EXPERT SOFT
software development

The Hidden Cost of Cache: How Misuse Undermines Performance in E-commerce Systems

Where caching goes wrong and how mature systems stay fast

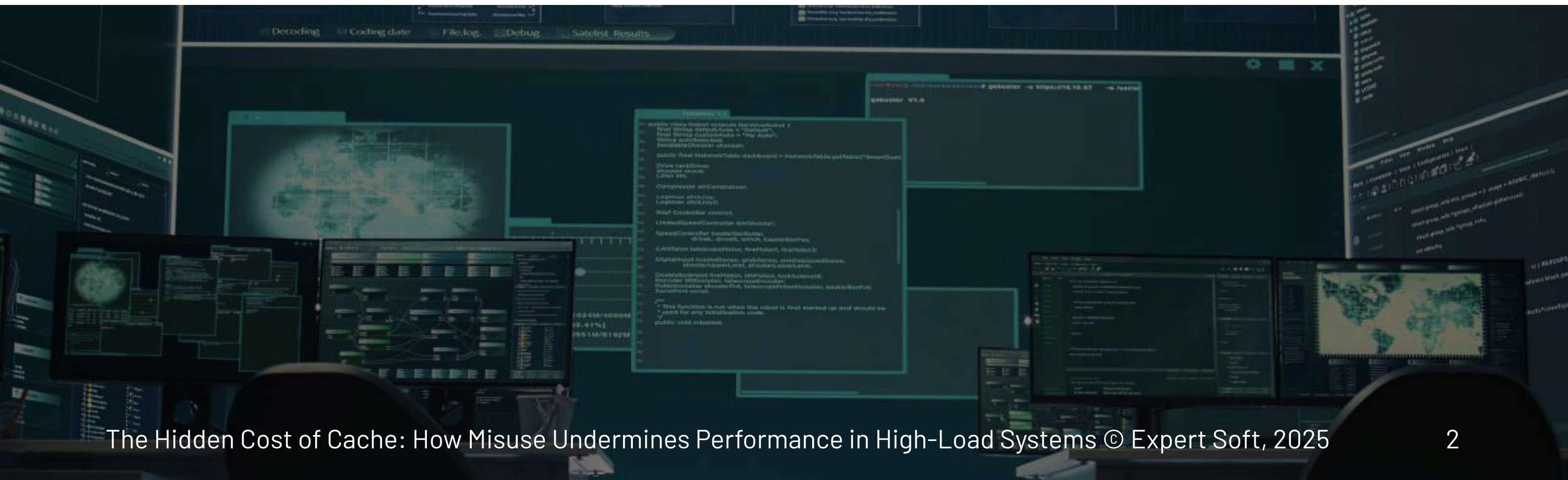
Many developers, especially those working on high-load ecommerce systems, assume that the more they use cache, the better the performance. But here's the thing: caching isn't always helpful. In fact, when used without a clear strategy or applied blindly, it can degrade performance, strain resources, and introduce hard-to-catch bugs.

This whitepaper looks at the less obvious side of caching, revealing the moments when it starts to work against you. Based on Expert Soft's hands-on experience, the document includes real-world red flags, examples of over-optimization, and field-tested best practices that challenge common assumptions.

You'll also find real cases, where caching added complexity instead of speed and how we at Expert Soft recognized and resolved those issues.

As well, we'll show you when not to cache and why that decision marks true engineering maturity. No fluff, just hard-earned insights. Let's dive in.

Table of Contents



Warning Signs That Your Cache Is Hurting You

When a response takes five seconds, alarms go off and teams jump into action. Obvious problems get obvious attention. But what if everything seems fine? The cache is configured, the system is stable, so it definitely can't be the cache. But don't leap to conclusions.

These red flags show that your cache is off, even if the app "still works"

Cache hit rate is high, but performance doesn't improve
Cache might be working, but for the wrong thing. You're likely to spend memory and effort optimizing something that no longer impacts user experience.

