

PERFORMANCE AUDIT · SAMPLE REPORT

What slows this store down, and how to fix it

Front-end speed, code quality and server-side performance of a Magento 2 store, with prioritised recommendations.

STORE

Magento 2 store (anonymised)

AUDIT TIME

Up to 3 working days

ENVIRONMENT

Local copy, live site untouched

PREPARED BY

SYSINT

This sample is based on a real audit carried out in 2020. The store, domain, module vendors and code paths have been removed. Front-end metrics reflect the Lighthouse version used at the time.

01

Summary

The store works, but every key page is slow, and the code base carries thousands of issues that make each new feature riskier and more expensive. Most of the speed can be won back with caching and front-end clean-up. The code issues need a planned, step-by-step refactoring.

31-46

Lighthouse performance score, out of 100

8.7 s

Server time to build a product page

1,057

Database queries on one category page

E

Security and reliability rating of the code (A is best)

Key findings

- **Pages are slow for mobile visitors.** Home, category and product pages take 12.8–14.4 s to become usable. Over half of visitors leave a site that takes more than 3 seconds to load.
- **Third-party modules weigh down every page.** They load their own scripts and styles on pages where they aren't needed, and one large merged script blocks the page.
- **The server does far more work than it should.** A category page runs 1,057 database queries. The same small queries repeat dozens of times.
- **Code quality is far below safe limits.** 3,637 bugs, 40 vulnerabilities and 7,794 maintainability issues, almost all inside customised and third-party code.

What to do first

Priority	Action	Expected effect
1	Turn on full-page caching (Varnish) and move sessions and cache to Redis	Up to 2× faster pages; less load on the disk
2	Cache repeated database queries, starting with product categories	Up to 52× fewer queries for that call
3	Defer non-critical scripts, lazy-load images on category and product pages	Faster first screen and earlier interactivity
4	Enable OPcache	–17% memory, –6.8% load time on a product page
5	Review modules and refactor the riskiest code step by step	Fewer bugs; safer, cheaper new features

02

Scope and method

We deploy a copy of the store locally and run every test there, so the live site is never touched. The audit covers three areas.

Area	What we check	Tools
Front-end	Load speed and interactivity of key pages; what blocks rendering	Google Lighthouse
Code quality	Bugs, vulnerabilities, security hotspots and maintainability in PHP, PHTML, JavaScript, CSS and XML	SonarQube, 180+ rules
Server-side	Page build time, CPU, memory, disk wait and every database query	PHP profiler

Test environment

Operating system	Ubuntu 16.04 x64
Memory	4 GB RAM, 4 GB swap
Storage	80 GB SSD
CPU	Intel Xeon E5-2650 v4, 2.20 GHz
Runtime	PHP 7.2 (FPM), Apache 2

Pages tested: home page, a category page and a product page. Database query counts also include checkout.

03

Front-end speed

Lighthouse scores below 50 mean the page is slow for most mobile visitors. All three key pages fall in that range.

Metric	Home	Category	Product	Target
Performance score	46	31	31	90+
First Contentful Paint	3.3 s	3.5 s	3.3 s	< 1.8 s
Speed Index	5.0 s	7.4 s	7.2 s	< 3.4 s
Time to Interactive	12.8 s	14.4 s	14.3 s	< 3.8 s
First CPU Idle	7.0 s	7.8 s	13.2 s	—
Max Potential First Input Delay	150 ms	160 ms	280 ms	< 130 ms

Why the pages are slow

- A large page structure (DOM), caused by banners, many products and product tags.
- Third-party modules load their own scripts and styles on every page, often for small tasks.

- All scripts are merged into one large file: one slow function drags down the whole file.
- Images in the header and menu load before the content that matters.
- The theme was bought ready-made. Its built-in modules add code to pages where it does nothing.

Recommendations

- Inline the styles and scripts for the menu, links and buttons, so visitors can use the page before everything loads.
- Load page-specific styles and scripts only on the pages that need them.
- Move non-critical scripts to the end of the page.
- Lazy-load images. On category and product listing pages this is a must.
- Remove unneeded elements from the page structure and avoid images in the header and menu.
- Check every new module for its effect on speed before it goes live.

04

Code quality

We scanned all store code with SonarQube against safe limits for a maintainable project. Every limit is exceeded, most by a wide margin.

