Strangling your Monolith:

Bypass ‘Big Bang’ Replatforming with Microservices

FOR MORE INFORMATION
marketing@skava.com | 877·554·2176 | www.skava.com

skava®

Table of contents

INTRODUCTION What every digital enterprise wants 3

SECTION 1 Signs you've outgrown your commerce platform 4

SECTION 2 Why no enterprise wants a Big Bang 5

SECTION 3 Bypass the Big Bang with the Strangler Pattern 6

- The benefits of microservices
- Monolith vs. microservices: at a glance

SECTION 4 The Strangler Pattern in action 10

- Case study: Best Buy
- Case study: Staples

SECTION 5 Frequently asked questions 12

- What's the difference between SOA and microservices?
- How are microservices different from headless commerce?
- Microservices: build or buy?

SECTION 6 Let's recap 15

skava®

INTRODUCTION

What every digital enterprise wants

Both IT and business teams want to remove friction and be more agile in releasing code and bringing new features to market.

In today's fast-moving digital world, closing customer experience gaps, creating internal efficiencies, and staying ahead of the market matters.

For many digital enterprises, legacy technology hinders progress more than it helps. Years of integrations and incremental business requirements can turn a once best-of-breed commerce system into a tangled architecture of half-solutions, patches and hacks. Developers have come and gone and documentation is out of date, if it ever existed at all. Some areas of the platform haven't been touched in a decade and technical debt haunts the whole system.

This makes new feature delivery slow.

Business leaders are under pressure to delight the customer and hit revenue targets, while IT must balance new code delivery with keeping the platform up and running. Development teams must be aware and mindful not to compromise each other's work — breaking the build can take days to fix. Every change must go through extensive regression testing or be released at great risk. Releases happen monthly or quarterly, and in some cases annually.

Both IT and business teams want to remove friction in the process and be more agile in releasing new code and bringing new features to market. Often, this means a fresh start — a migration to a new commerce platform. But experienced business and IT leaders know ripping and replacing an enterprise commerce system isn't easy, and doesn't guarantee greener pastures. Many want to avoid a replatform project at all costs!

The good news is there's an alternative to both the pain of monolithic legacy architecture and Big Bang replatforming. Migrating to a modern, modular architecture that supports more agile delivery can be achieved with the Strangler Pattern, a systematic replacement of monolithic components over time with microservices. This incremental migration delivers critical functionality faster with minimal disruption to existing systems. The end result is not a new, heavy monolith that will ultimately suffer the same pains as the legacy platform, but a flexible and scalable architecture that supports agility and innovation.

SECTION 1

Signs you've outgrown your commerce platform

You're pushing your limits

Your business has grown, but success is slowing down your site and back end processes. Unacceptable response times or frequent outages are crippling conversion rates and revenue. Keeping pace with growth requires making a copy of your entire application and investing in more licenses or hardware to stay performant — even when increased load applies to only a single or handful of components within your platform.

You have new business requirements

Things have changed since your platform went live. Initiatives to expand to new markets, adopt new business models, pursue new revenue-driving strategies and lead innovation are hamstrung by your technology. You struggle to extend commerce to new touchpoints, add features to meet consumer expectations, match competitor offerings and embrace digital experience trends because your legacy platform was built for the past.

Development is too slow, costly and risky

Your legacy code has been so heavily customized over time, any change requires ample lead time, coordination between multiple development groups and potentially introduces bugs that can affect multiple components. IT spends as much time testing as developing new code, and applies more resources to maintenance than shipping new functionality.

Integrations are too expensive or inefficient

Your platform may do core commerce well, but hasn't kept pace with innovation in the industry. Adding third-party tools like personalization engines, advanced search and merchandising, artificial intelligence, digital experience management or replacing an ERP system can run six figures just to integrate, if you can find an efficient way to integrate them at all. Extending to new touchpoints like mobile apps, POS systems, in-store digital or Internet of Things is either cost prohibitive or too complex.

Your TCO is out of control

Like an automobile, an ecommerce platform can get to the point where it becomes a money pit to keep running. It is not uncommon for a mature application to require more development time in maintenance than innovation.

SECTION 2

Why no enterprise wants a Big Bang

Replatforming is no trivial endeavor, and leaving a legacy solution doesn't always mean improvement.

