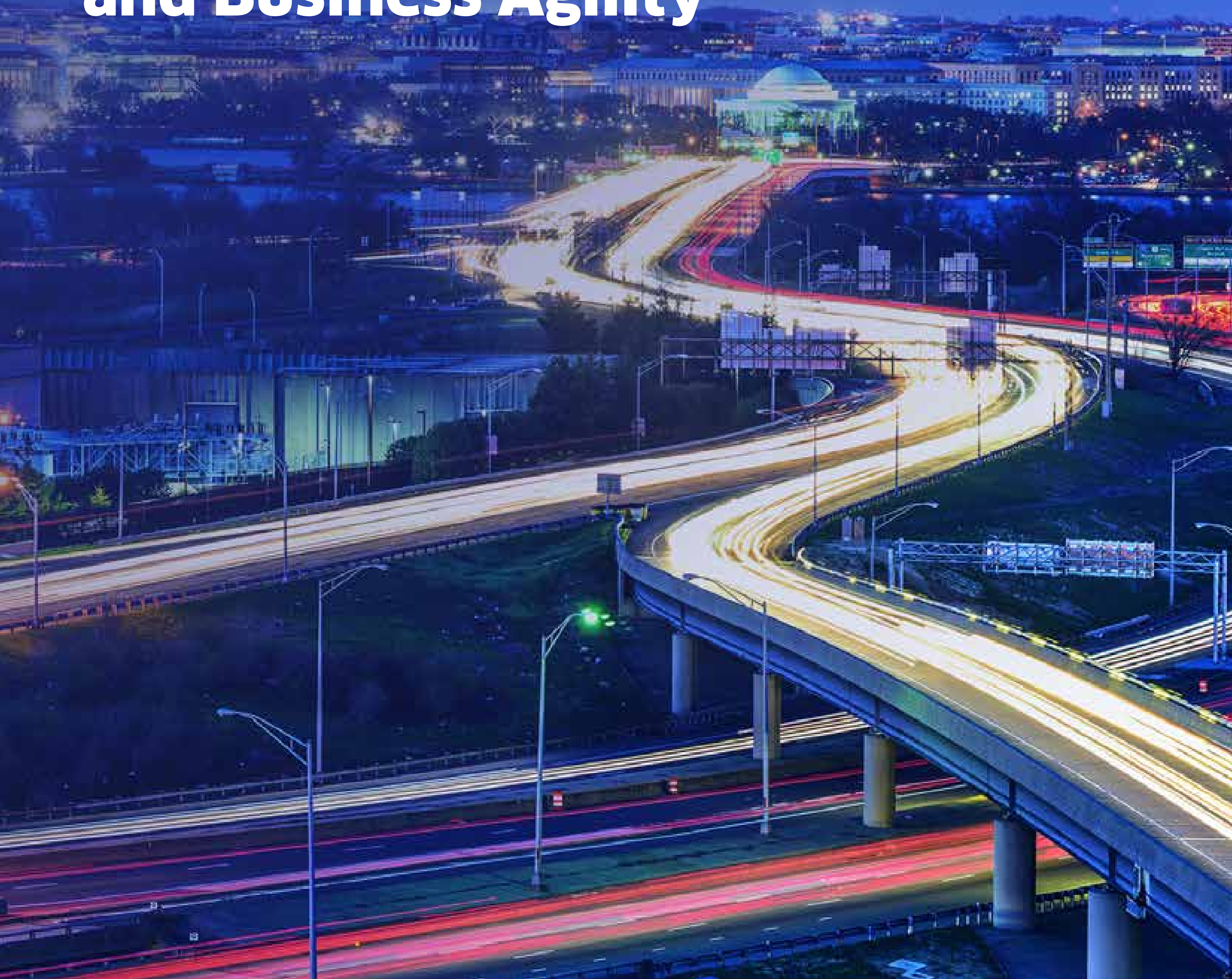
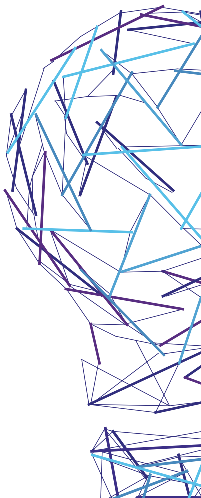
TIME-TO-MARKET DRIVEN DISRUPTION

Starter Guide to Microservices and Business Agility



CONTENTS



EXECUTIVE SUMMARY

Digital transformation has for some time been a top priority for Fortune 500 organizations although very few are on the course to actively achieve this. In reality, it is people and technology who, together, contribute to the success of digital transformation. Initially it begins with changing people's mindset to become customer centric, dynamic, accepting and responding quickly to change. This is followed by the technology.

The challenge associated with technology is not in the adoption of new systems and platforms, rather more, it lies mostly in the IT infrastructure. Some organizations face challenges with their infrastructure which hinders their efforts towards achieving digital transformation. Lagging behind the competition when it comes to releasing new features is one obstacle that an organization may face. Consequently, organizations have started to embrace new ways of transforming their infrastructure and platforms to fit with their dynamic needs and to increase their business agility.

EXECUTIVE SUMMARY

The software industry itself is being disrupted by new trends and methods with regard to how organizations adopt new digital platforms and build software. The software industry has been disrupted by new tech giants such as Netflix, Amazon, LinkedIn etc. and those companies have changed how the world adopts and builds software. Those new trends can be summarized as follows:

1. Software is being built not bought: most of those companies and digital transformation leaders are building their platforms rather than buying off the shelf software.
2. Software is being updated on a daily and weekly basis: Those companies release new features multiple times a day across multiple datacenters rather than running long release cycles on a quarterly or annual basis.
3. Software engineering paradigms have changed from monolithic and MVC design patterns to microservices which have made the new platforms far more scalable and extensible.
4. Software processes and methods have changed from waterfall and running long release cycles to Agile and DevOps with a high reliance on test automation, continuous integration and automated deployments.
5. Migration from traditional infrastructure and data centers to cloud platforms and container orchestration platforms whether external or in house.

These new emerging habits are becoming widely adopted and transforming the way in which IT executives think about how to undertake their digital transformation.

This e-Book explores microservices architecture and its benefits in accelerating the digital transformation of organizations. The following section also discusses key use cases including legacy systems transformation, agile integration, building cloud-native apps amongst others. We will then list some of the key technologies and share a case study featuring one of our customers.

WHAT IS MICROSERVICES ARCHITECTURE?

WHAT IS MICROSERVICES ARCHITECTURE?

Microservices architecture is a software development approach where the software application is modular and created through **independent components**. Each of those components is built, tested, deployed and operates totally independently of the other components.

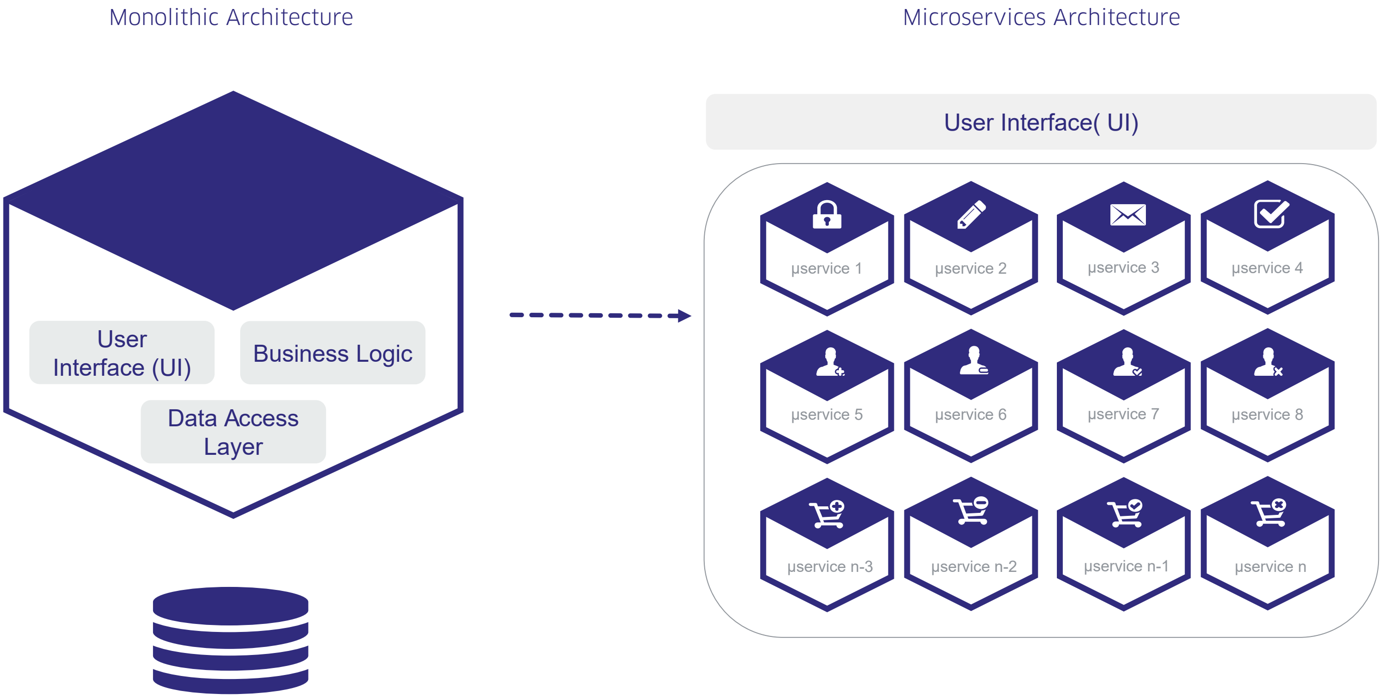
These modules perform a given task and work autonomously, providing continuous delivery even if another module is offline. They are designed to handle one task, communicating with the other modules through standard APIs and events.

