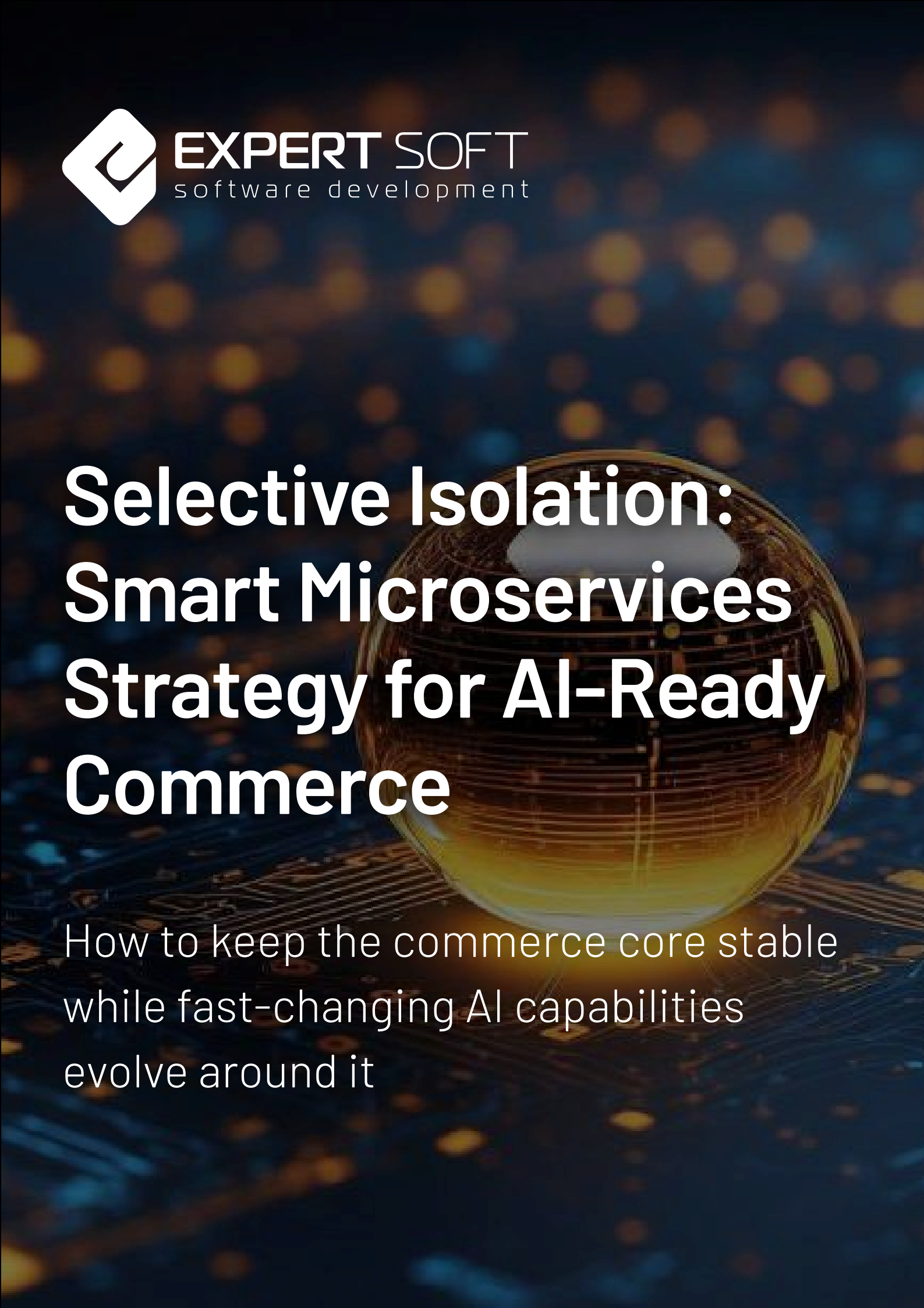


EXPERT SOFT
software development

Selective Isolation: Smart Microservices Strategy for AI-Ready Commerce

How to keep the commerce core stable while fast-changing AI capabilities evolve around it