Memory usage creeps up during indexing or imports
Without smart eviction, the cache fills up with bulky temporary data. You waste memory, strain infrastructure, and gain little-to-no performance in return.

Cache invalidation depends on human actions
One forgotten refresh opens up space for stale data, bugs, and user-facing errors. It's error-prone, non-scalable, and usually signals poor architecture.

Same cache logic for static and dynamic data
Different data types, different behaviors, and treating them the same leads to stale stock, wrong prices, and bugs your users will definitely notice.

Frequent cache misses despite active caching
High miss rates usually mean your key design or TTLs are misaligned with access patterns, leading to avoidable DB calls and latency.

Caching introduces more delay
If adding a cache makes responses slower, it's often due to remote cache location or heavy (de)serialization overhead.

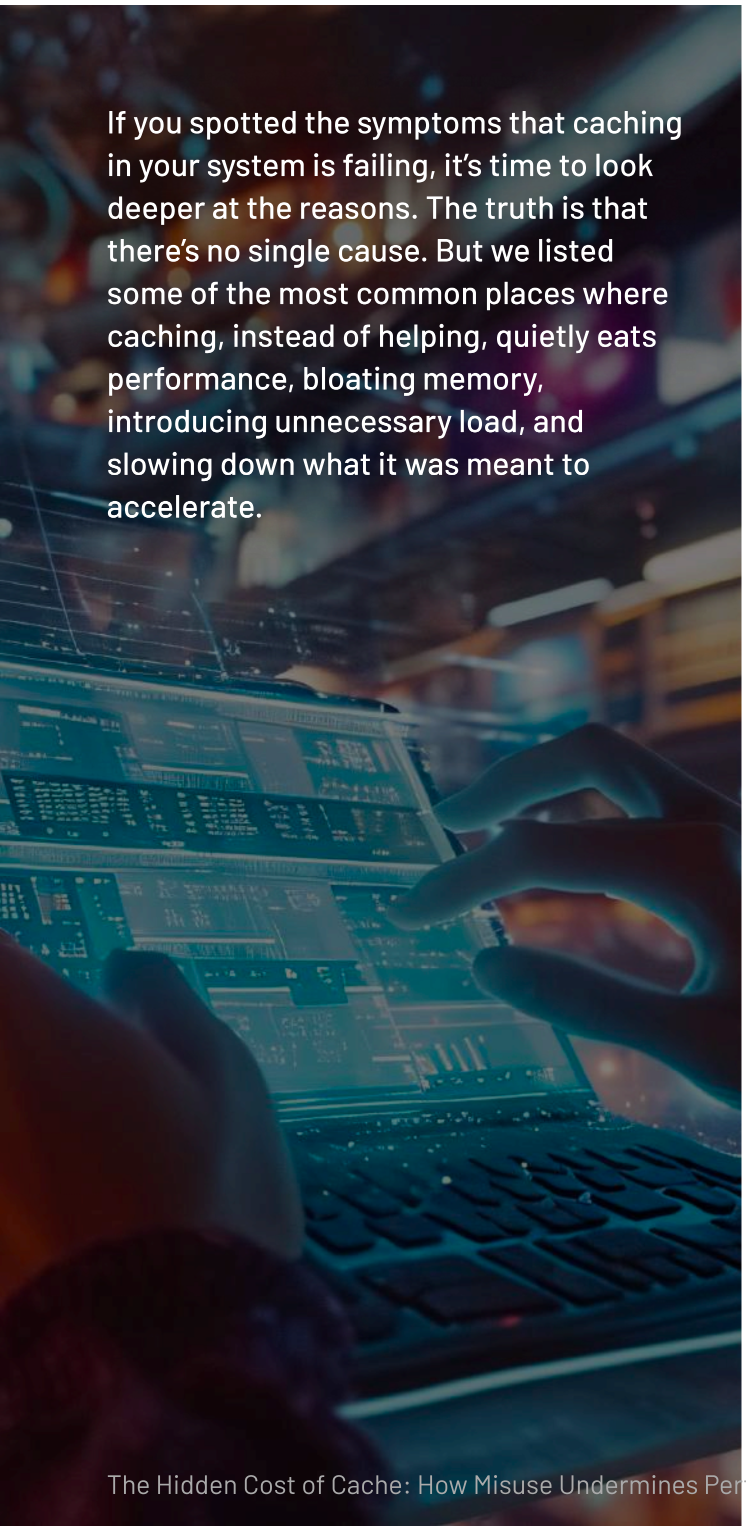
Database is still under load with caching enabled
Improper cache usage. e.g. cache stampedes or no pre-warming, may increase DB traffic instead of relieving it, hurting the performance.

