ROYAL CYBER

White Paper
www.royalcyber.com

Innovative Commerce with μ-Services

Executive Summary

Commerce is an environment that is constantly changing and upgrading as technologies improve and companies battle against each other to win a greater share of the pie. The "one platform" fits all has proven to be challenging in regards of innovation, scalability and speed to market, and there is rarely any implementation that uses the full features set of any platform.

Forrester has characterized seven patterns that allow organizations to stitch commerce and content together. While all have pros and cons, they see a general industry pivot toward decoupling into lighter, faster solutions. Web CMS supported by headless commerce and solo commerce are emerging as the two most logical and efficient patterns. Forrester concerns with the API approach are the track record and the development skillset.

The "**Innovative Commerce**" is a solution that addresses the monolithic approach limitations by introducing micro-services (μ-services) architecture that puts innovation as the top priority, while creating experienced capability teams to support this new model.

Wadih Bargach
Commerce Practice Lead, Royal Cyber

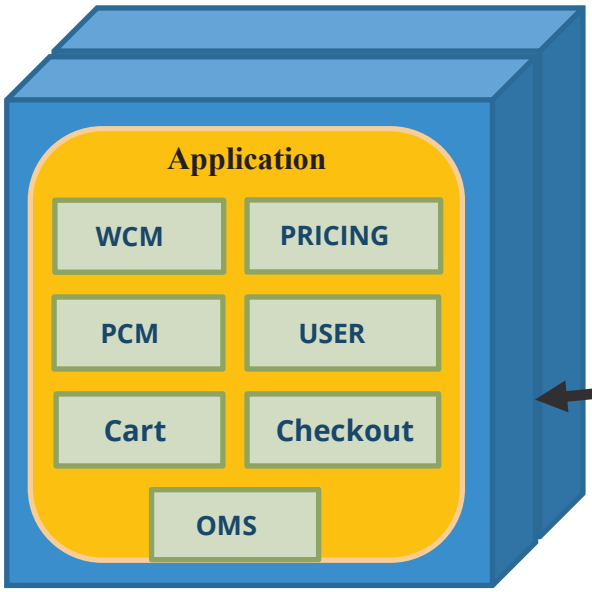
July 5, 2019

Table of Contents

1. Monolithic Architecture

Simple to develop, deploy, test, and scale.

1.1 Traditional Architecture

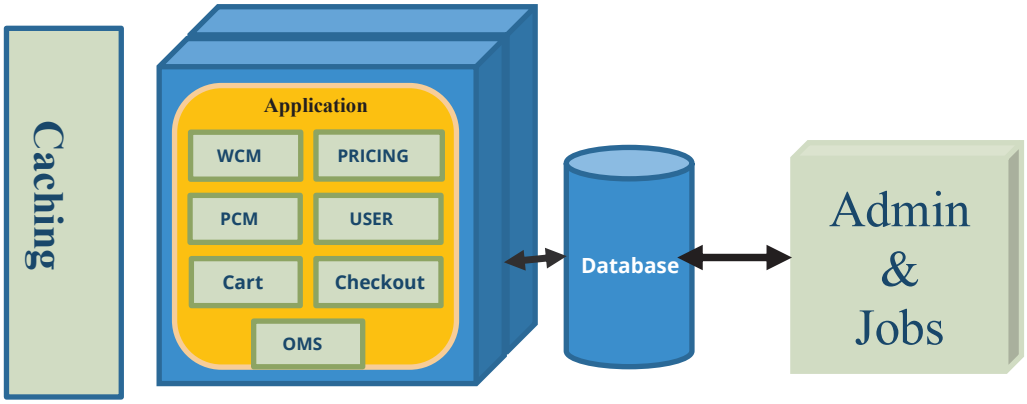


This solution has several benefits:

- Simple to develop – Has been around for a while and the current development tools and IDEs are tailored to support the development of monolithic applications
- Simple to deploy - we simply need to deploy the single artifact.
- Simple to scale - we can scale the application by running multiple copies of the application behind a load balancer

How we scale further?

We add caching layers, admin nodes (jobs).



1.2 Trends to Consider

Per Gartner report (Market Trends: Digital Commerce Platforms in Growth Mode, Worldwide), gone are the days when most brands offered just one or two international sites. Today, the end goal is a global commerce footprint (large data volume), with Asia Pacific been on the top given the size and growth potential of many of the region's markets. However, brands looking to expand globally are more likely to use international shipping or marketplaces (an integration model) as part of their approach to new markets.

