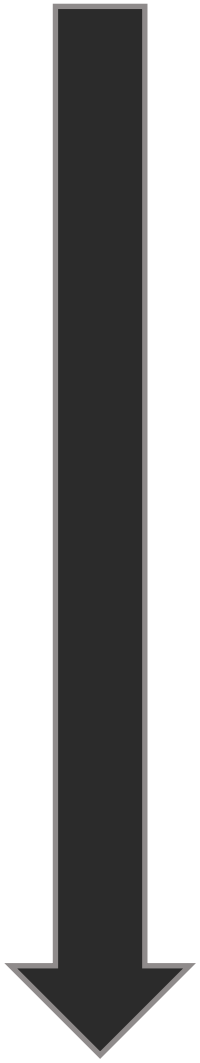
Enterprise App Modernization & Microservices

Raghavendra Deshpande
Developer Advocate, IBM @ragdeshp

IBM **Developer**

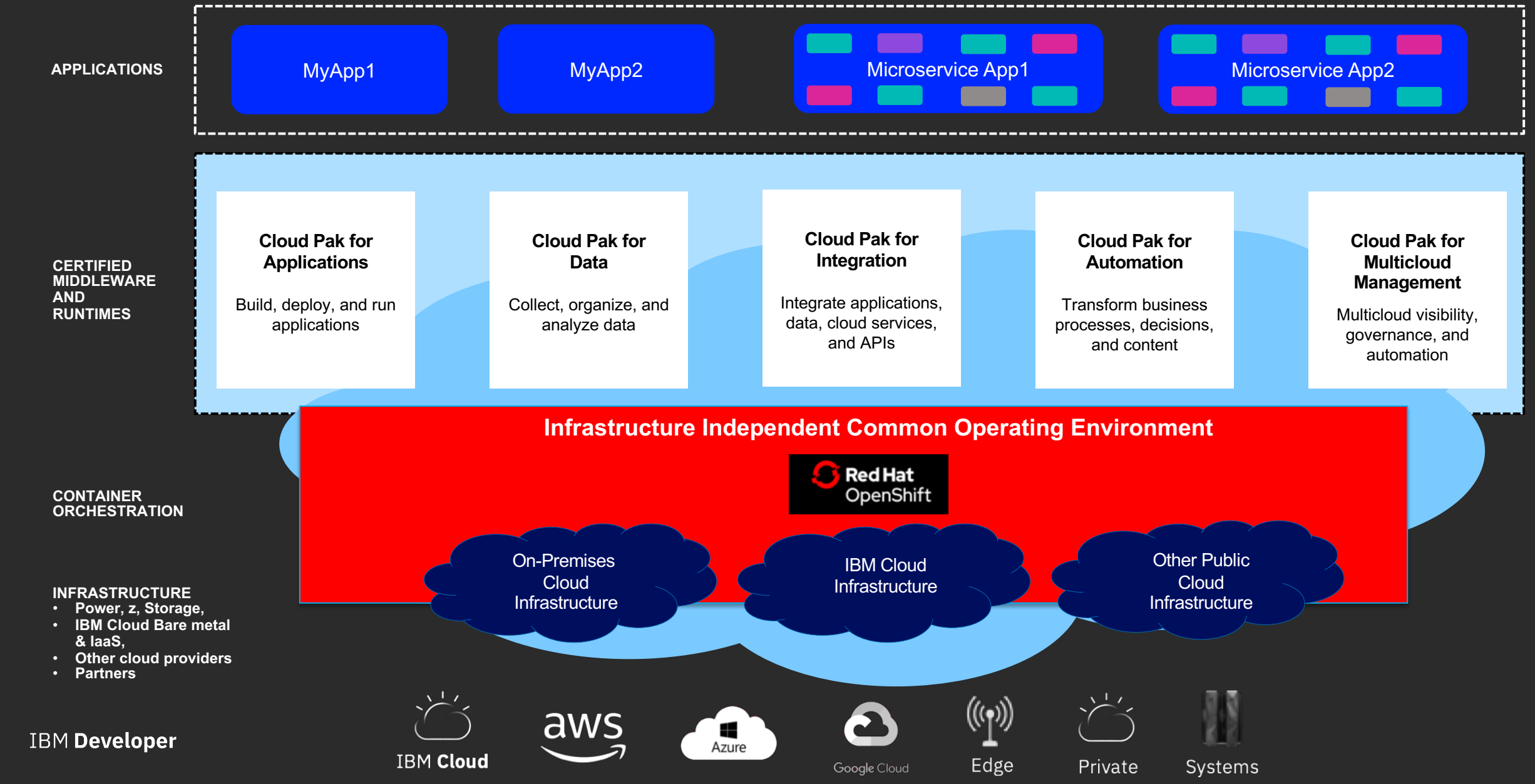
App modernization is inevitable

Evolution of application architectures



Late 90’s	**Enterprise Application (EAI) Services and Models** Addressed integration and transactional challenges primarily by using message oriented middleware. Mostly proprietary systems needing a proliferation of custom interfaces.
Mid 00’s	**Service Oriented Architectures** Based on open protocols like SOAP and WSDL making integration and adoption easier. Usually deployed on an Enterprise ESB which is hard to manage and scale.
Early 10’s	**API Platforms and API Management** REST and JSON become the defacto standard for consuming backend data. Mobile apps become major consumers of backend data. New Open protocols like OAuth become available further simplifying API development .
2015 and beyond	**Microservice Architecture** Applications are composed of small, independently deployable processes communicating with each other using language-agnostic APIs and protocols.

