The Reputation

You have heard the joke before you opened this book: PHP is the language of 2004, held together with duct tape, written by people who never learned to program properly. **The reputation is old, it was earned, and most of what earned it is no longer true.** That sentence is the whole chapter, and the rest of this book exists to let you check it yourself rather than take it from me.

Start with what is still visibly true. As of 2026-09-17, PHP runs **69.9 percent of all websites whose server-side language is known**, according to W3Techs, which recomputes this figure continuously from live crawl data rather than survey responses. The number moves week to week (it read 71.8 percent in March 2026 and 70.2 percent on 3 September 2026), so treat any single percentage as a snapshot, not a constant, and expect the number printed here to already be slightly wrong by the time you read it. What the movement does not change is the order of magnitude: most of the server-rendered web still runs PHP, which means the language is not a bet you would be making alone.

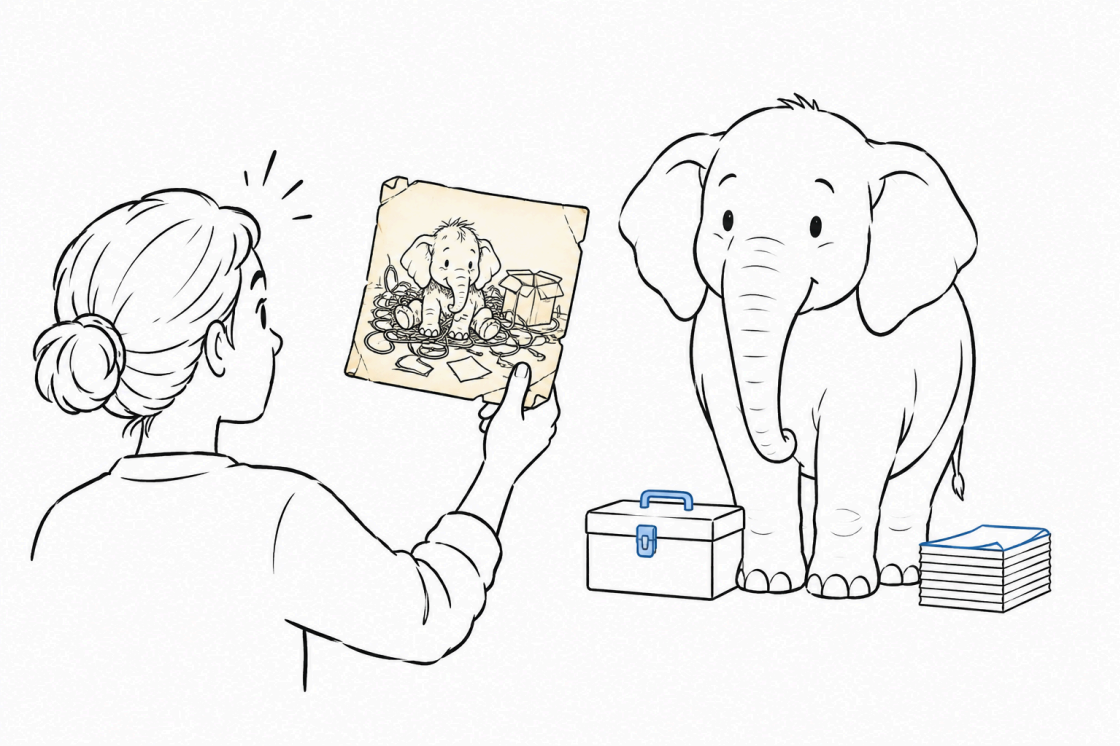
That scale is also the source of the reputation. A language that a large fraction of the web has run for three decades accumulates a large fraction of the web's worst code, written under every level of time pressure and skill, in every era of the language's own history. Early PHP, the PHP of the phpBB and osCommerce era, mixed HTML and logic on one page with no enforced structure, no dependency manager, and no standard way to test what you had written. A program that ran was often good enough, because there was no cheap way to prove it was correct. Criticism of that era's code was accurate, and a meaningful share of it is still running in production today, which you can read about in Governance and Versions. Judging the current language by that code is like judging a company by its founding-year prototype.

The code itself earned the reputation too, not just the era. A page that mixed the database query, the business logic, and the HTML output in one file, with no enforced boundary between them, was not a caricature: it was a common, working pattern, because nothing in the language stopped you from writing it that way and nothing in the toolchain caught the result before it shipped. There was no standard way to declare that a function expected an integer and not a string that merely looked like one, no standard way to pull in a third-party library without downloading a zip file and hoping its internal file paths did not collide with your own, and no standard, shared convention for how one library's code should be organized so another library could find it. Individually, each gap was a minor inconvenience. Together, across millions of independently written sites, they produced exactly the ecosystem the reputation describes.

What changed did not change quietly. The language added a real type system: scalar and return type declarations, union and intersection types, and `declare(strict_types=1)` to opt a file out of PHP's historical type coercion entirely, covered with code in The Runtime. It gained a dependency manager and a public package registry, covered in Cost, that turned "download a zip and hope"

into a resolvable, lockable, auditable install step. A standards process, PSR, adopted voluntarily by framework and library authors rather than imposed by the language itself, gave the ecosystem a shared convention for how autoloading, logging, and HTTP messages should be shaped, so that two libraries written by strangers could be relied on to fit together. Static analysis tools that read a codebase without running it, PHPStan and Psalm among them, turned "does this function actually always return what it claims" from a runtime surprise into a question you can answer before deploying. None of this erases the old code still running somewhere on your vendor's server. It does mean that "PHP" and "the worst PHP you have seen" stopped being the same referent sometime in the last decade, and a due-diligence review has to treat them as two separate questions.

The ecosystem around the language changed on the same timeline. A page you might have described a decade ago as "just PHP" is now, more often than not, built on one of a handful of full-stack frameworks, among them CakePHP, Laminas, Laravel, Symfony, and Yii, each imposing its own structure, routing, and testing conventions on the code written inside it. None of them is presented here as the right or the popular choice; the point is only that the language stopped being something you wrote entirely by hand somewhere around the time the reputation calcified, and the reputation did not update.



> **The limit.** The reputation is not entirely obsolete. A meaningful share of the PHP install base still runs versions with no vendor support, and a codebase inherited rather than written under discipline can still be exactly as unstructured as its reputation suggests. Neither fact is about the current language; both are about what a specific organization did with an old version of it, and Where PHP Is the Wrong Choice names the pattern plainly rather than arguing it away.

Every chapter after this one follows the same shape, because a decision document that asks to be trusted has to be checkable, not just readable. Each one opens with the question an evaluator actually asks, states the sourced answer plainly, then shows the evidence before the argument built on it. Each one names, in its own terms, where PHP falls short of the question it is answering, because a chapter with no limit in it is not a chapter you should trust. And each one closes with something you can check yourself in an afternoon: a command to run, a public page to open, a benchmark to rerun. None of this is an argument that PHP is the right choice for your next project. It is an argument that the reputation and the current language are two different things, and that an evaluation worth trusting has to look at the second one directly instead of quoting the first one from memory. The rest of this book is that look: how the language actually executes, who actually runs it today and at what scale, what it costs, who maintains it, and, as plainly as the first four questions, where it is the wrong tool for the job.

What to verify yourself. Open w3techs.com/technologies/details/pl-php right now and compare the live percentage to the one printed above; the gap between them is a small, honest demonstration of how this book treats a moving number. Then read the version-fragmentation figures in Governance and Versions before you decide how much of the old reputation still applies to a vendor or a codebase you are actually evaluating.

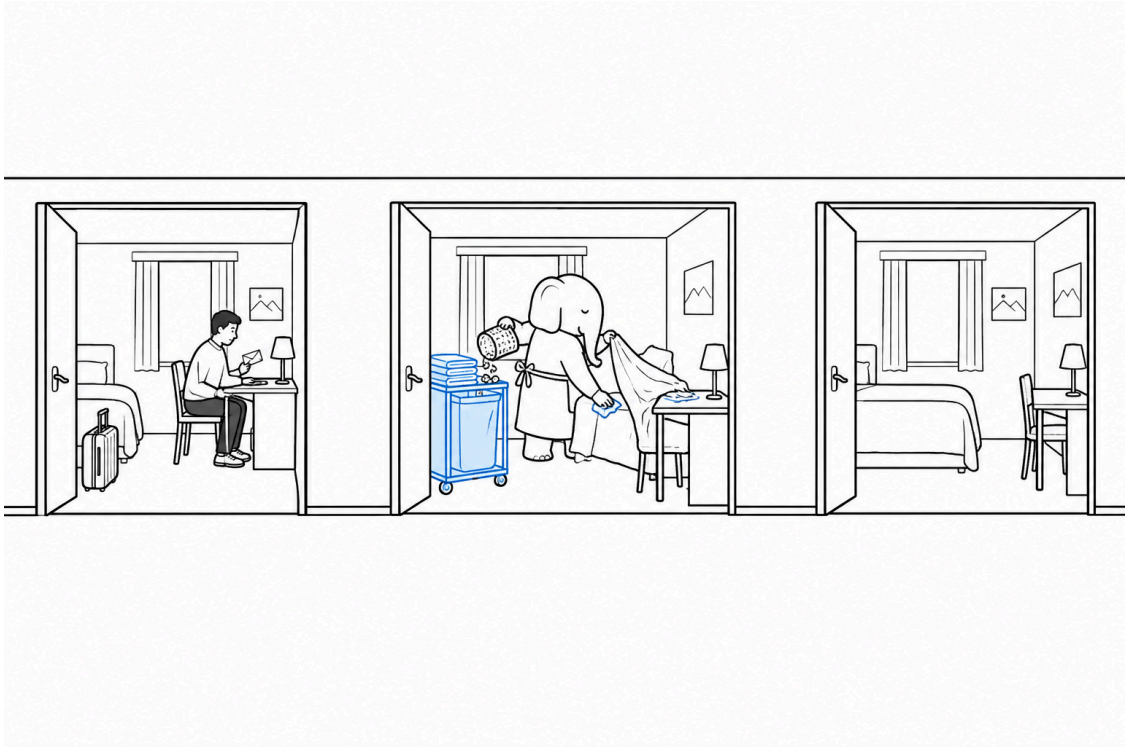
The question this book keeps coming back to is not whether PHP used to deserve its reputation. It is whether the language in front of you today, the one with a type system, a dependency manager, and a funded maintenance process, completes the architecture you are actually building. The Runtime is where that question starts.

The Runtime

PHP executes one request per process or thread, on the Zend Engine, shared-nothing by default: nothing your code puts in memory survives past the response it just sent, unless you explicitly change the serving model to keep the application booted. That single sentence is the answer to most of the runtime questions an evaluator brings to this book, and it deserves a pause before you read anything else in this chapter, because almost everything else PHP does or does not do well is a consequence of it.

Shared-nothing execution means a request boots the application, runs your code, and tears the whole thing down, every time. A variable you set, a database connection you opened, a cache you warmed in memory: none of it is there for the next request, which gets its own clean boot. This is not an accident of a slow language catching up; it is the model PHP was built around from the start, and it buys you a specific, valuable property. A memory leak in one request cannot accumulate across a thousand requests, because there is no shared memory for it to accumulate in. A crash in one request cannot corrupt the state the next request depends on, because there is no shared state. The cost is the one you would expect: nothing is warm. Every request that needs a database connection opens one; every request that needs the application's routes and configuration loaded rebuilds them, unless something below the language itself is caching the result.

That something is `opcache`. PHP source code is compiled to bytecode before it runs, and by default that compilation happens again on every single request, from scratch. Opcache, bundled with PHP since version 5.5, caches the compiled bytecode in shared memory so the next request skips recompilation. Stated precisely, rather than the way it usually gets described: opcache is a `zend_extension`, and it has to be loaded in `php.ini` to do anything. Most distribution packagers turn it on by default, but PHP itself does not, and "PHP is fast because it caches bytecode" is only true of a specific, checkable configuration, not of the language as installed.