According to the same report the main drivers behind the digital business include social, mobile, big data, cloud and Internet of Things (IoT). So, tomorrow's features are almost here, and companies will need to be agile and get onboard to stay competitive.

Build for the "Internet of things" and "Autonomous Machines"

Predict customer preferences, and personalize the experience online and in the store

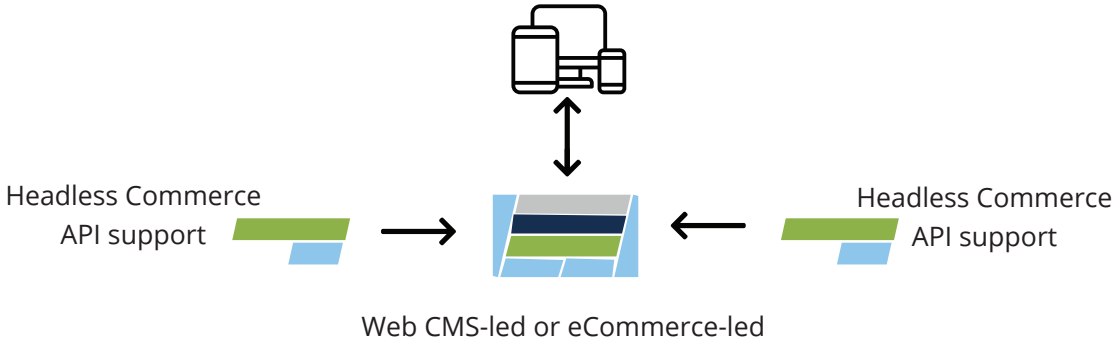
Voice and Virtual experience that connects online and store experiences

Support curated products and digital currencies

An OMS that supports the speed of goods allocation

Forrester research states that on the packaged software side of this discussion, the side-by-side content and commerce model is dead, and the market is simplifying to two primary patterns:

1. Web CMS with headless commerce
2. Solo commerce with good enough experience management capabilities.



Pros

Marketers (typically) gain web creativity and campaign flexibility
Ease transition to multi-front-end support
Elimination of full-stack shelfware
Faster implementation and system agility

cons

Limited enterprise track recode for headless commerce and content
Increased need for developer talent

Same research concluded that taking headless and micro-service trends to their logical conclusion, every solution will be headless. This architecture is only a vision today, as most enterprises cling to commercial stability and a consolidated portfolio.

1.3 Challenges

For certain companies to stay competitive, adapt, and expand, their monolithic approach represents a bottleneck. With a large application and team, these major challenges are going to be difficult to overcome:

- **Deploying new features become slow and expensive** - to update one component, we must redeploy the entire application. This will interrupt background tasks (e.g. Quartz jobs in a Java application), regardless of whether they are impacted by the change, and possibly cause problems to other unchanged components as well. Thus, the risk associated with redeployment increases, which discourages frequent updates.

- **Platform or technology stack dependency** – when our application uses a platform framework that subsequently becomes obsolete then it can be challenging to incrementally migrate the application to a newer and better framework. It's possible that to adopt a newer platform framework we must rewrite the entire application, which is a risky undertaking. It is difficult to incrementally adopt a newer technology. Per example, if we chose the JVM, then other components written in non-JVM languages do not have a place within our monolithic architecture.