Remember, a quiet system does not always mean a health system.
So, keep an eye out.

Smart systems need smarter thinking.
Find ours in your LinkedIn feed

JOIN! in

Where Cache May Eat Performance Instead of Improving It



If you spotted the symptoms that caching in your system is failing, it's time to look deeper at the reasons. The truth is that there's no single cause. But we listed some of the most common places where caching, instead of helping, quietly eats performance, bloating memory, introducing unnecessary load, and slowing down what it was meant to accelerate.

OVER-CACHING

If caching improves performance, then caching as much data as possible may seem like a smart move, especially in high-load systems where speed is king. But in this case, you risk ending up overarching, creating hidden pressure points:

- **Cache fills with low-priority objects**, pushing out actually important ones.
- **Serialization/deserialization time increases**, especially with complex objects that rarely need to be reused.
- **CPU cycles maintain entries no one reads**, increasing garbage collection frequency in JVM-based systems or memory churn in cloud-native environments.

Over-caching slows down the system by bloating the cache, polluting memory, and masking real performance issues. It gives a false sense of optimization and forces developers to find issues elsewhere, while the cache itself quietly becomes the bottleneck.

DISPROPORTIONATE GAINS

It's a common trap to push the cache too hard for minimal wins. Say you trim response time from 80ms to 50ms, which is invisible to users. Meanwhile, you may store heavy entries in memory, syncing data across nodes, and burning resources to chase a micro-optimization. Multiply that by thousands of requests per minute, and you're consuming serious infrastructure for barely noticeable user-facing improvement.

Here, you should also remember that every kilobyte in cache has a cost. If it's not improving speed, it's just squatting in your memory.

Performance gains should always be proportional to what they cost. Otherwise, you're scaling inefficiency, not velocity. Always weigh the benefit: if you're not gaining real responsiveness or reducing real load, that cache might be eating more than it feeds.