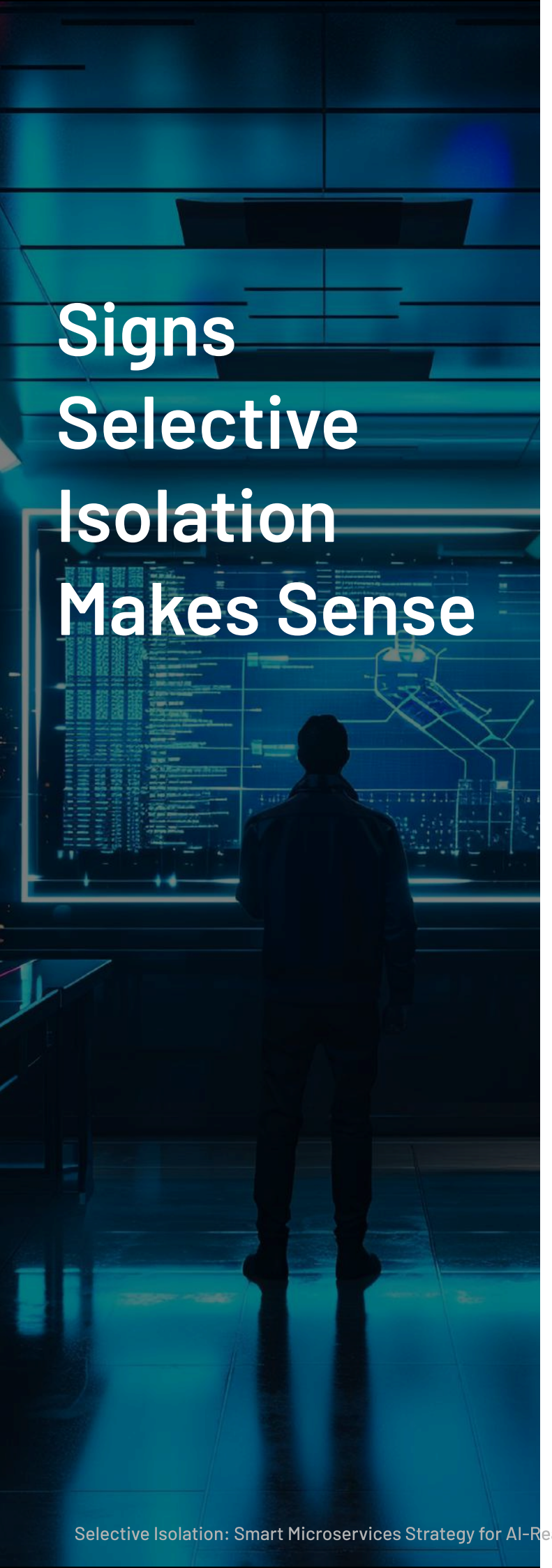
Enterprise architecture has spent years swinging between two extremes: either the monolith keeps taking on more than it should, or teams split the system too aggressively and end up paying for that complexity elsewhere. Neither approach holds up especially well over time.

What is emerging instead is a middle path: selective isolation. The idea is to separate only those parts of the system whose rate of change, dependencies, scaling needs, or operating rules differ meaningfully from the core.

This matters even more as enterprises add AI into commerce platforms, because AI-enabled capabilities often bring a different release rhythm, cost model, and set of operating rules than the platform around them.

In this white paper, we look at two practical questions. First, how to recognize the domains that should be isolated in the first place. Then, how this approach becomes useful with AI-driven capabilities inside a large enterprise commerce environment. To make the idea concrete, we'll look at Expert Soft's case, which shows how teams can keep the core platform stable while giving AI-related logic its own space to evolve.

Table of Contents



Signs Selective Isolation Makes Sense

The shift toward selective decomposition is already visible in enterprise commerce projects. Among our clients, we see more teams moving specific responsibilities into separate microservices without trying to rebuild the entire platform around them.

A good example is a large retail and health and beauty company running multiple ecommerce projects on a shared SAP Commerce Cloud codebase. For this kind of environment, turning the whole platform into a microservice landscape would be expensive, risky, and hard to justify. Still, some responsibilities clearly benefit from being separated from the core.

That's where the next question arises. If selective isolation works, what exactly should move out, and where should the line be drawn? There are usually a few signs.

SIGN 1. THE DOMAIN CHANGES FASTER THAN THE CORE PLATFORM

Some parts of a commerce platform need predictable change. Checkout, order creation, catalog integrity, and other core flows usually fall into this category. Other areas move differently: business rules shift, integrations evolve, experiments need adjustment, and teams cannot wait for a full platform release every time.