Hybrid Cloud Platform Architecture



Moving apps to modern architectures starts with planning

Assessment for each legacy app

i. **Business value**

- Measure cost of migration vs benefits gained by migrating
  - Good value usually found in apps that have
    - ✓ Multiple upgrades per year
    - ✓ Lots of business rules requiring complex regression testing and extended service outages to implement
    - ✓ Large, stable (or growing) user base

ii. **Technical effort**

- This is where Transformer Advisor helps

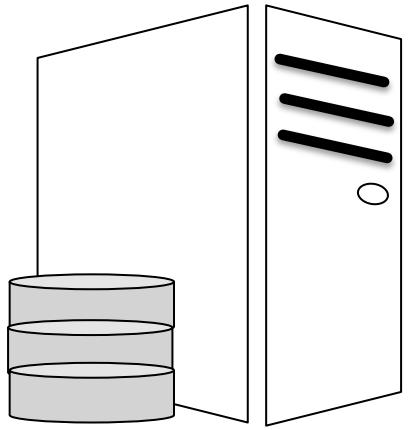
Best candidates are those with relatively high business value and relatively small technical effort

Three ways to migrate...

Replatform

IBM Cloud

Repackage

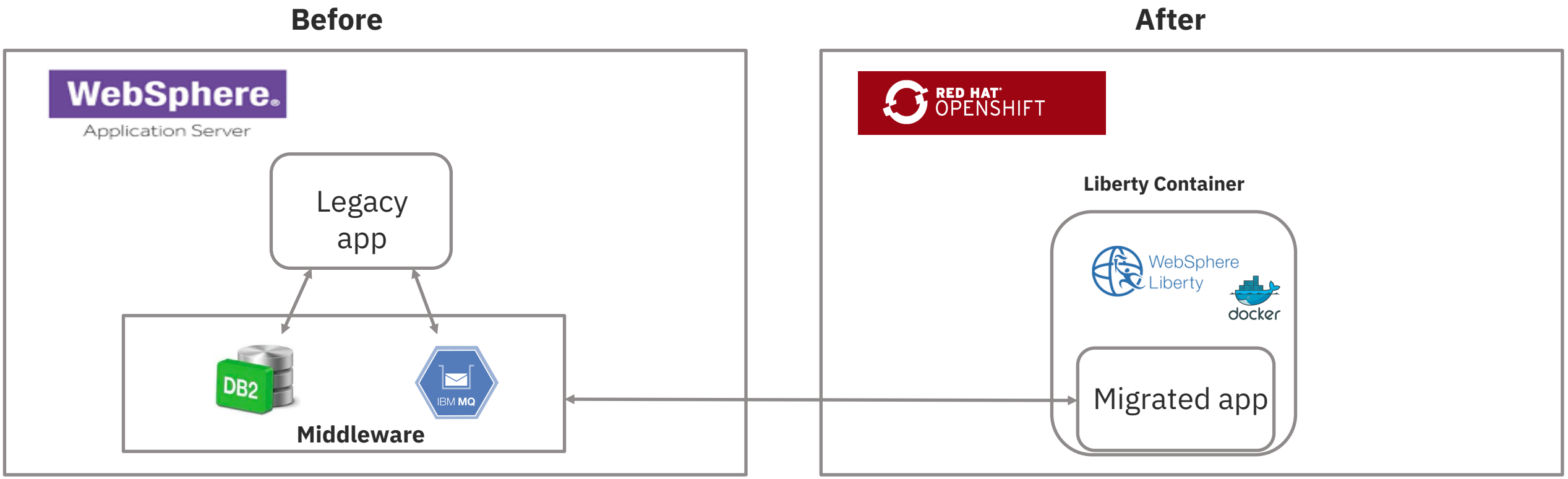


Refactor

Refactor what is necessary, but don’t necessarily refactor

Replatform

- Legacy app migrated in full from WebSphere Application Server Traditional to WebSphere Liberty running in a Kubernetes cluster
- Transformation Advisor provides guidance with equivalent Liberty config for migrated app
- Associated middleware is left in place and still used by migrated app



Containers and Docker

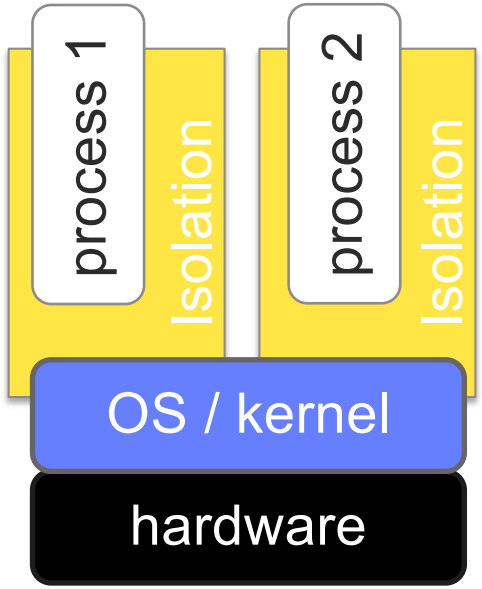
Containers – not a new idea

- chroot ('80s) process spawned in isolated file space
- FreeBSD jails
- OS-level virtualization (user-mode-linux, virtuozzo)
- Solaris Containers
- LinuX Containers (LXC)
- Cloud Foundry (Warden, Garden)

More efficient than VMs but less mindshare...

Docker – ecosystem approach transformed perception

- application container image
- Docker Hub registry (public) for sharing images
- engine widely available



How do Containers work?

