Domain-Driven Design (DDD) & Microservices: Patterns and Practices

Nevin Dong 董乃文
Principle Technical Evangelist
Microsoft Cooperation

Microsoft

New **patterns** and new **technologies**

Microservices

Autonomous
Bounded Context
Nomad & addressable services
API Gateway
Isolated
Decoupled
Events
Async. communication
Event Bus
Message Brokers
Service Discovery
Health Checks
Circuit Breakers
Transient Failures Handling
Resiliency
Commands
Retries with Exponential Backoff

Domain-Driven Design

Aggregates
CQRS simplified
Domain Events
Domain Entity
Mediator

Docker Containers

Linux Containers
Docker Image
Docker Host
Docker Registry
Windows Containers
Docker Hub
RabbitMQ
Hyper-V Containers
Azure Service Bus
Azure Container Registry
NServiceBus
MassTransit
Brighter
Polly

Orchestrators

Stateful Services
Azure Service Fabric
Actors
Azure Container Service
Kubernetes
Docker Swarm
Mesos DC/OS

Microservices Architecture

Microservice architecture benefits

Scale Independently

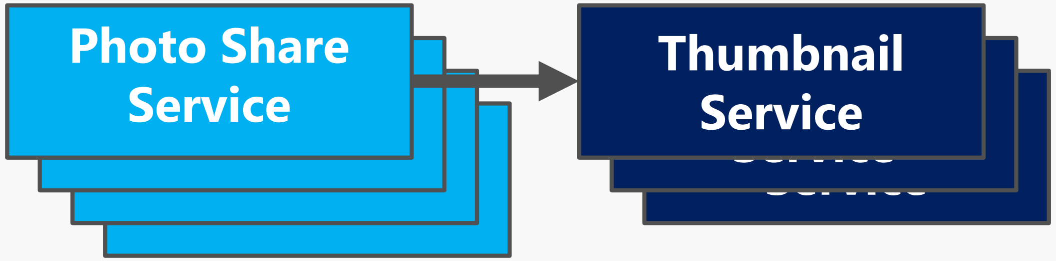


Photo Share Service

Thumbnail Service

Different Technology Stacks

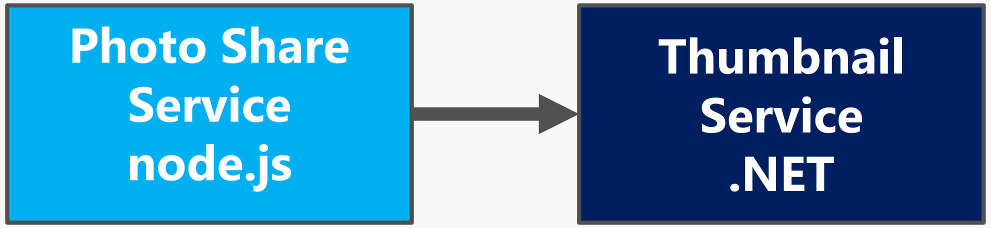


Photo Share Service node.js

Thumbnail Service .NET

Independent Deployments

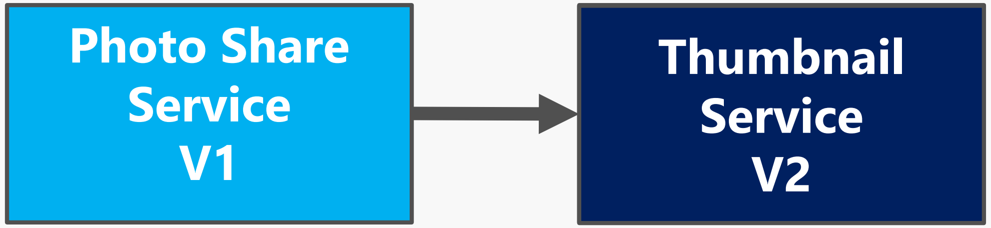
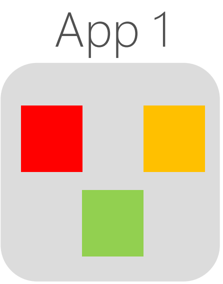


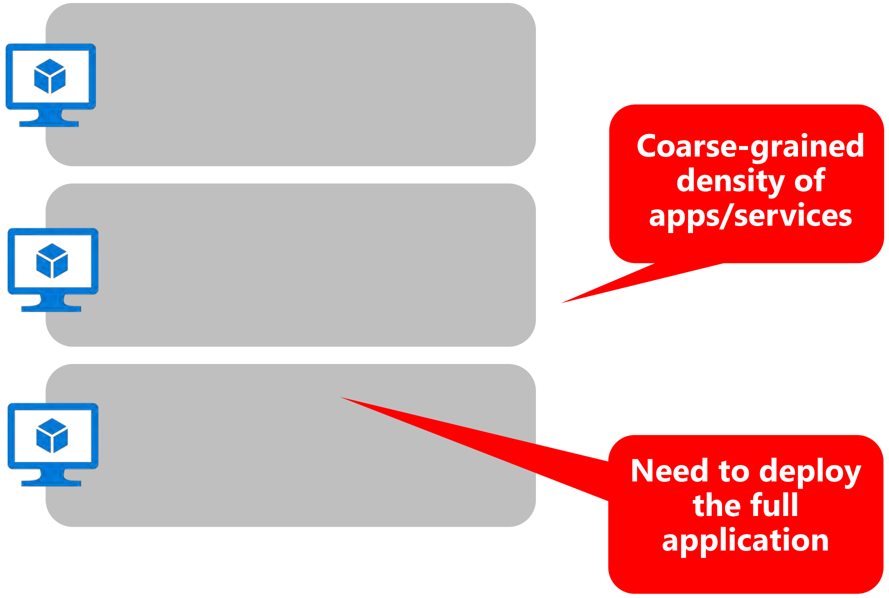
Photo Share Service V1

Thumbnail Service V2

Traditional application approach

- A traditional application has most of its functionality within a few processes that are componentized with layers and libraries.
- Scales by cloning the app on multiple servers/VMs



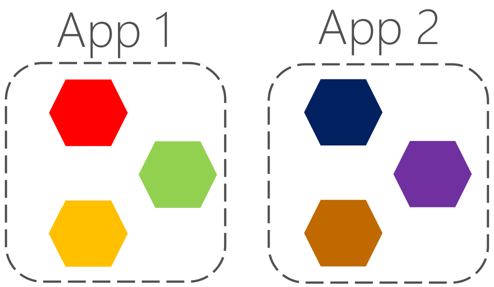


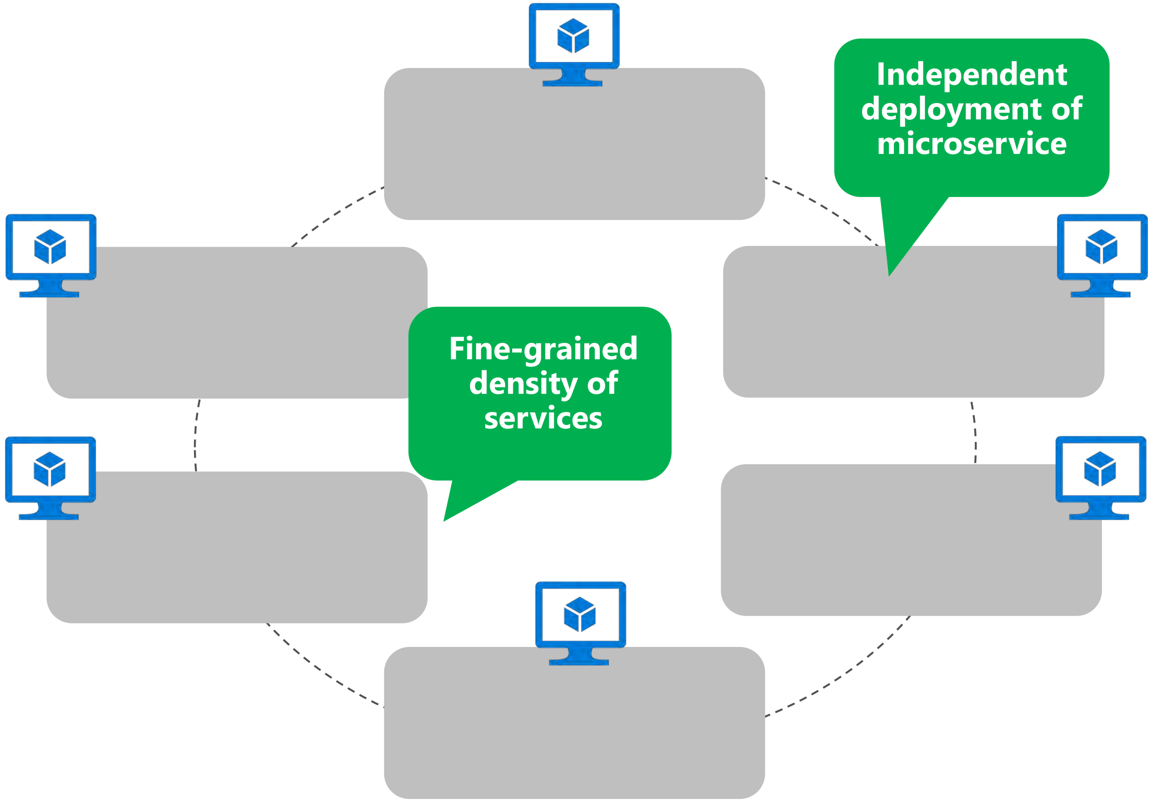
Coarse-grained density of apps/services

Need to deploy the full application

Microservices application approach

- A microservice application segregates functionality into separate smaller services.
- Scales out by **deploying each service independently** with multiple instances across servers/VMs



