRUD generator, but Custom Post Types plus Advanced Custom Fields (ACF) end up working as one. **A management screen for a new kind of content is a registration call, not a new admin.**

```
add_action('init', function () {
    register_post_type('team_member', [
        'label' => 'Team Members',
        'public' => true,
        'show_in_rest' => true,
        'menu_icon' => 'dashicons-groups',
        'supports' => ['title', 'thumbnail'],
    ]);
});
```

That alone adds a list screen to the admin sidebar, with search, bulk actions, and pagination. ACF then adds structured fields through a visual field builder, or from code:

```
acf_add_local_field_group([
    'key' => 'group_team_member',
    'title' => 'Team Member Details',
    'fields' => [
        ['key' => 'field_role', 'label' => 'Role', 'name' => 'role',
'type' => 'text'],
        ['key' => 'field_linkedin', 'label' => 'LinkedIn', 'name' =>
'linkedin', 'type' => 'url'],
    ],
    'location' => [[['param' => 'post_type', 'operator' => '==',
'value' => 'team_member']]],
]);
```

The result is a full create, edit, list and delete interface for team members, built from configuration. The `show_in_rest` flag also exposes it through the WordPress REST API at no extra cost.

When to reach for this

A WordPress site that needs a second kind of structured content beyond posts and pages, for a client already at home in the WordPress admin.

When it's the wrong fit

Relational data with many-to-many relationships and heavy business logic. The post-based model bends a long way, but it stays a content model, not a relational database admin.

> **Under the hood:** Custom Post Types do not create new database tables. Every post type, `post`, `page`, or your own `team_member`, is a row in the same `wp_posts` table, told apart by a `post_type` column. That is why registering a new content type is one function call instead of a migration.

Laravel Nova ($): The Official, Supported Alternative to Filament

Nova is the admin panel built and maintained by the Laravel team itself. Feature for feature it overlaps heavily with Filament: resources generated from Eloquent models, searchable and filterable tables, custom fields and actions. **The difference is who answers when something breaks or a new Laravel version ships.**

```
composer require laravel/nova
php artisan nova:install
php artisan make:nova-resource Product
```

```
class Product extends Resource
{
    public function fields(NovaRequest $request): array
    {
        return [
            ID::make()->sortable(),
            Text::make('Name')->sortable()->rules('required',
'max:255'),
            Currency::make('Price'),
            BelongsTo::make('Category'),
        ];
    }
}
```

Licensing

Nova is sold per developer or per site, with no free tier for commercial use. In exchange you get official support channels, compatibility with new Laravel releases on the same schedule, and a roadmap set by the team that builds the framework.

When to reach for this

Teams or agencies that want a support contract behind their admin panel, or that already pay for other Laravel products (Forge, Vapor, see Shipping to Production) and want one vendor accountable for the whole stack.

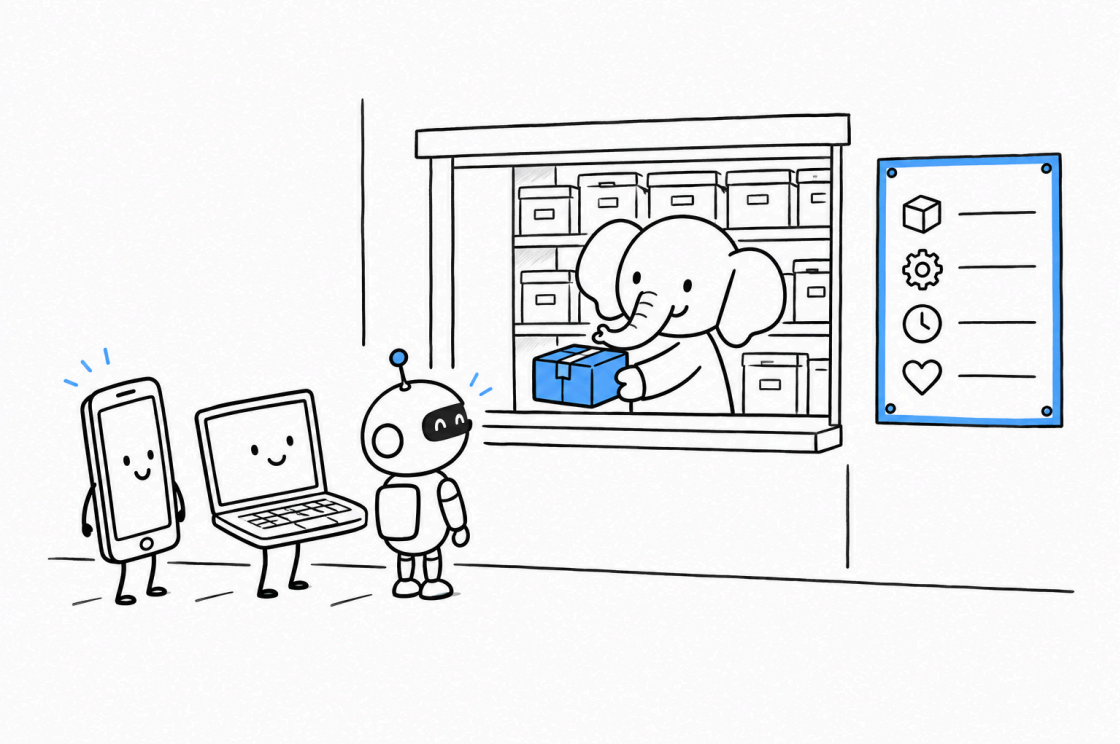
When it's the wrong fit

A tight budget, or a team happy to rely on Filament's open-source community. From the outside, most projects cannot tell the two panels apart.

> **Under the hood:** Being a paid product changes nothing about how Nova integrates with Laravel. It installs through Composer like anything else, with the same service provider and package discovery mechanism as a free package. The license is a business decision layered on top of ordinary PHP package architecture.

Shipping an API Other Teams Can Use

“The mobile team needs to read our data by next sprint.” Someone else always ends up talking to your data: a mobile app, a partner’s system, a front end built by another team. That means end-points, request validation, consistent error responses, and documentation that does not go stale the moment the code changes. **Writing all of that by hand for every resource is the repetitive work the PHP tooling exists to remove.**



The decision is how much you want generated for you, and how much you want to keep explicit.

- API Platform: A Full API From One PHP Class generates REST, GraphQL, and OpenAPI docs from one annotated class.
- Laravel: Sanctum, Resources, and API Versioning is the lighter, more manual, more common approach.
- WordPress: The Built-In REST API is what a WordPress site already exposes, before you install anything.

API Platform: A Full API From One PHP Class

API Platform's whole pitch fits in one example. **One class with one attribute becomes a complete, documented API.**

```php
<?php

namespace App\Entity;

use ApiPlatform\Metadata\ApiResource;
use Doctrine\ORM\Mapping as ORM;

#[ORM\Entity]
#[ApiResource]
class Product
{
    #[ORM\Id, ORM\GeneratedValue, ORM\Column]
    public ?int $id = null;

    #[ORM\Column(length: 255)]
    public string $name;

    #[ORM\Column]
    public float $price;
}
```

That is all. With no further code you get:

- `GET /api/products` and `GET /api/products/{id}`
- `POST /api/products`, `PUT`, and `DELETE`, with request validation
- An interactive OpenAPI/Swagger documentation page at `/api/docs`
- A GraphQL endpoint at `/api/graphql`, if the GraphQL package is installed

Restricting the operations, or adding filters, is configuration on the same attribute rather than new controller code:

```php
#[ApiResource(
    operations: [new GetCollection(), new Get(), new Post(security:
'is_granted("ROLE_ADMIN")')],
)]
#[ApiFilter(SearchFilter::class, properties: ['name' => 'partial'])]
class Product
{
    // ...
}
```

When to reach for this

A project where the API *is* the product, or a data model still evolving fast enough that hand-maintaining matching controllers, serializers, and docs for every change would slow you down.

When it's the wrong fit

An API with one or two endpoints and no plan to grow. There, the conventions and setup of API Platform cost more than writing two controller methods.

> **Under the hood:** PHP attributes are readable at runtime through reflection. API Platform inspects your class, its properties, and their types, then generates routing, validation, and an OpenAPI schema from that single source of truth instead of three separate ones.

Laravel: Sanctum, Resources, and API Versioning

Laravel's default approach is more manual than API Platform's, and that is often the point. **Routes, controllers, and response shapes stay explicit, at the cost of a little more code per endpoint.**

Sanctum handles authentication for SPA and mobile clients (session-based) and for third-party consumers (token-based), from one package.

```
composer require laravel/sanctum
php artisan vendor:publish --
provider="Laravel\Sanctum\SanctumServiceProvider"
php artisan migrate
```

```
// issuing a token
$token = $user->createToken('mobile-app')->plainTextToken;

// protecting a route
Route::middleware('auth:sanctum')->get('/api/user', fn (Request $r) =>
$r->user());
```

API Resources shape a model into a consistent JSON response. Your database columns and your public contract stop being the same thing:

```php
class ProductResource extends JsonResource
{
    public function toArray(Request $request): array
    {
        return [
            'id' => $this->id,
            'name' => $this->name,
            'price_usd' => number_format($this->price, 2),
        ];
    }
}

// in a controller
return ProductResource::collection(Product::paginate());
```

Versioning is usually a route prefix, and deliberately nothing more:

```php
Route::prefix('v1')->group(base_path('routes/api_v1.php'));
Route::prefix('v2')->group(base_path('routes/api_v2.php'));
```

When to reach for this

The default for a Laravel project that exposes an API next to a web app, or a small to medium standalone API where control over each response shape matters more than generation.

When it's the wrong fit

A large, fast-moving data model where a hand-written Resource class per model turns into maintenance. That is the case API Platform is built for.

> **Under the hood:** Sanctum stores API tokens hashed, like passwords, but with PHP's `hash()` function rather than `password_hash()`. Tokens are high-entropy random strings, not user-chosen passwords, so a fast hash is enough of a defense here.

WordPress: The Built-In REST API

Every current WordPress install already exposes a REST API, with no plugin and no configuration. **Posts, pages, media, users, and any post type marked `show_in_rest` can be queried right now:**

```
curl https://example.com/wp-json/wp/v2/posts
curl https://example.com/wp-json/wp/v2/posts/42
curl "https://example.com/wp-json/wp/v2/posts?
search=laravel&per_page=5"
```

The block editor itself runs on this API. It is exercised on every edit of every WordPress site, not bolted on for external consumers.

Custom post types opt in with one flag (see Custom Post Types and ACF as a CRUD Engine):

```
register_post_type('case_study', [
    'public' => true,
    'show_in_rest' => true,
]);
```

Custom endpoints, for anything the post and page shape does not cover, take a few lines:

```
add_action('rest_api_init', function () {
    register_rest_route('myapp/v1', '/stats', [
        'methods' => 'GET',
        'callback' => function () {
            return ['total_posts' => wp_count_posts()->publish];
        },
        'permission_callback' => '__return_true',
    ]);
});
```

Writes (`POST`, `PUT`, `DELETE`) require authentication: Application Passwords for server-to-server use, or a plugin adding OAuth or JWT for third-party client apps.

When to reach for this

A headless setup where WordPress is the content backend for a separate front end (a JavaScript app, a mobile app), or any integration that needs to read existing WordPress content programmatically.

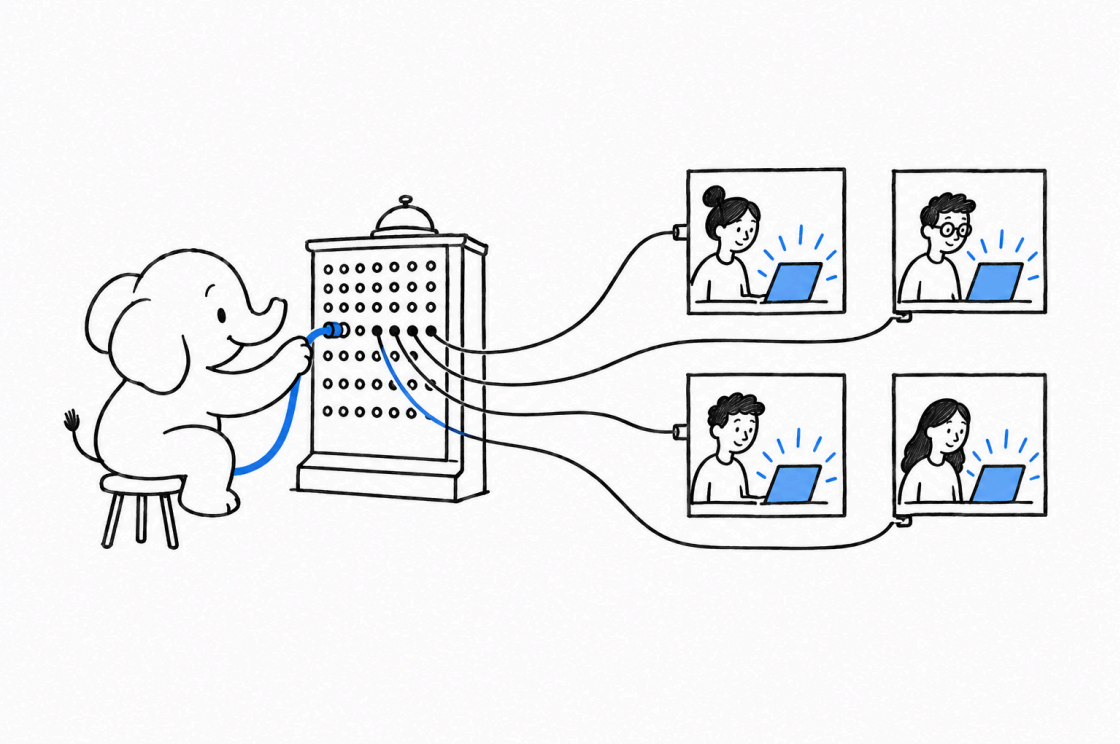
When it's the wrong fit

A data model that looks nothing like posts, pages, or a custom post type. At that point you are fighting a content-shaped API, and a dedicated tool (API Platform, Laravel) fits better.