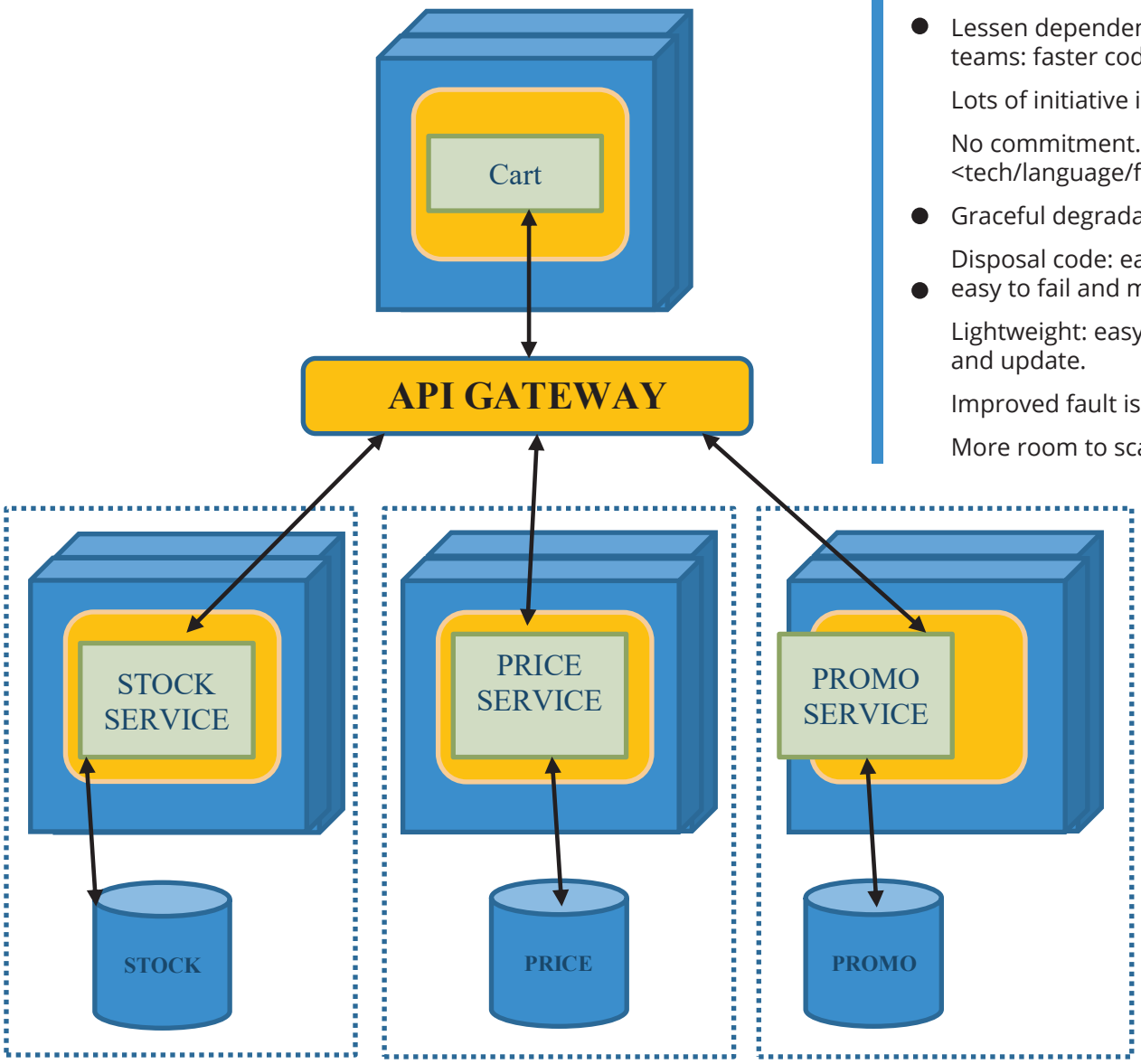
- **Scaling can be difficult** – as I have mentioned earlier a monolithic architecture can only scale in one dimension with couple techniques (clustering, caching, dedicated nodes). However, it can't scale with an increasing data volume. Each copy of the application instance will access all the data, which makes caching less effective and increases memory consumption and I/O traffic. Also, different application components have different resource requirements - one might be CPU intensive while another might memory intensive. With a monolithic architecture, we cannot scale each component independently.

- **A loss of development productivity:**
  - The large monolithic code base can be difficult to understand and modify. Thus, development typically slows down and the quality of the code declines over time. It's a downwards spiral.
  - The mammoth code repository has no hard module boundaries, and every developer needs to merge and regression test against it.
  - The larger the application the longer it takes to start up.
  - Slow development as it is hard to scale it. Monolithic application prevents the teams from working independently. The teams must coordinate their development efforts and redeployments.

2. µ-services Architecture

Lightweight, fast, and independent.

2.1 Sample Architecture



Why people fall in love with µ-services?

- Lessen dependencies between teams: faster code-to-production.

  Lots of initiative in parallel

  No commitment. Favorite <tech/language/framework> here.
- Graceful degradation of service.