> In short, the microservice architectural style is an approach to developing a single application as a suite of small services, each running in its own process and communicating with lightweight mechanisms, often an HTTP resource API. These services are built around business capabilities and independently deployable by fully automated deployment machinery. There is a bare minimum of centralized management of these services, which may be written in different programming languages and use different data storage technologies.

-- James Lewis and Martin Fowler (2014)

In a monolithic architecture, one big box functions as one application. This box consists of the User Interface (UI) component as well as a business logic and data access layer. Conversely, microservices allow the building of small shippable functions where each function consists of its own UI, logic and data components.

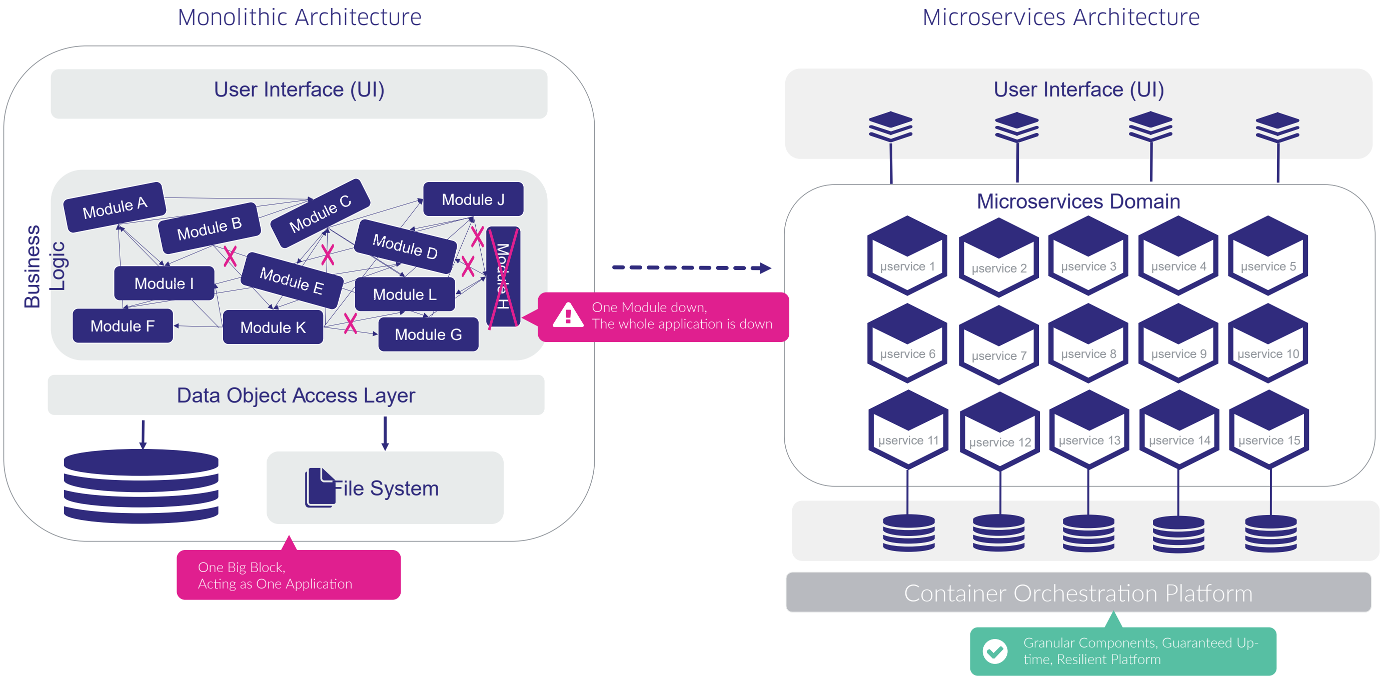
This is why microservices architecture speeds up the process of software development and has a great impact on business agility. All these benefits will be explored in the next section.



As end users, we sometimes encounter errors when using software applications. Those issues could involve the entire application crashing or a major issue as a result of one feature failing. In the case of an application crashing, there will be a period of down-time while the development, testing and operations teams fix, test and deploy the new release. This has a significant impact on end users and customer experience.

On the other hand, if only one feature fails, this has far less impact on the end users of the application. Furthermore, the time taken to release a fix for just one feature is far less than redeploying the whole application. In other words, a major issue may require a change in two lines of code but, the question is, is it necessary to test and release this feature update or test and redeploy the entire application?

To summarize, one feature that causes the whole application to fail is an example of a monolithic architecture. However, when only one feature fails, this represents a microservices context.



One Module down,
The whole application is down

One Big Block,
Acting as One Application

Granular Components, Guaranteed Up-time, Resilient Platform

BENEFITS OF MICROSERVICES ARCHITECTURE

- Time-to-Market
- Scalability
- Guaranteed up-time
- No Vendor Lock
- Resilience & High Availability
- Dynamic Talent Pool

BENEFITS OF MICROSERVICES ARCHITECTURE

Time-to-Market

How long does it take your team to ship new features or updates? Time-to-market has become an essential key performance indicator (KPI) to measure the throughput of your IT and DevOps teams. With the changing demands of business, organizations are competing over their responsiveness to customers' needs.

A team's throughput is measured by the amount of builds (new features) that can be released per month. In other words, a high throughput implies you are increasing your organization's speed in responding to business needs. Hence, you are improving your time-to-market.

What is DevOps?

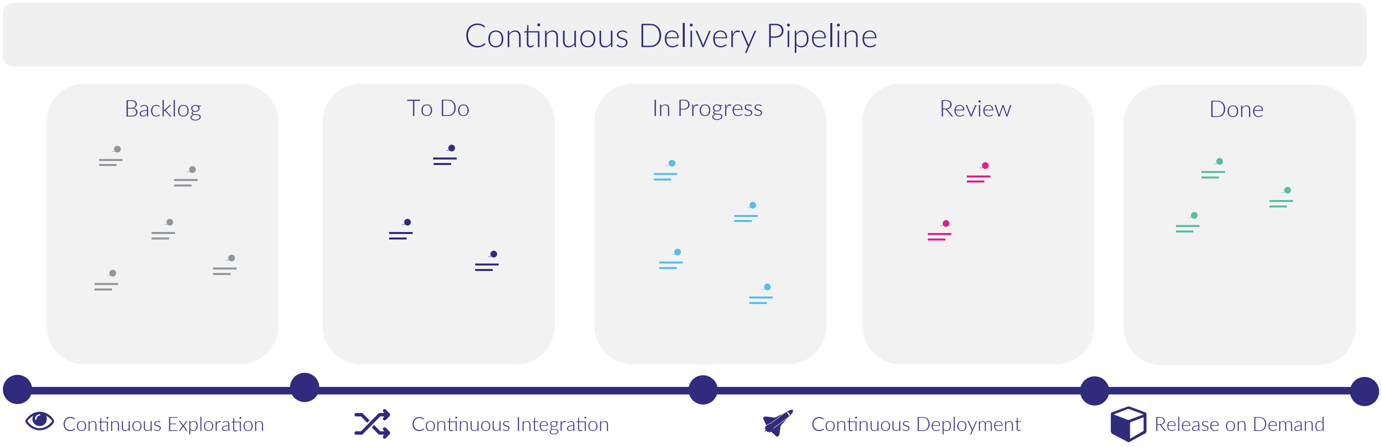
DevOps is an engineering culture bringing developers and IT operations together with the objective of delivering a high quality of software more quickly.

It's also a set of continuous practices automating testing, integration, delivery and deployment.

Microservices architecture enables teams to apply DevOps practices to speed up their development process since each feature is built, tested and deployed independently thus enabling teams to ship new features quickly and independently of each other. Also, due to the nature of the architecture, multiple teams can work together in parallel, thereby also increasing the frequency of releases per month.

Below is an example of a continuous delivery pipeline. Each card on the board represents a shippable piece of software starting from the "Backlog" column and moving through the phases until it has been built and deployed.

The time-to-market in a microservices environment can be improved by 10 times or even more. Basically, you are transforming your culture to a "Release on Demand" approach, a customer centric culture that allows your organization to deliver releases in hours or days rather than in months and years.



Scalability

If you are the owner of a coffee shop and people start queuing up because they love your coffee, would you consider replacing your small coffee maker with a new one? Let's assume that you bought a bigger coffee machine that helped you to deliver the orders faster. Luckily, more and more people continue to queue and, at some point, the biggest coffee maker available won't be able to meet this need. You will need to have more than one coffee maker to be able to cater for the occasional high customer traffic.

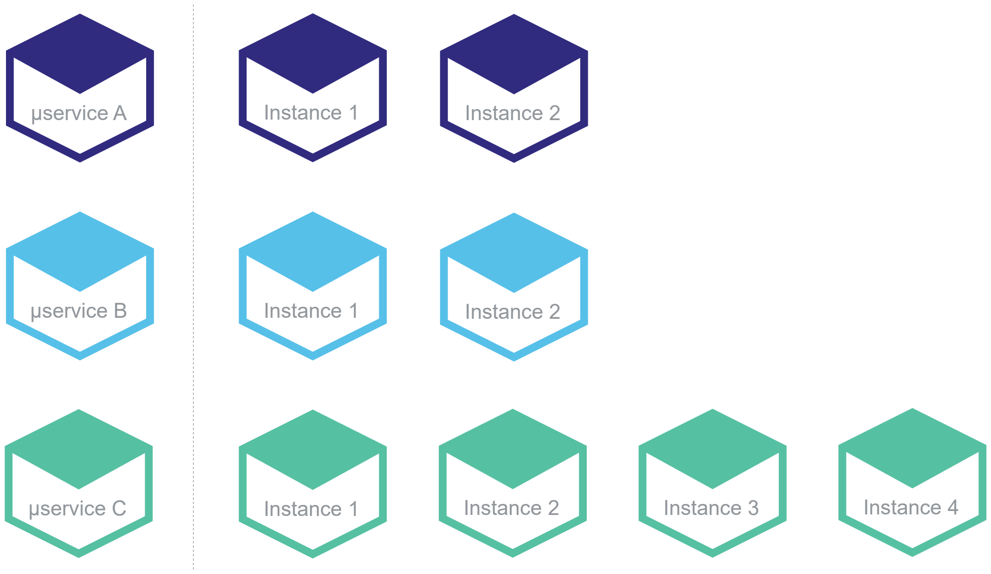
In this example, the original small coffee maker that you are trying to replace with a bigger one is the monolithic application. On the other hand, the small coffee makers that you should be adding to augment the existing one represent the flexibility of multiplying the application instances according to your customers' needs.