> **Under the hood:** The REST API's routing and response shaping are built on WordPress's hooks and filters, the same `add_action` / `add_filter` pattern behind themes and plugins. There is no separate API framework underneath: it is the extension mechanism WordPress has used since 2004, pointed at JSON instead of HTML.

Shipping Real-Time Features

A notification that appears on its own, a presence dot, a chat widget, a dashboard number that updates itself: the page just knows something changed, and nobody pressed refresh. PHP's request and response model does not naturally hold a connection open to push updates. **The ecosystem has closed that gap from several directions, and the choice is whether you run the push server yourself or hand that job to someone else.**



- Laravel: Reverb and Livewire Without Writing JavaScript pairs a first-party, self-hosted WebSocket server with a reactive component model.
- Symfony: UX Turbo and Mercure builds on an open real-time protocol rather than a custom server.
- Nextcloud Talk: Real-Time Built Into a Larger Platform is how a large PHP application handles chat and calls at scale.
- Pusher ($): Managed WebSockets Without Running Your Own Server trades infrastructure for a subscription.

Laravel: Reverb and Livewire Without Writing JavaScript

Reverb is Laravel's own WebSocket server, self-hosted and free. **It slots into Laravel's existing broadcasting system, so a third-party service becomes optional.**

```
composer require laravel/reverb
php artisan reverb:install
php artisan reverb:start
```

```
// broadcasting an event from anywhere in the app
broadcast(new OrderShipped($order))->toOthers();

// the event class
class OrderShipped implements ShouldBroadcast
{
    public function __construct(public Order $order) {}

    public function broadcastOn(): Channel
    {
        return new PrivateChannel('orders.' . $this->order->user_id);
    }
}
```

Paired with **Livewire**, the front end often needs no JavaScript at all. A Livewire component listens for the broadcast and re-renders itself:

```php
class OrderStatus extends Component
{
    public Order $order;

    #[On('echo-private:orders.{order.user_id},OrderShipped')]
    public function refreshStatus(): void
    {
        $this->order->refresh();
    }

    public function render()
    {
        return view('livewire.order-status');
    }
}
```

The Blade template renders `$order->status` and nothing else. When the event fires, Livewire re-renders that fragment over the WebSocket connection, with no client-side state written by hand.

When to reach for this

A Laravel project that wants notifications, live status updates, or simple chat without adopting a separate front-end framework or paying for a WebSocket service.

When it's the wrong fit

A JavaScript-heavy front end (a canvas editor, a complex single-page app), where Livewire's round trip to the server adds latency a client-side framework would not. Reverb still works there as the transport. Livewire is the piece to reconsider.

> **Under the hood:** Reverb is built on PHP Fibers, introduced in PHP 8.1. A single PHP process can hold many open connections and switch between them cooperatively instead of blocking. That is what makes a persistent WebSocket server practical in PHP, a job the one-request-per-process model was never suited for.

Symfony: UX Turbo and Mercure

Symfony's real-time story is built on Mercure, an open protocol rather than a Symfony invention. **Updates reach the browser over plain HTTP, as Server-Sent Events, so any client can subscribe, not only a Symfony front end.**

```
composer require symfony/mercure-bundle
# Mercure hub runs as a small standalone binary, or via Docker
```

```
use Symfony\Component\Mercure\HubInterface;
use Symfony\Component\Mercure\Update;

class OrderController
{
    public function ship(Order $order, HubInterface $hub): Response
    {
        $order->markShipped();

        $hub->publish(new Update(
            "orders/{$order->getId()}",
            json_encode(['status' => 'shipped']),
        ));

        return new Response('OK');
    }
}
```

Turbo, through the `symfony/ux-turbo` package, subscribes the page to that update and swaps the matching HTML fragment. One data attribute replaces the hand-written JavaScript:

```
<turbo-stream-source src="{{ mercure('orders/' ~ order.id) }}"></turbo-stream-source>

<div id="order-{{ order.id }}-status">
    {{ order.status }}
</div>
```

When the server broadcasts a matching Turbo Stream update, Turbo replaces that `<div>` in place. No reload, no custom client code.

When to reach for this

A Symfony project that wants live updates without a separate JavaScript framework, and in particular one where an open, HTTP-based protocol matters for infrastructure or compliance reasons.

When it's the wrong fit

High-frequency, two-way traffic: live multiplayer interaction, a trading dashboard with sub-second updates. Server-Sent Events are one-way by design, and a WebSocket connection fits that job better.

> **Under the hood:** Server-Sent Events are a plain HTTP feature: a long-lived response that keeps sending chunks over time. Mercure builds a full publish/subscribe hub on that one primitive, which is why it passes through ordinary HTTP infrastructure (proxies, load balancers) that a raw WebSocket sometimes needs special configuration for.

Nextcloud Talk: Real-Time Built Into a Larger Platform

Nextcloud Talk is a chat and video calling system at production scale, built as an app inside a PHP application rather than as a product from scratch. You may never touch Nextcloud's code, and it still teaches something. **"PHP can't do real-time" was never true; the trick is knowing what to keep in PHP and what to hand off.**

Talk installs as an app inside any Nextcloud instance:

```
php occ app:install spreed
php occ app:enable spreed
```

For chat, Talk relies on the same polling and Server-Sent Event techniques as the rest of this chapter. For audio and video calls at scale, it hands off to a dedicated High-Performance Backend, a separate signaling server, instead of forcing media through PHP's request lifecycle:

```
php occ talk:signaling:add wss://signaling.example.com "shared-secret-here"
```

That split is the pattern to borrow. PHP owns accounts, permissions, and chat history. A dedicated service owns the media stream.

When to reach for this

Not as a package you install into your own project, but as a reference architecture the day your real-time feature starts wanting video or audio and not only data updates: keep PHP as the source of truth for accounts and permissions, and delegate the media transport to a service built for it.

When it's the wrong fit

If the question is whether to self-host Nextcloud for your organization (see Shipping File Storage and Collaboration), Talk is one of the apps that comes with it, not a separate decision.

> **Under the hood:** Nextcloud's app system lets a feature like Talk be installed, updated, and disabled independently of the core, using PHP's own package and autoloading conventions. It is the same instinct as WordPress plugins or Symfony bundles: a stable core, and functionality layered on as discrete, swappable units.

Pusher ($): Managed WebSockets Without Running Your Own Server

Pusher does one thing: it runs the WebSocket infrastructure for you. **A feature that would need Reverb or Mercure running as a service you operate becomes an API call and an npm package.**

```
composer require pusher/pusher-php-server
```

```
$pusher = new Pusher\Pusher(
    getenv('PUSHER_KEY'),
    getenv('PUSHER_SECRET'),
    getenv('PUSHER_APP_ID'),
    ['cluster' => 'us2', 'useTLS' => true],
);

$pusher->trigger('orders-channel', 'order.shipped', [
    'order_id' => $order->id,
    'status' => 'shipped',
]);
```

Laravel's broadcasting system was built with Pusher as its original target, and it still swaps in for Reverb with the same event classes and the same `broadcast()` call. Only a config value changes:

```
# .env
BROADCAST_CONNECTION=pusher
```

Pricing

Pusher bills by concurrent connections and messages per day. The free tier covers development and small production apps; past that, the monthly cost grows with traffic.

When to reach for this

A team that does not want to operate WebSocket infrastructure at all, or a project with traffic unpredictable enough that a managed, auto-scaling connection layer beats self-hosting Reverb or a Mercure hub.

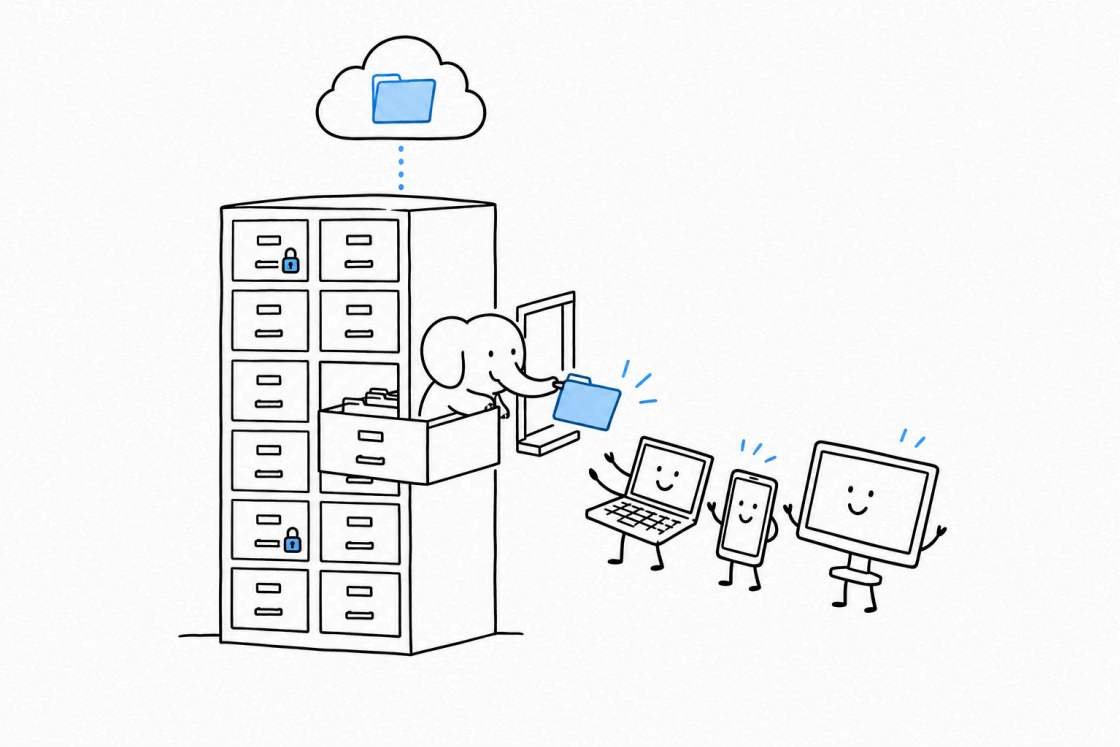
When it's the wrong fit

Predictable, modest real-time traffic. Reverb and Mercure are free and self-hosted, and cost nothing beyond the server you are probably already running.

> **Under the hood:** Because Laravel's broadcasting abstraction was designed against Pusher's API shape, swapping Pusher for Reverb is a configuration change, not a code change. The lesson generalizes: a good abstraction is what makes "self-host it" versus "pay someone to run it" a decision you can defer, or reverse, without a rewrite.

Shipping File Storage and Collaboration

“We need our own Dropbox.” Uploads, sharing links, folder permissions, version history: the request comes up more often than its scope suggests, and the honest answer is rarely to build a Dropbox. **Either a ready-made platform is the right call, or the request is really just file uploads in an existing app.**



- Nextcloud: A Ready-Made Drive, Not a DIY Project stands up a full file-sharing platform in an afternoon.
- Laravel: Filesystem Abstraction and S3-Compatible Storage adds uploads and storage to an application you already have.
- WordPress: The Media Library at Scale is what is already built in, and what breaks first as it grows.

Nextcloud: A Ready-Made Drive, Not a DIY Project

When the request is “we need our own Dropbox,” the pragmatic answer is almost never to build one. **Nextcloud is a complete self-hosted file sync and sharing platform, written in PHP, and it deploys today.** Desktop and mobile sync clients, sharing links with expiration and passwords, version history, a full permissions model: all of it is already there.

```
docker run -d -p 8080:80 \
  -v nextcloud:/var/www/html \
  nextcloud
```

For a production setup with the reverse proxy, database, and Redis cache already wired, the All-in-One Docker image (see Shipping to Production) is the officially recommended path.

Once it runs, provisioning users and setting quotas is a command-line job, with no custom code:

```
php occ user:add jane.doe
php occ user:add-app-password jane.doe --group=editors
php occ files:external:create "shared-drive" local null::null -c
datadir=/mnt/shared
```

When a client needs something the built-in feature set does not cover, you write an app in Nextcloud’s plugin system, the same architecture behind Nextcloud Talk, instead of forking the platform.

When to reach for this

A request that is truly “our own file storage and sharing system,” and especially a client whose data residency or privacy requirements rule out Google Drive and Dropbox.

When it’s the wrong fit

A feature that amounts to “let users upload a profile picture” or “attach a PDF to this order.” That is ordinary upload handling (see Laravel: Filesystem Abstraction and S3-Compatible Storage), not a

reason to stand up a platform.

> **Under the hood:** Nextcloud's sync relies on chunked uploads and ETags to detect what changed since the last sync, so a client transfers only the parts of a large file that changed rather than the whole file again. That problem was solved once, in PHP, and no app built on top has to solve it again.

Laravel: Filesystem Abstraction and S3-Compatible Storage

Laravel's `Storage` facade wraps League/Flysystem behind a disk-agnostic API. **The same code writes to the local disk in development and to S3 in production, and configuration alone decides which.**

```
// config/filesystems.php
return [
    'disks' => [
        's3' => [
            'driver' => 's3',
            'key' => env('AWS_ACCESS_KEY_ID'),
            'secret' => env('AWS_SECRET_ACCESS_KEY'),
            'region' => env('AWS_DEFAULT_REGION'),
            'bucket' => env('AWS_BUCKET'),
        ],
    ],
];
```

```
use Illuminate\Support\Facades\Storage;

// storing an uploaded file
$path = $request->file('avatar')->store('avatars', 's3');

// generating a temporary, signed download link
$url = Storage::disk('s3')->temporaryUrl($path, now()->addMinutes(10));

// reading it back
$contents = Storage::disk('s3')->get($path);
```

Switch `'driver' => 's3'` to `'driver' => 'local'` for development, or point it at a DigitalOcean Spaces or Cloudflare R2 endpoint in production. Nothing else in the application changes.

When to reach for this

Any feature with user uploads: avatars, attachments, generated PDFs, exported reports. This is the boring, correct way to handle files in a Laravel app, and boring is what you want here.