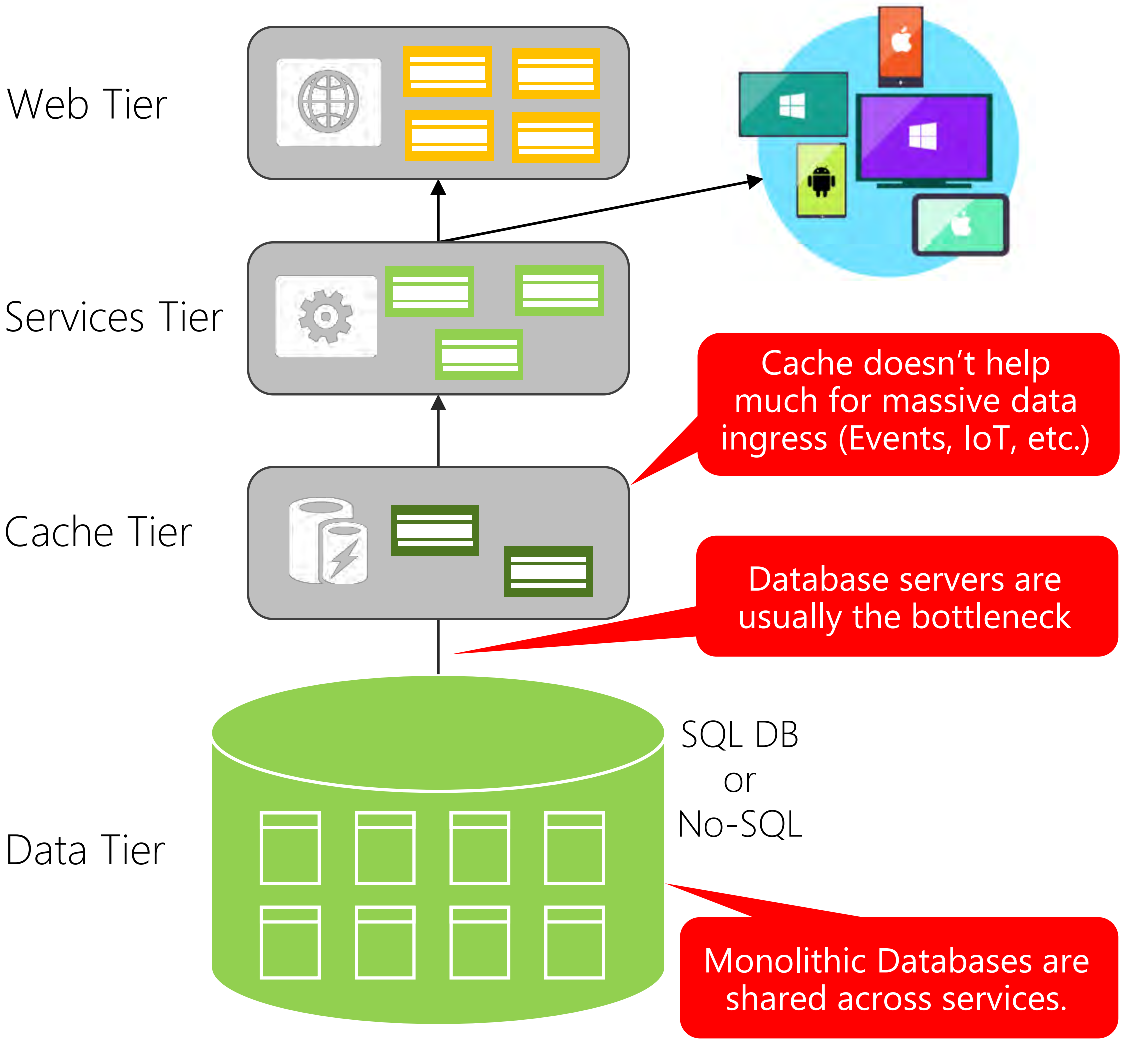


Independent deployment of microservice

Fine-grained density of services

Data in Traditional approach

- Single monolithic database
- Tiers of specific technologies



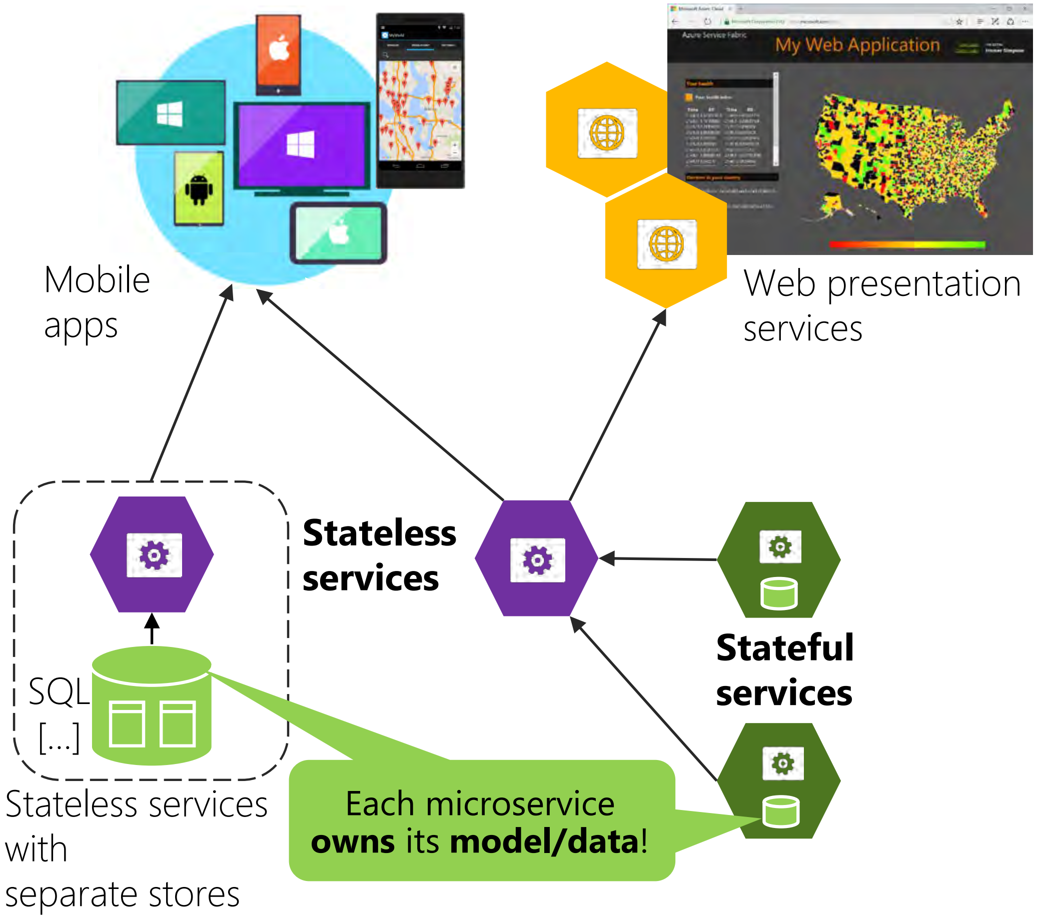
Cache doesn't help much for massive data ingress (Events, IoT, etc.)

Database servers are usually the bottleneck

Monolithic Databases are shared across services.

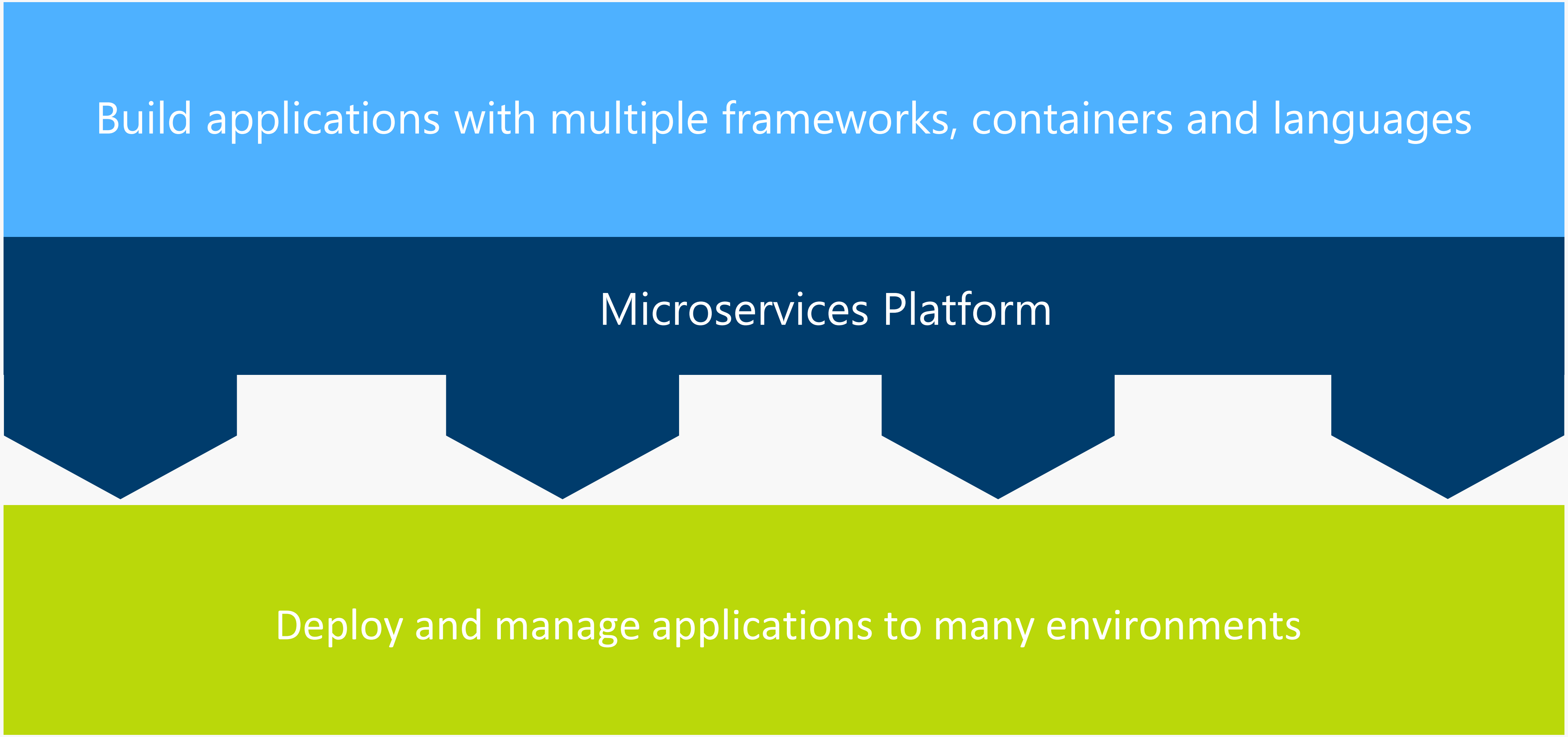
Data in Microservices approach

- Graph of interconnected microservices
- State typically scoped to the microservice
- Remote Storage for cold data



Each microservice **owns** its **model/data**!