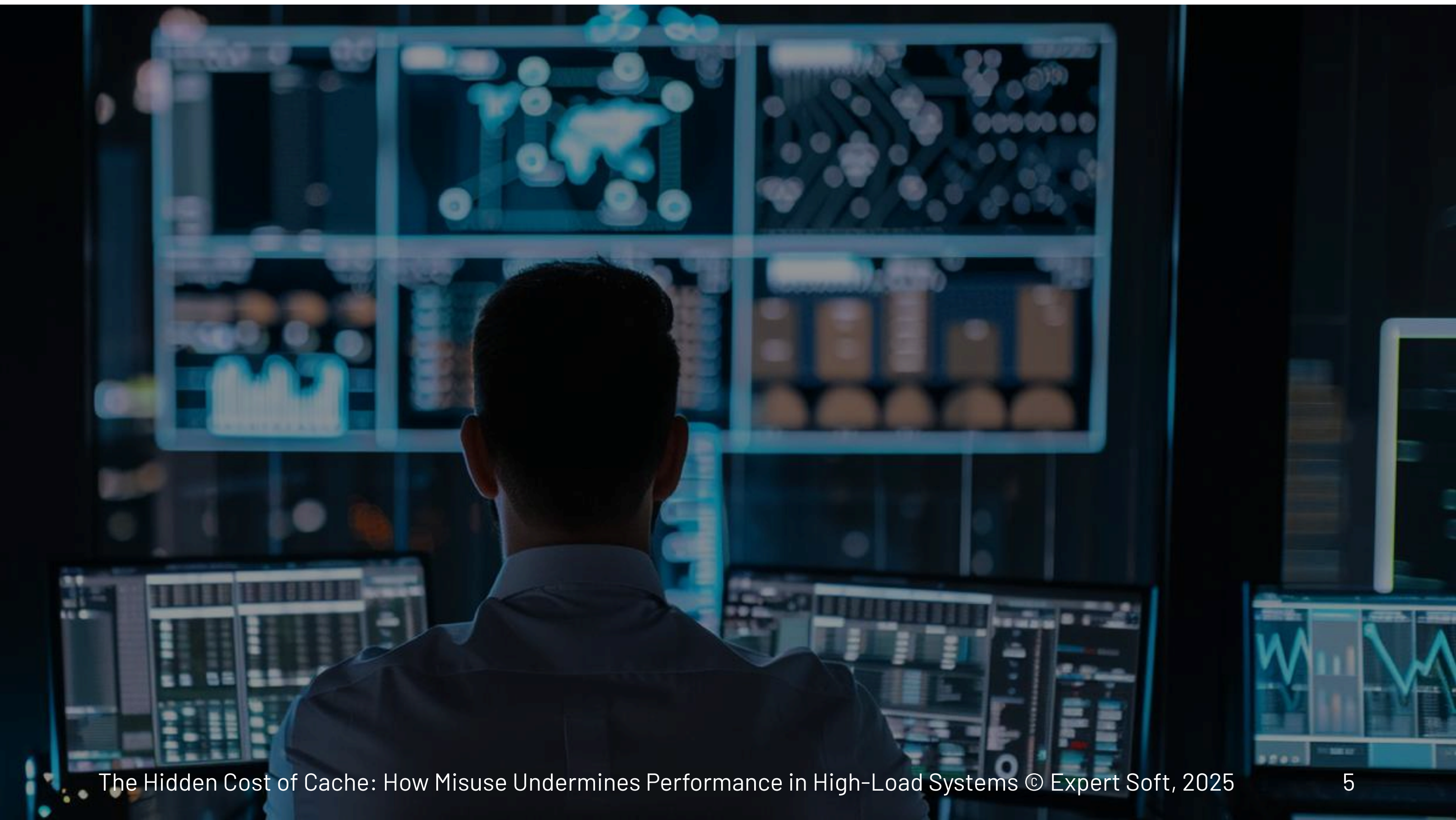
HEAVY CACHE TRAFFIC

When caching is too aggressive or misaligned with system architecture, it can increase the load, not reduce it.

