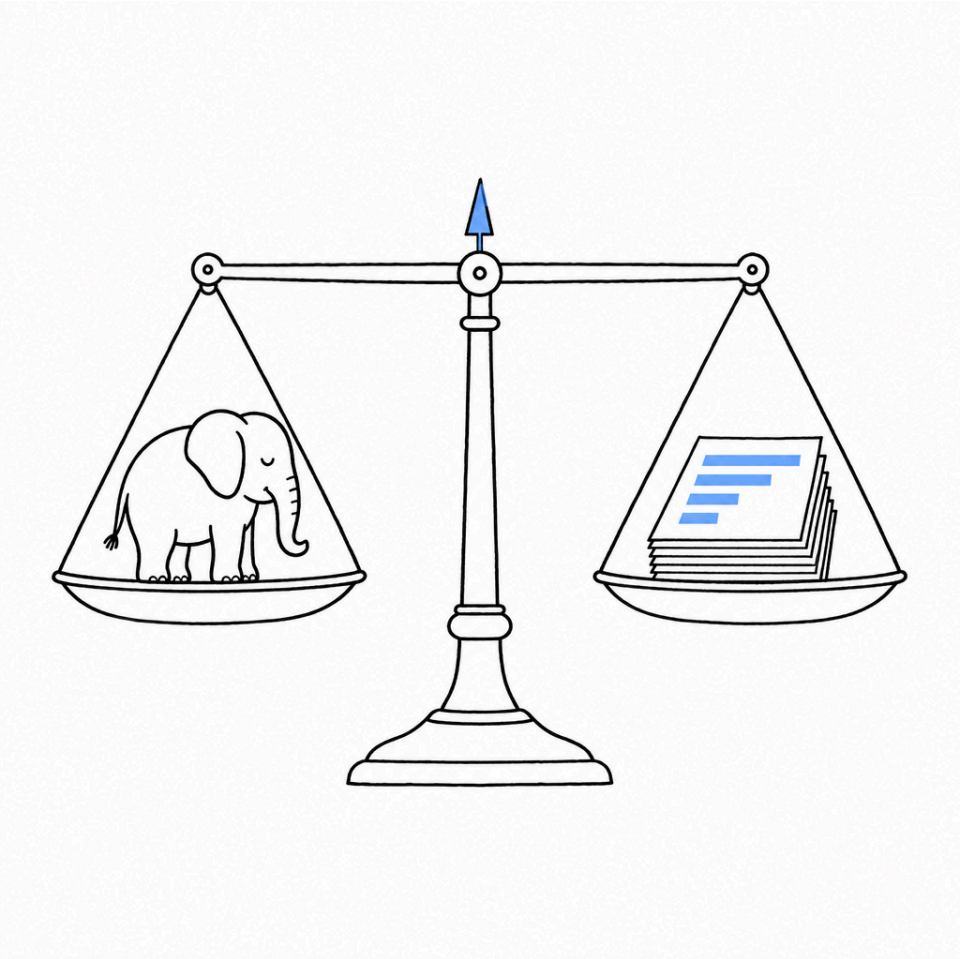
🐘 PHP in 2026, the Facts



You have heard plenty of jokes about PHP. But you have also seen it in a great many job postings, in the stack of a respectable company, in the CMS of a site that is never down and answers fast… And you wonder what to make of all that. Who should you believe? The PHP haters, or the PHP fanboys?

This book is here to give you facts, so that you can form your own opinion of PHP as it is, and decide whether it fits your project, your company, your ambitions.

You will see a lot of figures and charts. Every figure is sourced and dated in the appendix. The comparisons are factual; none is left out because it would put PHP in an uncomfortable position.

Give it ninety minutes to two hours, and the opinion you leave with will be your own.

How to Read This Book

You are being asked to take PHP seriously, and you would rather not. The language is thirty years old, it is associated with the worst code you have ever been shown, and nobody at the conferences you follow talks about it. Yet it keeps turning up: in the stack of a company you respect, on the job board, behind a site that handles more traffic than yours. I wrote this book to settle that contradiction with evidence rather than with enthusiasm.

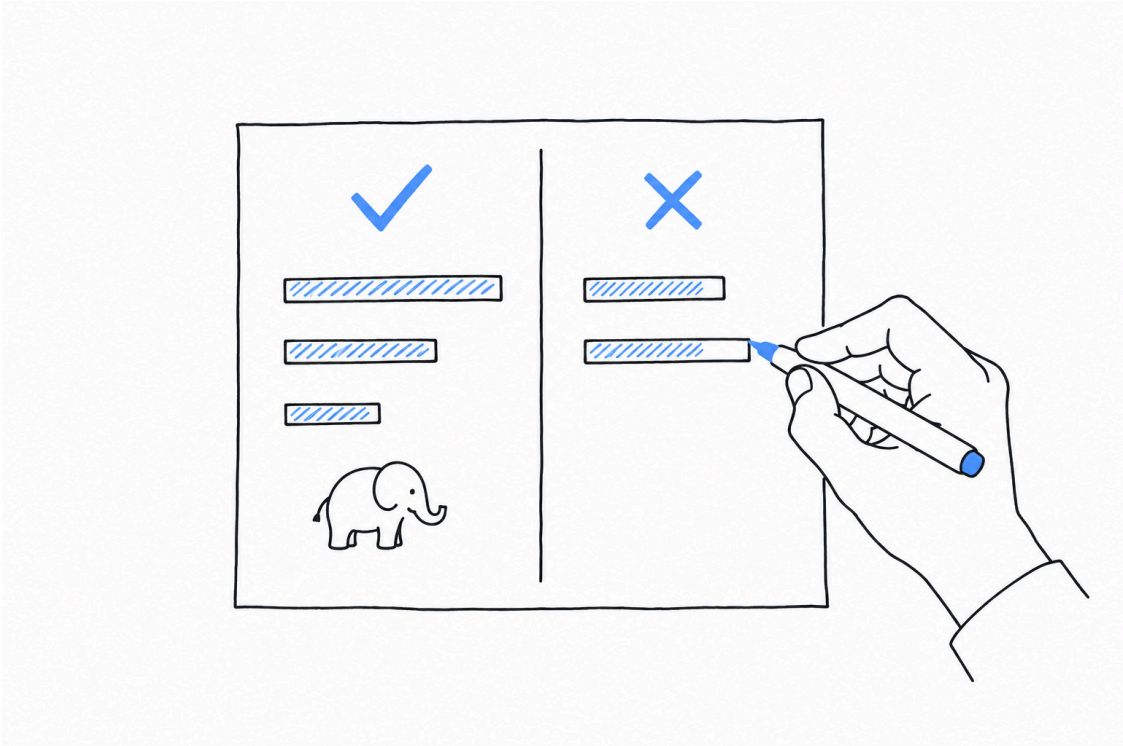
Every figure in this book has a source and a date, and the appendix lists them so you can check each one. A number in the text is followed by where it comes from and when, in a short parenthesis. A vendor figure, a self-reported case study or a synthetic benchmark is labelled as such in the same sentence as the number, and when I do not know something, I say so. You should never have to wonder whether a sentence is a fact or a wish.

Why the doubt is reasonable

The reputation was earned. For its first decade PHP was permissive to a fault: variables appeared from nowhere, `"abc" == 0` was true, errors were printed into the page and execution went on, database queries were built by string concatenation, and the standard library grew one function at a time with whatever name its author preferred that week. A generation learned to program on that PHP and wrote a great deal of it, and much of that code still runs. Most of what you have been shown as “PHP” comes from that period.

The language you would evaluate today is a different object. PHP 7 (2015) rebuilt the engine and added scalar type declarations. PHP 8 (2020) added union types, `match`, named arguments, attributes, enums, `readonly` properties, first-class callables and a JIT compiler, and made the interpreter throw where it used to guess. A release has shipped every year since, in late November or early December. The PHP Foundation has employed core developers since 2021, the package manager is universal, and two static analysers give you most of what a compiler would. The current syntax is in The Language in 2026, and the people who ship it are in Governance and Longevity.

Some of the reputation is still deserved. Strings are byte sequences, the standard library keeps its historical names, there are no generics and no threads in userland, and the runtime is synchronous by default. A large share of the PHP running on the public web is old, because the hosting that runs it is old. Each of these limits sits in the chapter where you would look for it, next to the current practice around it, and Where PHP Is the Wrong Choice collects the cases where the honest advice is to pick something else.



The questions, in order

Each chapter answers one question you would ask, in the order of a due-diligence review, from the outside in.

Footprint asks who runs on PHP and at what scale, and answers with web share data, with the platforms built on it, and with organisations that describe their PHP production in their own words. The Runtime explains the execution model, because most of what PHP does well and most of what it cannot do follow from it. Throughput and Latency and Concurrency give the performance figures, with the benchmark round, the hardware and the test named, and with the languages that beat PHP on the same test on the same chart.

The Language in 2026 is the shortest possible tour of the syntax, for those who judge a language by reading it. The Ecosystem counts packages, frameworks and tools. Governance and Longevity covers the RFC process, the release calendar, the security process and the money, and Cost of Ownership looks at hiring, hosting and upgrades.

Where PHP Is the Wrong Choice is the chapter that makes the others credible. An Evaluation in One Week is a protocol: what to install, what to measure, what to read, so that the decision you reach rests on your own numbers and not on mine.

The rules I follow

Comparisons are symmetrical. When a chart shows PHP ahead of a language on one measure, the text names a measure where that language is ahead, when one exists. The mainstream options you

would expect to see, Node.js, Python, Java, C#, Go, Ruby, are never left off a chart because they would look good on it.

Names come with evidence. A company appears here only when its own engineers, in a blog post, a talk, a repository or a report, say it runs PHP in production, and the appendix links to that document with its date. Companies that run Hack on HHVM, a language that forked from PHP, are not presented as PHP users, however tempting the logo was.

Frameworks and tools are listed, not recommended. They appear in alphabetical order. The PHP Foundation, under whose umbrella this book is published, promotes the language and its standards rather than a vendor, and so do I.

How to check it

Every chapter ends with something you can verify in an afternoon: a public dashboard to open, a benchmark to rerun on your own hardware, a command to type. Take those seriously. A figure I got wrong, or a figure that aged since I wrote it, is exactly what you should find, and Sources gives you the URL to find it with.

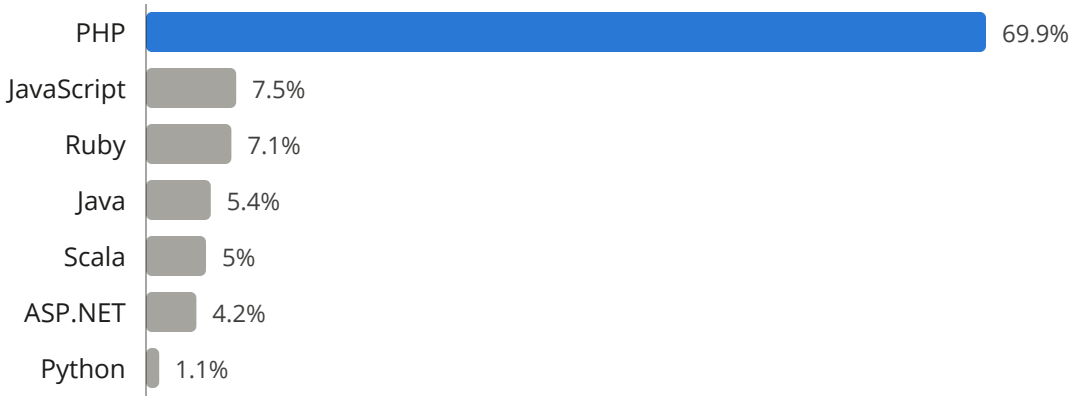
Your first question is probably the one I started with too: who actually runs on this?

Footprint

Your first question is who actually runs on this, at what scale, and whether the answer is a list of logos or a list of documents. **PHP is the server-side language of about seven websites in ten among those whose language can be detected, and that share has been falling for a decade.** Both halves of that sentence matter, and neither means much until you know what the survey counts.

What the web share measures

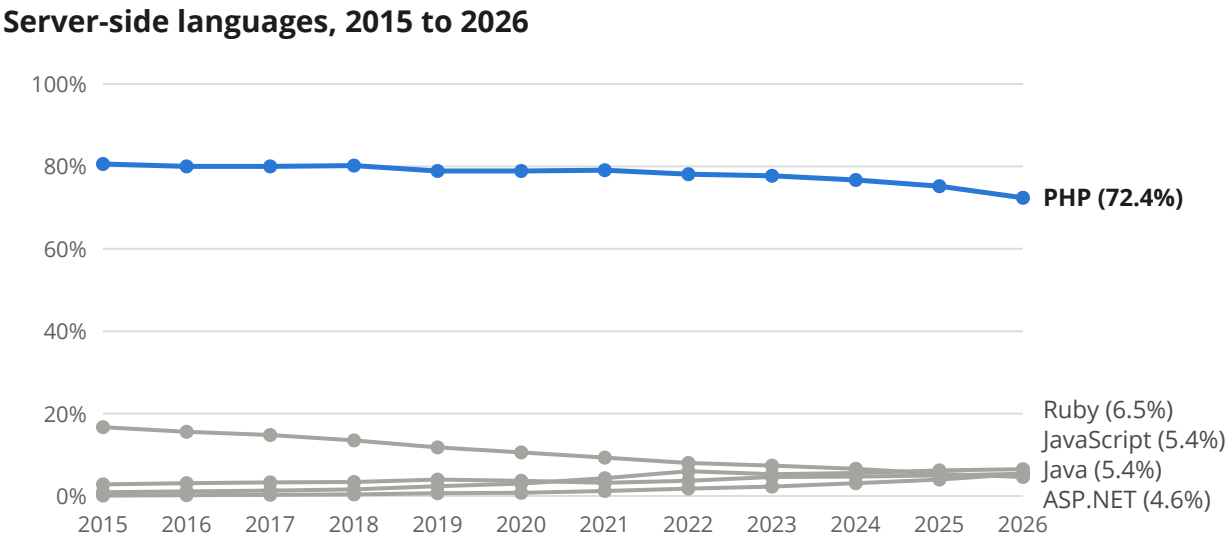
Server-side languages, share of websites



W3Techs, 17 September 2026. Share of websites whose server-side language is detectable.
A site may use several languages. Sample of over 20 million sites; wordpress.com counts as one site.

W3Techs, the survey company behind the figure, inspects a sample of more than twenty million websites every day and detects the server-side language from response headers, cookies, file extensions and similar traces. The percentage is taken over the sites whose language could be detected at all, so a site behind a framework that leaves no trace is not counted. Each site is counted once, and the whole of wordpress.com or wix.com weighs as much as a personal blog. A site may also use several languages, which is why the columns do not add up to one hundred.

On the same date JavaScript on the server stands at 7.5 percent, Ruby at 7.1, Java at 5.4, Scala at 5.0, ASP.NET at 4.2 and Python at 1.1. Within its limits the survey is stable in what it says: the bulk of the addressable web runs on PHP, mostly through WordPress, and the languages people talk about more are small on this measure. Scala ahead of ASP.NET is the reminder that the survey counts what its detection can see, and that a few platforms leave a signature out of proportion to their use.



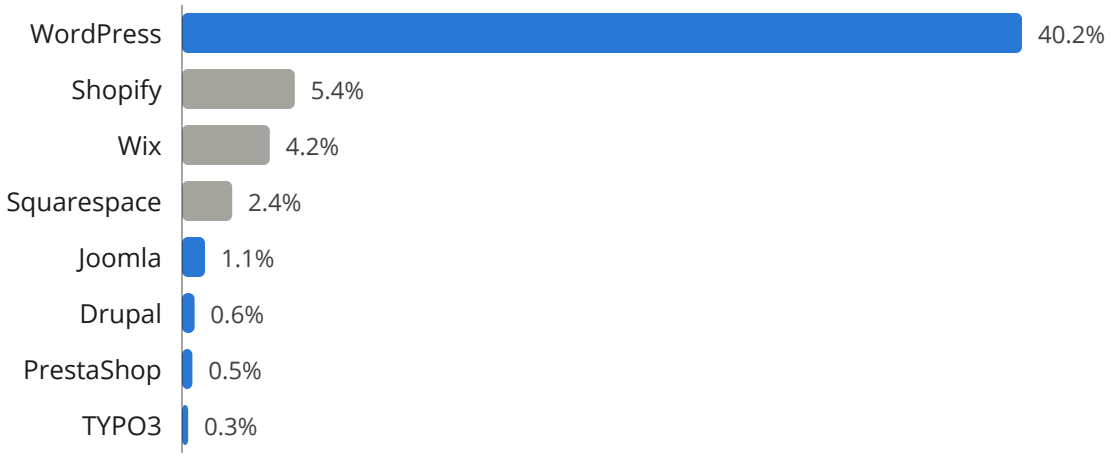
W3Techs historical yearly trends, values on 1 January of each year. Checked 17 September 2026. Share of websites whose server-side language is detectable. On 17 September 2026: PHP 69.9, JavaScript 7.5, Ruby 7.1.

PHP's share on this measure was 80.6 percent on 1 January 2015, 75.2 on 1 January 2025 and 72.4 on 1 January 2026, then 69.9 in September 2026. **The trend is down, and you should see it before anything else.** The decline accelerated in 2025 and 2026, and the share that left PHP went mostly to server-side JavaScript, which W3Techs noted in July 2026 had overtaken Ruby as the second language. Two facts sit side by side here: an installed base on the public web that nothing else matches, and a curve that is not moving in PHP's favour.

The platforms

Most of that installed base is products, not custom code. WordPress alone runs 40.2 percent of all websites, which is 58.8 percent of the sites that use a detectable content management system, and the next PHP-based systems, Joomla, Drupal, PrestaShop and TYPO3, are each under two percent. WooCommerce, the commerce plugin for WordPress, is on 8.0 percent of all websites and represents 47.7 percent of detected e-commerce systems.

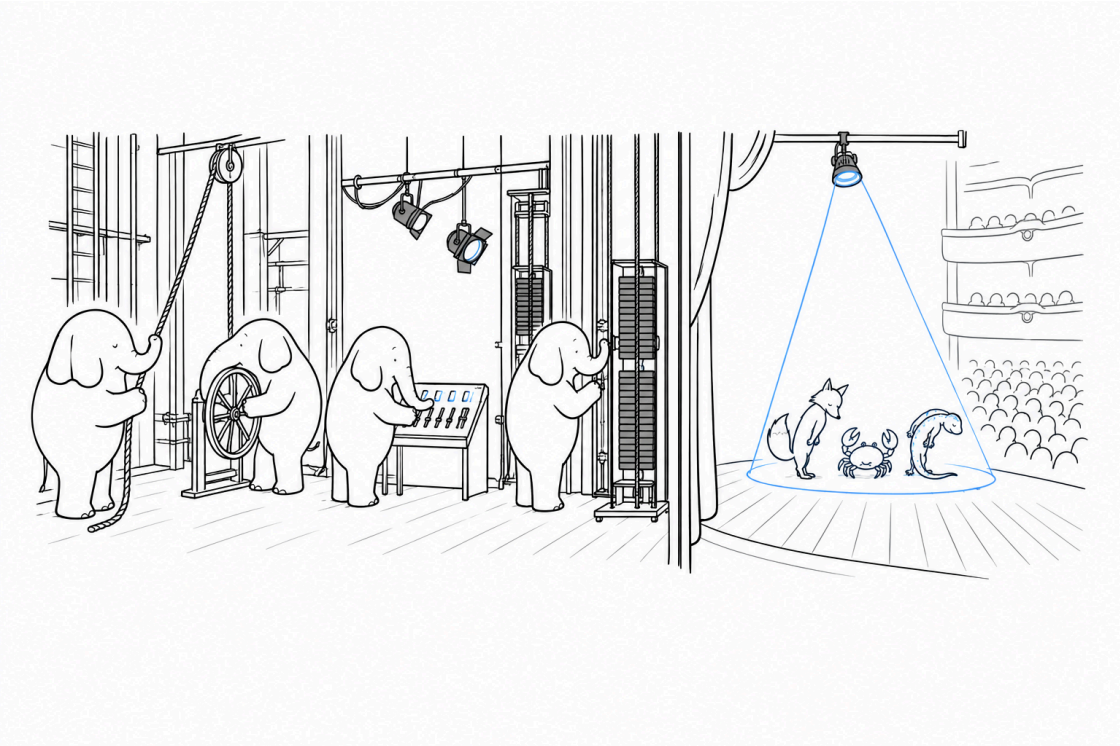
Content management systems, share of all websites



W3Techs, 17 September 2026. Blue: written in PHP. 31.5% of sites use no detected CMS.
Shopify, Wix and Squarespace are hosted platforms, not installable software.