Historically, a Big Bang replatforming project (ripping your existing commerce application out of service and replacing a legacy monolith with a new monolith) has been the only way to escape the pain of a commerce system that is too costly to maintain or no longer serves the needs of the business. But replatforming is no trivial endeavor, and leaving a legacy solution doesn't always mean improvement.

Replatforming an enterprise commerce platform carries heavy up-front costs, interrupts the business and poses significant risk to your timeline and budget. Legacy systems have typically been heavily customized over time to suit your unique business cases. They also serve as critical systems of record and are deeply integrated with other key enterprise systems like ERP, CRM and even WMS. These customizations and integrations need to be replicated in the new system, and implementation can take months to years.

During the lengthy development and implementation phase, any enhancements to your digital strategy and feature offering need to be added to both your existing and replacement systems, duplicating development and regression testing efforts. Mid-implementation updates affect your project scope and delivery timelines, inflating your costs and cannibalizing your ROI. To meet deadlines and budget constraints, lower priority features may need to be cut or deferred to post-launch. Slipping your deployment date requires supporting your legacy system longer than you had anticipated, costing even more money.

After go-live, revenue often takes a hit as search engines reindex new URLs and customers reorient themselves to your new site. Business users have new tools to learn and IT has a new platform to manage. The new monolithic platform carries the same limitations around delivery speed, scalability and stability as the legacy application (until the next Big Bang).

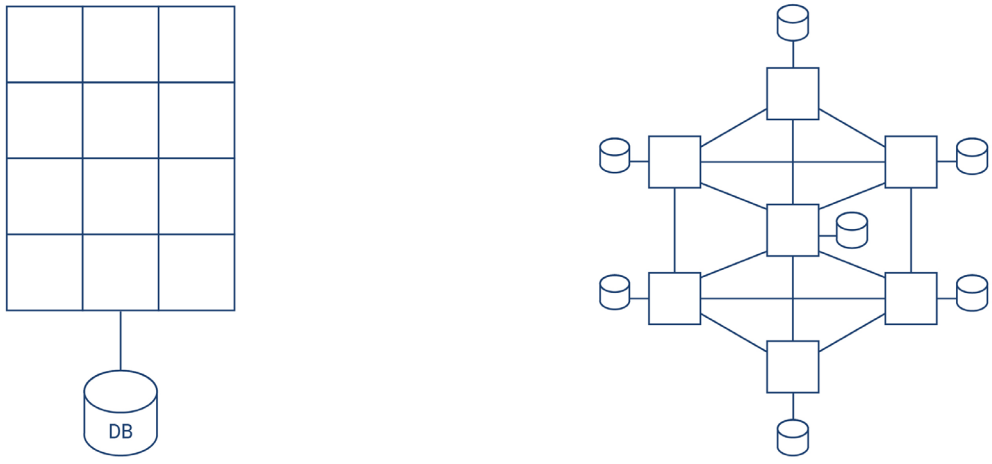
Some organizations discover too late they've chosen the wrong platform. Requirements diligence may not have been conducted thoroughly enough to uncover limitations of the platform, or development and integration issues may only become apparent during implementation. In some cases, decision makers' jobs are on the line if a project fails, and the organization must choose another platform or stay on painful legacy systems even longer.

SECTION 3

Bypass the Big Bang with the Strangler Pattern

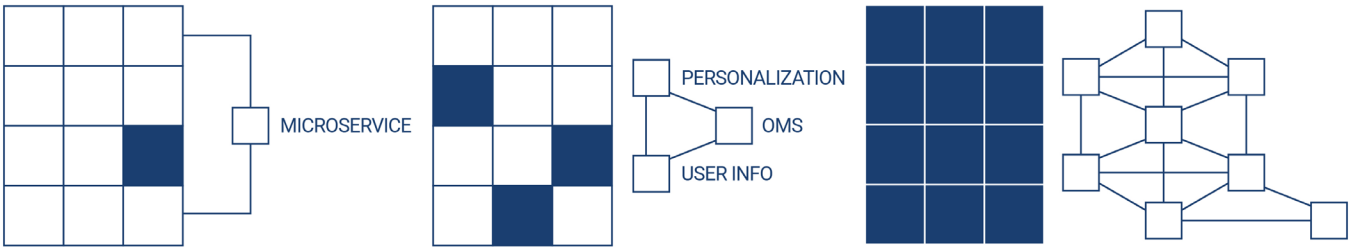
Until now, enterprise retailers had to choose between undertaking a rip-and-replace replatforming project or kicking the can down the road a little further and living with an outdated legacy system.