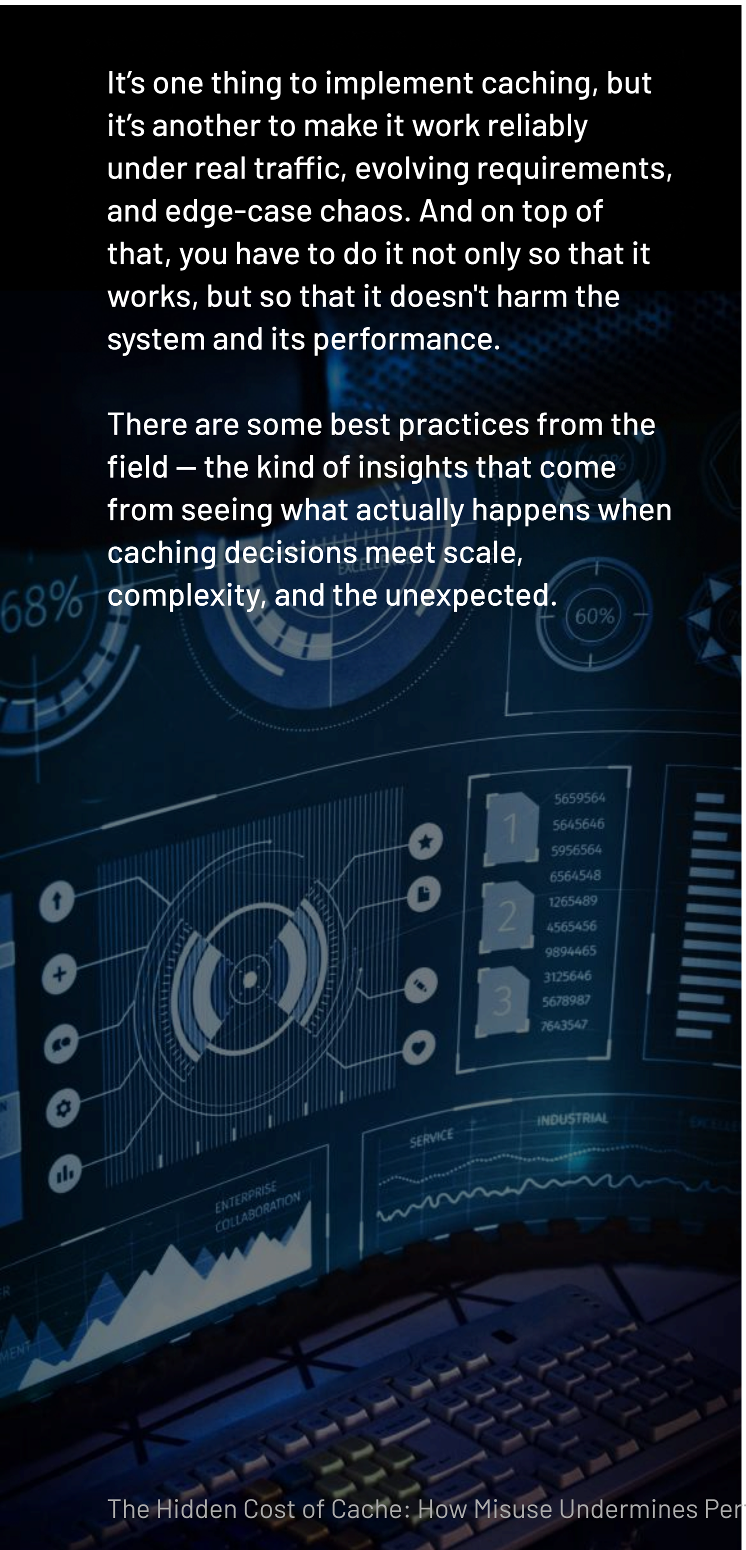
In setups with Redis or Memcached, frequent cache updates can flood the network with internal calls. Instead of offloading the system, your cache layer becomes the busiest part of it. In distributed environments, it gets worse, as syncing cache across nodes adds replication delays and can break data consistency under load.

When your cache constantly updates, it acts more like a real-time database with all the latency and none of the control.

We've seen systems slow down not because of missing cache, but because of too many cache operations. It's a case of "optimized to death", where the caching layer eats more performance than it saves.



Cache and Performance: Best Practices from the Field



It's one thing to implement caching, but it's another to make it work reliably under real traffic, evolving requirements, and edge-case chaos. And on top of that, you have to do it not only so that it works, but so that it doesn't harm the system and its performance.

There are some best practices from the field — the kind of insights that come from seeing what actually happens when caching decisions meet scale, complexity, and the unexpected.