The platforms also publish their own counts, each with its own bias. Moodle, the learning platform, reports 146,634 registered sites and 531 million users, counting only the sites that chose to register. Nextcloud reports more than 500,000 servers. Drupal counts 470,795 sites reporting their version through its update module, an undercount of sites that disabled it. PrestaShop states nearly 250,000 sites and more than 22 billion euros of sales through them in 2024, a self-reported figure. Shopware cites an EHI study placing it on 115 of the 1,000 largest German B2C shops in 2025, and Matomo, the analytics platform, states more than 1.4 million websites. Adobe publishes no merchant count for Adobe Commerce, so I give none.

No single number in that list matters as much as its shape. Content management, commerce, learning, file sharing and analytics are the categories where a product is installed on a server and left to run for years, and they are the categories that PHP software dominates. That is where the share of the previous section comes from.



Organisations that say so themselves

A company appears here only when its own engineers or its own documents say it runs PHP, and the date of that document is part of the evidence.

Wikimedia Foundation. Wikipedia and its sister projects run on MediaWiki, a PHP application served by PHP-FPM. The foundation's audited financial statements state more than 19.4 billion page views per month across its projects. Its production moved from PHP 8.1 to PHP 8.3 on 25 November 2025, and at the time of writing the migration to PHP 8.5 was planned for late 2026. The same infrastructure ran on HHVM until 2019, when the foundation completed its move back to the PHP interpreter. The foundation also reports peaks of 800,000 requests per second across its seven data centres, a figure that counts requests at the edge, most of which are served by the cache layer and never reach PHP, so do not read it as PHP throughput.

Automattic. WordPress.com and Tumblr are stated by Automattic to "run on PHP primarily". WordPress VIP, the company's enterprise hosting arm, publishes 2.4 trillion requests served per year and 22 billion requests on the night of the 2024 United States election, figures that include its CDN.

Etsy. The marketplace's engineering careers page states that engineers "primarily code in PHP and JavaScript", alongside Java, Go and Swift. Etsy moved its production to PHP 7 in 2016 and documented the move at the time with its production graphs.

Mailchimp. The company's developer blog describes its job runner and its application monolith in PHP, and its 2025 and 2026 engineering job postings ask for PHP alongside React and Go. No scale figure is published.

Bumble (Badoo). The engineering blog described, in 2017, more than three million lines of PHP and hundreds of application servers moved to PHP 7, with a stated saving of one million dollars in hardware. That figure is nine years old and you should weigh it as such; the company's current job postings still list PHP next to Go.

Public sector. The European Commission's websites run on Drupal, which their pages declare in their metadata. Commission staff presented the platform at Drupal4Gov EU in January 2026, at 770 live sites, a figure reported by attendees that I could not confirm in a Commission publication. The White House website runs on WordPress, hosted on WordPress VIP under a FedRAMP Moderate authorisation. Germany's federal Government Site Builder, the standard CMS of federal authorities, is built on TYPO3 and serves more than 80 authorities and 250 websites. Australia's GovCMS platform states more than 370 government sites on Drupal, and 77 councils in the United Kingdom share the LocalGov Drupal distribution.

Who left, and who never was

The list of companies that stopped using PHP is as instructive as the list above, and it is shorter than the reputation suggests.

Meta and Slack are the two names most often cited for PHP at scale, and neither runs PHP. Both run Hack, a language Facebook announced in 2014, on HHVM, a virtual machine it had developed for PHP since 2011 and which dropped compatibility with PHP in 2019. Slack removed its last PHP code for the move to HHVM 4 and today maintains about five million lines of Hack. Both companies show that a codebase started in PHP can grow very large, and neither says anything about the PHP interpreter, so I never count them.

Zalando rewrote its Magento shop in Java in 2010, by the account of a former engineer. Trivago replaced a large PHP codebase with a TypeScript application between 2020 and 2021. Dailymotion, which served its site from PHP and Symfony from 2005, states in a 2026 job posting that it is "gradually reducing PHP" and that new development happens in Go, Java and Python. BlaBlaCar, a Symfony shop in 2015, describes in its postings a migration "from a PHP/Symfony stack towards a Java/JS-dominated one". Those are the cases I found with a primary source, and they share a pattern: a company whose product had outgrown a web monolith moved its services to a compiled or JVM language, as companies leaving Ruby or Python do at the same stage.

The developer population

Web share measures servers. Surveys measure people, and they place PHP lower.

Measure	Where PHP stands	Source and date
Used in the past year, professional developers	19.1 percent, 12th language	Stack Overflow survey, 2025
Used in the past year, weighted sample	17 percent, 13th language	JetBrains Developer Ecosystem, 2025
Primary language	9 percent, 9th	JetBrains Developer Ecosystem, 2025
Monthly contributors on GitHub	6th language, unchanged since 2023	GitHub Octoverse, October 2025
Pull requests and Stack Overflow tags	4th, tied with C#	RedMonk, January 2026
Search engine mentions	14th, 1.04 percent	TIOBE, September 2026

Each of these measures something different, and each has a known bias: the Stack Overflow sample is self-selected among its users, the JetBrains sample is weighted toward its customers, GitHub counts open-source activity, and TIOBE counts search results. Read together, they say that between one developer in six and one in five wrote PHP in the past year, that PHP is a first language for fewer people than it is a second, and that its activity on public repositories is steady. JetBrains describes PHP as in “long-term decline” alongside Ruby and Objective-C, while the same report’s PHP-specific survey found that 58 percent of PHP developers do not plan to migrate to another language. What this means for hiring is a question for Cost of Ownership.

> The limit: the footprint is wide and old. Its width comes from products more than from custom applications, its age shows in the share of the installed base still on unsupported versions, and the language’s share of developers is smaller than its share of running servers. If you are choosing a language for a new service, weigh the second fact more than the first.

What to verify yourself

Open the W3Techs pages for server-side languages and for content management systems; they are updated daily and the historical view is public. For Wikimedia, the Phabricator tasks I cite are readable without an account and show the migration work as it happened, with dates. For any company named here, search its engineering blog and its job postings yourself, and treat the absence of PHP in recent postings as the signal it is.

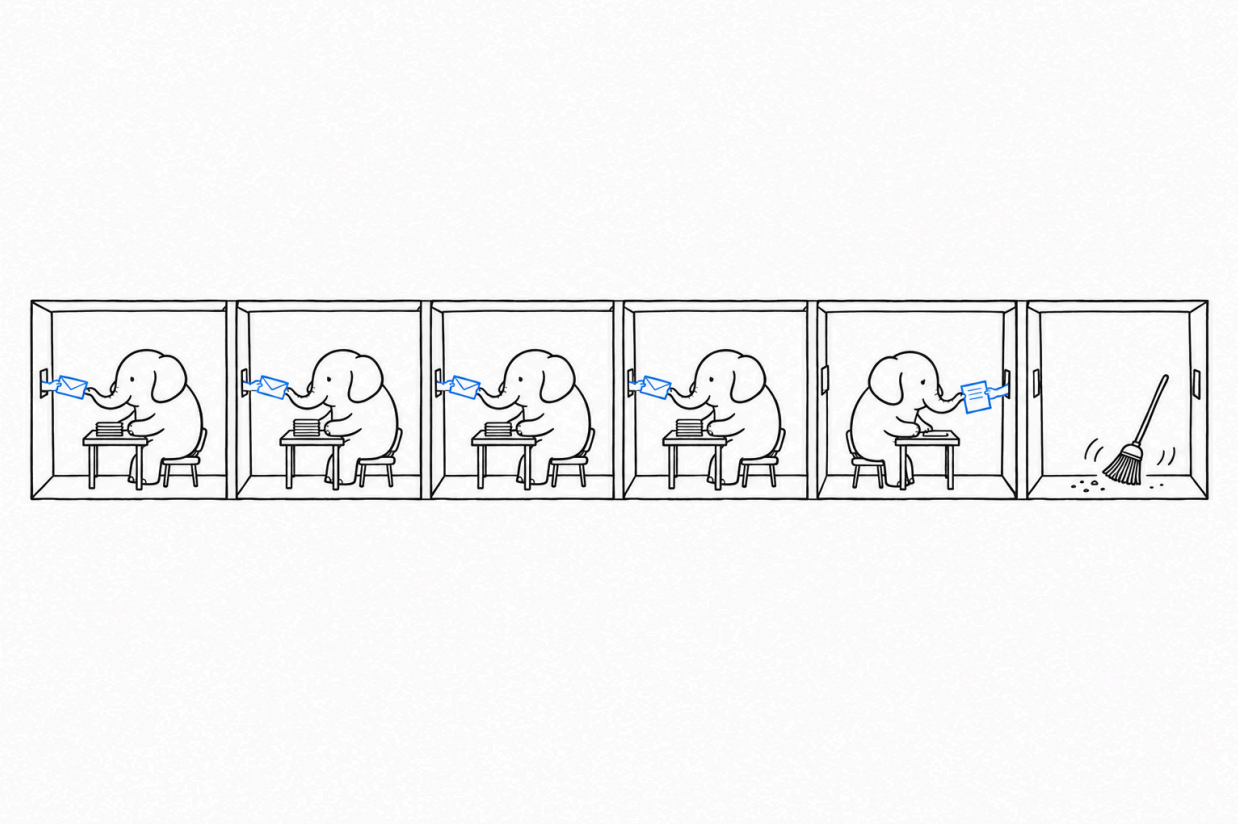
Scale of deployment says nothing about how a request is served. That is where the runtime comes in.

The Runtime

Most of what PHP does well, and most of what it cannot do, follows from the way it serves a request. **A PHP request starts with nothing, runs your code from top to bottom, sends its response, and is discarded; concurrency comes from a pool of processes, each handling one request at a time.** Nothing survives from one request to the next inside the process, which is why the model is called shared-nothing: there is no application object that stays alive, no event loop and no thread.

One request, one process

The standard deployment is PHP-FPM, the FastCGI process manager, behind a web server such as nginx, Apache or Caddy. The manager keeps a pool of worker processes, hands each incoming request to an idle worker, and the worker executes the script, writes the response and returns to the pool with its memory wiped. The pool size is a configuration line. A machine with more cores runs more workers, and more machines behind a load balancer run more pools, with no coordination between them.



A platform team notices the good side of this model first. A request that crashes, leaks memory or runs out of time takes one worker with it and nothing else; the manager replaces the worker and

the other requests never see it. A memory leak cannot accumulate beyond one request, so a long-running PHP application does not degrade over days the way a long-lived process can. Deployment is a file copy followed by a cache reset, because there is no process to restart gracefully and no in-memory state to drain.

The same model has costs, and a performance engineer sees them first. Every request pays for booting the application: reading configuration, wiring dependencies, registering routes. There is no in-process cache across requests, so a lookup table computed once per process in another language is computed once per request here, or stored in shared memory through the APCu extension, or in an external cache. Database connections are opened and closed per request unless configured as persistent, so a connection pool, when needed, lives outside PHP. A workload that must hold state between requests, a WebSocket server or a game lobby, does not fit the model at all, and Concurrency covers what fits it instead.

OPcache, preloading and the JIT

The obvious objection to booting on every request is the cost of parsing and compiling the source every time, and PHP removed it in 2013. **OPcache keeps the compiled form of every file in shared memory, so that a file is compiled once per deployment rather than once per request.** It has been part of the standard distribution since PHP 5.5 and is enabled in the `php.ini-production` template that ships with it; a benchmark run without it measures nothing about PHP.

Preloading, added in PHP 7.4, goes one step further. A list of files is compiled and linked at manager startup and kept in memory, so that classes exist before the first request and no autoloading happens at all. The RFC that introduced it measured a 30 percent gain on one framework's hello-world page and 50 percent on another, and states in the same paragraph that real-world gains "will likely be lower" and depend on the ratio of bootstrap to actual work.

The JIT compiler, added in PHP 8.0 inside OPcache, compiles hot code paths to machine code, and its own release notes are more sober than the headlines. php.net states that the tracing JIT "shows about 3 times better performance on synthetic benchmarks and 1.5 to 2 times improvement on some specific long-running applications", and that "typical application performance is on par with PHP 7.4". The RFC's own measurement on WordPress was 326 requests per second with the JIT against 315 without. The JIT has never been on by default: before PHP 8.4 its buffer size was zero, and since 8.4 a 64-megabyte buffer is reserved but `opcache.jit` is set to `disable`, so it must be enabled explicitly. A web application gains little from it, and a CPU-bound script can gain a lot.

Worker runtimes

The one cost that OPcache does not remove is the application boot, and a second family of runtimes removes it by keeping the application in memory. **In worker mode, a process boots the application once and then handles requests in a loop, one at a time, for thousands of requests**

before being recycled. Alphabetically: FrankenPHP, an application server written in Go on top of the Caddy web server, which offers both the classic one-request-per-process mode and a worker mode; RoadRunner, an application server also written in Go, which keeps a pool of PHP workers alive and passes requests to them over a protocol; and Swoole or its fork OpenSwoole, a C extension that gives PHP its own event loop and HTTP server. FrankenPHP has been hosted under the php organisation on GitHub since 8 June 2025, with its governance unchanged.

What the switch buys depends entirely on how expensive the boot is, and the honest way to show it is on the same code with the runtime as the only variable. On a bare hello-world script, where there is no boot to skip, Tideways, a PHP profiler vendor and a founding member of The PHP Foundation with no stake in either runtime, found FrankenPHP in classic mode and PHP-FPM within half a percent of each other, at about 18,400 requests per second on an eight-core virtual machine. On a full framework, the benchmark suite that Throughput and Latency reads in detail shows the Symfony entry moving from 26,000 requests per second under PHP-FPM to 74,000 under FrankenPHP and 111,000 under Swoole on its Fortunes test, with the framework code unchanged. Vendors and framework authors publish larger ratios on their own pages. I keep to these two figures because you can reproduce both from published material.

The price is the loss of the shared-nothing guarantee. A worker that keeps the application booted also keeps whatever the application leaked, so static properties, caches and open resources now persist between requests, and a class of bugs that PHP-FPM made impossible becomes possible again. The FrankenPHP documentation says it plainly: PHP “was not originally designed for long-running processes”, and the worker mode ships with a maximum-requests counter to recycle processes as a safeguard. A team choosing worker mode takes on the discipline that Node.js or Java teams already have.

What a process costs

You will want a baseline before any framework is added. On the machine I wrote this book on, a PHP 8.3 command-line process with no script allocates 2 megabytes for its own heap and reaches 33 megabytes of resident memory including the interpreter and its loaded extensions, and starts and exits in 20 milliseconds, measured with `memory_get_usage(true)` and `/usr/bin/time`. Reproduce those on your own hardware, and remember that a PHP-FPM worker shares the interpreter’s read-only pages with its siblings, so the marginal cost of one more worker is lower than the figure suggests. Wikimedia’s production configuration, published in its task tracker, runs eight PHP-FPM workers per MediaWiki container with a 500 megabyte OPcache and an APCu cache of 768 megabytes, in one to two gigabytes of memory per container.

> The limit: the shared-nothing model has no threads, no in-process shared state and no background work inside a request. Anything that must outlive a request, a connection pool, a warm cache, a scheduled job, lives in another process or another system, and the architecture around a PHP application reflects it. That is a real constraint, and it is also why the operational story is simple.

What to verify yourself

Install PHP-FPM and a web server on a small virtual machine, enable the FPM status page, and watch the pool under a load generator such as `wrk` or `ab`: the process count, the memory per worker and the request time are all visible. Then run the same hello-world under FrankenPHP in worker mode and compare. An afternoon of this teaches you more about the runtime than I just did, and the numbers will be yours.

A throughput figure means something only once you know which of these runtimes produced it.

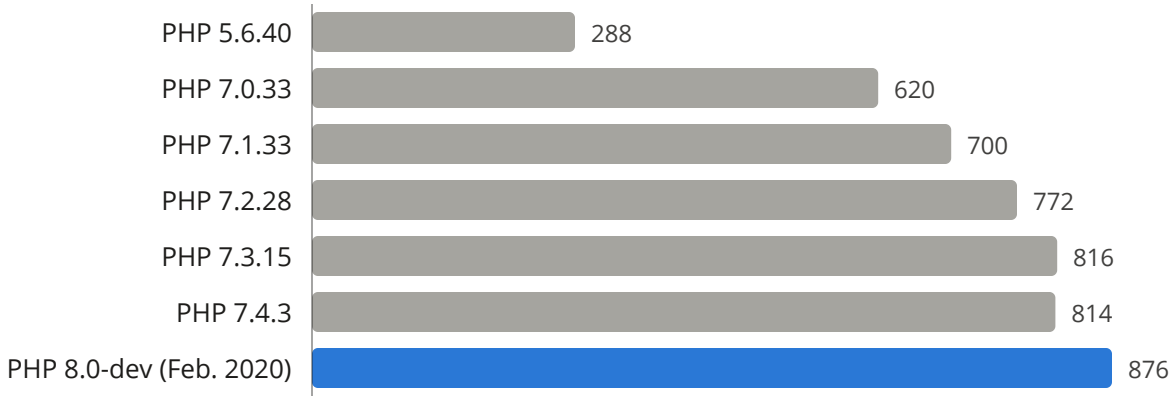
Throughput and Latency

How fast is it, on which measure, and against whom: your second question, and the one where hype does the most damage. **On the largest public benchmark that compares web frameworks across languages, PHP entries span two orders of magnitude depending on the runtime they run on, from the top forty overall to the bottom tenth, so the language alone predicts nothing and the deployment predicts almost everything.** A figure that wide only helps you when it comes with its round, its hardware and its test, and every figure here carries all three.

The engine's history

The interpreter itself, on CPU-bound code, improved by a large step once and by small steps since. The step was PHP 7.0, in December 2015, which replaced the engine's internal data structures. The vendor of the engine claimed at the time that execution time was "often halved" compared with PHP 5.6, in a white paper without a published methodology. An independent run on one machine, with every version of the interpreter from 5.6 to a PHP 8.0 development build, agrees on the shape: the PHPBench score, a synthetic suite of interpreter micro-benchmarks, went from 288,000 on PHP 5.6 to 620,000 on PHP 7.0, then rose in smaller steps, between nothing and thirteen percent per version, to 876,000 on the 8.0 build.

PHPBench score by PHP version, same machine (higher is better)



OpenBenchmarking.org, result 2002269-VE-PHPBENCHM24 by Michael Larabel, February 2020, Core i9-9900KS, Ubuntu 20.04, PHPBench 0.8.1, mean of three runs.
PHPBench is a synthetic CPU micro-benchmark of the interpreter, not a web application. PHP 8.0 was a development snapshot of February 2020, before the tracing JIT landed.