Similar to VMs but managed at the **process level**

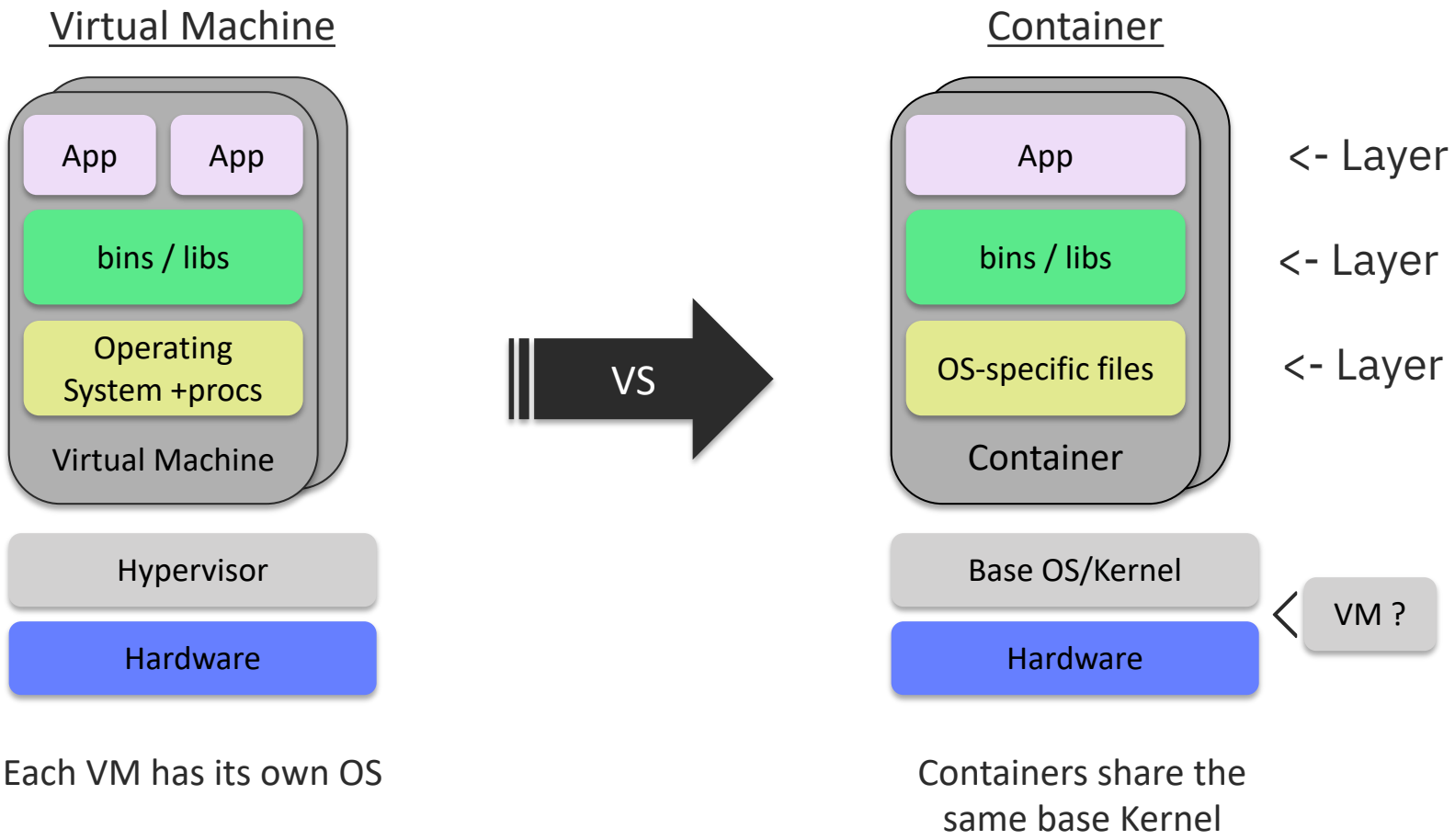
"VM-like" isolated achieved by set of "**namespaces**" (isolated view)

- PID –isolated view of process IDs
- USER- user and group IDs
- UTS - hostname and domain name
- NS - mount points
- NET - Network devices, stacks, ports
- IPC - inter-process communications, message queues

cgroups - controls limits and monitoring of resources

The key statement: **A container is a process(es) running in isolation**

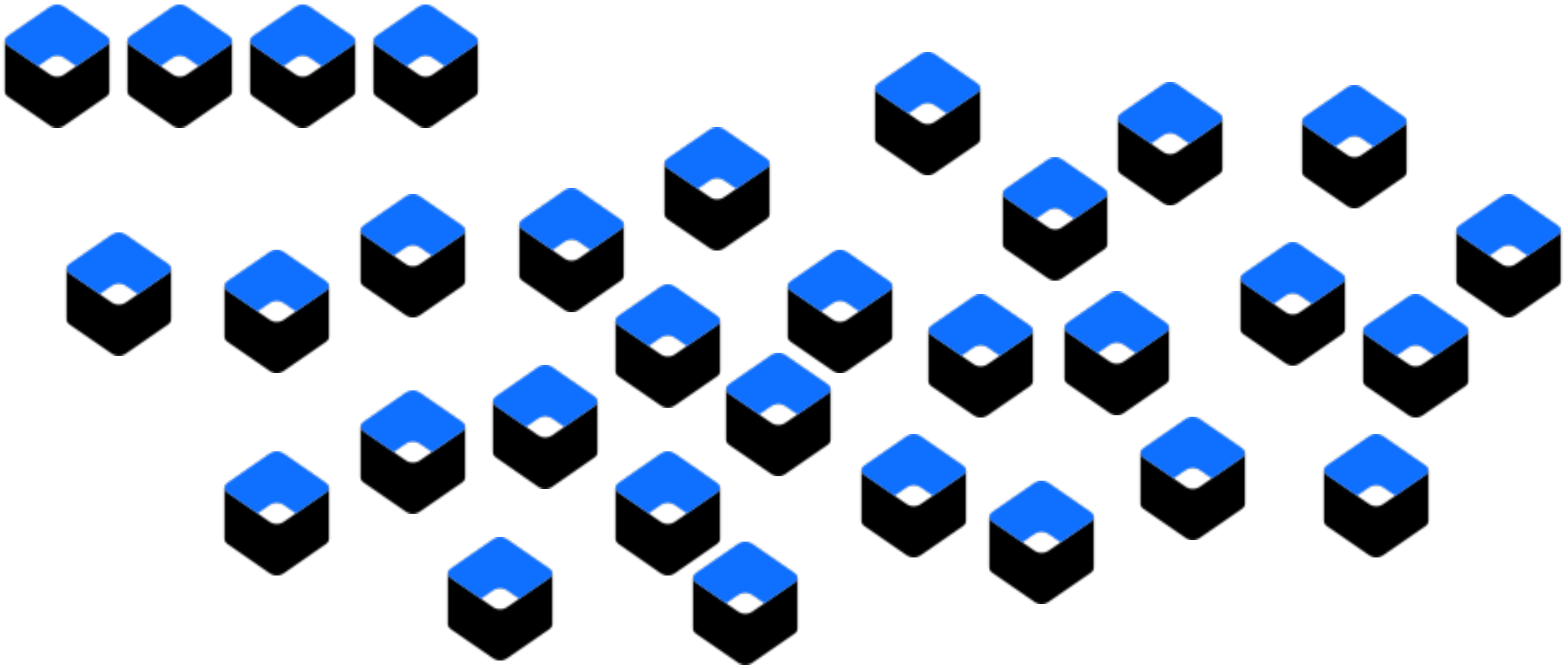
VM vs Container: Notice the layers!