Why it matters

When fast-changing logic stays in the core, the stable part of the platform starts depending on something more volatile. Small domain changes begin to slow delivery and make the core more fragile.

What to look for

Look for local changes with platform-wide impact. If a narrow business adjustment triggers broad regression testing and cross-team release coordination, the boundary may be in the wrong place.

SIGN 2. CORE LOGIC IS EXPOSED TO EXTERNAL SYSTEM CHANGES

The signal is when external integration formats, limits, failures, or release cycles start shaping core platform logic. When adapters and conversion layers are too thin, unstable provider behavior leaks into the part of the system that should work through stable internal contracts.

Why it matters

When external variability reaches the core, every provider change becomes a platform concern. The core has to absorb behavior it does not control, instead of working through a stable internal contract.

What to look for

Watch for integration changes that force updates inside core logic. If a provider format shift, API limit, or third-party failure requires urgent platform fixes instead of adapter-level changes, the boundary is probably too deep.

SIGN 3. THE DOMAIN HAS ITS OWN COST LOGIC

Not every platform cost grows the same way. Some follow traffic and infrastructure load, while others come from API calls, tokens, third-party tiers, generated content, or refresh frequency.

Content and digital assets show the pattern well: CDN usage, media processing, transcoding, and storage can add pressure when tied too closely to the core database and runtime.

Why it matters

When this cost and load profile is hidden inside the core, teams pay twice: through higher operational costs and through performance degradation in the platform that should stay focused on commerce flows.

What to look for

Look for domains where storage, processing, refresh frequency, or provider usage grows faster than the core can comfortably absorb. If keeping the logic inside the platform increases database pressure, infrastructure spend, or performance limits, the boundary may need to move.

A strong boundary decision still needs disciplined implementation. Explore microservices best practices for building scalable and maintainable systems.

SIGN 4. THE DOMAIN NEEDS ITS OWN FALLBACK LOGIC

Standard platform metrics do not always show domain-specific issues. Some domains need signals for freshness, provider latency, retries, partial failures, and fallback behavior.

Why it matters

Without domain-specific visibility, teams can miss the real failure mode. The platform may look healthy while one capability quietly returns stale, incomplete, delayed, or degraded output.

What to look for

Look for domains where general platform monitoring misses the key health signals. If teams need custom dashboards, alerts, or fallback paths, the boundary may need to be clearer.

SIGN 5. THE DOMAIN CARRIES A DIFFERENT RISK PROFILE

Some capabilities can fail softly. Others sit close to checkout, payment, inventory, customer data, or compliance-sensitive flows, where the same release and recovery model can create the wrong trade-off.

Why it matters

A shared risk model makes teams either too cautious with low-risk capabilities or too casual with high-impact ones. Both outcomes are expensive: one slows innovation, the other increases the blast radius.

What to look for

Look for mismatched failure tolerance. If one part of the system can degrade without hurting the business flow while another must stay predictable under all conditions, they should not be governed by the same operational rules.

SIGN 6. NEW FEATURE DELIVERY BECOMES DISPROPORTIONATELY EXPENSIVE

A single release path can be useful while the platform is still easy to reason about. It keeps dependencies visible and reduces the risk of incompatible components. Over time, though, every new feature can start carrying too much baggage: broad regression testing, hidden dependency checks, and coordination with the full platform delivery cycle.

Why it matters

Many business ideas need fast validation before serious investment. If even a small experiment requires heavy QA, long release planning, and platform-wide verification, the cost of learning becomes too high, and promising features may never get a fair chance.

What to look for

Watch for features where the implementation is small, but the release effort is huge. Rising QA effort, longer release cycles, and growing verification costs often show that the domain is too tightly tied to the core.

Taken together, these signs point to the same architectural pattern: some domains no longer fit comfortably inside the core because they operate by different rules. They may still depend on the platform, but they should not necessarily live inside it. AI makes that question hard to ignore. It may appear on the surface as another storefront feature, but its operating logic often lives by a different playbook.

Why AI Capabilities Often Need Selective Isolation

AI is not a reason to redesign the whole commerce platform. In most enterprise environments, that would only replace one kind of complexity with another. The point is more specific: many AI-enabled capabilities already carry the same traits that make a domain difficult to keep inside the core.

Model and prompt logic move faster than platform logic

AI behavior often needs tuning after it meets real traffic, real content, and real customer behavior. Model choices, prompt structure, provider APIs, and output rules can change faster than the core platform planning cycle.