That's why microservices offer easy scalability since the modules work independently. At the infrastructure level, each microservice is scaled and managed on its own. You don't need to scale the whole system in situations of peak time and increasing load. In fact, only the microservices experiencing a high load are scaled. This is also done automatically without human intervention via auto-scaling capabilities and self-healing.

What is Auto Scaling?

Auto scaling controls the number of instances of each microservice, depending on the need. That is, the instances of a microservice are increased during periods of high load while they are reduced during periods where a lower load is present. The question has always been when to scale up or down and the answer lies in application monitoring and container management using Kubernetes.

Microservices Scalability



For example, let's assume we have containerized microservices A, B and C and we are using Kubernetes as an orchestration platform. Kubernetes will be able to monitor microservices loads and automatically scale these up or down accordingly. As illustrated below, microservice C has a higher load than A and B which is why there are four instances for C and only two for the other microservices. During other times, microservice B could experience higher demand and would have 10 instances. All these dynamic changes in the number of instances are achieved with autoscaling using Kubernetes.

Guaranteed Up-time

With the right microservices architecture, it's almost impossible for the whole system to go down. Since each microservice is built, deployed and managed independently and is totally isolated at runtime, the whole system can therefore never go down. Should a microservice crash, this will only impact its own functionality, the rest of the system will behave normally. Furthermore, microservices platforms such as Kubernetes have high availability, load balancing and autoscaling capabilities that scale each microservice independently in the case of a high load when certain thresholds are reached.

In a monolithic architecture, when one function crashes due to performance or memory management issues, then the whole system is likely to go down whereas in a microservices world there will only be an error in processing this one function while the rest of the system will behave normally.

Not only that, developers will also be able to fix the issue, test and deploy the new release in a matter of hours instead of weeks or months. That is the benefit of time-to-market or fast deployment as discussed above.

No Vendor Lock

Microservices architecture and its principles are based on open standards and are strongly backed up by the open source community. Almost all commercial microservices platforms follow these open standards which makes it easy to migrate from them whenever necessary

As previously mentioned, each microservice is built and deployed independently with a bound context. So, by design, microservices have standard based APIs which makes it easy to extend and even use different programing languages and technologies.

You can easily build new modules and functionality that use existing services since the whole platform is open for extension. Also, those new modules can be deployed to the same microservices platform or even other external platforms. This is why there is no need to worry about vendor lock-in.

Resilience & High Availability

Kubernetes has an “auto-scaling” feature which is basically when Kubernetes automatically monitors the resources utilization of each microservice. It has the ability to increase the number of instances for a specific microservice based on its load and resource utilization and to load balance the traffic across the multiple instances of this microservice.

There is therefore no need to scale the whole platform to cater for peak times. This is done automatically and only the microservices that need to be scaled up and down based on the load are scaled.

What happens when a microservice fails unexpectedly? Not only does this not impact upon the whole application runtime, the platform also detects the failure and restarts the microservice. This is what we call “self-monitoring” and “self-healing”.

Kubernetes, OpenShift Container platforms or orchestrators have the capability to automatically detect failures and self-heal. These capabilities measure the system’s health thereby ensuring resilience and high availability.

An orchestration layer is fundamental to the self-monitoring and healing of microservices.

Dynamic Talent Pool

With microservices, it's easier to onboard new team members as well as hire software engineers with different coding backgrounds. Instead of a large complex monolithic system that needs a lot of knowledge transfer and code handover before a developer can start participating in the platform, in a microservices world it's a lot easier since the developer is building and deploying simple microservices isolated from the complexity of the rest of the platform.

A microservice is a black box to other services with a given standard API that is built once and reused across the organization. Also, building microservices involves simple programing that all software engineers have good knowledge of unlike other enterprise platforms that require specific training and certification.

It doesn't matter what language this service is written or designed in. For one coherent application, you can have microservices developed in different languages such as Java, .Net, etc. In that respect, there are no more boundaries when it comes to hiring new engineers to join your team. However, although this is technically feasible, this capability shouldn't be overused. You should unify your technology stack and different programing languages based on your needs as you don't want to end up with five or six programing languages that would be difficult to maintain.

MICROSERVICES USE CASES

- Legacy Transformation
- Agile Integration
- Build Cloud-Native Apps
- Digital Channels & Portals
- Multiple Teams

MICROSERVICES USE CASES

Legacy Transformation

Are your legacy systems lingering behind your digital transformation strategy? Do you currently face challenges in extending your legacy applications with new features?

Transforming your legacy applications over microservices architecture is one of the common use cases among organizations. With changing business needs, legacy applications reach a complex point where adding a new feature or update becomes challenging. It may take months to release this new change instead of hours or days due to the complexity of the system.

Microservices facilitate your modernization journey whether by refactoring, migrating or extending your legacy systems.

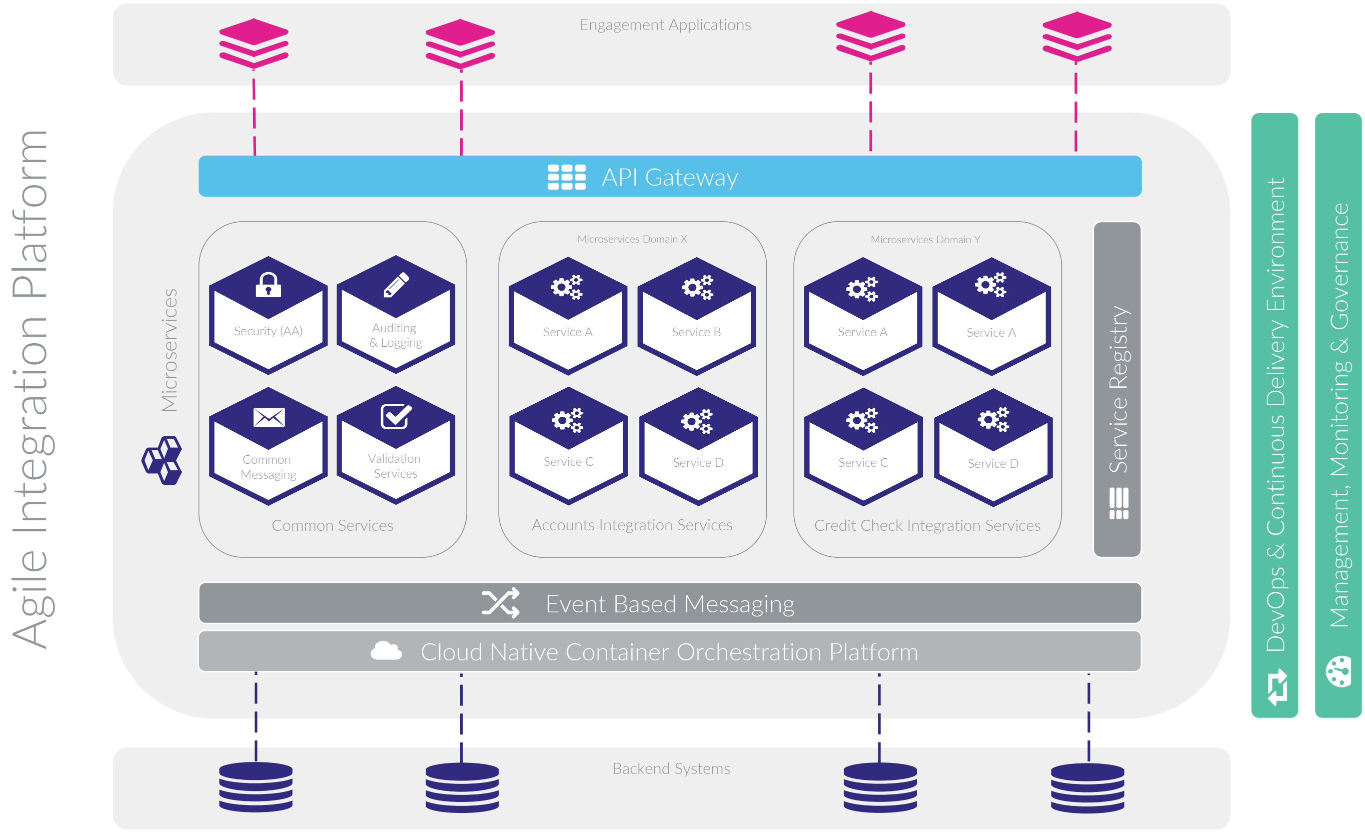
Agile Integration

Have you been facing challenges with your current integration methodology? Do you have proper updated and maintained documentation for your integration services? Does your architecture require a complete system integration testing if you alter one of your applications?

Agile integration transforms the traditional ESB methodology to a decentralized approach thereby enabling business agility and the rapid development of API and integration services. With microservices architecture, each integration service is built, tested, deployed and managed independently.

Similar to the nature of microservices, each integration service will be auto-scaled based on the needs of the workload. In addition to building integration services over microservices, an event-based architecture is used ensuring run-time decoupling. Instead of services directly calling one another, a service will publish an event in a queue which will be listened to and acted upon by a subscriber.