Since PHP 8.0 the interpreter's speed on web applications has been flat, and the sources that measure it say so themselves. A hosting company that publishes yearly benchmarks measured WordPress at 146 requests per second on PHP 8.2 and 148 on PHP 8.5 on the same machine, and wrote that "incremental releases rarely produce large speed jumps on their own". The measurement is the hosting vendor's own, run at fifteen concurrent requests on a thirty-core

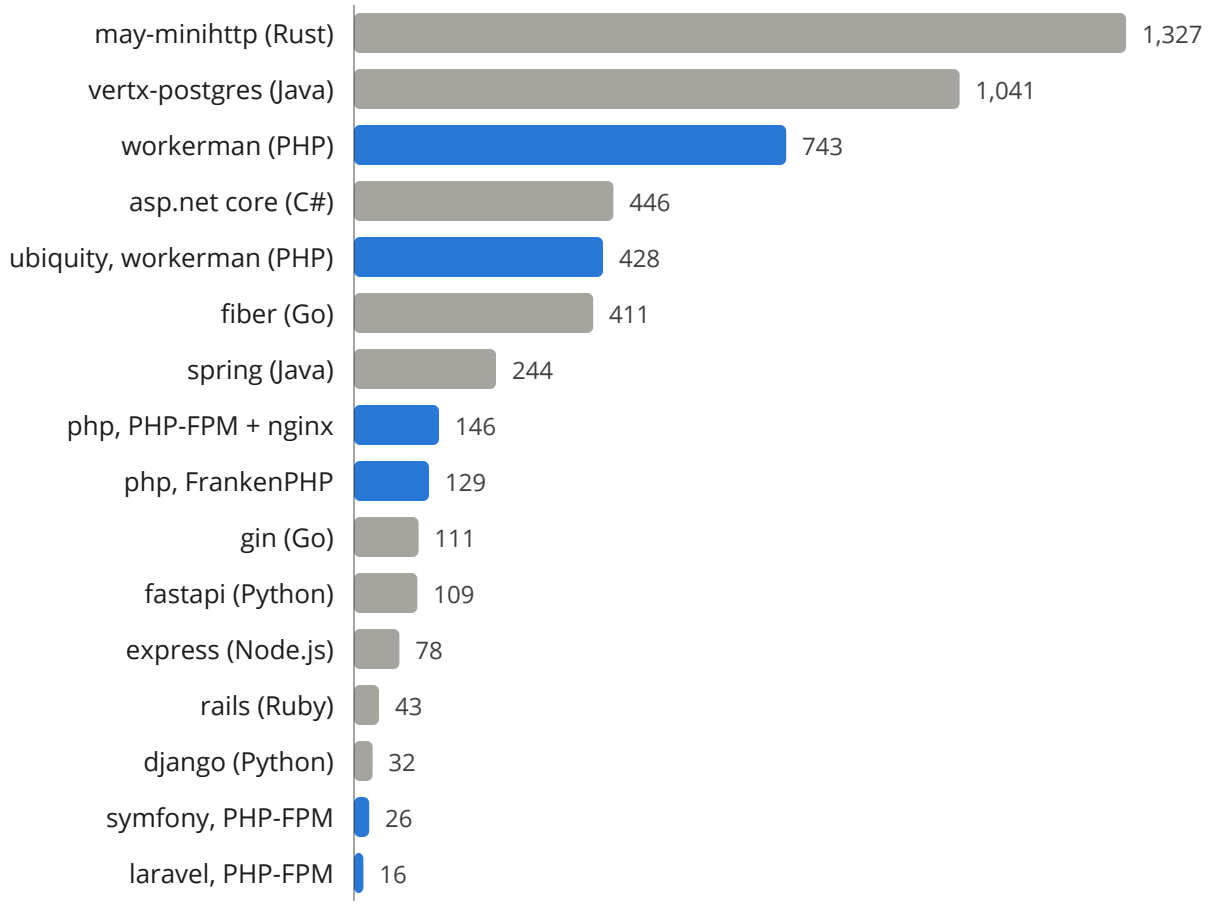
machine, so it measures response time rather than capacity. Expect a current PHP version to be roughly twice as fast as PHP 5 on the interpreter's own work, and expect nothing from a minor version upgrade beyond bug fixes and features.

The cross-language benchmark

TechEmpower's Framework Benchmarks is a public suite that ran hundreds of web frameworks, in dozens of languages, through the same six tests on the same hardware, with every implementation contributed and maintained by volunteers in a public repository. Its last completed round is Round 23, dated 24 February 2025, run on a server with a Xeon Gold 6330 processor, 28 cores and 56 threads, and a 40 gigabit network. The project's repository was archived on 24 March 2026, so Round 23 is the final round and its figures are the last of their kind. Nothing else compares this many frameworks under one protocol, which is why I keep them, and the tests measure different things, which is why each figure names its test.

The Fortunes test is the closest to a web page: one database query returning a dozen rows, one HTML template rendered with escaping. On that test, 510 entries completed Round 23. The chart picks sixteen of them, the overall leaders, the mainstream framework of each major language and the PHP entries at each kind of runtime, and the raw table with every entry is stored beside the book's chart data, so you can make your own selection.

TechEmpower Round 23, Fortunes test, thousands of requests per second

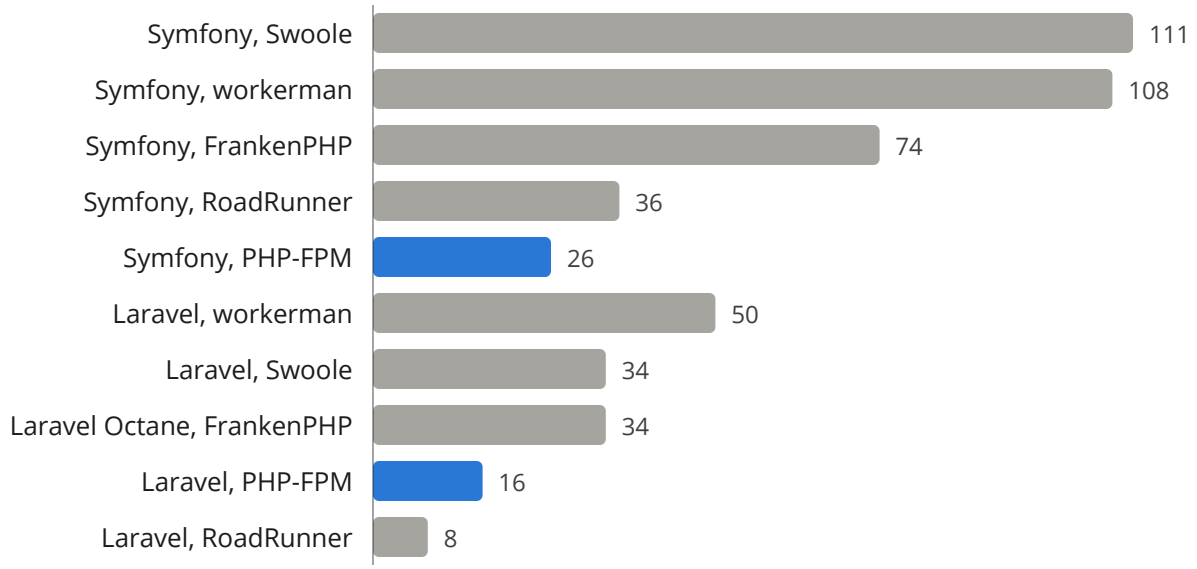


TechEmpower Framework Benchmarks, Round 23 (24 February 2025), Fortunes test, Citrine environment: Xeon Gold 6330 (28 cores, 56 threads), 40 GbE. Selection of 16 entries out of 510.
Fortunes: one database query, HTML rendering with escaping. Blue: PHP entries. Each bar names the framework and its runtime; the language alone decides nothing here.

Read the PHP bars from the top. The fastest PHP entries, on the workerman and Swoole runtimes with raw SQL and no framework, sit at rank 31 to 38 of 510, above 730,000 requests per second, ahead of the plain ASP.NET Core entry, the Go entries and Spring. A full-stack PHP framework with a full ORM on a worker runtime, the Ubiquity entry, reaches 428,000 at rank 92. Plain PHP under PHP-FPM and nginx, the standard deployment, reaches 146,000 at rank 248, about a third ahead of Gin and FastAPI. The two most used PHP frameworks under PHP-FPM sit near the bottom, Symfony at 26,000 and Laravel at 16,000, below Rails at 43,000 and Django at 32,000. One benchmark supports both the claim that PHP can be among the fastest web runtimes and the claim that a typical PHP application is among the slowest, and you should expect to hear both together.

Part of the spread is the runtime. An entry that keeps the application booted between requests removes the boot cost that dominates a framework's request time.

Same framework, different runtime (Fortunes, thousands of requests per second)



TechEmpower Framework Benchmarks, Round 23 (24 February 2025), Fortunes test. Blue: the PHP-FPM baseline.
Entries are maintained by volunteers; a slow entry may reflect its configuration rather than the runtime.

The same Symfony code moves from 26,000 requests per second under PHP-FPM to 74,000 under FrankenPHP and 111,000 under Swoole, and the same Laravel code from 16,000 to 50,000 under workerman. The rest of the spread is the entry itself. TechEmpower entries are maintained by whoever cares to, and a slow entry can reflect a dated configuration as much as the framework. The Laravel entry under RoadRunner, at 8,000, is slower than under PHP-FPM; I have no explanation for it and report it as measured.

The other tests move the ranks but not the story. On the JSON serialisation test, which has no database, the Swoole entry reaches 2.5 million requests per second at rank 73, plain PHP-FPM 428,000 and Laravel under PHP-FPM 27,000, while ASP.NET Core stands at 1.4 million, Node.js at 1.1 million, Spring at 328,000 and Django at 167,000. On the twenty-query test, which is bound by the database driver, the leaders of every language converge under 90,000 and the fastest PHP entries sit at 56,000, ahead of FastAPI at 37,000 and Spring at 32,000.

Latency

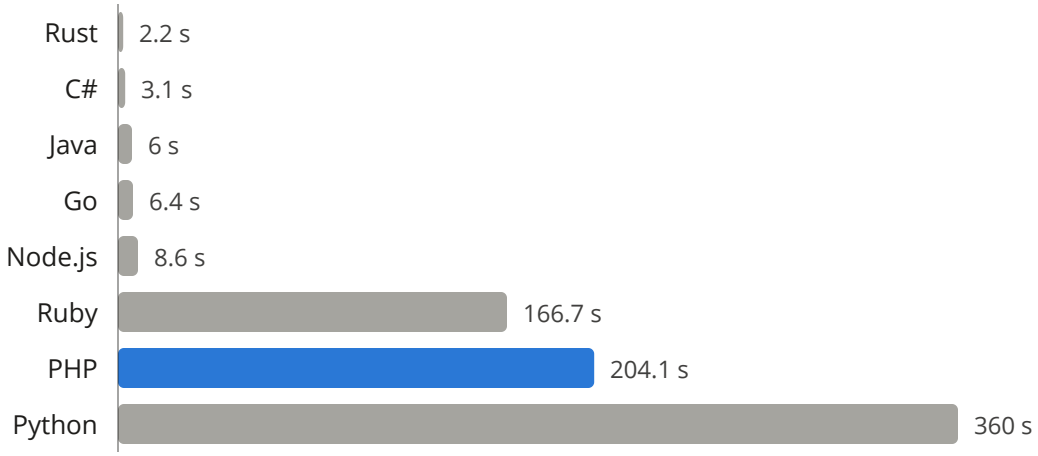
Throughput says how many requests a machine can absorb. Latency says how long one user waits, and it is dominated by what the request waits for, not by the interpreter. Tideways, a profiler vendor and a founding member of The PHP Foundation, measured a PHP-FPM hello-world behind nginx on an eight-core virtual machine and reported a 99th percentile of 0.9 milliseconds at 18,000 requests per second. That is the floor: the interpreter and the process manager together cost under a millisecond when there is nothing else to do, and everything above it is the application and its dependencies.

The most useful public latency figures come from an organisation that publishes its own targets and its own dashboards. Wikimedia's engineering guidelines require a GET request to complete in 50 milliseconds at the median and 200 milliseconds at the 99th percentile of the time spent in PHP, and 500 milliseconds at the 99th percentile for a POST. Its Grafana instance is public and shows the measured distribution for the requests that reach the application servers, that is, the cache misses and the logged-in users, which are the expensive ones. On the day this chapter was checked the median was about 170 milliseconds and the tail well above the target, for pages that include rendering wikitext from a cold parser cache. I give you the dashboard rather than the snapshot, because a week of it tells you more than an hour.

Where PHP loses

On CPU-bound code with no I/O, PHP is an interpreter with an optional JIT, and it is slower than a JIT-compiled runtime such as V8 by an order of magnitude. The Computer Language Benchmarks Game, which compares contributed programs on the same small tasks, puts the fastest single-threaded n-body program at 2.2 seconds in Rust, 3.1 in C#, 6.0 in Java, 6.4 in Go, 8.6 in Node.js, 167 in Ruby with YJIT, 204 in PHP 8.4 and 360 in Python 3.13.

CPU-bound work: n-body, elapsed seconds (lower is better)



The Computer Language Benchmarks Game, n-body, fastest entry per language, measured on an Intel i5-3330; PHP 8.4.1, Python 3.13, Ruby 3.4 with YJIT, Node.js 23.8. Checked 17 September 2026. Contributed programs, one core each; the C# and Java entries are ahead-of-time compiled builds. The PHP command line sets a JIT buffer but not opcache.jit, which PHP 8.4 disables by default.

The chart carries its caveats. The PHP programs are run with a JIT buffer configured but with `opcache.jit` left at its PHP 8.4 default, which is disabled, so they measure the interpreter alone; the JIT RFC reports a four-fold gain on Mandelbrot with the JIT on, which would narrow the gap without closing it. The programs are also contributed, so they measure the best program someone bothered to write. With both caveats, the order stands. For numerical work, simulation, parsing large inputs in a loop or anything that keeps a core busy, PHP is in the class of Python and Ruby, not in the class of Node.js, and thirty to a hundred times behind Go, Java, C# and Rust on this program. Where PHP Is the Wrong Choice draws the consequence.

> The limit: PHP's good throughput figures come from worker runtimes and raw database access, which a typical team does not run on day one. Its typical figures, a full-stack framework under PHP-FPM, sit with Rails and Django, at a few tens of thousands of requests per second on a large machine, which is more than most applications will ever receive. And on CPU-bound work the interpreter is an order of magnitude behind V8.

What to verify yourself

The TechEmpower toolset still runs from its archived repository with Docker, and the raw Round 23 tables for six tests are stored in the book's `charts/raw/` folder as CSV, so you can check my selection above and make any other. Closer to home, take the application you would build, or an equivalent open-source one, run it under PHP-FPM and under one worker runtime on the same machine, and measure with `wrk` at the concurrency you expect in production; the ratio you obtain is the only one that matters. For latency, open Wikimedia's public "Backend Pageview Timing" dashboard and read a week of it.

Throughput assumes the requests are independent. The next question is what happens when they are not.

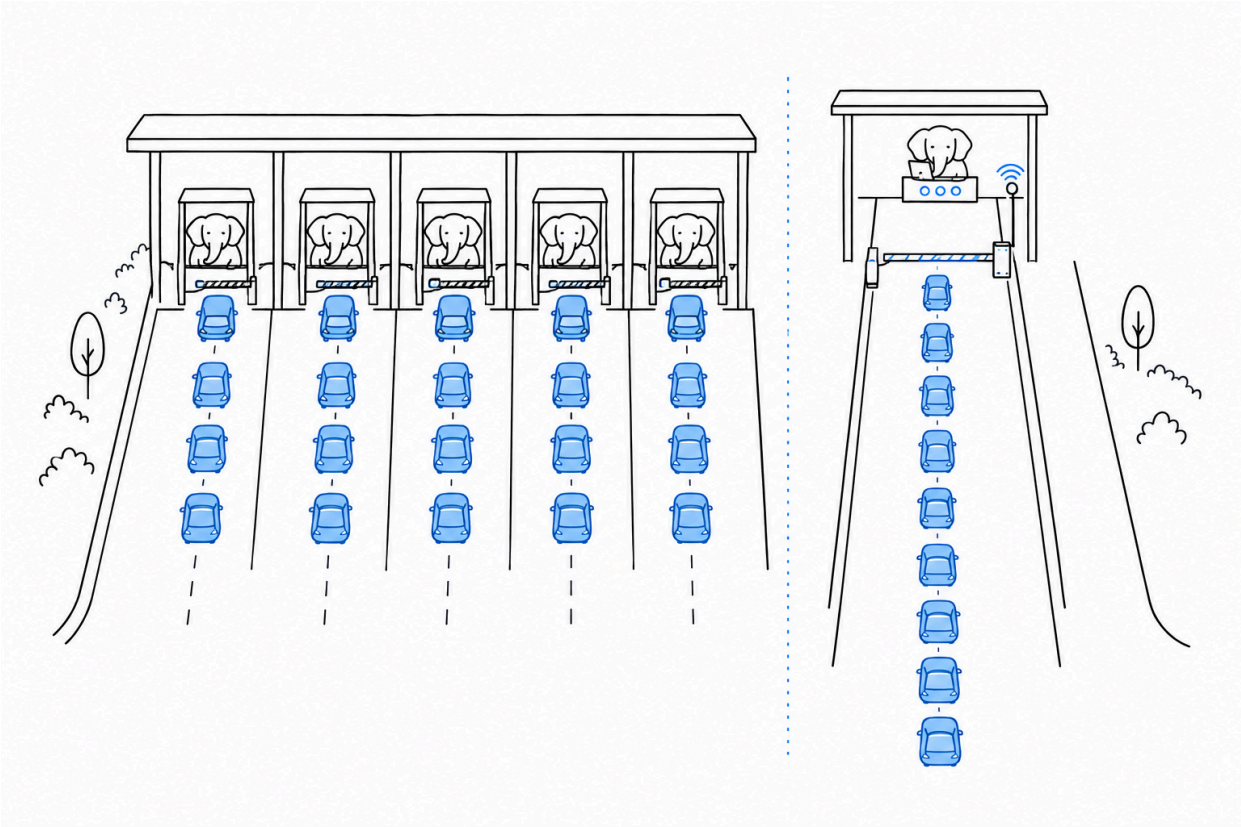
Concurrency

Ten thousand idle connections, a hundred outbound calls to fire at once, a computation that should keep every core busy: at some point your requests stop being independent, and that is where your third question begins. **PHP has no threads in its standard build and no built-in event loop; concurrency comes from processes, from libraries built on the coroutine primitive the language added in 8.1, and from runtimes that bring their own event loop.** Each of those three fits some workloads and fails others, and the difference is what you need to know before you commit a service to it.

Many requests

A worker waiting 40 milliseconds on a query holds one process for 40 milliseconds, and no other request notices. That is the whole model for the ordinary case of many independent HTTP requests: the process pool of The Runtime, where each worker blocks on its database call, its cache lookup and its outbound HTTP request while the operating system schedules the other workers around it. An event-loop runtime such as Node.js needs asynchronous code so that one slow call does not freeze every connection. PHP gave each connection its own process instead, and the code stays synchronous.

The bill for that simplicity is memory per idle connection. A worker holding a WebSocket open, waiting on a long poll or streaming a response for a minute occupies a whole process for the duration, and a pool of a few hundred processes cannot hold ten thousand idle connections. For that workload PHP's default model is the wrong one.