What the type system actually enforces

The type declarations added since PHP 7 are not decoration. A file that opts in enforces its contracts at the engine level, not just in a linter:

```php
<?php

declare(strict_types=1);

function total(float ...$prices): float
{
    return array_sum($prices);
}

total(9.99, "4.50"); // TypeError: total(): Argument #2 must be of type float, string given
```

Without `declare(strict_types=1)`, PHP would coerce a numeric string into a float and run anyway. With it, the mismatch is a `TypeError` at the point it occurs, not a silent corruption three functions downstream. This is the concrete, checkable version of the claim in The Reputation that the type system changed: it is opt-in per file, and it is real once you opt in.

Serving models: how “shared-nothing” gets bent

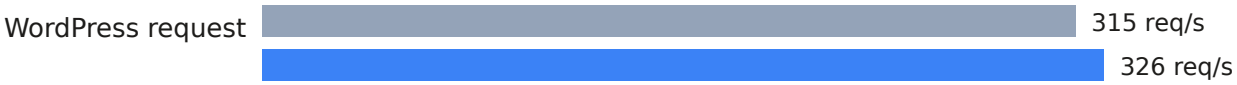
Three serving approaches, alphabetically, cover most of how PHP applications run in production today: FrankenPHP, PHP-FPM behind a web server, and RoadRunner. PHP-FPM is the traditional model: a pool of worker processes, each one handling one request at a time, each one restarted periodically, none of them keeping application state between requests. It is the purest expression of shared-nothing, and it is still the default most hosting environments assume.

FrankenPHP and RoadRunner both offer a “worker mode” that bends the shared-nothing default on purpose: the application boots once and stays in memory, handling many requests in sequence without a full reboot between them. FrankenPHP’s own documentation describes this plainly, including the trade-off it introduces: PHP’s superglobals (`$_GET`, `$_POST`, `$_COOKIE`, `$_FILES`, `$_SERVER`, `$_REQUEST`) are reset automatically between requests, so a worker does not leak one visitor’s form data into the next visitor’s request, but `$_ENV` is not reset, and anything your own code deliberately stores outside a superglobal will persist unless you clear it yourself. The documentation is candid about why that matters: in its words, PHP “was not originally designed for long-running processes,” and the standard mitigation is a periodic worker restart to bound the memory a long-lived process can quietly accumulate. RoadRunner takes a related approach from a different direction, as a Go-based application server managing a pool of long-lived PHP workers that persist across requests rather than being torn down after each one.

The JIT: a real feature, a narrow benefit

PHP 8.0 added a Just-In-Time compiler, and the RFC that introduced it is unusually direct about what it does and does not help. The compiler translates hot code paths to machine code at runtime, which helps CPU-bound, numeric work substantially. It does comparatively little for a typical web request, because a typical request’s time is spent waiting on a database or the network, not executing arithmetic, and the RFC ships its own benchmark showing exactly that gap on a realistic workload.

WordPress requests per second, JIT RFC's own benchmark



PHP JIT RFC, wiki.php.net/rfc/jit, WordPress request benchmark, data checked 2026-09-17

Three hundred and twenty-six requests per second with the JIT enabled, compared to three hundred and fifteen without it, on a WordPress request. That is a real, measured, single-digit-percent difference on the workload most PHP installations actually run, and it comes from the RFC that shipped the feature, not from a critic. The JIT is not nothing: it is the reason the numeric-code comparisons in Performance look the way they do. It is just not the reason a web application built on PHP gets faster.

Concurrency without threads

Four options, in the order this book will keep using, cover how PHP code achieves concurrency inside a single process: AMPHP, ReactPHP, Swoole (or OpenSwoole), and, as a language feature rather than a library, Fibers. AMPHP is a set of libraries built around the same event-loop model as the other two, aimed at the same class of I/O-bound concurrency problem: many things waiting at once, handled by one process that never blocks entirely on any single one of them. ReactPHP describes itself, in its own documentation, as a low-level, event-loop-based library aimed at servers and clients that need to hold open many concurrent connections. Swoole describes itself as an event-driven, coroutine-based networking engine written in C for PHP that converts PHP's normally blocking I/O calls into non-blocking calls under its own scheduler. Fibers, added in PHP 8.1, are cooperative: a fiber suspends and resumes itself via `Fiber::suspend()` and `Fiber::resume()`, and only one fiber's code runs at any instant. All of this is concurrency, the ability to have many logical tasks in flight and switch between them while one waits on I/O. None of it is parallelism, the ability to have two tasks actually executing at the same instant on two CPU cores.

> **The limit.** PHP has no native multithreading in userland. The `parallel` and `pthreads` PECL extensions exist and give real OS-thread parallelism, but they are third-party, narrowly adopted, and worth naming honestly as "exists, but niche," not folded silently into the concurrency story above and not omitted from it either. If your workload needs two CPU-bound tasks running at the same literal instant inside one process, nothing in this chapter's list does that, and Where PHP Is the Wrong Choice is where that limit gets its full treatment.

What to verify yourself. On any machine with PHP installed, run:

```
php -v
php -i | grep -i opcache
```

The first line tells you which PHP version and, usually, which SAPI (FPM, CLI, or something else) you are looking at. The second tells you whether opcache is actually loaded on that specific install, not whether it theoretically could be: a hosting environment's marketing page is not evidence; a grep of your own `phpinfo()` output is.

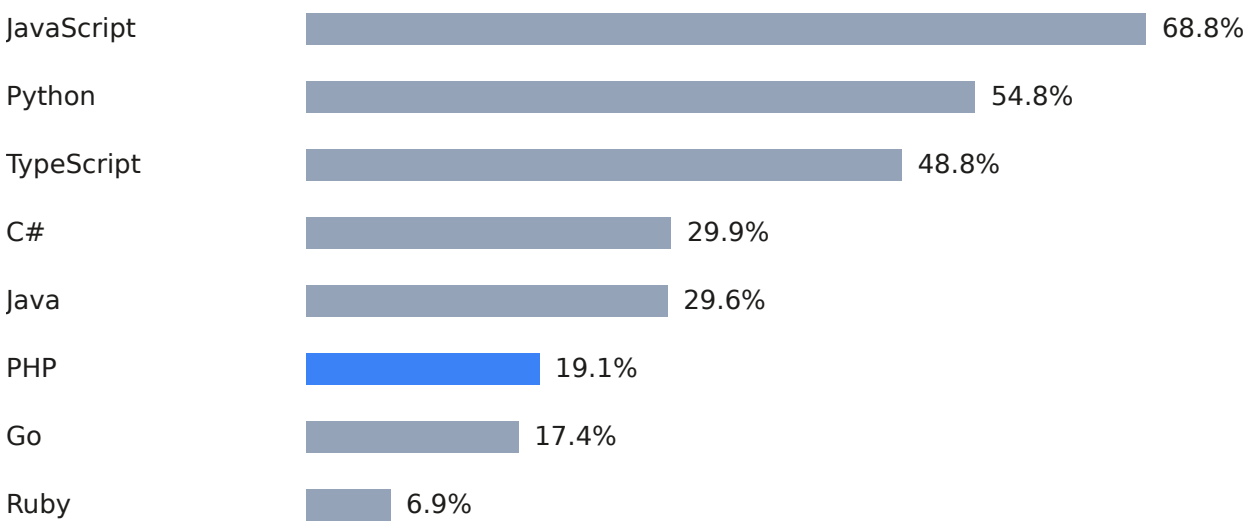
The runtime model in this chapter, shared-nothing by default, bendable by specific serving layers, concurrent but not parallel, is the fact everything downstream in this book is built on. The next question is not how PHP executes a single request, but who is actually running it, at what scale, and on what evidence, which is where Who Runs PHP picks up.

Who Runs PHP

PHP runs the majority of the web by server footprint and a minority of the web by developer headcount, and both of those numbers are true at the same time because they measure different things. An evaluator who has heard only one of them has heard half the picture, and the half you have heard probably depends on who told you.

By server footprint: W3Techs, which crawls live websites rather than surveying people, puts PHP at **69.9 percent of all websites whose server-side language it can detect**, as of 2026-09-17. That figure moves continuously as the crawl updates, so treat it as a snapshot rather than a constant, the same caveat The Reputation already put on the number. By developer headcount, the picture looks different. The Stack Overflow Developer Survey 2025, drawing on 24,759 professional-developer respondents, found 19.1 percent reporting they use PHP, against 68.8 percent for JavaScript, 54.8 percent for Python, 48.8 percent for TypeScript, 29.9 percent for C#, and 29.6 percent for Java.

Professional developers using each language, 2025



Stack Overflow Developer Survey 2025, 24,759 professional respondents, data year 2025

Read that chart the way it is meant to be read, not the way either side of an argument would prefer. PHP sits behind JavaScript, Python, TypeScript, C#, and Java, and ahead of Go and Ruby, at 17.4 and 6.9 percent respectively. It is not the dominant language among professional developers, and it is not a niche one either; it occupies a specific, middling, well-populated position, and a hostile reader who checks the survey directly will find exactly this chart, not a rounder or friendlier one.

The two numbers are not in tension. They are counting different populations. The W3Techs figure counts running websites, most of which were built once and are maintained lightly if at all; a huge fraction of the web is smaller commerce sites, blogs, and content platforms built on a CMS, and CMS platforms, among them Drupal, Joomla, TYPO3, and WordPress, run on PHP. The Stack

Overflow figure counts developers who chose to answer a survey aimed at professionals actively writing code day to day, which is a different, smaller, and more self-selected population than "everyone who has ever deployed a website." Both numbers are honest. Neither one, alone, answers "is PHP the language my new project's target developer pool already knows," which is closer to the developer-headcount question, or "will my application be running alongside a lot of other PHP infrastructure," which is closer to the server-footprint question.



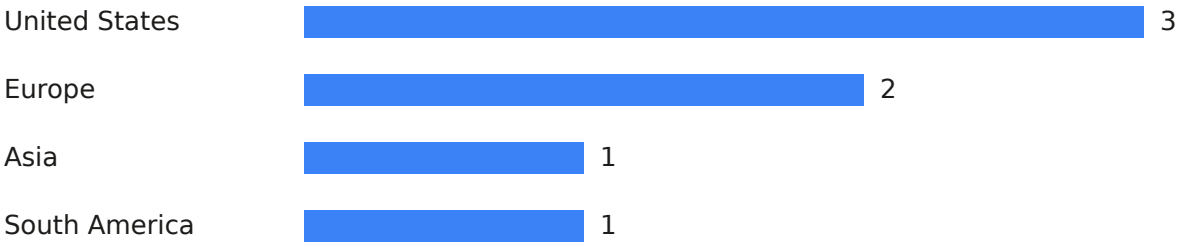
For a hiring decision specifically, the developer-headcount number is usually the more useful one, because it is closer to answering "how many people who already know this language could I plausibly hire," while the server-footprint number is closer to answering "how much of the surrounding infrastructure, from hosting control panels to CMS plugins to other companies' internal tools, will assume PHP is in play." Both questions are legitimate parts of a due-diligence review, and conflating them into a single "PHP is popular" sentence loses the distinction an evaluator actually needs.

A primary-sourced case at scale

Numbers about the web in aggregate are useful, but an evaluator also wants to know what PHP looks like under real, sustained load, not just how often it appears. Wikimedia is one of the few organizations that documents this in detail, in its own words, on its own engineering wiki rather than through a third party. MediaWiki, the software behind Wikipedia and its sister projects, is roughly 70 percent PHP by Wikimedia's own account, amounting to close to two million lines of Wikimedia-maintained PHP code. It runs across seven data centers, three in the United States, two in Europe, one in Asia, and one in South America, behind a layered caching setup (Varnish and

Apache Traffic Server in front, APCu and Memcached behind the application) that keeps a top-ten global site responsive under continuous, worldwide traffic.