Agile Integration Platform



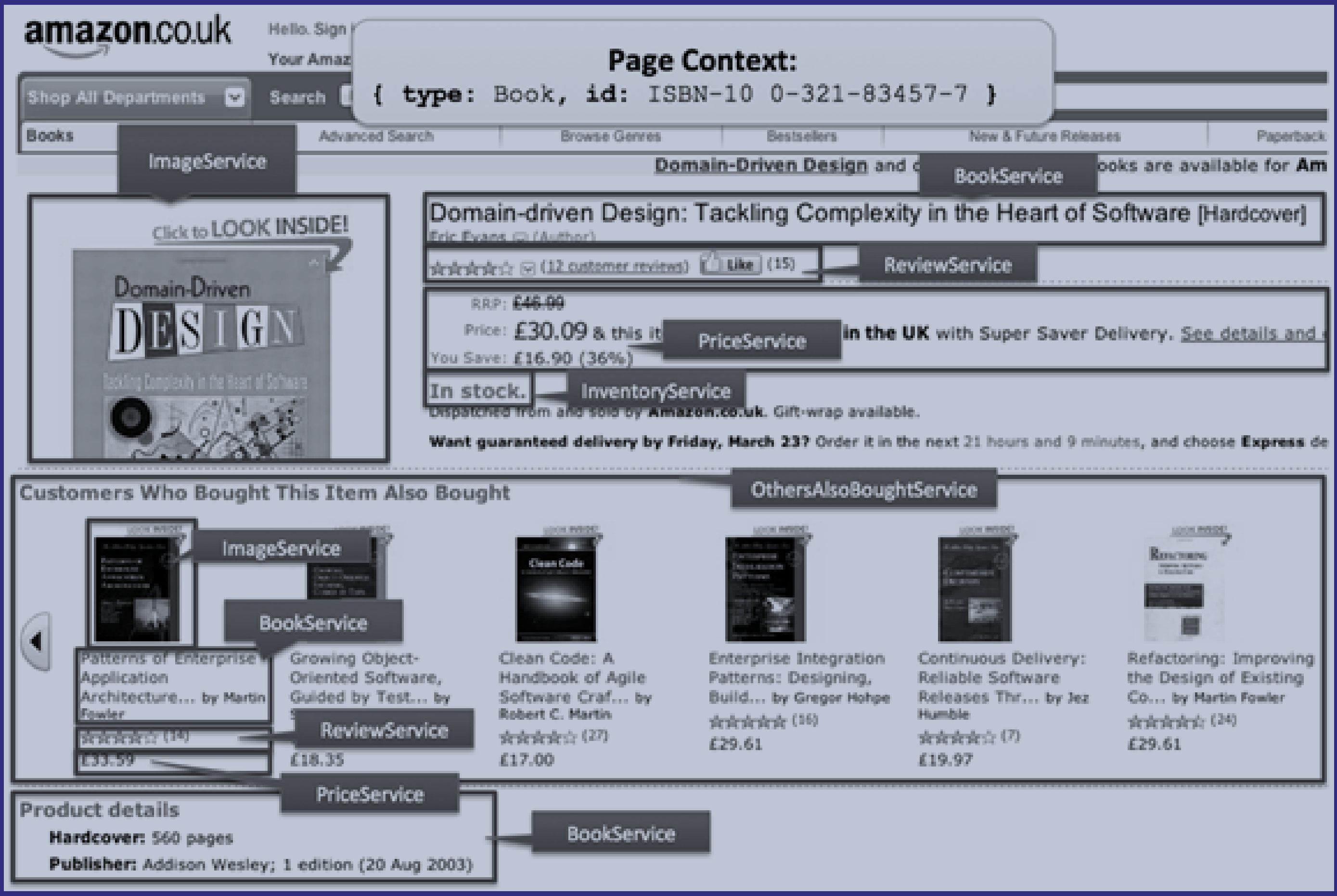
Build Cloud-Native Apps (Deliver FAST)

Cloud-native applications, small, independent and loosely decoupled services, are built over microservices architecture. The goal of cloud-native app development is to build and deliver fast new applications and features. The key characteristic of cloud-native applications is their flexibility to be built once and their ability to be run anywhere – private, public or hybrid clouds.

As mentioned previously, legacy transformation is essential for digital transformation to overcome the challenges of old-style software systems. Nevertheless, building fast applications is crucial especially if your organization is going to adopt cloud computing.

Digital Channels & Portals

Organizations are digitizing their services to offer exceptional customer experience and therefore digital channels and portals have become more and more important. In fact, building digital engagement platforms and e-Services is far more efficient with microservices architecture. Not only does it boost your time-to-market as discussed above but enables your team to be equipped with the necessary set-up for the fast shipment and delivery of new features and updates. In addition, portals built over microservices are more resilient, highly scalable and have the flexibility to integrate with external or third-party applications.



Page Context:

```
{ type: Book, id: ISBN-10 0-321-83457-7 }
```

Illustrated below is how and why microservices are the best method to build agile portals. This is a screenshot from an Amazon online store. As you can see, the book page contains information about the book's details, an image preview, price, reviews, inventory as well as guidance for customers showing what other books they may also be interested in.

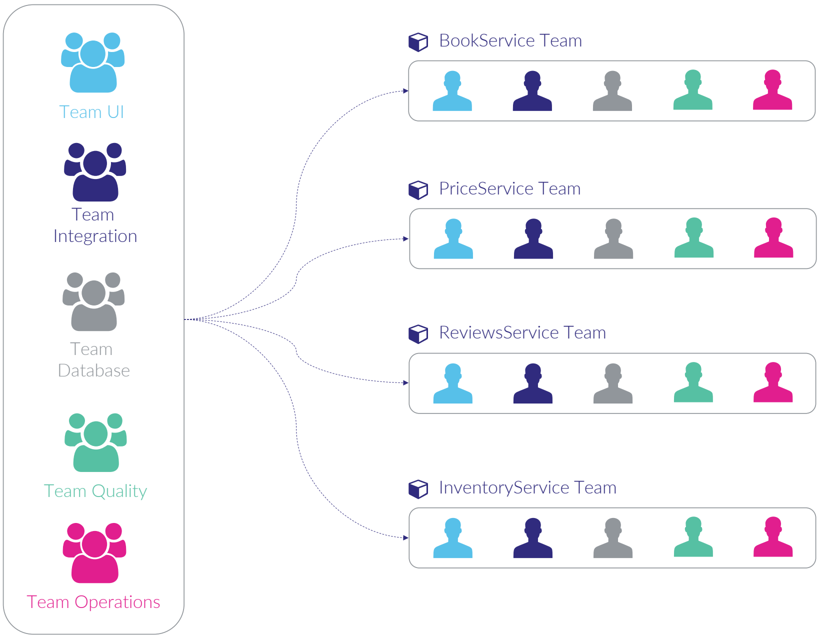
Each piece of aforementioned information is built as one microservice. Let's see how this impacts upon the portal flexibility and business agility:

- **Guaranteed up-time:** If the reviews service goes down, the entire page remains up and running apart from the user possibly seeing a message in the reviews box advising that book reviews are not currently available.
- **Scalability:** If the microservices orchestration platform detects a high-load on the inventory service, the service will be automatically scaled up depending on need.
- **Flexibility of Integration:** If the price service needs to integrate with Kindle APIs to retrieve information or the inventory service needs to provide stock information from an external warehouse system, each service is sufficiently flexible to integrate through APIs instead of integrating the whole portal.

Multiple Teams

Instead of having one team working on a build at any one time, microservices architecture allows multiple teams to work in parallel. Traditionally, functional teams work mutually exclusively on a build and the teams are formed in relation to UI, Integration/API, Database, Quality and Operations.

Instead of having one team working on a build at any one time, microservices architecture allows multiple teams to work in parallel. Traditionally, functional teams work mutually exclusively on a build and the teams are formed in relation to UI, Integration/API, Database, Quality and Operations.

