1v0 Ship v1.0 Faster

# Laravel Horizon vs Queue Workers: Which One Should You Use?

Laravel Horizon vs Queue Workers: Which One Should You Use?

Laravel gives you two main ways to process background jobs: the classic `queue:work` workers and the Horizon dashboard/manager for Redis queues. Both run the same jobs and use the same queue system—but they differ in *operability*, *visibility*, and *auto-scaling* features. In this guide, you'll learn their trade-offs, when to pick one over the other, how to configure each in production, and how to migrate smoothly between them.

## 1 – The Short Answer

- **Use `queue:work`** if you want maximum simplicity, minimal dependencies, or you're not on Redis yet.
- **Use Horizon** if you're on Redis and need auto-balancing, per-queue scaling, live metrics, failure insights, and an operations-friendly dashboard.

If your app is growing or is already in production with meaningful throughput, Horizon usually pays for itself in visibility and control. For the fundamentals of queues, read Article #42 – Queues. For the Horizon dashboard and features, see Article #45 – Horizon.

## 2 – Classic Workers with `queue:work`

Classic workers are just long-running PHP processes that pull jobs from your configured backend (database, Redis, SQS, etc.). They're easy to run and script with systemd or Supervisor.

```bash
# Run a worker on the default connection
php artisan queue:work --queue=default --sleep=1 --tries=3

# Run multiple workers for throughput
php artisan queue:work --queue=emails --sleep=1 --tries=3
php artisan queue:work --queue=reports --sleep=1 --tries=3
```
Code language: Bash (bash)

The command pulls jobs from the specified queue list and processes them. Flags like `--sleep` control polling behavior; `--tries` sets the automatic retry count before failure.

```ini
; /etc/systemd/system/laravel-queue@.service
[Unit]
Description=Laravel Queue Worker for %i
After=network.target

[Service]
User=www-data
Restart=always
RestartSec=3
WorkingDirectory=/var/www/current
ExecStart=/usr/bin/php artisan queue:work --queue=%i --sleep=1 --tries=3
ExecStop=/bin/kill -s SIGTERM $MAINPID

[Install]
WantedBy=multi-user.target
```
Code language: TOML, also INI (ini)

This systemd template lets you run one service per queue (e.g., `laravel-queue@emails`, `laravel-queue@reports`). It automatically restarts workers, making basic operations simple without a dashboard.

## 3 – Horizon: Managed, Observable Redis Workers

Horizon is a layer on top of Redis queues that adds a web UI, auto-balancing, per-queue processes, tags, batches, and real-time metrics.

```bash
composer require laravel/horizon
php artisan horizon:install
php artisan migrate
php artisan horizon
```
Code language: Bash (bash)

Installing Horizon publishes config and migrations for job monitoring. Running `php artisan horizon` starts the Horizon master, workers, and a dashboard available at `/horizon`. See our full walkthrough in Article #45.

```php
// config/horizon.php (snippet)
'supervisors' => [
    'app-supervisor' => [
        'connection' => 'redis',
        'queue' => ['default', 'emails', 'reports'],
        'balance' => 'auto',
        'minProcesses' => 2,
        'maxProcesses' => 12,
        'tries' => 3,
    ],
],
```
Code language: PHP (php)

Supervisors define how many worker processes run per queue group. With `balance: auto`, Horizon dynamically shifts processes to hot queues, improving throughput during spikes.

```
; /etc/systemd/system/horizon.service
[Unit]
Description=Laravel Horizon
```

```
After=network.target

[Service]
User=www-data
WorkingDirectory=/var/www/current
ExecStart=/usr/bin/php artisan horizon
Restart=always
RestartSec=3

[Install]
WantedBy=multi-user.target
```
Code language: TOML, also INI (ini)

Running Horizon under systemd makes it resilient across deploys and reboots. It will automatically reload workers when your code changes, reducing manual restarts compared to classic workers.

## 4 - Side-by-Side Comparison

- **Backends**: `queue:work` supports Database/Redis/SQS/etc. Horizon focuses on **Redis**.
- **Visibility**: Classic workers = logs only. Horizon = live dashboard (throughput, latency, failures, retries, tags, batches).
- **Scaling**: Classic = manual process counts. Horizon = per-queue supervisors, `min/maxProcesses`, auto-balancing.
- **Ops**: Classic = simplest, fewer moving parts. Horizon = more features, easier on-call debugging.
- **Cost**: Horizon adds Redis dependency & instance size considerations, but saves ops time under load.
- **Security**: Protect `/horizon` via auth/gates; in classic mode there's no dashboard to secure.