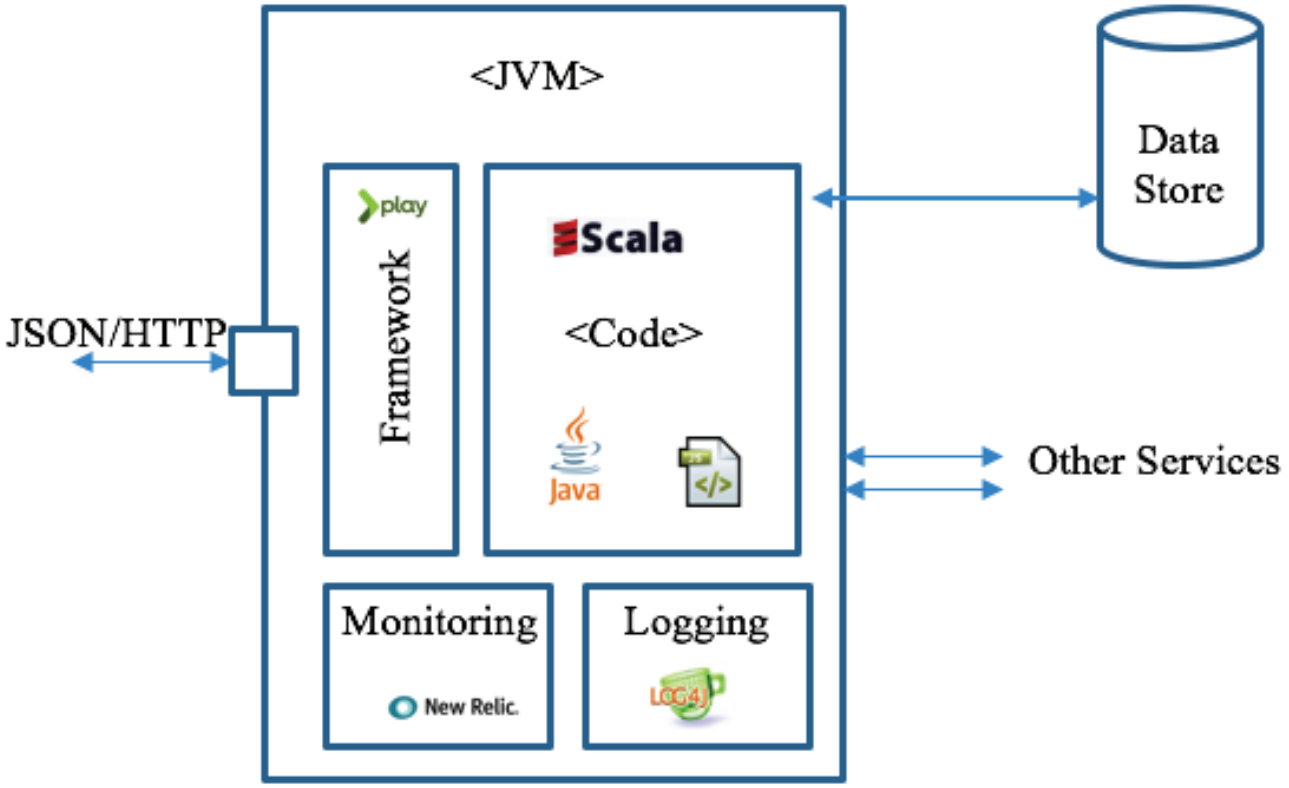
  Disposal code: easy to innovate,
- easy to fail and move one.

  Lightweight: easy to understand and update.

  Improved fault isolation.

  More room to scale.

2.2 Anatomy



In a monolithic application, we have one big piece of machinery everyone must move in lockstep right to the point where either everything goes out or it doesn't, thus a top-down approach. In a μ-services space there are dozens or hundreds of instances, so we have community where service owners and service teams are all doing different things and deploying it to production. Once we're in this community it's important to set some rules and respect them, and below are some major ones:

- Design for failure
- Expect to be throttled
- Retry with exponential back off
- Degrade gracefully
- Cache when appropriate
- Publish performance metrics
- Keep the implementation details private
- Maintain backward compatibility
- Enforce API consistency
- Provide authentication and authorization
- Keep it stateless as much as possible (expect for persistence and caching)

3 The Vision

Organize, engage, then lead.

