upsun
Formerly Platform.sh

shopware

The definitive guide to Shopware performance on Shopware PaaS

Document information

Title	The definitive guide to Shopware performance on Shopware PaaS
Authors	• Vincenzo Russo • Vincent Robert
Reviewers	• Gauthier Garnier
Publishers	Upsun
Version	v1.1 (September 2025)
Last revised	30 September 2025
License	© 2025 Upsun. All rights reserved.
Keywords	Shopware, Upsun, PaaS, performance, caching, Blackfire, profiling
Contact	fabian.meissner@upsun.com

Table of contents

Executive summary

This white paper shares the results of a rigorous load testing campaign on Shopware PaaS, a managed e-commerce platform built on Upsun and tailored specifically to run Shopware at scale. But beyond benchmark data, this document is a practical guide: it distills the lessons we've learned about optimizing Shopware performance, identifying bottlenecks, and running real-world storefronts sustainably.

The goal of these tests was to explore how Shopware behaves under real-world traffic patterns, and to surface actionable insights for building fast, resilient storefronts. Our findings confirm that the true bottlenecks often lie not in the infrastructure, but in how Shopware is configured, extended, and deployed. In short: Shopware is a demanding software, and running it well requires a deliberate, performance-first approach.

To that end, this white paper does two things:

1. It **quantifies** how Shopware PaaS performs under load across multiple infrastructure configurations, offering real numbers on response time, order throughput, CPU saturation, and failure rates.
2. It **teaches** technical practitioners—developers, architects, and DevOps teams—how to optimize Shopware itself: from caching strategy and container sizing to load test planning, cache prewarming, and using tools like Blackfire to diagnose and fix application-level inefficiencies.

Throughout, we emphasize key architectural truths:

- **"Never blindly trust your code** (or someone else's)**."** Extensions are rarely optimized for your exact workload.
- **Cache everything.** Backend calls, dynamic queries, and per-customer customisations are expensive.
- **Test, measure, and plan for real traffic**. Assumptions cost money; data saves it.

Shopware PaaS exists to give teams the best possible foundation: a Upsun- native environment with CI/CD, staging, observability, and elastic scale built in.

But infrastructure is only half the story. The rest depends on what you run—and how you run it.

This paper is for those who want to run Shopware right.

Glossary

This glossary explains key terms and acronyms used throughout the paper. It is intended as a quick reference for technical readers who want clarity on performance metrics, infrastructure concepts, and platform terminology.

Term	Definition
VU	*Virtual User* – A simulated user in load testing tools like K6.
TTFB	*Time to First Byte* – The time it takes to receive the first byte of a server response. A key performance indicator for backend and network responsiveness.
p95	*95th Percentile* – A statistical measure showing that 95% of response times were faster than this value. Used to assess worst-case performance under normal conditions.
ERP	*Enterprise Resource Planning* – Software systems for managing business operations like product data, stock, pricing, and orders.
CDN	*Content Delivery Network* – A network of servers (e.g., Fastly) that deliver cached content closer to users, reducing latency.
CI/CD	*Continuous Integration / Continuous Deployment* – Practices for automating code testing, integration, and delivery to production.

Who should read this paper and why

This white paper is written for technical professionals working with Shopware—whether you're building, scaling, or maintaining storefronts based on Shopware 6. If you're a **developer, lead developer, software architect, or DevOps engineer,** this document is for you.

Shopware itself is a flexible and powerful platform. But as many experienced teams know, achieving great performance in production often depends less on Shopware core and more on how the project is structured, extended, and deployed. That's where this paper aims to help.

Common challenges we've seen

Across dozens of Shopware projects, several recurring issues surface—regardless of the underlying infrastructure:

- Plugin complexity and fragility
  Custom or third-party plugins frequently become the weak link. They're often poorly optimized, hard to debug, and difficult to support—especially under load.
- Poor cache invalidation behavior
  ERP updates, imports, or admin changes can trigger mass cache purges, causing backend spikes and latency bursts. Without careful planning or use of soft purges, this becomes a recurring pain.
- Composable frontends and uncacheable APIs
  Shopware's shift toward headless architectures has outpaced its native caching model. The Store API, relying heavily on POST requests, isn't best suited for HTTP caching through tools like Fastly or Varnish. Workarounds exist, but are not always robust or standardized.
- Lack of DevOps standardization
  Teams often reinvent the wheel when it comes to deployment, monitoring, staging, and rollback workflows—wasting precious engineering hours on infrastructure instead of product.

These issues are not due to bad intentions—they stem from complexity. Shopware is modular, customizable, and capable, but without discipline and the right tools, performance suffers. **Shopware PaaS was designed to change that.**

Why this paper exists

Built on Upsun, Shopware PaaS offers:

- Enterprise-grade infrastructure with scaling, HA, and performance SLAs.
- Standardized developer workflows: Git-based environments, CI/CD, and rollback built in.
- Tools for diagnosis and performance tuning, including native support for Blackfire.
- Best-practice templates and architectural guidance for building stores that perform.

This paper combines data, experience, and opinionated guidance to help you:

- Avoid common mistakes.
- Optimize your deployments.
- Better understand the trade-offs behind infrastructure, plugins, caching, and API usage.

If you're asking questions like...

- How do we avoid breaking performance when adding new features?
- Which infrastructure tier do we really need?
- How do we use Blackfire to improve performance?
- What can we cache, and what should we avoid invalidating?
- How do we prepare our Shopware store for real-world scale?

...then you’re exactly who this paper is for.

Read on to explore the infrastructure options, testing methodology, and proven best practices that will help you run Shopware right.

How Shopware really performs (and why it matters)

Shopware, like any complex web application—and especially as an e-commerce platform—can be fast or painfully slow. Performance is not a fixed trait of the system; it depends entirely on how it's configured, extended, and maintained. The difference between a snappy storefront and an unresponsive one often comes down to decisions made during development and integration.

Performance is a variable

A recent Shopware 6 performance benchmark report by Tideways (Q1 2025) analyzed hundreds of real-life projects and found a **10× difference in Time-to-First-Byte (TTFB)** across similar page types. For example, product search pages—an essential element of any storefront—ranged from under 200ms to well over 2s depending on the project.

This wide disparity reflects what we see in practice: the software isn't inherently slow, but it is highly sensitive to how it's used. Plugins, logging levels, integration design, underlying components (e.g., OpenSearch), admin configuration—each can shift a site's performance profile dramatically.

Real-world example: debug-level logging

In late 2024, we received a support ticket about a Shopware storefront suffering from sudden, severe slowness. Some pages were taking over **10 seconds** to load. In extreme cases, requests exceeded **3 minutes**.

Using Blackfire's Alerting and Profiling capabilities, we quickly identified the root cause: the PayPal plugin was running with DEBUG-level logging enabled. Under heavy traffic, this configuration overwhelmed the disk with write operations, causing I/O contention as processes queued to write their own logs. The result was widespread performance degradation. Once debug mode was disabled, response times recovered almost immediately.