When it's the wrong fit

A request for sharing, sync, and collaboration (see Nextcloud) rather than upload and retrieve. `Storage` handles files. It gives you no sharing links, no version history, no sync client.

> **Under the hood:** `temporaryUrl()` generates a pre-signed URL, a link with a cryptographic signature and an expiration baked into the query string, which S3 validates itself without any request reaching your Laravel app. Serving a private file this way costs your server no bandwidth at all.

WordPress: The Media Library at Scale

The Media Library handles uploads, thumbnail generation, and a searchable grid out of the box, with no plugin. **For most content sites, that is the whole feature, already done.**

```
$attachment_id = media_handle_upload('file', $post_id);
$url = wp_get_attachment_url($attachment_id);
$thumb = wp_get_attachment_image_url($attachment_id, 'medium');
```

The first scaling problem is almost always local disk space and bandwidth, once a site holds thousands of images. The standard fix offloads storage to S3-compatible object storage through a plugin, and editors keep the same interface:

```
wp plugin install amazon-s3-and-cloudfront --activate
wp media-offload run
```

From then on, uploads go through the same screen, and files are stored on and served from object storage instead of the server's disk.

The second problem is image processing itself. Generating several thumbnail sizes for every upload is CPU-heavy at volume, and the usual answer is to let a CDN resize on the fly instead of pre-generating every size at upload time.

When to reach for this

Any WordPress site with more than a handful of images, which is nearly all of them. Even a small brochure site gets the automatic thumbnail sizes and the searchable grid for free.

When it's the wrong fit

Video hosting, or a media-heavy application at real scale. A dedicated media platform integrated through its API serves that better than a Media Library stretched far past its design.

> **Under the hood:** WordPress generates thumbnail sizes at upload time with PHP's GD or Imagick extension, whichever the server has. Resizing eagerly rather than lazily is why large libraries eventually need the offloading described above: the cost is paid once per upload, but paid by every upload.

Shipping a Storefront

Products, a cart, a checkout, and real money changing hands without a credit card number ending up somewhere it should not. Taxes, refunds, inventory, and payment compliance all carry risk when handled casually. **E-commerce is the area where "just build it yourself" is almost never the pragmatic answer.**



- WooCommerce: E-Commerce on Top of WordPress is the fastest path when the site is, or should be, WordPress.
- Sylius: A Symfony-Based E-Commerce Framework fits a custom storefront with business logic of its own.
- Laravel: Cashier and Stripe ($) for Custom Checkouts fits a Laravel app that needs a subscription or a checkout, not a store.

WooCommerce: E-Commerce on Top of WordPress

When a WordPress site (see Shipping a Content Site) needs to sell things, WooCommerce is the default answer. **It adds a full store as a plugin, inside the admin your client already knows:** products, variations, cart, checkout, orders, coupons.

```
wp plugin install woocommerce --activate
wp wc --version
```

Products are a custom post type, so everything you know about managing WordPress content carries over:

```
$product = new WC_Product_Simple();
$product->set_name('Wireless Mouse');
$product->set_regular_price('29.99');
$product->set_stock_quantity(50);
$product->save();
```

Payments plug in through gateway extensions rather than custom integration code:

```
wp plugin install woocommerce-gateway-stripe --activate
```

Custom logic (a pricing rule, a loyalty program, an external inventory system) hooks into WooCommerce's own actions and filters, the same extension pattern as WordPress core:

```
add_filter('woocommerce_product_get_price', function ($price, $product)
{
    if (is_user_logged_in() && current_user_can('wholesale_customer'))
{
        return $price * 0.85;
    }
    return $price;
}, 10, 2);
```

When to reach for this

A storefront on a site that is already WordPress, or a client whose need is closer to “a content site that also sells a modest catalog” than to a high-volume commerce platform.

When it’s the wrong fit

High transaction volume with custom checkout logic, where the WordPress architecture starts fighting both performance and customization. That is Sylius territory.

> **Under the hood:** WooCommerce products live in the same `wp_posts` table as any other content, with product data (price, stock) in `wp_postmeta`. It is the “everything is a post” architecture from Custom Post Types, stretched to model a whole catalog.

Sylius: A Symfony-Based E-Commerce Framework

Sylius takes the opposite approach from WooCommerce. Instead of a plugin on a content platform, it is an e-commerce framework built from Symfony components. **It is aimed at stores whose business logic a generic plugin architecture would fight rather than support.**

```
composer create-project sylius/sylius-standard my-shop
cd my-shop
symfony console sylius:install
symfony server:start
```

Because it is Symfony, extending it means writing ordinary Symfony code (events, services, Doctrine entities) rather than learning a commerce-specific plugin API:

```
class OrderPlacedListener
{
    public function __construct(private LoyaltyPointsCalculator
$calculator) {}

    public function onOrderPlaced(ResourceControllerEvent $event): void
    {
        $order = $event->getSubject();
        $this->calculator->awardPoints($order->getCustomer(), $order-
>getTotal());
    }
}
```

```
services:
    App\EventListener\OrderPlacedListener:
        tags:
            - { name: kernel.event_listener, event:
sylius.order.post_create, method: onOrderPlaced }
```

That is the event-listener pattern used throughout Symfony (see Shipping Background Work), applied to commerce events.

When to reach for this

A storefront with custom pricing rules, order workflows, or B2B logic: tiered pricing, approval workflows, multi-warehouse fulfillment.

When it's the wrong fit

A plain catalog with standard checkout needs. That is far more setup, and more Symfony to know, than WooCommerce asks for the same result.

> **Under the hood:** Sylius is assembled almost entirely from reusable Symfony components (the Form component, the Workflow component for order state machines, Doctrine for persistence), the same building blocks as Chapter 2, put together for commerce rather than invented from scratch.

Laravel: Cashier and Stripe ($) for Custom Checkouts

Not every “we need to take payments” request is a storefront. Often it is one subscription plan, a one-time checkout, or usage-based billing bolted onto an app that is not a store at all. **Cashier is Laravel’s official wrapper around Stripe (and, in a separate package, Paddle), built for that case.**

```
composer require laravel/cashier
php artisan vendor:publish --tag="cashier-migrations"
php artisan migrate
```

```
class User extends Authenticatable
{
    use Billable;
}

// starting a subscription
$user->newSubscription('default', 'price_monthly_pro')
    ->create($paymentMethodId);

// checking access
if ($user->subscribed('default')) {
    // grant access
}

// a one-time checkout, no subscription
return $user->checkout(['price_one_time_item']);
```

Webhooks (payment succeeded, subscription canceled, card expired) land on a route Cashier registers for you. Your database stays in sync with Stripe without polling.

Pricing

Stripe charges a percentage plus a fixed amount per successful charge; that is the `$` in the title. Cashier itself, the Laravel package, is free and open source.

When to reach for this

A Laravel app that needs subscriptions, one-time payments, or usage billing, without a product catalog. SaaS pricing tiers, not a shopping cart.

When it's the wrong fit

A catalog with inventory, variations, and shipping. That is WooCommerce or Sylius territory. Cashier has no notion of a product beyond a Stripe price ID.

> **Under the hood:** Cashier verifies each incoming Stripe webhook with a cryptographic signature check against a shared secret, so the request is known to come from Stripe and not from a spoofer before its payload is trusted to update a subscription in your database.

Shipping Search That Feels Instant

“Can search find it even if I misspell it?” A search box backed by `WHERE title LIKE '%query%'` answers no. It has no typo tolerance, no relevance ranking, and it scans the whole table once the data grows. **Search that feels instant is a solved problem, and the job is to wire your data into a tool that solves it, not to build one.**



- Laravel: Scout With Meilisearch or Algolia ($): a search index kept in sync with your models, on a server you run or one you rent.
- TYPO3: Solr and Elasticsearch Integration: search for thousands of pages that respects permissions and languages.
- WordPress: Search Plugins and When to Reach for Elasticsearch: what is built in, and the two plugins that replace it when it is not enough.

Laravel: Scout With Meilisearch or Algolia ($)

Scout adds full-text search to Eloquent models. **Every time a model is saved, updated or deleted, Scout updates the search index for you**, so the synchronisation code you would otherwise write does not exist.

```
composer require laravel/scout meilisearch/meilisearch-php
```

```
class Product extends Model
{
    use Searchable;

    public function toSearchableArray(): array
    {
        return ['name' => $this->name, 'description' => $this-
>description];
    }
}

// searching
$results = Product::search('wireles mouse')->get();
```

The typo in “wireles” still returns the mouse. Typo tolerance and relevance ranking are the search engine’s job, and a SQL `LIKE` never did either.

Meilisearch is open source and self-hosted: a single binary with sensible defaults.

```
docker run -p 7700:7700 getmeili/meilisearch
```

Algolia is the managed, paid alternative. Same Scout driver, no infrastructure to run.

```
composer require algolia/scout-extended
```

```php
// config/scout.php
return [
    'driver' => env('SCOUT_DRIVER', 'algolia'),
];
```

Pricing

Meilisearch is free to self-host. Algolia bills by search volume and records indexed, with a free tier for small projects. The same Scout code works against either, so the choice comes down to running it yourself or paying someone else to.

When to reach for this

Any Laravel app with a search box over more than a trivial amount of data: a product catalog, a documentation site, a directory of listings.

When it's the wrong fit

A search need simple enough that an indexed column and a basic `LIKE` query do the job. Adding a search engine there is infrastructure without a problem.

> **Under the hood:** Scout's driver system is a plain PHP interface; Meilisearch and Algolia drivers both implement the same handful of methods (`update`, `delete`, `search`). Swapping between them is a config change because Scout was designed against that interface rather than either engine's API.

TYPO3: Solr and Elasticsearch Integration

A large TYPO3 site (see Structured Content at Enterprise Scale) has the search problem dedicated engines exist for. Thousands of pages, several languages, and editors who expect results to respect the same permissions and page tree as the rest of the site.

The Solr extension indexes the page tree with its access restrictions, so a search never surfaces a page a visitor is not allowed to see.

```
composer require apache-solr-for-typo3/solr
```

```
plugin.tx_solr {
  solr {
    host = solr.example.com
    port = 8983
    scheme = https
  }
  search {
    faceting = 1
    faceting.facets {
      contentType {
        field = type
      }
    }
  }
}
```

Faceted search, where visitors filter results by content type, department or date, is configuration once Solr is connected. No custom PHP.

When to reach for this

Enterprise TYPO3 sites where search correctness matters to the organisation: multilingual content, page permissions, faceted filtering. Not a nice-to-have search box.

When it’s the wrong fit

A smaller TYPO3 site where the built-in indexed search extension is enough. A separate Solr or Elasticsearch cluster is infrastructure the project does not need yet.

> **Under the hood:** The Solr integration mirrors TYPO3’s own permission model when indexing, storing which user groups can see each document alongside its content. That is what keeps search from becoming an accidental way to leak restricted content, a detail generic search integrations sometimes miss.

WordPress: Search Plugins and When to Reach for Elasticsearch

WordPress's built-in search is a `LIKE` query against post titles and content, and it shows. No relevance ranking to speak of, no typo tolerance, and it slows down once a site has thousands of posts.

The fastest fix is a plugin that replaces the built-in query without touching template code.

```
wp plugin install relevanssi --activate
```

Relevanssi reindexes existing content and takes over the search query. You get relevance scoring and fuzzy matching, with no code changes to speak of.

For larger sites, or ones that need faceted filtering (category and price range together, for instance), an Elasticsearch-backed plugin swaps the search backend entirely:

```
wp plugin install elasticpress --activate
wp elasticpress index --setup
```

```
$args = [
    's' => 'wireless mouse',
    'post_type' => 'product',
];
$query = new WP_Query($args); // ElasticPress intercepts this
transparently
```

The application code barely changes. `WP_Query` works as before, and ElasticPress answers it from Elasticsearch instead of MySQL.

When to reach for this

Relevanssi for most sites where the complaint is search quality, not scale. ElasticPress once the site has enough content, or complex enough filtering, that even a smarter SQL query will not be fast or flexible enough.

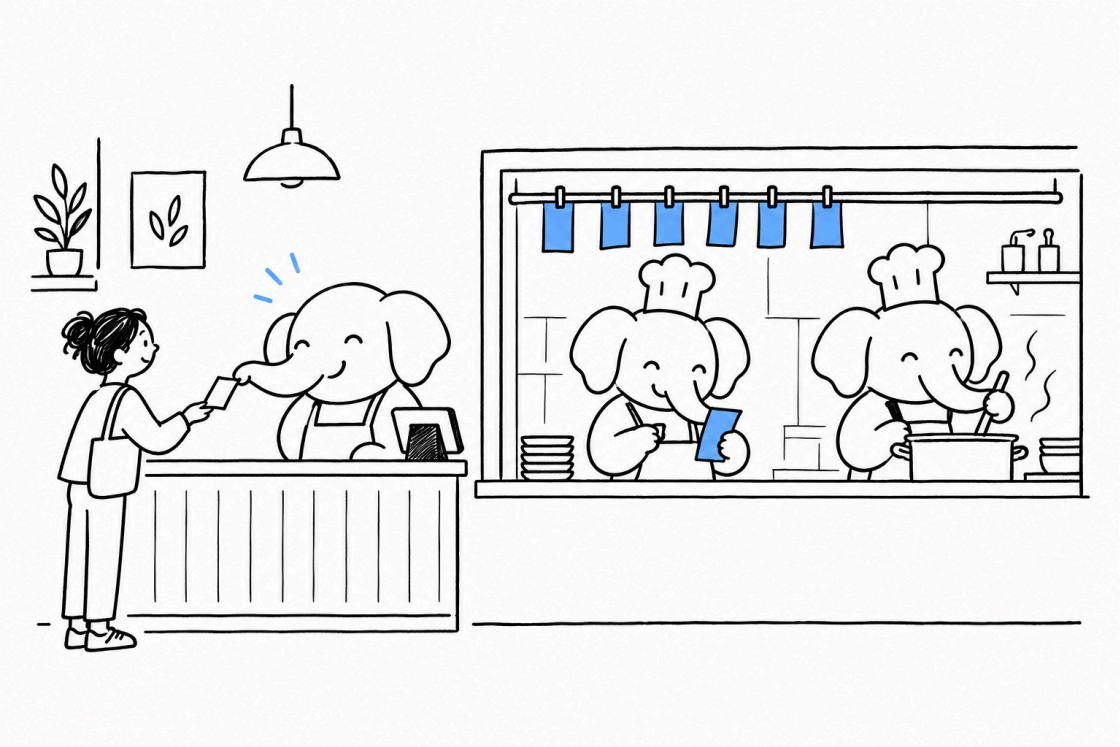
When it’s the wrong fit

A small site with a few dozen pages, where the default search is fast enough and nobody has complained. Do not add search infrastructure ahead of a problem.