Many connections

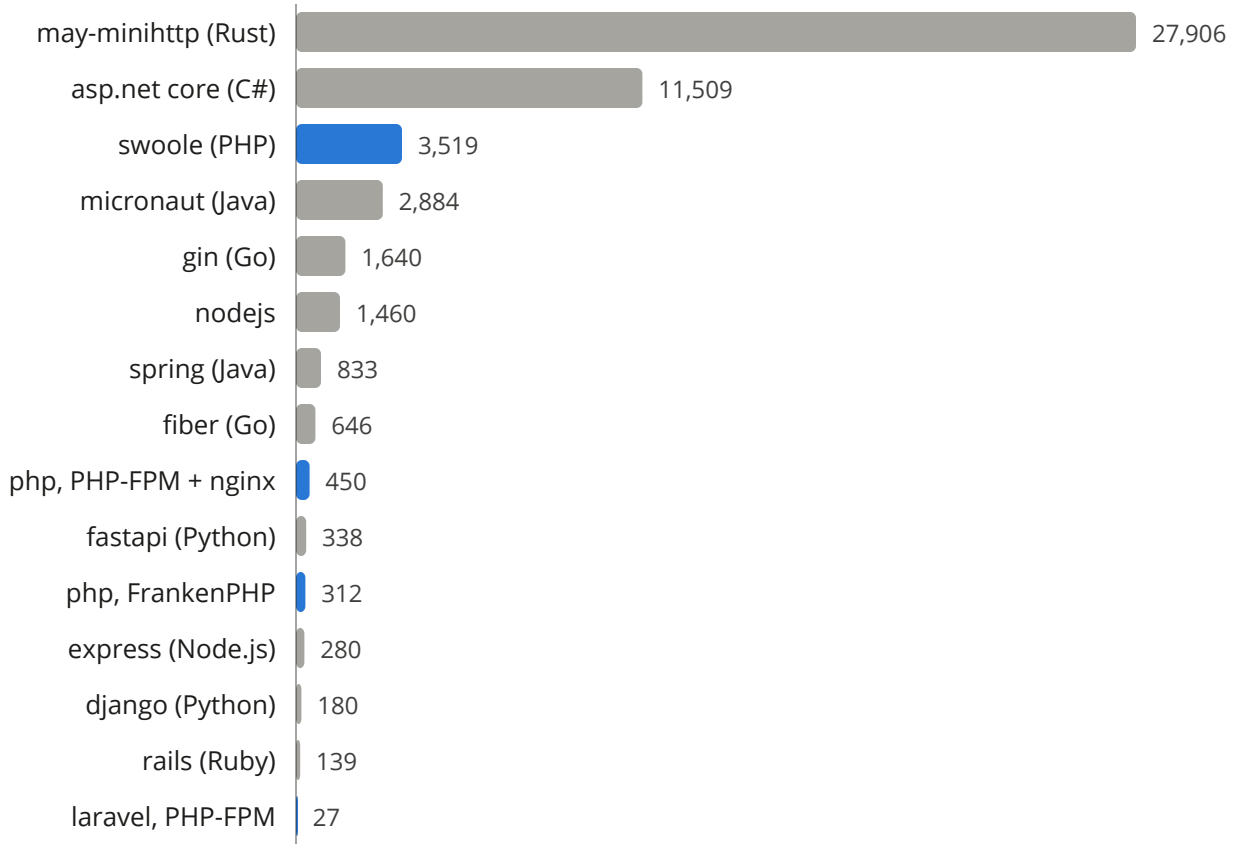
For workloads that need many concurrent connections in one process, PHP has async runtimes, and they are libraries or extensions rather than language features. Alphabetically, they are AMPHP, a set of non-blocking libraries built on an event loop and on the language's fibers; ReactPHP, a low-level event-driven library set with its own event loop, HTTP server and stream components; and Swoole or its fork OpenSwoole, a C extension that gives PHP an event loop, coroutines and a built-in HTTP and WebSocket server, with I/O calls that yield instead of blocking. FrankenPHP and RoadRunner, the worker runtimes of The Runtime, are not on that list because they answer a different need: they keep the application booted between requests, still one request at a time per worker, and do not make code concurrent.

Under the first two sits the Fiber, added in PHP 8.1: a function that can suspend itself and be resumed later by whoever holds it. The RFC that introduced fibers is precise about what they are not: "all fibers exist within a single thread, only a single fiber may execute at a time", and "blocking code, such as `file_get_contents()`, will continue to block the entire process, even if other fibers exist". The language ships no scheduler; a library provides one, and every I/O call in the program has to go through that library to yield. An async PHP application is therefore written against AMPHP's or ReactPHP's HTTP client, database client and filesystem functions rather than the standard ones. One synchronous library pulled in by mistake blocks the whole loop.

How far an event loop carries PHP shows in the cross-language benchmark's plaintext test, which does no application work at all and measures only how many pipelined connections an HTTP layer

can serve.

TechEmpower Round 23, plaintext test, thousands of requests per second



TechEmpower Framework Benchmarks, Round 23 (24 February 2025), plaintext test with HTTP pipelining, up to 16,384 concurrent connections. Selection of 15 entries out of 514.
Plaintext measures the HTTP layer and connection handling, with no application work. Blue: PHP entries.

The Swoole entry serves 3.5 million requests per second, ahead of Micronaut, Gin, Node.js and Spring, while plain PHP under PHP-FPM serves 450,000 and Laravel under PHP-FPM 27,000. The ceiling is elsewhere: ASP.NET Core at 11.5 million, and at the top a C-backed Python server and a Rust server at about 28 million. With an event loop PHP sits in the second group, not the first; without one it sits in the fourth.

Many cores

The fastest PHP spectral-norm program in the Benchmarks Game finishes in 18 seconds of wall time while using 72 seconds of CPU across cores, against 90 seconds for the fastest Python program and 1.6 seconds for Node.js. That gap between CPU time and wall time is what parallel computation looks like in PHP, and once again it comes from processes. The command-line interpreter forks with `pcntl_fork()`, starts subprocesses with `proc_open()` or runs several scripts under a supervisor, and the results come back through pipes, files, a database or a queue. The `parallel` extension offers threads on the thread-safe build of the interpreter, which few

distributions ship. Fork-based parallelism works, and it works inside the interpreter's speed class described in Throughput and Latency.

Whether the scheduler will ever move into the engine is an open question on the internals list. An RFC titled "Concurrency Support in the PHP Engine", which would add a scheduler and structured concurrency to the language itself, was under discussion there at the time of writing, with no vote held, after an earlier proposal on the same subject was cancelled in July 2026. As I write, in September 2026, the language has fibers and the scheduler is a library.

Work that outlives the request

Most of the concurrency you will actually need is the kind the user should not wait for. A request has a time limit and a response to send, so anything that takes longer has to leave the request. **The idiom is a queue and a worker process.** The request writes a job to a table or a message broker and returns; a command-line PHP script, kept alive by a process supervisor, loops over the queue and runs the jobs one after another. That script has no time limit and no memory limit unless you configure them, so a production worker sets both and exits cleanly after a few thousand jobs, letting the supervisor start a fresh process. Most full-stack frameworks ship the pattern with a driver for the usual brokers, and cron covers the scheduled case.

> The limit: PHP's concurrency is coarse. Processes for requests, processes for cores, a queue for background work, and a separate runtime with its own I/O libraries for the connection-heavy case. If your product is a real-time service with tens of thousands of open connections, you will spend your first month choosing and learning one of the async runtimes, and you will find that most of the ecosystem's libraries were not written for it. If your product is request and response, you will not meet the question.

What to verify yourself

Take the connection-heavy scenario you actually have, if you have one, and write it twice in an afternoon: once with AMPHP or ReactPHP, once with Swoole or OpenSwoole. Then check how many of the libraries you would depend on offer a non-blocking client. If the answer is most of them, the async path is open to you; if it is few, the honest conclusion is that PHP is the wrong tool for that one service, and the rest of your system is a separate decision.

Whether the language itself is one you would want to write is a different question, and you answer it by reading it.

The Language in 2026

Is this a language you would want to write for a few years? No benchmark answers that; you answer it by reading code. **PHP today is an object-oriented, garbage-collected language with declared types enforced at runtime at function and property boundaries, with enums, immutability, closures, attributes and a package manager.** One example is enough to judge it, and every construct younger than 2022 carries the version that introduced it, so you know what a given project can use.

One complete example

```php
<?php
declare(strict_types=1);

enum Status: string
{
    case Draft = 'draft';
    case Published = 'published';
    case Archived = 'archived';

    public function isVisible(): bool
    {
        return $this === self::Published;
    }
}

final readonly class Article
{
    public function __construct(
        public string $title,
        public Status $status,
        public ?\DateTimeImmutable $publishedAt = null,
    ) {
    }

    public function publish(\DateTimeImmutable $at): static
    {
        return new static($this->title, Status::Published, $at);
    }
}

function summary(Article ...$articles): string
{
    $visible = array_filter($articles, fn (Article $a): bool => $a-
>status->isVisible());
    $titles = array_map(fn (Article $a): string => $a->title,
$visible);

    return match (count($titles)) {
```

```
        0 => 'nothing published',
        1 => $titles[array_key_first($titles)],
        default => implode(', ', $titles),
    };
}

$draft = new Article(title: 'Benchmarks', status: Status::Draft);
$live = $draft->publish(new \DateTimeImmutable('2026-09-01'));

echo summary($draft, $live), PHP_EOL;   // Benchmarks
echo $draft->status->value, PHP_EOL;    // draft
```

Read it the way you would read a pull request. `declare(strict_types=1)` switches the file to strict typing, so a string passed where an `int` is declared throws a `TypeError` instead of being converted; the switch works per file, and a project turns it on in every one. The `enum` is a real enumeration, a closed set of singleton objects that can carry methods and a backing value. The `readonly` class (PHP 8.2) makes every property immutable after construction, and its constructor declares and assigns those properties in one place.

The rest of the syntax you already know from elsewhere: named arguments at the call site, `?` for nullable types, `static` as a return type, `fn` for one-line closures, and a `match` that compares strictly, never falls through and throws when no arm matches.

What the example does not show matters as much. Every boundary in it is typed, though the language does not require that; a read of an undefined variable is a warning in PHP 8, and the static analysers turn it into an error. The `$` sigil on variables and the `->` arrow for member access are the two pieces of syntax that look foreign to everyone else, and they stop being visible after an hour.

The recent additions

The last two releases changed how code is written, and a project on PHP 8.4 or later will use what they added.

```php
// PHP 8.4
final class Money
{
    public function __construct(
        public private(set) int $cents,
        public string $currency,
    ) {
    }

    public string $formatted {
        get => number_format($this->cents / 100, 2) . ' ' . $this-
>currency;
    }
}

$price = new Money(1999, 'EUR');
echo $price->formatted, PHP_EOL;     // 19.99 EUR
$price->cents = 5;                   // Error: Cannot modify
private(set) property Money::$cents
```

`public private(set)` is asymmetric visibility (PHP 8.4): the property is read from anywhere and written only from inside the class, which removes most of the getters a Java or C# developer expects to write. The `get` block under `$formatted` is a property hook (PHP 8.4), logic attached to a property without changing its call sites, and that removes most of the rest. The pipe operator (PHP 8.5) chains functions left to right:

```php
// PHP 8.5
$slug = '  Hello, World  '
    |> trim(...)
    |> strtolower(...)
    |> (fn (string $s): string => preg_replace('/[^a-z0-9]+/', '-',
$s))
    |> (fn (string $s): string => trim($s, '-'));

echo $slug, PHP_EOL;   // hello-world
```

The `trim(...)` form is a first-class callable (PHP 8.1), a reference to a function as a value, with the arity checked by the engine. Add attributes, structured metadata read by reflection and used by

every framework for routing, validation and mapping, and you have the constructs you will meet most in a codebase started after 2024.

What the type system does and does not do

Types are declared on parameters, return values, properties and class constants, and the engine enforces them at runtime. Union types (`int|string`), intersection types (`Countable&Traversable`), nullable types, `never`, `mixed` and enums are all part of the language. A type error is an exception, not a warning, and in strict mode there is no implicit coercion between scalars, except the widening of an `int` into a `float`.

You should hear from me now what is missing, rather than discover it in week three. **There are no generics in the language.** A `list<Order>` cannot be expressed in a signature; the parameter is `array`. The ecosystem answers with a docblock syntax that both static analysers, PHPStan and Psalm, understand and enforce. `@param list<Order> $orders` is checked at analysis time, in the editor and in continuous integration, along with template types, conditional types and shape types richer than anything the language itself can express, so a typed PHP project runs an analyser at its strictest level on every build and treats its output as compiler errors. Whether that is an acceptable substitute for language-level generics is your call, against your own habits. It is a substitute, and you should weigh it as one.

The other absence comes from the runtime rather than the language: no threads in userland and no built-in event loop. Concurrency describes what exists instead.

The standard library

`strpos` sits next to `str_replace`, `array_key_exists` next to `in_array`, and some functions take the needle first while others take the haystack. That inconsistency is historical, it is real, and it is not going away. Since PHP 8.0 new functions follow one naming scheme (`str_contains`, `array_is_list`, `array_find`), internal functions throw `TypeError` and `ValueError` on bad input instead of returning `false`, and every editor with a PHP language server completes the names, which is how a team lives with the rest. The second wart is that strings are byte sequences: `strlen('é')` is 2, and text handling goes through the `mb_` family. Those are the two you will meet first.

> The limit: PHP's type system is enforced at runtime and at the boundaries of functions and properties, not inside expressions and not across generic containers. A team that wants compile-time guarantees over collections gets them from a static analyser, not from the language.

What to verify yourself

Install PHP 8.5 (your package manager, or the official Docker image `php:8.5-cli`), paste the first example into a file and run it with `php file.php`. Then break it: pass a string where the `Status` enum is expected, remove `strict_types`, write a `match` over the enum and leave a case out, then call it with that case. The error messages are the part of a language you live with, and five minutes are enough to judge them. For the type system, run PHPStan or Psalm at its strictest level on any open-source PHP project of a size you care about, and read the first twenty findings; they show you what the analyser catches that the engine does not.

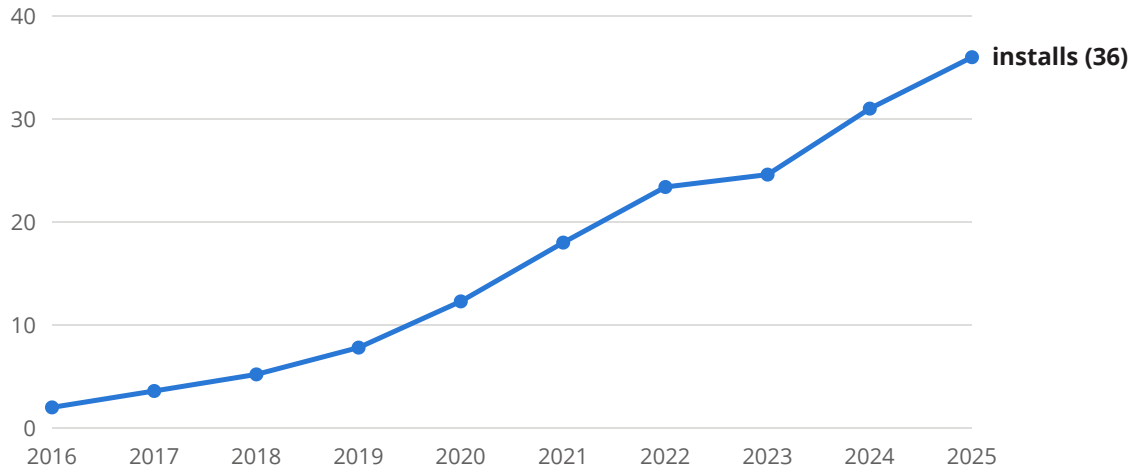
The Ecosystem

Nobody adopts a language on its own; you adopt its libraries, its tools and its conventions with it, and your question is whether PHP's hold up against what npm, pip or Maven give you elsewhere. **PHP has one package manager, one public registry, an interoperability body that publishes standards rather than products, and at least two mature options in every tooling category, none of which the language itself endorses.** Size and activity can be measured, and the figures are here; quality is something I can only point you at.

Composer and Packagist

Composer is the dependency manager, and there is no second one. It reads a `composer.json`, resolves versions against semantic versioning constraints, writes a `composer.lock` that pins every transitive dependency, and generates the autoloader that maps class names to files under the PSR-4 standard. Packagist, its default registry, listed 468,099 packages and 5.8 million package versions on 17 September 2026, and had recorded 199.6 billion package installs since April 2012.

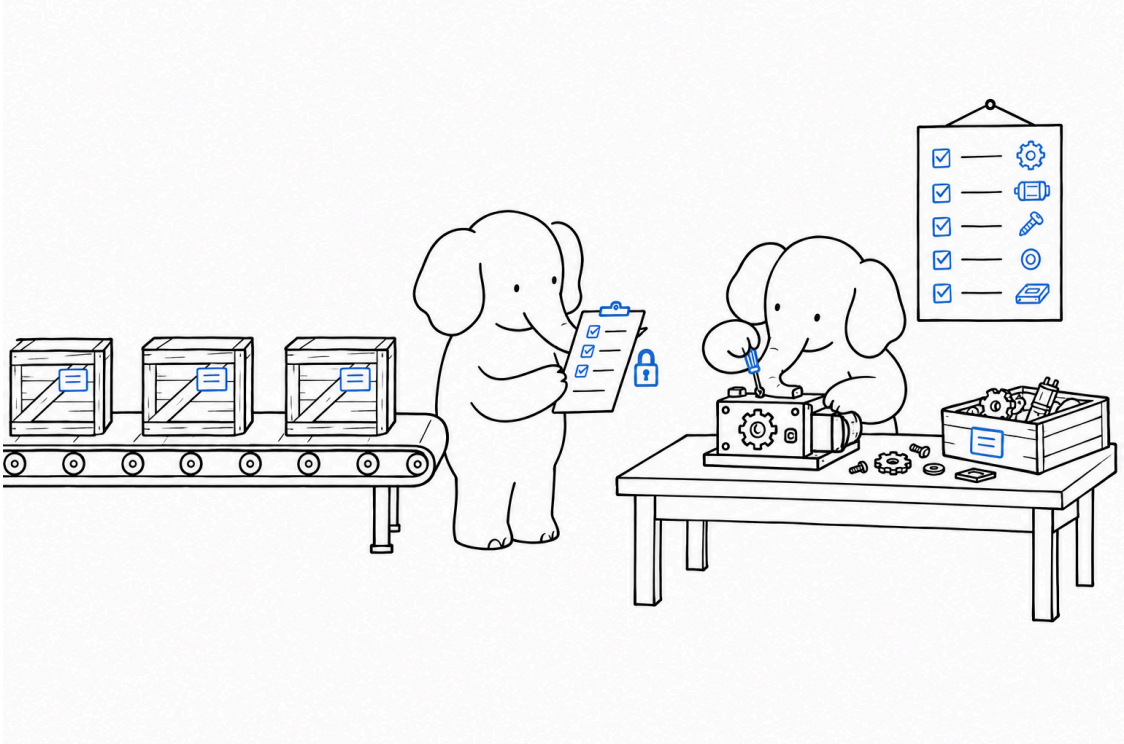
Package installs through Composer, billions per year



packagist.org/statistics, monthly install counts summed per calendar year. Checked 17 September 2026.
An install is one package installed by Composer and reported to packagist.org; continuous integration and container builds weigh heavily in it.

Installs through Composer grew from 2.0 billion in 2016 to 36.0 billion in 2025, and August 2026 alone recorded 4.9 billion, 91 percent more than August 2025. That is the activity signal, and its definition carries a caveat: an install is one package installed by Composer and reported to the registry, so continuous integration pipelines and container image builds account for a large and unknown share of it. The curve says that PHP projects are built, tested and deployed in growing numbers. It does not count projects, and you cannot set one registry's package count against another's either, because what counts as a "package" differs between ecosystems.

Since Composer 2.4, released in August 2022, `composer audit` checks every installed package against the security advisories that Packagist serves through its API, and fails the build when it finds a known vulnerability, an abandoned package or a package flagged as malware. The advisories come from the ecosystem's shared database, which held 1,649 of them in September 2026. Extensions written in C, which Composer never handled, gained an installer of their own in 2025 and 2026: PIE, the PHP Installer for Extensions, funded by a public grant and hosted under the php organisation.



Frameworks and platforms

The full-stack frameworks in active development are CakePHP, Laminas, Laravel, Symfony and Yii; the micro-frameworks, Mezzio and Slim; the content platforms, Drupal, Joomla, TYPO3 and WordPress; the commerce platforms, Adobe Commerce with its Magento Open Source edition, PrestaShop, Shopware, Sylius and WooCommerce. The order is alphabetical throughout, and I recommend none of them.