Microservices platform



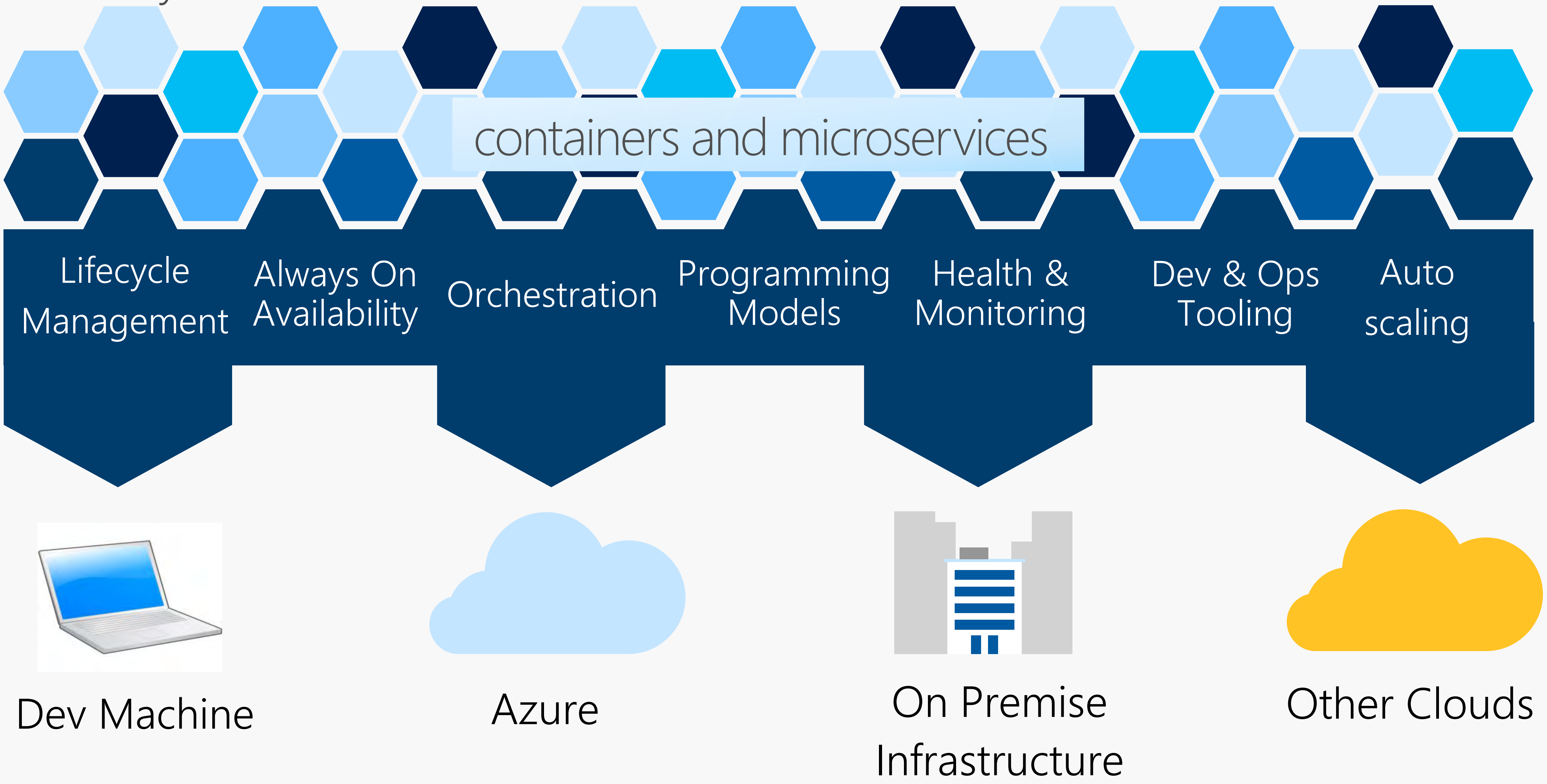
Build applications with multiple frameworks, containers and languages

Microservices Platform

Deploy and manage applications to many environments

Azure Service Fabric

Any OS, Any Cloud



containers and microservices

Lifecycle Management

Always On Availability

Orchestration

Programming Models

Health & Monitoring

Dev & Ops Tooling

Auto scaling

Dev Machine

Azure

On Premise Infrastructure

Other Clouds

Services Powered by Service Fabric

SQL Database
2.1 million DBs

Cosmos DB
Billions transactions/day

IoT Hub
Millions of messages

Event Hubs
60bn events/day

Microsoft runs its business on Service Fabric

Skype for Business
Skype

Cortana

Windows Intune
Intune

Microsoft Dynamics 365
Dynamics

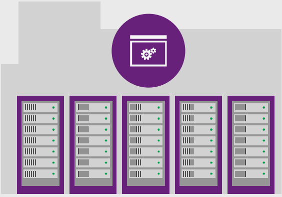
Power BI

Designed for mission critical tier 1 workloads

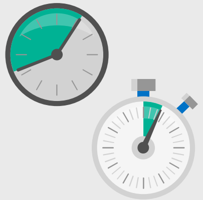
30% of Azure cores run Service Fabric

Service Fabric on Azure

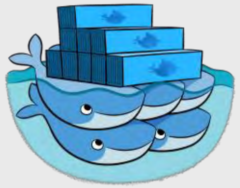
Microservices Platform



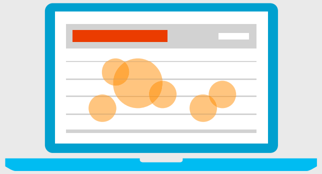
Highly scalable



24 X 7 High availability and failover



Windows and Linux container orchestration

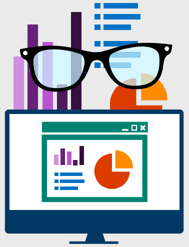


DevOps and Lifecycle management

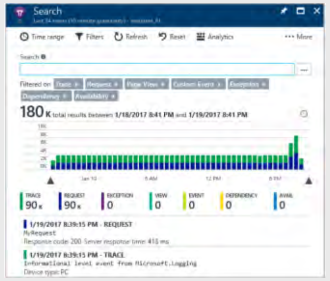
Managed Service

Built-in auto scale

Automated platform upgrades



Built-in health and diagnostics



180 K total results between 1/18/2017 8:41 PM and 1/19/2017 8:41 PM

TRACE 90 K | REQUEST 90 K | EXCEPTION 0 | VIEW 0 | EVENT 0 | DEPENDENCY 0 | AVAIL 0

1/19/2017 8:39:15 PM - REQUEST
MyRequest
Response code: 200 Server response time: 415 ms

1/19/2017 8:39:15 PM - TRACE
Informational level event from Microsoft.Logging
Device type: PC

Integrated with AppInsights and OMS

Productive Development

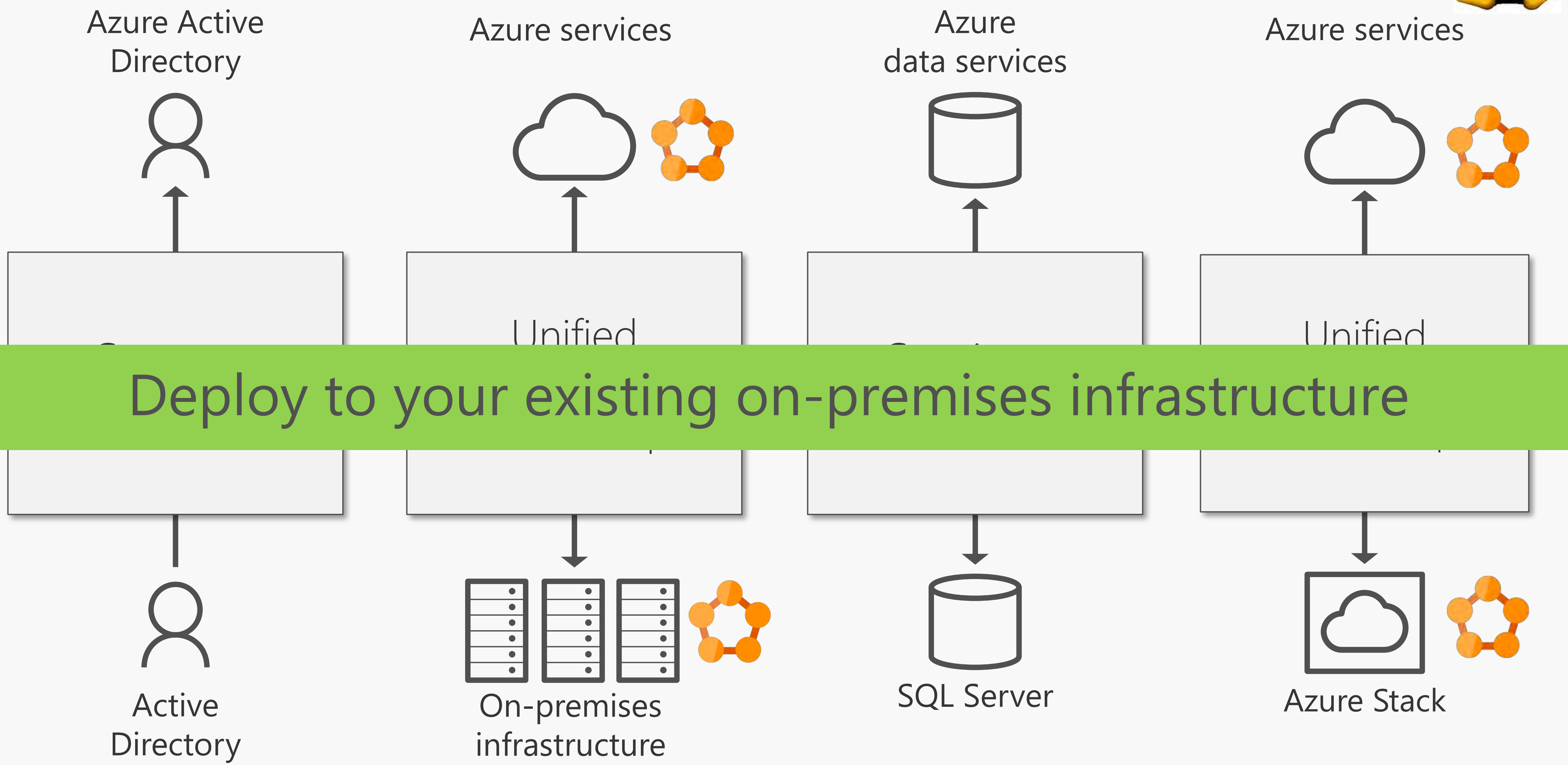
Simple Programming Models for. NET, Java