> **Under the hood:** `WP_Query` is WordPress’s central abstraction for “get me some posts,” used by nearly every theme and plugin. ElasticPress hooks into that same class and reroutes its query at the last moment, which is why adopting it rarely means rewriting theme code.

Shipping Background Work

Send the confirmation email later. Resize the uploaded image later. Generate the monthly report at 2 a.m. with nobody watching. Anything that should not block the user's request, or that runs on a schedule rather than in response to one, is background work. **Doing it well means a queue and a worker process, not a `sleep()` call and hope.**



- Laravel: Queues and Horizon: jobs on a queue, a worker to run them, and a dashboard to watch them.
- Symfony: The Messenger Component: one message bus for background jobs and for talking between services.
- WordPress: WP-Cron and the Action Scheduler: what WordPress uses for scheduled tasks, its well-known quirk, and the fix.

Laravel: Queues and Horizon

Any slow or non-essential piece of a Laravel request can become a Job, pushed onto a queue and run by a separate worker process. The user's request finishes at once instead of waiting on it.

```
php artisan make:job SendWelcomeEmail
```

```
class SendWelcomeEmail implements ShouldQueue
{
    use Queueable;

    public function __construct(private User $user) {}

    public function handle(): void
    {
        Mail::to($this->user)->send(new WelcomeEmail($this->user));
    }
}

// dispatching it
SendWelcomeEmail::dispatch($user);
```

```
php artisan queue:work
```

That last command is the worker. A process manager such as Supervisor keeps it running; it pulls jobs off the queue (Redis, in most production setups) and executes them.

Horizon adds a dashboard on top of Redis queues: throughput, failed jobs, retry controls and per-queue metrics, with no separate monitoring tool.

```
composer require laravel/horizon
php artisan horizon:install
php artisan horizon
```

Failed jobs retry with backoff. The ones that fail for good land in a `failed_jobs` table for inspection instead of vanishing.

When to reach for this

Anything that should not block a web request: sending email, processing an upload, calling a slow third-party API, generating a report. If a user would notice the delay, it belongs on a queue.

When it's the wrong fit

Work that must complete before the response, such as validating a form before saving it. Queuing that adds latency and complexity where synchronous code was already correct.

> **Under the hood:** Jobs are serialized (usually with PHP's native `serialize()`) before being stored in the queue, which is why job classes should only hold simple, serializable properties like a model ID rather than large objects or open resources; the worker process deserializes and re-constructs the job from scratch when it runs.

Symfony: The Messenger Component

Messenger is Symfony's message bus. **The same abstraction handles background jobs, like Laravel's queues, and messaging between parts of an application or between separate services**, through one interface.

```
composer require symfony/messenger
```

```php
final class SendWelcomeEmail
{
    public function __construct(public readonly int $userId) {}
}

#[AsMessageHandler]
final class SendWelcomeEmailHandler
{
    public function __invoke(SendWelcomeEmail $message): void
    {
        $user = $this->users->find($message->userId);
        $this->mailer->send(new WelcomeEmail($user));
    }
}

// dispatching it, anywhere in the app
$this->bus->dispatch(new SendWelcomeEmail($user->getId()));
```

```yaml
# config/packages/messenger.yaml
framework:
  messenger:
    transports:
      async: '%env(MESSENGER_TRANSPORT_DSN)%'
    routing:
      App\Message\SendWelcomeEmail: async
```

```
php bin/console messenger:consume async
```

Messages and handlers only meet through the bus. The same `SendWelcomeEmail` message can later be routed to a different transport, say a queue shared with another service, by changing configuration rather than application code.

When to reach for this

Any Symfony application that needs background processing, and especially one that may one day need service-to-service messaging. The same component handles both, without a second tool.

When it's the wrong fit

A single small script that does one slow thing now and then. A full message bus for one cron job is more ceremony than the problem needs; a plain Console command on a schedule is enough.

> **Under the hood:** Messenger routes messages to handlers using PHP's type system: the `#[AsMessageHandler]` attribute plus the type-hint on `__invoke()` tells Symfony which messages a handler accepts, so adding a new message type is a matter of defining a class, not registering it in a lookup table by hand.

WordPress: WP-Cron and the Action Scheduler

WP-Cron is WordPress's built-in scheduler, and it has a quirk to know before you rely on it. **It does not run on a system timer: it checks for due tasks every time a visitor loads a page.** A site with no traffic can quietly stop running its scheduled tasks on time.

```
add_action('daily_report_hook', function () {
    // generate and email the report
});

if (!wp_next_scheduled('daily_report_hook')) {
    wp_schedule_event(time(), 'daily', 'daily_report_hook');
}
```

The standard fix, for anything that matters, is to disable the page-load trigger and call it from a system cron job instead:

```
// wp-config.php
define('DISABLE_WP_CRON', true);
```

```
# real crontab entry, running every 5 minutes
*/5 * * * * curl https://example.com/wp-cron.php?doing_wp_cron
>/dev/null 2>&1
```

Beyond simple scheduled hooks, and especially for queued work such as bulk emails or an import, the **Action Scheduler** library adds a proper queue with retries and logging on the same idea. It ships with WooCommerce and works standalone.

```
as_schedule_single_action(time(), 'process_import_batch', ['batch_id'
=> 42]);

add_action('process_import_batch', function ($batch_id) {
    // process it
});
```

When to reach for this

WP-Cron with a system crontab for anything time-sensitive: scheduled reports, subscription renewals, cache warming. Action Scheduler once you need retries, batching, or a way to see whether a background task succeeded.

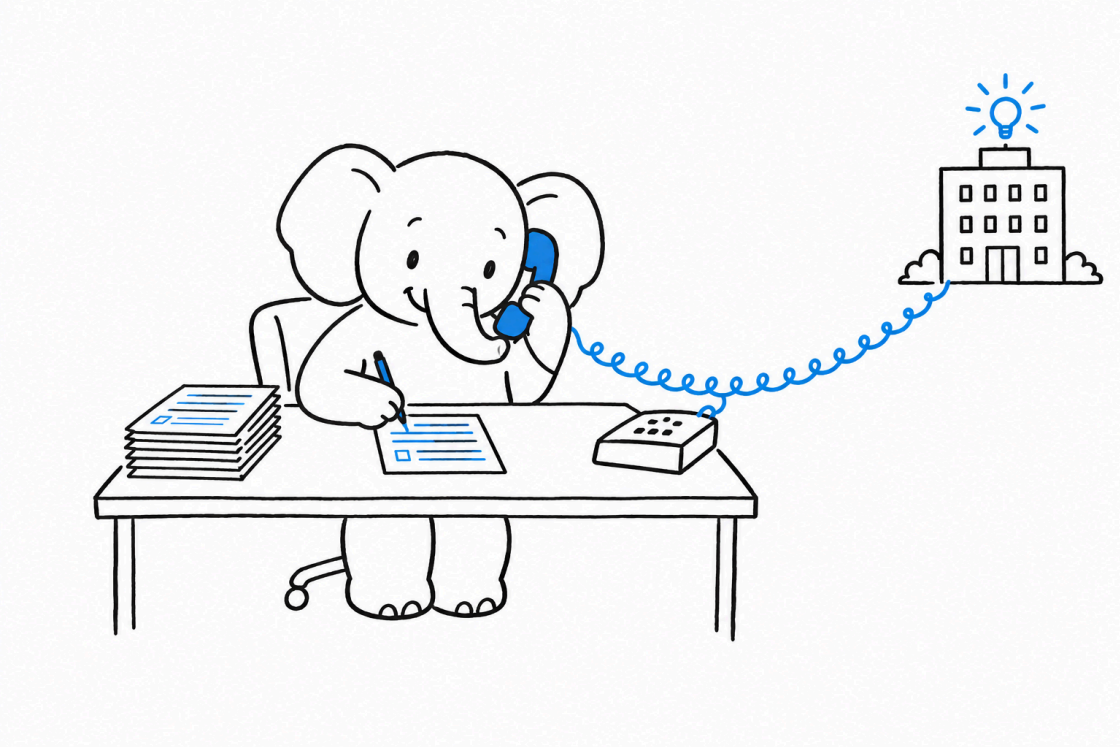
When it's the wrong fit

High-frequency or high-reliability background processing. Laravel's queues and Symfony Messenger were built around a queue backend from the start, and stand on firmer ground than a page-load trigger.

> **Under the hood:** WP-Cron's "check on every page load" design comes from shared WordPress hosting that historically did not let users configure system cron jobs. It is a reasonable workaround for that constraint rather than a design flaw, and one to override the moment your hosting allows real cron.

Shipping an AI Feature This Sprint

“Can we add a chatbot?” “Can search understand plain language?” “Can it summarise this for the user?” AI features arrive as small bolt-on requests far more often than as a rewrite of the product. **Adding one to an existing PHP app almost never means running a model. It means calling an API and handling the answer well.**



- Laravel: Prism and the OpenAI/Anthropic PHP Clients: one API for several model providers, swappable in a line.
- Symfony: The AI Bundle: the same idea in Symfony’s own conventions, configured rather than constructed.
- WordPress: AI Plugins and When to Call an API Instead: the fastest path for a WordPress site, and where it stops being enough.

Laravel: Prism and the OpenAI/Anthropic PHP Clients

Prism is a Laravel-native package for talking to large language models (OpenAI, Anthropic and others) behind one API. **A feature built against one provider is not locked to it.**

```
composer require prism-php/prism
```

```
use Prism\Prism\Prism;
use Prism\Prism\Enums\Provider;

$response = Prism::text()
    ->using(Provider::Anthropic, 'claude-sonnet')
    ->withPrompt("Summarize this support ticket in two
sentences:\n\n{$ticket->body}")
    ->generate();

echo $response->text;
```

Comparing cost or quality between providers for the same feature is a one-line change:

```
Prism::text()->using(Provider::OpenAI, 'gpt-4o')->withPrompt($prompt)-
>generate();
```

When the result has to go back into your database rather than onto a screen, ask for structured output. You give a schema instead of parsing text by hand:

```
$response = Prism::structured()
    ->using(Provider::Anthropic, 'claude-sonnet')
    ->withSchema($ticketClassificationSchema)
    ->withPrompt("Classify this ticket: {$ticket->body}")
    ->generate();

$category = $response->structured['category'];
```

For a thinner layer, the official `openai-php/client` and `anthropic-php` packages call each provider's API directly. They fit a feature that only ever needs one provider.

When to reach for this

A specific, bounded feature (summarisation, classification, a support chatbot, content suggestions) added to an app that is otherwise a normal Laravel product.

When it's the wrong fit

A feature where the model must act on live, private data at request time, which an API call alone cannot provide. That usually points to a retrieval step first (search your own data, then put it in the prompt), not to a reason to avoid the approach.

> **Under the hood:** Prism's provider abstraction is a PHP interface, the same pattern behind Flysystem's storage adapters: one contract, several interchangeable implementations, so your application code depends on the contract rather than a specific vendor's SDK.

Symfony: The AI Bundle

Symfony's AI Bundle brings the "talk to a model behind a clean interface" idea into Symfony's own conventions. **The agent is configured, not constructed by hand, so the model provider never gets hard-wired into your business logic.**

```
composer require symfony/ai-bundle
```

```
# config/packages/ai.yaml
ai:
  platform:
    anthropic:
      api_key: '%env(ANTHROPIC_API_KEY)%'
  agent:
    support_assistant:
      model: 'claude-sonnet'
      system_prompt: 'You are a concise, friendly support assistant.'
```

```
final class TicketSummaryController
{
    public function __construct(private AgentInterface
$supportAssistant) {}

    public function summarize(Ticket $ticket): Response
    {
        $result = $this->supportAssistant->call(
            new UserMessage("Summarize this ticket:\n\n{$ticket-
>getBody()}"),
        );

        return new JsonResponse(['summary' => $result->getContent()]);
    }
}
```

Swapping the model or provider for a feature is a YAML change. In tests, Symfony's own tooling (see Shipping Confidence) substitutes a fake agent, and no real API is ever called.

When to reach for this

A Symfony project adding an AI feature, where consistency with the app's configuration and dependency injection matters, or where the team wants an easy way to swap providers or mock the call in tests.

When it's the wrong fit

A tiny script or a non-Symfony project. There, the bundle's configuration layer is more setup than calling the provider's HTTP API with Guzzle.

> **Under the hood:** The bundle's `AgentInterface` is, again, ordinary interface-based dependency injection: Symfony's service container hands your controller whatever concrete agent implementation is configured, the same mechanism it uses to inject a database connection or a logger.

WordPress: AI Plugins and When to Call an API Instead

The fastest AI feature you will ever ship on WordPress is one you do not write. Plugins already exist for the common requests: a writing assistant in the block editor, automatic alt text for images, an on-site chatbot trained on the site's own content.

```
wp plugin install ai-engine --activate
```

Most of these plugins hold your own API key for a provider (OpenAI, Anthropic) and wrap it in a settings screen and an editor integration. They do not reinvent the model access.