This case illustrates a simple but recurring theme: **a small misconfiguration in a third-party extension can destabilize the entire stack.**

Common pitfalls behind performance issues

Beyond logging, we routinely identify other causes of performance degradation:

- ERP integrations that trigger cache invalidations
  Inventory and product updates—especially when unbatched—cause sweeping invalidations in Shopware's cache layers, resulting in spikes in backend load and frontend latency.
- Excessive personalization

  Serving unique content—such as customer-specific pricing or postal code–based adjustments—can reduce cache efficiency and increase infrastructure load. Where possible, identify shared elements across visitor segments and cache them using reusable keys.
- Lack of performance budget
  Teams deploy features or plugins without defining acceptable performance thresholds, leading to slow creep over time.

In other words: **Shopware's performance is everyone's responsibility**. Every feature, every integration, every checkbox in the admin panel can have an outsized effect on what your customers experience.

Cache is king

No matter how efficient your custom code is, nothing beats a fast cache. Whether at the edge (Fastly), on the application side (Symfony HTTP cache), or in the database/query layer (Redis, Doctrine metadata, etc.), **caching is the primary strategy for sustainable performance.**

This is especially true given how Shopware handles dynamic content and API usage. The more often content can be served from cache, the less pressure on CPU, I/O, and application rendering. And the higher your chances of delivering the sub-second experience customers expect.

Dozens of studies correlate every second of added load time with a drop in conversion rate.[1] For Shopware stores competing in a crowded retail market, **performance isn't just a technical concern—it's a commercial one.**

Summary

Shopware can perform extremely well, but only if treated with care:

- Measure everything, especially when introducing plugins or integrations.
- Define performance budgets—don't assume "fast enough" is good enough.
- Use tools like Blackfire to catch regressions early.
- Design ERP and import flows for efficiency, and batch updates wherever possible.
- Cache everything you safely can.

The rest of this paper will show you how we quantify performance, choose infrastructure, and apply best practices to get the most out of Shopware in production.

Understanding the infrastructure options

Running Shopware efficiently requires more than good application code—it demands the right infrastructure behind it. Shopware PaaS is built on **Upsun**, which offers several infrastructure tiers tailored to different performance, scalability, and isolation requirements. This section explains how these infrastructure types work, what differentiates them, and how to choose the best option based on your workload.

[1] For example: SOASTA reported that a **1-second delay in mobile page load time can impact retail conversions by up to 20%**. Cloudflare found that pages loading in 2.4 seconds had a **1.9% conversion rate**, dropping to **0.6% at 5.7 seconds**. Portent showed that **every second counts**: a 1s load time yielded **~40% conversion**, falling to **~20% by 5s**.

Grid plans (XL, 2XL, 4XL)

The **Grid infrastructure** is a shared-resource model where your project runs inside isolated containers on virtual machines (VMs) shared with other customers. Each service—application, database, Redis, workers—is isolated into its own container, drawing from a **predefined CPU and RAM quota.**

- **Isolation**: Logical (container-based), but shares VM-level resources.
- **Best for**: Entry-level and small-to-medium projects.
- **Scaling**: Vertical only (upgrade to a larger plan).
- **Considerations**: Cost-effective, but performance predictability is limited under extreme load.

Dedicated Grid hosts (DGH-16, DGH-32, DGH-64, DGH-128)

The **DGH tier** maintains the container-based model but shifts from shared to **dedicated VMs**. All resources on the VM are exclusive to your project, enabling **burst capacity** and more consistent performance.

In addition to dedicated resources and full CPU availability, DGH plans offer further architectural advantages. All containers are co-located on the same host machine, eliminating inter-service network latency and improving communication efficiency— particularly beneficial for services like Redis that rely on frequent internal calls. Moreover, DGH plans provide significantly higher memory limits than standard Grid plans, making them ideal for workloads involving large datasets, memory-intensive caching, or demanding product catalogues.

- **Isolation**: Dedicated compute, shared storage and network.
- **Best for**: Mid-sized to high-traffic storefronts needing predictable CPU access.
- **Scaling**: Vertical scaling by moving to a higher DGH plan.
- **Performance tip**: Burstable containers can leverage full VM capacity under load, especially for peak-hour resilience.

Dedicated plans (D-12, D-24, D-48, D-96, D-192)

The **Dedicated tier** consists of three fully dedicated VMs configured for **high availability**. Services are distributed across nodes using Shopware-aware patterns:

- MySQL in multimaster cluster configuration.
- Redis with multi-master setup.
- Redundant application containers.
- **Isolation**: Fully dedicated VMs, including compute, network, and storage.
- **Best for**: Production environments with high concurrency and uptime demands.
- **Scaling**: Zero-downtime Vertical scaling via larger plan sizes.
- **Resilience**: Redundancy ensures no single point of failure.

Dedicated split clusters

Dedicated Split separates concerns explicitly: **three core nodes** handle backend services (DB, cache, queues), while **three or more web nodes** serve application traffic.

- **Isolation**: Maximum—dedicated compute, storage, and service separation.
- **Best for**: Enterprise-grade setups with high frontend concurrency or traffic spikes.
- **Scaling**: Core nodes scale vertically; web nodes scale both vertically and horizontally for flawless dynamic auto-scaling.
- **Advantage**: Allows for fine-grained tuning and load segregation between backend and frontend responsibilities.

Choosing the right option

Each infrastructure model comes with trade-offs. Here's how to think about them:

Scenario	Recommended option
Evaluation, MVP, or low-traffic site	Grid (XL–4XL)
Growing production store with peaks	DGH (DGH-16 to DGH-64)
High traffic, reliability-critical	Dedicated (D-24+)
Multi-AZ, heavy concurrency	Dedicated Split

Remember: infrastructure is only part of the performance equation. Even the best hosting can be undercut by inefficient plugins, slow custom code, or misconfigured caches. The sections that follow will walk through how to optimize those variables—regardless of which tier you're on.

Best practices for high-performance Shopware

Achieving reliable performance in Shopware is not a matter of chance, but of deliberate architecture, disciplined development, and consistent operational practices. While Shopware is capable of excellent performance, it requires active attention to application structure, infrastructure sizing, cache strategy, and runtime configuration.

This section outlines the principles and techniques that have proven most effective in sustaining performance across real-world Shopware PaaS projects.

Infrastructure and runtime configuration

One of the foundational elements of Shopware performance lies in how the application is sized and deployed. Upsun allows for precise control over memory allocation and CPU resources through the use of runtime hints. These should not be left at defaults. Instead, developers are encouraged to set memory usage expectations explicitly in `.platform.app.yaml`, enabling the platform to optimize PHP-FPM concurrency settings accordingly.

Maintaining CPU usage below 70% under load is advisable to preserve headroom during peak traffic events. Infrastructure tiers should be selected not only based on average load, but also with burst capacity and business seasonality in mind. Many performance issues arise not from insufficient resources per se, but from failure to anticipate the operational profile of the store under real-world conditions.

Caching strategy

Caching is a critical pillar of Shopware performance, particularly in high-traffic environments. It must be designed, not assumed. HTTP caching, application-side caching (such as Symfony's HTTP cache), and object caching (e.g. Redis) must work in concert.