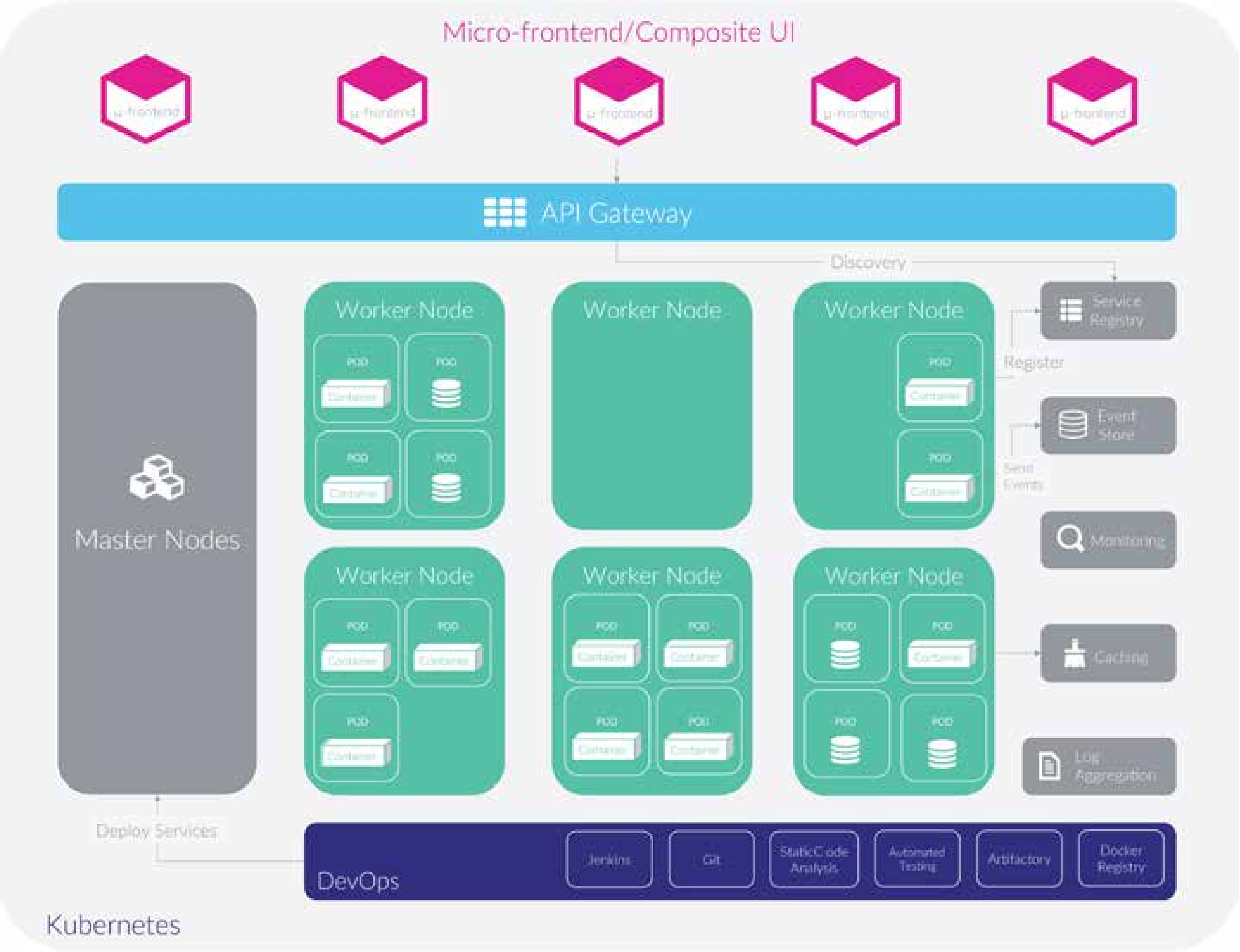


On the other hand, in a microservices architecture environment, teams are cross-functional. That is, each team is responsible for one service. As mentioned in the example above, services such as the book details, price, reviews and inventory are all independent microservices. Consequently, cross-functional teams can work in parallel rolling-out new features and updates. Hence throughput, i.e. the frequency of releases, is increased.

As illustrated below, the structure of the teams has been transformed to become cross-functional. The “BookService” team is responsible for all the tasks related to the book’s details in terms of UI, integration, data and operations. With this set-up, the “BookService” team can build, test and release in parallel with the “PriceService” team.

TECHNOLOGIES

TECHNOLOGIES



Docker – Container Platform

Since microservices are self-contained, independent application units, with each fulfilling only one specific business function, they can be considered small applications in their own right. What would happen if you created a dozen microservices for your app? And what if you decided to build several microservices with different technology stacks? Your team would soon be in trouble as developers have to manage even more environments than they would normally do with a traditional monolithic application. There's a solution though: using microservices and containers to encapsulate each microservice. Docker helps you to manage those containers. Docker is simply a container platform that was initially designed to provide a simpler way to handle containerized applications.

Virtual machines (VMs) were introduced to optimize the use of computing resources. You can run several VMs on a single server and deploy each application instance on a separate virtual machine. With this model, each VM provides a stable environment for a single application instance. Unfortunately, however, when the application is scaled, issues with performance will soon be encountered since VMs still consume a lot of resources.

Because microservices are similar to small apps, microservices must be deployed to their own VM instances to ensure discrete environments. And, as you can imagine, dedicating an entire virtual machine to deploying only a small part of an app isn't the most efficient option. With Docker, however, it's possible to reduce performance overhead and deploy thousands of microservices on the same server since Docker containers require a lot fewer computing resources than virtual machines.

Docker is an excellent tool for managing and deploying microservices. Each microservice can be further broken down into processes running in separate Docker containers which can be specified with Dockerfiles and Docker Compose configuration files. Combined with a provisioning tool such as Kubernetes, each microservice can then be easily deployed, scaled and collaborated on by a developer team. Specifying an environment in this way also makes it easy to link microservices together to form a larger application.

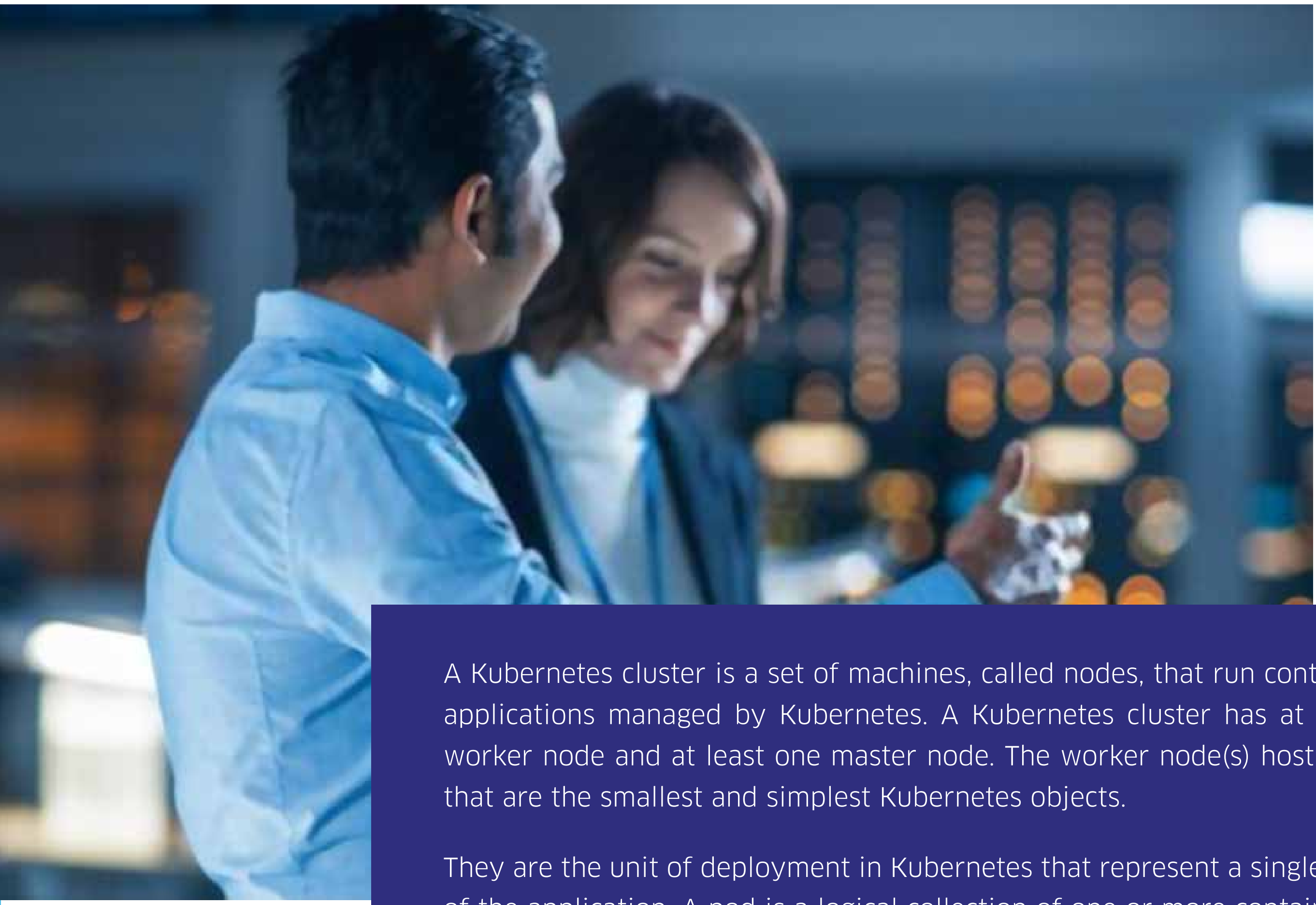
Docker's advantages

- Faster start time. A Docker container starts in a matter of seconds because a container is just an operating system process. A virtual machine with a complete OS can take minutes to load.
- Faster deployment. There's no need to set up a new environment; with Docker, web development team members only need to download a Docker image to run it on a different server.
- Easier management and scaling of containers. You can destroy and run containers faster than you can destroy and run virtual machines.
- Better usage of computing resources. You can run more containers than virtual machines on a single server.
- Support for various operating systems: You can get Docker for Windows, Mac, Debian and other OSs.

Kubernetes – Container Orchestration

Kubernetes is an open-source orchestrator, grouping systems together to form clusters. In these clusters, the deployment and management of containers is automated at scale while meeting fault-tolerance, on-demand scalability, optimal resource usage, accessibility from the outside world and seamless updates/rollbacks without any downtime.

Google open-sourced the Kubernetes project in 2014. Kubernetes therefore builds upon a decade and a half of Google's experience of running production workloads at scale, combined with best-of-breed ideas and practices from the community.



A Kubernetes cluster is a set of machines, called nodes, that run containerized applications managed by Kubernetes. A Kubernetes cluster has at least one worker node and at least one master node. The worker node(s) host the pods that are the smallest and simplest Kubernetes objects.

They are the unit of deployment in Kubernetes that represent a single instance of the application. A pod is a logical collection of one or more containers. The master node(s) manages the worker nodes and the pods in the cluster.

Kubernetes Capabilities

- **Horizontal Scaling:** Kubernetes helps scaling up or down applications based on the load on each application and predefined rules based on metrices. Scaling up and down means creating more instances on demand and load balancing traffic between them.
- **Self-healing:** Kubernetes monitors containers, self-heals and ensures up-time. The system restarts failed containers, replaces, reschedules or kills containers that don't respond to user-defined health check.
- **DevOps:** The platform provides the devOps tools to automate, scale and build resilient applications. Kubernetes allows applications to be deployed anywhere independent of the underlying infrastructure.
- **Containers Provisioning:** Kubernetes places containers based on their needed resources. It automatically calculates the best location for containers; ensuring better utilization for resources.
- **Automated rollouts and rollbacks:** If changes are rolled out and something goes wrong with an application, Kubernetes will rollback the changes based on predefined rules and configuration.