For something no plugin covers, a small custom plugin can call the provider's API directly. WordPress's own HTTP API does the job in a few lines, no separate library:

```
add_action('save_post_product', function ($post_id) {
    $description = get_post_field('post_content', $post_id);

    $response = wp_remote_post('https://api.anthropic.com/v1/messages',
[
        'headers' => [
            'x-api-key' => get_option('anthropic_api_key'),
            'content-type' => 'application/json',
        ],
        'body' => wp_json_encode([
            'model' => 'claude-sonnet',
            'max_tokens' => 100,
            'messages' => [['role' => 'user', 'content' => "Write a
one-line SEO summary for: {$description}"]],
        ]),
    ]);

    $data = json_decode(wp_remote_retrieve_body($response), true);
    update_post_meta($post_id, 'ai_seo_summary', $data['content'][0]
['text']);
});
```

When to reach for this

An existing plugin for the well-trodden requests, writing assistance or alt text. A small custom plugin with `wp_remote_post()` for anything specific to the site's own content model.

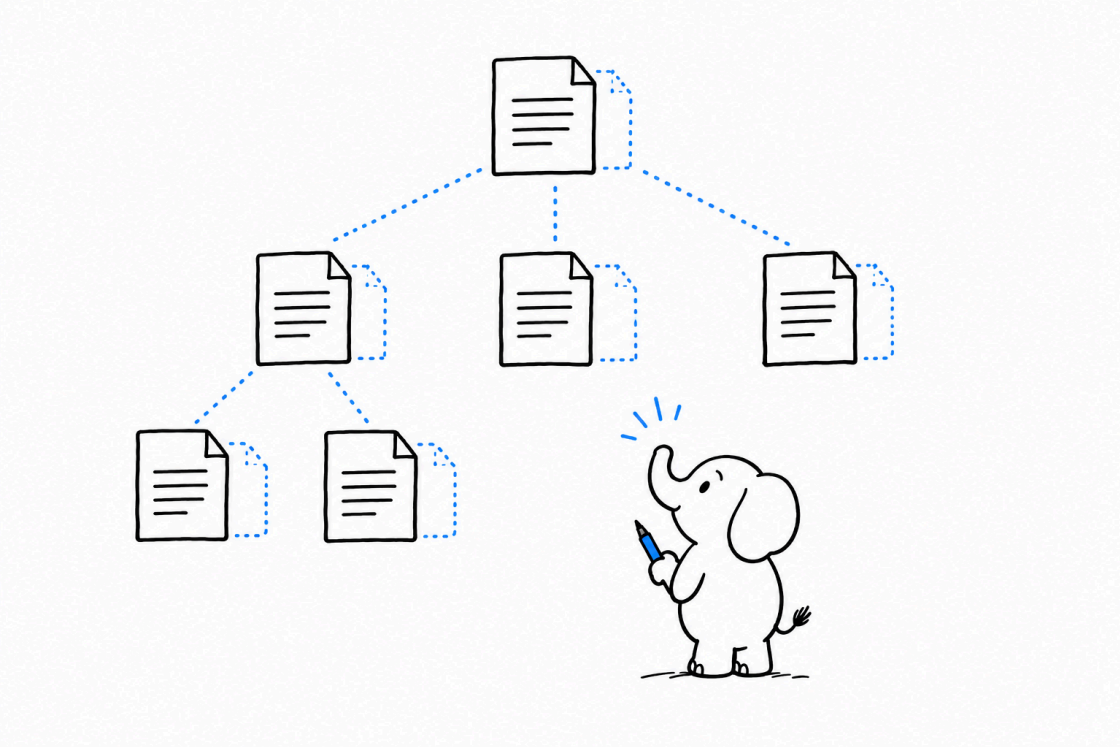
When it's the wrong fit

A feature that is a full custom application (a recommendation engine, a multi-step AI workflow) wearing a thin WordPress skin. At that point Prism or Symfony's AI Bundle gives you far more structure than raw API calls hooked into WordPress actions.

> **Under the hood:** `wp_remote_post()` is WordPress core's own HTTP client, built to work consistently across server environments (some hosts restrict which PHP HTTP functions are available), which is why WordPress code conventionally avoids calling `curl` or `file_get_contents()` directly for outbound requests.

Shipping Multi-Language, Multi-Site

One codebase, several languages, several national subsidiaries with slightly different content. The request sounds like a translation problem. **Underneath, it is a content architecture problem: how a translated page relates to its original, and who may edit what, matters more than the translation mechanism.**



- TYPO3: Multilingual Content Trees Done Properly: a content structure built for this from the start.
- WordPress: Multisite and WPML: two WordPress answers to two different versions of the question.
- Symfony: The Translation Component: translating the application's own interface, not its content.

TYPO3: Multilingual Content Trees Done Properly

TYPO3 has treated languages as a first-class part of its page tree since long before most competitors. **A translation is not a separate copy of the site but a record connected to its original, inside the same structure.**

```yaml
# config/sites/main/config.yaml
languages:
  - title: English
    languageId: 0
    locale: en_US.UTF-8
    base: /
  - title: French
    languageId: 1
    locale: fr_FR.UTF-8
    base: /fr/
    fallbackType: strict
```

Editors translate page by page from the backend's language view. It shows which translations are complete, which are outdated because the original changed since, and which are missing:

```
./vendor/bin/typo3 language:update
```

`fallbackType: strict` means an untranslated page does not exist in French, rather than silently showing English content. That is a deliberate choice, not a default that ships half-translated pages to visitors.

When to reach for this

Large institutional sites with true multilingual content management: editors in each region manage their own language, and the link between an original and its translations has to survive years of edits by different people.

When it’s the wrong fit

A small site that needs a couple of static translated pages. WPML on WordPress, or a simpler translation plugin, gets there with far less setup.

> **Under the hood:** TYPO3 stores translated content as separate database records connected by a shared identifier (`l10n_parent`), rather than one record with multiple language columns. That structural choice is what lets a page have three finished translations and two missing ones at the same time, tracked cleanly rather than crammed into one row’s worth of columns.

WordPress: Multisite and WPML

WordPress has two answers here, for two different questions. **Picking the wrong one is a common mistake, and an expensive one to undo.**

"We have several distinct sites that should share users and plugins." Several national subsidiaries with their own content and design, not a translation of the same pages. That is what WordPress Multisite solves: one install running a network of separate sites, sharing code and users but not content.

```
wp core multisite-convert
wp site create --slug=fr --title="Example France"
wp site create --slug=de --title="Example Germany"
```

"We have one site that needs to exist in several languages." The same pages, translated. Multisite is the wrong tool for it, since it would mean maintaining separate copies of every page by hand. A translation plugin such as WPML keeps one site with linked translations of each page:

```
wp plugin install sitepress-multilingual-cms --activate
wp wpml language add fr de
```

```
// getting a translated post ID for the current language
$translated_id = apply_filters('wpml_object_id', $post_id, 'post',
true);
```

When to reach for each

Multisite when the sites differ beyond translation: different design, different content strategy, a different admin team per country. WPML when it is the same site and the same content in more than one language.

When it’s the wrong fit

Multisite as a translation tool, or WPML to run what are in fact separate, independently managed sites. Both mistakes are common, and both get expensive once a year of content has piled up on the wrong structure.

> **Under the hood:** Multisite works by adding a `blog_id` to WordPress’s core tables and routing requests through a network-aware bootstrap, essentially running several logically separate installs against shared code. WPML instead adds its own linking table between translated posts, leaving WordPress’s core single-site architecture untouched.

Symfony: The Translation Component

Everything else in this chapter translates content, what an editor wrote. **Symfony's Translation component solves a narrower problem: the application's own interface**, the labels, buttons and error messages that live in the code.

```
composer require symfony/translation
```

```
# translations/messages.en.yaml
welcome_message: "Welcome back, %name%!"
cart.empty: "Your cart is empty."
```

```
# translations/messages.fr.yaml
welcome_message: "Content de vous revoir, %name% !"
cart.empty: "Votre panier est vide."
```

```
<h1>{{ 'welcome_message'|trans({'%name%': user.firstName}) }}</h1>
```

```
$message = $translator->trans('cart.empty');
```

The active locale is set per request, from a URL prefix, a user preference or the `Accept-Language` header. Everything wrapped in `trans()` follows it.

When to reach for this

Any Symfony application whose interface must speak several languages: labels, validation messages, email templates, navigation. For a Symfony app with international users this is standard practice, not an advanced feature.

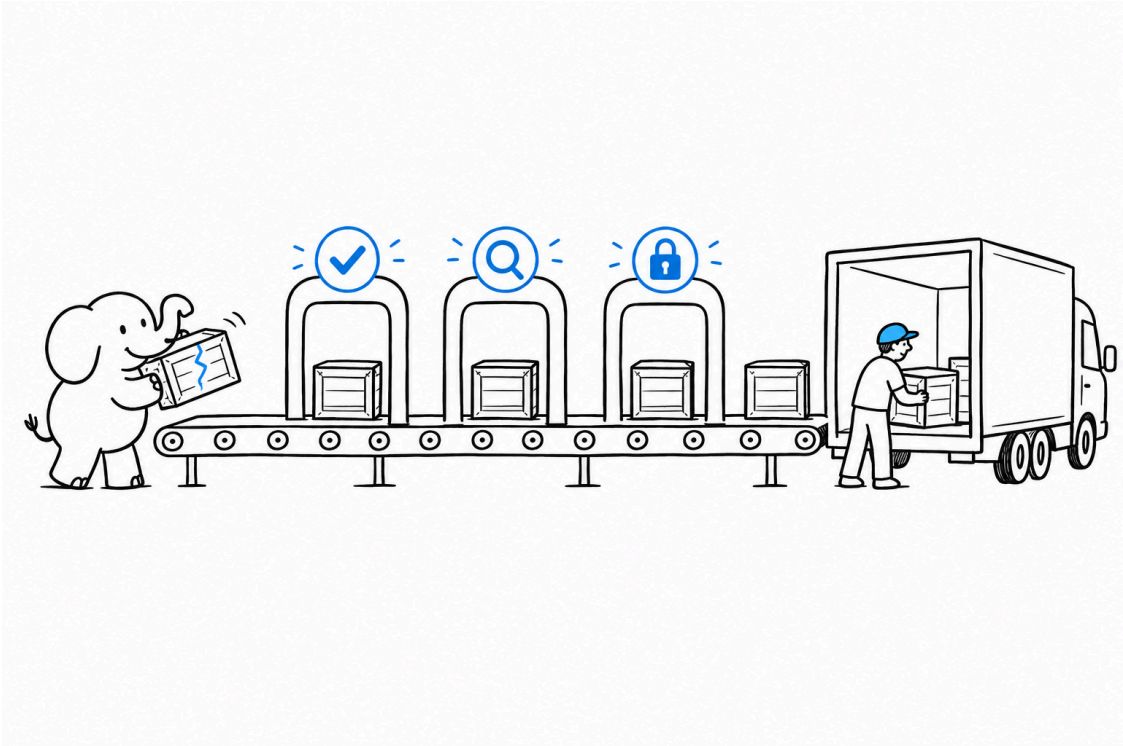
When it’s the wrong fit

Editorial content such as blog posts or product descriptions. That is the problem TYPO3’s multilingual content trees or WPML solve. This component has no concept of content, only of message keys.

> **Under the hood:** Translation files are loaded once and cached as compiled PHP arrays in Symfony’s cache directory, so looking up a translated string in production is an array lookup, not a file read or a database query, even with hundreds of keys across a dozen languages.

Shipping Confidence

“It works on my machine” is not a feature. The bug your user finds at 11 p.m. was there on your machine too; nothing was looking for it. **Confidence is a feature, and it belongs in every project in this book, not only the ones with time left over.** It comes from three checks that run without you: a test suite that catches regressions, a static analyzer that catches whole classes of bugs before the code runs, and a scan that flags a vulnerable dependency before it ships.



Each ecosystem has its own set of gates. The last two pages are the ones that work everywhere.

- Laravel: Pest, Larastan, and `composer audit`: tests that read like sentences, a type checker that knows Laravel, and a vulnerability scan in one command.
- Symfony: PHPUnit, PHPStan/Psalm, and Rector: the same three, plus a tool that rewrites your code for the next major version.
- WordPress: PHPUnit, PHPCS/WPCS, and WPScan: tests against a real WordPress, a linter that spots unescaped output, and a scanner for known plugin holes.
- Cross-Ecosystem: SAST, Dependency Scanning, and CI Gates: what runs the same in every stack, wired so a broken build cannot merge.
- Catching It in Production: Sentry and Flare ($): the moment confidence has to reach past your test suite, into what real users hit.

Laravel: Pest, Larastan, and `composer audit`

Pest is a testing framework built on PHPUnit, designed so a test reads close to plain English. Tests that are easy to read get written.

```
composer require pestphp/pest --dev --with-all-dependencies
php artisan pest:install
```

```
it('rejects an order with no items', function () {
    $order = Order::factory()->create();

    expect(fn () => $order->submit())
        ->toThrow(EmptyOrderException::class);
});

it('applies a discount code correctly', function () {
    $order = Order::factory()->create(['total' => 100]);

    $order->applyDiscount('SAVE10');

    expect($order->total)->toBe(90.0);
});
```

```
php artisan test
```

Larastan wraps PHPStan (see Symfony's PHPStan/Psalm for the framework-agnostic version) with Laravel-specific type knowledge. It catches a call to a method that does not exist on a model, or the wrong type passed to a job, without running the code.

```
composer require larastan/larastan --dev
```

```
# phpstan.neon
includes:
    - vendor/larastan/larastan/extension.neon
parameters:
    level: 6
    paths: [app]
```

```
vendor/bin/phpstan analyse
```

`composer audit` checks your installed dependencies against a database of known vulnerabilities. The day a package you depend on gets a CVE, this command tells you, before an attacker does.

```
composer audit
```

When to reach for this

Every Laravel project past the prototype stage. **Tests and static analysis are cheapest on day one and most expensive to retrofit** onto a codebase that has already grown without them.

When it's the wrong fit

There is no wrong fit here. The failure mode is skipping this page under deadline pressure.

> **Under the hood:** PHPStan, and Larastan with it, reads your type hints, including union types, `readonly` properties and enums, and reasons about what is possible without executing anything. The stricter your types, the more bugs it catches before a single test runs.