Stateless and Stateful microservices

Local development identical to cloud development

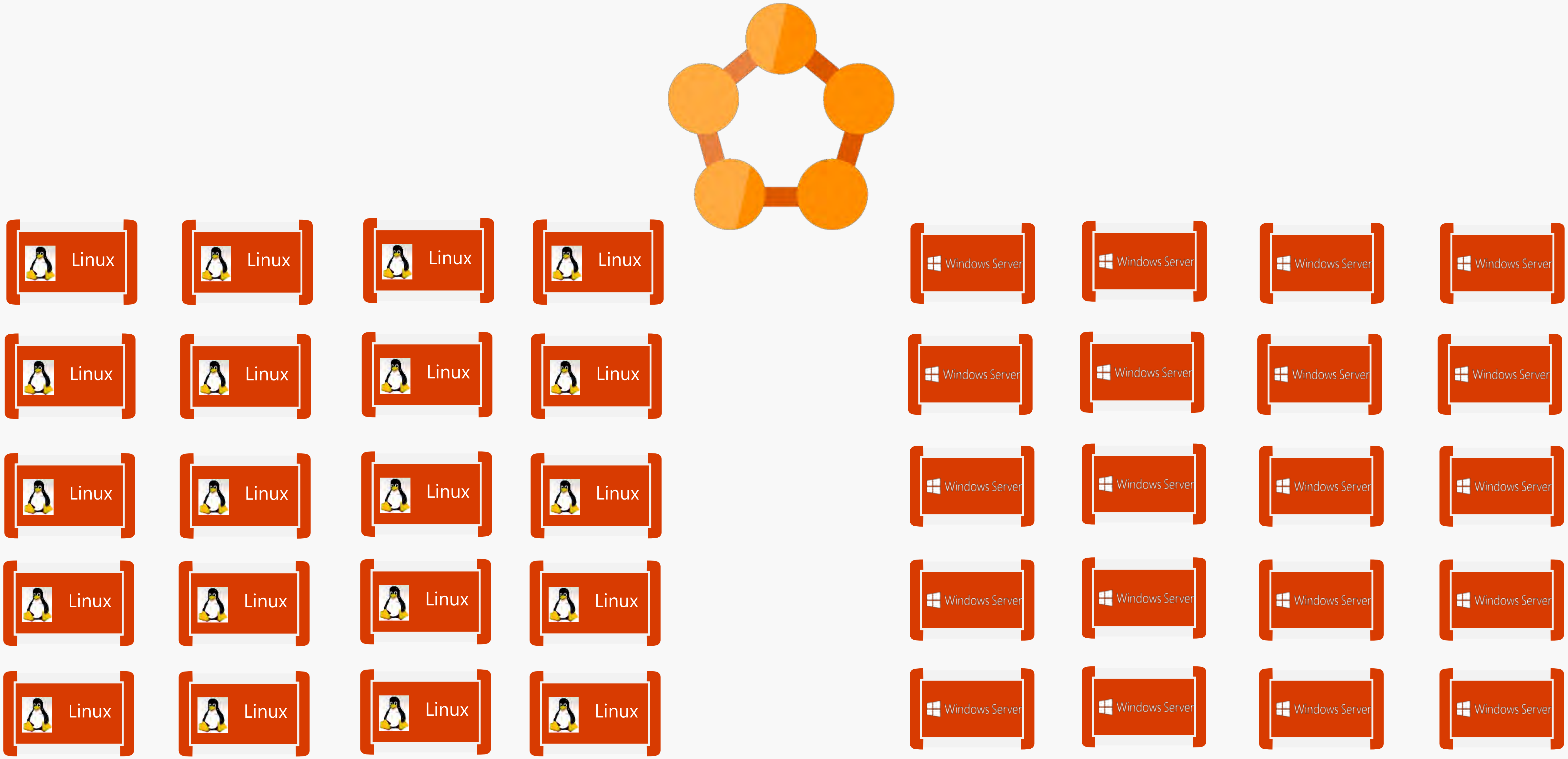
Tooling with Visual Studio, VSTS Eclipse & Jenkins

Service Fabric on premises



Deploy to your existing on-premises infrastructure

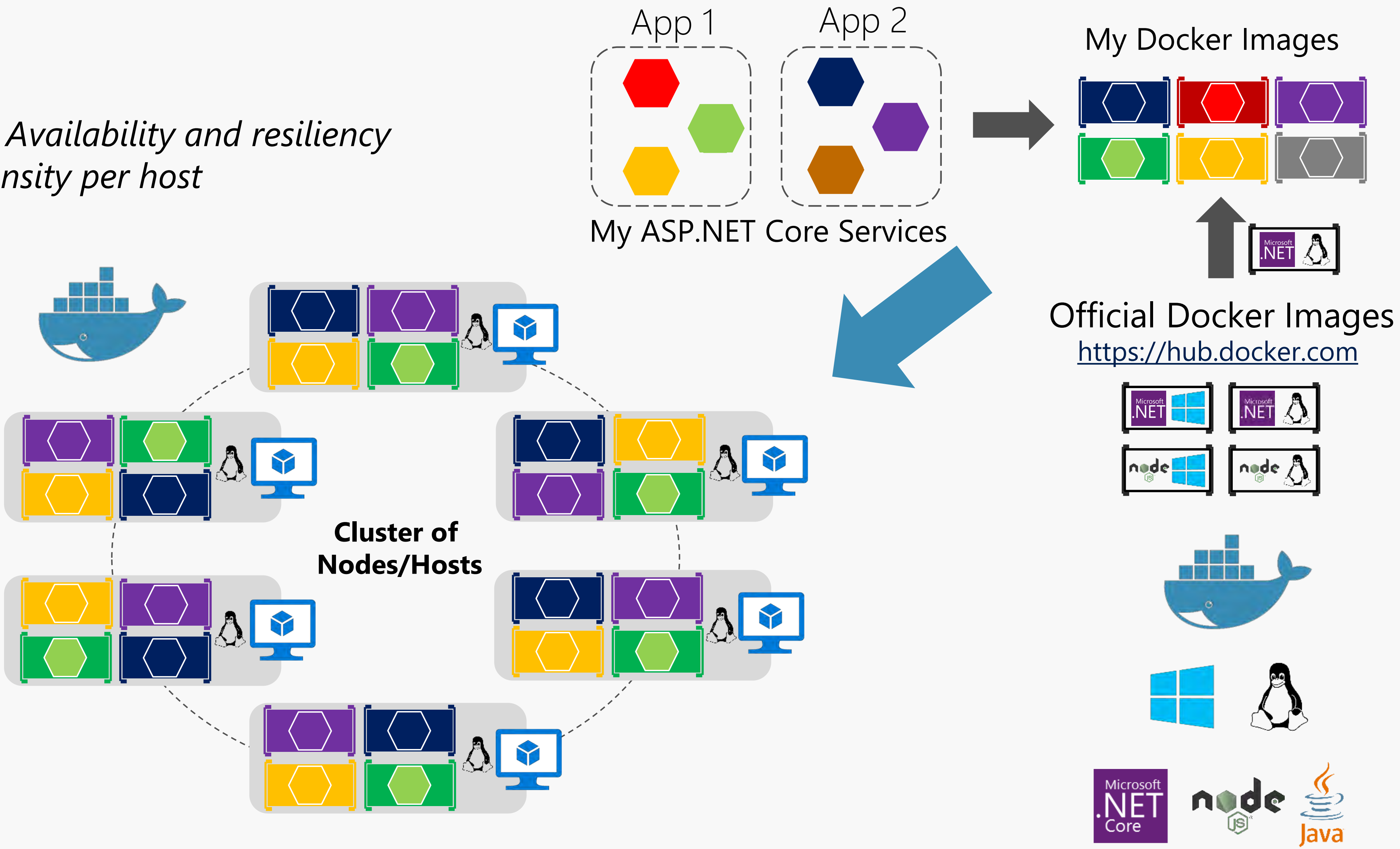
Service Fabric: Microsoft's Container Orchestrator

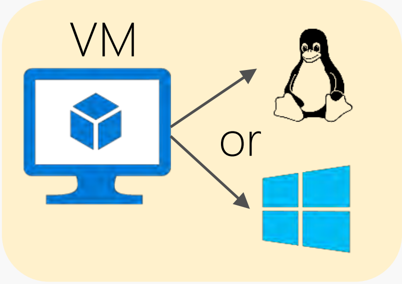


Orchestrator's Cluster managing microservices/containers

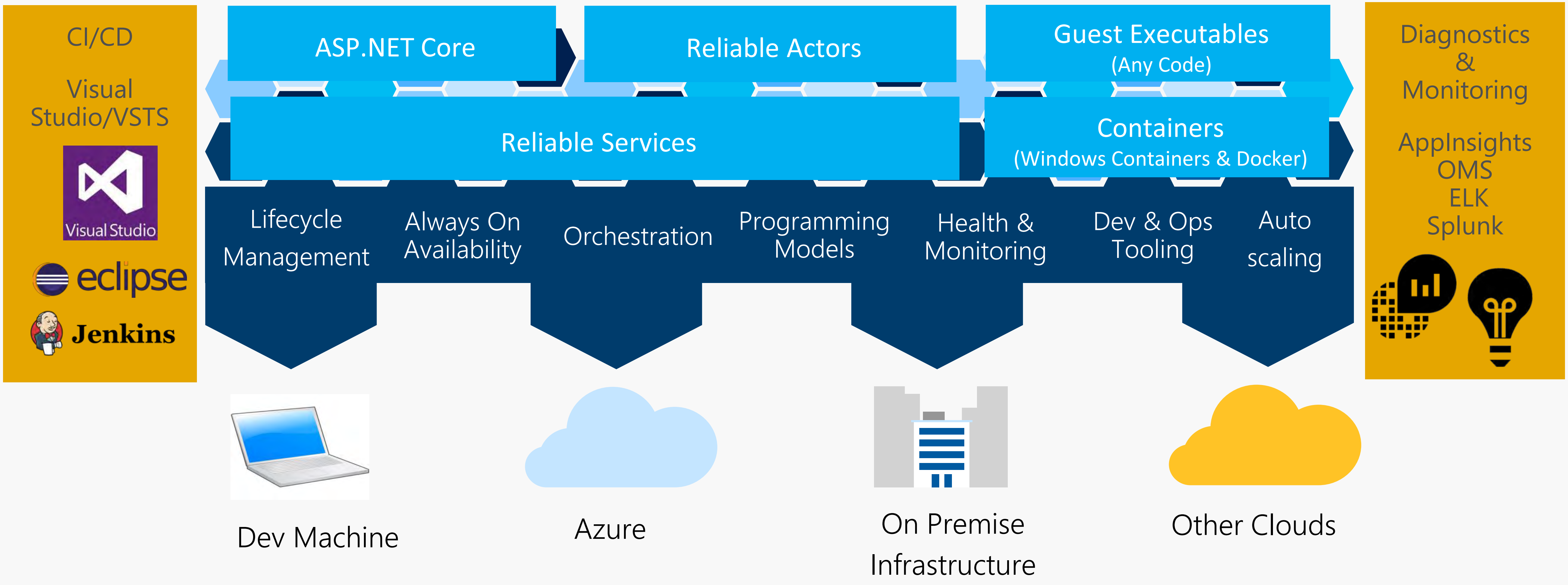
Clusters provide:

- *High scalability*
- *Automatic High Availability and resiliency*
- *High services density per host*





CI/CD, diagnostics and monitoring



CI/CD

Visual Studio/VSTS

Visual Studio

eclipse

Jenkins

ASP.NET Core

Reliable Actors

Guest Executables (Any Code)

Reliable Services

Containers (Windows Containers & Docker)

Lifecycle Management

Always On Availability

Orchestration

Programming Models

Health & Monitoring

Dev & Ops Tooling

Auto scaling

Diagnostics & Monitoring

AppInsights

OMS

ELK

Splunk

Dev Machine

Azure

On Premise Infrastructure

Other Clouds

Key patterns for microservices and Domain-Driven Design

Key Patterns for Microservices

1. Direct communication vs. API Gateway
2. Health checks
3. Resilient cloud applications:
   - Retries with exponential backoff plus Circuit breaker
4. Async. pub/subs communication (Event Bus)
5. Scale-out with Orchestrators

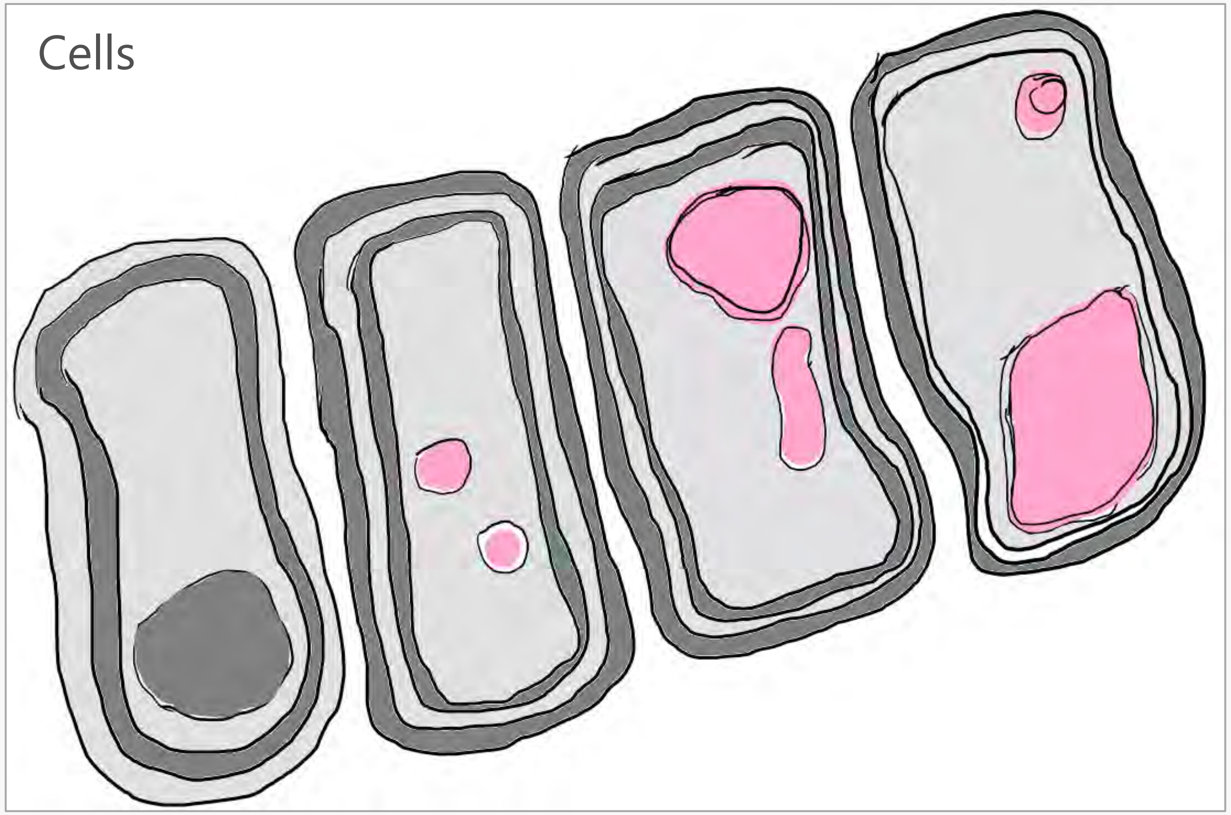
Domain-Driven Design (DDD) Patterns

Bounded Context == Business Microservice boundary

Use in your Core-Domain microservices, task oriented with lots of business rules & transactions

1. Simplified CQRS when using DDD in a microservice
2. Rich Domain Model vs. Anemic Domain Model
3. Domain Entity
4. Aggregates
5. Value Object
6. Domain Events (within a single microservice)

The Bounded Context pattern



Independent
Autonomous
Loosely coupled composition

"Cells can exist because their membranes define what is in and out and determine what can pass"
[Eric Evans]

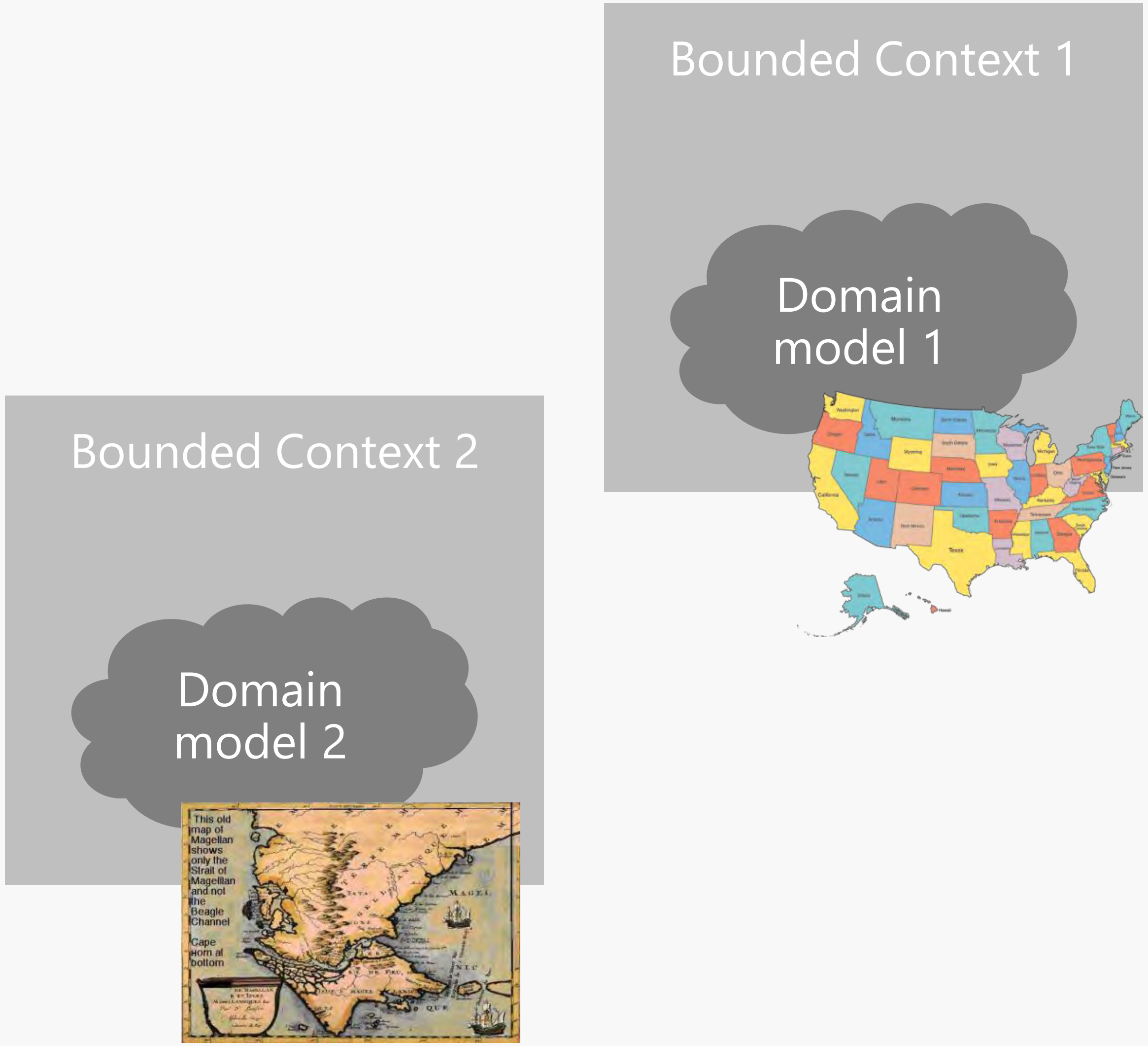
Bounded Context pattern in Domain-Driven Design

A domain model applies within a *Bounded Context*

In a typical enterprise system, there are multiple Bounded Contexts

Thus, multiple domain models

Not one big domain model across the entire system!