Migrating your monolithic platform to microservices offers an alternative to Big Bang replatforming. Microservices are standalone applications built around a single function or business process. Each microservice is independently deployable, uses its own dedicated database and has a well-defined API. For example, components like catalog, search, PIM, promotions, accounts, loyalty, order management, cart and checkout can all be decomposed into microservices, decoupled from each other.



Monolithic vs microservices architecture

With the **Strangler Pattern,** microservices can be built around a legacy monolith piece by piece, gradually replacing it over time.



*With the **Strangler Pattern,** microservices replace the monolith over time*

skava®

Coined by Martin Fowler, the **Strangler Pattern** concept is based on the metaphor of strangler vines. Growing out of the tops of fig trees, strangler vines work their way down to take root in the soil and eventually choke out their host (and along the way, weave themselves into some remarkable shapes that are a sight to behold).

In a digital commerce context, the **Strangler Pattern** represents an alternative to the complex and risky endeavor of rewriting or ripping-and-replacing an aging and inflexible legacy system. Microservices provide the modularity that allows an enterprise to gradually build a new system around the edges of the old over time until the legacy platform is no longer needed.

The **Strangler Pattern** minimizes disruption to existing operations and user experience, and allows you to deliver your most important system upgrades faster. As you extend your capabilities, your monolith won't get bigger or more complex.

Rather than replace one monolith for another (which over time will ultimately become as heavy and expensive to maintain as your existing platform), transitioning to a modular architecture lets you take advantage of the efficiency, stability, scalability and flexibility of microservices.

The benefits of microservices

Stability

- Because microservices are decoupled and store their own data, they can be independently modified and deployed without affecting each other or the monolith, and without a full restart of the system.
- With monolithic applications, even small changes can introduce bugs or take a site down. Finding and fixing issues is far more difficult with large applications. You're working with millions of lines of tightly coupled code, previous changes may not have been properly documented, and developers may have left or changed projects, taking their expertise with them.
- With microservices, teams can make smaller, more frequent updates, which makes it easier to identify problems or rollback changes. And if one microservice slows down or fails, it doesn't affect the entire system.

Speed to market and efficiency

- Microservices can be managed by small, focused teams rather than a centralized team responsible for multiple areas of the system. Teams can make decisions independently without consulting other groups or worrying how updates may affect concurrent projects.
- Microservices architecture accelerates development, testing and deployment. You can deploy multiple releases per hour, day or week instead of week, month, quarter or year. You can respond more quickly to customer behavior, competitor moves and industry trends, and "fail faster." Innovation carries less risk — it's quicker and cheaper to deliver and simpler to undo if a new tactic doesn't pay off.

Scalability

- Scaling a monolith is costly and inefficient. When a monolith hosted on-premises needs to scale — even if only seasonally — investment in more licenses and hardware is required. Though the cloud offers the ability to scale up and down only if and when necessary, the entire application still scales as a unit, even if only a single or handful of components benefit from the extra bandwidth. Microservices scale independently.

Flexibility

- While monolithic platforms may be "flexible" by virtue their code can be modified, monolithic architecture is by nature very rigid. Microservices architecture supports much more than just custom development.
- Individual microservices don't need to share the same tech stack or database structure as your monolith or each other. Microservices can also be swapped-in and swapped-out over time to keep your environment up to date with best-of-breed technologies.
- Modularity helps you extend functionality to new touchpoints efficiently. Rather than hardwiring new touchpoints to your monolith, you can work with only the microservices needed for each use case and context. For example, an AI-driven chatbot application may only need to call customer accounts and order management, but not catalog, PIM, search, promotions, cart or checkout. Or, visual or voice search in your mobile app may only need to talk to your catalog.
- Microservices can run in parallel – something not possible with a monolith. For example, you may want to run two versions of your checkout microservice: one for your domestic and one for your internationalized site. Each can take advantage of optimized forms, currency conversion, payment options, tax and shipping rules while scaling independently. Or, your B2B portal may share catalog, PIM, promotions and cart with your B2C offering, but have its own account, order management and checkout modules. Parallel microservices allow you to keep your code clean between services and customize integrations.
- Running microservices in parallel also allows you to run more complex A/B tests than a monolith, which doesn't support running two sets of code within it. For example, you may wish to test two search engines, each with their own algorithmic tuning. Test code can be deployed and rolled back independently of the monolith.