Micro-Frontends

Micro-frontends is an architecture pattern around decomposing frontend monoliths into smaller, simpler chunks that can be developed, tested and deployed independently while still appearing to customers as a single cohesive platform. You can think of this as breaking down a single page into smaller widgets or building blocks where each block is developed, tested and deployed independently. All these “micro frontends” are geared up to be displayed as one page.

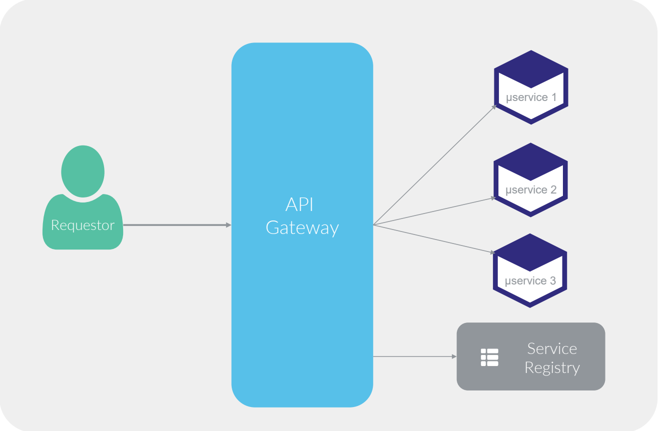
Micro-frontends is the visualization of a website or web app as a composition of features that are owned by independent teams. Each team has a distinct area of business or mission it cares about and specializes in. To achieve this, we have two options: server-side composition and client-side composition.

The benefits of micro frontends are:

- Smaller and maintainable frontend services
- Faster delivery and updating of features
- More scalable organizations with decoupled autonomous teams
- Ability to update or even rewrite parts of the frontend in a more incremental agile manner

API Gateway

An API gateway is a core component of any microservices platform. Since almost all microservices expose their services and interactions through APIs, the API gateway acts as the single entry point for any microservice. The API gateway encapsulates all the detailed service information and system design providing a single entry point for this to the system. The API GW has other responsibilities such as authentication, monitoring, load balancing, caching, request shaping and management and static response handling.



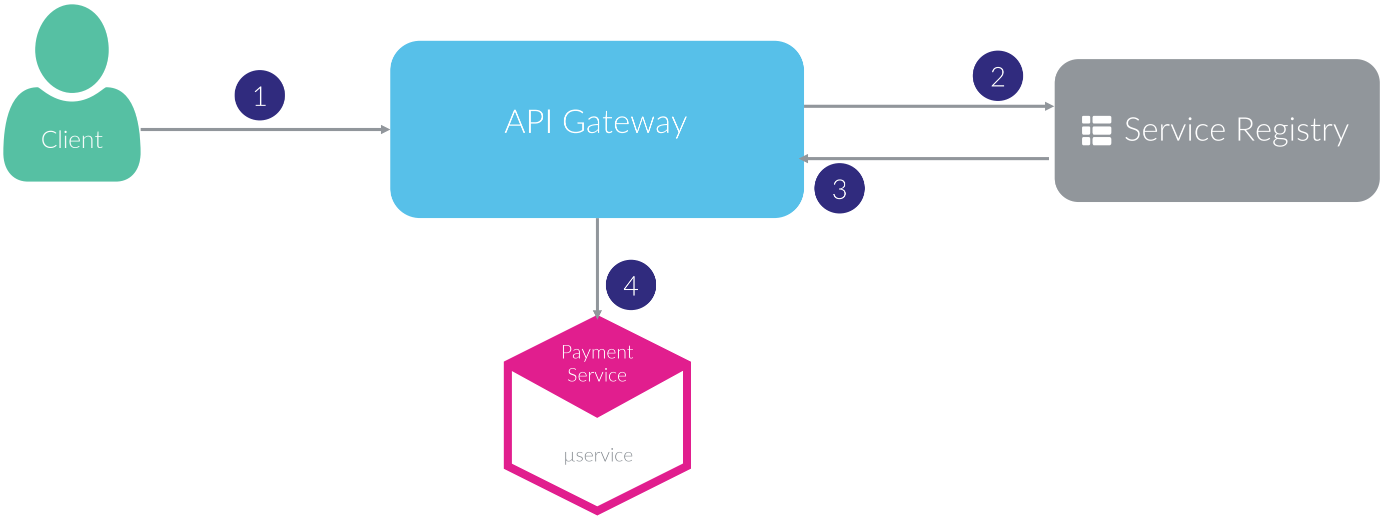
The API Gateway is responsible for request routing. All requests from clients first go through the API Gateway which then routes requests to the appropriate microservice. The API Gateway can handle a request by invoking multiple microservices and aggregating the results. It can translate between web protocols and also undertake message transformation and enrichment such as adding information to the message header or body.

Service Registry

The service registry is a database that contains information about all the microservices on a specific microservices platform. All microservices are registered in this database where the API gateway dispatches requests based on the information registered.

The service registry has two main responsibilities:

1) Registration of services: whenever a new service is added or updated, the new information should be registered in the service registry.
2) Discovery: this is the counterpart of registration, when a client wants to access a service, information such as location, parameters, version and so forth comes from the service registry.

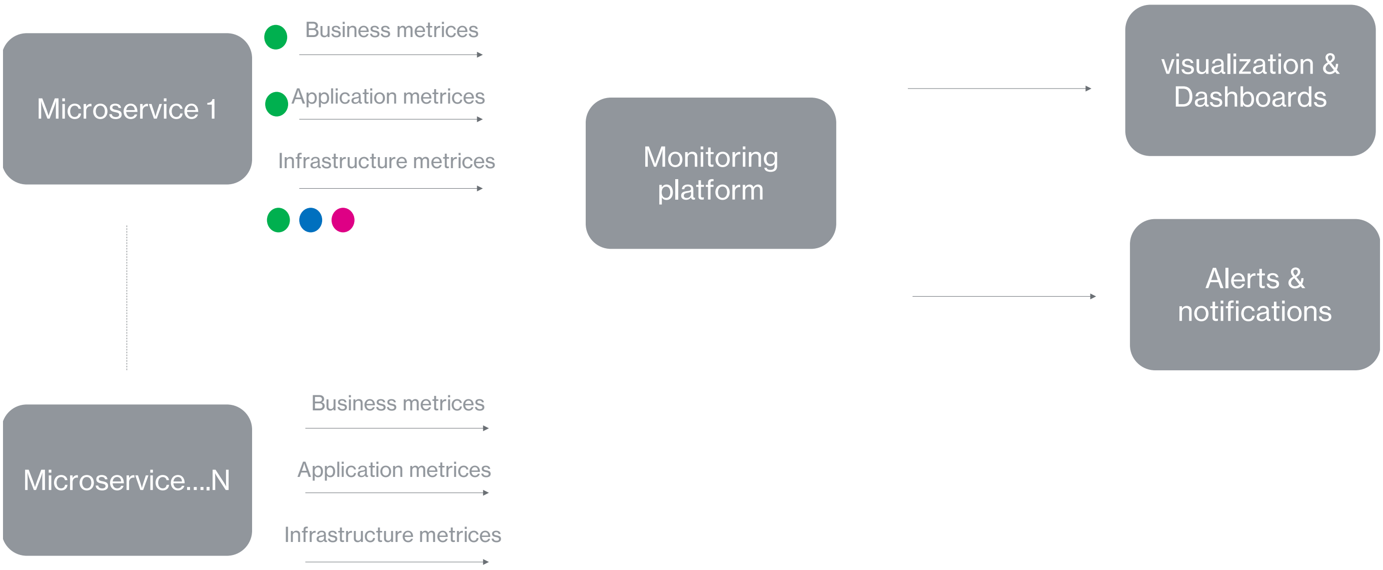


Monitoring

When considering microservices, one of the biggest challenges you think of is monitoring. Unlike the traditional large monolithic systems, in microservices architecture there are hundreds of small services where each one needs to be monitored. If you have the right microservices monitoring design and tools in place you can detect and troubleshoot issues more easily than with a traditional system. You can build much more resilient platforms.