Wikimedia data centers running MediaWiki, by region



Wikimedia wikitech.wikimedia.org/wiki/Wikimedia_infrastructure, retrieved 2026-09-17

That geographic spread matters for a reason beyond redundancy: it demonstrates that PHP's shared-nothing request model, explained in The Runtime, is not a small-site limitation. A request in each of those seven locations boots, runs, and tears down independently, with no shared memory between them by design, and the aggregate system still serves one of the most-visited sites on the internet, with traffic routed geographically so a visitor is answered by the nearest data center rather than by a single central one. Wikimedia's own documentation is a living wiki page rather than a dated report, so this book cites it by retrieval date, 2026-09-17, and treats it the way any rolling source should be treated: accurate as of the day it was read, not fixed for all time.

What the case study does not do is stand in for a survey. One organization, however large, is an existence proof, not a distribution. It tells you PHP can be operated at that scale by an organization willing to invest in the caching and routing layers around it; it does not tell you what fraction of PHP deployments actually reach that scale, and this book will not imply otherwise by presenting one example as typical.

Who is correctly excluded

Two names come up reflexively in any "who runs PHP at scale" conversation, and both belong on the other side of the ledger. Meta and Slack are commonly assumed to run PHP because of PHP's historical association with Meta's early engineering culture. Both actually run Hack, a related but distinct language, on HHVM, a runtime that diverged from PHP compatibility years ago. Counting them as PHP evidence would be the kind of cherry-picking this book's own neutrality rule exists to prevent, so they are named here only to be excluded, not to be claimed.

> **The limit.** Beyond Wikimedia, verifiable, primary-sourced, large-scale PHP case studies are harder to produce than the reputation would suggest. Several of the widely repeated figures about other major PHP-running platforms trace back to sponsor blog posts, conference talks summarized secondhand, or vendor material years out of date rather than to a company's own current, dated statement. This book would rather print one case study it can fully stand behind than several it cannot, and it names that trade-off here instead of quietly padding the chapter.

What to verify yourself. Open w3techs.com/technologies/details/pl-php and read the live percentage against the one printed above. Then open the Stack Overflow Developer Survey's technology results for the current year and check where PHP actually sits against the languages your own team already knows, since that comparison, not the aggregate web-share number, is usually the one that matters for a hiring decision.

Adoption answers who is running PHP and at what footprint. It does not answer how fast any of it actually is, which is a separate, frequently conflated question, and the next one this book takes on directly in Performance.

Performance

How fast PHP is depends entirely on what you measure: on a typical web request it is fast because most of the request's time is spent waiting on a database or the network, not on PHP itself, and on CPU-bound computation it is faster than Python and slower than compiled or JIT-mature languages such as Go, Rust, Java, and C#. Both halves of that sentence are load-bearing, and a benchmark that only shows you one of them is not showing you the whole picture.

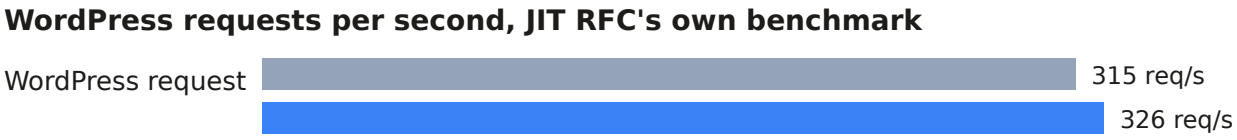
"Fast" is doing three different jobs in most performance conversations, and they need separating before you look at a single number. Throughput, how many requests a fixed amount of hardware can serve per second, is what capacity planning cares about. Latency, how long one specific request takes to answer, is what a user waiting on a page load cares about. Raw computational speed, how quickly a fixed piece of arithmetic finishes, is what a background job doing heavy numeric work cares about. PHP's shared-nothing, request-per-process model, covered in The Runtime, makes the first two mostly a function of how much I/O your request does and how well your serving layer is tuned, not a function of the language's raw arithmetic speed. The third is where the language itself is the bottleneck, and it is the one this chapter spends the most care on, because it is also the one most often measured with a synthetic microbenchmark and then quietly applied to a claim about web performance, a substitution this chapter tries hard not to make.

The benchmark this book cannot use

The industry's standard cross-language web-framework benchmark, TechEmpower Framework Benchmarks, was discontinued on 24 March 2026, and its GitHub repository is now archived. The last completed round, Round 23, published 17 March 2025, covered more than 330 framework implementations across languages, and the discontinuation is stated here plainly, with its date, rather than quietly cited an outdated round as if it were still current: a reader who goes looking for a newer round will find only the archive notice, and this book would rather explain the gap than pretend it does not exist. This is also why this chapter leans more heavily on two other sources: a benchmark shipped by PHP's own JIT proposal, and an independently maintained cross-language benchmark suite that is still active.

What the JIT actually buys you on a web request

The Runtime already introduced PHP's Just-In-Time compiler and the RFC's own admission that it helps CPU-bound code far more than typical web code. The same benchmark is the cleanest performance evidence in this book, precisely because it comes from the people who built the feature and had every incentive to show it in the best light.



PHP JIT RFC, wiki.php.net/rfc/jit, WordPress request benchmark, data checked 2026-09-17

Three hundred and twenty-six requests per second with the JIT on, three hundred and fifteen without it, on a WordPress request. That is roughly a three percent difference, on a real application, from the RFC that introduced the feature. The reason is architectural, not a tuning failure: a WordPress request spends most of its time in database queries and I/O, not in the numeric computation the JIT accelerates. If your evaluation criteria include "will enabling the JIT make my web application meaningfully faster," the honest answer, from PHP's own numbers, is usually not much, and any benchmark claiming otherwise on a web workload deserves a second look at what it actually measured.

The JIT is not pointless; it is aimed at a narrower target than "PHP" in general. A CLI script doing image processing in a tight loop, a background worker running a numeric simulation, a queue consumer doing heavy string or array transformation with no network call in the middle: these are the workloads where the compiler's gains actually show up, because they are CPU-bound in the way a WordPress request is not. An evaluator deciding whether to enable the JIT should ask what fraction of the actual workload is arithmetic versus waiting, not whether the JIT exists.

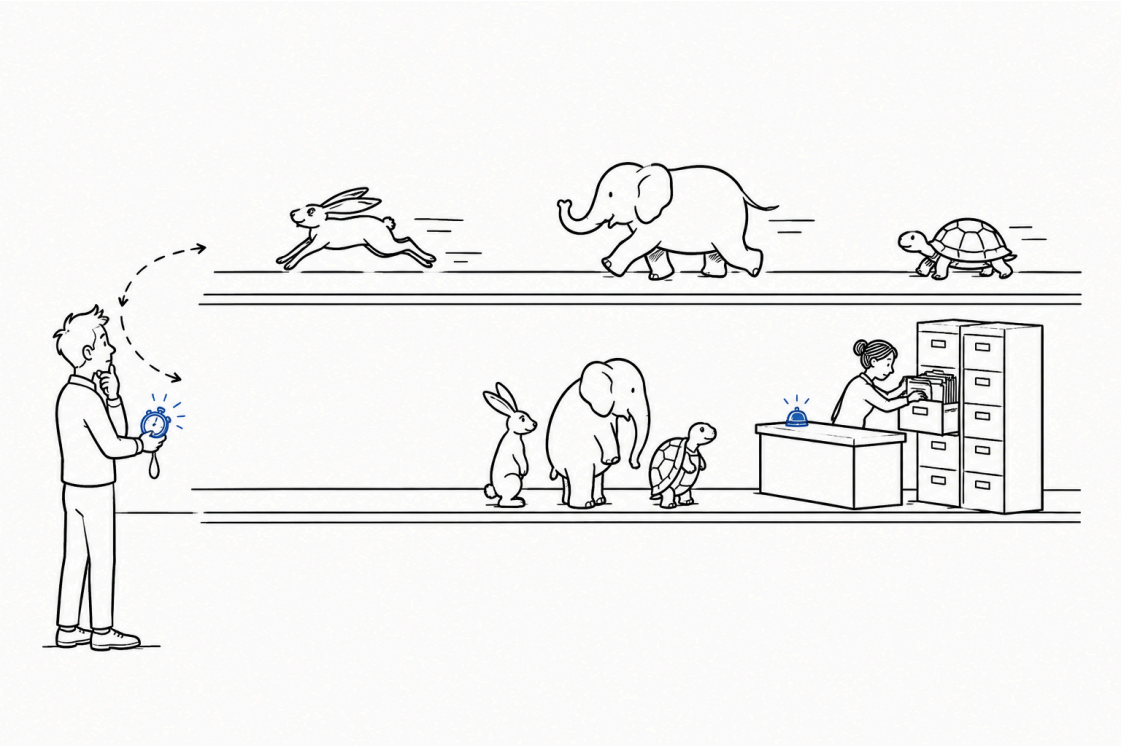
The comparison PHP usually loses on the wrong axis

The most common performance criticism of PHP compares it to Python and calls it slow. The Computer Language Benchmarks Game, an independently maintained, actively updated cross-language suite, tells a different story on CPU-bound tasks. In its tests of PHP 8.4.1 against Python 3.13 on tasks including fannkuch-redux, n-body, and spectral-norm, three classic CPU-bound benchmarks with no I/O to hide behind, PHP came out **faster than Python on most of them, by roughly one and a half to two times**.

Task	What it stresses	Which language was faster
fannkuch-redux	Permutation generation, tight loops	PHP
n-body	Floating-point arithmetic, simulation	PHP
spectral-norm	Matrix and vector math	PHP

This matters for how you frame the limit, not just for the number itself. **"PHP is slow" is not true relative to Python; it is true relative to compiled and JIT-mature languages.** A comparison that stops at Python and calls the result settled is comparing PHP to the wrong reference class, and a due-diligence reviewer who runs this benchmark herself will find the same ranking this book

prints. This is also the finding most likely to surprise a reader who arrived with the reputation described in The Reputation already in mind: a language remembered as slow turns out to beat one of the languages most associated with well-regarded data and scripting work, on the specific axis both languages are actually being measured on.



> **The limit.** PHP trails compiled and JIT-mature languages, Go, Rust, Java, and C#, on sustained CPU-bound throughput, and this book does not have a single, clean, primary-sourced multiplier to put a number on that gap the way it can for Python. That is a real limit of the available evidence, not a claim that the gap is small: a workload that is genuinely CPU-bound, doing heavy numeric computation with little I/O to overlap it with, is a workload PHP is architecturally not the fastest tool for, and Where PHP Is the Wrong Choice treats that case directly.

What to verify yourself. The Computer Language Benchmarks Game publishes its full, current results at its own site, and its methodology page explains exactly what hardware and what version of each language it used; rerun a task that resembles your own workload rather than trusting the summary table. If you want to see PHP's request-handling throughput for yourself instead of trusting either benchmark, the most honest test is the one described in The Evaluation: run your own representative workload against your own PHP version, with opcache and JIT in the state you would actually deploy them.

Raw speed on a single machine is one part of a capacity question. The part that usually matters more in production is what happens as load grows and you add machines, which is where Scale picks up.

Scale

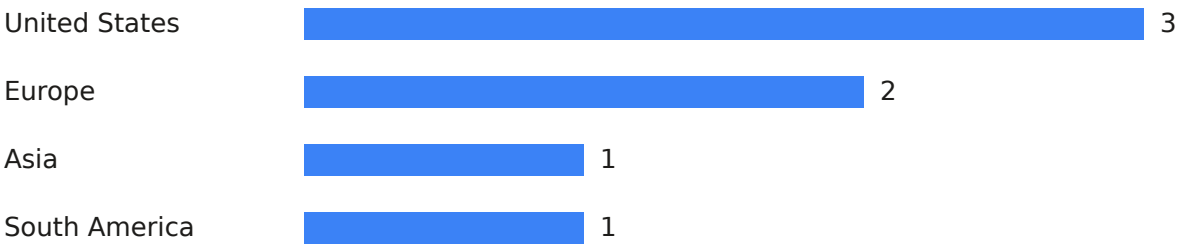
PHP scales the way its runtime model predicts: horizontally, by adding more identical, interchangeable workers behind a load balancer, because a shared-nothing process has no internal state that needs to be reconciled with any other process. That is the entire mechanism. It is not exotic, and it is not PHP-specific in principle, but PHP's default execution model, described in The Runtime, makes it close to the only option, which turns out to be a feature more often than a constraint.