Monolith vs microservices: at a glance

Monolithic Architecture	Microservices Architecture
Each application in a monolithic architecture operates in relation to the other applications in the architecture.	Each microservice is an app unto itself with its own API and well-defined functionality, and can be deployed and function as a standalone entity.
Customizing software is order of magnitude riskier. A change to one area can impact many other areas (e.g. by introducing a bug or degrading performance) and requires full regression testing across the entire application.	Enhancing an existing microservice or deploying a new one doesn't disrupt other services or existing application functions. Testing and rollback are much faster.
Subsequent enhancements are expensive and labor-intensive.	Enhancements can be deployed quickly and cost-effectively.
Monoliths can only scale as a whole.	Microservices can scale independently of each other.
Components of a monolith share a common technology stack and database schema.	Developers can choose the programming languages and database structures that suit each microservice best.
Can only be extended to limited degree with APIs and is often vendor-dependent.	Unlimited potential to extend via new microservices and APIs.
New touchpoints must be hardwired to the monolith. They can't leverage only the components they need to access.	Microservices can be extended to new touchpoints independently.
Monolithic architecture doesn't support multiple versions of a service.	Microservices can be run in parallel to satisfy specific business requirements or A/B testing.

SECTION 4

The Strangler Pattern in action

Case study: Staples

For the brand famous for its Easy Button, the weight and rigidity of its monolithic commerce platform made extending commerce to new touchpoints and adopting new business models anything but easy.

Driven by the desire to own their customer-facing experiences across multiple platforms and touchpoints (including B2B and international sites, apps and in-store features) Staples undertook efforts to decouple services like cart, checkout, catalog and payments from their IBM WebSphere platform, keeping core functionality of WebSphere Enterprise Java server.[1]

Staples' new microservices architecture allows reuse of service across multiple domains, including its Canadian and B2B properties. Modular architecture also supports new business models and sources of revenue, including white-labeled photo and printing platforms Staples licenses to other businesses.

Staples has also built an Easy Button system that leverages AI and personalization technology to enable customers to quickly reorder supplies, track orders and self-serve. Freed from the glut of the monolith, integration with leading-edge technologies is more streamlined.

> "The future [is] basically going to be all microservices for us. We find it to be much more flexible and an environment where we can have a product we own end-to-end"
>
> —Faisal Masud, CTO, Staples

Products like Easy Button are owned end-to-end by small DevOps teams in an Agile environment. This organizational approach combined with flexible architecture allows Staples to release code weekly, rather than every few months.

1. https://thenewstack.io/staples-targets-Microservices-cloud-cognitive-flexibility-growth/

Case study: Best Buy

Digital directly accounts for 20% of Best Buy's business and supports a large portion of in-store sales and omnichannel customer service. For a big box retailer in the Age of Amazon, any choke point in delivering innovation and top-tier customer service is a strategic disadvantage.

By the late 2000's, Best Buy was struggling with a complex monolithic commerce platform (ATG). Bogged down by dependencies, even simple changes were difficult and essential integrations exceedingly costly. The system was riddled with single points of failure, and traffic spikes would easily take the site down.

Code releases happened infrequently and required months of planning. With most development and maintenance outsourced to partners, updates were largely project based. Contracted developers frequently moved on to different projects, taking their experience with Best Buy's system with them. Under the crunch of project-based delivery, only top priority changes could be squeezed into each deployment, and releases were regularly delayed. Teams of over 60 people would routinely pull all-nighters during deployment.

In 2010, Best Buy embarked on a strangler mission[2] to systematically choke out the monolith, replacing tightly coupled services with microservices while building new innovations around the monolith.