Symfony: PHPUnit, PHPStan/Psalm, and Rector

Symfony's testing story starts with plain PHPUnit. A bridge package smooths over the framework-specific parts: booting the kernel, making requests against it, querying Doctrine from a test.

```
composer require symfony/test-pack --dev
```

```
class OrderControllerTest extends WebTestCase
{
    public function testSubmittingAnEmptyOrderFails(): void
    {
        $client = static::createClient();
        $client->request('POST', '/orders/1/submit');

        $this->assertResponseStatusCodeSame(422);
    }
}
```

```
php bin/phpunit
```

PHPStan, or its close cousin Psalm, analyzes code without running it. It catches type errors, unreachable code and calls to methods that do not exist, before they become runtime bugs. Both work outside Symfony too, on the standalone components from earlier in this book.

```
composer require phpstan/phpstan --dev
vendor/bin/phpstan analyse src --level=8
```

Rector automates upgrades and refactors. Point it at a target PHP or Symfony version and it rewrites your syntax to match, mechanically, across thousands of files at once.

```
composer require rector/rector --dev
```

```
// rector.php
return RectorConfig::configure()
    ->withPaths([__DIR__ . '/src'])
    ->withSets([SymfonySetList::SYMFONY_64]);
```

```
vendor/bin/rector process
```

When to reach for this

Any Symfony project, at any size. **Rector earns its place the day a major Symfony or PHP upgrade looms** and editing every affected file by hand stops being realistic.

When it's the wrong fit

Rector on a codebase with no tests at all. Its changes are mechanical, but something has to check they did not change behavior. Pair it with at least the tests above.

> **Under the hood:** Rector parses your code into an AST (Abstract Syntax Tree), the same structure PHP's engine builds before execution, applies a rule to it, and prints modified PHP back out. It is mechanical code transformation, not a language model guessing at intent.

WordPress: PHPUnit, PHPCS/WPCS, and WPScan

Testing a WordPress plugin or theme means testing against WordPress itself, not a lightweight mock of it. The official scaffold sets up precisely that.

```
wp scaffold plugin-tests my-plugin
cd wp-content/plugins/my-plugin
composer install
./bin/install-wp-tests.sh wordpress_test root '' localhost latest
```

```
class DiscountCalculationTest extends WP_UnitTestCase
{
    public function test_bulk_discount_applies_over_ten_items(): void
    {
        $product = $this->factory->post->create(['post_type' =>
'product']);

        $price = apply_filters('calculate_bulk_price', 100, 12);

        $this->assertEquals(90, $price);
    }
}
```

PHP_CodeSniffer with the WordPress Coding Standards (WPCS) ruleset catches style violations, and with them a good share of the security mistakes WordPress plugins are known for: unescaped output, unsanitized input.

```
composer require --dev squizlabs/php_codesniffer wp-coding-
standards/wpcs
vendor/bin/phpcs --standard=WordPress my-plugin.php
```

WPScan checks a running site's plugins and themes against a database of known WordPress vulnerabilities. When something you depend on gets a disclosed CVE, you hear about it from the scan

rather than from your logs.

```
wpscan --url https://example.com --api-token YOUR_TOKEN
```

When to reach for this

Any custom plugin or theme with logic in it, not only template markup. Code that handles user input or payments benefits most from WPCS and its checks for unescaped output.

When it’s the wrong fit

A purely visual child theme with no PHP logic has little to test. Spend the effort where there is code, not markup.

> **Under the hood:** WP_UnitTestCase runs each test inside a database transaction and rolls it back afterward. Tests can create posts, users and options freely without leaving the database dirty for the next one, the same isolation strategy most PHP testing frameworks use.

Cross-Ecosystem: SAST, Dependency Scanning, and CI Gates

Everything so far in this chapter depends on the framework. This page covers what works the same whichever stack you ship with, and how to wire it so **a broken or vulnerable build never quietly reaches production.**

Static analysis as lightweight SAST: PHPStan and Psalm, already covered per framework, double as a first line of static application security testing (SAST). They catch SQL built from unsanitized input, or a variable used before it is guaranteed to be set.

GitHub CodeQL goes deeper. It traces how untrusted input flows through the code toward a dangerous sink, a raw SQL query, an `eval()`, an unescaped output, instead of checking types.

```
# .github/workflows/codeql.yml
- uses: github/codeql-action/init@v3
  with:
    languages: php
- uses: github/codeql-action/analyze@v3
```

Dependabot watches your `composer.lock` and opens a pull request the moment a dependency has a known vulnerability with a fix available.

```
# .github/dependabot.yml
version: 2
updates:
  - package-ecosystem: "composer"
    directory: "/"
    schedule: { interval: "weekly" }
```

Snyk ($) covers similar ground with a paid product's polish: a dashboard, license compliance checks, and coverage beyond Composer dependencies, for a subscription past its free tier.

Wiring it into CI

The point of all of this is a gate, not a report nobody reads:

```yaml
# .github/workflows/ci.yml
jobs:
  quality:
    steps:
      - run: composer audit
      - run: vendor/bin/phpstan analyse
      - run: vendor/bin/pest # or php bin/phpunit
```

If any step fails, the pull request cannot merge. That is the feature. Not that the tools exist, but that nobody has to remember to run them.

> **Under the hood:** CodeQL models data flow. It follows a value from where it enters the program (a `$_GET` parameter, a form field) to where it is used, instead of checking each line on its own. That is a different, and more expensive, kind of analysis than the type checking PHPStan and Psalm perform.

Catching It in Production: Sentry and Flare ($)

Tests catch what you thought to test for. Static analysis catches what the type system can prove. Neither catches what a user does at 11 p.m. that nobody anticipated. **Error tracking is confidence past the deploy button:** you learn that something broke for a real person the moment it happens, with a stack trace, instead of a support email that says "it's broken."

Sentry works across every framework in this book, with SDKs for Laravel, Symfony and plain PHP.

```
composer require sentry/sentry-laravel
php artisan sentry:publish --dsn=your-dsn-here
```

```
try {
    $order->submit();
} catch (\Throwable $e) {
    \Sentry\captureException($e);
    throw $e;
}
```

Uncaught exceptions are captured automatically once the SDK is installed. The manual call above is for an error you handle gracefully but still want to know about.

Flare ($), from Spatie in the Laravel ecosystem, is scoped to Laravel and adds framework-aware context to every report: which route, which query, which job, next to the stack trace.

```
composer require spatie/laravel-flare
```

Pricing

Sentry has a usable free tier for small projects and a paid plan as event volume grows. Flare is a paid product with a free trial and no free tier for ongoing use. Both carry the `$` here, Sentry's more conditional than Flare's.

When to reach for this

Any application in production. Waiting for a user to report a bug they cannot describe costs more debugging time than the subscription.

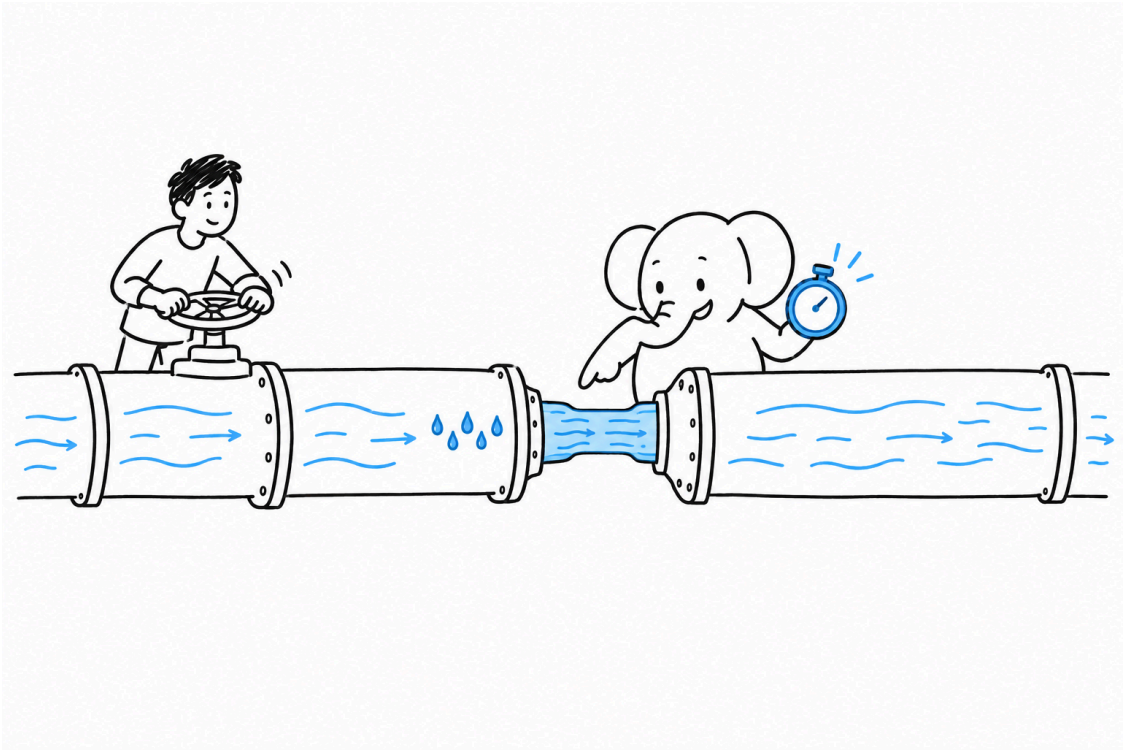
When it's the wrong fit

A prototype, or an internal tool with a handful of users who will ping you directly when something breaks. The setup is not justified yet.

> **Under the hood:** These tools hook into PHP's own exception and error handlers (`set_exception_handler()`, `set_error_handler()`). That is why installing them is a few lines of bootstrap code rather than a wrapper around every function call in your application.

Shipping It Fast, At Scale

Most features in this book ship fine on ordinary PHP: a fresh process per request, a database query, a response, done. Then traffic grows, or one page turns out to be slow, and "it works" quietly stops being enough. **This chapter is for the second phase, when performance has become a measured requirement rather than a worry.**



Three of the four pages are fixes. Read the fourth first: it tells you which fix you need.

- FrankenPHP and Laravel Octane: Worker Mode Performance: keep the application booted in memory between requests instead of rebuilding it every time.
- TYPO3: The Built-In Caching Framework: a layered cache with targeted invalidation, built into a large content platform.
- Nextcloud: Scaling a Self-Hosted Platform: what changes when a self-hosted PHP application grows from a team to an organization.
- Blackfire ($): Finding the Actual Bottleneck: measure where the time goes before you fix anything.

FrankenPHP and Laravel Octane: Worker Mode Performance

Traditional PHP rebuilds your whole application on every request, framework included: boot the container, load the configuration, then throw it all away once the response is sent. That habit is a large part of why PHP earned a reputation for being slower than runtimes that keep an application resident in memory.

Worker mode boots the application once and handles many requests against that same booted instance, resetting only what needs resetting between them.

```
composer require laravel/octane
php artisan octane:install --server=frankenphp
php artisan octane:start
```

FrankenPHP is a PHP application server written in Go. It runs PHP in worker mode natively, and in classic mode too, with built-in HTTPS and no separate web server:

```
frankenphp php-server --worker /path/to/public/index.php
```

The gain is large for applications whose bottleneck was framework boot time rather than their own logic. Benchmarks published by the projects show a multiple of PHP-FPM's throughput for the same application code, unchanged; your own numbers depend on how much of each request was boot.

The trade-off deserves a plain statement. Global state that used to reset between requests, a static property, a singleton holding request data, can now leak from one request to the next. The same process serves them in sequence instead of starting fresh.

When to reach for this

An application under measured load where profiling (see Blackfire) shows framework bootstrap, not your own business logic, as the bottleneck.

When it’s the wrong fit

A low-traffic site or an internal tool. Worker mode adds a category of bug, state leaking between requests, that is not worth taking on before you need the throughput.

> **Under the hood:** This is the same problem Reverb’s WebSocket server solves with Fibers, approached from the other side. Instead of one process holding many concurrent connections, worker mode keeps one process alive across many sequential requests. Both exist because PHP’s “die after every request” model, once a limitation, became optional.

TYPO3: The Built-In Caching Framework

TYPO3's caching framework deserves a look even outside a TYPO3 project, because **it treats caching as a layered problem rather than an on/off switch.** Page output, individual content elements, database query results and configuration each get their own cache, with their own invalidation rules.

```
# config/system/settings.php
$GLOBALS['TYPO3_CONF_VARS']['SYS']['caching']['cacheConfigurations']
['pages'] = [
    'backend' =>
\TYPO3\CMS\Core\Cache\Backend\Typo3DatabaseBackend::class,
    'options' => ['defaultLifetime' => 86400],
];
```

Editors never manage invalidation. When content changes, TYPO3 clears the caches that depend on it and nothing else, instead of a blunt "clear everything" after every edit:

```
$cacheManager = GeneralUtility::makeInstance(CacheManager::class);
$cacheManager->getCache('pages')->flushByTag('pageId_' . $pageId);
```

Tag-based invalidation is what makes aggressive caching safe on a large site. A typo fix on one page does not force every other page to be regenerated.

For high-traffic sites, TYPO3 adds a reverse proxy cache in front of all this, typically Varnish. An anonymous visitor asking for an unchanged page gets it fully rendered, without PHP running at all.

When to reach for this

Large TYPO3 sites under load, where page generation adds up across thousands of pages and content changes often enough that clearing the whole cache would defeat the purpose.

When it’s the wrong fit

A low-traffic site where the default caching is already fast enough. Tuning lifetimes and tags for a site nobody is straining solves a problem that does not exist yet.

> **Under the hood:** Tag-based invalidation associates each cached item with one or more tags (a page ID, a content type) at write time. Flushing “everything tagged `pageId_42`” is then a targeted operation, not a full wipe. Most well-designed caching systems, PHP or otherwise, rest on the same principle.

Nextcloud: Scaling a Self-Hosted Platform

A self-hosted platform like Nextcloud asks a question most cloud products never expose to their customers: someone on your team has to operate the growth curve, not just pay for a bigger plan.