Why shared-nothing scales in one direction easily

PHP-FPM's process model is worker-based: a pool of processes, each one handling a single request at a time, none of them holding memory in common. Because no worker knows anything a sibling worker does not also have equal access to reconstruct, a load balancer can send any request to any worker, on any machine, with no coordination step. Doubling capacity is adding a second identical machine running a second identical worker pool, not redesigning how the first one shares state, because there was never any state to share in the first place. This is the specific property that makes stateless HTTP services, the majority of what PHP is used to build, straightforward to scale without the class of bugs that comes from two processes disagreeing about what they both think is true.

Who Runs PHP already covers the clearest primary-sourced demonstration of this at real scale: Wikimedia's MediaWiki, roughly 70 percent PHP by the project's own account, running across seven data centers on three continents, each one independently capable of serving a request because nothing about PHP's execution model requires them to coordinate with each other beyond what the caching and routing layers explicitly decide to share.

Wikimedia data centers running MediaWiki, by region



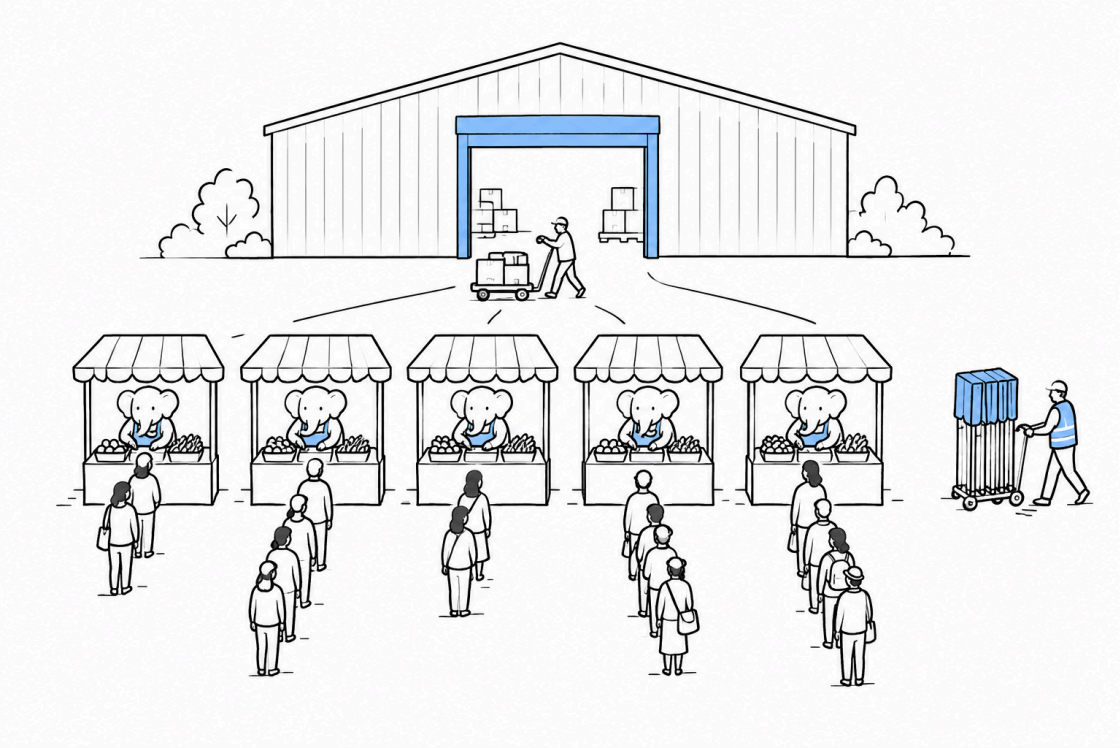
Wikimedia wikitech.wikimedia.org/wiki/Wikimedia_infrastructure, retrieved 2026-09-17

The scale lesson in that distribution is not the total count of data centers; it is that none of them needs to know what the others are doing to correctly answer a request. Add an eighth data center in a new region, and the change is a routing decision and a deployment, not an architectural re-

write, because every worker in every location is already running the identical, self-contained unit of shared-nothing PHP. A stateful architecture would have to solve replication and consistency to make the same move; this one does not, because it never had shared state to replicate in the first place.

What has to move outside the process for this to work

Shared-nothing scaling has a precondition that is easy to state and easy to violate by accident: nothing that needs to persist between requests can live inside the PHP process. A user's session, a piece of data cached to avoid a repeated expensive query, an uploaded file waiting to be processed: all of it has to live somewhere external to the worker that first touched it, a database, a dedicated cache layer, or shared storage, because the next request for that same user has no guarantee of landing on the same worker, or the same machine. This is not a limitation unique to PHP; it is the standard shape of any stateless web tier. It is named plainly here because a team new to PHP's specific defaults sometimes discovers it the hard way, by storing something in a plain variable or an in-process cache and being surprised when it vanishes on the very next request.



Scaling within one worker, not just across many

Horizontal scaling by adding workers is not the only lever. The Runtime already introduced four ways PHP code can hold open many things at once inside a single process: AMPHP, ReactPHP, Swoole or OpenSwoole, and Fibers as a language feature. For a workload dominated by waiting, many open connections to slow backends, a chat relay, a webhook fan-out, these tools let one process serve far more concurrent work than one request per worker would allow, because the

process is not blocked while it waits; it is free to make progress on something else and come back. This does not replace horizontal scaling, and it does not give PHP OS-level parallelism, the limit named plainly in The Runtime. It does mean the choice in front of you is not only "how many workers," but also "how much concurrency can one worker itself absorb before you need another one," and the answer depends on which of these tools, if any, sits underneath your application.

Containers change the packaging, not the model

Most new PHP deployments today run inside containers rather than directly on a machine, and it pays to be precise about what that changes and what it does not. A container gives each worker pool its own isolated filesystem and dependency set, which makes deployment repeatable; it does not change the shared-nothing execution model underneath, because PHP-FPM inside a container still boots, runs, and tears down one request at a time exactly as it does outside one. Scaling a containerized PHP service is the same horizontal operation described above, run by an orchestrator instead of by hand: more identical pods behind the same kind of load balancer, each one as interchangeable as the last. This book does not have a clean, primary-sourced, vendor-neutral figure for how PHP's container image size or cold-start time compares to another language's, so it will not print one; the architectural claim that matters, no proprietary orchestration layer is required to run PHP this way, is verifiable directly from PHP-FPM's own documentation rather than from a benchmark.

> **The limit.** Moving state out of the PHP process does not make the cost of that state disappear; it moves the scaling problem to the database and cache layer, which now has to handle the aggregate load of every worker on every machine. A PHP application that scales its web tier cleanly can still be bottlenecked entirely by a database that was not scaled alongside it, and this book will not claim PHP's own scaling story extends to a piece of infrastructure PHP does not control. Long-running worker-mode servers, covered in The Runtime, also reintroduce a scaling concern shared-nothing PHP-FPM does not have: a worker that stays booted for hours accumulates memory the way any long-lived process can, which is exactly why FrankenPHP's own documentation recommends periodic restarts rather than claiming the concern away.

What to verify yourself. If you run PHP-FPM today, check its pool configuration for `pm.max_children`, the setting that caps how many worker processes one pool will run at once, and compare it against the memory available on the machine; a pool sized without that arithmetic is a common, self-inflicted scaling ceiling that has nothing to do with the language. If you are evaluating a worker-mode server instead, ask specifically how it restarts workers and on what schedule, and treat the answer as a capacity-planning input, not an afterthought.

Scaling a web tier and staffing the team that maintains it are different problems, and the second one is where the evaluation usually gets harder to reduce to a number. Who Maintains It is where this book turns to talent, testing practice, and the people question.

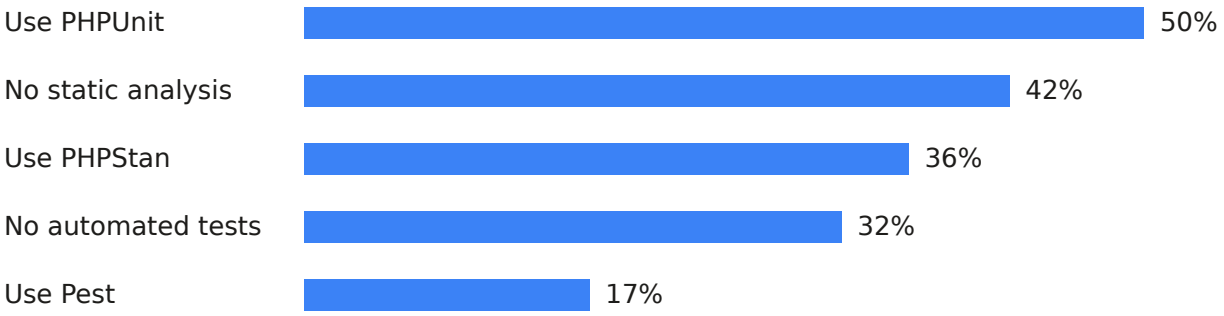
Who Maintains It

PHP has a large, experienced developer population that is mostly not planning to leave, and the practices that population reports around testing and static analysis vary widely enough that "PHP developers" is not a single, uniform quality signal you can hire against. Both halves matter to a staffing decision, and the second one is the kind of finding a defensive version of this book would have left out.

What working PHP developers report about themselves

JetBrains' "The State of PHP 2025," part of a broader State of Developer Ecosystem survey fielded between April and June 2025, drew 1,720 PHP-specific respondents out of 24,534 developers surveyed overall. JetBrains states its own caveat plainly, and this book repeats it every time the survey is cited: results may skew toward JetBrains product users, which is not the same population as "all PHP developers everywhere." With that caveat attached, the self-reported figures are still informative. Eighty-eight percent report three or more years of PHP experience. Fifty-eight percent say they do not plan to migrate away from PHP in the next year; of the smaller group that does plan to leave, Go and Python are the most commonly named destinations.

PHP developers reporting each practice, self-reported



JetBrains The State of PHP 2025, blog.jetbrains.com, 1,720 PHP respondents, fielded April-June 2025, may skew tow

Read this chart carefully, because it is the one place in this book where the good news and the limit are the same data. Half of respondents report using PHPUnit, and seventeen percent report using Pest, the two testing frameworks this book names without ranking one above the other. Thirty-six percent report using PHPStan, up nine points from the year before, a gain that stands on its own. But thirty-two percent report no automated testing at all, and forty-two percent report no static analysis at all. A hiring manager reading only the adoption numbers would conclude PHP has a maturing testing culture. A hiring manager reading only the gap numbers would conclude the opposite. Both readings are true of different, overlapping parts of the same self-reported

population, and an evaluation that only quotes one side is choosing its conclusion before looking at the data.