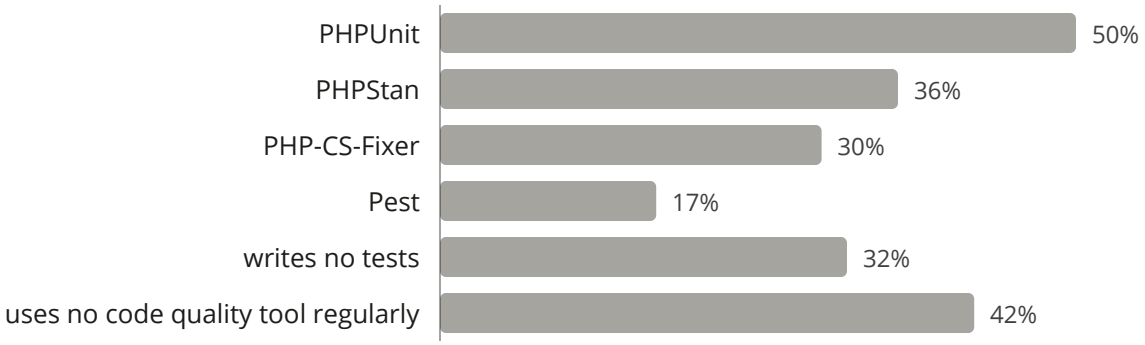
How they relate to each other can be said with a source. Symfony publishes the list of projects built on its components, and it includes Drupal, Joomla, TYPO3, Magento, PrestaShop, Shopware and Sylius, as well as Composer itself. Those components, and Laravel's, are what you meet under most of the platforms of Footprint other than WordPress, and that is why the ecosystem is less fragmented than a list of names suggests. The usage figures come from a survey: among developers who name PHP as their main language, 64 percent reported using Laravel, 25 percent WordPress and 23 percent Symfony, with several answers allowed. The sample leans toward the vendor's customers, so read those as survey responses, not as a ranking I endorse.

Tooling

Every category has at least two maintained options, and an afternoon with each is the way to pick. For tests, Pest and PHPUnit. For static analysis, PHPStan and Psalm, both of which read a docblock type syntax richer than the language and enforce the generics, shapes and conditional types that the engine cannot. For code style, PHP-CS-Fixer and PHP_CodeSniffer, both able to enforce the PER Coding Style standard. For editing, PhpStorm and VS Code with a PHP extension, each with a language server; for serving, FrankenPHP, PHP-FPM behind a web server and RoadRunner. Two tools stand alone: Xdebug, the debugger, which also profiles, and Rector, which rewrites code to newer syntax and is how a large codebase moves across PHP versions.

How much of that tooling gets used is another matter, and the survey figure on it is the least flattering one in this chapter.

Tools PHP developers report using



JetBrains, The State of PHP 2025 (October 2025), 1,720 respondents with PHP as their main language, from the Developer Ecosystem survey.
Several answers per respondent. Sample weighted toward JetBrains users.

Half of the PHP developers in the JetBrains sample use PHPUnit, a third use PHPStan, and a third write no tests at all; 42 percent use no code quality tool regularly. Whether that is worse than in other ecosystems the survey does not say, and I will not guess. The tooling exists; whether it gets used is a property of the team, not of the language.

Standards

A library that types its logger as `Psr\Log\LoggerInterface` works under any framework, and that is what the PHP Framework Interoperability Group, PHP-FIG, exists for. It publishes the standards that let libraries from different authors fit together: 14 accepted PHP Standards Recommendations as of September 2026, covering autoloading (PSR-4), logging (PSR-3), HTTP messages and handlers (PSR-7, 15, 17), caching (PSR-6, 16), containers (PSR-11), events (PSR-14) and clocks (PSR-20), plus the PER Coding Style, at version 3.1, which replaced PSR-12 as the coding standard.

The language's own distribution is the last piece. php.net has published source tarballs signed by the release managers since 2012, with their checksums; the official Docker images `php:8.5-cli`

and `php:8.5-fpm` track each release; and the mainstream Linux distributions package a version they patch themselves, which is not always one php.net still supports.

> The limit: the ecosystem is a web ecosystem. Its depth in HTTP, templating, ORMs, queues, payment and content is not matched in numerical computing, machine learning, data pipelines or desktop and mobile applications, where the packages are few and the community small. If your product needs those, you will use another language for that part, and Where PHP Is the Wrong Choice says so at length.

What to verify yourself

Run `composer create-project` with the skeleton of any framework of your choice, then `composer audit`, and read the lock file it produced; the whole supply chain is in front of you in ten minutes. Then search Packagist for the three libraries your project cannot live without, and check the date of their last release and the number of their open issues. That check tells you more than the package count does.

Under all of it sits an interpreter, and who maintains that is a question of its own.

Governance and Longevity

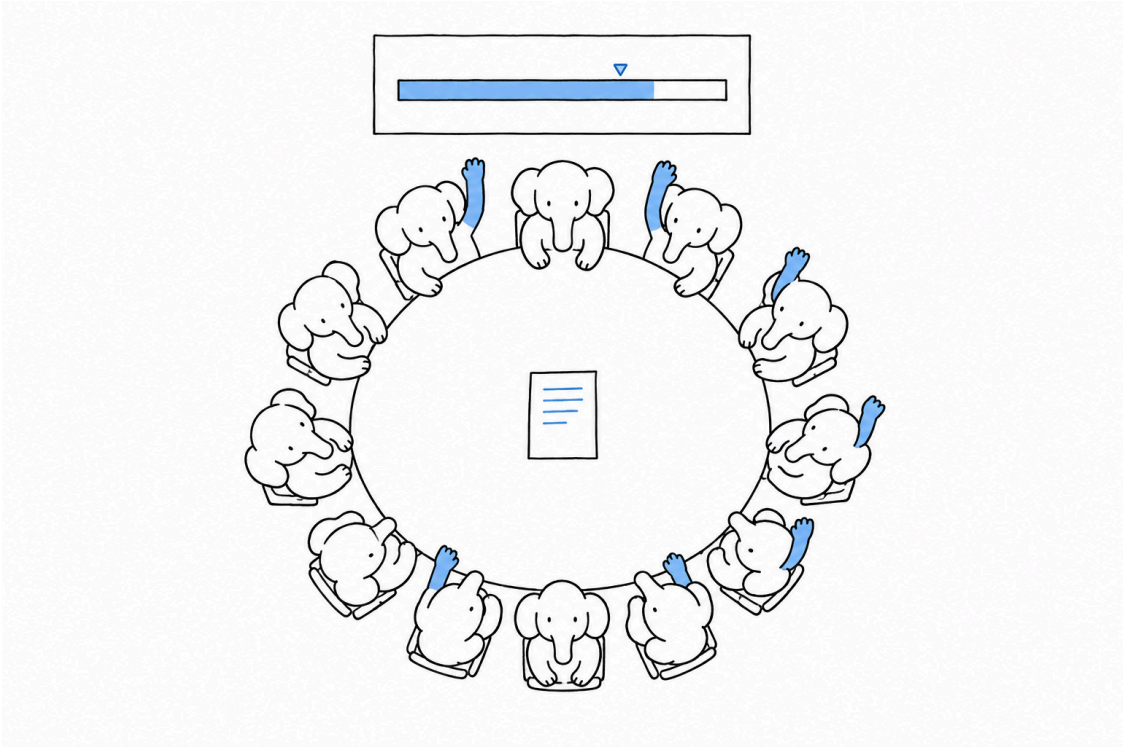
Before you bet a codebase on anything, you want to know who decides what goes into it, who is paid to maintain it, and how long a version stays supported. **PHP changes through a public RFC process with a two-thirds vote, ships one minor version every year and supports each for four, and since 2021 has a foundation that contracts thirteen engineers, who authored 42 percent of the interpreter's commits in 2025.** The money behind that foundation is under one million dollars a year, and that figure is as much part of the answer as the rest.

How a change gets in

Every change to the language goes through a Request for Comments on wiki.php.net. The proposal is written down, discussed for at least two weeks on the internals mailing list, then put to a vote open for at least two weeks, and it passes only with two thirds of the votes cast; the voters are contributors with a php.net account and a small number of community representatives, and the two-thirds rule has been strict since February 2019. Every step is public, down to the votes cast by name.

Nullable intersection types were rejected 12 to 26 in 2021. Auto-capturing multi-statement closures won a majority of 27 to 16 in 2022 and were declined all the same, for missing the two-thirds bar. Asymmetric visibility was declined 14 to 12 in January 2023, revised, accepted 24 to 7 in August 2024 and shipped in PHP 8.4, and nested classes were declined 2 to 20 in May 2025. That list of refusals is the evidence that the process is real.

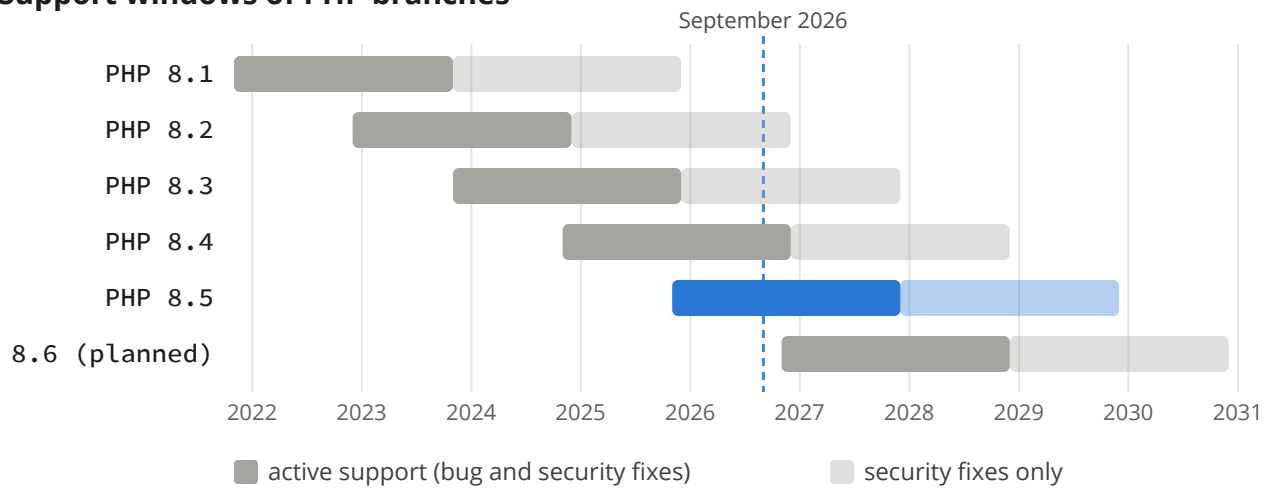
The pace, according to a secondary index, is 31 RFCs for PHP 8.4, 19 for PHP 8.5, and 29 already listed for PHP 8.6 in September 2026. If you come from a language steered by a single vendor or a benevolent dictator, weigh what this means: nothing enters PHP because someone important wants it, and features that a majority wanted have been refused. The process is slow, and it is public.



The release calendar

Since December 2015 PHP has shipped one minor version every year, eleven in a row, each between 20 November and 8 December. Each branch then gets two years of active support, with bug and security fixes in monthly point releases, followed by two years of security fixes only. Since an RFC voted in April 2024, both windows end on 31 December of their final year, so you can plan upgrades by calendar year.

Support windows of PHP branches



php.net/supported-versions and php.net/eol, checked 17 September 2026. PHP 8.6 dates from wiki.php.net/todo/php86, planned, not released.

In September 2026, four branches are supported: PHP 8.2 in security support until 31 December 2026, 8.3 until the end of 2027, 8.4 in active support until the end of 2026, and 8.5 in active support until the end of 2027. PHP 8.6 is scheduled for general availability on 19 November 2026, with its

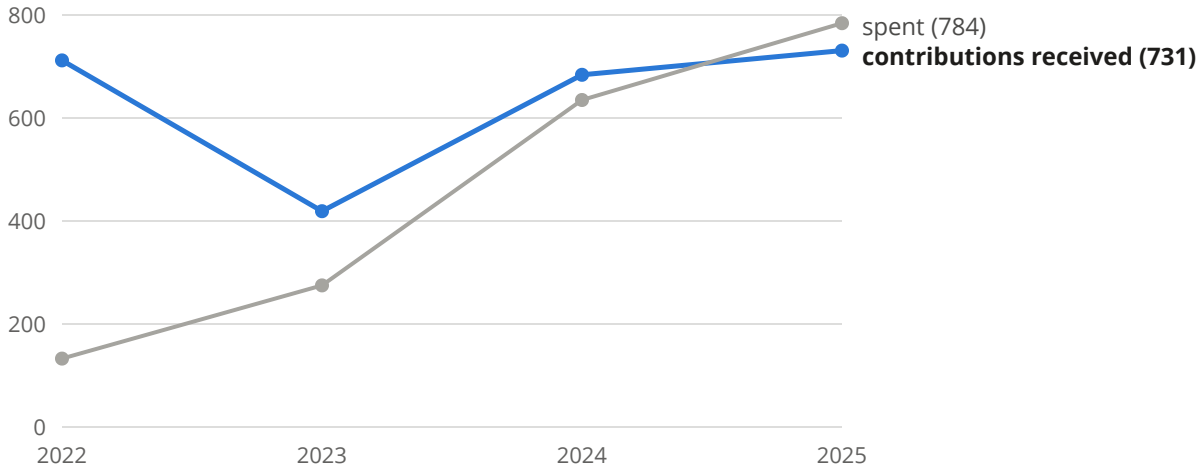
feature freeze in August, release candidates from 24 September and three named release managers. No date exists for a PHP 9.0, and I give none. PHP Versions, 2015 to 2026 lists every release with its dates.

Behind the calendar, the repository shows the activity. In the twelve months to 17 September 2026, the main branch of php-src received 5,316 commits from 171 distinct authors, and the repository counts 1,644 contributors across a GitHub history that begins in 2011.

Who is paid

Until 2021 the interpreter was maintained by volunteers and by a handful of engineers employed by companies with a stake in PHP. In November 2021, when one of the most active core developers announced a move away from core work, ten companies created The PHP Foundation to fund core development directly: Acquia, Automattic, Craft CMS, JetBrains, Laravel, PrestaShop, Private Packagist, Symfony, Tideways and Zend by Perforce. The foundation is hosted fiscally by Open Source Collective, lists every transaction on its Open Collective page and publishes a yearly transparency report.

The PHP Foundation, thousands of US dollars per year



The PHP Foundation transparency reports for 2022, 2023, 2024 and 2025 (published November 2022, February 2024, March 2025, May 2026).
2022 spending covers April to November only; 2025 spending is the total reported, with a deficit of about 139,000 dollars stated as deliberate.

Contributions were 712,000 dollars in 2022, 419,000 in 2023, 684,000 in 2024 and 731,000 in 2025. Spending on engineers rose from 275,000 dollars in 2023 to 635,000 in 2024 and 784,000 in 2025, a year the foundation closed with a deficit it describes as deliberate. That is the scale, and the reports give it themselves.

What the money buys, in September 2026, is thirteen full-time and part-time engineers under contract, an executive director and a board of ten unpaid members; those engineers authored 42 percent of the commits to php-src and 32 percent of the merged pull requests in 2025. Two public grants add to the sponsors' money. The German Sovereign Tech Agency funded 205,000 euros of

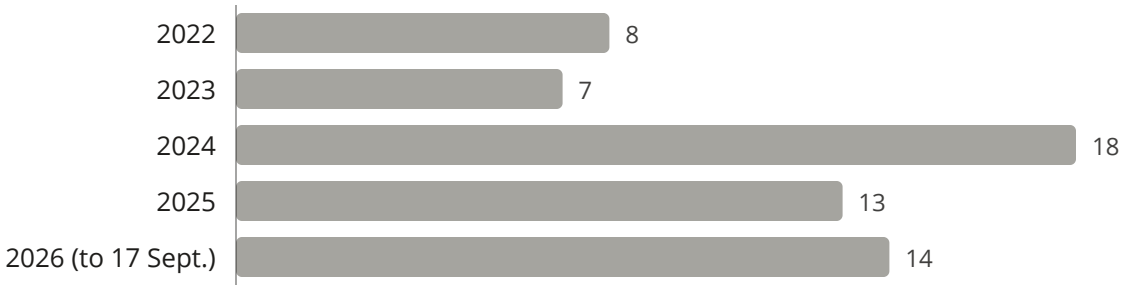
work in 2023 and 2024 and 223,680 euros in 2025 and 2026, and a grant from Alpha-Omega, a Linux Foundation project, has funded an ecosystem security team since May 2026.

The dependence is in the same reports. Two sponsors, Automattic and JetBrains, account for 762,500 and 476,670 dollars respectively of the 3.37 million dollars recorded on the foundation's Open Collective page since November 2021, about 37 percent between them; that running total and the yearly reports do not use the same scope, and I have not reconciled them. The number of sponsoring organisations and individuals fell from 658 in 2024 to 536 in 2025, which the foundation itself calls "substantially fewer". A budget under a million dollars is small for a language with the footprint described in Footprint: it pays for a dozen engineers, not for a research group. The foundation publishes every one of those figures itself.

Security

A vulnerability reaches the project through GitHub's private advisory workflow on the php-src repository or by email to the security team. A published policy classifies it as high, medium or low severity, and the two higher classes are fixed in a private repository before a coordinated release. Release tags have been signed by the release managers since April 2012, and every tarball ships with a detached signature and a published checksum. I found no software bill of materials and no build attestation for the release tarballs, and I report that as "not found" rather than "does not exist".

Vulnerabilities published for the PHP interpreter, per year



NVD, records for cpe:2.3:a:php:php by publication year, queried 17 September 2026. The php-src advisory list on GitHub gives 5, 18, 14 and 15 for 2023 to 2026.
PHP assigns no CVE to most low-severity issues; the 2024 rise coincides with a funded external audit. Counts cannot be compared across languages.

The interpreter had 8 vulnerabilities published in 2022, 7 in 2023, 18 in 2024, 13 in 2025 and 14 in 2026 up to 17 September. The 2024 rise coincides with an external audit of the interpreter, commissioned with public money, which found 27 issues of which 17 had security implications, three of them high, and produced several of that year's CVEs. The chart is easy to misread. PHP's own policy assigns no CVE to most low-severity issues, so the count reflects the policy as much as the code. Nor can a count of vulnerabilities be compared between languages: what counts as "the language", a runtime, a standard library, a package registry, differs in each, and so does the disclosure policy.

> The limit: the language is governed by its contributors, funded by a foundation with a budget under a million dollars a year and two dominant sponsors, and secured by a small team and a policy that assigns fewer CVEs than some others would. Those are the facts of a community project, and they are published by the project itself; if you compare them with a language backed by a large vendor's payroll, compare the risks on both sides, including the vendor's freedom to change course.

What to verify yourself

Open wiki.php.net/rfc and read one RFC currently under vote, then the internals mailing list thread behind it; an hour there shows how decisions are made better than any description. Open the foundation's Open Collective page, where every contribution and every expense is listed by date. Open the php-src security advisories on GitHub and read the most recent three, including the time from report to fix.

What all of this costs you, in salaries, hosting and upgrades, is the next question.

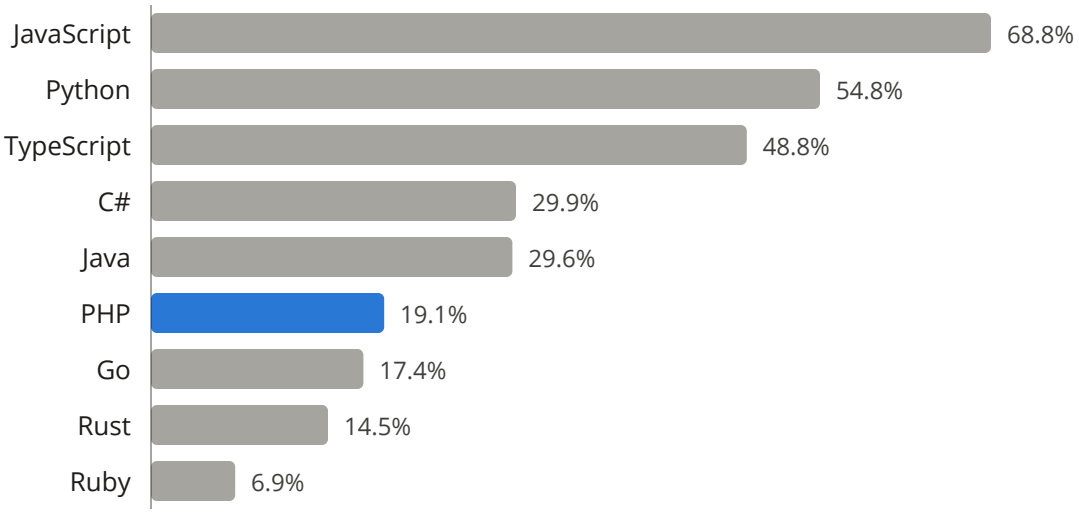
Cost of Ownership

A language costs you three things: the people who write it, the machines that run it, and the calendar you spend keeping it current. PHP has a figure for each, and each figure has two readings. **PHP developers are numerous and, by the surveys' own medians, paid less than developers of most other languages; the runtime has the fewest moving parts to host, files served by a process pool; and the yearly release cadence imposes a yearly upgrade budget that a large share of the installed base has not been paying.** The cheap reading and the dear reading come from the same sources, and you will need both.