Microservices allowed Best Buy to deliver new experiences that would have been impossible on its legacy platform. One example is the Enterprise Returns app which allows customers with a Geek Squad return plan to exchange a broken TV for a gift card for a new TV of the same value. The app calls the catalog microservice to search comparable televisions by dimension, features and price.

For an extensive catalog like Best Buy's spanning multiple departments and third-party marketplace listings, it was critical that the catalog be scoped to include only relevant SKUs and data with its own search logic to stay performant, scalable and useful. Services within a monolith simply can't be decoupled and duplicated for bespoke and innovative projects and use cases.

Running independently of the monolith, the Enterprise Returns application can scale with demand, be modified as frequently as needed and extended to multiple consumer touchpoints like mobile apps, in-store kiosks and the Internet of Things without adding any code to the monolith or compromising other services and functionality.

Today, Best Buy enjoys the agility and scalability of full microservices architecture and continues to lead the industry in digital experience.

2. https://blog.runscope.com/posts/monolith-Microservices-transforming-real-world-ecommerce-platform-using-strangler-pattern

SECTION 5

Frequently asked questions

What's the difference between SOA and microservices?

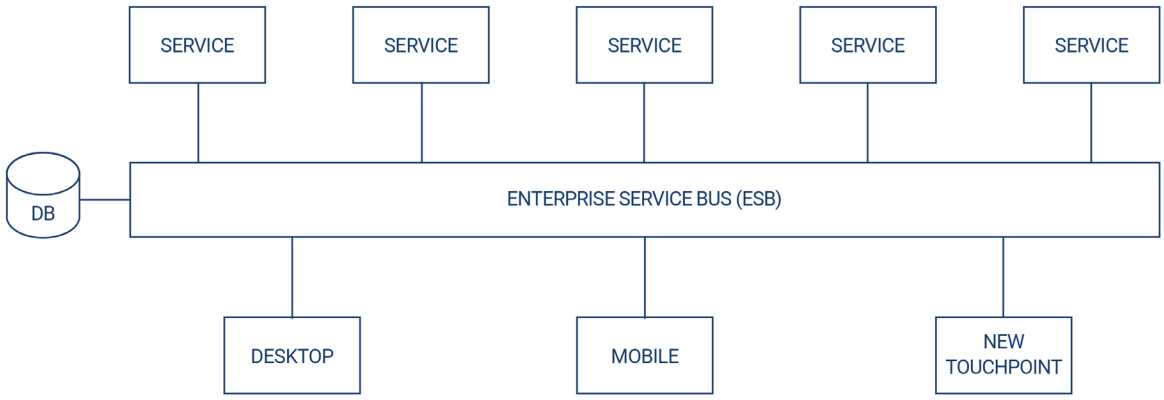
Both SOA (service oriented architecture) and microservices architecture are alternatives to monolithic architecture that leverage smaller, more manageable applications that are scalable and can be swapped-out for best-of-breed capabilities. But microservices architecture is not to be confused with SOA.

SOA architecture relies on a central Enterprise Service Bus (ESB) to relay messages between services. The ESB is a "smart pipe" that contains all the business logic for all services across the network, thus code becomes tightly coupled within the ESB. Changes to any service (endpoint) requires an update to the ESB, which in turn affects any or all other connected services. Over time, the ESB can become as bogged down with spaghetti code as any monolith.

As with monolithic architecture, if one service fails or slows down, the entire ESB can get hammered by requests for that service, affecting performance across the entire system (and negatively impact customer experience).

In contrast, microservices are "smart endpoints." Microservices contain their own code, rather than code living in the ESB. Development teams can work on projects independently of other teams and deploy code faster without disrupting other services. Microservices have less dependencies than SOA, and therefore fewer failure points.

Services Oriented Architecture



How are microservices different from headless commerce?