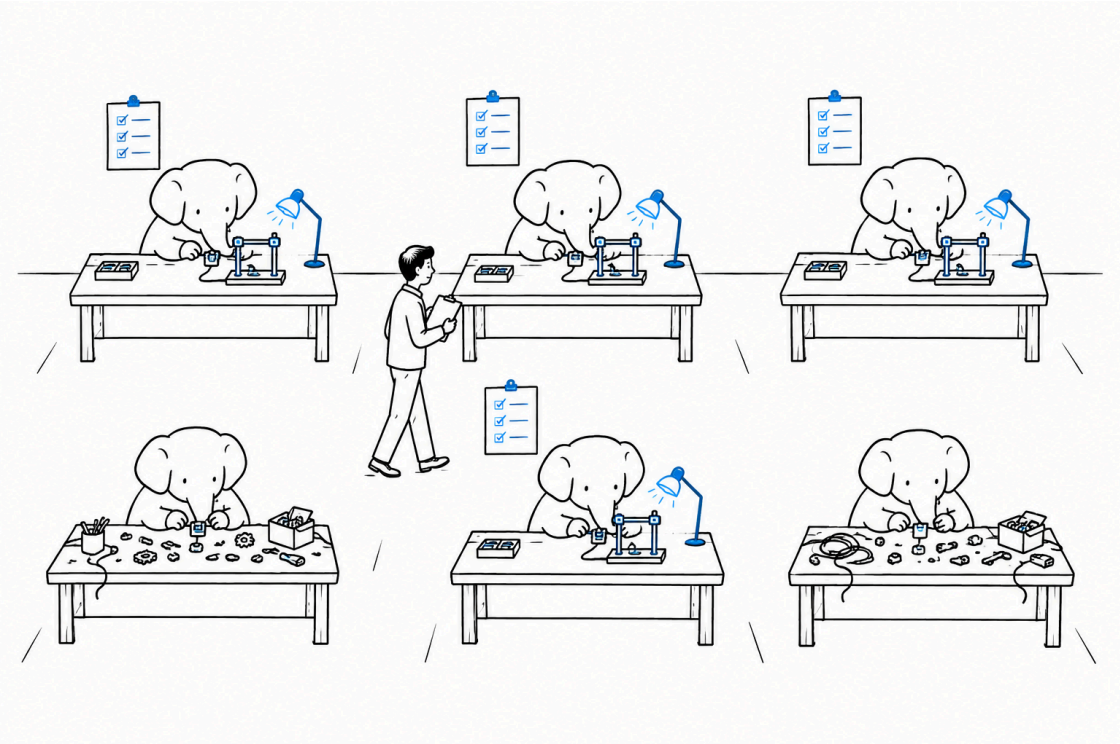
For a due-diligence review of a specific codebase rather than the language in general, this finding has a direct, practical use: it means "the team writes PHP" tells you almost nothing about maintenance quality on its own, and "can I see the test suite and the static-analysis configuration" is a more useful question to ask in an acquisition or a vendor review than any question about the language itself. The tooling to answer that question exists and is standard enough to expect: PHP-CS-Fixer and PHP_CodeSniffer for code style, PHPStan and Psalm for static analysis, Pest and PHPUnit for tests, all named here in the same alphabetical, non-competing order this book uses throughout. A codebase with none of them configured is not evidence that PHP itself is immature; it is evidence about that specific codebase, and the distinction is worth insisting on in the room where the decision gets made.

Version currency, self-reported

The same survey found 89 percent of respondents multi-selecting PHP 8.x as a version they work in, 33 percent still touching 7.x, and 8 percent still touching PHP 5.6 or older. This is a developer's self-report of what they personally work in day to day, not a measurement of the installed base of running servers; Governance and Versions covers the server-side figure, from a different source, and the two numbers should not be blended into one, because they answer different questions about currency: one is about what a developer's job asks of them, the other is about what is actually deployed and exposed to the internet.

Where local talent availability fits, honestly

A due-diligence review in a regulated or non-anglophone market often asks a narrower question than "how many PHP developers exist worldwide": can you hire locally, without going remote-first or asking someone to relocate. This book does not have a clean, geography-broken-out, primary-sourced figure to answer that precisely, and it would rather say so than borrow one from a marketing page. What it can say, structurally rather than statistically, is that PHP has been a standard component of general web-development education, in university courses, in bootcamps, in vocational tracks, across most regions for two decades, which is the same adoption longevity already established in The Reputation and Who Runs PHP. That is a reasonable basis for expecting junior-to-mid PHP talent to exist locally in most markets; it is not a substitute for checking your specific local market before you commit a hiring plan to it.



The language's core itself is maintained the same way: by a distributed base of contributors to the php-src project, overseen by the governance process described in Governance and Versions, rather than by a single vendor with a roadmap it controls unilaterally. That distinction matters for a different kind of risk than hiring risk: it means no single company's acquisition, layoff, or strategy change can unilaterally end the language's maintenance, the way it can for a project owned outright by one vendor.

Onboarding speed is part of a maintenance question too, and it is kept separate from the survey data above because it comes from the tooling rather than from a self-report. A new hire's first task on a PHP codebase is usually running Composer (Cost covers what it manages) against a lockfile that pins every dependency to an exact, reproducible version. That is a mechanical, verifiable property of the toolchain, not a claim about how good any particular team's engineering culture is, and it is the same category of guarantee `npm` gives a Node.js team or `cargo` gives a Rust team: a new machine can reproduce the exact dependency graph the rest of the team is already running, which is a smaller but real contributor to how quickly a new maintainer becomes productive.

> **The limit.** This book cannot give you a precise, sourced, PHP-versus-other-language salary or job-demand comparison. The two obvious sources, Stack Overflow's and JetBrains' own survey data, render the relevant breakdowns as interactive charts rather than as fetchable text at the precision this book requires, and a widely repeated older figure for the total global PHP developer population is stale enough that this book declines to repeat it rather than reprint a number it cannot stand behind. Where precision is not available, this book says so instead of rounding a guess into an authoritative-looking statistic.

What to verify yourself. Read the current JetBrains State of Developer Ecosystem survey yourself and check whether the caveat about its own respondent skew is still attached the way this book

has attached it; a survey that drops its own caveat in a later year is a signal worth noticing. Separately, github.com/php/php-src publishes its contributor activity in the open: look at the actual pace and diversity of commits yourself before forming a view, in either direction, about the health of the language's core maintenance.

Talent answers who can build and maintain a PHP application. What that talent, that infrastructure, and that testing culture actually cost to run is the next question, and Cost is where the evidence for it lives.

Cost

PHP itself is free and permissively licensed, its package ecosystem is free to use at a scale of close to half a million published packages, and the real cost of a PHP application is almost entirely in hosting, staffing, and maintenance discipline, not licensing. The more useful cost question for an evaluator is not "what does PHP cost" but "what does PHP's architecture let you avoid paying for," and this chapter spends most of its attention there.

The license, precisely

PHP is distributed under the PHP License, an OSI-approved, permissive license that does not require you to open-source anything you build with it. Today's supported floor for this book, PHP 8.4, ships under version 3.01 of that license. A newer license, version 4, based on the Modified BSD (BSD-3-Clause) text, applies starting with PHP 8.6, which had not shipped as of this writing; treat that as an upcoming change to track, not as something already governing the version you are most likely deploying today. Neither license requires a fee, a registration, or a usage report to anyone.

What "free" actually includes

Packagist, the public registry that Composer (PHP's dependency manager) resolves packages against, lists close to half a million published packages, with roughly 5.8 million published versions between them, and a cumulative install count approaching 200 billion since the registry's launch in April 2012.

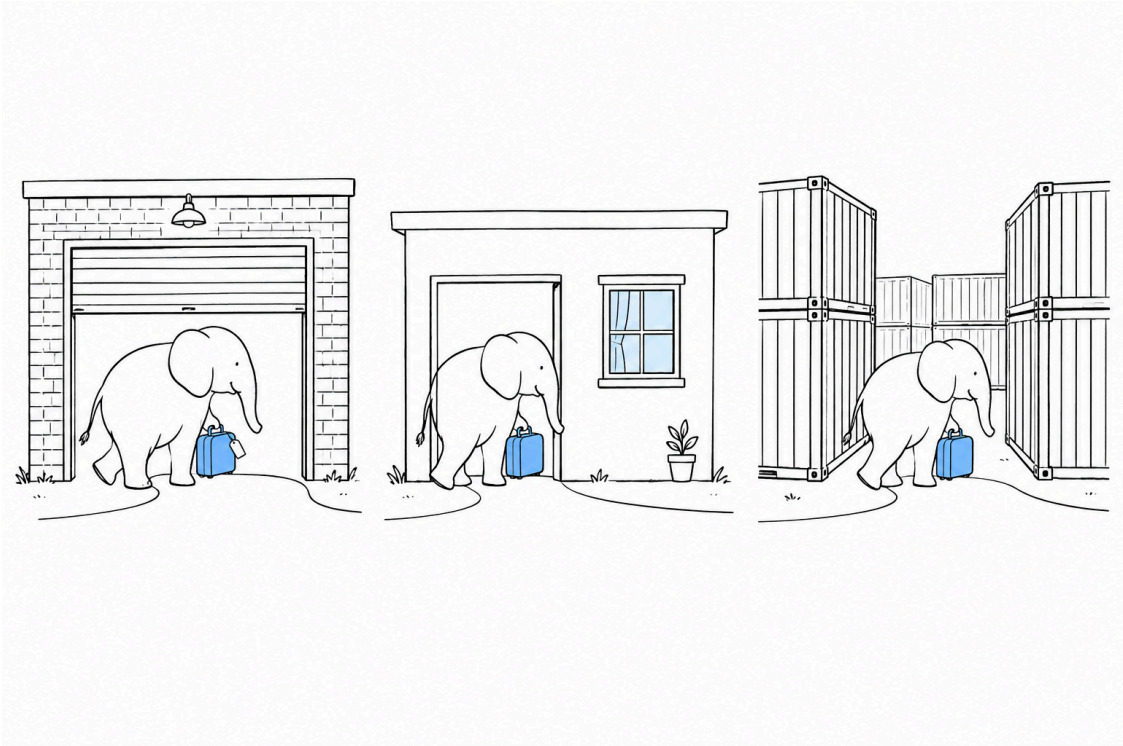
Metric	Approximate figure
Published packages	~468,000
Published versions	~5.8 million
Cumulative installs since April 2012	~200 billion

Those are live counters, so treat the exact digits as a snapshot rather than a fixed fact, the same discipline this book has applied to every other continuously updated figure. The magnitude is the point: a dependency you need for a PHP project is very likely already published, versioned, and installable through one command, at no cost beyond the time it takes to audit what you are pulling in.

Publishing to Packagist costs nothing and requires no vetting beyond what Composer itself checks mechanically: every install is verified against a hash recorded in the project's lockfile, so a depen-

dency cannot silently change between the version your team reviewed and the version that actually gets installed. That is a real, useful guarantee, and it is also a narrower one than full supply-chain provenance: a hash confirms the code has not changed since it was locked, not that the code was safe when it was first added. The responsibility for that first review still sits with the team pulling the dependency in, in PHP's ecosystem the same as in any other language's.

What you are not paying for: proprietary orchestration



PHP-FPM's process model, described in The Runtime, requires only a POSIX-compliant operating system, a web server, and the PHP runtime itself to serve requests at scale. It does not require a proprietary serverless orchestration layer, and nothing about the architecture forces you to adopt one. That is a real, verifiable, architectural fact, not a claim about any specific hosting company, and this book states it that way on purpose: the same application can run on a bare-metal machine you own, a minimal virtual server, or a container orchestrator, without re-architecting it to fit any one of them.

That property has a specific, current relevance for organizations operating under data-localization requirements or a sovereign-cloud mandate, where compute has to stay on infrastructure that meets a jurisdiction's own residency rules rather than on whichever platform is most convenient. A runtime that does not require a specific vendor's managed orchestration to function is easier to relocate onto infrastructure that meets those rules, because the constraint being satisfied is about where the machine sits, not about which proprietary service the application was written against.

> **The limit.** This portability claim is about the PHP runtime, not about your whole operational stack. A team that has also adopted a cloud-specific managed database, a cloud-specific managed queue, or observability tooling built around one vendor's agent has already reintroduced the same lock-in at those layers, regardless of which language sits in front of them. State this argument as "the runtime does not force cloud lock-in," not as "a PHP application is portable," because the second claim is broader than the evidence supports.

What PHP gives you for security, by default

Part of PHP's reputation, addressed once in The Reputation, includes an assumption that the language is inherently insecure. The current language ships specific, checkable primitives that argue against treating that as still true by default. `password_hash` and `password_verify` provide a maintained, algorithm-agnostic password-hashing API, so an application does not need to hand-roll one. Libsodium bindings have been bundled with PHP since version 7.2, giving direct access to audited cryptographic primitives without a third-party extension. PDO's prepared statements separate a query's structure from its data at the driver level, which is the standard, effective defense against SQL injection when it is actually used. None of these facts make an application secure by themselves; a team that ignores them can still write an insecure application in any language. They do mean the tools to do it correctly are already in the standard library, not an extra purchase or a third-party dependency you have to find and vet first.