Each VM has its own OS

Containers share the same base Kernel

Managing just a few containers is easy...



But that doesn't reflect a real workload

Container orchestration with Kubernetes unlocks value of containers as application components

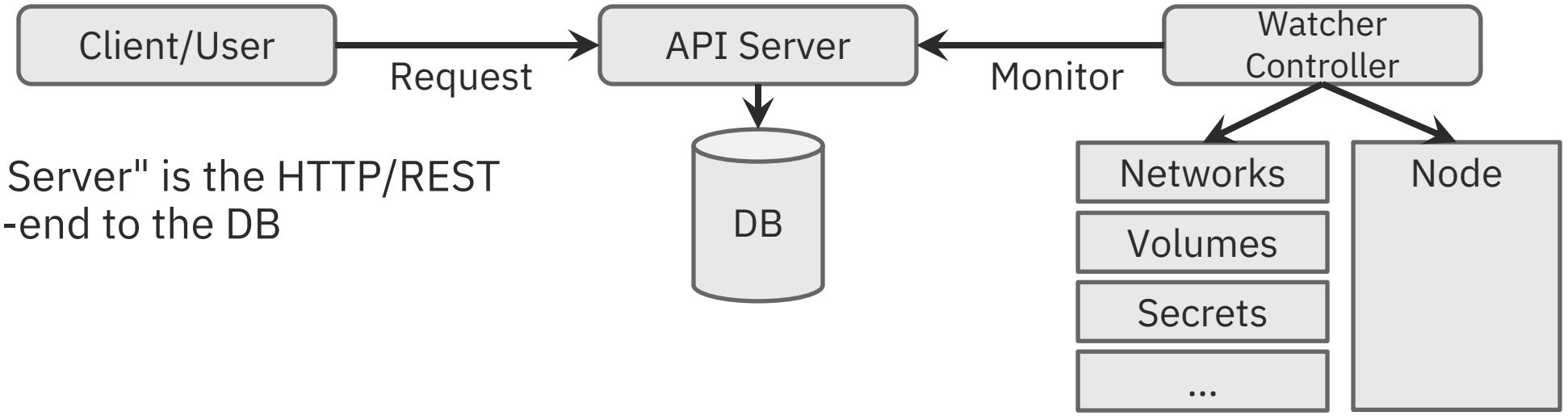
What is Kubernetes?

- **Container Orchestrator**
  - Provision, manage, scale applications
- Manage infrastructure resources needed by applications
  - Volumes
  - Networks
  - Secrets
  - And many many many more...
- Declarative model
  - Provide the "desired state" and Kubernetes will make it happen
- What's in a name?
  - Kubernetes (K8s/Kube): "Helmsman" in ancient Greek

Kubernetes Architecture

- At its core, Kubernetes is a database (etcd).
  With "watchers" & "controllers" that react to changes in the DB.
  The controllers are what make it Kubernetes.
  This pluggability and extensibility is part of its "secret sauce".

- DB represents the user's desired state
  - Watchers attempt to make reality match the desired state