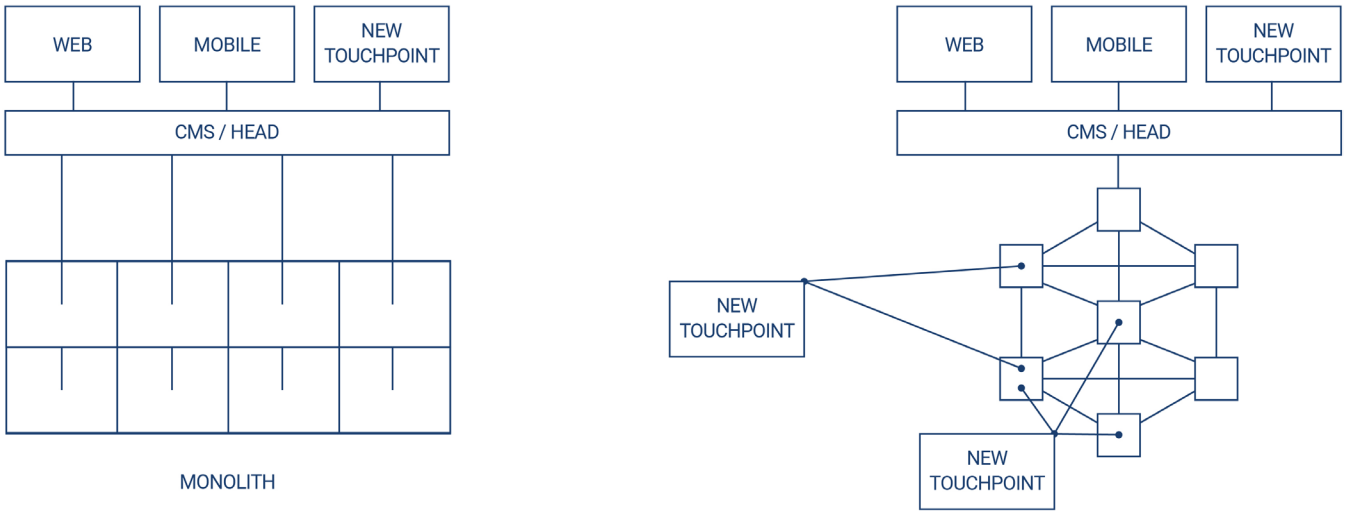
"Headless commerce" decouples the presentation layer from the rest of a monolith's commerce components to allow best-of-breed Web Content Management (WCM) or Digital Experience platforms (DXP) to power the front end. Business logic and data remain in the monolith, and code remains tightly coupled.

Headless commerce certainly offers more flexibility than a monolith and is a step towards omni-touchpoint, "commerce everywhere." In addition to WCM and DXP, external touchpoints like mobile apps, interactive lookbooks, in-store kiosks and wearable devices can serve as a "head" and access the monolith through APIs.

While “heads” contain their own presentation logic, they’re limited to the monolith’s business and data logic. Modifying the monolith to satisfy the requirements of a new endpoint impacts all aspects of the monolith, including other wired heads. This can limit innovation because the business value of a new endpoint must justify the effort and risk of updating and regression testing the entire system.

With headless commerce, endpoints can’t scale independently and increase the load on the monolith even if they access only a single service, forcing the entire platform to scale. With an omni-touchpoint strategy, a handful of small endpoints may increase license and hosting costs exponentially.

Headless commerce with a monolith vs microservices



Microservices: build or buy?

Generally, as with any software build or buy decision, build gives you greater control and choice of programming languages, features and structure of your microservices, while buy gets you to market faster.

When migrating to microservices architecture, you can also employ a combination of both build and buy, depending on your resources, timeline and business needs.

Refactoring your existing monolith

Extracting and refactoring existing monolithic code into microservices is not without its challenges. While reusing code can be faster than from-scratch rewrites, undoing existing dependencies can become very time consuming, and you may need to copy and modify code extensively for areas where dependencies are “too big to bring along.” You’ll likely need to refactor data structures, which adds complexity to your project.

From-scratch rewrites

Depending on the age, size and complexity of your existing monolith, building new greenfield microservices may be faster than working through code dependencies and data structures. It also allows you to select the "best tools for the job" with respect to programming languages, databases and features that best reflect today's (and tomorrow's) business requirements. However, from-scratch rewrites delay time-to-market and may require you to hire new developers skilled in these languages, rather than leverage resources familiar with the monolith's stack.