Where possible, content should be made cacheable by design. This includes structuring endpoints to use `GET` rather than `POST` methods where appropriate, ensuring consistent cache headers, and avoiding patterns that lead to cache fragmentation (e.g. user-specific variations of product or category pages). Prewarming the Fastly cache of the most critical pages after deployments or content imports ensures that traffic is not served cold, avoiding costly backend rendering during critical periods.

It is also important to validate that the cache is functioning as intended. This can be done by inspecting HTTP response headers, particularly the X-Cache header provided by Fastly. A HIT value indicates that the response was served from cache, while a MISS suggests a cache bypass or expiration. Persistent or unexpected MISS responses should be investigated promptly, as they may signal misconfiguration, over-aggressive cache invalidation, or content that is not cacheable due to dynamic query parameters or incorrect headers.

Equally important is managing cache invalidation responsibly. Unbatched product updates or poorly scheduled ERP synchronisations can result in widespread cache purges, leading to backend saturation and degraded user experience. Where invalidation is necessary, techniques such as soft purges and deferred purging intervals can help mitigate their impact.

In real-world deployments, we have observed large Shopware installations generating hundreds of millions of cache invalidation requests per day. In such cases, the system spends a disproportionate amount of its resources regenerating and invalidating cache entries rather than serving users. This pattern not only

consumes infrastructure inefficiently, but also results in a noticeably degraded browsing experience. Reducing unnecessary product updates—or throttling and batching them—can significantly improve system performance and ensure a smoother, more stable storefront.

Advanced cache features in Shopware

Caching is not just about volume; it's about precision. Beyond the default HTTP cache mechanisms, Shopware now supports more advanced techniques that enable high performance even in dynamic storefront contexts. Two of the most important are **ESI (Edge Side Includes)** and **soft purges**.

ESI (Shopware ≥ 6.6.10.0) lets developers split page rendering into independently cacheable fragments. This is ideal when only parts of a page—like dynamic menus or widgets—need frequent updates. ESI allows most of the page to be cached, improving hit ratios and keeping response times consistent even for complex layouts.

Soft purge (Shopware ≥ 6.4.15.0) is essential for managing large-scale cache invalidations. Rather than immediately deleting expired entries, it marks them as stale but still serves them until refreshed. The first request triggers regeneration in the background, allowing users to receive functional content without backend spikes. This dramatically smooths out the impact of ERP updates and other cache-heavy operations—and often means the difference between resilience and outage.

Used together, these features unlock more sophisticated caching strategies, allowing developers to achieve faster storefronts without compromising on freshness or flexibility.

To get the most from these features, two key strategies should guide cache implementation:

- **Maximize cache coverage for anonymous traffic.**
  While not every page is cacheable—search results being a common exception— most storefront pages can and should be served entirely from cache. This reduces load from search engine crawlers, boosts SEO performance, and delivers faster, more consistent page loads. It also frees backend infrastructure to focus on personalised or transactional content

such
as checkouts or recently updated product data.

- **Optimize cache behaviour for logged-in users.**
  Cache coverage typically drops when a user creates a session (e.g., by adding an item to the basket). Since Shopware 6.6.10.0, however, it's possible to cache session fragments using ESI blocks for elements like headers and footers. This enables partial caching even during active sessions—extending performance gains deeper into the customer journey.

Used together, these strategies make caching not only fast and fresh, but resilient across all visitor states.

Extension and plugin management

In our experience, performance regressions are most frequently traced to third-party or custom plugins. These may introduce costly hooks, register redundant services, or execute blocking logic during template rendering or checkout flows. Performance-aware development and QA processes must include plugin impact assessment—both at the time of installation and during updates.

Even unused plugins may still consume resources, depending on how they are structured. Developers should routinely audit the full plugin stack, remove obsolete packages, and validate that active plugins are not introducing avoidable inefficiencies.

Admin configuration and application logic

The Shopware admin interface is a powerful tool, but misconfiguration at this level can significantly impact performance. For example, configuring category pages to display an excessive number of products can lead to inefficient query execution and costly rendering chains. Similarly, dynamic rule conditions and promotions must be designed with query complexity in mind—especially when operating on large catalogues.

Where possible, avoid relying on raw SQL-based search behaviour. Shopware PaaS supports integration with scalable search engines like OpenSearch, which offload search logic from the database and enable faster, more resilient querying. Properly configuring these services helps ensure that performance remains consistent even under high catalogue volume or complex filtering logic.

External API calls are another frequent oversight. Whether unsubscribing a user from a newsletter or retrieving data from a third-party service, these requests should never block storefront responses. APIs are inherently brittle—slowness or outages can cascade across your entire system. Whenever possible, external communications should be handled asynchronously to prevent third-party issues from degrading the shopping experience.

Logging levels must also be adjusted appropriately for production environments. Enabling verbose or debug-level logs in live storefronts can overwhelm disk I/O and delay application responses—an issue we have seen firsthand in customer projects.

It is also essential to review deployment logs with care. In at least one recent case, a misconfiguration resulted in a large volume of runtime errors, which in turn prevented Shopware-specific VCL snippets from being correctly loaded into the Fastly cache layer. Such issues are not always immediately visible through frontend behaviour alone. Particular attention should be given to logs related to hook execution, as these often surface critical errors that may otherwise go unnoticed during routine testing.

Performance as a discipline

A recurring theme across successful Shopware implementations is the presence of a defined performance budget. This includes measurable targets such as TTFB thresholds, acceptable CPU usage under peak load, memory constraints, and failure rate ceilings. These metrics should guide the development process and be validated continuously, especially during feature additions or third-party integrations.

Performance expectations must also be reflected in the development lifecycle. Git-based CI/CD workflows, environment consistency (dev, staging, production), and continuous observability are essential to detect regressions early. Teams that rely on manual uploads or inconsistent environments often struggle to identify or reproduce performance issues.

Preparing for critical events

Traffic peaks—whether seasonal, promotional, or unplanned—are not edge cases. They are the real test of architectural robustness. Teams should simulate load

using realistic traffic blends, test checkout flows under pressure, and ensure that the full application stack (including caching layers) behaves predictably under duress. A clear rollback strategy should be in place before any campaign, with staged deployments and observability instruments ready.

Summary

Running Shopware well is not merely about code quality—it is about anticipating the operational realities of a production storefront. Performance must be addressed proactively at every layer: infrastructure, configuration, application logic, caching, and deployment processes. When treated with discipline, Shopware can deliver excellent performance even under demanding conditions. But this discipline must be cultivated through intentional design, testing, and operational maturity.

Load testing methodology

To understand how Shopware PaaS performs under realistic conditions, we designed a series of controlled load tests. These tests were not intended to simulate edge-case stress, but to reflect **real-world traffic patterns** across browsing, checkout, and API activity. Our goal was to observe how the infrastructure handles load, identify scaling behaviours, and extract actionable best practices for tuning Shopware deployments.

Tooling and scenario design

We used Grafana K6 as the load testing framework, building on the Shopware-specific K6 scenarios published by the Shopware team. The most complete test script, `reference-scenario`, includes four key traffic patterns:

- browse_only
  - Simulates casual users or bots: homepage visits, search, category browsing, product views.
- browse_and_buy
  - Emulates a full customer journey: browsing, account creation, cart operations, and checkout.