People

One professional developer in five wrote PHP in the past year. That places it twelfth among languages in the largest developer survey, within a few points of Go and C.

Languages used by professional developers, past year

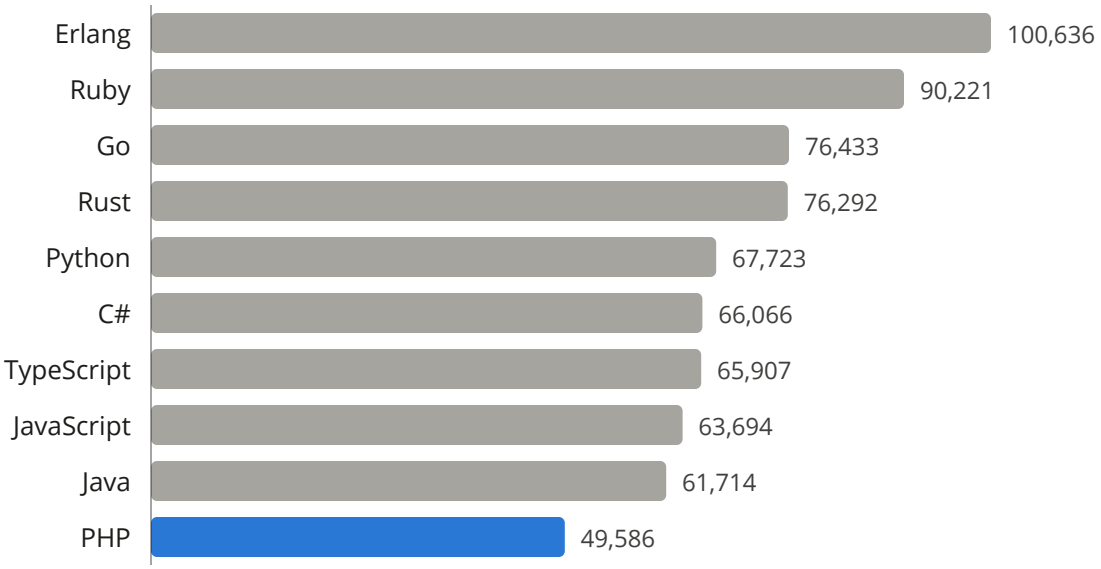


Stack Overflow Developer Survey 2025, professional developers (n = 24,759), fielded May to June 2025.
Self-selected respondents recruited through Stack Overflow. Several answers per respondent.

Another survey agrees on the order of magnitude: 17 percent of respondents used PHP in the JetBrains survey of 2025, and 9 percent named it their primary language. The labour market itself is thinly documented in public. One aggregator of British job advertisements counted 613 permanent positions citing PHP in the six months to September 2026, up from 404 a year earlier, at a median advertised salary of 48,676 pounds, up 14.5 percent. I found no comparable public series for other countries, so I report none.

The salary figure that does exist is the survey median, and it is low.

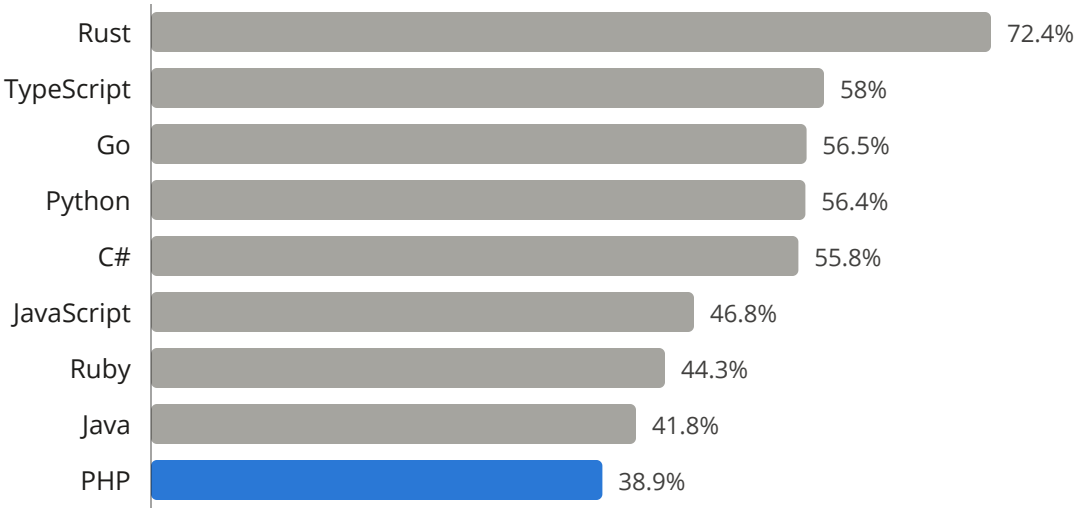
Median yearly salary reported by developers using each language, US dollars



Stack Overflow Developer Survey 2024, "Top paying technologies", programming languages, median in US dollars. The 2025 edition did not publish this table.
Self-reported, worldwide, not adjusted for country. PHP was the second-lowest of fifty languages listed.

Developers using PHP reported a median of 49,586 dollars a year in the last edition of the Stack Overflow survey to publish salaries by language, the second lowest of fifty languages, against 63,694 for JavaScript, 67,723 for Python and 90,221 for Ruby. The figure is worldwide and self-reported, and PHP's respondents live disproportionately in countries with lower wages, so it says less about your city than it appears to. Read it as a cost and PHP staffing is inexpensive in the aggregate. Read it as a signal and the market values the median PHP position below the median position in other languages. If you are building a team, you need both readings.

Developers who used a language and want to keep using it



Stack Overflow Developer Survey 2025, "admired" measure (n = 31,771).
Share of respondents who used the language in the past year and want to continue.

Sentiment is harder to price than salary. Among developers who used a language in the past year, 38.9 percent of PHP's users want to keep using it, the lowest of the mainstream languages and well below Rust at 72.4, TypeScript at 58.0 or Python at 56.4. The same year, 58 percent of developers whose main language is PHP said they did not plan to migrate to another. The two figures describe different populations, the occasional user and the specialist, and together they describe a language people are more often assigned to than drawn to. For hiring, that means a large pool and a retention question, and the weighting is yours.

Machines

A PHP application in production is files on a disk, served by a pool of processes. There is no long-running application process to monitor, restart or drain, no warm-up, no heap to size: that is the shared-nothing model of The Runtime, and it gives PHP the fewest moving parts to host. It is why every shared hosting provider in the world offers it, and why the platforms of Footprint can be installed by people who are not engineers. For most applications, a container image of the official `php:8.5-fpm` build, a web server in front and a load balancer are the whole production architecture, and adding capacity is adding a copy.

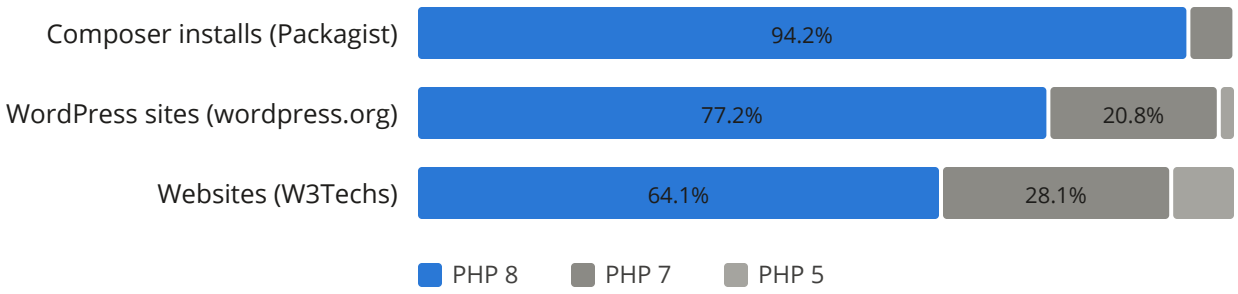
What you pay for that simplicity is efficiency per machine. In the benchmark of Throughput and Latency, a framework under PHP-FPM serves a few tens of thousands of requests per second on a large server; a worker runtime multiplies that by three to four, and brings with it the operational discipline of a long-running process. The engine's own progress shows in the same ledger. Badoo reported in 2017 that moving its hundreds of application servers from PHP 5 to PHP 7 saved it about one million dollars in hardware, a self-reported figure from the company's engineering blog. I have no independent measurement of hosting cost per request across languages, and I make no claim about one.

Calendar

Wikimedia's upgrade history is public: production moved from PHP 7.4 to 8.1 in March 2025, to 8.3 in November 2025, and was planning 8.5 for late 2026. **A minor version ships every year in late November or early December and is supported for four years, so you must plan one upgrade a year to stay in active support, or one every two to three years to stay in security support.** The work of an upgrade is bounded: the language deprecates in a minor version and removes at the next major, the analysers report the deprecations, and Rector applies the mechanical rewrites. On a well-tested codebase a version upgrade is measured in days, on an untested one in weeks.

The installed base shows what happens when that budget is not paid.

PHP versions in use, three populations



Packagist php-statistics, August 2026; api.wordpress.org/stats/php, 17 September 2026; W3Techs, 17 September 2026.
PHP 4 (0.1% of websites) folded into PHP 5.

Among packages installed through Composer, 94 percent of installs in August 2026 ran on PHP 8 and 0.2 percent on PHP 5. WordPress sites reporting to wordpress.org stood at 77 percent PHP 8, 21 percent PHP 7 and 2 percent PHP 5. The web as W3Techs detects it ran 64 percent PHP 8, 28 percent PHP 7 and 8 percent PHP 5, and those last two are versions that have received no security fix since November 2022 and December 2018 respectively. Those are three populations: the developers who build, the sites that update themselves, and the web as it is. The gap between the first and the third is the ecosystem's technical debt, and if you inherit a PHP codebase, ask which population it belongs to before you quote a price.

> The limit: PHP's staffing is cheap by the medians and its hosting has the fewest moving parts, and each of those facts comes with its shadow, a retention question and an installed base of which more than a third runs unsupported versions. Budget the yearly upgrade and hire for the specialist rather than the occasional user, and you get the cheap side of each; skip either, and you get the other.

What to verify yourself

Search the job boards of your own city for the language, then for the frameworks you would use, and compare the count and the advertised range with the languages you are considering, which beats any worldwide median. Ask your hosting provider or your platform team what PHP version they run today and how they would upgrade it. Then take one open-source PHP project of your codebase's size that is two versions behind, and run Rector's upgrade set on it: the diff it produces is the cost of a version, in front of you.

The cases where this arithmetic stops applying, because PHP is the wrong tool rather than a dear one, are collected in Where PHP Is the Wrong Choice.

Where PHP Is the Wrong Choice

An evaluation that cannot name the cases where its subject loses is not an evaluation, and PHP loses in several. **PHP is the wrong choice for CPU-bound computation, for services whose job is to hold many long-lived connections, for anything that runs outside a server, and for teams whose skills, platform and scale have already carried them somewhere else.** In each of those cases the honest advice is rarely “rewrite everything” and more often “not this part”, with a different tool for the part.

Computation

The n-body program takes 204 seconds in PHP with the JIT off, against 8.6 for Node.js, and the JIT narrows that gap without closing it, as Throughput and Latency showed. On code that keeps a core busy, the interpreter sits in the class of Python and Ruby, an order of magnitude behind V8, the JVM, .NET, Go and Rust. Around it there is no numerical stack of any depth: no array library with the reach of NumPy, no dataframe library with the reach of pandas, no machine learning framework that a research team would recognise. The packages that exist are few and maintained by small groups, and I found no scale figure to cite for any of them.

If your product is a model, a simulation, a signal processing pipeline or a large batch computation, use Python for the ecosystem or a compiled language for the speed. If that product also has a web front, the front is where PHP may still fit.

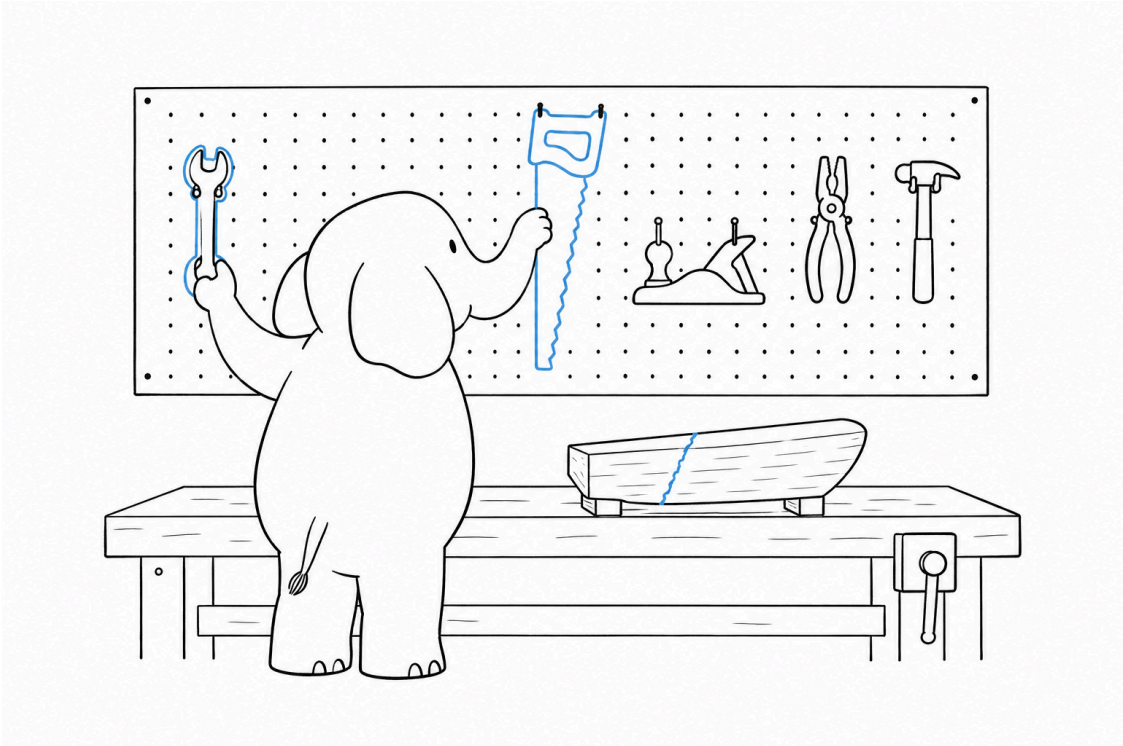
Long-lived connections

Ten thousand idle connections cost ten thousand processes under PHP’s default model, and a process is too expensive to spend on an idle socket. The async runtimes of Concurrency exist and perform, the Swoole entry serving 3.5 million pipelined requests per second on the plaintext test. But they are libraries with their own I/O clients rather than a property of the language, and most of the ecosystem’s packages were written for the blocking model. A chat backend, a multiplayer game server, a trading gateway, a streaming pipeline or a push notification service is the normal case for Node.js, Go, Elixir and Erlang, Java or C#, and those ecosystems have the libraries to match.

Latency budgets under a millisecond fall on the same side. The floor of a PHP-FPM request is below a millisecond on an idle machine, and nothing built on it stays there under load, so a system with that budget is written in a compiled language.

Outside the server

PHP is a language for servers. It has a command-line interpreter, and a great deal of tooling is written with it, but for desktop applications, mobile applications, browsers, embedded devices or the distribution of a standalone binary to end users, it has no story that any team should build on. If your users do not run a PHP interpreter, write the tool you give them in something else.



Language-level guarantees

A `match` with a missing arm throws at runtime. That one fact describes the type system of The Language in 2026: it is enforced at runtime, at the boundaries of functions and properties, with no generics in the language, no compile step, no ownership or lifetime model and no exhaustiveness checking. The ecosystem's answer is a static analyser run at its strictest level in every build, which gives a typed project most of what a compiler would, and many large PHP codebases live with that answer. If "the compiler guarantees it" is a requirement for you rather than a preference, because of the domain, the regulator or the size of the codebase, TypeScript, Java, C#, Kotlin, Go or Rust will serve you better, and I will not argue otherwise.

Teams and scale

One professional developer in five wrote PHP last year, one in eleven names it a primary language, and its share of the web has fallen from 80 to 70 percent in a decade. The share of developers is smaller than the share of running servers. Footprint also documents the companies that left PHP, and they left for Java, Go or TypeScript when a web monolith had grown into a set of services:

Zalando in 2010, Trivago in 2021, Dailymotion and BlaBlaCar in 2025 and 2026. Their public documents give the architecture as the reason, not the runtime's cost.

If your engineers already work in another language, on a platform built for it, adding PHP for a new service gains you little. The runtime's advantages, described throughout this book, are advantages of simplicity, and a second stack removes them. And if your company has reached the scale where it rewrites its monolith into typed compiled services, you are following a pattern in which PHP is the thing being left, as Ruby and Python are at the same stage elsewhere.

Where the doubt does not apply

The same evidence supports the opposite verdict, and without adjectives. Request-and-response web applications at any scale you are likely to reach: content, commerce, back offices, APIs, the platforms of Footprint and the custom applications built beside them. Teams that value a runtime with no process to babysit and a deployment that is a file copy. Organisations that must run software for a decade on hosting they do not control, where the installed base and the four-year support window matter more than throughput. And products whose hard part is the domain rather than the machine, which is most products.

> The limit of this chapter: it names the cases the evidence covers. A domain I have not measured I have not judged, and if yours is one of them, run the week that follows before believing either the reputation or me.

What to verify yourself

Write down the one workload in your product that worries you most, and look for it above. If it is here, the advice stands unless your own measurement contradicts it. If it is not, An Evaluation in One Week exists for it.

An Evaluation in One Week

Nothing I wrote should be the reason for your decision. The reason should be a measurement you made on your own workload, and it does not take long to make one. **Five working days are enough to install the language, read it, run it under load, build a slice of your product, and check the ecosystem and the governance for yourself.** You need a laptop, a small virtual machine and nobody else, and what you hold at the end is a page of numbers that nobody handed you.