What this chapter cannot price

There are two cost questions this book would like to answer with a number and cannot, and it says so here rather than guessing. A clean, sourced, PHP-versus-other-language hosting-cost comparison does not exist without relying on a specific hosting vendor's own marketing figures, which this book's sourcing rules exclude; the architectural argument above is offered instead of a number precisely because the number is not available without that compromise. The salary and job-demand gap already named in Who Maintains It applies here too: this book cannot tell you, with a defensible source, whether a PHP developer costs more or less to hire than a developer in another language in your specific market, and a number invented to fill that gap would be worse than no number at all.

What to verify yourself. Check packagist.org/statistics directly for the current package, version, and install counts, and compare them to the figures printed here; the gap is a live demonstration of how fast the ecosystem is still growing. Then check php.net/license directly against whichever PHP version you are actually planning to run, since the license version that applies depends on that specific version, not on the one this book treats as current.

Licensing and hosting cost are the parts of total cost of ownership an evaluator can price directly. The parts that are harder to price, how the language is governed, how often it changes, and what

happens when a version reaches its support deadline, are exactly what Governance and Versions covers next.

Governance and Versions

PHP's language changes go through a public proposal-and-vote process that requires a supermajority for anything that changes the language itself, releases ship on a fixed annual schedule with a four-year support window, and core maintenance is funded by a foundation independent of any single company's product roadmap. That combination, predictable process, predictable cadence, and funding that does not depend on one vendor staying interested, is what an evaluator is actually asking about when they ask "who is in charge of this language."

The release cadence, and what it commits to

PHP ships one feature release per year, in late November. Each branch then gets two years of active support, meaning bug fixes and security fixes both, followed by two more years of security-only support, four years of support in total before a branch reaches end of life.

Version	Released	Active support ends	Security support ends
PHP 8.2	8 Dec 2022	31 Dec 2024	31 Dec 2026
PHP 8.3	23 Nov 2023	31 Dec 2025	31 Dec 2027
PHP 8.4	21 Nov 2024	31 Dec 2026	31 Dec 2028
PHP 8.5	20 Nov 2025	31 Dec 2027	31 Dec 2029

This table is the single most useful piece of evidence in this chapter, because it converts "is PHP well maintained" into a checkable date specific to the version you are actually running. A branch past its security-support end date is not receiving fixes for newly discovered vulnerabilities, from anyone, and that is a fact about your specific deployment, not an opinion about the language.

PHP major version in use across PHP-running websites

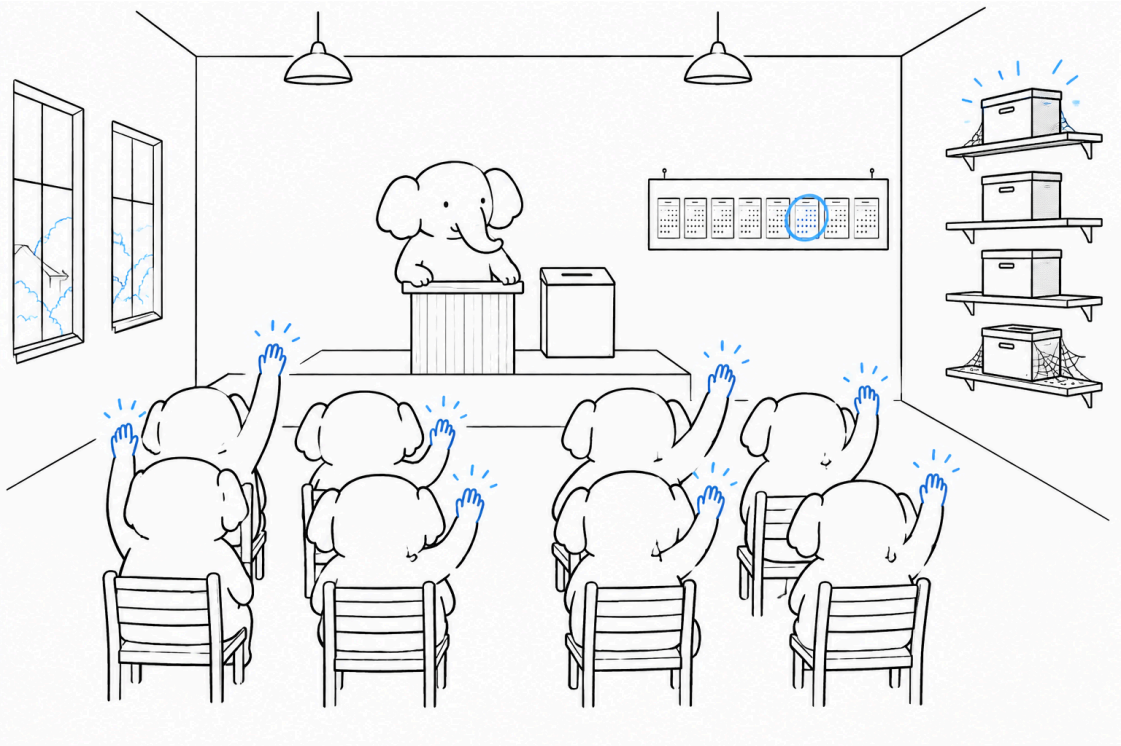


PHP 8.x (64.1%)
PHP 7.x (28.1%)
PHP 5.x (7.8%)
PHP 4.x (0.1%)

W3Techs, w3techs.com/technologies/details/pl-php, data checked 2026-09-17

What the RFC process actually requires

Language changes are proposed and voted on in the open, at a public wiki dedicated to the process, and PHP's own rule requires a two-thirds majority for a change to the language itself to pass, a higher bar than a simple majority. That threshold exists precisely to slow down changes that would break existing code or fragment the language's own consistency, at the cost of moving slower than a benevolent-dictator model might. Neither side of that trade is free, and this book states it both ways rather than presenting the higher bar as an unambiguous good: a supermajority requirement protects stability, and it also means a change with strong majority support but not overwhelming support can still fail.



Who funds the work

The PHP Foundation was established in November 2021, in response to concern that core-language maintenance had become dependent on the availability of a small number of individual contributors' paid time from their employers, rather than on a funding base broad enough to survive any one employer's decision to reduce that time. The Foundation publishes its funding on a public ledger, showing ongoing contributions from a range of technology companies and open-source organizations with a stake in PHP's continued maintenance. The mechanism matters more than any single contributor's name: funding spread across multiple independent organizations, visible on a public ledger rather than disclosed only in a press release, is a different and more durable arrangement than maintenance that depends on one company's continued goodwill.

Funding of this kind pays for specific, unglamorous work: reviewing and merging RFCs, triaging and responding to security reports against the language's own engine, and doing the release engi-

neering that keeps the annual cadence in the table above actually landing in late November rather than slipping. PHP maintains a dedicated process for handling security reports against the core language, separate from the RFC process used for new features, precisely because a vulnerability report needs a faster, more controlled path than a public debate about a language feature does. This book does not have a dated, primary-sourced count of how many core vulnerabilities that process has handled in a recent window to print here, and it would rather leave the number out than borrow one from a third-party aggregator that was not the process's own account.

Compare this to the alternative most evaluators have implicitly in mind: a language or framework whose direction is set entirely by one company, where a change ships because that company decided to ship it, not because it survived a public vote. That model can move faster in the short term. It also means the language's future is exactly as stable as that one company's continued interest, a risk PHP's multi-sponsor funding and public RFC process are both explicitly structured to avoid, even at the cost of the slower-moving supermajority bar described above.

Where the tidy cadence meets a messier reality

The release table above describes what PHP ships. It does not describe what is actually running. W3Techs' crawl of live, PHP-running websites, as of 2026-09-17, found 64.1 percent on the PHP 8.x line, 28.1 percent still on 7.x, whose last branch reached end of life in November 2022, and 7.8 percent still on PHP 5.x, out of support since 2018 or 2019. A residual 0.1 percent was still detectable on PHP 4.

That is a market-reality caveat, not a claim about the current language, and the distinction needs holding onto carefully: PHP 8.x's own support story, laid out in the table above, is exactly as predictable as this chapter describes. What a meaningful share of the installed base actually does with that predictability is a separate question, about specific organizations' operational discipline, not about governance. It pairs with, and should never be blended with, the self-reported version split in Who Maintains It: one number describes servers actually running on the internet, the other describes what developers say they personally work in, and a chart that mixes the two is making a comparison that was never apples to apples.

> **The limit.** A governance process this predictable only protects you if your own organization keeps pace with it. The support table above is a promise about what PHP Foundation-backed maintainers will do on a fixed schedule; it is not a promise that your specific deployment will be upgraded before its branch's security support ends. Roughly a third of the PHP-running web is currently not keeping that pace, and Where PHP Is the Wrong Choice treats the risk of inheriting one of those deployments directly.

What to verify yourself. Check php.net/supported-versions.php directly against the version you are running or evaluating, since the dates in the table above will eventually be superseded by a newer release cycle. Then check w3techs.com/technologies/details/pl-php for the current version-

fragmentation split, and compare it to the one printed here to see how much, if at all, the installed base has caught up.

Governance and version support describe how PHP is maintained when everything goes as designed. Where PHP Is the Wrong Choice is where this book turns to what happens when the language itself, not a specific deployment's neglect, is genuinely the wrong tool, and makes that case as plainly as it has made every other one.

Where PHP Is the Wrong Choice

PHP is the wrong choice for true parallel computation, for machine learning and numeric data science work, and for an organization that cannot commit to upgrading before a branch's security support ends; it is a weaker but not impossible choice for native desktop or mobile applications and for workloads with heavy, sustained CPU computation. This chapter exists so the rest of this book can be believed, and it names each of these plainly, with the evidence behind it, rather than softening the ones that complicate a simple story.

No true parallelism, by design

The Runtime already named this limit once: PHP has no native multithreading in userland. Fibers, added in PHP 8.1, give cooperative concurrency, one fiber suspending itself so another can run, which is genuinely useful for I/O-bound work but is not parallelism, because only one fiber's code executes at any given instant. The `parallel` and `pthreads` PECL extensions give real operating-system-thread parallelism, and they are real, current, and worth naming rather than omitting; they are also third-party and narrowly adopted, which is a fair reason to treat them as an exception rather than a default answer. If your workload is defined by needing two CPU-bound tasks to run at the literal same instant inside one process, nothing in PHP's default toolchain does that, and no serving layer or async library changes it, because the limit is in the language runtime itself, not in how you deploy it.

The same architectural gap shows up in real-time, bidirectional communication. A WebSocket connection that needs to stay open and push data to a client without the client asking for it does not fit cleanly into PHP-FPM's one-request-in, one-response-out model; it becomes workable through the same async libraries, Swoole or OpenSwoole in particular, already introduced in The Runtime, or through a serving layer built for long-lived connections. That is a real, working answer, not a dead end, but it is an added piece of infrastructure a request-response-native language like PHP needs for a job some other runtimes handle as a first-class case.

The CPU-bound comparison, stated correctly

Performance already showed the counterintuitive half of this limit: PHP is faster than Python on CPU-bound microbenchmarks, not slower. The defensible version of the "PHP is slow" limit is narrower and still real: PHP trails compiled and JIT-mature languages, Go, Rust, Java, and C#, on sustained CPU throughput.

Comparison	What the evidence shows
PHP vs Python, CPU-bound tasks	PHP faster, by roughly 1.5 to 2x
PHP vs compiled/JIT-mature languages	PHP slower; this book has no single clean multiplier to print

A workload that is genuinely CPU-bound, heavy numeric simulation, large-scale in-memory data transformation, anything where the bottleneck is arithmetic rather than waiting on a network or a database, is a workload PHP is architecturally not the fastest available tool for. That does not mean PHP cannot do it; it means the honest comparison set is Go, Rust, Java, and C#, not Python, and a vendor or a team that frames this limit against the wrong reference class is either uninformed or shading the comparison in PHP's favor.