- logged_in_fast_buy
  - Models repeat customers: quick login and direct checkout for a known product.
- api_import
  - Represents system integrations: product imports, stock updates, price changes.

These scenarios reflect a realistic e-commerce blend of anonymous traffic, buyer intent, repeat usage, and backend automation.

We ran this test using, at first, the free and open-source version of Grafana K6. However, the nature of a load test is to generate a lot of traffic. Shopware PaaS, being a production-grade solution, has some integrated DDOS mitigation layers. An abnormal amount of traffic coming from a single IP address is automatically blocked by these security layers.

So, we had to switch to the commercial Grafana K6 SaaS solution. Switching to it enables us to use different load zones, dividing the traffic by as much.

Configuration strategy

We tuned the tests to mirror production-like behaviour:

- Fixed dataset and codebase across all tests, to ensure comparability.
- Consistent cache state using Crawlee to prewarm Fastly's HTTP cache.
- Database reset before each run to guarantee clean start conditions.

We adjusted parameters such as the number of virtual users (VUs) and the delays (pauses) between actions to simulate natural usage, not raw concurrency:

- **browse_only VUs**: high volume, low-intent users.
- **buying VUs**: kept proportional to browse traffic to simulate realistic conversion rates.
- **pauses**: introduced to avoid saturation from constant back-to-back requests.

Without these pauses, smaller plans saturated almost instantly, producing results more characteristic of a stress test than a load test. Stress testing deliberately

pushes systems beyond their limits to expose failure modes, whereas load testing focuses on performance under realistic traffic levels — the kind of usage a Shopware site is expected to handle daily, including peak periods. Adding delays allowed us to regulate intensity and better replicate human behaviour, ensuring our measurements reflected practical conditions rather than theoretical extremes.[2]

To contextualize the results, it's important to note that our test dataset represented a mid-sized production catalogue. It included over 10,000 products and a realistic order volume designed to simulate stores processing tens of thousands of orders per day. This scale ensured that performance insights were grounded in practical use cases and applicable to typical enterprise deployments, rather than limited demo scenarios.

Target metrics

We calibrated each test to align with several production-facing criteria:

- Conversion rate of ~3% (visitors to buyers), derived from real Shopware PaaS customer data.
- API requests share between 5–10% of total traffic.
- **Response time target**: p95 Time-to-First-Byte (TTFB) under 600ms.
- **CPU usage ceiling**: kept near but below 70% to reflect sustainable peak operation.
- **Error rate**: under 0.05% failed requests.

Tests were executed for 5 minutes per configuration, repeated across six different plan types (from Grid XL to DGH) to assess scale progression.

Scenario calibration

To ensure load was both realistic and consistent:

- Visitor counts were tracked via counters in each scenario.
- A 50% bot traffic assumption was applied to browse_only.

[2] These changes have now officially been merged into the Shopware K6 repository, so that you can benefit from them, too.

- API pressure was varied to assess how cache invalidation affects system load.

Conversion rate was calculated as:

```
(Visitors from browse_and_buy + logged_in_fast_buy) /

((browse_only visitors / 2) + browse_and_buy +
logged_in_fast_buy)
```

API request share was calculated as:

```
API scenario requests / Total requests
```

Summary

This methodology allowed us to observe system behaviour under well-shaped, representative traffic loads—balancing realism with precision. In the next section, we'll present the results of these tests and show how performance scales across infrastructure tiers.

Using Blackfire to identify bottlenecks

No matter how well you size your infrastructure or tune your caching layers, performance bottlenecks in e-commerce applications almost always come down to application logic. That's where Blackfire comes in.

Shopware PaaS includes Blackfire by default—giving you powerful tools to profile, observe, and continuously optimise your Shopware storefronts.

Performance optimization is best approached as a continuous cycle. In many cases, the process begins not with a known issue, but with patterns observed through monitoring or continuous profiling. These tools help surface unusual trends, anomalies, or regressions. Once an issue has been detected, deterministic profiling enables teams to dig deeper—to understand precisely why a slowdown occurs, where time or resources are being spent, and what can be improved. From those insights, developers can define performance budgets, write targeted tests, and automate their enforcement in CI. This section focuses on the role of deterministic profiling within that broader cycle.

Let us now go through how to use Blackfire effectively in the Shopware context and why it's an essential part of your optimisation workflow.

Why profiling matters

Profiling is about more than tracking load time. It helps answer questions like:

- Why does this category page spike CPU?
- What's behind this sudden I/O contention?
- Is my ERP update logic inefficient?
- Why does this plugin kill performance in staging but not locally?

Blackfire works by instrumenting requests and providing **visual call graphs, function-level timing,** and **resource usage** insights—directly in your staging or production environments (non-intrusively).

In Shopware projects, profiling is especially valuable when:

- Deploying new plugins or features.
- Handling high-traffic campaigns (e.g. sales, seasonal events).
- Diagnosing unexplained slowness.
- Preparing a new site for go-live.

How to profile a Shopware page

You can initiate a profile from:

- The Blackfire **browser extension** (Chrome/Firefox)
- The Blackfire CLI in staging or development
- Your CI pipeline (for automated test-based profiling)

Each profile gives you:

- A flame graph of the full request lifecycle.
- Execution time per function or service.

- I/O, memory, and HTTP client usage hotspots.
- Time spent in templates, Doctrine queries, Redis calls, and plugins.

For Shopware, focus on:

- `Shopware\Core\Content\Product` and `ProductListingRoute`
- Template rendering layers
- Event dispatchers and plugin listeners

Shopware-specific insights in Blackfire

Beyond standard profiling capabilities, Blackfire offers Shopware-specific instrumentation that exposes performance metrics tightly aligned with the platform's architecture. These include precise timings for components such as ProductListingRoute, template rendering performance, and plugin listener overhead. These metrics are surfaced as part of Blackfire's custom timeline view, enabling developers to correlate bottlenecks with specific Shopware layers. This integration helps teams prioritize optimizations more effectively—whether in response to slow category pages, API routes, or admin interactions. Shopware's own engineering team makes extensive use of these metrics to support their internal performance workflows, including regression detection, CI integration, and large-scale tuning efforts.[3]

Common bottlenecks in Shopware projects

Based on real usage, here are some patterns to look out for:

- **Plugins with inefficient hooks**
  Profiling reveals plugins that hook into every request and perform expensive logic—e.g., redundant DB lookups or external API calls.
- **Overloaded product listings**
  Pages that render too many products or lack pagination can trigger massive SQL queries and DOM tree rendering.

[3] See Optimize your Shopware 6.x applications with new specific metrics & Meeting Uwe Kleinmann:
Shopware performance optimization with Blackfire

- **Unbatched ERP updates**
  Frequent product updates without batching or delay can saturate I/O, invalidate caches, and block PHP workers.
- **Verbose logging in production**
  As seen in the PayPal debug case, this can kill disk I/O and delay responses by orders of magnitude.

Continuous profiling (optional)

With Blackfire Enterprise, you can enable **continuous profiling**, which:

- Runs lightweight, always-on profilers on your live environment.
- Detects regressions in memory usage, I/O time, and function duration.
- Surfaces issues before they **impact customers**.