DON'T JUST CACHE THE MAIN QUERY — LOOK AT NESTED DATA

It's common to cache direct queries, such as fetching a category by ID or loading static reference data. But **what often gets missed is how that same static data gets reused deep inside other operations.**

We've seen this during large-scale indexing, where product records include breadcrumb-style category paths in multiple languages. Categories are relatively static and already cached, but when each product reconstructs its full category tree, it triggers redundant lookups, often bypassing the cache altogether.

While it seemed like "we're already caching categories", in practice, that doesn't cover the full data flow.

> **From the field:**
> By caching shared category metadata used inside product indexing, we cut processing time by a factor of 30.

Caching shouldn't stop at the entry point. Trace how data flows through the system because even static dependencies can turn into dynamic bottlenecks.

KNOW WHEN TO LOOK BEYOND CACHE

Effective caching decisions come from understanding how the system really behaves. There are no universal rules that apply everywhere.

From the field:
The ability to step back, understand the business logic, and identify which data truly needs to be fast — in the context of that specific system — is the sign of true engineering maturity.

And sometimes, it means looking beyond traditional caching and turning to more specialized solutions.

For example, in a typical ecommerce catalog tightly integrated with an ERP, product availability and pricing shift constantly across stores, regions, or time slots. In this case, caching volatile data at the application layer becomes more of a liability than an optimization.

A search engine with built-in caching and index-level freshness tracking can act as a middle layer, serving current data without overloading the database or relying on manual invalidation.

This kind of setup goes a step beyond conventional caching. The search engine functions as both an index and a smart cache, selectively returning fresh results without reprocessing everything. It's not the most straightforward path, but when data changes frequently and at scale, it's often the only one that holds.

BUILD FOR INVALIDATION FROM THE START

One of the most common sources of stale data isn't the cache logic itself, but how the system handles changes outside that logic.

When data is modified directly in the database without notifying the cache, and developers are left to trigger invalidation manually, inconsistencies start to build up.

This typically happens when cache invalidation is treated as an afterthought. There's no internal tool or automated process to update the cache when underlying records change. And practice shows: human error is inevitable. One overlooked step — often during a rushed fix or urgent update — is enough to leave users with outdated content.

From the field:
If a fully automated invalidation system is not in your project scope, introduce a small internal tool that pairs DB updates with cache refreshes to avoid manual intervention.

Adding a cache is easy. Invalidating it — that's the real challenge. And it needs to be addressed from the start, not bolted on later.

Solutions like this demand deep system insight and engineering capabilities, exactly what we at Expert Soft bring to enterprise ecommerce development.
Explore how

DON'T CACHE WHAT'S ALREADY FAST

Not everything that's reused needs to be cached. And not everything that can be cached is worth it.

Rushing to add caching to parts of the system that are already fast, such as lightweight calculations or basic string operations, at best, adds overhead. At worst, it introduces unnecessary complexity without any performance gain.

Take prebuilt product URLs, for example. In many systems, caching them isn't needed at all as those operations are typically fast and already optimized. But that assumption doesn't always hold. In Java, string template matching comes with its own cost due to language-level specifics.

> **From the field:**
> In one project, caching prebuilt product URLs significantly improved performance not because the logic was complex, but because Java specifics made it worth it.

The point isn't whether URL caching is "right" or "wrong". It's that caching decisions need to be grounded in how your stack behaves under load as what's unnecessary in one technology can be essential in another.