Each step below answers one measured bottleneck, in the order an organization usually meets them after outgrowing a single small server:

```
# 1. Move sessions and caching to Redis instead of the filesystem
php occ config:system:set redis host --value=redis.internal
php occ config:system:set memcache.distributed --
value='\OC\Memcache\Redis'

# 2. Move file storage to S3-compatible object storage instead of local
disk
php occ config:system:set objectstore class --
value='\OC\Files\ObjectStore\S3'

# 3. Add read replicas and split heavy background jobs onto dedicated
workers
php occ background:cron
```

Redis comes in when concurrent users start contending on file-based session locks. Object storage comes in when the local disk fills up, or when one server's I/O becomes the limit. Separate workers come in when scheduled tasks, such as the file previews Nextcloud generates on its own, start competing with live users for CPU.

An organization that does not want to own this curve at all has two supported routes: the All-in-One Docker image (see Nextcloud: All-in-One Docker Deployment) or Nextcloud's own hosted offering, trading operational ownership for a subscription.

When to reach for this

A Nextcloud instance that has grown from a pilot team to an organization-wide rollout, where the single-server setup shows measured strain.

When it’s the wrong fit

A small team’s instance, comfortably within the range of one well-specced server. Redis and object storage ahead of a bottleneck are complexity paid early for no measured benefit.

> **Under the hood:** Nextcloud’s storage abstraction is built on Flysystem-style adapters (see League/Flysystem). The code that reads and writes files does not change when the backend moves from local disk to S3, because it was never written against “the filesystem” in the first place.

Blackfire ($): Finding the Actual Bottleneck

Every fix in this chapter assumes you already know what is slow. Guess wrong and you lose more time than the fix would have cost: a cache in front of a query that was never the bottleneck fixes nothing. **Blackfire profiles a real request and shows where the time and memory went, function by function.**

```
composer require blackfire/php-sdk --dev
blackfire run php artisan test
```

Or, for a live request, wrap it directly:

```
blackfire curl https://example.com/checkout
```

The profile is a call graph: which function called which, how long each took, how many times it ran. The usual culprits surface at once. An N+1 query loop. An external API called on every iteration when once would do. A template step that dominates the request despite looking trivial in the code.

```
// what profiling often reveals: 200 queries where one would do
foreach ($orders as $order) {
    echo $order->customer->name; // N+1: one query per order
}

// the fix, informed by the profile rather than a guess
$orders = Order::with('customer')->get();
```

Pricing

Blackfire has a free tier for individual use. Team collaboration and continuous profiling in CI are paid, hence the `$` for anything beyond solo, occasional use.

When to reach for this

The moment a page or an endpoint is reported slow and the cause is not obvious from reading the code. Profile first, then fix. The other order wastes a cycle.

When it’s the wrong fit

A page nobody has complained about and no metric has flagged. Profiling tools are for measured problems, not for anxiety about hypothetical ones.

> **Under the hood:** Blackfire is a PHP extension that hooks into the Zend Engine’s function call mechanism and times every call transparently. That is why profiling adds measurable overhead, and why it runs against one request on demand rather than on all production traffic.

Shipping to Production

A feature is not shipped until it is live. And "live" means someone, possibly you, is now responsible for keeping it up, keeping it patched, and surviving the traffic it gets. **This chapter is the last mile between "works locally" and "a client is using this," and it involves no new code.**



The right page depends less on your framework than on who holds the hosting contract.

- Laravel: Forge and Vapor ($): the Laravel team's own deployment products, one for servers, one for serverless.
- Cloud Hosting: AWS, GCP, and Azure the Pragmatic Way: the least-effort option on each cloud when the client already has a contract there.
- Platform.sh ($): One Deploy Story for Several Frameworks: a managed platform that does not care which framework is underneath.
- WordPress: Managed Hosting Done Right (Kinsta, WP Engine $): hosting built around what WordPress needs.
- Nextcloud: All-in-One Docker Deployment: the self-hosted path Nextcloud itself recommends.

Laravel: Forge and Vapor ($)

The Laravel team sells two deployment products, for two different ideas of what a server is.

Forge provisions and manages a traditional server on your own AWS, DigitalOcean or Hetzner account, with no manual administration. It configures Nginx, PHP-FPM, a database, SSL certificates and queue workers, then gives you a dashboard to deploy, monitor and scale.

```
# after connecting a server through Forge's dashboard, deployment is a git push
git push forge main
```

The deploy script runs whatever your project needs on each push, and you edit it per site:

```
cd /home/forge/example.com
git pull origin main
composer install --no-dev --optimize-autoloader
php artisan migrate --force
php artisan queue:restart
```

Vapor is a different model: serverless deployment onto AWS Lambda. There is no server to patch or scale. **The application runs only while it handles a request**, and scales on its own from zero to thousands of concurrent requests.

```
composer require laravel/vapor-cli --dev
vapor deploy production
```

Pricing

Both are subscriptions on top of the infrastructure they provision. Forge charges per managed server; Vapor charges by usage plus the underlying AWS costs.

When to reach for each

Forge for an app that benefits from a persistent server: background workers, WebSocket connections through Reverb. Vapor for spiky or unpredictable traffic, where paying only for use, and never thinking about capacity, outweighs the constraints serverless imposes.

When it's the wrong fit

A team with infrastructure expertise and existing cloud tooling may prefer managing deployment directly to paying for Forge's convenience layer on top of it.

> **Under the hood:** Vapor's serverless model works because Laravel's request lifecycle was stateless from the start. Nothing assumes the same process handles the next request, which is the assumption Lambda's "cold start per invocation" model needs in order to work at all.

Cloud Hosting: AWS, GCP, and Azure the Pragmatic Way

Sometimes the deployment decision is not yours. The client already has an AWS, Google Cloud or Azure contract, for compliance, procurement or existing-infrastructure reasons that have nothing to do with what is easiest for you. **The pragmatic move is the option on that cloud that costs the least operational effort**, not infrastructure you hand-roll and do not need.

AWS: App Runner takes a container and runs it with automatic scaling and load balancing, with far less configuration than a hand-built EC2/ECS setup.

```
aws apprunner create-service \
  --service-name my-app \
  --source-configuration file://apprunner-config.json
```

Google Cloud: Cloud Run does the same job for a container on GCP. It scales to zero when idle, which suits a modest-traffic app on a usage-based bill.

```
gcloud run deploy my-app --source . --platform managed --region us-
central1
```

Azure: App Service runs PHP directly, no container required. It is closer in spirit to traditional managed hosting than the container-first options above.

```
az webapp up --runtime "PHP:8.3" --name my-app
```

A `Dockerfile` written once, on a standard `php:8.3-fpm` or FrankenPHP image, works on App Runner and Cloud Run with little or no change. Both only want a container that listens on a port.

When to reach for this

The client already has, or requires, infrastructure on one cloud, and a framework platform like Forge or a third-party PaaS like Platform.sh is ruled out by procurement or compliance.

When it’s the wrong fit

No existing cloud relationship and no compliance requirement pointing at one. A framework-native option or a dedicated PaaS usually reaches production faster, with less infrastructure knowledge required.

> **Under the hood:** All three platforms run the same PHP-FPM or FrankenPHP process your laptop does; a container is a container. The differences are in deployment mechanics and billing, not in how PHP executes your code once it is running.

Platform.sh ($): One Deploy Story for Several Frameworks

Platform.sh sells framework-agnostic infrastructure as code: **one configuration format, whether the project underneath is Laravel, Symfony, WordPress, TYPO3 or something else.** An agency shipping across several of the stacks in this book learns one deployment platform instead of five.

```
# .platform.app.yaml
name: app
type: php:8.3
dependencies:
  php:
    composer/composer: "^2"
relationships:
  database: "db:mysql"
hooks:
  build: composer install --no-dev --optimize-autoloader
  deploy: php artisan migrate --force
```

```
# .platform/services.yaml
db:
  type: mysql:8.0
```

```
git push platform main
```

Every push creates a full, isolated environment with its own database and its own storage, rather than deploying over a shared one. Previewing a feature branch as a running application becomes routine, not a special setup.

Pricing

Platform.sh is a paid managed service, billed per project and environment, with no free tier for production use.

When to reach for this

Agencies or teams running several projects across different PHP frameworks and CMSes, where one deployment workflow for all of them matters more than any single platform's framework-specific conveniences.

When it's the wrong fit

A single project committed to one framework for the long term. A framework-native option like Forge or managed WordPress hosting is simpler, and often cheaper, for that one case.

> **Under the hood:** The per-branch environment model works because Platform.sh treats infrastructure as versioned configuration next to your code. It is the instinct behind a `composer.lock` file pinning dependency versions, extended to the database and the services a deployment needs.

WordPress: Managed Hosting Done Right (Kinsta, WP Engine $)

Generic hosting treats WordPress like any other PHP application. Managed WordPress hosting, from providers like Kinsta and WP Engine, is built around what WordPress is known to need: object and page caching tuned to its query patterns, one-click staging, core and plugin updates on a schedule you control, and malware scanning aimed at known plugin vulnerabilities.

```
# a typical managed-host workflow: staging, then push to production
wp @staging plugin update --all
wp @staging cache flush
# after review in the staging environment
wp @production deploy
```

The difference from self-managed hosting shows up during an incident. Managed hosts include automatic backups with one-click restore, and support staff who already know how WordPress fails, rather than generic server support learning the platform from your ticket.

Pricing

Kinsta and WP Engine are paid, tiered by traffic and site count, with no free tier. The cost sits well above generic shared hosting and well below an in-house team doing the same reliability and performance work by hand.

When to reach for this

Any client-facing, revenue-generating or otherwise important WordPress site, where downtime or a slow page has a cost and nobody in-house wants to own WordPress-specific server tuning and security monitoring.

When it’s the wrong fit

A low-stakes personal blog or an internal tool. Cheaper generic hosting is enough, and the specialized tooling would sit unused.

> **Under the hood:** Much of what these hosts sell is object caching: the results of expensive WordPress queries kept in memory, through Redis or Memcached, so a popular page does not re-run the same queries for every visitor. It is the caching instinct behind TYPO3's caching framework, pre-configured for WordPress's usual query patterns.

Nextcloud: All-in-One Docker Deployment

Standing up Nextcloud by hand means configuring a web server, PHP-FPM, a database, Redis, a reverse proxy with valid TLS and a background cron job. Each is a separate chance to misconfigure something security-sensitive. **The All-in-One (AIO) Docker image is Nextcloud's own answer: all of that, pre-wired, as a coordinated set of containers.**

```
docker run \
  --name nextcloud-aio-mastercontainer \
  --restart always \
  -p 8080:8080 \
  -v nextcloud_aio_mastercontainer:/mnt/docker-aio-config \
  -v /var/run/docker.sock:/var/run/docker.sock:ro \
  nextcloud/all-in-one:latest
```

The container's web interface then walks you through TLS certificates, the optional apps to enable (including Talk's high-performance backend, covered in Real-Time Features) and backup configuration. One dashboard, instead of a dozen config files.

Updates to Nextcloud and to each enabled app go through the same interface:

```
# update the mastercontainer image, then trigger an update from its UI
docker pull nextcloud/all-in-one:latest
docker stop nextcloud-aio-mastercontainer
docker rm nextcloud-aio-mastercontainer
# re-run the original docker run command to restart it on the new image
```

When to reach for this

Self-hosting Nextcloud for ongoing use, for a small team or a whole organization. The AIO image encodes hard-won operational knowledge about securing and maintaining an instance, the kind that is easy to get subtly wrong by hand.

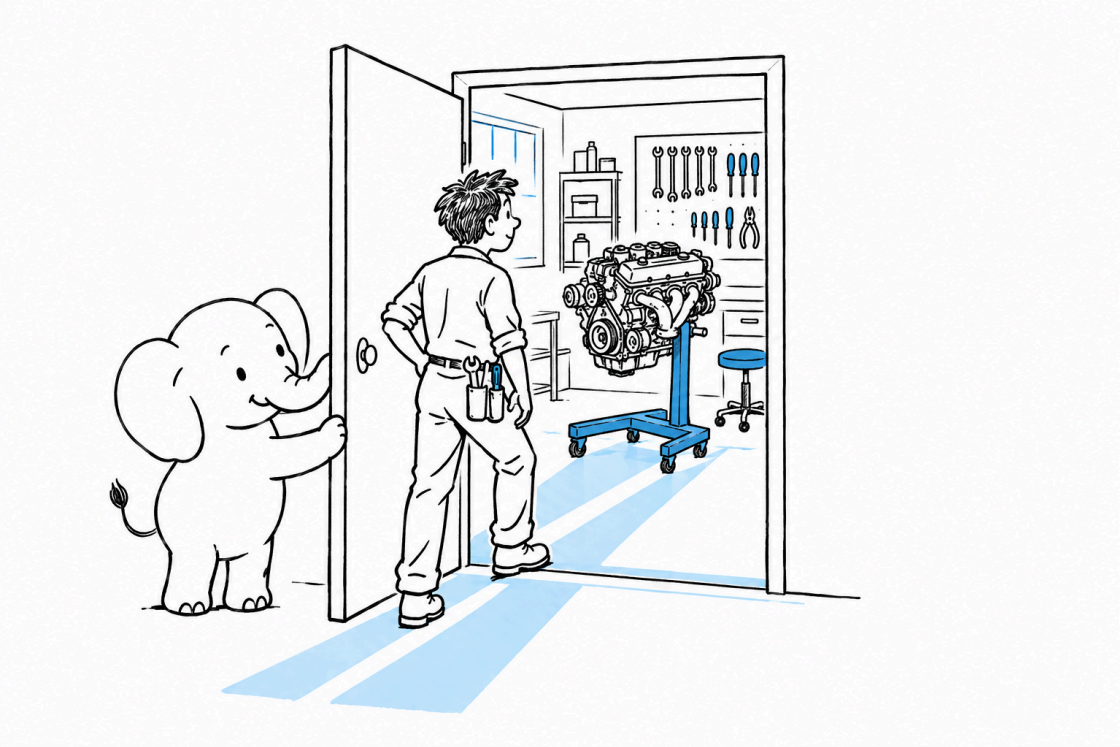
When it's the wrong fit

A quick local evaluation, where the plain Docker image (see Nextcloud: A Ready-Made Drive) is faster to throw away. Or an environment where Docker itself is not an option.