> **Operating implication:** AI logic needs room for controlled iteration; otherwise, teams either slow down every AI improvement or increase the release pressure on the commerce core.

AI costs follow different mechanics

Commerce platform costs usually follow traffic, caching, compute, and storage. AI adds another layer: inference, tokens, provider calls, generated content volume, and regeneration frequency.

> **Operating implication:** teams need explicit rules for when AI should run, how often it should refresh, and when the cost is justified.

AI requires limits on data access

Not all data is suitable for feeding into the AI engines. Personal information, billing data, unique business-specific rules - these types of data have to be unavailable for the AI engine.

> **Operating implication:** certifications and data governance goals are easier to achieve in case of isolation and explicit protocol control. This builds trust in AI usage and prevents concerns from all security-related stakeholders.

AI output needs policy

Standard platform features usually depend on deterministic logic. AI capabilities often need thresholds, output checks, routing rules, fallback behavior, and decisions about when new output is better than stable existing output.

> **Operating implication:** the architecture needs the understanding of when to refresh, when to keep an existing result, when to block or review output, and how to trace the decision later.

When these traits are kept inside the core, AI starts competing with the platform for release attention, cost visibility, and operational control. Selective isolation changes that setup: the AI domain can still serve the commerce flow, but the rules around its behavior no longer have to live inside the core.

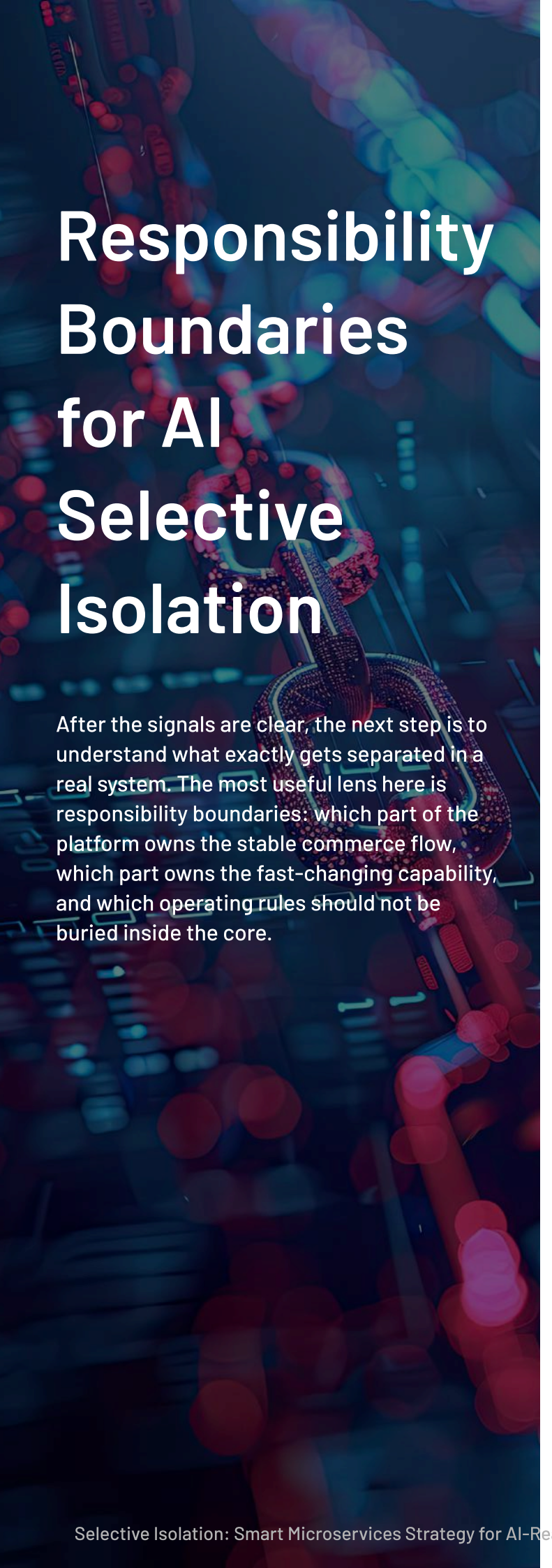
WHAT SELECTIVE ISOLATION ENABLES FOR AI CAPABILITIES

The value of isolation is not the physical separation itself. A separate service is only useful when it gives teams better control over decisions that were previously buried in platform logic.