This is ideal for Shopware projects where:

- Development is fast-moving.
- Code may be maintained by multiple development teams.
- Unexpected performance issues can damage sales.

Integrating Blackfire into CI/CD

You can integrate Blackfire directly into your deployment pipeline:

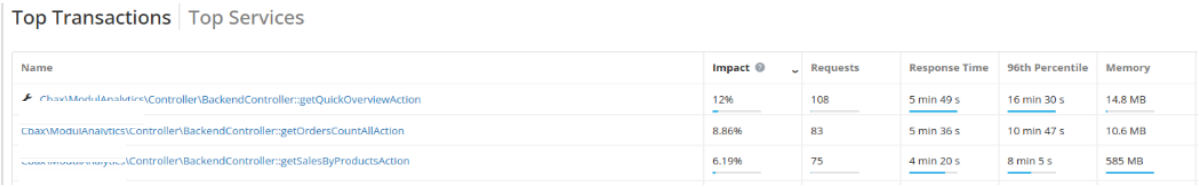
1. Define performance assertions (e.g. “X function must not exceed Y ms”).
2. Run tests as part of your deployment or pull request builds.
3. Fail builds if regressions are detected.

This keeps performance **measurable, trackable, and testable**—just like functional bugs.

Analytics extension induces downtime

Three years prior, a pilot customer for Shopware PaaS encountered significant performance degradation, occasionally resulting in downtime lasting several minutes, occurring multiple times monthly. These incidents transpired sporadically

throughout the day, thereby complicating the identification of the underlying issues by the development team. Utilizing Blackfire, enabled the immediate detection of a Statistics module employed by the customer for the aggregation of data regarding their online store activity. This module was executing exceptionally large queries that effectively locked the MySQL database for durations of up to 16 minutes.



While such incidents are not uncommon, it is imperative to underscore the transformative role of Blackfire. Upon the manifestation of performance anomalies, it facilitated the rapid identification of the root cause within a matter of minutes. Such visibility converts prolonged, speculative analyses into swift and assured diagnoses.

Regarding recommended practices: For the execution of extensive database queries, it is advisable to operate on a live replica of the production database, leveraging the MySQL binary log feature. This strategy mitigates the risk of impacting the performance of the online store.

Summary: Blackfire in your Shopware toolbox

Use case	Blackfire benefit
Plugin evaluation	Detect runtime costs before go-live
Slowness in category pages	Reveal expensive queries and rendering hotspots
ERP sync optimisation	Profile I/O and cache invalidations
Go-live prep	Compare staging vs prod traces

Shopware PaaS includes Blackfire so you can go beyond “works on my machine” and understand exactly **why** your store performs the way it does.

Learn more at docs.blackfire.io or the Shopware PaaS Blackfire guide.

Performance results across plans

With the load testing framework and scenarios in place, we executed tests across seven Shopware PaaS infrastructure plans: Grid (XL, 2XL, 4XL), Dedicated Grid Hosts (DGH16, DGH32, DGH64 and DGH128). The goal was to measure how each plan handles real-world traffic patterns, including browsing, buying, and API operations, and to quantify their scalability and performance characteristics.

Key metrics collected

For each plan, we collected and computed a range of performance metrics:

- Total Requests
- Failures and Failure Rate
- Peak Requests per Second (RPS)
- p95 Response Time (TTFB)
- CPU Usage (Application, Redis, DB)
- Conversion Rate
- Orders per hour / day
- API Load Share
- Virtual User Distribution per Scenario

These metrics were used to assess both infrastructure efficiency and the overall user experience under load.

We especially and consistently aimed for a TTFB (Time To First Byte) below 600 milliseconds.[4] This threshold is not arbitrary: it aligns with industry standards for excellent web performance. According to Google's Core Web Vitals, a TTFB under

[4] While Time to First Byte (TTFB) remains a key indicator for identifying backend and infrastructure issues, it should not be the sole metric guiding performance evaluation. For a more complete picture of perceived performance, we recommend supplementing TTFB measurements with tools such as Google PageSpeed Insights and Chrome UX Reports. These offer broader visibility into frontend experience and user-centric metrics.

800ms is considered “good.” Our 600ms target reflects a more ambitious goal designed to account for real-world conditions, backend processing complexity, and variability in user devices and network speeds. Achieving sub-600ms TTFB across realistic Shopware use cases is a strong indicator that a storefront will load quickly, remain responsive under load, and perform well on both SEO and UX benchmarks. It also sets a defensible baseline against which performance regressions can be measured.

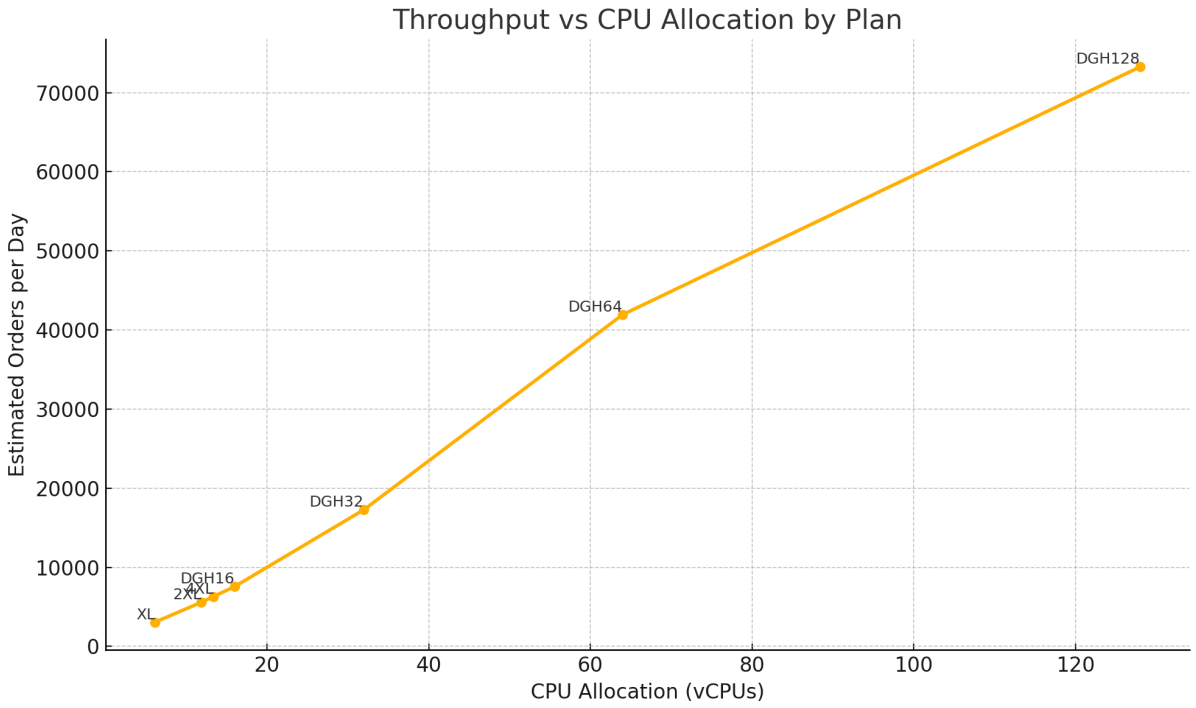
Summary of results

Plan	Total CPU	Peak RPS	P95 TTFB	Orders/day	% CPU used	% failures
XL	6.15	32	537ms	3,060	78.4%	0.01%
2XL	11.9	55.3	549ms	5,580	68.6%	0.03%
4XL	13.4	61.7	549ms	6,300	67.7%	0.01%
DGH16	16	76	514ms	7,560	70.2%	0.00%
DGH32	32	165.7	598ms	17,280	70.9%	0.01%
DGH64	64	459.67	578ms	41,940	75.1%	0.00%
DGH128	128	898.33	545ms	73,260	59.6%	0.00%