Allowing smaller, independent development teams to build their own microservices can also be problematic if the overall strategy is not overseen by an experienced software architect. The end result can be a mish-mash of languages, frameworks, data structures and lack of consistent documentation, with unnecessary redundancies between microservices.

Setting up your microservices environment

Whether you refactor or rewrite, you'll need to build an API layer, messaging system and business tools in addition to your microservices, all while concurrently maintaining your live monolith. If you choose to build, expect to spend approximately 50% of your effort keeping your live system running and 50% on setting up your microservices environment in your initial stages.

Both in-house development options (refactoring and from-scratch rewrites) carry a learning curve and extend time-to-market significantly. Leveraging third-party, prebuilt microservices can help you hit the ground running much faster, especially when they ship with ready-to-go business tools, API and messaging layer. Choosing flexible microservices built on a similar stack to your monolith allows you to leverage existing developer talent and customize to suit your business needs.

How should we approach strangling our monolith?

The **Strangler Pattern** offers you flexibility and control over your migration roadmap and allows you to prioritize which modules you migrate first. Ideally, you will begin with components that are most constrained by your monolithic architecture. For example, services that have an immediate need to scale, or that benefit most from frequent updates and continuous delivery. Or, new innovative experiences that aren't worth baking into your current platform.

However, business value should not be your only consideration. How you tackle migrating to microservices architecture depends on your monolith's structure, business pain points and IT strategy. Your plan should balance migration with minimizing disruption to existing operations. If you're extracting and refactoring services in-house, ensure you've conducted a full discovery of dependencies and allow some wiggle room for unforeseen dependencies to pop up during development.

Remember, you need an environment for your new microservices to live in and communicate with each other. If you're building microservices completely in-house, you'll need to build your API layer and messaging system first. If you use third-party microservices, you can leverage this infrastructure out of the gate and use it for your own builds as well.

Don't forget business tools! The goal of microservices is to foster IT agility, not create more organizational dependence on IT for everyday administration. A microservices vendor with out-of-the-box business tooling accelerates your project and lightens the IT burden over time.

SECTION 6

Let's recap

Upgrading digital commerce technology no longer requires a Big Bang replatform from one heavy monolithic system to another. Using the **Strangler Pattern,** an enterprise can systematically migrate components of the monolith to microservices, with minimal disruption to existing systems during the transition. The end result is a more modular and flexible architecture.

1. Microservices are loosely coupled, independently deployable, have their own databases and communicate through lightweight APIs. Smaller developer teams can own delivery end-to-end without needing to know the ins and outs of the entire monolithic application or co-ordinate with other teams. New code can be delivered faster without extensive regression testing or risk to other parts of the system.

2. Unlike monolithic systems which require the entire application to scale – even if only one component is hitting its limits, microservices can scale independently, providing efficient use of hardware and bandwidth.

3. Microservices don't need to share the same tech stack, code base or database structure as your monolith or each other, and can be individually replaced over time if needed. They can also be independently integrated with other applications and consumer touchpoints, leveraging only the required components, rather than hardwired to the entire platform.

4. Microservices can be run in parallel, allowing the flexibility to suit unique use cases for internationalization, B2B/B2C, A/B testing and more.

5. The **Strangler Pattern** allows you to focus your initial effort on the components causing the most pain. Choose components that have immediate need to scale, would benefit most from frequent updates and delivery, or services that could be leveraged for innovation projects.

6. To communicate with your monolith and each other, your microservices need an API layer and messaging system. Leveraging third-party microservices can help you get to market faster. To avoid creating IT dependencies for routine tasks and streamline efforts to code delivery, ensure your microservices include user-friendly business tools.

About Skava

Skava's cloud-native commerce platform, Skava Commerce, uses microservices-based technology to help retailers and enterprise-sized companies quickly create personalized omnichannel experiences. Businesses can deploy our full platform or choose only the components they need to enhance their existing digital commerce stack. Skava's modular architecture enables brands to continuously innovate, accelerate time to market, and delight customers with fewer resources. Category leaders and Fortune 500 companies make up Skava's customer roster which includes Kraft Heinz, Barnes & Noble, T-Mobile, Urban Outfitters, and many more. Headquartered in San Francisco, the company has offices in Europe and India. To learn more, please visit Skava.com.