Performance

Runtime inspection costs, discovery caching, and measurements with many small workers

Anonymous · 15/11/2026

Performance

Dispatch uses runtime inspection of the selected worker. The number of workers in the registry does not cause a scan of every worker on each call. The relevant costs are the selected worker's public methods, its parameter types, the active policy, and the number of calls.

Work performed during dispatch

`DirectDispatcher` resolves the package, component, and worker from the registry, then asks `Inspector::getOperationMethod()` to resolve the external operation name. `Inspector` examines the selected worker's public methods and their `#[Operation]` attributes, including inherited public methods, to build the external-name index and detect collisions.

The dispatcher evaluates the operation policy before resolving arguments. `TaggedOperationPolicy` asks the inspector whether the operation carries `#[Operation]`, which resolves the external name again and repeats the public-method inspection. Other policies can perform different work.

`Resolver` obtains the resolved PHP method's parameter metadata, validates the supplied values, and casts or deserializes them before invoking the method. Parameter inspection uses reflection on every call. It also reads PHPDoc when a parameter is natively typed `array`, to resolve element types such as `Item[]`. Parameters without a native `array` type skip that PHPDoc processing. `Caster` also uses reflection to identify the backing type of backed enums.

The dispatcher tiers add their own work: `TypedDispatcher` collects execution metadata, and `SafeDispatcher` serializes the result and converts failures to structured output. Business logic, deserialization, serialization, and transport costs contribute to the total time observed by a caller.

Discovery and caching

Discovery reads documentation and can traverse multiple workers. A complete tree therefore has a different cost profile from dispatching one operation: adding workers increases the amount of metadata a full traversal must retrieve and process.

`CachedInspector` stores the results of `getClassDoc()` and `getPublicMethods()` in an application-provided PSR-6 pool. It forwards `getOperationMethod()`, `isOperation()`, `hasOperationAttribute()`,

and `getOperationParameters()` directly to the wrapped inspector. Configuring this cache does not cache operation execution metadata or remove per-dispatch reflection. Cached documentation also requires cache access and validation; policy checks and traversal still run.

See Discovery for cache configuration. Measure the backend and discovery workload that the application actually uses when evaluating its benefit.

Measurements with many small workers

The reference benchmark uses 100, 500, and 1,000 distinct worker classes with 3, 4, or 5 operation methods per class. Each operation receives `int $a` and an optional `int $b = 10`, and returns their sum plus the operation index. Alternating methods declare an explicit external name through `#[Operation(name: ...)]`.

The environment is PHP 8.5.12 CLI on an Apple M1 Pro, arm64, with Xdebug disabled, CLI OPcache enabled, and JIT disabled. Each variant and call pattern uses 10,000 warmup calls followed by 11 rounds of 20,000 calls. Tables report medians in microseconds per call. The timed loops include schedule selection and loop overhead.

The distributed pattern cycles through a precomputed, fixed-seed shuffled schedule containing every worker/operation pair. The same-worker pattern varies operations on one worker. Each variant runs in a separate PHP process, and the order of variants and timed patterns is randomized. Workers are constructed outside the timed regions. Names, attribute presence, and parameter metadata are checked for every worker/operation pair; full-dispatch results are checked as well.

Three variants are compared:

- **Runtime inspection:** the package's `Inspector`.
- **Compiled names:** an experimental compiler pass generates external-name mappings and attribute-presence data and injects them through a PHP container generated by Symfony's `PhpDumper`. Parameter inspection still runs at dispatch time.
- **Compiled names and parameters:** the same experimental approach also includes the scalar parameter metadata of the generated operations.

The compiler pass reads class definitions without constructing workers. The compiled variants are benchmark prototypes; the package's supplied services use runtime inspection. They do not represent an available compiled-registry configuration.

Classes with exactly 3-5 public methods

This isolates the execution-metadata calls: resolving the operation name, checking its attribute, and obtaining its parameters. These classes have exactly 3, 4, or 5 public methods in total. They do not implement `WorkerInterface`, whose infrastructure requires additional methods. This table measures inspection, not a complete dispatch.