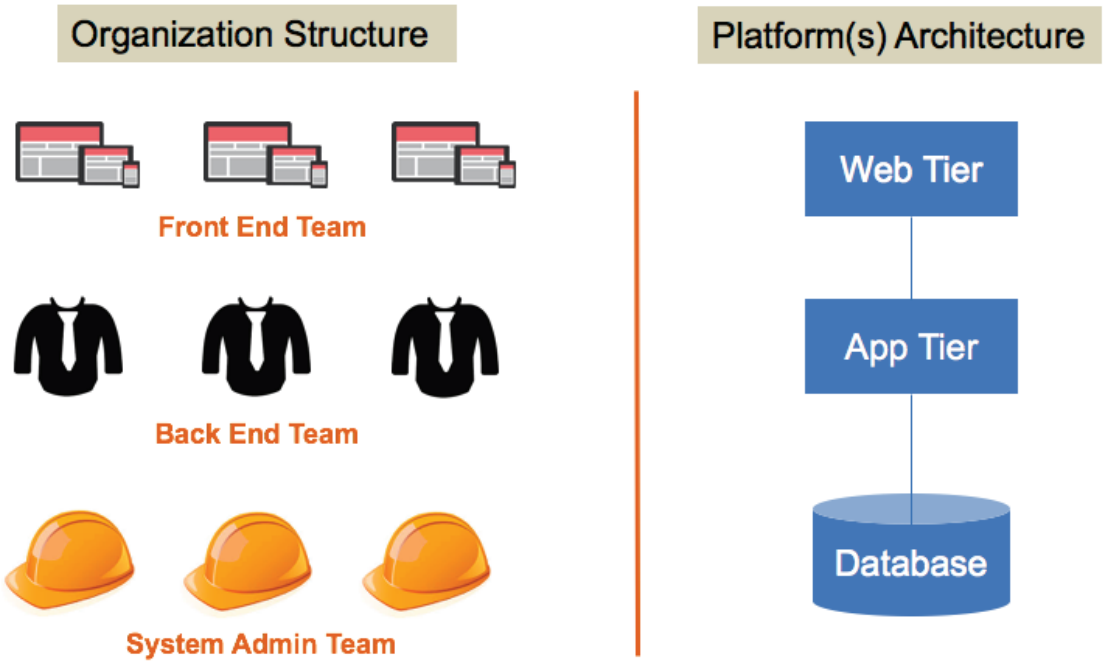
3.1 Organizational changes

Most common questions in this area are:

- How can we make it easy and fast to get the change to production?
- How can we arrange our teams around strategic initiatives?
- How can we scale up independently?

Monolithic team structure

Currently most of the organization structures are mirroring the platforms structure (Hybris, EP, ATG...)



Also:

- The technical expertise is boxed within a platform capabilities and technology.
- Thigh dependency on platform's market demand.

µ-services Team Structure

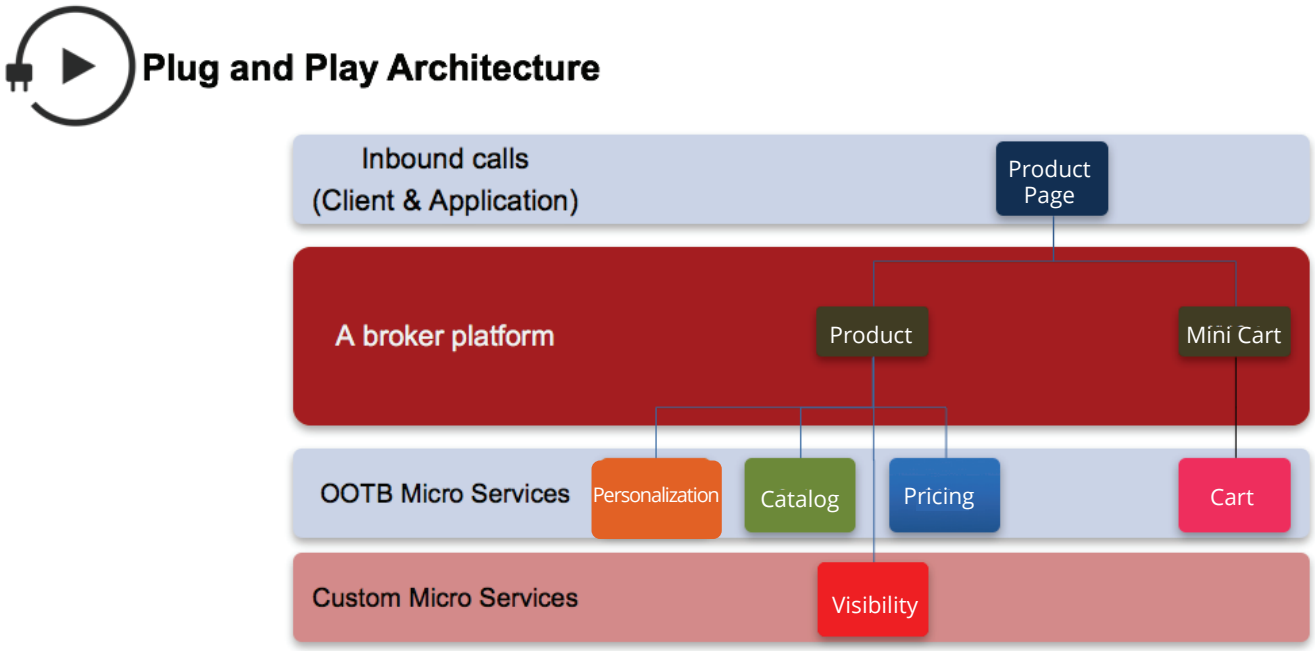
The organizational changes that we should see as we move into µ-services land, tend to structure the teams around a capability or business initiative. For example, we should have an account management team, a PCM team, a personalization team, a cart team, a checkout team, an email team, a push notifications team, etc...
Then we should tackle team composition and understand what skillset and SMEs we need. Each team will have different needs so no need to come up with the fixed composition.
Most of the technical folks are strong in one or more commerce capability. However, the new element here is that these people will need to expand their knowledge beyond the single platform & technology.
Each team:

- Operates independently
- Has full ownership of their service
- Defines their own timeline
- Responsible for testing their service
- Responsible for production releases
- Owns their source code repository
- Responsible for code review
- Free to select the appropriate technology & design
- Platform independent
- Should stay up to date with the capability's features within different platforms, new trends, & limitations.
- Should be able to compare multiple offerings for a capability and recommend the appropriate solution based on the need.

3.2 How to Engage

New implementation

µ-services architecture is not a solution that fits all either. For new implementations that a single platform can satisfy most of their requirements for the next decade and innovation/speed/scale is not a concern, using a monolithic approach may be the answer. However, for new comers that are starting small where most of the platform becomes shelfware, µ-services can be a cost-effective solution with a good runway to grow.



For most of commerce new implementations the architecture will be a more extended picture of the diagram above, where abstraction and lightweight are the key. Each µ-service needs to be a good citizen as I described earlier.

OTTB vs Custom? Only customize when it is not available. Use and contribute to open source services as much as possible.

Single service per container/instance: which provides independent monitoring, independent scaling,
clear ownership, and immutable deployment.

Avoid hotpots: use dependency injection (get the data once and pass it over).

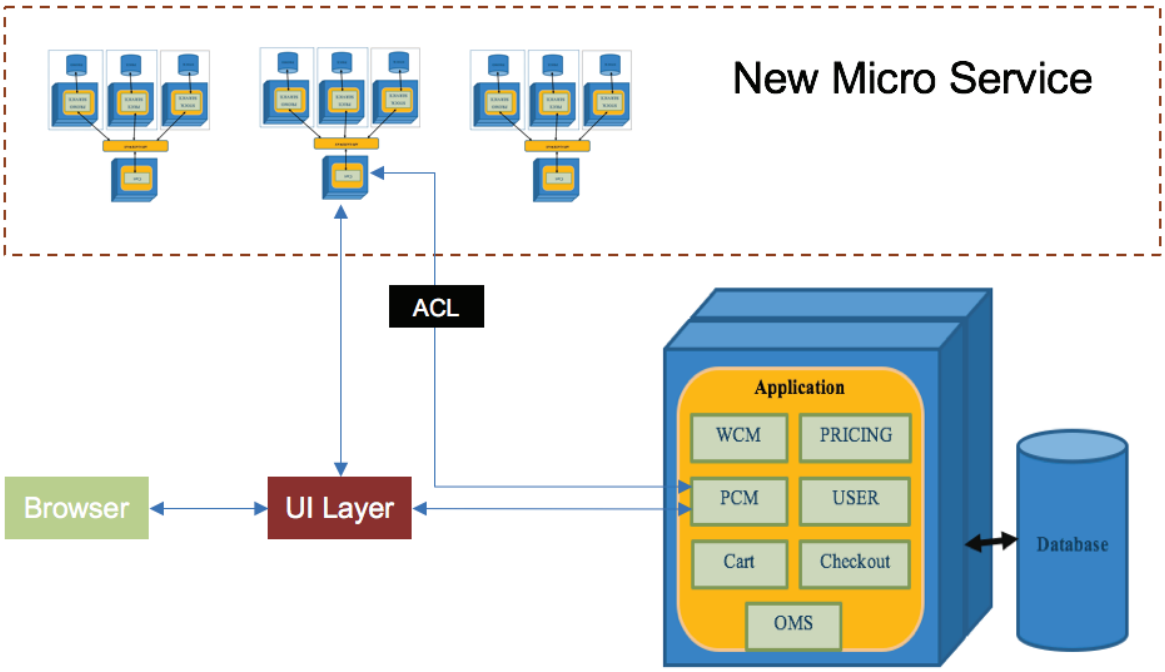
Monitoring: Publish externally relevant metrics (latency, RPS, Error rate…) and understand internally relevant ones (Basic, OS, Application…). When we are the consumer, aggregate our metrics to account for
the slowest service call.