For high-traffic environments, the *observability and auto-scaling* that Horizon brings typically outweigh the extra moving parts. Pair Horizon with Redis and caching strategies from Article #43 for best results.

## 5 - Job Tagging & Batches (Horizon Extras)

Horizon can group and filter jobs by tags and show batch progress. Tagging makes debugging easier during incidents.

```php
// app/Jobs/SendWelcomeEmail.php (snippet)
public function tags(): array
{
    return ['user:'.$this->user->id, 'emails'];
}
```
Code language: PHP (php)

These tags let you filter the Horizon dashboard to see only jobs for a specific user or category—handy for troubleshooting failures and replays.

```php
// Dispatch a batch with progress tracking
use Illuminate\Bus\Batch;
use Illuminate\Support\Facades\Bus;

$batch = Bus::batch([
    new ImportRow($file, 1),
    new ImportRow($file, 2),
    // ...
])->then(fn (Batch $batch) => logger('Import complete'))
 ->catch(fn (Batch $batch, Throwable $e) => logger('Import failed'))
 ->name('Customer Import')
 ->dispatch();
```
Code language: PHP (php)

Batches group multiple jobs and show progress/failures in Horizon. This is invaluable for long-running imports where you need visibility and retry control at scale.

## 6 – A Tiny Admin UI Toggle (Optional)

Here’s a minimal Blade UI to *simulate* switching strategies (note: in reality you enable one or the other at deploy time). It’s useful for documenting your operations runbook.

```
<!-- resources/views/admin/queues.blade.php -->
@extends('layouts.app')

@section('content')
<div class="container">
  <h1 class="mb-4">Queue Strategy</h1>

  <div class="card mb-3">
    <div class="card-body">
      <p class="mb-3">Current: <strong>Horizon</strong> (Redis)</p>
      <a href="/horizon" class="btn btn-theme r-04">Open Horizon
Dashboard</a>
      <a href="{{ route('admin.docs.queue') }}" class="btn btn-
outline-secondary ms-2">Operations Runbook</a>
    </div>
  </div>

  <p class="text-muted">Switching between Horizon and classic workers
is usually done during deployment with systemd services.</p>
</div>
@endsection
```
Code language: HTML, XML (xml)

This page links operators to Horizon and your internal docs. Treat “switching” as a deployment concern (start/stop services), not a runtime toggle.

## 7 – Deployment & Reliability Notes

- **Supervise processes**: Use systemd/Supervisor for both Horizon and classic workers so they auto-restart.
- **Environment parity**: Keep dev/staging using the same backend (Redis) to catch scaling issues early.
- **Back-pressure**: Use queue priorities (multiple queues) and Horizon supervisors to keep critical jobs flowing.
- **Observability**: Pair Horizon with Telescope for request/query insights; they complement each other.
- **High concurrency**: If request throughput is also high, consider Octane for the web tier and Horizon for workers.

For production checklists and CI, see Article #58 – Deployment Checklist and Article #54 – CI/CD.

## 8 – Migrating from Classic Workers to Horizon

- Ensure `QUEUE_CONNECTION=redis` and Redis is sized properly.
- Install/configure Horizon supervisors with sane `min/maxProcesses`.
- Roll a canary: point a subset of queues to Horizon, watch metrics, then cut over fully.
- Tag key jobs and use batches where visibility helps during the transition.
- Decommission old `queue:work` services after verifying parity.

This staged approach reduces risk while giving you immediate operational benefits—especially the dashboard and auto-balancing during traffic spikes.

## Wrapping Up

**Classic workers** are simple and effective for small to medium workloads or non-Redis backends. **Horizon** shines when you need operational visibility, auto-balancing, tags/batches, and a production-grade dashboard—all on Redis. Many teams start with classic workers and graduate to Horizon as load and complexity grow. Choose based on your current scale, team needs, and infrastructure.

## What's Next

- How to Use Laravel Queues for Faster Performance — foundations, retries, chaining, and delays.
- How to Use Laravel Horizon for Queue Monitoring — supervisors, balancing, alerts, and dashboard.
- 10 Proven Ways to Optimize Laravel for High Traffic — pair queues with caching, indexing, and Octane.