"API Server" is the HTTP/REST front-end to the DB

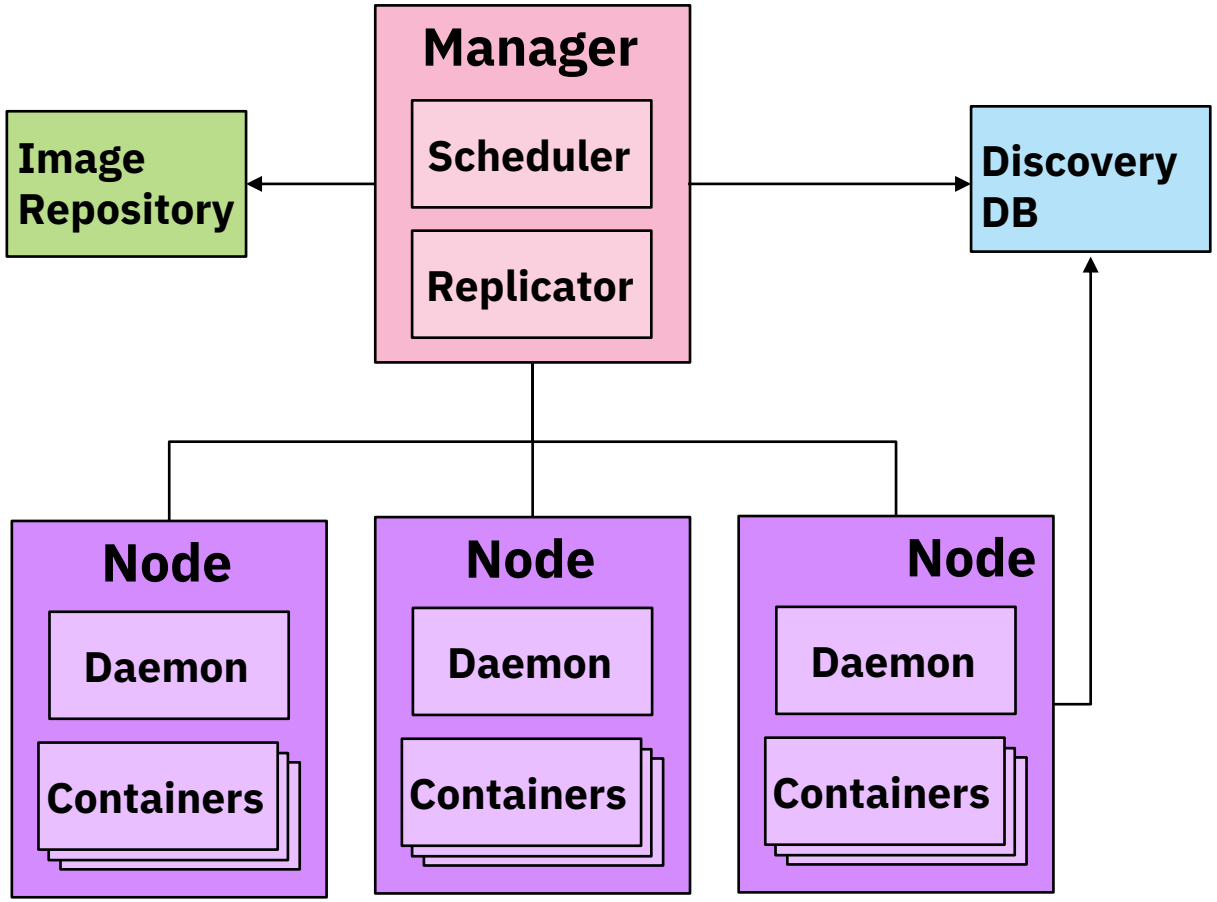
Benefits of Container Orchestration

Automated scheduling and scaling

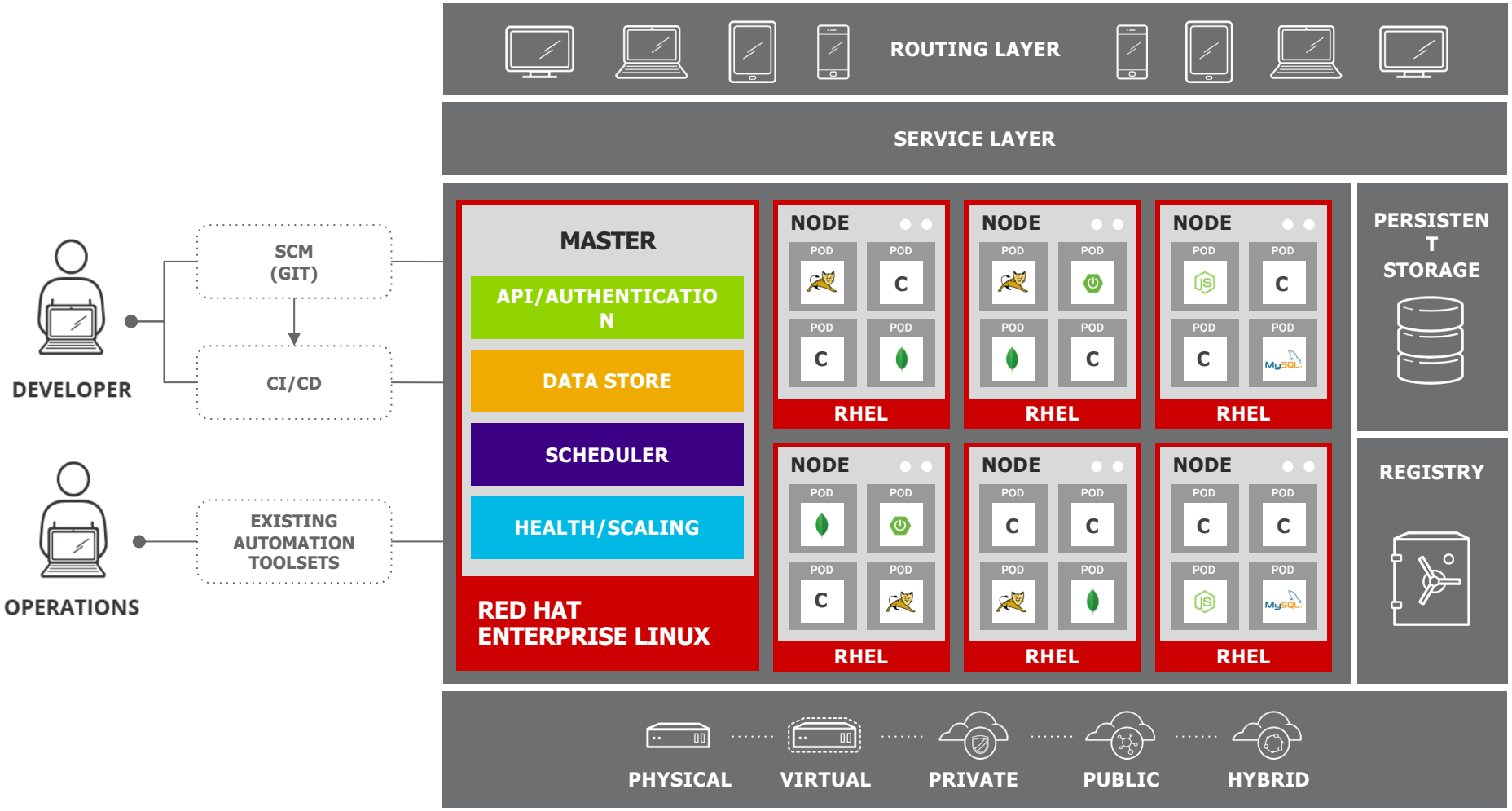
Zero downtime deployments

High availability and fault tolerance

A/B deployments



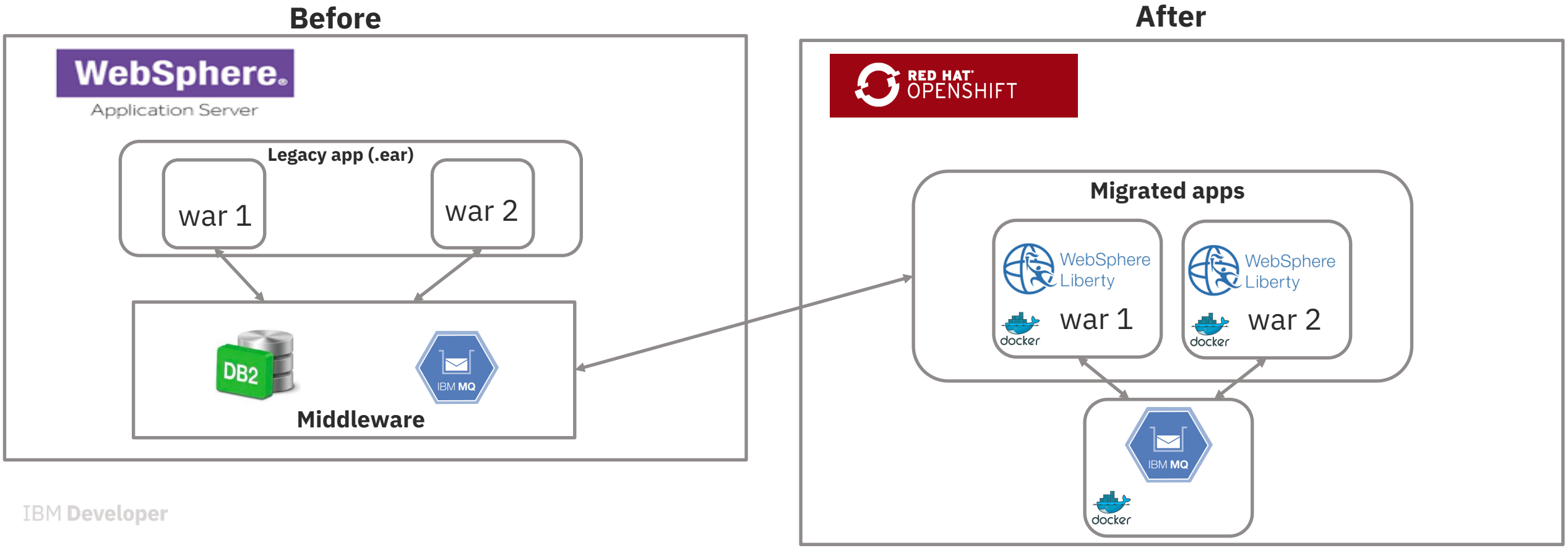
Red Hat OpenShift – Kubernetes for Enterprise



Microservices are an architectural style that realizes the full benefits of cloud native

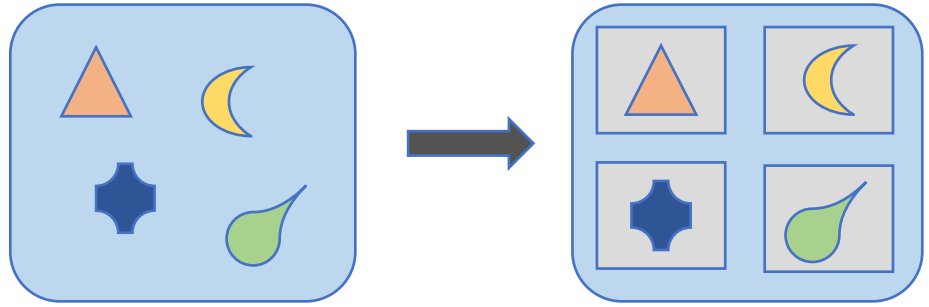
Repackage to “macroservices”

- Components (.war files) of legacy app (.ear) migrated to separate Liberty containers
- DB2 is left in place and still used by migrated app
- IBM MQ is partitioned
  - Container version handles communications between migrated modules
  - Legacy version handles communications between migrated modules and other legacy apps



Key tenets of a microservices architecture

1. **Large monoliths are broken down into many small services**
   - Each service runs into its own process
   - Generally accepted rule is one service per container
2. **Services are optimized for a single function**
   - One business function per service