Logging: we need to pick a common log aggregation solution and on the log entry formats, naming convention, and correlation strategy.

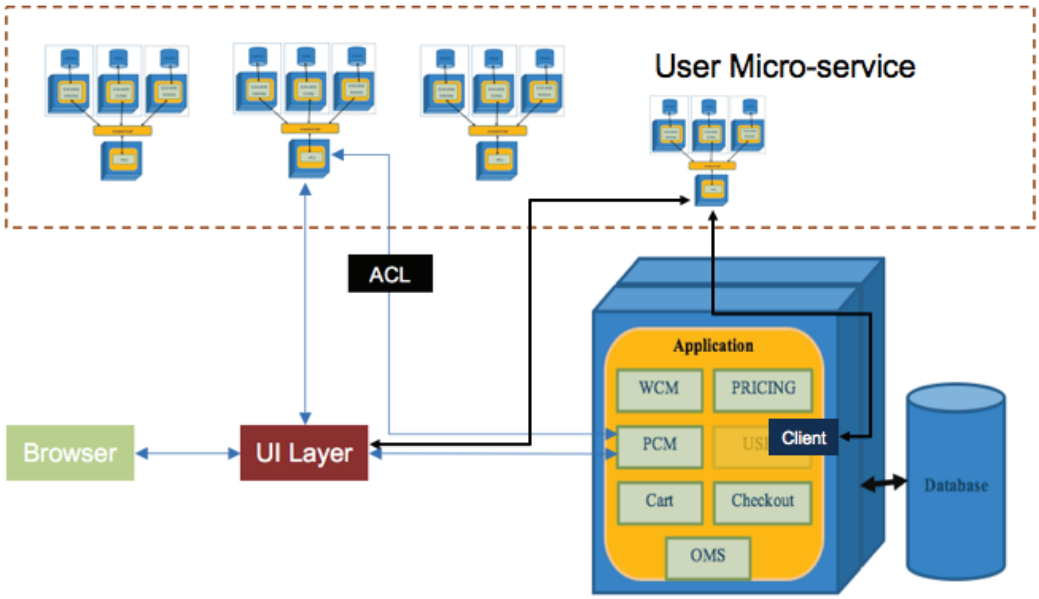
Existing Implementation

It's a journey to move from a monolithic architecture to a μ-services one, however it is possible. Several companies have done, and they have recommended baby steps as described below:

1. New capability should be built as μ-services. We can use an anti-corruption layer (ACL) to keep our implementation clean and independent from the monolithic structure.