Classes	Public methods per class	Runtime inspection	Compiled names	Compiled names and parameters
100	3	4.91	1.28	0.26
100	4	6.64	1.24	0.27
100	5	7.10	1.28	0.27
500	3	5.48	1.36	0.44
500	4	6.09	1.37	0.31
500	5	7.24	1.35	0.31
1,000	3	5.20	1.28	0.30
1,000	4	6.19	1.39	0.34
1,000	5	7.03	1.37	0.31

Complete direct dispatch

This uses the real `DirectDispatcher`, `TaggedOperationPolicy`, `Resolver`, `Caster`, and `Validator`. Each worker has 3-5 operation methods plus 14 inherited public infrastructure methods supplied by the benchmark base class and Backbone/Config traits, for totals of 17-19 public methods. The inspector examines those inherited methods too.

Workers	Operations per worker	Runtime inspection	Compiled names	Compiled names and parameters
100	3	14.27	2.86	1.77
100	4	15.60	2.79	1.86
100	5	16.72	2.82	1.82
500	3	15.41	3.03	2.09
500	4	15.80	2.92	2.15
500	5	16.96	3.04	2.02
1,000	3	14.90	3.01	1.89
1,000	4	15.54	3.02	2.54
1,000	5	17.13	3.29	2.24

Both tables use distributed calls. With calls to the same worker, complete dispatch ranges from 14.21 to 16.74 microseconds with runtime inspection and from 1.70 to 1.85 microseconds with compiled names and parameters.

Across these scenarios, increasing the worker count from 100 to 1,000 does not multiply the per-call

cost by ten. Compiled names and parameters reduce the measured complete-dispatch cost by approximately 6-9 times, saving about 12-15 microseconds per call for this scalar workload. That ratio describes dispatcher overhead, not the speedup of an application or its business operations.

Compilation, loading, and memory

Compiled metadata also has preparation and storage costs. For the full-dispatch prototype with 1,000 workers and 5 operations per worker:

Measurement	Runtime inspection	Compiled names	Compiled names and parameters
Register definitions, compile, and generate PHP	16.90 ms	105.23 ms	231.88 ms
Load generated container in a fresh CLI process	2.04 ms	15.95 ms	31.91 ms
Additional PHP heap plus OPcache allocation on load	0.24 MiB	2.84 MiB	6.24 MiB

Compilation values are medians of five rounds in a reduced benchmark container, with compilation infrastructure autoloaded. They exclude fixture generation, file writing, and worker construction. They do not measure a complete application build.

Loading and memory values are medians of seven fresh CLI processes per variant, with worker classes and common support code already loaded. Loading includes requiring the generated container and obtaining the inspector. `opcache.file_update_protection=0` ensures consistent treatment of generated files. These CLI processes do not share a persistent warm OPcache.

Memory is the measured increase in PHP heap plus OPcache allocation, not resident process memory or private memory per request. A server can share OPcache between requests. The cold CLI loading times do not predict loading with a warm shared OPcache, and the benchmark does not establish that a compiled registry speeds up the first operation in a request or command.

When to consider compiled metadata

Runtime inspection keeps execution metadata tied to the selected worker without requiring a compiled operation registry. A large worker registry alone is insufficient evidence that per-call inspection is a bottleneck. Repeated calls accumulate inspection work, and workers with more public methods or parameters requiring PHPDoc have different costs.

Consider a compiled metadata registry when profiling shows operation inspection contributes a significant share of the application’s processing time. Evaluate absolute time saved, invocation volume, compilation and loading costs, and memory usage together. A microbenchmark speedup alone does not establish a worthwhile application-level improvement.

These measurements cover scalar operations and direct dispatch. They exclude business workloads, full discovery, HTTP or Python transport, lazy initialization, and array, object, union, or enum parameter handling. They do not establish the cost or behavior of compiling those additional cases.

Last updated on 15/11/2026