- **A release rhythm that matches the AI domain.** AI logic can be tuned, tested, and improved without turning each adjustment into a core platform release.
- **Staged data processing with clear quality gates.** Teams can clean, enrich, validate, or skip data before AI generation, keeping weak inputs out of the flow and avoiding wasted processing costs.
- **Cost logic that can be governed closer to the source.** Instead of treating AI spend as a side effect of platform traffic, teams can manage thresholds, frequency, reuse, and provider usage as part of the AI domain itself.
- **Observability that covers performance and responsible AI signals.** Monitoring can track technical health, output quality, usage patterns, and fallback behavior in one AI-specific view, instead of burying those signals in generic platform metrics.
- **Safer degradation when the AI layer is unavailable.** The storefront can keep working even if an AI capability is delayed, skipped, or temporarily disabled, because the core flow does not depend on the AI layer to complete its job.

These controls need to live in the architecture, not just in governance docs. That means clearer lines between the commerce platform, the AI capability, and the logic that decides when AI should run.

Keeping AI outside the core solves only part of the problem. Discover how prepared data layers help enterprises make AI outputs more reliable.



Responsibility Boundaries for AI Selective Isolation

After the signals are clear, the next step is to understand what exactly gets separated in a real system. The most useful lens here is responsibility boundaries: which part of the platform owns the stable commerce flow, which part owns the fast-changing capability, and which operating rules should not be buried inside the core.

A **practical example** from Expert Soft's delivery experience helps make those boundaries more concrete. The case comes from a large retail and health and beauty company running several ecommerce projects on a shared SAP Commerce foundation. The setup is familiar for enterprise commerce: one core platform, multiple business units, and a shared codebase.

CONTEXT

The company wanted to make product reviews easier to scan on the product detail page. To enable that, our team introduced an **AI-generated pros-and-cons summary based on customer reviews.**

The flow was designed so that SAP Commerce would not become the place where review interpretation, prompt logic, and LLM interaction live.

Review data was collected and passed into a **dedicated AI microservice**, where the summary was generated outside the core platform. SAP Commerce then worked with the result as prepared content: it received the summary, linked it to the right product context, and made it available for storefront rendering.

This setup gives us a practical way to look at responsibility boundaries. The feature depends on the commerce platform, but not every part of the feature belongs inside it. The boundaries below show how the stable customer-facing flow, AI generation, and AI operating rules can be separated without disconnecting the feature from the platform it serves.

BOUNDARY 1. CUSTOMER-FACING FLOW VS. AI GENERATION

The first boundary appears between the part of the system that serves the customer experience and the part that creates AI-generated content. In a setup like this, the platform does not need to own the internal mechanics of generation to use the result effectively. Instead, it needs a reliable way to receive, store, and present the output in the right customer context.

That matters because AI generation may need frequent refinement, while the product page experience should remain predictable.

How this works in the case
Product reviews are sent to a separate AI microservice, which generates a summary with key pros and cons. SAP Commerce does not interact with the model directly. It receives the prepared summary and renders it on the product detail page.

What changed for the team
The team could introduce AI-generated summaries without turning the commerce core into the place where model behavior, prompt logic, and generation rules had to live.

Why this split matters
This keeps the platform focused on customer experience continuity, while the generation layer can be tuned, replaced, or improved without treating every AI-side change as a platform change.

BOUNDARY 2. AI CAPABILITY VS. AI OPERATING CONTROL

The second boundary appears between the capability itself and the logic that decides how this capability should be used. Generating a summary is one responsibility. Deciding when generation is justified, which signals should trigger it, and what rules make the result suitable for use belongs to another layer.

In AI-enabled flows, that second responsibility matters because every regeneration can affect cost, freshness, and customer-facing quality.

How this works in the case
Review counting and refresh logic were moved into an Azure Function. It tracks review counts from external providers, stores them separately, and triggers regeneration only when a meaningful threshold is reached.

What changed for the team
The team gained a way to control AI costs before they turned into background platform spend. Instead of regenerating summaries every time new review data appears, the system can wait until the change is significant enough to justify another AI call.

Why this split matters
Economic and operational rules are what make AI usage manageable at scale: when generation is worth the cost, how often it should run, and what signals should trigger it. This boundary gives those rules a dedicated place instead of letting them disappear into the commerce core.