> **Under the hood:** The mastercontainer pattern, one container managing the lifecycle of several others through the Docker socket, is orchestration lighter than Kubernetes and heavier than a single `docker run`. It exists so that updates to interdependent services (the app, the database, the proxy) happen in the right order on their own.

Where to Go From There

This book covered a lot of ground without once asking you to commit to a stack for good. That was the point. **The pragmatic builder's skill is not mastering one framework; it is matching a feature to the tool that ships it fastest, again and again, across projects and clients.** This last chapter is about keeping that skill sharp once the book is closed.



- Evaluating a New Stack in a Day: a repeatable way to size up a tool you have never used.
- Communities Worth Joining: where each ecosystem in this book gets its questions answered.
- If You Get Curious About What's Underneath: the door back to the language itself, for the day "just ship it" turns into "wait, how does this work?"

Evaluating a New Stack in a Day

A new client, a new tool, a framework you have never shipped anything in. This book covers the options that exist today; new ones will exist tomorrow, and clients do not wait for the next edition. **One day, four steps, and you know whether to commit project time to it.**

Morning: scaffold and orient

Run the tool's own quickstart, as documented, no shortcuts. Note how long it took, how much was decided for you, and how much freedom is left. Scaffolding a First Project in Under a Minute shows what this looked like for the frameworks and CMSes in this book.

Midday: build one real, small thing

Not a "hello world." A login form, a data table, a single API endpoint. Time it, and note where you got stuck in the documentation. That friction is the most honest signal you will get about what a project in this tool will feel like.

Afternoon: check the edges

Look for what every chapter of this book treats as decision-relevant. Is there an official, maintained path for testing (see Shipping Confidence)? For deployment (see Shipping to Production)? Does a stuck question get answered in hours rather than weeks (see Communities Worth Joining)?

Evening: decide against the brief, not the marketing

Every tool's homepage claims to be fast, modern and developer-friendly. None of that matters as much as the question Chapter 1 asked first: framework, CMS or headless? And does this option fit the feature you were asked to build?

> **Under the hood:** The same practice (quickstart, then a small feature, then the edges) evaluates tools outside PHP too. What is specific to this book is the list of edges: PHP's ecosystem has mature answers for testing, deployment and community support, so a new tool lacking any of them deserves to be treated as a yellow flag, not a minor gap.

Communities Worth Joining

Documentation answers the question you knew to ask. **A community answers the one you didn't.** Every ecosystem in this book has a place where practitioners answer specific problems, usually faster than a search engine finds you a current answer.

- **Laravel**: the Laravel Discord and the Laracasts forum are both active, and the official docs at laravel.com are unusually well maintained release after release.
- **Symfony**: the Symfony Slack and the community forum linked from symfony.com are where most framework and component questions get resolved.
- **WordPress**: make.wordpress.org hosts the contributor teams building WordPress core, while the WordPress.org support forums cover plugin and theme questions.
- **TYPO3**: the TYPO3 Slack, linked from typo3.org, gives depth to the enterprise-CMS questions this book only had room to introduce.
- **API Platform**: GitHub Discussions on the project's own repository, linked from api-platform.com, is where most implementation questions land.
- **Nextcloud**: the community forum and GitHub discussions linked from nextcloud.com cover both self-hosting operations and app development.

One habit to keep

Beyond any single framework, PHP has an active conference and user group scene, and the major frameworks publish detailed changelogs and upgrade guides rather than surprising you. Follow a framework's release notes, not only its announcements. It is the single most effective habit for staying current without relearning your stack from scratch.

> **Under the hood:** Not applicable here. Communities are made of people, not code, and that is the point. The fastest answer to a stuck problem is often someone who hit the same wall last month, not a deeper read of the source.

If You Get Curious About What's Underneath

If you opened every "Under the hood" box in this book, you already know more about the engine than this book asked you to learn. You have seen Fibers make a WebSocket server possible, attributes and reflection generate a whole API from one class, and a type system solid enough for static analysis to catch bugs before a single test runs. None of it was framed as a language lesson. It was framed as "here is why the convenience you just used works."

If that pattern started to interest you for its own sake, this book was quietly built to lead you here.

The PHP Book picks up at that point: variables, types, control flow, classes, and the engine's own error model, taught properly rather than reverse-engineered from a framework's behavior. It assumes what you brought here, coding experience elsewhere, and points it at the language instead of at one framework's conventions.

You do not need it to keep shipping. Every feature in this book works today, in production, without opening it once. It is there for the day "just ship it" turns into "wait, how does this work?" If you have read this far, that day may come sooner than you expected.

Read The PHP Book

Appendix

Three reference pages, for looking things up rather than reading through:

- A - The Stack Cheat Sheet: every “which one should I pick” decision in this book, on one page.
- B - Glossary of Ecosystem Terms: the recurring vocabulary, defined by what it does for you.
- C - Index of “Under the Hood” Boxes: every optional aside, in reading order, for when one of them stuck with you.

A - The Stack Cheat Sheet

One page: the need on the left, the pick on the right. A `$` marks anything that is not open source. Where two picks appear, the first is usually the faster start.

You need to ship…	Reach for	Chapter
A custom application with business logic of its own	Laravel or Symfony	Ch. 1
A marketing site or blog the client edits themselves	WordPress	Ch. 3
A large, multi-department, multilingual institutional site	TYPO3	Ch. 3 / Ch. 13
A one-off script or CLI tool, no framework	Symfony/Console, Guzzle, Monolog	Ch. 2
Login, sign-up, password reset	Laravel Breeze/Fortify, Symfony Security, or WordPress roles	Ch. 4
An internal admin panel	Laravel Filament, Symfony EasyAdmin, or Nova `$`	Ch. 5
A REST/GraphQL API with docs	API Platform	Ch. 6
Live updates without a page refresh	Laravel Reverb, Symfony Mercure, or Pusher `$`	Ch. 7
File sharing and sync (“our own Dropbox”)	Nextcloud	Ch. 8
Ordinary file uploads in an existing app	Laravel Storage / League/Flysystem	Ch. 8
A product catalog and checkout	WooCommerce, Sylius, or Cashier + Stripe `$`	Ch. 9
Search that tolerates typos and ranks by relevance	Laravel Scout + Meilisearch (or Algolia `$`)	Ch. 10
Background jobs and scheduled tasks	Laravel Queues, Symfony Messenger, or Action Scheduler	Ch. 11
An AI feature added to an existing app	Laravel Prism or Symfony AI Bundle	Ch. 12
Several languages or several regional sites	TYPO3, WPML, or Symfony Translation	Ch. 13
Tests, static analysis, and vulnerability scanning	Pest/Larastan, PHPStan/Psalm, `composer audit`	Ch. 14
Errors seen in production	Sentry (or Flare `$`)	Ch. 14
More throughput without rewriting the app	FrankenPHP / Laravel Octane	Ch. 15
Finding the performance bottleneck	Blackfire `$`	Ch. 15
Deploying a Laravel app	Forge or Vapor `$`	Ch. 16
Deploying onto an existing AWS/GCP/Azure contract	App Runner, Cloud Run, or App Service	Ch. 16
Deploying several frameworks the same way	Platform.sh `$`	Ch. 16
Deploying WordPress where downtime costs money	Kinsta or WP Engine `$`	Ch. 16

You need to ship...	Reach for	Chapter
Self-hosting Nextcloud in production	The All-in-One Docker image	Ch. 16

B - Glossary of Ecosystem Terms

Each term is defined by what it does for you, not by its formal definition.

Adapter A small piece of code that makes one thing (a storage backend, a search engine) speak the interface your application expects, so swapping the service underneath does not mean rewriting the application. See League/Flysystem.

Attribute Metadata written directly above a PHP class or method (`#[ApiResource]`, `#[IsGranted('ROLE_ADMIN')]`) that a framework reads at runtime to decide how to treat it. It replaces what used to need separate configuration files.

Bundle Symfony's word for an installable package that adds a feature to an application: the Security Bundle, the AI Bundle. Roughly a Laravel package or a WordPress plugin.

Facade A Laravel convention: a short, static-looking call (`Storage::get(...)`) backed by a real, swappable object underneath. You get a memorable syntax without losing the flexibility of dependency injection.

Fiber A PHP 8.1 feature that lets a single process pause and resume execution at will, holding many things in flight at once without a thread per task. It is what makes Reverb's WebSocket server possible.

Hook / Filter WordPress's extension mechanism: a named point in core code where a plugin runs its own function (`add_action`) or modifies a value on its way through (`add_filter`), without editing WordPress core.

Migration A version-controlled, incremental change to a database schema, written in code rather than applied by hand. Every developer and every environment ends up with the same structure by running the same migration files.

ORM (Object-Relational Mapper) The layer that turns database rows into PHP objects and back (Eloquent in Laravel, Doctrine in Symfony), so day-to-day code reads `$product->price` instead of hand-written SQL.

Provider (Service Provider / Service Container) The part of a framework that builds objects and hands them to whatever needs them. A class declares "I need a logger" in its constructor and never learns how that logger was built or configured.

PSR A PHP Standards Recommendation: an agreed interface (PSR-3 for logging, PSR-7 for HTTP messages) that lets packages from different vendors work together without knowing each other's internals.

Resource In an API, one type of thing the API exposes (a `Product`, an `Order`) with the operations available on it. Laravel also uses the word, more narrowly, for the class that shapes a model into JSON.

Webhook A callback. Instead of your app asking a service "did anything happen yet?" over and over, the service sends your app an HTTP request the moment something does. It is how Stripe tells your app a payment succeeded.

Worker A long-running process that pulls tasks off a queue and runs them, separate from the process serving web requests, so slow work never blocks a user waiting for a page. See Shipping Background Work.

C - Index of “Under the Hood” Boxes

Every optional aside in this book, in reading order, for the day one of them stuck with you and you want to find it again.

Chapter	What it covers
Framework, CMS, or Headless?	Why frameworks, CMSes, and API tools share the same language features under different packaging
Scaffolding a First Project	Composer's dependency resolution and autoloading
Symfony/Console	Console commands as plain PHP classes, no special runtime
Guzzle	PSR-7 and PSR-18, the interfaces that let HTTP clients interoperate
League/Plates	Why plain-PHP templates need no compile step
Monolog	PSR-3, the standard logging interface
Respect/Validation	Validation libraries leaning on PHP's own type system
League/Flysystem	Interface-based polymorphism applied to storage
WordPress Block Themes	Blocks stored as structured HTML, not PHP templates
TYPO3 Structured Content	TCA as metadata-driven UI generation
Statamic	Flat files, PHP's filesystem functions, and caching instead of SQL
Craft CMS	Dynamic property access behind flexible content fields
Laravel Breeze/Fortify	Why password hashing is deliberately slow (bcrypt/argon2id)
Symfony Security	The `#[IsGranted]` attribute and reflection
WordPress Roles	Capabilities as plain strings, not a formal permissions engine
Laravel Filament	Fluent, chainable objects enabled by named arguments and enums
Symfony EasyAdmin/Sonata	Reading Doctrine entity metadata to infer field types
API Platform Admin	Generating an OpenAPI/Hydra schema from attributes
WordPress CPT as CRUD	Every post type sharing one `wp_posts` table
Laravel Nova	Why a paid license changes nothing in the package mechanism
API Platform from One Class	Attributes plus reflection generating routing, validation, and docs
Laravel Sanctum	Why tokens are hashed differently from passwords
WordPress REST API	The REST API built on WordPress's hooks and filters
Laravel Reverb/Livewire	PHP Fibers, and how they make a WebSocket server possible
Symfony Mercure/Turbo	Server-Sent Events as a plain HTTP feature
Nextcloud Talk	Nextcloud's app system and PHP's own autoloading conventions
Pusher	Why a good abstraction makes "self-host or pay" a reversible decision

Chapter	What it covers
Nextcloud Storage	Chunked uploads and ETags for efficient sync
Laravel Storage/S3	Pre-signed URLs and cryptographic signatures
WordPress Media Library	Eager thumbnail generation with GD/Imagick
WooCommerce	Commerce data modeled on the same `wp_posts`/`wp_postmeta` tables
Sylius	Built from reusable Symfony components, not from scratch
Laravel Cashier/Stripe	Verifying webhook signatures before trusting a payload
Laravel Scout	A driver interface making search engines swappable
TYPO3 Solr	Mirroring page permissions into the search index
WordPress Search	`WP_Query` as the hook point for smarter search backends
Laravel Queues	Job serialization and why jobs should stay simple
Symfony Messenger	Routing messages to handlers through PHP's type system
WordPress WP-Cron	Why WP-Cron triggers on page loads instead of a system timer
Laravel Prism	A provider interface, the same pattern as Flysystem's adapters
Symfony AI Bundle	Dependency injection applied to an AI agent
WordPress AI Plugins	Why WordPress ships its own portable HTTP client
TYPO3 Multilingual Trees	Translations as linked records, not extra columns
WordPress Multisite/WPML	Two structural answers to two different problems
Symfony Translation	Translation files compiled into cached PHP arrays
Laravel Pest/Larastan	Static analysis reasoning over type hints without running code
Symfony PHPStan/Rector	Rector rewriting code by parsing and modifying its AST
WordPress Testing	Test isolation through database transaction rollback
Cross-Ecosystem SAST	Data-flow analysis versus type checking
Sentry/Flare	Hooking into PHP's own exception and error handlers
FrankenPHP/Octane	Worker mode as the mirror image of Fibers
TYPO3 Caching	Tag-based cache invalidation
Nextcloud Scaling	Storage adapters again, this time under scaling pressure
Blackfire	Profiling through a Zend Engine extension
Laravel Forge/Vapor	Statelessness as the enabler of serverless deployment

Chapter	What it covers
Cloud Hosting	The same PHP process, whichever cloud runs it
Platform.sh	Infrastructure as versioned configuration
WordPress Managed Hosting	Object caching with Redis/Memcached
Nextcloud AIO	Container orchestration through the mastercontainer pattern