     - The service will have only one reason to change



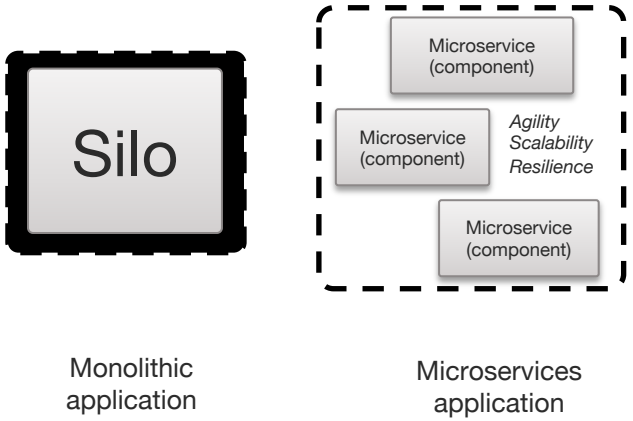
3. **Services are tightly encapsulated behind concrete programming interfaces**
   - Have to balance between evolving the interface and maintaining backward compatibility
4. **Communication via REST API and/or message brokers**
   - Avoids tight coupling and allows for flexibility of synchronous and asynchronous access
5. **Per-service continuous integration and continuous deployment (CI/CD)**
   - Services can evolve at different rates
6. **Per-service HA and clustering**
   - Services can be scaled independently at different rates as needed

Microservices – Fundamentals

Microservices Architecture

Simplistically, microservices architecture is about breaking down large silo applications into more manageable fully decoupled pieces

SOA done well



Monolithic application

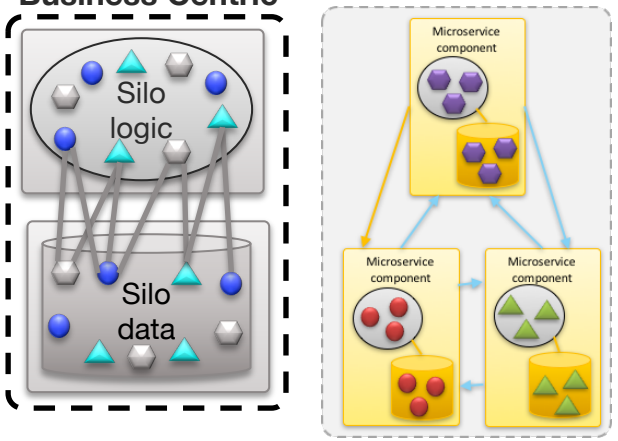
Microservices application

A *microservice* is a granular decoupled component within a broader application

Encapsulation is key

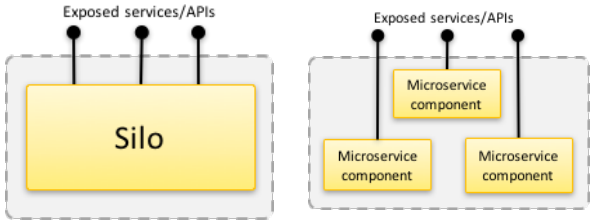
Related logic and data should remain together, and which means drawing strong boundaries between Microservices.