BOUNDARY 3. TRANSACTIONAL CORE VS. CHANGE-HEAVY AI POLICY

This boundary separates predictable platform logic from AI policy that will likely change after launch. The core should preserve commerce continuity, product and page integrity, and stable storefront behavior.

AI policy is less settled by nature. Thresholds may shift, provider behavior may change, output rules may need tuning, and traceability may need to show which reviews, prompt version, and generation conditions produced each summary.

How this works in the case

The team kept AI refresh policy outside SAP Commerce instead of embedding it into platform logic. The Azure-based control layer became the place to adjust regeneration rules and track the context behind each summary without changing the transactional core.

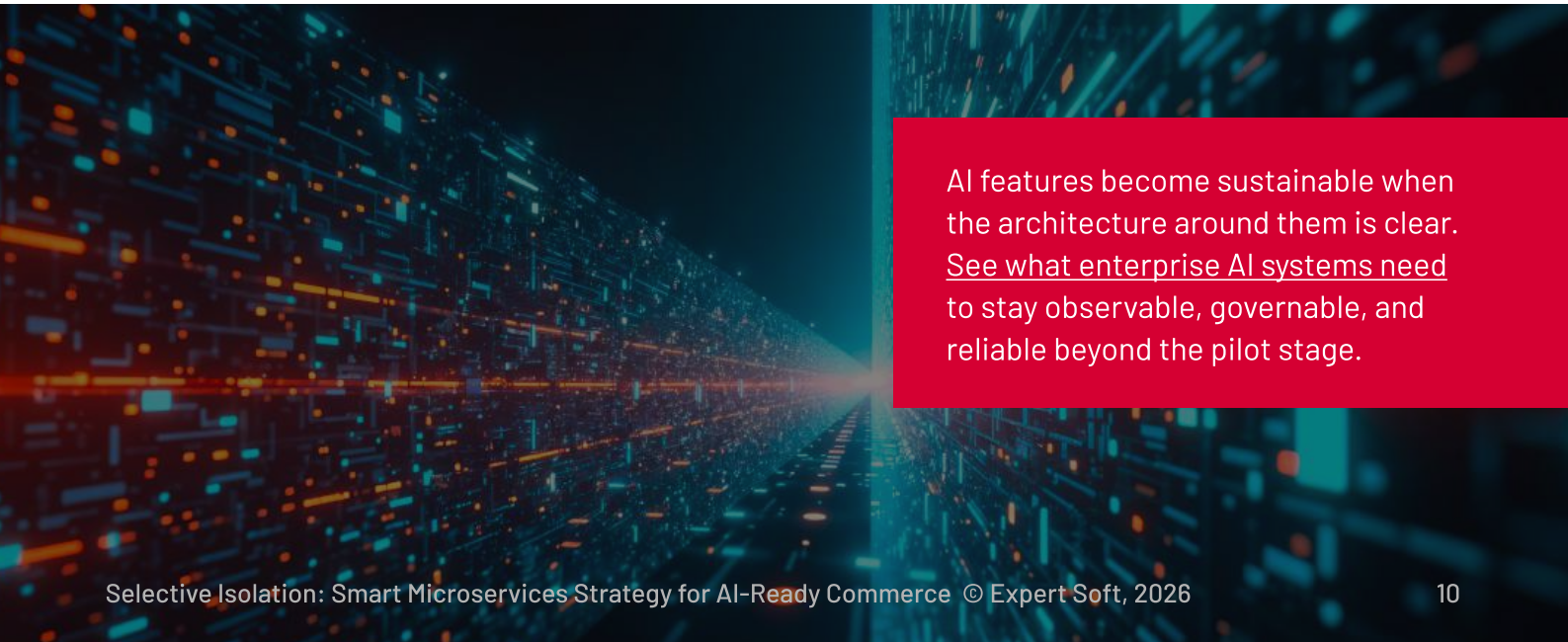
What changed for the team

AI policy became easier to refine after launch, while SAP Commerce stayed focused on stable commerce responsibilities. The team also gained a clearer place to manage summary context outside the core platform flow.