Bounded Context 1

Domain model 1

Bounded Context 2

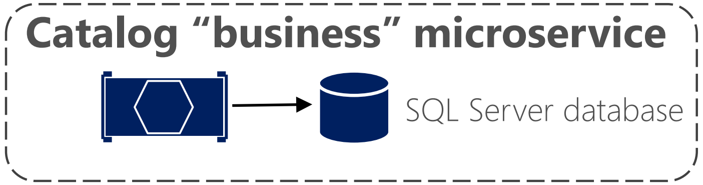
Domain model 2

This old map of Magellan shows only the Strait of Magellan and not the Beagle Channel
Cape Horn at bottom

Bounded Context == "Business Microservice" boundary

Business/Logical Microservices (Bounded Contexts)

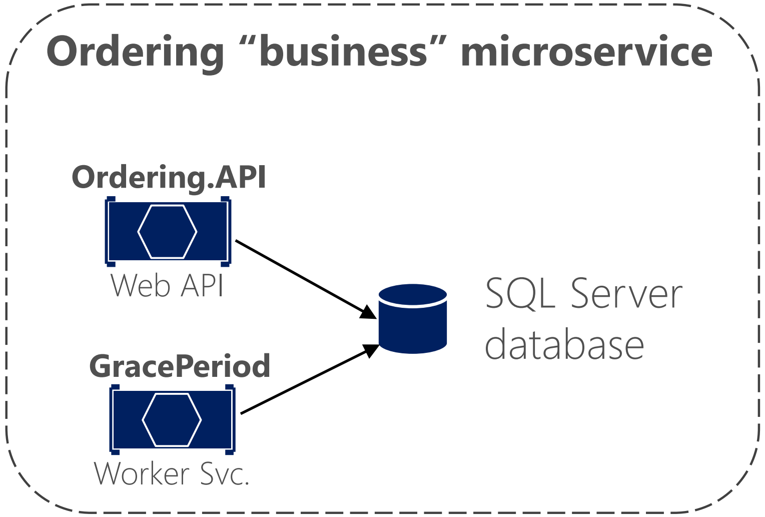
Example 1



Catalog "business" microservice

SQL Server database

Example 2



Ordering "business" microservice

Ordering.API

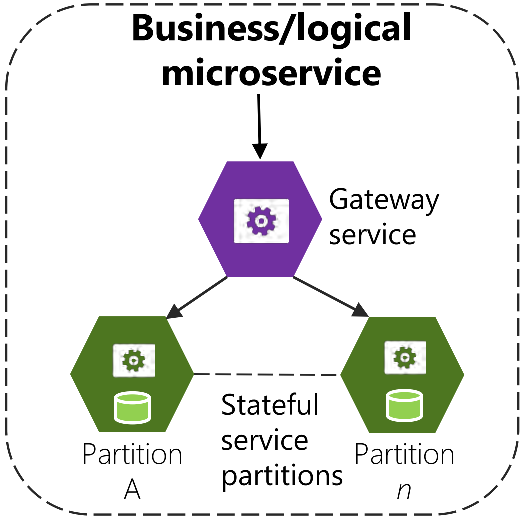
Web API

GracePeriod

Worker Svc.

SQL Server database

Example 3



Business/logical microservice

Gateway service

Partition A

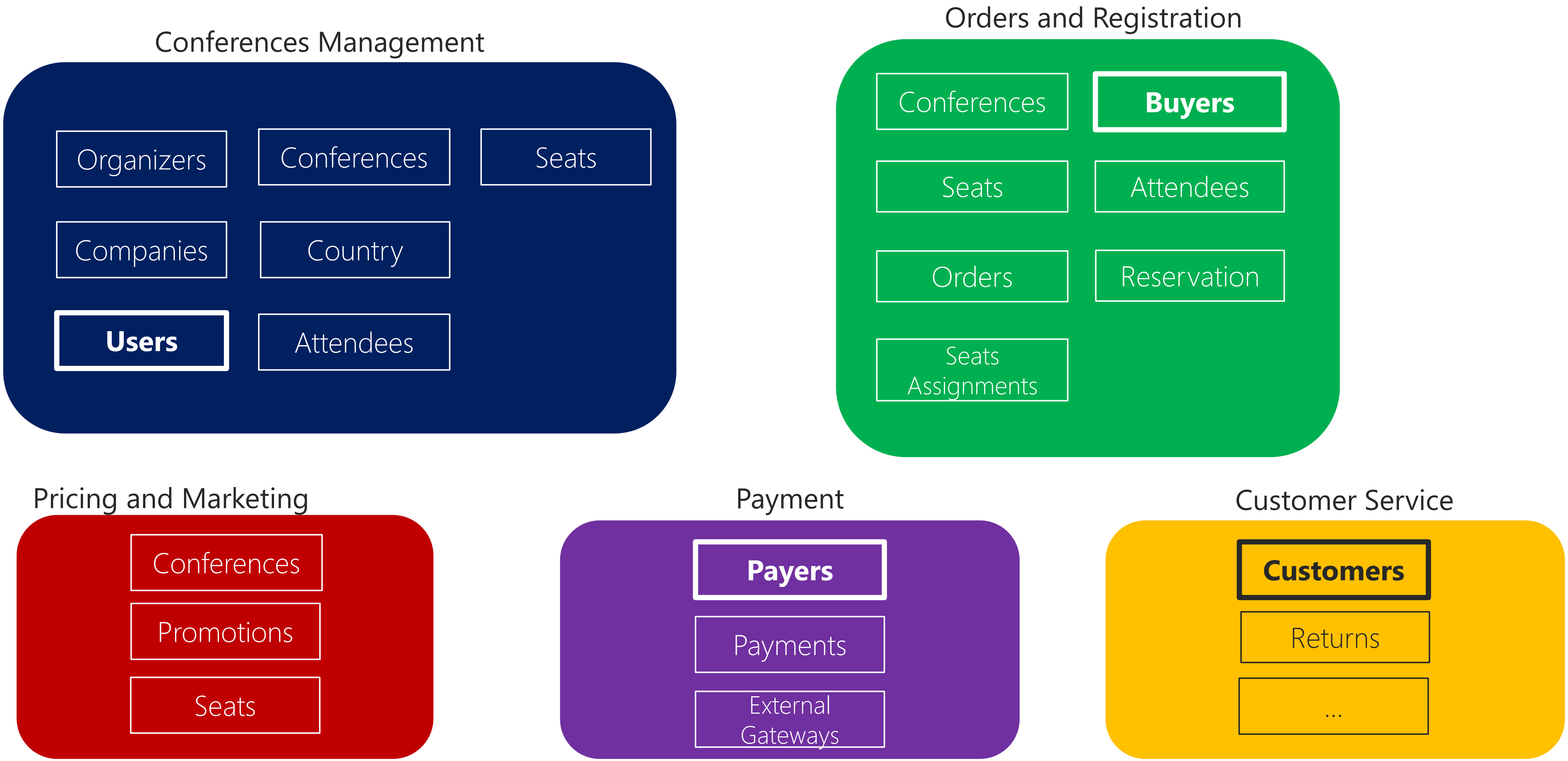
Stateful service partitions

Partition *n*

(Using Azure Service Fabric Stateful Reliable Services)

- The Logical Architecture can be different to the Physical/Deployment Architecture
- A Bounded Context can be implemented by 1 or more services (i.e. ASP.NET Web API)

Identifying a Domain Model per Microservice/BoundedContext



Conferences Management

Organizers

Conferences

Seats

Companies

Country

Users

Attendees

Orders and Registration

Conferences

Buyers

Seats

Attendees

Orders

Reservation

Seats Assignments

Pricing and Marketing

Conferences

Promotions

Seats

Payment

Payers

Payments

External Gateways

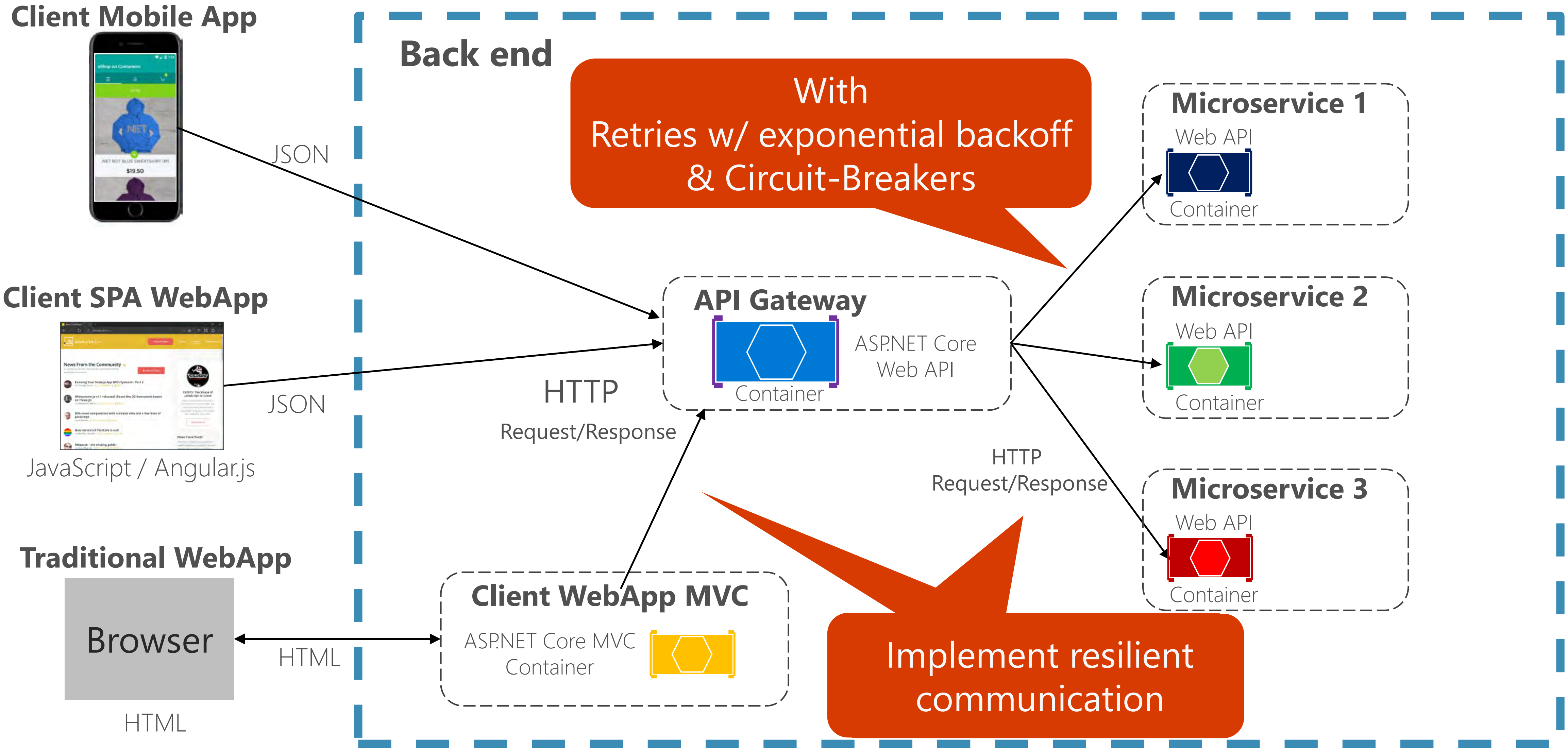
Customer Service

Customers

Returns

...