Day one: the language

Install PHP 8.5 from your package manager or run the official image, `php:8.5-cli`. Paste the first example of The Language in 2026 into a file, run it, then break it in the ways that chapter suggests and read every error message. Clone one open-source PHP project of a size you care about and run its test suite. Then run a static analyser, PHPStan or Psalm, at its strictest level and read the first twenty findings:

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

composer require --dev vimeo/psalm
vendor/bin/psalm --init src 1 && vendor/bin/psalm
```

What the analyser catches and the engine does not is the practical answer to the question of generics. Record the time it took you to read the codebase, and whether the findings were things you would want caught.

Day two: the runtime

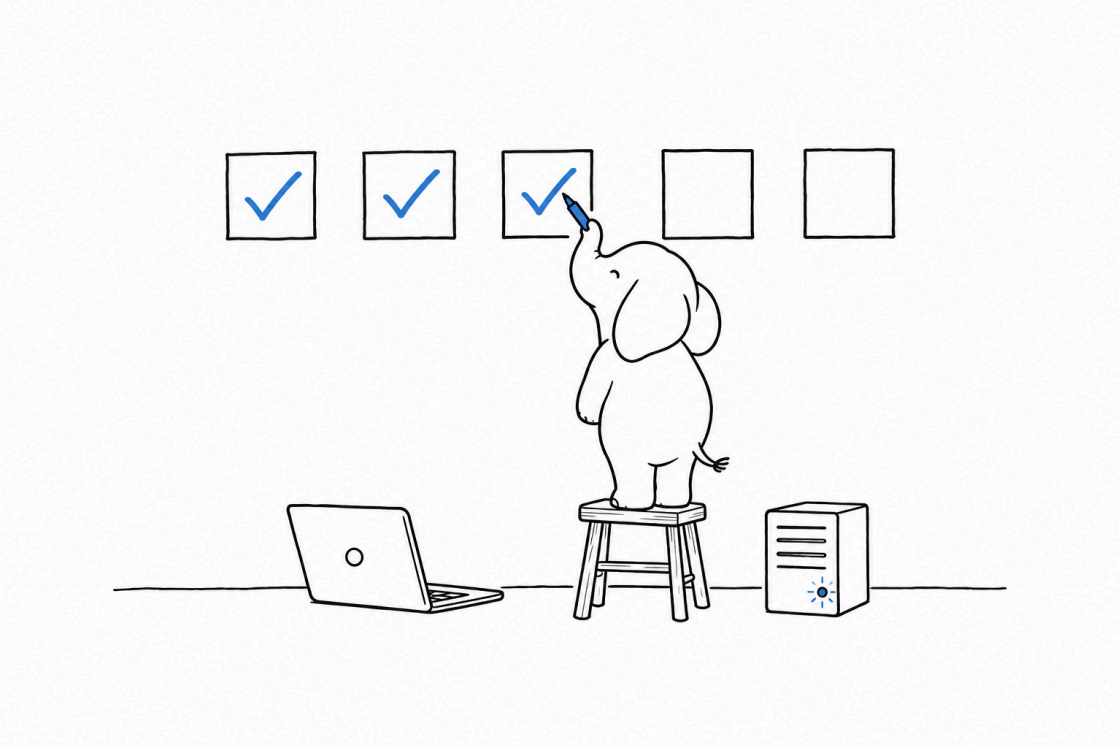
On the virtual machine, install PHP-FPM with nginx, enable OPcache, and serve a hello-world script. Then serve the same script under FrankenPHP in worker mode, and load both with the same tool at the same concurrency:

```
wrk -t4 -c64 -d30s --latency http://127.0.0.1/
```

Record for each run the requests per second, the median latency, the 99th percentile, and the memory per worker from the FPM status page or from `ps`. Then replace the hello-world with the skeleton of a full-stack framework, any of CakePHP, Laminas, Laravel, Symfony or Yii, and repeat. The ratio between the four runs is the runtime cost of your future application on your hardware, the figure that Throughput and Latency could only give you for someone else's.

Day three: a slice of the product

Pick the one feature of your product that best represents it: an authenticated endpoint that reads and writes a database, renders or serialises something, and sends one message to a queue. Build it in the framework you chose on day two, with tests, using only what the framework and Packagist provide, and do not optimise. Record how long it took, how much of it you wrote and how much you configured, how many packages it pulled in, and what `composer audit` said about them.



Day four: the slice under load

Load the slice from day three the way you loaded the hello-world, at the concurrency you expect in production and at ten times that. Profile one slow request with Xdebug's profiler or with `hrtime()` around the suspect calls and see where the time goes; on most slices it is in the database, and the language is a thin layer at each end. Record the throughput and the 99th percentile at both concurrencies, and the share of a request spent outside PHP.

Then do the one thing benchmarks never do. Kill a worker mid-request, fill the memory limit, throw an uncaught exception in a queue job, and record what the user saw, what the logs said, and

what recovered by itself.

Day five: the ecosystem and the governance

Search Packagist for the three libraries your product cannot live without, and record the date of each one's last release and its open issue count. Open wiki.php.net/rfc and read the RFC currently under vote, then the internals thread behind it. Open php.net's supported versions page and write down the date your chosen version stops receiving security fixes. Open the foundation's Open Collective page and read the last month of transactions, then the three most recent php-src security advisories, with their time from report to fix. The whole of it fits in an afternoon, and it is the raw material that Governance and Longevity summarised for you.

The decision

The week produced numbers, and a decision needs weights that only you can set. Each question below has its public evidence in one chapter and your own answer in one day of the week.

Question	Public evidence	Your result from the week
Will I want to read and write this language for years?	The Language in 2026	Day one
Is the runtime's throughput enough for my traffic, on my hardware?	Throughput and Latency	Day two, day four
Does my workload fit the request model, or does it hold connections or burn CPU?	The Runtime, Concurrency, Where PHP Is the Wrong Choice	Day four
Do the libraries I need exist and are they maintained?	The Ecosystem	Day three, day five
Who maintains the language, and until when is my version supported?	Governance and Longevity	Day five
Can I hire for it, host it, and afford the yearly upgrade?	Cost of Ownership	Your job boards, your platform team
Is it running, at scale, at organisations I would trust to have checked?	Footprint	Their public documents

Some outcomes recur. When the workload is request-and-response, the throughput on day two exceeded your need by a margin and the libraries on day five were alive, my evidence and yours agree, and the decision is about your team rather than the language. When day four showed a CPU-bound or connection-bound core, the honest conclusion is a different language for that core, with PHP possibly around it. And when the week was pleasant but the job boards of your city were

empty, or the retention figures of Cost of Ownership worry you more than the runtime does, that is a legitimate reason to decline. I will not argue with it, even under the umbrella of the language's own foundation.

> The limit: a week measures a slice, not a product, and the team that runs the slice is not the team that will run the product for ten years. The week removes the reputation from the decision, and leaves the judgment.

What to verify yourself

Everything above. If one figure in these pages turns out wrong when you check it, the sources appendix gives the URL where the correct one lives, and the repository behind this book accepts corrections. That is the arrangement you should expect from any document that asks for your time.

Appendix

Three reference sections close the book, and each one exists so that you can check rather than trust.

A - Sources lists every figure used in the book, chapter by chapter, with the publisher, the URL, the date the data refers to and the date it was last checked. It closes with the figures I refused, and why.

B - PHP Versions, 2015 to 2026 gives the release date, the support window and the headline changes of every version from PHP 7.0 to PHP 8.5, so that a figure about “PHP 8” can be placed in time.

C - Glossary defines the terms I use that you may not know if you come from another ecosystem.

A - Sources

Every figure used in the book, chapter by chapter, with its publisher, the date the data refers to, and the URL. “Checked” is the date the page was last read for this edition. Where a source is a vendor, a self-report or a synthetic benchmark, the entry says so, as the chapter did. Where a chapter cites a document I could not read directly, the entry says that too.

Charts are generated from the data files in `charts/data/`, each of which repeats its source, and the raw TechEmpower tables used by chapters 3 and 4 are stored as CSV in `charts/raw/`.

How to Read This Book

- PHP release history and the features named: php.net release pages and changelogs, checked 17 September 2026. https://www.php.net/releases/
- The PHP Foundation employing core developers since 2021: JetBrains announcement, 22 November 2021. https://blog.jetbrains.com/phpstorm/2021/11/the-php-foundation/

Footprint

- Share of websites by server-side language (PHP 69.9 percent, JavaScript 7.5, Ruby 7.1, Java 5.4, Scala 5.0, ASP.NET 4.2, Python 1.1): W3Techs, data of 17 September 2026. https://w3techs.com/technologies/overview/programming_language
- Historical share by year, values on 1 January (PHP 80.6 percent in 2015, 75.2 in 2025, 72.4 in 2026): W3Techs historical trends, checked 17 September 2026. https://w3techs.com/technologies/history_overview/programming_language/ms/y
- JavaScript overtaking Ruby as second server-side language: W3Techs “technology fact of the day”, 27 July 2026. https://w3techs.com/
- W3Techs methodology (sample of over 20 million sites, one site per domain, redirected domains excluded, daily updates): checked 17 September 2026. https://w3techs.com/technologies
- WordPress 40.2 percent of all websites and 58.8 percent of CMS market; Joomla 1.1, Drupal 0.6, PrestaShop 0.5, TYPO3 0.3; 31.5 percent of sites without a detected CMS: W3Techs, 17 September 2026. https://w3techs.com/technologies/overview/content_management
- WooCommerce 8.0 percent of all websites, 47.7 percent of e-commerce systems: W3Techs, 17 September 2026. https://w3techs.com/technologies/details/cm-woocommerce
- Moodle 146,634 registered sites and 531 million users (self-registered sites only): stats.moodle.org, checked 17 September 2026. https://stats.moodle.org/

- Nextcloud “500K+ servers”: Nextcloud, 2025 wrap-up, 11 December 2025. https://nextcloud.com/blog/nextcloud-2025-wrap-up/
- Drupal 470,795 sites reporting their version: drupal.org usage statistics, week of 6 September 2026. https://www.drupal.org/project/usage/drupal
- PrestaShop nearly 250,000 sites and more than 22 billion euros of sales in 2024 (self-reported): prestashop.com, checked 17 September 2026. https://prestashop.com/about-us/
- Shopware on 115 of the 1,000 largest German B2C shops (EHI study 2025): Shopware, 10 January 2025. https://www.shopware.com/en/news/shopware-is-the-market-leader-among-ecommerce-platforms-in-germany/
- Matomo “more than 1.4 million websites”: Matomo, 1 September 2026. https://matomo.org/blog/2026/09/matomo-launches-reseller-partner-programme/
- Wikimedia 19.4 billion page views per month: Wikimedia Foundation audited financial statements, fiscal year 2023 to 2024. https://upload.wikimedia.org/wikipedia/foundation/f/f6/Wikimedia_Foundation_2024_Audited_Financial_Statements.pdf
- Wikimedia production on PHP 8.3 since 25 November 2025: Phabricator T360995, resolved 18 December 2025. https://phabricator.wikimedia.org/T360995
- Wikimedia migration to PHP 8.5 planned for late 2026: Phabricator T413223, created 19 December 2025. https://phabricator.wikimedia.org/T413223
- Wikimedia end of HHVM in production: Phabricator T176370, resolved 23 November 2019. https://phabricator.wikimedia.org/T176370
- Wikimedia 800,000 requests per second at the edge and seven data centres: Wikimedia Foundation, 26 November 2025. https://wikimediafoundation.org/news/2025/11/26/how-does-the-wikimedia-foundation-use-donations-to-wikipedia/
- Automattic: Tumblr and WordPress “run on PHP primarily”: automattic.com, checked 17 September 2026. https://automattic.com/work-with-us/tumblr-migration/
- WordPress VIP 2.4 trillion requests per year and 22 billion on election night 2024 (marketing page, CDN included): wpvip.com, checked 17 September 2026. https://wpvip.com/capabilities/performance/
- Etsy engineers “primarily code in PHP and JavaScript”: careers.etsy.com, checked 17 September 2026. https://careers.etsy.com/engineering
- Etsy on PHP 7 in 2016: talk “Deploying PHP 7”, PHP Conference Philippines, 13 October 2016. https://talks.php.net/ph16
- Mailchimp PHP job runner and monolith: Mailchimp developer blog, “Computers are the easy part”, undated, checked 17 September 2026. https://mailchimp.com/developer/blog/computers-are-the-easy-part/ Job postings on jobs.intuit.com asking for PHP, 2025 to 2026, were read but expire and could not be archived; the chapter relies on the blog post.
- Badoo three million lines of PHP, hundreds of servers, one million dollars saved: Bumble Tech, 7 February 2017. https://medium.com/bumble-tech/how-badoo-saved-one-million-dollars-switching-to-php7-fcbda94ad3ae

- European Commission websites on Drupal: the generator metadata of commission.europa.eu declares Drupal 11, checked 17 September 2026. The figure of 770 live sites comes from a Commission staff presentation at Drupal4Gov EU, 29 January 2026, as reported by attendees; the slides were not found online and the figure is not confirmed by a Commission publication. https://drunomics.com/en/blog/drupal4gov-eu-2026-how-drupal-powers-european-governments-247
- White House on WordPress VIP with FedRAMP Moderate authorisation: wpvip.com, March 2025. https://wpvip.com/blog/fedramp-moderate/
- Germany's Government Site Builder on TYPO3, more than 80 authorities and 250 sites: ITZBund and typo3.com, checked 17 September 2026. https://www.itzbund.de/DE/itloesungen/standardloesungen/gsb/gsb.html
- GovCMS more than 370 sites for 115 agencies: govcms.gov.au, 2025, as cited by The PHP Foundation, 2 September 2026. https://thephp.foundation/blog/2026/09/02/digital-sovereignty-is-written-in-php/
- LocalGov Drupal 77 council sites: localgovdrupal.org, checked 17 September 2026. https://localgovdrupal.org/
- HHVM ending PHP support: hhvm.com, 12 September 2018 and 11 February 2019. https://hhvm.com/blog/2018/09/12/end-of-php-support-future-of-hack.html and https://hhvm.com/blog/2019/02/11/hhvm-4.0.0.html
- Slack on Hack, about five million lines: Slack Engineering, 14 April 2020 and 8 February 2023. https://slack.engineering/hacklang-at-slack-a-better-php/ and https://slack.engineering/hakana-taking-hack-seriously/
- Zalando rewrite in Java in 2010: account of a former Zalando engineer, 3 March 2020. https://srcco.de/posts/one-decade-in-zalando-tech.html
- Trivago move from PHP to TypeScript: trivago tech blog, 16 May 2022. https://tech.trivago.com/post/2022-05-16-warp-a-web-application-rewrite-project
- Dailymotion "gradually reducing PHP", new development in Go, Java and Python: job posting "Senior Backend Software Engineer, Video Platform", read 17 September 2026 (postings expire). https://jobs.smartrecruiters.com/Dailymotion/744000136260231-senior-backend-software-engineer-video-platform-full-remote-from-france- Historical use: Symfony blog, 18 February 2011.
- BlaBlaCar migration "from a PHP/Symfony stack towards a Java/JS-dominated one": job posting "Software Engineer, Backend", read 17 September 2026 (postings expire). https://jobs.smartrecruiters.com/BlaBlaCar/743999670001836-software-engineer-backend Historical use: SymfonyCon Paris, 4 December 2015.
- Developer survey figures (PHP 19.1 percent of professional developers, 12th): Stack Overflow Developer Survey 2025, fielded May to June 2025, 49,009 responses. https://survey.stackoverflow.co/2025/technology
- JetBrains 17 percent used (13th language), 9 percent primary (9th), "long-term decline": JetBrains State of Developer Ecosystem 2025, 24,534 respondents, fielded April to June 2025.

https://devecosystem-2025.jetbrains.com/

- 58 percent of PHP developers not planning to migrate: JetBrains, The State of PHP 2025, October 2025, 1,720 respondents. https://blog.jetbrains.com/phpstorm/2025/10/state-of-php-2025/
- PHP sixth language by monthly contributors: GitHub Octoverse 2025, 28 October 2025. https://github.blog/news-insights/octoverse/octoverse-a-new-developer-joins-github-every-second-as-ai-leads-typescript-to-1/
- PHP fourth, tied with C#: RedMonk language rankings, January 2026, published 14 April 2026. https://redmonk.com/sogrady/2026/04/14/language-rankings-1-26/
- PHP 14th at 1.04 percent: TIOBE index, September 2026. https://www.tiobe.com/tiobe-index/

The Runtime

- PHP-FPM process model and configuration: php.net manual, checked 17 September 2026. https://www.php.net/manual/en/install.fpm.php
- OPcache: php.net manual, checked 17 September 2026. https://www.php.net/manual/en/book.opcache.php
- Preloading figures (30 percent and 50 percent on two framework hello-world pages, “will likely be lower”): RFC Preloading, wiki.php.net, 2018, accepted 48 to 0. https://wiki.php.net/rfc/preload
- JIT: “about 3 times better performance on synthetic benchmarks”, “typical application performance is on par with PHP 7.4”: php.net, PHP 8.0 release announcement. https://www.php.net/releases/8.0/en.php
- JIT on WordPress, 326 against 315 requests per second: RFC JIT, wiki.php.net, 2019. https://wiki.php.net/rfc/jit
- JIT disabled by default since PHP 8.4: php.net, OPcache configuration, `opcache.jit`. https://www.php.net/manual/en/opcache.configuration.php
- FrankenPHP under the php organisation since 8 June 2025: The PHP Foundation, 15 May and 8 June 2025. https://thephp.foundation/blog/2025/05/15/frankenphp/ and https://thephp.foundation/blog/2025/06/08/php-30/
- FrankenPHP worker mode, “not originally designed for long-running processes”, maximum requests setting: frankenphp.dev documentation, checked 17 September 2026. https://frankenphp.dev/docs/worker/
- RoadRunner: roadrunner.dev, checked 17 September 2026. https://roadrunner.dev/
- Swoole and OpenSwoole: openswoole.com and github.com/swoole/swoole-src, checked 17 September 2026.
- FrankenPHP classic mode against PHP-FPM, 18,403 against 18,478 requests per second, p99 0.9 ms, eight-vCPU virtual machine: Tideways, 23 September 2025 (independent of both projects). https://tideways.com/profiler/blog/testing-if-franken-php-classic-mode-is-faster-and-more-scalable-than-php-fpm

- Symfony under PHP-FPM, FrankenPHP and Swoole on the Fortunes test: TechEmpower Round 23, see the next section.
- Bare process cost (2 MB heap, 33 MB resident, 20 ms): measured by the author on PHP 8.3.33 on Linux, 17 September 2026, with `memory_get_usage(true)` and `/usr/bin/time -v`. Reproducible with `php -r ''`.
- Wikimedia container configuration (8 workers, 500 MB OPcache, 768 MB APCu, 1 to 2 GB per container): Phabricator T280497, 2021. https://phabricator.wikimedia.org/T280497

Throughput and Latency

- “Execution time is often halved”: Zend (Rogue Wave), white paper “Maximize performance and mitigate risks with PHP 7”, 9 November 2017; a vendor document without published methodology. https://www.zend.com/sites/default/files/pdfs/rw-maximize-performance-mitigate-risks-wp-fnl-20171109.pdf
- PHPBench scores for PHP 5.6.40 to 8.0-dev (288,326 to 876,420): OpenBenchmarking.org, result 2002269-VE-PHPBENCHM24 by Michael Larabel, February 2020, Core i9-9900KS, Ubuntu 20.04. https://openbenchmarking.org/result/2002269-VE-PHPBENCHM24
- WordPress 146.09 requests per second on PHP 8.2 and 148.30 on PHP 8.5, “incremental releases rarely produce large speed jumps”: Kinsta, 13 December 2025; a hosting vendor’s benchmark, ApacheBench at 15 concurrent requests on a 30-vCPU machine, JIT off. https://kinsta.com/blog/php-benchmarks/
- TechEmpower Round 23: results dated 24 February 2025, hardware described by TechEmpower as “ProLiant DL360 Gen10 Plus, Intel Xeon Gold 6330 (56 cores), 64 GB, 40 Gbps Ethernet”; the Xeon Gold 6330 has 28 cores and 56 threads. Results page (JavaScript-rendered), read with a headless browser on 17 September 2026, and stored as CSV in `charts/raw/`. https://www.techempower.com/benchmarks/#hw=ph&test=fortune&section=data-r23 Round announcement: https://www.techempower.com/blog/2025/03/17/framework-benchmarks-round-23/
- TechEmpower repository archived: github.com/TechEmpower/FrameworkBenchmarks, archived, last push 24 March 2026, read through the GitHub API on 17 September 2026.
- Wikimedia latency targets (50 ms median, 200 ms p99 for GET, 500 ms p99 for POST): Wikimedia, “Backend performance practices”, last edited 27 June 2026. https://wikitech.wikimedia.org/wiki/MediaWiki_Engineering/Guides/Backend_performance_practices
- Wikimedia public latency dashboard: Grafana, “Backend Pageview Timing”; the median of about 170 milliseconds quoted in the chapter was read from it on 17 September 2026 around 10:25 UTC, over one hour, for requests reaching the application servers. https://grafana.wikimedia.org/d/QLtC93rMz/backend-pageview-timing
- n-body elapsed times, fastest single-threaded entry per language (Rust 2.19 s, C# 3.13 s with native AOT, Java 6.02 s with native image, Go 6.39 s, Node.js 8.55 s, Ruby 166.67 s, PHP 204.10 s,

Python 360 s), spectral-norm (PHP 18.29 s elapsed for 72.34 s CPU; Python 90.37 s; Ruby 56.52 s; Node.js 1.60 s): The Computer Language Benchmarks Game, checked 17 September 2026. https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/nbody.html and https://benchmarksgame-team.pages.debian.net/benchmarksgame/performance/spectralnorm.html

- PHP command line used by the Benchmarks Game (PHP 8.4.1, `opcache.jit_buffer_size=64M`, no `opcache.jit` setting): program page for n-body PHP #3. https://benchmarksgame-team.pages.debian.net/benchmarksgame/program/nbody-php-3.html
- JIT four-fold gain on Mandelbrot: RFC JIT, wiki.php.net, 2019. https://wiki.php.net/rfc/jit

Concurrency

- Fibers: “all fibers exist within a single thread”, “blocking code … will continue to block the entire process”: RFC Fibers, wiki.php.net, accepted 50 to 14, March 2021. https://wiki.php.net/rfc/fibers
- AMPHP, ReactPHP: amphp.org and reactphp.org, checked 17 September 2026.
- TechEmpower Round 23 plaintext test figures: see the previous section; raw table in `charts/raw/techempower-r23-plaintext.csv`.
- “Concurrency Support in the PHP Engine” RFC under discussion: wiki.php.net RFC index, checked 17 September 2026. https://wiki.php.net/rfc The earlier “PHP True Async” RFC shows status Canceled, last edited 22 July 2026. https://wiki.php.net/rfc/true_async

The Language in 2026

- Every feature and its version: php.net release pages and the fact sheet in the polyglot book’s writing guide, both checked against php.net. Examples were run under PHP 8.5 with the official Docker image on 17 September 2026.

The Ecosystem

- Packagist 468,099 packages, 5.8 million versions, 199.6 billion installs since 2012; monthly install series (2.0 billion in 2016 to 36.0 billion in 2025; 2.58 billion in August 2025 and 4.93 billion in August 2026): packagist.org/statistics, read 17 September 2026. https://packagist.org/statistics
- `composer audit` introduced in Composer 2.4.0, 16 August 2022: getcomposer.org changelog. https://getcomposer.org/changelog/2.4.0 Documentation: https://getcomposer.org/doc/03-cli.md#audit
- 1,649 advisories in the shared database: github.com/FriendsOfPHP/security-advisories, counted on 17 September 2026.

- PIE 1.5, funded by the Sovereign Tech Agency: The PHP Foundation, quarterly report Q2 2026, 21 July 2026. https://thephp.foundation/blog/2026/07/21/quarterly-report-q2-2026/
- Projects built on Symfony components (self-declared): symfony.com/projects, checked 17 September 2026. https://symfony.com/projects
- Laravel 64 percent, WordPress 25 percent, Symfony 23 percent among PHP developers; PHPUnit 50, PHPStan 36, PHP-CS-Fixer 30, Pest 17, 32 percent writing no tests, 42 percent using no quality tool: JetBrains, The State of PHP 2025, October 2025. https://blog.jetbrains.com/phpstorm/2025/10/state-of-php-2025/
- 14 accepted PSRs, PER Coding Style 3.1: php-fig.org, checked 17 September 2026. https://www.php-fig.org/psr/ and https://www.php-fig.org/per/coding-style/
- Signed release tags since April 2012: php.net GPG keys page. https://www.php.net/gpg-keys.php

Governance and Longevity

- RFC process and voting (two-thirds majority, two weeks of discussion, two weeks of voting, who votes): wiki.php.net, “How to create an RFC” and “Voting”. https://wiki.php.net/rfc/howto and https://wiki.php.net/rfc/voting
- Declined RFCs: Nullable Intersection Types (12 to 26, August 2021) https://wiki.php.net/rfc/nullable_intersection_types ; Short Closures 2.0 (27 to 16, declined, July 2022) https://wiki.php.net/rfc/auto-capture-closure ; Asymmetric Visibility v1 (14 to 12, January 2023) https://wiki.php.net/rfc/asymmetric-visibility and v2 (24 to 7, August 2024) https://wiki.php.net/rfc/asymmetric-visibility-v2 ; Nested Classes (2 to 20, May 2025) https://wiki.php.net/rfc/short-and-inner-classes
- RFC counts per version (31 for 8.4, 19 for 8.5, 29 for 8.6 as of September 2026): php.watch RFC index, a secondary source. https://php.watch/rfcs
- Release dates of every version: php.net release archive and changelogs, checked 17 September 2026. https://www.php.net/releases/ and https://www.php.net/ChangeLog-8.php
- Support policy and current windows: php.net, supported versions and end of life pages, checked 17 September 2026. https://www.php.net/supported-versions.php and https://www.php.net/eol.php
- Release cycle update (security support extended, windows aligned to 31 December; votes 31 to 0 and 33 to 0): RFC, wiki.php.net, accepted April 2024. https://wiki.php.net/rfc/release_cycle_update
- PHP 8.6 schedule (GA 19 November 2026, feature freeze 22 September 2026): wiki.php.net, PHP 8.6 release schedule. https://wiki.php.net/todo/php86
- php-src activity (5,316 commits and 171 authors on master in the twelve months to 17 September 2026; 1,644 contributors): measured on a clone of github.com/php/php-src with `git log`, and the GitHub contributors API, 17 September 2026.
- The PHP Foundation founding (22 November 2021, ten founding members, the departure that triggered it): JetBrains, 22 November 2021, and thephp.foundation.

https://blog.jetbrains.com/phpstorm/2021/11/the-php-foundation/ and https://thephp.foundation/

- Transparency reports: 2022 (22 November 2022) https://thephp.foundation/blog/2022/11/22/transparency-and-impact-report-2022/ ; 2023 (26 February 2024) https://thephp.foundation/blog/2024/02/26/transparency-and-impact-report-2023/ ; 2024 (31 March 2025) https://thephp.foundation/blog/2025/03/31/transparency-and-impact-report-2024/ ; 2025 (27 May 2026) https://thephp.foundation/blog/2026/05/27/impact-and-transparency-report-2025/
- Structure (thirteen engineers, executive director, board of ten): thephp.foundation/structure, checked 17 September 2026. https://thephp.foundation/structure/
- All-time contributions by sponsor (Automattic 762,500 dollars, JetBrains 476,670, total 3,368,941): opencollective.com/phpfoundation, read 17 September 2026. https://opencollective.com/phpfoundation
- Sovereign Tech Agency grants (205,000 euros in 2023; 223,680 euros in 2025): sovereign.tech. https://www.sovereign.tech/tech/php and https://www.sovereign.tech/tech/php-2025
- Alpha-Omega grant and the Ecosystem Security Team: The PHP Foundation, 18 May 2026. https://thephp.foundation/blog/2026/05/18/announcing-ecosystem-security-team/
- Security policy and classification: github.com/php/php-src security policy and github.com/php/policies. https://github.com/php/php-src/security/policy and https://github.com/php/policies/blob/main/security-classification.rst
- External audit results (27 issues, 17 with security implications, 3 high): The PHP Foundation, 10 April 2025. https://thephp.foundation/blog/2025/04/10/php-core-security-audit-results/
- CVE counts by publication year (8, 7, 18, 13, 14): NVD API, product `cpe:2.3:a:php:php`, queried 17 September 2026. https://nvd.nist.gov/ GitHub advisories for php-src: https://github.com/php/php-src/security/advisories

Cost of Ownership

- Stack Overflow 2025 usage (JavaScript 68.8 percent, Python 54.8, TypeScript 48.8, C# 29.9, Java 29.6, PHP 19.1, Go 17.4, Rust 14.5, Ruby 6.9 among professional developers) and “admired” figures (Rust 72.4, TypeScript 58.0, Go 56.5, Python 56.4, C# 55.8, JavaScript 46.8, Ruby 44.3, Java 41.8, PHP 38.9): Stack Overflow Developer Survey 2025. https://survey.stackoverflow.co/2025/technology
- Median salary by language (PHP 49,586 dollars, second lowest of fifty): Stack Overflow Developer Survey 2024, “Top paying technologies”. https://survey.stackoverflow.co/2024/technology#top-paying-technologies
- 613 permanent UK job advertisements citing PHP in the six months to 17 September 2026, 404 a year earlier, median 48,676 pounds: IT Jobs Watch, a UK aggregator, secondary. https://www.itjobswatch.co.uk/jobs/uk/php.do
- Badoo one million dollars saved: Bumble Tech, 7 February 2017, as in Footprint.

- Wikimedia upgrade cadence: PHP 7.4 to 8.1 announced on wikitech-l on 17 March 2025 and completed that month; 8.1 to 8.3 completed 25 November 2025 (Phabricator T360995); 8.5 planned for late 2026 (Phabricator T413223).
- PHP versions by population: Packagist php-statistics, August 2026 https://packagist.org/php-statistics ; wordpress.org statistics API, read 17 September 2026 https://api.wordpress.org/stats/php/1.0/ ; W3Techs PHP versions, 17 September 2026 https://w3techs.com/technologies/details/pl-php
- End of security support for PHP 7.4 (28 November 2022) and PHP 5.6 (31 December 2018): php.net end of life page. https://www.php.net/eol.php

Where PHP Is the Wrong Choice

- All figures are repeated from the chapters above and carry the same sources.

Not used, and why

Figures that circulate about PHP and that I left out because no primary source could be found, or because the primary source contradicts them:

- “Spotify serves 600,000 requests per second with Symfony”: a global traffic figure repeated by agency blogs; the only primary source is a 2015 talk about the spotify.com website.
- “Etsy moved from HHVM back to PHP 7 in 2019 or 2020”: Etsy was on PHP 7 in 2016; HHVM was only ever used on its API cluster.
- “Laravel is used by OpenAI, Apple, Nike, NASA”: a claim on laravel.com without a source per company.
- “Adobe Commerce processed 6.2 billion dollars on Black Friday 2024”: found only on third-party aggregators; Adobe’s own releases give figures for all United States e-commerce measured by Adobe Analytics.
- “Mercedes-Benz sponsors The PHP Foundation”: not on the foundation’s sponsor list.
- “PHP 8 is three times faster than PHP 7”: the figure is for synthetic benchmarks and comes from the JIT RFC; the same page says typical applications are on par with PHP 7.4.
- Vimeo’s “one million lines of PHP” (December 2020): the article is no longer online; Psalm’s origin at Vimeo is the only claim I keep.
- Any figure on the number of PHP developers worldwide: the last primary figure found is 7.3 million in the third quarter of 2021 (SlashData), and nothing more recent could be traced to a primary page.

B - PHP Versions, 2015 to 2026

Each version has a section below, with its release date, its support window and its headline changes. The dates come from php.net. Active support means bug fixes and security fixes, and security support means security fixes only. Since the release cycle update voted in 2024, both windows end on 31 December of their final year.

Version	Released	Active support until	Security support until
PHP 5.6	28 Aug 2014	19 Jan 2017	31 Dec 2018
PHP 7.0	3 Dec 2015	3 Dec 2017	10 Jan 2019
PHP 7.1	1 Dec 2016	1 Dec 2018	1 Dec 2019
PHP 7.2	30 Nov 2017	30 Nov 2019	30 Nov 2020
PHP 7.3	6 Dec 2018	6 Dec 2020	6 Dec 2021
PHP 7.4	28 Nov 2019	28 Nov 2021	28 Nov 2022
PHP 8.0	26 Nov 2020	26 Nov 2022	26 Nov 2023
PHP 8.1	25 Nov 2021	25 Nov 2023	31 Dec 2025
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

The pattern to retain from the table is eleven consecutive yearly releases, each between 20 November and 8 December. The schedule of the next one is published on wiki.php.net months in advance, with its feature freeze and release candidate dates.

PHP 7.0, December 2015

- New engine, with a large reduction in memory use and, on the interpreter's own work, about twice the speed of PHP 5.6; see Throughput and Latency for the vendor claim and the independent synthetic measurement, with their caveats.
- Scalar type declarations (`int`, `float`, `string`, `bool`) for parameters, and return type declarations.

- `declare(strict_types=1)`.
- Null coalescing operator `??`, spaceship operator `<=>`.
- Anonymous classes.
- Engine errors became exceptions (`Error` hierarchy), so a fatal error can be caught.
- Removal of the `mysql_*` functions, `ereg_*` functions and PHP 4 style constructors.

PHP 7.1 to 7.4, 2016 to 2019

- 7.1: nullable types (`?int`), `void` return type, `iterable`, class constant visibility.
- 7.2: `object` type, Argon2 password hashing, Libsodium in the core.
- 7.3: flexible heredoc syntax, `is_countable()`, JSON errors as exceptions.
- 7.4: typed properties, arrow functions (`fn`), OPcache preloading, the foreign function interface (FFI), the `??=` operator, covariant returns and contravariant parameters.

PHP 8.0, November 2020

- Named arguments: `str_pad(string: 'a', length: 3)`.
- Attributes: `#[Route('/home')]` as structured metadata read by reflection.
- Constructor property promotion: `public function __construct(private int $x) {}`.
- Union types: `int|string $id`.
- `match` expression: strict comparison, no fallthrough, exhaustive.
- Nullsafe operator: `$user?->address?->city`.
- `mixed` and `static` as types.
- `throw` as an expression.
- `str_contains()`, `str_starts_with()`, `str_ends_with()`.
- `Stringable` interface, `WeakMap`.
- JIT compiler, inside OPcache.
- Saner string-to-number comparisons: `0 == 'foo'` is `false`.
- Internal functions throw `TypeError` and `ValueError` instead of warning and returning `null` or `false`.

PHP 8.1, November 2021

- Enums, pure and backed.
- `readonly` properties.
- First-class callable syntax: `strlen(...)`.

- Fibers: stackful coroutines, the building block of async libraries.
- `new` in initializers.
- Pure intersection types: `Countable&Traversable`.
- `never` return type, `final` class constants, `array_is_list()`, explicit octal notation.

PHP 8.2, December 2022

- `readonly` classes.
- Disjunctive normal form types: `(A&B)|null`.
- Standalone `true`, `false` and `null` types.
- Dynamic properties deprecated; `#[\AllowDynamicProperties]` opts a class back in.
- `#[\SensitiveParameter]` to redact an argument from stack traces.
- Constants in traits, `Random\Randomizer`.

PHP 8.3, November 2023

- Typed class constants.
- `#[\Override]` attribute: the engine checks that a parent method exists.
- `json_validate()`.
- Dynamic class constant fetch: `Foo::{$name}`.
- `readonly` properties can be reinitialised inside `__clone()`.

PHP 8.4, November 2024

- Property hooks: `get` and `set` logic declared on the property itself.
- Asymmetric visibility: `public private(set) int $x`.
- `new Foo()->bar()` without wrapping parentheses.
- Lazy objects through reflection (`newLazyGhost()`, `newLazyProxy()`).
- `#[\Deprecated]` attribute.
- `array_find()`, `array_find_key()`, `array_any()`, `array_all()`.
- `mb_trim()` and friends, `mb_ucfirst()`, `mb_lcfirst()`.
- New DOM extension with an HTML5 parser (`Dom\HTMLDocument`).
- BCMath object API (`BcMath\Number`), PDO driver subclasses (`Pdo\Sqlite`, `Pdo\Mysql`, `Pdo\Pgsql`).
- Implicitly nullable parameter types deprecated.

PHP 8.5, November 2025

- Pipe operator: `$value |> trim(...) |> strtoupper(...)`.
- `clone` with property updates: `clone($obj, ['prop' => $value])`.
- `#[\NoDiscard]` attribute, with the `(void)` cast to silence it deliberately.
- `array_first()`, `array_last()`.
- Closures and first-class callables in constant expressions (attribute arguments, default values, constants).
- Attributes on constants.
- New `uri` extension (`Uri\Rfc3986\Uri`, `Uri\WhatWg\Url`).
- Fatal errors include a backtrace.
- `get_error_handler()`, `get_exception_handler()`.
- The backtick shell-execution operator is deprecated.

C - Glossary

The terms I use that you may not know if you come from another ecosystem, in alphabetical order.

Active support. The first two years of a PHP release branch, during which bugs and security issues are fixed in monthly point releases. Two years of security support follow, with security fixes only. See Governance and Longevity.

Composer. The dependency manager of the PHP ecosystem, comparable to npm, pip or Maven. It reads `composer.json`, resolves the versions, writes `composer.lock` and generates the autoloader that maps class names to files.

Fiber. A stackful coroutine, added in PHP 8.1: a function that can suspend itself and be resumed later by whoever holds it. The language provides the mechanism only, and scheduling belongs to libraries. See Concurrency.

FrankenPHP. An application server for PHP, written in Go on top of the Caddy web server. It runs PHP either in the classic one-process-per-request mode or in a worker mode that keeps the application booted between requests. See The Runtime.

Hack and HHVM. Hack is a language that forked from PHP at Facebook in 2014, and HHVM is its virtual machine. HHVM dropped support for PHP itself in 2019. Organisations that run Hack do not run PHP, and I do not count them as PHP users.

JIT. The just-in-time compiler that has shipped inside OPcache since PHP 8.0, compiling hot code paths to machine code. It benefits CPU-bound code far more than typical web requests, and I say so wherever a JIT figure appears.

NTS and ZTS. Non-thread-safe and Zend-thread-safe builds of the interpreter. The standard build is NTS, because the process model never needed threads. ZTS builds exist for embedding and for the `parallel` extension.

OPcache. The extension that keeps the compiled form of every PHP file in shared memory, so that a file is parsed and compiled once rather than on every request. It has been standard in production since PHP 5.5.

p50, p95, p99. Percentiles of a latency distribution: the response time under which 50, 95 or 99 percent of requests complete. A benchmark that reports only the average hides the tail, which is why I prefer sources that publish percentiles.

Packagist. The public package registry Composer uses by default, comparable to npmjs.com or PyPI.

PER Coding Style. The coding style standard maintained by the PHP-FIG and the successor of PSR-12. Tools such as PHP-CS-Fixer and PHP_CodeSniffer enforce it.

PHP-FIG. The PHP Framework Interoperability Group, which publishes the PSR standards so that libraries from different authors fit together.

PHP-FPM. The FastCGI Process Manager, the standard way to run PHP behind a web server. It keeps a pool of worker processes, each handling one request at a time.

Preloading. An OPcache feature (PHP 7.4) that compiles a list of files once at startup and keeps them linked in memory, so that no class needs autoloading during a request.

PSR. PHP Standards Recommendation, a numbered interoperability standard published by the PHP-FIG: PSR-4 for autoloading, PSR-3 for logging, PSR-7 for HTTP messages, and so on.

Rector. A tool that rewrites PHP code automatically: it upgrades syntax from one version to the next, applies refactorings and removes deprecated calls. Teams use it for large-scale version upgrades.

RFC. Request for Comments, the document through which any change to the language is proposed, discussed on the internals mailing list and voted on. A change to the language requires a two-thirds majority. See Governance and Longevity.

RoadRunner. An application server for PHP written in Go. It keeps worker processes alive and passes requests to them over a protocol, so that the application is booted once rather than on every request.

Shared-nothing. The execution model in which each request starts with fresh memory, runs, responds and is discarded, sharing nothing with the previous or the next request. Most of PHP's operational properties follow from it. See The Runtime.

Static analysis. Reading code without running it, to find type errors and other defects. PHPStan and Psalm are the two analysers of the PHP ecosystem, and both understand a docblock type syntax richer than the language's own.

Swoole and OpenSwoole. A C extension and its fork. Each gives PHP an event loop, coroutines and a built-in HTTP server, for workloads that need many concurrent connections in one process.

TechEmpower Framework Benchmarks. A public, continuously run benchmark suite that compares hundreds of web frameworks across languages on a fixed set of tests (JSON serialisation, single and multiple database queries, "fortunes" HTML rendering, plaintext). I cite it with its round number, its hardware and the test, and I say what each test measures.

W3Techs. A web survey company that publishes the usage of server-side technologies across a sample of more than twenty million websites. Its figures count only the sites whose language can be detected, which biases them in ways I state when I use them.

Worker mode. A way of running PHP in which the application is booted once and kept in memory, and each worker process handles many requests in sequence. FrankenPHP and RoadRunner offer it, among others. It removes the per-request startup cost at the price of the shared-nothing guarantee.