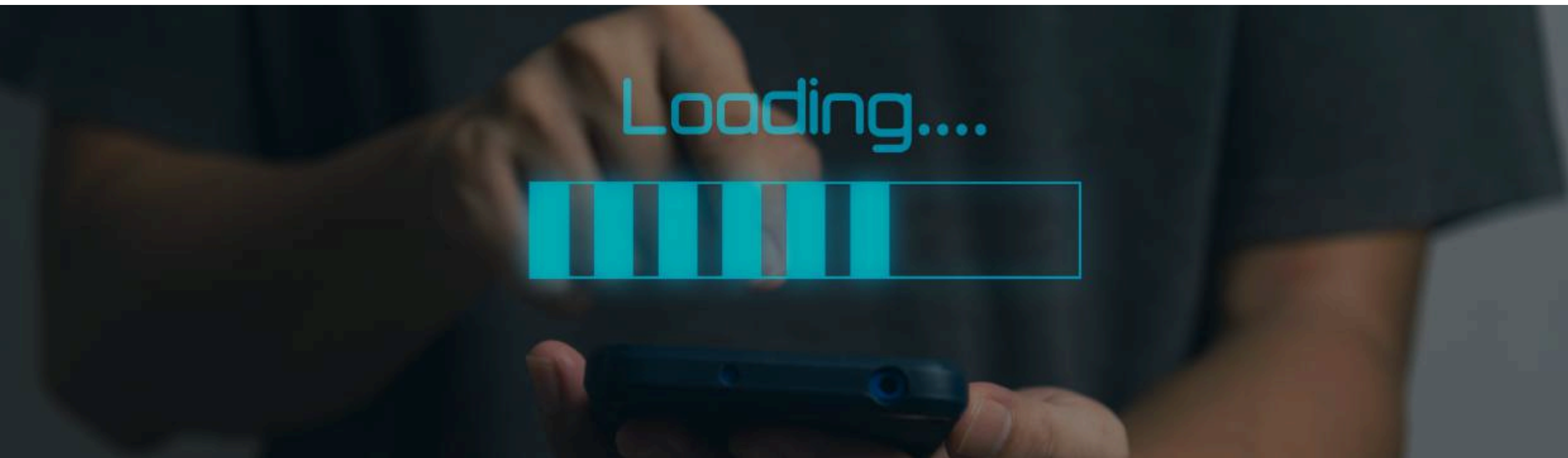
BUILD TTLS INTO REAL BEHAVIOR

Setting a TTL on a cache entry might feel like a way to ensure data doesn't overstay its welcome. But in high-load systems, TTLs often don't work the way teams expect.

In practice, **frequently accessed data can keep getting refreshed before the TTL expires.** The entry stays in memory indefinitely, never triggering the eviction you planned for. What was intended as temporary storage turns into a permanent memory resident.

> **From the field:**
> In many systems, TTLs are configured but rarely tested under load. They often don't fire as expected due to steady reuse.

That's why TTLs should never be set and forgotten. They need to reflect not just data volatility, but actual traffic behavior. And just like any part of your caching logic, they need to be verified in staging and monitored in production.



TEST CACHE LOGIC AS PART OF CI/CD

Caching is often treated as a helper layer — something that improves speed but doesn't need to be tested like core logic. But in reality, **caching can introduce subtle bugs: outdated data, inconsistent responses, and edge cases that only show up after deployment.**

Cache issues can arise when key formats change, TTLs don't behave as expected, or invalidation logic quietly breaks. And if that logic isn't covered in automated tests, those problems reach production where they're harder to debug and more expensive to fix.

From the field:
Including cache behavior in CI/CD with validating key generation, TTL expiration, and invalidation rules, helps catch stale data bugs before release.

Testing the cache like any other part of the system through integration and scenario tests in the pipeline ensures that it does what you expect, even as the application evolves.

ALWAYS MONITOR YOUR CACHE

Once a cache is in place, it's easy to forget about it, allowing it to live unnoticed and possibly hurting system performance. That's why monitoring isn't optional. Without visibility, it's impossible to know when your cache is bloated, stale, or doing more harm than good.

You should be tracking more than hit/miss ratios. **Volume, memory usage, freshness of data, and eviction behavior all offer crucial insight into how your cache behaves under load.** When these metrics are ignored, the cache grows unchecked, living in RAM that's often more expensive and more limited than you may think.

From the field (observation):
In our projects, we use tools like DataDog to catch memory pressure and stale data issues before they degrade system performance or inflate cloud costs.