Native mobile and desktop apps: weaker than assumed, not solved

The traditional answer here was simple: PHP has no first-party toolkit for building native desktop or mobile applications, full stop. That answer is a year or two out of date. NativePHP, a community project built specifically on top of Laravel, one of the frameworks named in this book's fixed list, packages a PHP application into desktop and mobile app-store builds, and its own documentation claims production deployments as of 2026. Naming it once, here, with that date, is more honest than pretending the gap is still absolute; naming it repeatedly, or treating it as equivalent in maturity to Swift, Kotlin, or a mature cross-platform toolkit, would overstate what a single, young, framework-specific community project has actually proven at scale. The honest framing is that no first-party toolkit exists, a community solution does, and it is unproven at the maturity level a decision maker would want before betting a flagship product on it.



Machine learning and numeric data science

This gap is real, and this book would rather make the ecosystem-density argument than borrow a usage statistic that is not actually about PHP. There is no PHP equivalent to NumPy, pandas, PyTorch, or TensorFlow with comparable maturity, community size, or hardware-acceleration support. You can check this yourself directly: search Packagist's own categories, or PHP's own extension index, for numeric computing or machine-learning tooling, and compare what you find to the equivalent search in Python's package index. The gap is not a matter of opinion, and it does not require a borrowed survey statistic to demonstrate; it is visible directly in what each ecosystem has actually built.

Long-running processes: partially solved, precisely

The Runtime and Scale both already cover this: FrankenPHP's own documentation states plainly that PHP "was not originally designed for long-running processes," and worker-mode serving mitigates the resulting memory-growth risk without eliminating it, which is why periodic worker restarts are the documented, standard mitigation rather than an admission of failure. That is a fair, non-defensive summary of a real limit, sourced from the vendor's own candor about it rather than from a critic.

No generics

PHP has no generics in the language itself. PHPStan and Psalm, the two static-analysis tools named throughout this book, both read the same `@template` docblock convention to approximate generic types, which gives you real, useful type-checking at development time, entirely outside the language engine: nothing about a generic type declared this way is enforced when the code actually runs. A codebase that leans on this convention is choosing a real, working substitute, not the language feature itself, and the distinction matters if your evaluation criteria specifically require compile-time generic enforcement rather than an opt-in static-analysis approximation of it.

The market-reality caveat, not a language caveat

Governance and Versions already gave you the number: roughly 28 percent of PHP-running websites are still on the 7.x line, unsupported since November 2022, and close to 8 percent are still on PHP 5, unsupported since 2018 or 2019. Inheriting one of those deployments, through an acquisition, a legacy internal tool, or a vendor relationship, is inheriting a real security and maintenance liability. This is not a claim about the current language, which is actively maintained on the schedule described in Governance and Versions; it is a claim about what a specific organization did, or failed to do, with an old version of it, and an evaluator doing due diligence on an existing codebase should treat the two as entirely separate risk assessments.

> **The limit, stated once more, directly.** None of the gaps in this chapter are solved by picking a different PHP framework, and this book will not imply otherwise. Where they are solved at all, they are solved by adding a second, purpose-built tool alongside PHP, not by PHP alone.

What to verify yourself. Read PHP's own Fibers documentation and decide for yourself whether cooperative concurrency actually covers your workload, rather than taking this chapter's summary as the final word. Then run the specific numeric or data-heavy task closest to your actual workload against both a PHP implementation and whatever language you would otherwise use, the same rerun-it-yourself standard Performance already asked of you, since a real benchmark of your own task beats any general comparison this book can print.

Every limit in this chapter has the same shape: PHP plus one additional, purpose-built piece of infrastructure, not a wholesale replacement. The Evaluation closes this book by turning that shape, and every sourced figure before it, into something you can check yourself, on your own machine, before you decide anything.

The Evaluation

Every figure in this book was checkable when it was written, and every figure in this book will have moved by the time you read it; this chapter exists to hand you the checks themselves, not another number to trust in their place. A due-diligence reviewer does not finish an evaluation by reading someone else's conclusions. She finishes it by rerunning the parts of the evidence that matter to her own decision, and this chapter is built entirely out of the moments across the nine chapters before it where this book already asked you to do exactly that.



What your own machine already knows

Before checking anything about the wider ecosystem, check the specific PHP install you would actually be deploying against. The following script, kept deliberately small, reports the facts The Runtime spent a chapter explaining: which version you are running, whether opcache is actually loaded rather than theoretically available, and whether the JIT is enabled.

```php
<?php

declare(strict_types=1);

function reportRuntimeStatus(): array
{
    return [
        'php_version' => PHP_VERSION,
        'sapi' => PHP_SAPI,
        'opcache_loaded' => extension_loaded('Zend OPcache'),
        'jit_enabled' => function_exists('opcache_get_status')
            && (opcache_get_status(false)['jit']['on'] ?? false),
    ];
}

foreach (reportRuntimeStatus() as $fact => $value) {
    printf("%-15s %s\n", $fact, var_export($value, true));
}
```

Run it with `php status.php` against any environment you are evaluating, including one a vendor hands you with assurances attached. A vendor's claim that "opcache and JIT are both on" is a sentence; this script's output is a fact about that specific machine, and the two are not the same kind of evidence.

Every check this book already asked of you, in one place

Each earlier chapter ended with something to verify in an afternoon. Rerunning them together, in order, reconstructs this book's entire evidence base from live sources rather than from this book's printed snapshot of them.

```
# Chapter 1 and 8: current adoption and version fragmentation
# https://w3techs.com/technologies/details/pl-php

# Chapter 2: your own install's runtime state
php -v
php -i | grep -i opcache

# Chapter 3: professional developer adoption by language
# https://survey.stackoverflow.co (current year's technology results)

# Chapter 4: CPU-bound benchmarks, still actively maintained
# https://benchmarksgame-team.pages.debian.net/benchmarksgame/

# Chapter 6: the PHP core's own contributor activity
# https://github.com/php/php-src

# Chapter 7: live package, version, and install counts
# https://packagist.org/statistics

# Chapter 8: the current support window for your PHP version
# https://php.net/supported-versions.php
```

None of these checks require special access, a paid account, or insider knowledge. That is deliberate: a book that asked you to trust a number you could not independently reach would be asking you to trust it the same way the reputation in The Reputation asked people to trust secondhand impressions of PHP instead of the current language itself, and this book has spent nine chapters arguing against exactly that habit.

Reading a vendor's own claim the way this book reads its own

The method matters more than the specific list above, because the list will age and the method should not. When a vendor, a framework's own documentation, or a colleague hands you a performance number, a security claim, or an adoption statistic, ask the same three questions this book asked of its own research before printing anything. Where does the number come from, in the source's own words, not a summary of it? What exactly was measured, and does that match what you actually care about, the way Performance had to separate a JIT microbenchmark from a web-request benchmark before either one meant anything? And what does the source itself say it does not know, since a source willing to state its own limits is more trustworthy than one that has none?

A worked example, using this book's own weakest citations

The most useful test of this method is to point it at this book itself. Who Maintains It cites a JetBrains survey that states its own limitation plainly: results may skew toward JetBrains' own product users. That is a source that passes the test, not because it is unbiased (no survey of self-selected respondents is) but because it tells you which way it likely leans before you have to guess. Performance does the opposite kind of honest work: it names a benchmark series, TechEmpower, that stopped publishing, with the exact date, rather than quietly citing its last round as if nothing had changed. And Governance and Versions leaves a handful of figures, the PHP Foundation's exact funding totals among them, out of its firm claims entirely, because they were confirmed only through a secondary account at the time of writing, not through the Foundation's own page directly. Sources marks each of those explicitly as pending direct verification instead of printing a number this book cannot stand behind. A source that tells you what it does not know, in its own words, is doing the same work this book has tried to do throughout, and it is the single clearest signal to look for in anything you are handed during your own evaluation.

> **The limit.** A checklist is only as good as the day it was run. Every command and every URL above will return a different number next quarter than it returns today, the same way every figure printed earlier in this book will have moved by the time you read it. This chapter cannot give you a permanently current answer; it can only give you a repeatable way to get a current one, which is the more durable thing to hand you.

What to verify yourself, one more time. Pick the single figure in this book that your decision most depends on, whichever chapter it came from, and rerun its source directly before you act on it. If it still says what this book said it says, you have independent confirmation. If it does not, you have caught this book being stale before it cost you anything, which is exactly the outcome the whole approach was built to make possible.

This book opened by separating PHP's reputation from the language currently in front of you. It closes by handing that separation back to you as a method, not a conclusion: the runtime model, the adoption figures, the performance comparisons, the cost structure, the governance process, and the plainly stated limits are only worth as much as your own willingness to check them again. Sources lists exactly where every figure in this book came from, chapter by chapter, so that checking is never more than one link away.

Appendix A: Sources

Every figure and sourced claim in this book is listed below, grouped by the chapter it appears in, with the publisher, the URL, and the date the underlying data refers to or was checked. Where a figure comes from a page that recomputes continuously (W3Techs, Packagist), the date is a retrieval date, not a publication date, and the chapter text says so at the point the figure is used.

A small number of entries below are marked **pending direct verification**. Each one was confirmed through a search-engine snippet or a secondary account rather than a direct fetch of the primary page at the time of writing, and each one is written into its chapter in a form narrow enough not to depend on a precise figure that has not yet been directly confirmed. They are listed here rather than silently dropped so that whoever finishes this book's production knows exactly what remains to be checked before publication.

Chapter 1: The Reputation

- PHP's share of websites with a known server-side language: 69.9 percent, checked 2026-09-17; 71.8 percent in March 2026; 70.2 percent on 2026-09-03. W3Techs, w3techs.com/technologies/details/pl-php.

Chapter 2: The Runtime

- PHP's execution model (Zend Engine, request-per-process or thread, shared-nothing by default). The PHP manual, current, checked 2026-09-17.
- Opcache bundled since PHP 5.5; loaded as a `zend_extension` via `php.ini`, not active by default from core alone. php.net/manual/en/opcache.installation.php, checked 2026-09-17.
- JIT RFC's own WordPress benchmark: 326 requests/second with the JIT enabled versus 315 without. PHP JIT RFC, wiki.php.net/rfc/jit, checked 2026-09-17.
- `declare(strict_types=1)` and scalar/return type enforcement behavior. The PHP manual, current, checked 2026-09-17.
- FrankenPHP worker mode: superglobals reset automatically between requests except `$_ENV`; the project's own documentation states "PHP was not originally designed for long-running processes" and recommends periodic worker restarts. frankenphp.dev/docs/worker/, checked 2026-09-17.
- RoadRunner as a Go-based application server managing a pool of long-lived PHP workers. **Pending direct verification** of docs.roadrunner.dev; the current description traces to a third-party comparison, not the project's own documentation.

- AMPHP as an event-loop-based library family for I/O-bound concurrency, described generically rather than quoted. **Pending direct verification** of github.com/amphp/amp README; not independently re-checked this pass.
- ReactPHP's self-description as a low-level, event-loop-based library for event-driven programming. github.com/reactphp/reactphp README, checked 2026-09-17.
- Swoole's self-description as an event-driven, coroutine-based concurrency engine written in C for PHP. github.com/swoole/swoole-src README, checked 2026-09-17.
- Fibers (PHP 8.1): cooperative, single-stack coroutines via `Fiber::suspend()` and `Fiber::resume()`, concurrency without OS-level parallelism. php.net/manual/en/language.fibers.php, checked 2026-09-17.
- `parallel` and `pthreads` as third-party PECL extensions offering real OS-thread parallelism. PECL and the PHP manual, general reference, checked 2026-09-17.