Building **resilient** cloud applications



Client Mobile App

JSON

Client SPA WebApp

JSON

JavaScript / Angular.js

Traditional WebApp

Browser

HTML

HTML

Back end

With
Retries w/ exponential backoff
& Circuit-Breakers

API Gateway

Container

ASP.NET Core
Web API

HTTP
Request/Response

Client WebApp MVC

ASP.NET Core MVC
Container

HTTP
Request/Response

Microservice 1

Web API

Container

Microservice 2

Web API

Container

Microservice 3

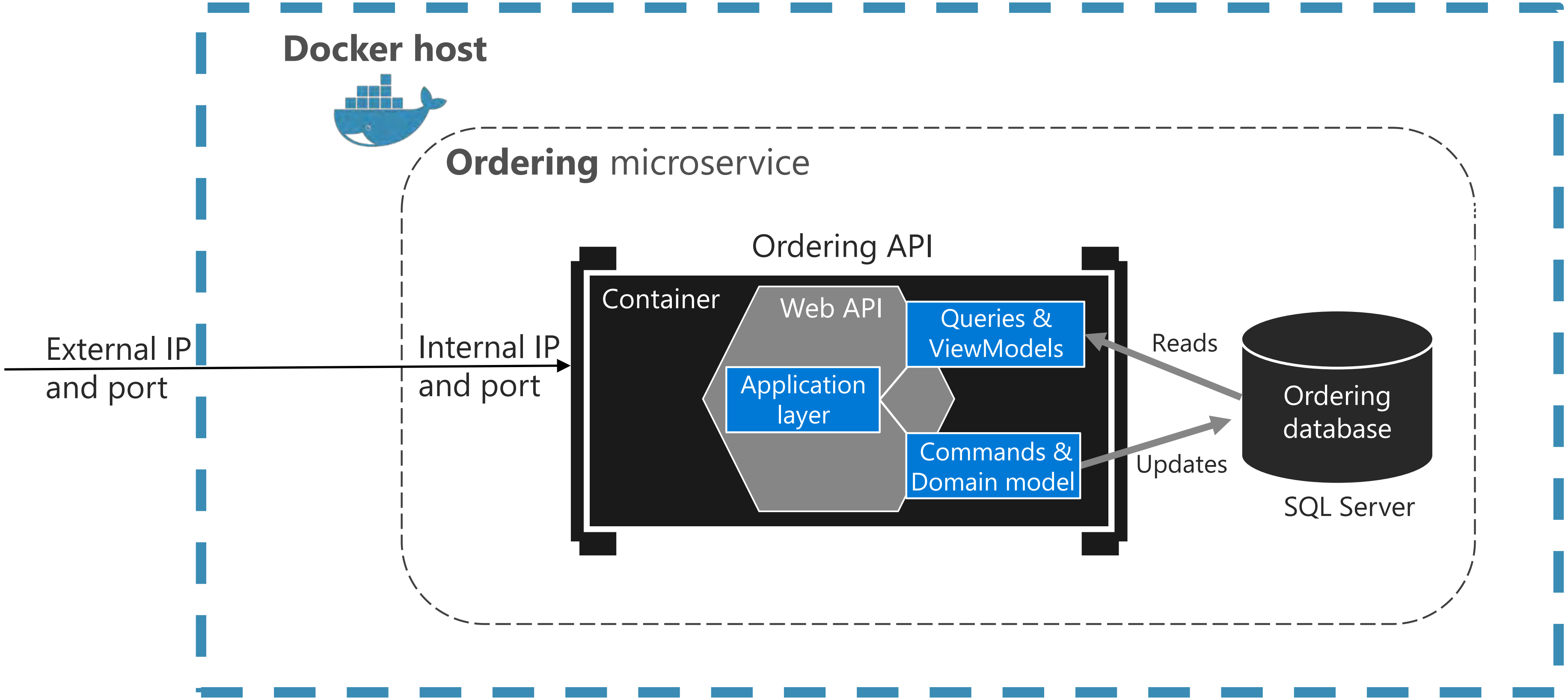
Web API

Container

Implement resilient
communication

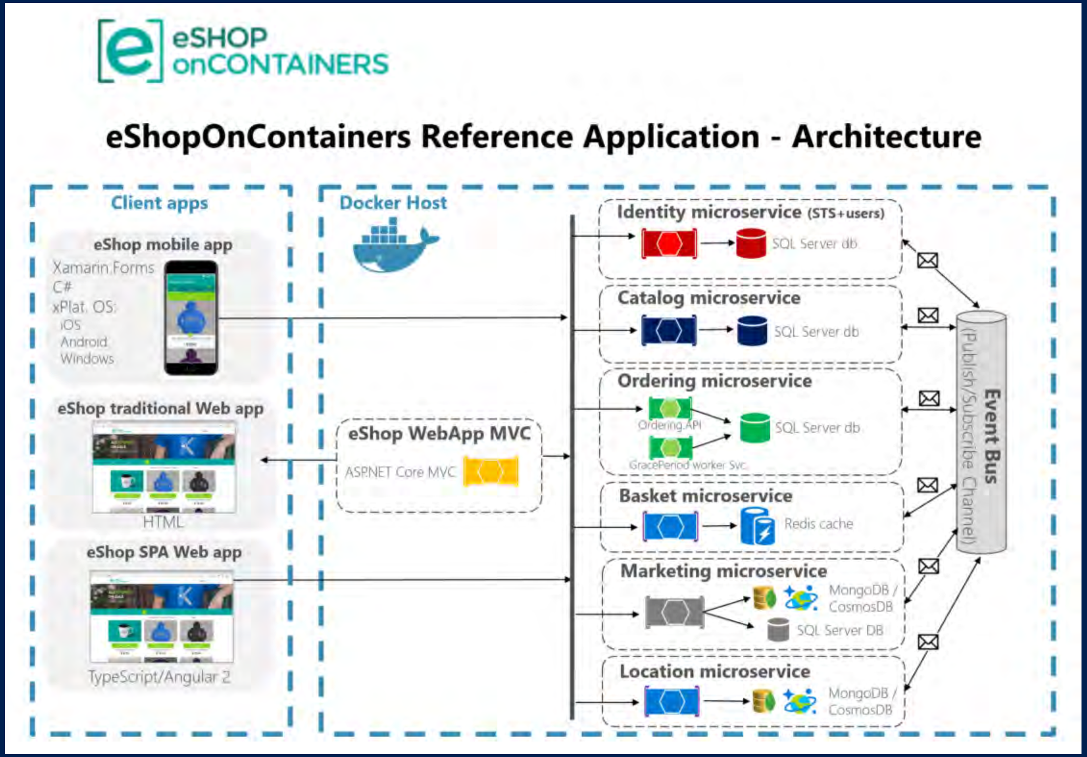
Simplified CQRS and DDD Microservice

High-Level Design

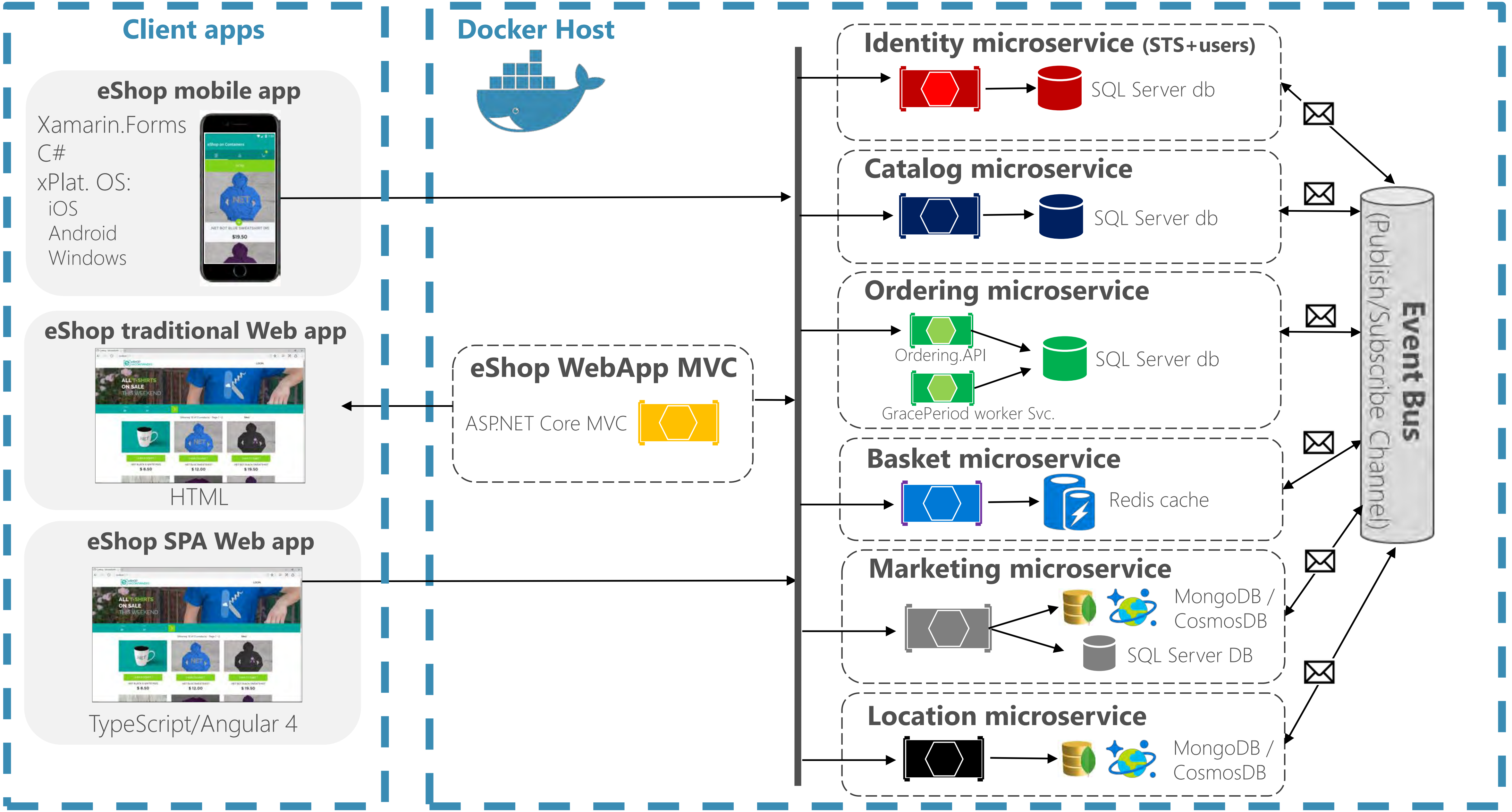


Practices: eShopOnContainer

eShopOnContainers Microservices and Docker Containers
End-to-end solution



eShopOnContainers Reference Application - Architecture



Client apps

eShop mobile app

Xamarin.Forms
C#
xPlat. OS:
iOS
Android
Windows

eShop traditional Web app

HTML

eShop SPA Web app

TypeScript/Angular 4

Docker Host

eShop WebApp MVC

ASP.NET Core MVC

Identity microservice (STS+users)

SQL Server db

Catalog microservice

SQL Server db

Ordering microservice

Ordering.API

GracePeriod worker Svc.