Monitoring gives you the data to right-size your cache, detect early failures, and keep caching aligned with actual performance needs.

Knowing When Not to Cache: The Mark of Engineering Maturity

The examples and best practices in this whitepaper show how excessive or poorly implemented caching can hurt performance and system stability. From planning invalidation early to monitoring memory usage and eviction cycles — caching demands attention at every step. But here's one more principle: **sometimes, the smartest move is not to cache at all.**

Take modern cloud infrastructure. Fully managed databases from top cloud providers are already optimized for fast, repeatable reads. In these environments, adding a custom cache layer can actually slow things down. Serialization, deserialization, memory pressure — all of it adds overhead with little to no real benefit.

**Caching doesn't guarantee improvement.
In some stacks, it becomes the bottleneck.
Engineering maturity means knowing when to walk away from it.**

Real-world example

On one of Expert Soft's projects, caching was added to speed up access to frequently queried data. But the invalidation logic was flawed: it didn't trigger as expected, leading to stale and even deleted records being served. The right call was to remove the cache entirely.

After all, what determines whether caching is worth it? Context. Data volatility, read/write ratios, infrastructure specifics, and consistency requirements all shape what kind of caching — if any — should be used.

That's how we approach caching strategy at Expert Soft: with sharp attention to context, load behavior, and business-critical data flows. Because the best systems aren't just fast — they're fast for the right reasons.

About Expert Soft

Expert Soft is a targeted ecommerce software delivery company, partnering with Fortune 500 companies and global corporations across the US and EU. With SAP Commerce Cloud and Java as our backbone, we know how to ensure scalable and high-performing solutions that can handle 1 mln requests per second, delivering a smooth customer experience.

Developing a payment engine that saved our client about $100 million in operational expenses, ensuring multi-country platform support, adapting solutions for new market entry with tailored enhancements — these are just a few of the challenges our specialists tackle.

We aim to deliver more than a software system. We aim to deliver tailored solutions that maximize profitability within available resources. Our success is driven by:

- All our engineers have a university background
- Specialists excel their skills in our training LABs
- Perfect English skills
- Ready to help 24/7

CLIENTS

We work with corporations around the world with revenue of over $20 billion and 150K+ employees.

APPROVALS BY AUDITS

Our ongoing work with corporations is consistently validated through rigorous audits, both by internal teams and Big 4 consulting firms.

HIGH-LEVEL SECURITY

Approved by assessments from global companies, who are leaders in their respective industries.

BUDGET EFFICIENCY

By carefully aligning technology investments with your business goals, we ensure optimal value and cost-effectiveness.

PROFESSIONAL TEAM

No offshore outsourcing and our team's average tenure of 4+ years means you get seasoned problem-solvers, not just coders.

EXPERT SOFT EXCELS IN

- PAYMENT ENGINE
- MICROSERVICES ARCHITECTURE
- CONTENT MANAGEMENT
- REDESIGN
- E-COMMERCE PLATFORM
- HEADLESS COMMERCE
- MICRO UI FRONT-END
- MIGRATION&INTEGRATION

OUR TECH CORE

FRONT-END
HTML, CSS, JavaScript (Angular, React, Vue, Next, TypeScript, Jquery), Spartacus

BACK-END
Java EE, Spring, SAP Commerce (Cloud), Node.JS.

DEVOPS
Docker, Kubernetes, CI/CD

UX/UI DESIGN
UX Research, UI Design, Figma, Adobe, Sketch

QUALITY ASSURANCE
Manual Testing, Test Automation

TARGETED DOMAINS

SHARED PATHS, LASTING ECOM VICTORIES

LET'S TALK SOLUTIONS!

EKATERINA LAPCHANKA
Chief Operating Officer
kate.lapsenco@expert-soft.com

+1 585 4997879

+371 25 893 015

PAVEL TSARYKAU
CEO & Founder
of Expert Soft

Let's connect

expert-soft.com

LinkedIn