Observations

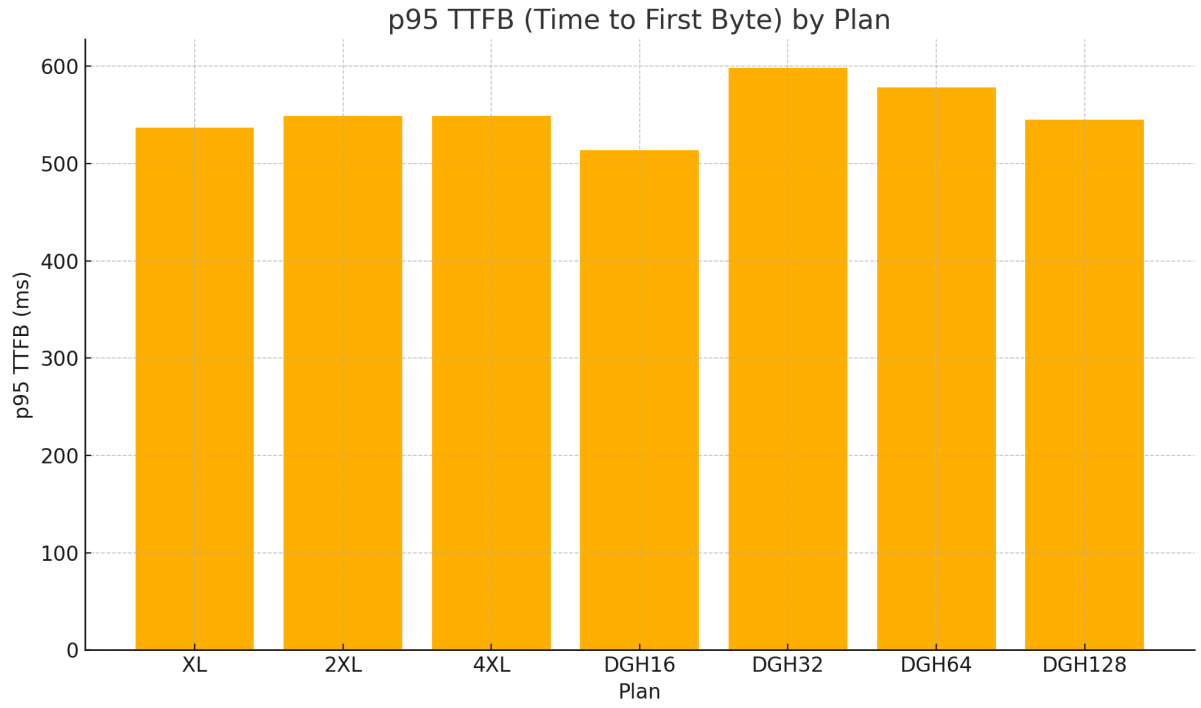
1. Performance scales predictably with resources

As expected, throughput (measured in orders/day and RPS) increased consistently with CPU allocation and infrastructure tier. Even modest plans like XL were capable of processing over 3,000 orders/day under tuned conditions.



2. p95 response time held steady

p95 TTFB remained under or near the 600ms target, confirming that Shopware PaaS delivers responsive performance under realistic usage. But to keep this objective, we had to cap the API import VUs at a maximum of 5. Going over five increased the TTFB and limited the RPS, even with the biggest plans.



3. API load is expensive

API requests, especially those that create or update products, invalidate multiple cache layers and put stress on both the database and the Fastly edge. Even at just 5–10% of total traffic, they account for a disproportionate share of resource usage.

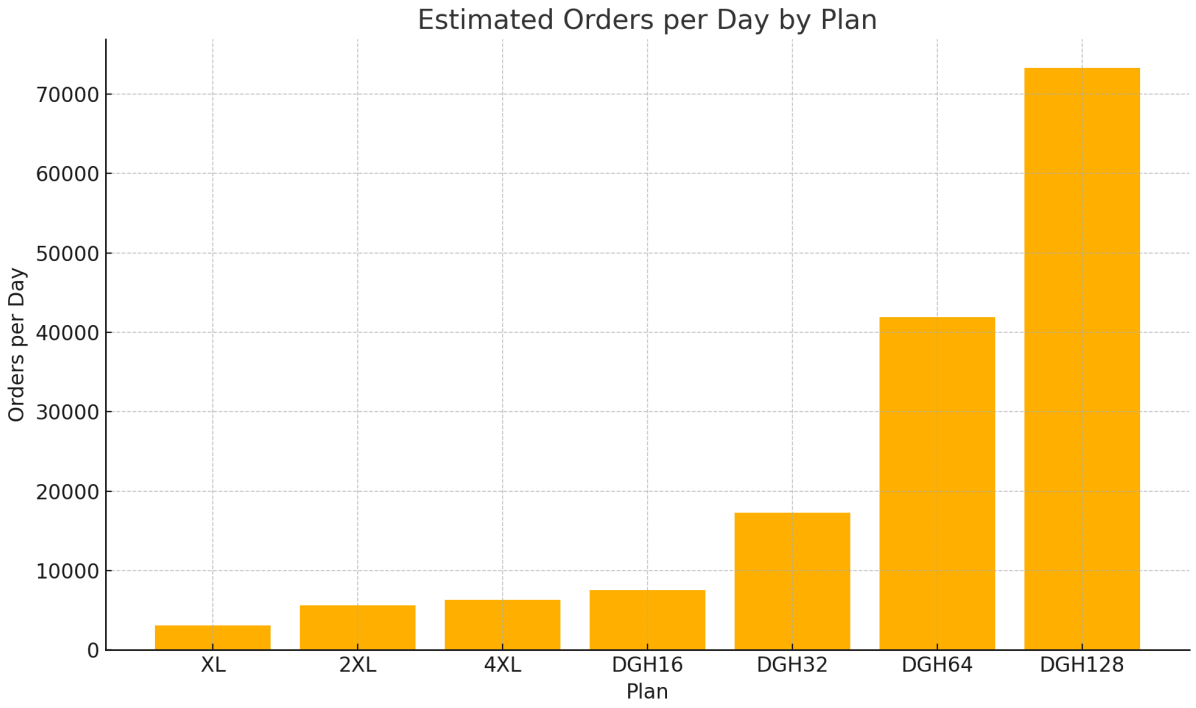
4. Low failure rates across all tiers

All plans maintained failure rates below 0.05%, with zero failures observed on DGH16 and only a handful across other tiers. This demonstrates Shopware PaaS's resilience under consistent load.