Metric	Safe limit	This store	
Reliability rating	A	E	Fail
Security rating	A	E	Fail
Maintainability rating	A	A	Pass
Blocker issues	< 10	103	Fail
Critical issues	< 10	2,657	Fail
Bugs	< 10	3,637	Fail
Vulnerabilities	< 10	40	Fail
Code smells	< 50	7,794	Fail
Cyclomatic complexity	< 50	72,337	Fail

Where the issues are

Code area	Lines	Bugs	Vulnerab.	Code smells	Hotspots
Whole store	445,089	3,637	40	7,794	574
Core application with third-party modules	431,943	3,528	39	7,369	572
Main custom module	7,714	88	1	183	2
17 smaller custom modules	5,432	21	0	242	0

Duplicated code across the store: 15.8%.

Most common issues

Issue	Count	Why it matters
Functions too complex to follow	110	Hard to review; most likely place for new bugs
Functions with too many parameters	99	A function doing too many things; one had 18 parameters
Local variables shadowing class fields	76	Easy to change the wrong value
Nested conditions that should be merged	57	Harder to read, easier to break
Missing curly braces in control structures	39	Easy to misread which line a condition covers
Empty blocks, often in error handling	38	Errors are hidden instead of logged

Issue	Count	Why it matters
Different methods with identical code	29	Duplicated logic drifts apart over time
Switch statements without a default case	21	Unexpected values go unhandled
Functions with too many lines	17	Slow and risky to change
Unused private fields	9	Unclear whether code is dead or forgotten

What this means for the business

A code base in this state makes every change slower and riskier: a new feature can break two existing ones. As traffic and data grow, the store becomes harder to extend, and when customers or partners ask for something new, the team may not be able to deliver it. Projects that have changed hands between several developers or agencies rarely pass even basic quality checks.

05

Server-side performance

We profiled how long the server takes to build each page and what it spends that time on.

Page	Build time	CPU	Disk wait	Memory	DB queries
Home	4.18 s	3.74 s	448 ms	42.9 MB	162 · 153 ms
Category	8.30 s	6.82 s	1.49 ms	62.9 MB	1,057 · 826 ms
Product	8.69 s	6.97 s	4.72 s	61.2 MB	717 · 1.02 s

Checkout page: 175 database queries.

Disk load on a product page

Database driver	57%
Cache read/write	17%
Sessions	11%
Other	15%

CPU load

XML layout and templates	60%
Functions and data processing	40%

What drives the load

- **The database.** The more queries and joined tables, the longer the disk wait. Data grows over time: a function written for 1,000 records may face ten times more within a year.
- **Sessions stored on disk.** Without Redis, disk load grows with every visitor.
- **Too many modules.** Each front-end module adds layout files the server must process, and loads more classes into memory.

Queries to fix first

What happens	Calls	Fix
A layered-navigation module reads the same attribute options again and again	121× and 86×	Cache the options per page
Product categories are fetched separately for every product	52×	Cache category IDs: 52× fewer queries
URL rewrites use a LEFT JOIN where every row must exist	—	Switch to INNER JOIN
Price-range filters recalculate the same results	repeated	Cache the results
Category and product attributes are loaded through large UNION queries	17× and 16×	Review and cache attribute loading

06

Recommendations

Ordered by effort. Quick wins can be done in days; refactoring is planned and done step by step alongside regular work.

Quick wins: configuration

- Turn on Varnish full-page caching. Expect up to 2× faster pages; in one test a 1.2 s page dropped to 200–250 ms.
- Store sessions and cache in Redis or Memcached instead of on disk.
- Enable OPcache: on a product page it cut memory use by 17% and load time by 6.8%.

Short term: front-end and database

- Inline critical styles and scripts; defer the rest to the end of the page.
- Lazy-load images on category and product listing pages.
- Move small shared stylesheets (calendar, print) into the main stylesheet.
- Cache repeated queries: category IDs, attribute options, price-range results.
- Remove references to unused classes to cut memory and disk reads.

Planned: code and modules

- Review all modules. Merge essential functions into one module; drop admin configuration from modules that don't need it.
- Refactor the riskiest code first: blocker and critical issues, empty error handlers, overly complex functions.
- Add a speed and quality check for every new module before release.
- Agree coding standards for everyone who works on the store, so quality doesn't drop again.

Want this for your store?

Performance audit: up to 3 days, \$560. We deploy a local copy, run the same tests and send you a report like this one, written for your store.

sysint.net · We reply within one business day