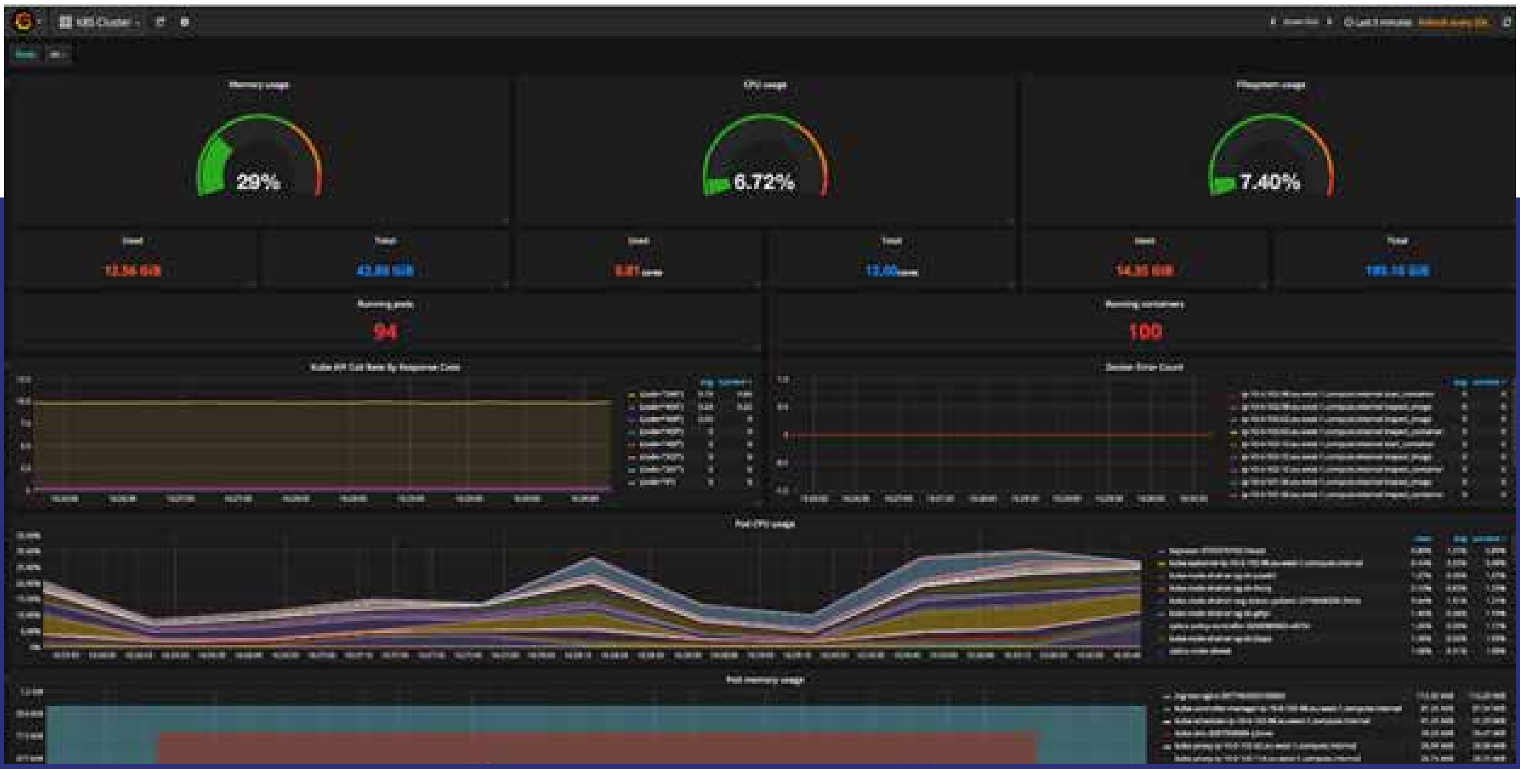
In a microservices world, each microservice is monitored independently. Each microservice exports its metrices; those metrices form the infrastructure such as memory, processing, etc, and applications such as the number of transactions, business exceptions, etc., all of which are normally aggregated in a monitoring platform where you can visualize, send alerts and configure automated actions.

Microservices Monitoring



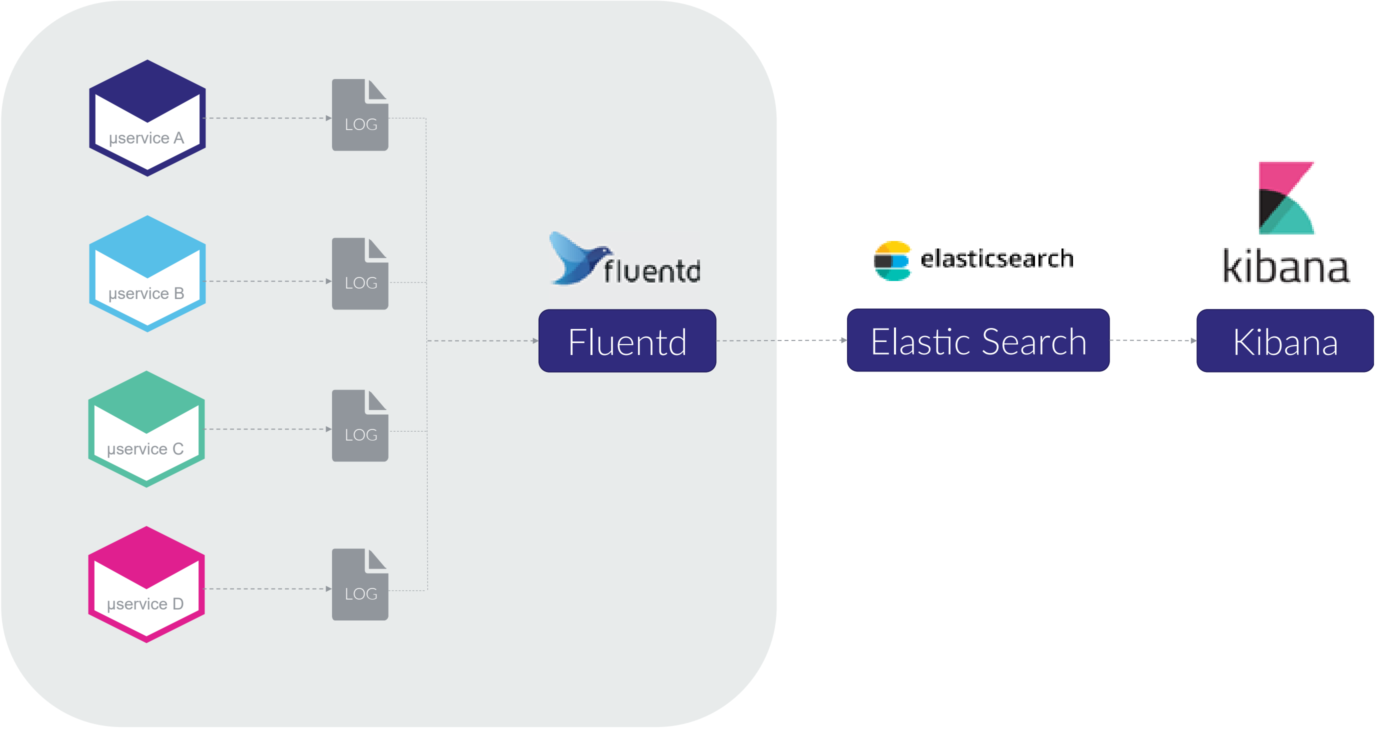
Monitoring Infrastructure

Prometheus is an application used for event monitoring and alerting. Like Kubernetes, it is a Cloud Native Computing Foundation Project and it ships with an extensive set of preconfigured Grafana reports and dashboards that help to monitor Kubenetes in real-time and proactively alert administrators should any issues be detected.



Collecting and Monitoring Application logs

In a microservices world, each microservice has its own logs. So, with multiple services and applications on a Kubernetes cluster, a centralized, cluster-level logging stack can help you quickly sort through and analyze the heavy volume of log data produced by your services. Elasticsearch, Fluentd and Kibana (EFK) is the stack that aggregates logs from all services into Elasticsearch and provides Kibana UI with the ability to view any logs.



Through this stack you can view your logs more easily, search for specific time periods and view specific events and exceptions when they occurred. It gives you far better visibility and more control over your application.

Events Streaming and Sourcing

Event driven design is a very common pattern when it comes to microservices architecture. Since each microservice is totally independent, we want it to stay loosely coupled with its business logic and data and we don't want high dependability between services and point to point interactions, these aspects became the focus of the event driven design.

The idea behind this is that each service publishes an event with the action it performed to an event streaming platform and then any other service can consume this event to perform its desired objective. Unlike direct REST calls,

services that create requests do not need to know the details of the services that consume the requests. By having this facility, you can add new services that take actions based on the event without the need for the providing service to undergo any changes or even have knowledge of the new service.

Furthermore, the application can reconstruct an entity's current state by replaying the events. Applications preserve events in an event store, a database of events, which has an API for adding and retrieving an entity's events. The event store also behaves like a message broker.

The benefits of using events streaming:

- Asynchronous communication between services
- Loose coupling
- Ease of scaling
- Events replay and recovery

Apache Kafka

Apache Kafka is a horizontally scalable, fault tolerant and fast messaging system. It's a pub-sub model in which various producers and consumers can write and read. It decouples source and target systems. Some of the key features are:

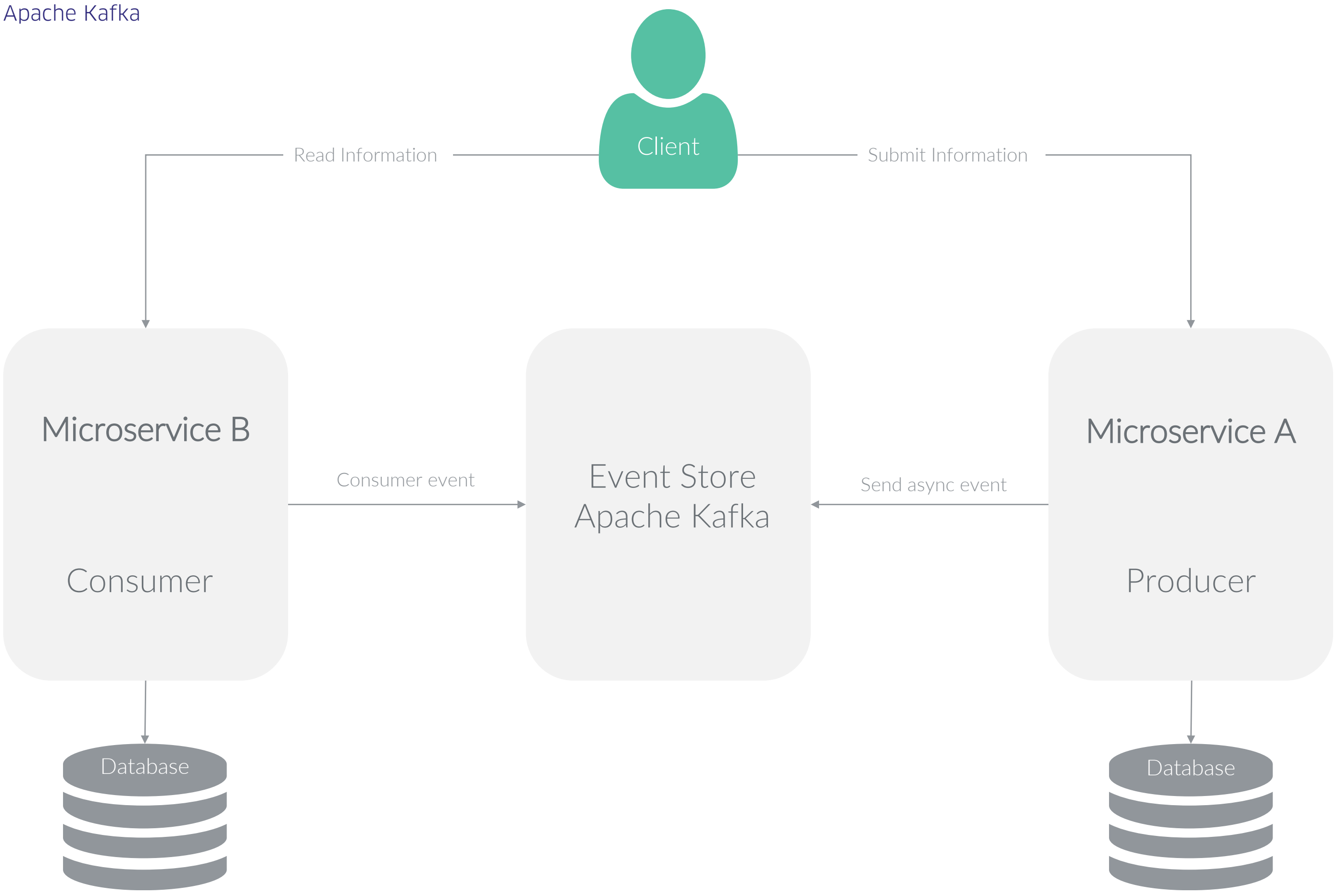
1 Scale to hundreds of nodes

2 Ability to handle millions of messages per second

3 Real-time processing (~10ms)

4 Events storing and replay

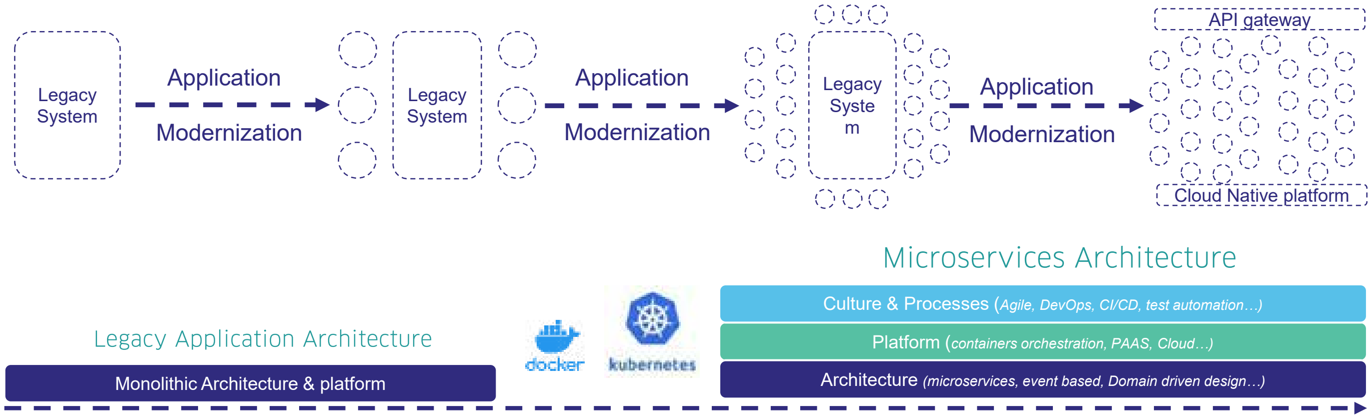
Apache Kafka



CASE STUDY

CASE STUDY

The Journey of Legacy Modernization to Microservices


Legacy System
Application
Modernization
Legacy System
Application
Modernization
Legacy Syste m
Application
Modernization
API gateway
Cloud Native platform
Microservices Architecture
Legacy Application Architecture
docker
kubernetes
Culture & Processes (Agile, DevOps, CI/CD, test automation…)
Platform (containers orchestration, PAAS, Cloud…)
Monolithic Architecture & platform
Architecture (microservices, event based, Domain driven design…)

A leading mortgage financing institution, serving citizens on low and middle income, faced several challenges with its existing legacy core system which was more than 20 years old. The system had continued to grow over the years with the addition of new features incrementally serving the ongoing business demands of the organization. The system grew exponentially until it eventually reached a stage where adding new features and changing existing ones was very costly, risky and time-consuming.

Here is a summary of the challenges they faced:

Not responsive to business demand

The current system was too complicated to add new features to it. Implementing new features was time-consuming in terms of development, testing and release time.

Change was risky

The current legacy code was complex to implement new changes as well as testing being time-consuming and risky.

Scalability

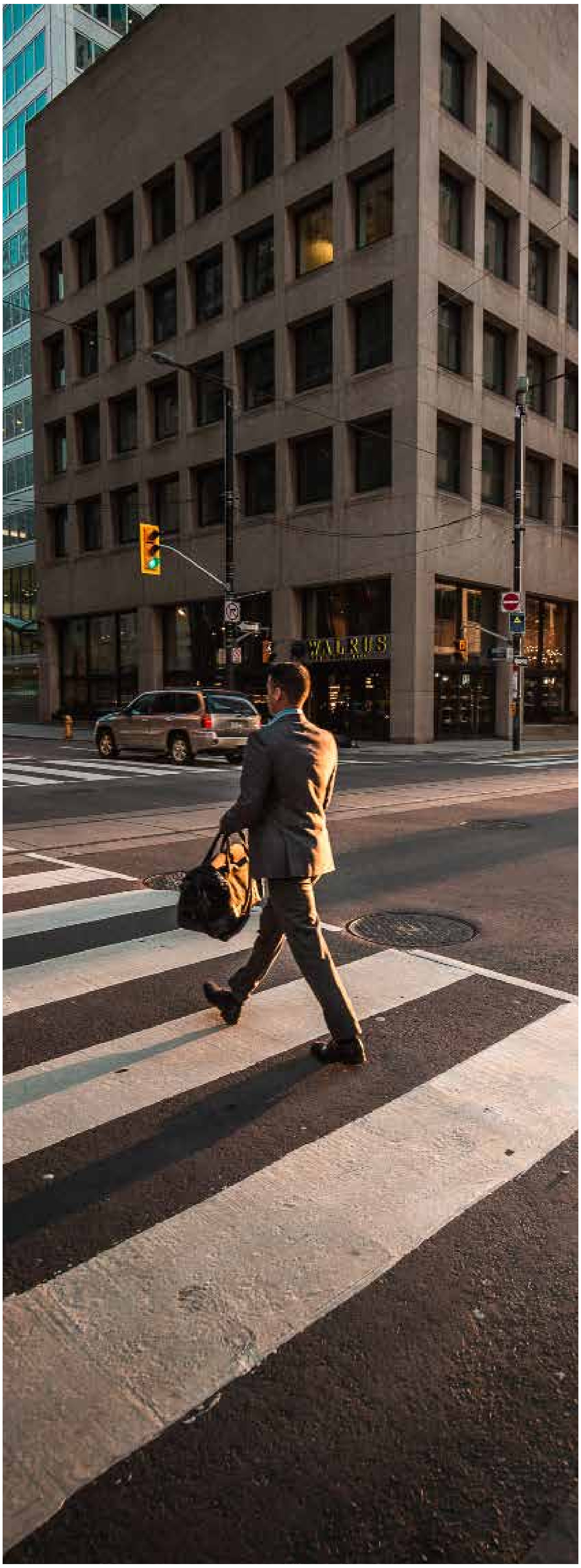
The current system was processing hundreds of thousands of transactions and the database contained millions of records. It was difficult to scale the system during peak times and when faced with a sudden high load.

APIs & Integration

Exposing APIs and integrating with external systems was challenging, complex and time-consuming.

Solution and approach

After assessing the situation, we decided to migrate the system to microservices architecture running on containers in a private cloud container orchestration environment.



Solution highlights:

- **Phased Agile Approach:** We started to break down the legacy system into smaller components, “microservices”, migrating these components to containers and exposing them as REST APIs. This was a low risk approach where the two systems co-existed together migrating components in sprints every two to four weeks.
- **Quick time to market:** Implementing new features and deploying these to production improved by 4-5x faster. All new components had automated DevOps pipelines with built test automation thus achieving quality at speed.
- **Performance Boost:** We moved those components with scalability issues and performance bottlenecks thereby offloading the legacy system to the modern scalable platform. Each microservice can now scale independently based on its load at runtime and there is no need to scale the whole system.
- **API first:** Easily integrated with internal systems such as digital channels and external third party systems and banks.

Authors:

- Dina Hemimy, Technology Consultant & Business Development Manager **(dhemimy@sumerge.com)**
- Khaled Sinbawy, Solution Architect **(ksinbawy@sumerge.com)**
- Mohamed Nour, Chief Technology Officer **(mnour@sumerge.com)**

ABOUT SUMERGE

We're a technology company yet our role is beyond offering technology; we combine innovation with solid technology expertise and deep business understanding to transform how you do business. As your partner, we fully integrate your business needs and industry requirements with our team's expertise, innovation and passion to take your business through a unique digital transformation journey.

Significant change, positive impact and passion are our fuel. We have a unique culture reflecting the way we think and act. A culture that encourages freedom and responsibility, high performance, customer centricity and innovation.

2005
established since

250+
Successful projects

13+
Countries Served

5M+
System Users

sumerge

FOLLOW US