Business Centric



Microservice as API enabler

The “service” in “microservice” refers to the granularity of the **components**, not the exposed interfaces



An application split into Microservices may well expose the same APIs as its monolithic equivalent

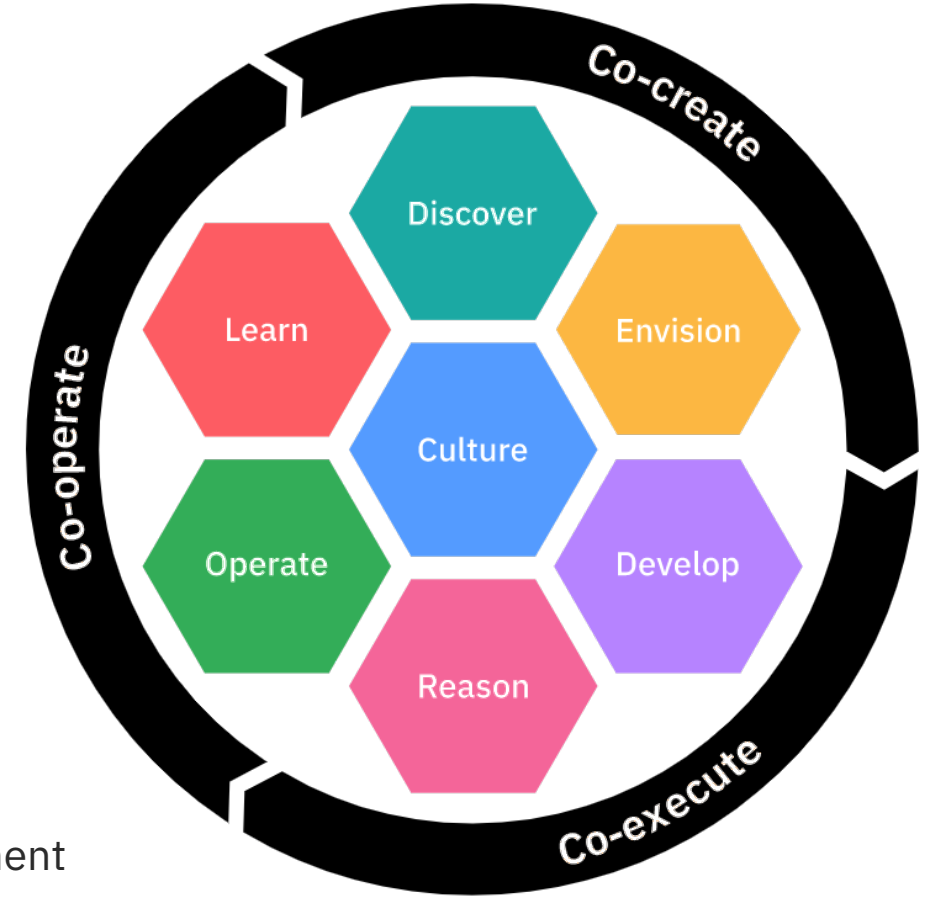
Don’t go it without Automation

Use the DevOps practices from the IBM Garage Method for Cloud

Continuous User Feedback and Optimization

Continuous Monitoring

Continuous Release and Deployment



Continuous Business Planning

Collaborative Development and Continuous Integration

Continuous Testing

CI/CD Pipelines for Microservices

Pipeline as Code

- Treat pipeline as another piece of code. Commit Jenkinsfile to repo
- Will now have multiple repos and each may have different build/deploy processes

Automation of CI/CD pipeline is necessary to ensure smooth deployments of microservices