SQL Server db

Basket microservice

Redis cache

Marketing microservice

MongoDB / CosmosDB

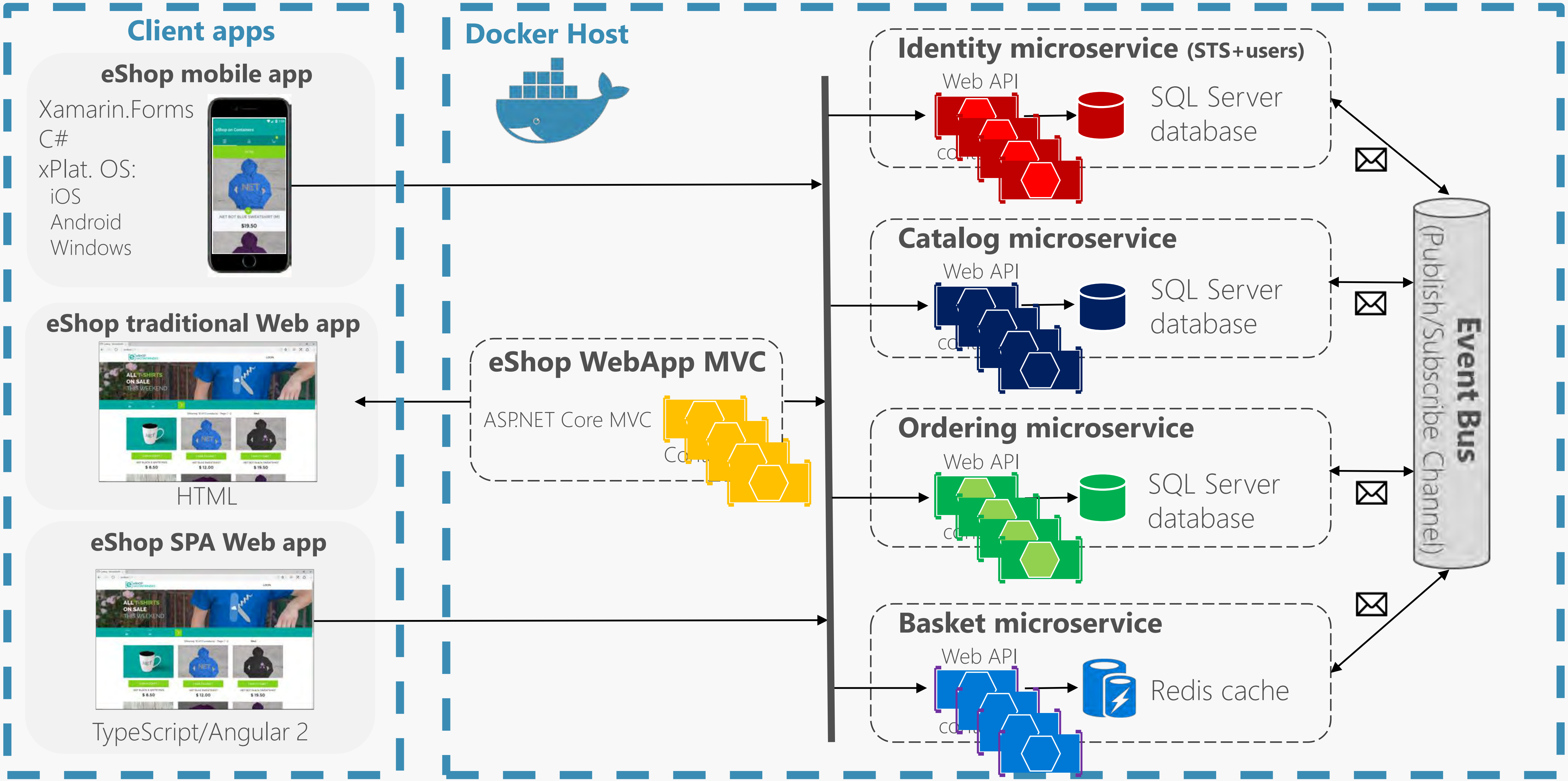
SQL Server DB

Location microservice

MongoDB / CosmosDB

Event Bus (Publish/Subscribe Channel)

Scaling out eShopOncontainers



Client apps

eShop mobile app

Xamarin.Forms
C#
xPlat. OS:
iOS
Android
Windows

eShop traditional Web app

HTML

eShop SPA Web app

TypeScript/Angular 2

Docker Host

eShop WebApp MVC

ASP.NET Core MVC

Identity microservice (STS+users)

Web API

SQL Server database

Catalog microservice

Web API

SQL Server database

Ordering microservice

Web API

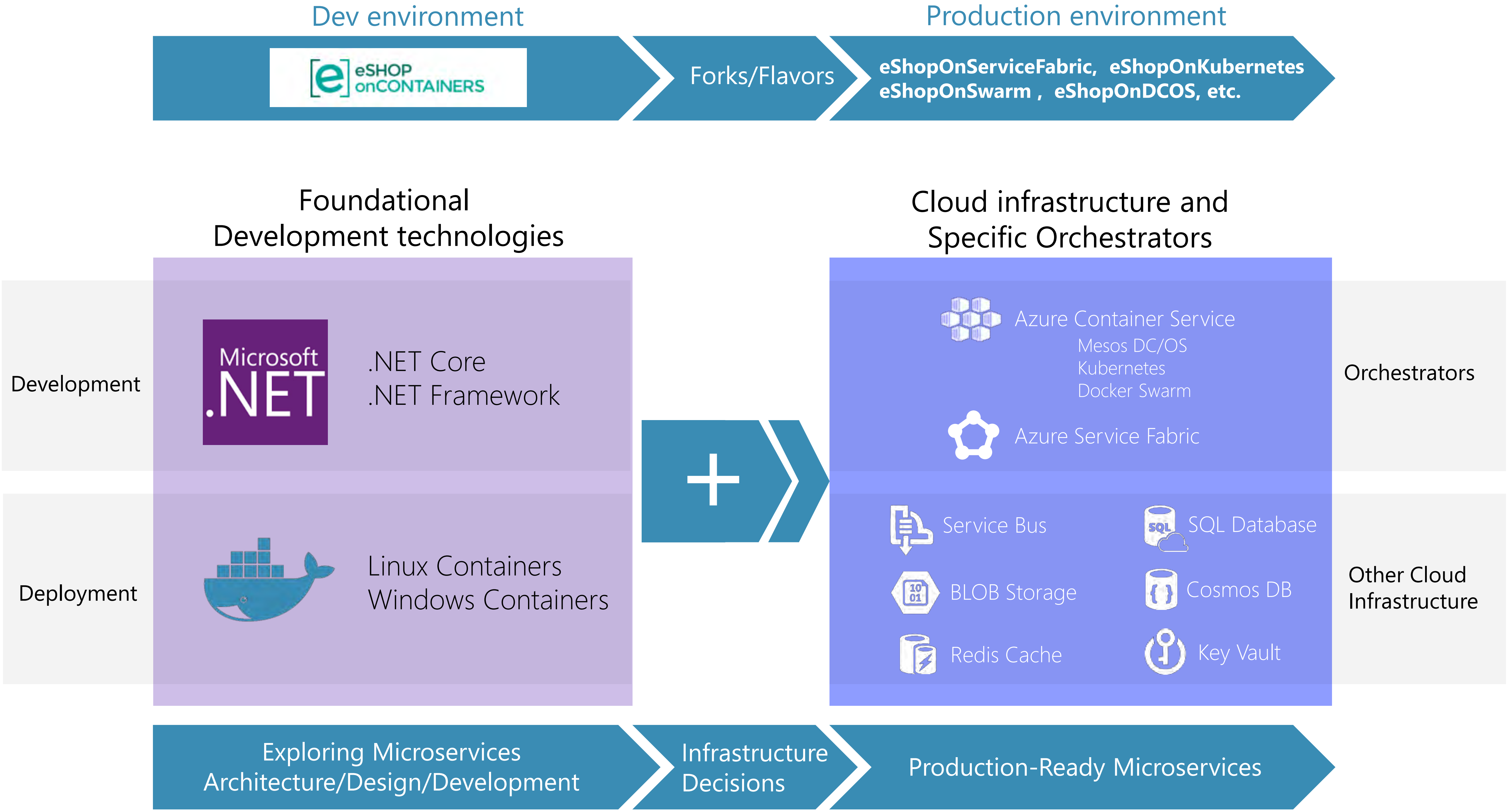
SQL Server database

Basket microservice

Web API

Redis cache

Event Bus
(Publish/Subscribe Channel)



Dev environment

eSHOP onCONTAINERS

Forks/Flavors

Production environment

eShopOnServiceFabric, eShopOnKubernetes
eShopOnSwarm , eShopOnDCOS, etc.

Foundational
Development technologies

Development

Microsoft .NET

.NET Core
.NET Framework

Deployment

Linux Containers
Windows Containers

+

Cloud infrastructure and
Specific Orchestrators

Azure Container Service
Mesos DC/OS
Kubernetes
Docker Swarm

Azure Service Fabric

Orchestrators

Service Bus

SQL Database

BLOB Storage

Cosmos DB

Redis Cache

Key Vault

Other Cloud
Infrastructure

Exploring Microservices
Architecture/Design/Development

Infrastructure
Decisions

Production-Ready Microservices

Domain-Driven Design (DDD) & Microservices: Patterns and Practices

THANK YOU

Nevin Dong 董乃文
Principle Technical Evangelist
Microsoft Cooperation