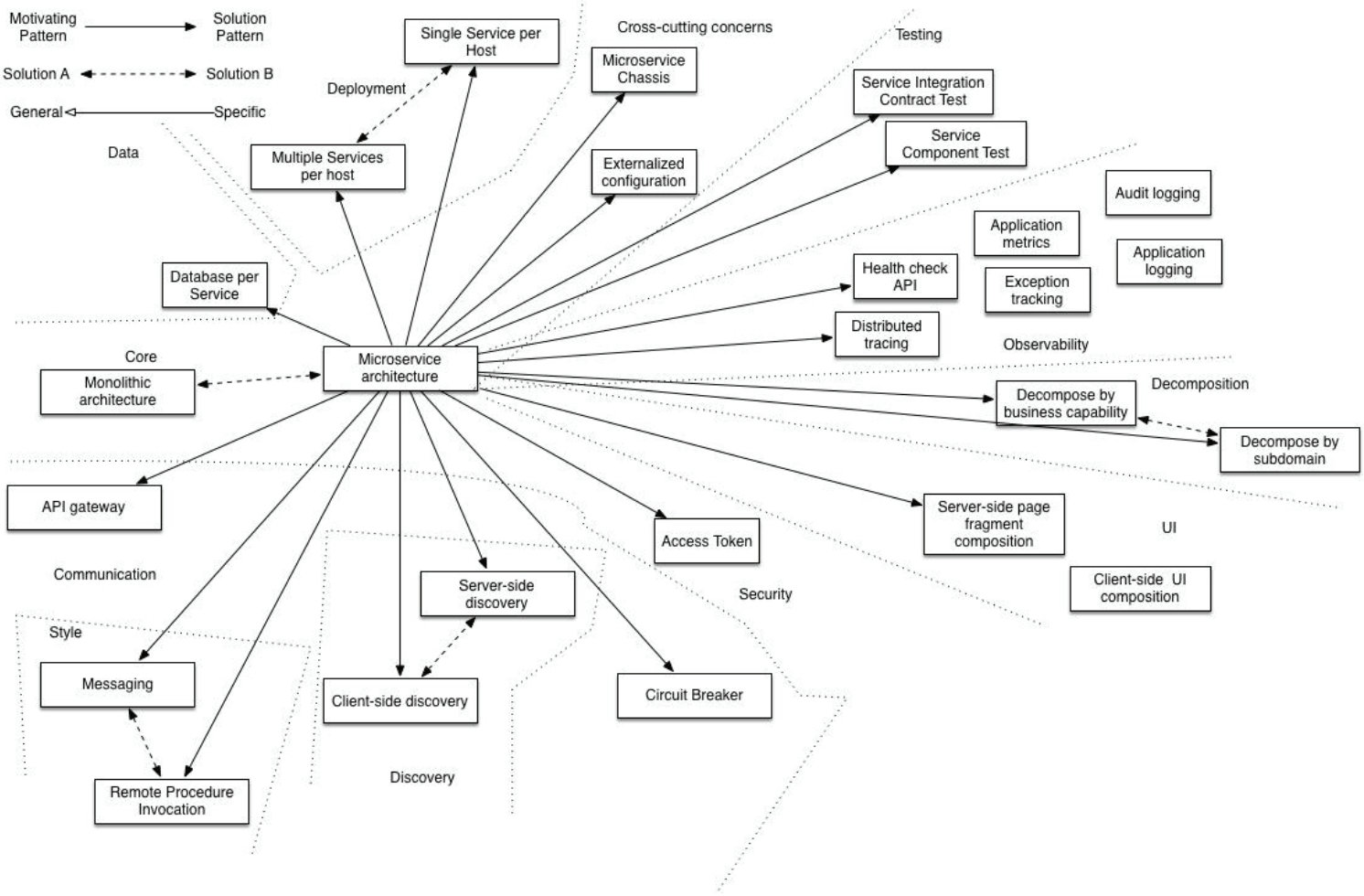
1. Move capability with the least dependencies first. Per example user functionalities do not depend on other functionalities so they can be moved first to their own micro-service.



1. Walk the walk – If we decide there is a benefit of a full transition, then move slowly respecting the previous rules.

4. Summary

1. **Micro-services are not a silver bullet** - It is not the only next thing. For less complex applications, monoliths are always better in both the long and short-run. Remember that if you are not solving for: innovative, scale, and speed you may not need it.
2. **Services are simple, surrounding architecture is not** - Building services is rather straightforward using one of the multiple frameworks currently available. However, there is a relatively large amount of concerns and capabilities, some of them unique to micro-services, that should be addressed before we decide to build micro-services



Source: Microservices.io by Chris Richardson

Royal Cyber | Simplifying IT for Customers & Partners

Royal Cyber Inc. Headquartered in Naperville, IL is a leading software organization that provides services ranging from application development and deployment to training and consultancy. We commenced the operations in the year 2002 as a specialized Technology provider striding in as a software deployment service provider, assisting clients to meet the standards and demands of doing business in the rapidly changing marketplace.

Today we stand tall as a One Stop Shop for all your IT needs.

55 Shuman Blvd, Suite 275, Naperville, IL 60563 USA. | Tel: +1.630.355.6292
Email: info@royalcyber.com | Website: www.royalcyber.com