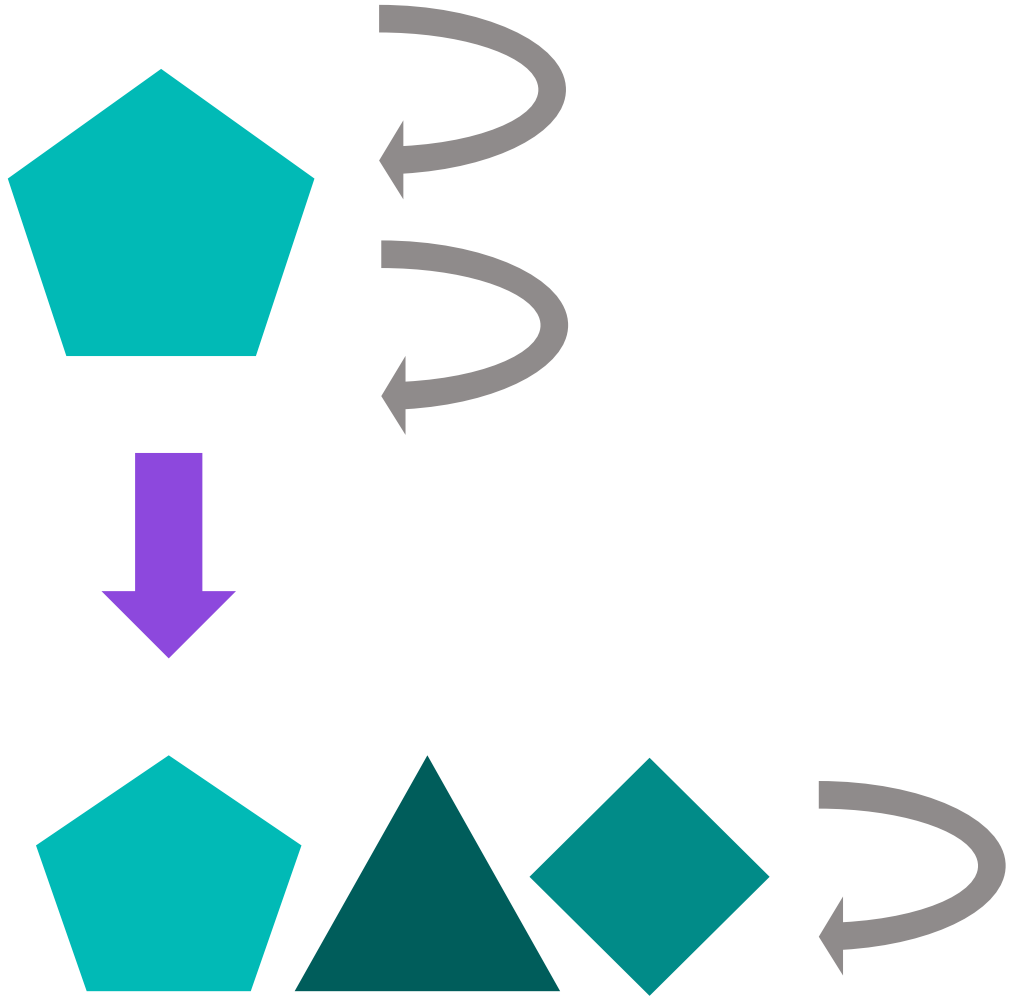
CI/CD Pipelines for Microservices

With microservices, each service can be tested separately.

- **Unit tests** can be done during the build stage with Maven along with a **static code analysis**

After publishing the microservice to the image registry, **integration testing** with the other components can begin.

- The newly published microservice is then tested against the latest stable versions of other components.



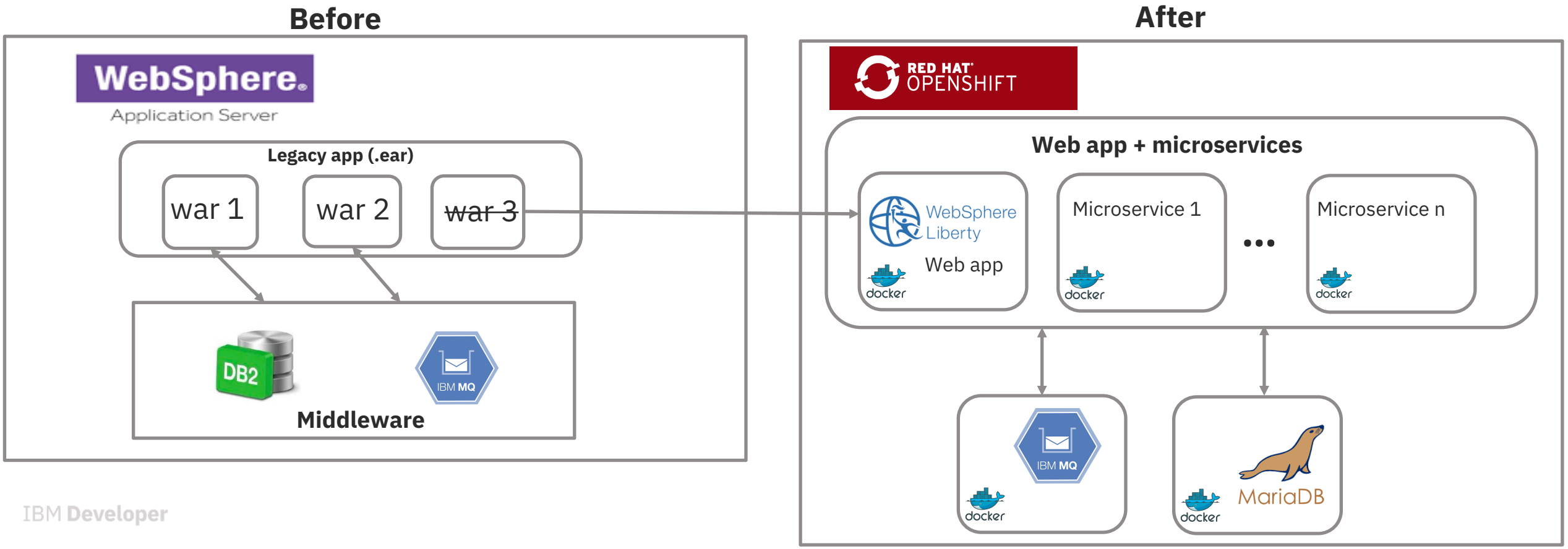
Twelve Factors – Cloud Native 101

https://12factor.net/

1. Codebase - One codebase tracked in revision control, many deploys
   - 1-1 relationship between app & code repo – use packages for shared code
2. Dependencies – Declared and isolated (no system wide dependencies)
3. Config - Store config in the environment (not in constants in the app)
4. Backing Services - Treat backing services as attached resources
   - Can be attached and reattached w/o affecting code, no differentiation between local and remote
5. Build, release, run - Strictly separate build and run stages
   - Release has unique id
6. Processes - Execute the app as one or more stateless processes
   - State shared via external services – no sticky sessions !
7. Port binding - Export services via port binding
   - In deployment, a routing layer handles routing requests from a public-facing hostname
8. Concurrency - Scale out via the process model
9. Disposability - Maximize robustness with fast startup and graceful shutdown
10. Dev/prod parity - Keep development, staging, and production as similar as possible
11. Logs - Treat logs as event streams
12. Admin processes - Run admin/management tasks as one-off processes

Refactor using the Strangler Pattern

- Component (war 3) of legacy app is rebuilt as a web app and a set of microservices
- New microservices and app use their own containerized data and messaging middleware
- Legacy app proxies all requests for war 3 to new implementation and handles everything else
- Over time more of the legacy app is rebuilt and eventually it will be no longer be needed



DEMO – Running Transformation Advisor on an existing Monolith

BADGE -

https://cognitiveclass.ai/badges/app-modernization-basics

Code Pattern -

https://developer.ibm.com/

Thank you!

developer.ibm.com