Orders per day by plan

We estimated daily order capacity using the following formula:

```
Orders per hour × 12 × (100 / 80)
```



This reflects the assumption that 80% of daily orders occur during a 12-hour peak period. Below are the estimated order capacities:

- XL: 3,060 orders/day
- 2XL: 5,580 orders/day
- 4XL: 6,300 orders/day
- DGH16: 7,560 orders/day
- DGH32: 17,280 orders/day
- DGH64: 41,940 orders/day
- DGH128: 73,260 orders/day

Choosing the right plan

These results make it clear that while every tier is capable of supporting production workloads, higher plans offer both **headroom and predictability** for teams expecting sustained traffic or conversion surges. That said, raw infrastructure is not the sole determinant of performance—your code,

configuration, and cache behaviour play a critical role in whether your storefront runs fast or falters.

The next sections will explore best practices, profiling techniques, and lessons learned from tuning Shopware in the field.

Lessons learned from the field

Over the course of supporting and evaluating numerous Shopware PaaS implementations, several patterns have emerged that consistently distinguish successful projects from those that struggle with performance and reliability. These lessons, drawn from real-world experience, offer guidance on avoiding common pitfalls and adopting practices that lead to operational stability.

Configuration must be intentional

One of the most common sources of performance degradation arises not from infrastructure limitations, but from default or ill-considered configuration choices. This is particularly evident in cases where category pages are set to display too many products, leading to expensive database queries and excessive template rendering. In other instances, logging levels were left at debug in production environments, introducing significant I/O contention and response time delays.

Projects that perform well tend to exhibit a high degree of intentionality in their setup. Pagination settings, logging verbosity, cache invalidation behaviour, and plugin configurations are all explicitly tuned to support performance, not simply left in their default state.

Plugin usage requires vigilance

While Shopware's extensibility is a strength, it also introduces risk. Third-party and custom plugins frequently account for the majority of runtime overhead in poorly performing storefronts. This is not due to malice or incompetence, but simply the complexity of integrating diverse behaviours into a shared runtime.

A plugin may introduce hooks that trigger on every request, perform redundant database operations, or interfere with cacheability. Teams often underestimate the cumulative cost of such extensions, especially when multiple plugins interact with the same lifecycle events. High-performing projects treat plugins as runtime

components to be evaluated and, where necessary, replaced or disabled—not just installed.

Caching is central, but often undervalued

Caching remains the single most effective strategy for performance improvement in Shopware. However, its success depends on deliberate implementation. The architecture must support cacheability, including the use of `GET` methods over `POST`, proper cache headers, and URL structures that avoid unnecessary variation.

In many cases, performance issues stem not from an absence of caching, but from cache invalidation processes triggered by unbatched ERP updates or administrative changes. Projects that succeed in maintaining performance under load are those that manage invalidation carefully, employ soft purge strategies, and avoid unbounded cache refresh cycles during business hours.

Performance testing must be realistic

Several underperforming projects had previously undergone performance testing—but in environments or conditions that failed to reflect actual usage. Effective testing requires more than synthetic requests; it demands representative traffic blends, realistic conversion rates, appropriate delays between actions, and accurate replication of backend behaviours.

This also includes your CDN. While including Fastly in the load test may seem controversial, it is in fact essential. Fastly is a key component of the runtime environment—excluding it prevents early detection of issues such as large-scale cache purges or invalidation patterns. On Shopware PaaS, each staging environment comes with its own dedicated Fastly service. Make sure it's correctly configured, and use it actively during UAT (User Acceptance Testing) to validate real-world caching behaviours before go-live.

Testing that omits elements like API imports, bot traffic, or concurrent checkout processes may fail to reveal scalability bottlenecks until they manifest in production. Realistic load simulation is critical not only for infrastructure sizing, but for understanding how services interact under stress.

Operational discipline drives resilience

Successful Shopware projects are characterised by their adherence to operational discipline. CI/CD pipelines are in place, staging environments match production, monitoring and alerting are configured, and rollback procedures are well-tested. Conversely, projects that rely on manual deployment, irregular updates, or inconsistent environments tend to experience avoidable instability and regressions.

This extends to campaign preparation: peak traffic events are treated as technical milestones, not just business opportunities. High-performing teams rehearse traffic scenarios in advance, prewarm caches, monitor system metrics throughout, and ensure rollback readiness.

Collaboration is essential

Finally, it is worth emphasising that achieving consistent performance in Shopware is not solely a matter of technical configuration—it is also a question of collaboration. Many of the most effective resolutions we have seen were the result of early engagement between development teams, infrastructure specialists, and support engineers. Performance issues often span concerns that no single stakeholder fully owns. Bringing expertise together early prevents avoidable missteps and enables faster, more comprehensive resolutions.

Summary

The most consistent lesson from the field is this: **Shopware performance is a shared, continuous responsibility**. It is shaped not only by infrastructure size or application code, but by how thoughtfully a project is configured, extended, observed, and operated. High-performing projects are those that approach performance as a design concern, an integration concern, and an operational concern—never as an afterthought.

This is especially evident when looking at real-world deployments. In one case, a misconfigured payment plugin logging at debug level generated excessive disk I/O, severely slowing the site. Another project experienced regular site freezes due to an analytics module issuing long-running SQL queries that locked the database. One customer even generated over 500 million cache invalidations per day

through unbatched ERP updates, leaving the cache perpetually cold and negating the performance benefits of edge caching entirely.

In contrast, projects that achieved stable, fast response times shared certain habits: they employed tools like Blackfire to locate bottlenecks, used deferred or soft purging to manage cache invalidation intelligently, and maintained close alignment between ERP updates and cache regeneration strategies. They didn't wait for performance to degrade — they designed for it, monitored it, and treated it as part of their delivery pipeline.

These patterns consistently underscore the same point: **Shopware's defaults are not always suitable for large-scale ecommerce, but performance can become exceptional with the right operational mindset.**

Conclusion and next steps

The results of our testing and field experience demonstrate that Shopware can perform exceptionally well—but only when supported by deliberate infrastructure decisions, thoughtful architectural patterns, and disciplined operational practices. Shopware PaaS provides a stable and scalable foundation, but the responsibility for performance extends far beyond the infrastructure itself. It reaches into every layer of development, configuration, integration, and deployment.

This white paper has outlined what distinguishes high-performing Shopware projects from unstable ones. From tuning PHP-FPM sizing to managing cache invalidation, from reviewing plugin behaviour to preparing for peak traffic events, the lessons are consistent: performance is not something to be retrofitted—it must be designed and maintained.

What you can do next

If you are evaluating Shopware PaaS, or currently running Shopware on any infrastructure, we encourage you to reflect on the following:

- **Review your caching strategy**: Is it intentional, layered, and actively prewarmed?
- **Audit your plugin stack**: Have runtime impacts been measured and reviewed?