Chapter 3: Who Runs PHP

- W3Techs adoption figure: see Chapter 1 entry above.
- Stack Overflow Developer Survey 2025: PHP 19.1 percent, JavaScript 68.8 percent, Python 54.8 percent, TypeScript 48.8 percent, C# 29.9 percent, Java 29.6 percent, Go 17.4 percent, Ruby 6.9 percent, among 24,759 professional-developer respondents. survey.stackoverflow.co/2025/technology, checked 2026-09-17.
- MediaWiki as roughly 70 percent PHP, close to two million lines of Wikimedia-maintained PHP, running across seven data centers (three United States, two Europe, one Asia, one South America) behind Varnish, Apache Traffic Server, and Apache HTTP Server, with APCu and Memcached caching. Wikimedia, wikitech.wikimedia.org/wiki/MediaWiki_at_WMF and wikitech.wikimedia.org/wiki/Wikimedia_infrastructure, living pages, retrieved 2026-09-17.
- Meta and Slack run Hack on HHVM, not PHP. **Pending direct verification** against each company's own current engineering material; treated as established background fact for the purpose of this book's exclusion rule.

Chapter 4: Performance

- TechEmpower Framework Benchmarks discontinued 24 March 2026, repository archived; last completed Round 23 published 17 March 2025, covering more than 330 framework implementations. techempower.com/blog/2025/03/17/framework-benchmarks-round-23/ and github.com/TechEmpower/FrameworkBenchmarks (archived), checked 2026-09-17.
- JIT RFC WordPress benchmark: see Chapter 2 entry above.
- Computer Language Benchmarks Game: PHP 8.4.1 versus Python 3.13, suite version 25.03, PHP faster on fannkuch-redux, n-body, and spectral-norm by roughly 1.5 to 2 times.

benchmarksgame-team.pages.debian.net/benchmarksgame/fastest/python3-php.html, checked 2026-09-17.

Chapter 5: Scale

- PHP-FPM's worker-based, shared-nothing process model. php.net/manual/en/install.fpm.php, checked 2026-09-17.
- Wikimedia data-center distribution: see Chapter 3 entry above.

Chapter 6: Who Maintains It

- JetBrains "The State of PHP 2025": 1,720 PHP-specific respondents of 24,534 developers surveyed overall, fielded April-June 2025, with the survey's own caveat that results may skew toward JetBrains product users. Figures used: 88 percent with 3+ years of PHP experience; 58 percent not planning to migrate away within a year (Go and Python most-cited among those who would); self-reported version split 89 percent PHP 8.x, 33 percent 7.x, 8 percent 5.6 or earlier; PHPStan usage 36 percent, up 9 points year over year; PHPUnit 50 percent; Pest 17 percent; 32 percent report no automated testing; 42 percent report no static analysis. blog.jetbrains.com/phpstorm/2025/10/state-of-php-2025/, checked 2026-09-17.
- Composer's lockfile-based dependency resolution as a reproducibility guarantee. getcomposer.org documentation, general reference, checked 2026-09-17.

Chapter 7: Cost

- PHP License version 3.01 applies to PHP 8.4 and 8.5; License version 4 (Modified BSD / BSD-3-Clause) applies from PHP 8.6 onward. php.net/license/index.php, checked 2026-09-17.
- Packagist statistics: approximately 468,201 published packages, 5,822,307 published versions, and approximately 199.8 billion cumulative installs since 13 April 2012. packagist.org/statistics, checked 2026-09-17, live counter.
- Composer's hash-based install verification against the project lockfile. getcomposer.org documentation, general reference, checked 2026-09-17.
- PHP-FPM's minimal hosting requirements: see Chapter 5 entry above.
- `password_hash` and `password_verify` as the standard password-hashing API. php.net/manual/en/function.password-hash.php, current, checked 2026-09-17.
- Libsodium bindings bundled with PHP since version 7.2. php.net/manual/en/book.sodium.php, current, checked 2026-09-17.
- PDO prepared statements separating query structure from data. php.net/manual/en/pdo.prepared-statements.php, current, checked 2026-09-17.

Chapter 8: Governance and Versions

- Release cadence and support windows: one feature release per year in late November; two years of active support followed by two years of security-only support. PHP 8.2 released 8 Dec 2022 (active support ended 31 Dec 2024, security support ends 31 Dec 2026); PHP 8.3 released 23 Nov 2023 (active until 31 Dec 2025, security until 31 Dec 2027); PHP 8.4 released 21 Nov 2024 (active until 31 Dec 2026, security until 31 Dec 2028); PHP 8.5 released 20 Nov 2025 (active until 31 Dec 2027, security until 31 Dec 2029). php.net/supported-versions.php, checked 2026-09-17.
- RFC process: proposed and voted on the public wiki, with a two-thirds majority required for a language change. **Pending direct verification** of wiki.php.net/rfc/change_required_votes_to_two_thirds for exact wording and adoption date.
- The PHP Foundation established November 2021. **Pending direct verification** of the Foundation's own founding account at thephp.foundation; the account used here was traced through a sponsor's blog post rather than the Foundation's own history page.
- A public funding ledger recording ongoing contributions from multiple independent organizations. **Pending direct verification** of current cumulative totals at opencollective.com/phpfoundation; this book states the ledger's existence and transparency without printing specific dollar figures that were not directly confirmed.
- A dedicated process for handling security reports against the core language, separate from the feature RFC process. **Pending direct verification** of php.net/security.
- W3Techs version fragmentation: PHP 8.x 64.1 percent, 7.x 28.1 percent, 5.x 7.8 percent, 4.x 0.1 percent of PHP-running websites. w3techs.com/technologies/details/pl-php, checked 2026-09-17.

Chapter 9: Where PHP Is the Wrong Choice

- Fibers, `parallel`, and `pthreads`: see Chapter 2 entries above.
- PHP-versus-Python CPU-bound comparison: see Chapter 4 entry above.
- NativePHP as a community project built on Laravel, packaging PHP applications for desktop and mobile app stores, with the project's own documentation claiming production deployments as of 2026. nativephp.com/docs/, checked 2026-09-17.
- Absence of a PHP equivalent to NumPy, pandas, PyTorch, or TensorFlow: an ecosystem-density observation, checkable directly via Packagist's own category search and PHP's extension index rather than a single dated citation.
- FrankenPHP's own statement on long-running processes: see Chapter 2 entry above.
- PHPStan and Psalm both approximating generics through a shared `@template` docblock convention, enforced only by static analysis, not the language engine. **Pending direct verification** of phpstan.org/writing-php-code/generics and psalm.dev/docs/annotating_code/templated_annotations/; current sourcing is secondary.
- W3Techs version fragmentation: see Chapter 8 entry above.

Chapter 10: The Evaluation

No new figures. This chapter reruns the live sources already listed above: w3techs.com/technologies/details/pl-php, survey.stackoverflow.co, benchmarksgame-team.pages.debian.net/benchmarksgame, github.com/php/php-src, packagist.org/statistics, and php.net/supported-versions.php. The runtime-status code sample uses `PHP_VERSION`, `PHP_SAPI`, and `opcache_get_status()`, all documented in the PHP manual, current, checked 2026-09-17.

Appendix B: Release Timeline

PHP ships one feature release per year, in late November. Each branch then receives two years of active support, bug fixes and security fixes both, followed by two years of security-only support: four years of support in total from release to end of life. The table below is the same one printed in Governance and Versions, reproduced here on its own for quick reference.

Version	Released	Active support ends	Security support ends
PHP 8.2	8 Dec 2022	31 Dec 2024	31 Dec 2026
PHP 8.3	23 Nov 2023	31 Dec 2025	31 Dec 2027
PHP 8.4	21 Nov 2024	31 Dec 2026	31 Dec 2028
PHP 8.5	20 Nov 2025	31 Dec 2027	31 Dec 2029

Source: php.net/supported-versions.php, checked 2026-09-17. This page updates as new versions ship and old ones retire; treat the table above as a snapshot taken on that date, and check the live page directly before making a decision that depends on an exact date.

What “active” and “security-only” actually mean

During active support, a branch receives both bug fixes and security fixes, and is the version the PHP project itself recommends running. Once active support ends, a branch moves to security-only support: it will still receive a fix for a newly discovered vulnerability, but not for an ordinary, non-security bug. Once security support ends, a branch receives nothing further from the project, regardless of what is found in it afterward. A production deployment still running a branch past its security-support end date is not a deployment with an outdated feature set; it is a deployment with no path to a fix for whatever is discovered in it next.

Reading this table against what is actually running

Governance and Versions and Where PHP Is the Wrong Choice both use the same, separate figure for context: as of 2026-09-17, W3Techs found 64.1 percent of PHP-running websites on the 8.x line this table covers, 28.1 percent still on the 7.x line, whose last branch left security support in November 2022, and 7.8 percent still on PHP 5.x, unsupported since 2018 or 2019. This table describes what the PHP project commits to shipping and supporting. It does not describe what your own organization, or a vendor’s, is actually running, which is a fact only a direct check of that specific deployment can establish.

As of September 2026

PHP 8.6 had not shipped as of this writing. This book does not project a release date for it, in keeping with the same discipline applied to every other figure here: an announced plan is described as announced, not treated as a fact until it ships. The one confirmed detail tied to that release is covered in Cost: PHP License version 4 applies starting with PHP 8.6, superseding License 3.01, which governs the versions this table currently treats as supported.

Appendix C: Vocabulary

Terms are defined the way this book uses them, in plain language, with a pointer to the chapter where each one is explained in context.

Composer. PHP's dependency manager. Resolves a project's required packages against Packagist, records the exact versions installed in a lockfile, and verifies each install against a hash so a dependency cannot silently change underneath a project after it was reviewed. See Cost.

Coroutine. A unit of execution that can pause itself and hand control back, then resume later from the same point, without the operating system tearing it down and rebuilding it. PHP's Fibers and libraries such as ReactPHP and Swoole are built around this idea. See The Runtime.

Fibers. A PHP 8.1 language feature giving cooperative concurrency: a fiber suspends itself via `Fiber::suspend()` and resumes via `Fiber::resume()`. Only one fiber's code executes at any instant, which makes this concurrency, not parallelism. See The Runtime.

FPM (FastCGI Process Manager). PHP's standard worker-process manager: a pool of processes, each handling one request at a time, each one shared-nothing with respect to every other. The traditional, most widely deployed way PHP applications are served. See The Runtime and Scale.

JIT (Just-In-Time compiler). Added in PHP 8.0. Translates hot code paths to machine code at runtime. Helps CPU-bound, numeric workloads substantially; helps a typical web request, whose time is mostly spent waiting on I/O, by only a small margin. See The Runtime and Performance.

Opcache. A bundled `zend_extension`, present since PHP 5.5, that caches compiled bytecode in shared memory so it does not have to be recompiled on every request. Must be explicitly loaded in `php.ini`; not automatically active from PHP core alone. See The Runtime.

Packagist. The public package registry Composer resolves dependencies against. Free to publish to and free to use. See Cost.

PSR (PHP Standards Recommendation). A shared convention, adopted voluntarily by framework and library authors through a standards process rather than mandated by the language itself, covering things like autoloading, logging interfaces, and HTTP message shapes, so independently written libraries can be relied on to interoperate. See The Reputation.

RFC (Request for Comments). PHP's public process for proposing and voting on changes to the language itself, conducted on a dedicated wiki. A change to the language requires a two-thirds majority to pass. See Governance and Versions.

SAPI (Server API). The interface between the PHP engine and whatever is running it, FPM, the CLI, or another server module. `PHP_SAPI` reports which one a given process is running under. See

The Evaluation.

Shared-nothing. PHP's default execution model: each request boots the application, runs, sends a response, and is torn down, with no memory or state carried over to the next request. The property that makes horizontal scaling straightforward, and the reason anything meant to persist has to live outside the PHP process. See The Runtime and Scale.

Static analysis. Checking code for errors, including type mismatches, without running it. PHPStan and Psalm are the two static-analysis tools this book names; both also approximate generic types through a shared docblock convention that the language engine itself does not enforce. See Who Maintains It and Where PHP Is the Wrong Choice.

Worker mode. A serving approach, offered by FrankenPHP and RoadRunner, that keeps a PHP application booted in memory across many requests instead of tearing it down after each one. Superglobals are reset automatically between requests; anything else your code deliberately persists is not, which is why worker restarts are the standard, documented mitigation for long-running memory growth. See The Runtime.