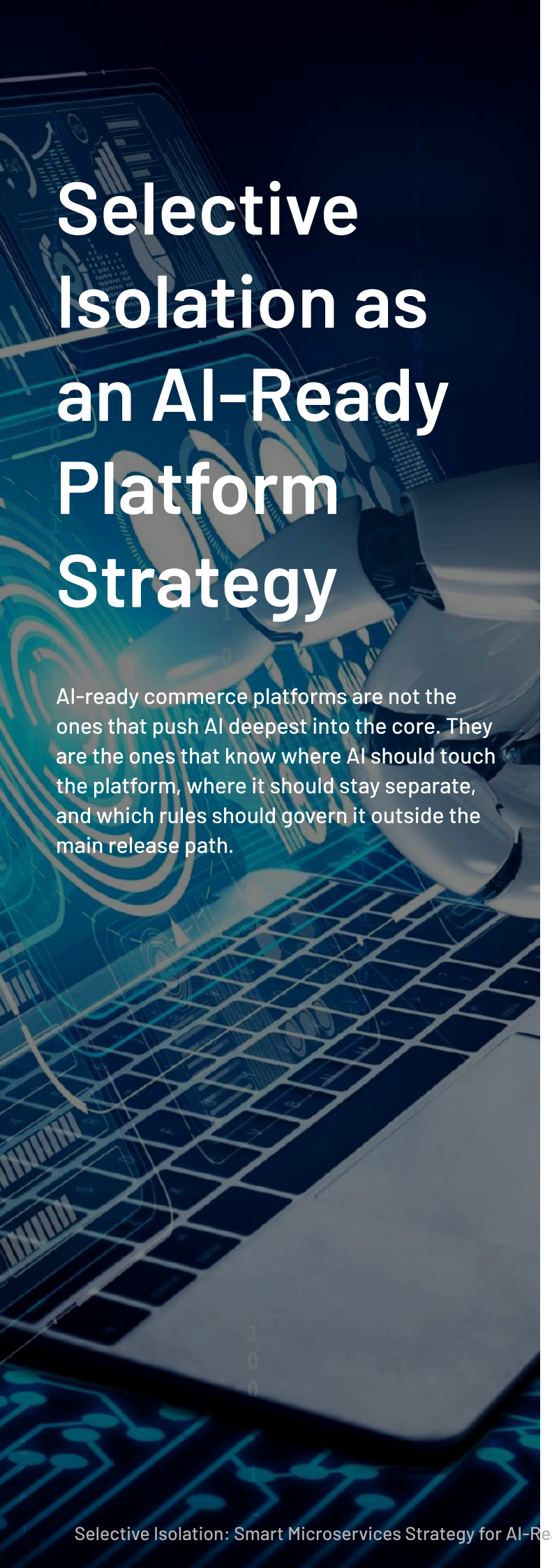
Why this split matters

AI features rarely arrive fully settled. Keeping their policy layer outside the core gives teams room to adapt thresholds, refresh logic, and traceability requirements as real usage, provider behavior, and governance expectations become clearer.

The case shows why selective isolation should be read through responsibility, not only through deployment. Once that logic is clear, the same approach starts to look bigger than one feature or one service. It becomes a way to design platforms that can keep absorbing fast-changing capabilities without letting the core lose focus.



AI features become sustainable when the architecture around them is clear. See what enterprise AI systems need to stay observable, governable, and reliable beyond the pilot stage.



Selective Isolation as an AI-Ready Platform Strategy

AI-ready commerce platforms are not the ones that push AI deepest into the core. They are the ones that know where AI should touch the platform, where it should stay separate, and which rules should govern it outside the main release path.

That is the practical role of selective isolation. It gives teams a way to add AI capabilities without turning the core into a container for every model change, provider adjustment, refresh threshold, and cost-control rule. For enterprise commerce teams, this comes down to three platform conditions.

Condition 1. The core stays focused on stable commerce behavior

The core should keep owning the flows where predictability matters most: product context, checkout continuity, order creation, page behavior, and critical integrations. Selective isolation keeps fast-changing logic close enough to serve these flows, but separate enough not to slow them down.

Condition 2. Volatility gets a defined place to live

AI brings volatility around models, providers, token costs, refresh expectations, and output behavior. When that volatility has its own boundary, teams can tune and monitor it without pushing every adjustment through core platform processes.

Condition 3. AI operating rules become explicit

AI capabilities need clear rules for when generation should run, when existing output is enough, how refresh should be triggered, and how spend should be controlled.

Selective isolation is not just a microservices pattern. It is a way to keep enterprise platforms ready for capabilities that change faster than the core should. The same thinking can apply to catalog enrichment, search support, merchandising logic, and other domains where value depends on fast iteration but the platform still needs stability.

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:

TEAM STRENGTHS

- **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 FRONTENDS
- 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