- **Validate configuration**: Are your sizing hints and logging settings appropriate for production?
- **Replicate traffic**: Have you tested your storefront under realistic user behaviour and concurrency levels?
- **Standardize deployment**: Are CI/CD pipelines and rollback procedures in place?
- **Use environment variables for configuration**: Ensure that differences between production, staging, and development environments are handled via environment variables. Shopware PaaS makes this easy—and doing so helps avoid deployment-time surprises and misconfigurations.
- **Protect your infrastructure:** Ask about the Fastly Next-Generation WAF and our Edge Rate limiting features (available in option) to mitigate risks of resource abuse.
- **Centralize your logs and audit them:** All Shopware PaaS plans include the log forwarding feature (available on all architectures). Ship your logs to your favorite tools (Sumo Logic, Splunk, New Relic and rsyslog supported) and audit them regularly. MySQL `slow_request.log`, PHP `error_log` often contains critical information you should not ignore.

Performance excellence is not reserved for large teams or enterprise budgets—it is the result of engineering discipline applied consistently.

Pre-launch readiness checklist

Before taking a Shopware storefront live, or launching a major feature or campaign, the following conditions should be confirmed:

- ☐ Product and category listings are paginated and optimized for efficient SQL queries (e.g. limits, indexing).
- ☐ Plugin and extension performance has been profiled; unnecessary or slow plugins are disabled or deferred.
- ☐ Logging levels are correctly set for production (no debug logs unless needed for short-term diagnostics).

- [ ] ERP and external synchronisations are batched and scheduled outside peak shopping hours.
- [ ] API-driven cache invalidations are delayed, batched, or throttled to preserve cache efficiency (Shopware ≥ 6.7).
- [ ] Cache prewarming is in place for key landing pages and navigation flows (homepage, categories, etc.).
- [ ] HTTP cache is working: `X-Cache` headers return `HIT` for expected routes. `MISS` anomalies are investigated.
- [ ] The volume of cache invalidations per day remains within expected bounds—investigate spikes.
- [ ] Infrastructure capacity has been sized for realistic peak load, with CPU usage staying below ~70% under test.
- [ ] Load tests have been run using realistic traffic mixes, device types, and session behaviours (not just synthetic spikes).
- [ ] CI/CD pipelines are tested and produce deterministic builds in staging and production.
- [ ] Monitoring and alerting are enabled for:
    - [ ] Application-level metrics (response time, error rate)
    - [ ] System-level metrics (CPU, memory, disk I/O)
    - [ ] Business-critical flows (checkout, search, login)
- [ ] Deployment logs are reviewed for uncaught exceptions or failed hooks.
- [ ] 3rd-party integrations (e.g. payment gateways, PIMs, CRM) are monitored for availability and latency.
- [ ] Security headers and HTTPS enforcement are tested (e.g. Content Security Policy, HSTS).
- [ ] SEO-critical pages (homepage, categories, product detail) are crawlable and performant under PageSpeed Insights.

While not exhaustive, this checklist reflects the key operational patterns we have found essential to stability, performance, and long-term success.

Final note

Throughout this guide, we've seen that high Shopware performance doesn't result from infrastructure alone—it is the product of intentional design, disciplined engineering, and operational maturity.

Shopware PaaS is built on Upsun, a cloud application platform that combines scalable infrastructure, native observability, and developer-first tooling. With features like built-in CI/CD, configurable environments, and integrated support for profiling tools such as Blackfire, Upsun provides a robust foundation for sustainable performance at scale.

But just as importantly, we don't stop at the contract line. Our teams work directly with agencies and merchants to surface misconfigurations, troubleshoot bottlenecks, and share best practices drawn from real-world deployments. Whether it's shaping cache invalidation behaviour, optimizing catalog indexing, or tuning PHP limits for large datasets, we work alongside you to get it right.

Performance, when treated as a shared responsibility, becomes a competitive advantage. The principles in this paper are not theoretical—they are field-tested and accessible. Whether you're maintaining one storefront or many, improvements in caching, configuration, and deployment discipline can yield measurable impact.

Start small. Choose one domain—cache behaviour, plugin management, deployment workflows—and inspect it with performance in mind. If you're already using Shopware PaaS, our support teams are ready to help you go further. If you're evaluating the platform, we invite you to connect.

Appendix

This appendix provides supplementary information to support the content presented throughout the white paper. It includes configuration examples, testing assumptions, calculation formulas, and reference links for teams seeking to replicate or adapt the testing and optimisation strategies described.

Load testing setup and parameters

Sample virtual user configuration

The number of virtual users (VUs) per scenario was tuned to achieve a realistic blend of browsing and buying behaviour while respecting CPU usage thresholds and conversion rate targets.

Scenario	Description	Typical VU count (DGH64)
`browse_only`	Anonymous traffic, including bot patterns	150
`browse_and_buy`	New customer flow	5
`logged_in_fast_buy`	Repeat customer quick purchase	5
`api_import`	ERP-originated product updates	5

Pause settings

To simulate human interaction pacing:

Scenario	Default pause
`browse_only`	300 ms
`browse_and_buy`	500 ms
`api_import`	10 ms

These delays prevented artificial saturation and better modelled production usage.

Performance metric calculations

Conversion rate

Calculated as a proportion of buying visitors over adjusted total visitor volume:

```
(visitors_browse_and_buy + visitors_logged_in_fast_buy)

÷

((visitors_browse_only / 2) + visitors_browse_and_buy +
visitors_logged_in_fast_buy)
```

A 50% bot share was assumed in `browse_only` traffic.

Orders per Day

Orders per hour were projected into daily throughput based on an 80/20 peak traffic model:

```
orders_per_day = (orders_per_hour × 12) × (100 / 80)
```

This assumes 80% of orders occur during a 12-hour peak window.

Sample configuration snippet (`.platform.app.yaml`)

Tuning PHP-FPM concurrency through memory sizing hints:

```
None
runtime:

    extensions:

        - redis

    directives:

        memory_limit: 512M


web:

    commands:

        start: |

            php-fpm -F


    sizing_hints:

        request_memory: 24

        reserved_memory: 70
```

These values inform automatic calculation of `pm.max_children` by the Upsun runtime.

Shopware Performance tweaks

Following the official performance tweaks documentation, we applied the following changes:

Sending mails with the Queue:

```
None
config/packages/prod/framework.yaml
+framework:
+    mailer:
+        message_bus: 'messenger.default_bus'
```

Prevent mail data updates:

```
config/packages/prod/shopware.yaml
+shopware:
+    mail:
+        update_mail_variables_on_send: false
```

Increment storage:

```
config/packages/paas.yaml
+    user_activity:
+        user_activity:
+            type: 'array'
+        message_queue:
+            type: 'array'
```

Recommended tools and resources

- Blackfire Documentation
  https://docs.blackfire.io
- Shopware PaaS Blackfire Integration
  https://developer.shopware.com/docs/products/paas/blackfire.html
- Grafana K6 (Open Source)
  https://k6.io
- Shopware K6 Scenario Repository
  https://github.com/shopware/k6-shopware
- Crawlee (for cache prewarming)
  https://crawlee.dev
- PHP-FPM Runtime Tuning (Upsun)
  https://fixed.docs.upsun.com/languages/php/fpm.html