Developing Microservices Architecture using OSS Technologies

Session learning objectives

- Understand the importance of Microservices Architecture
- Understand a real-life case study of an Online Auction System
- Learn patterns and practices used in the application
- Use Microsoft Azure for cloud-native
- Why CI/CD is important
- Monitoring

Speaker Profile

Ovais Mehboob Ahmed Khan

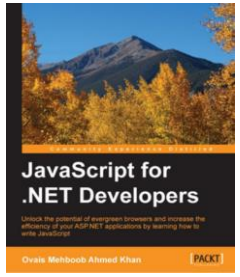
Senior Premier Field Engineer – Development

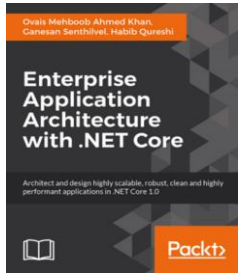
Microsoft Corporation

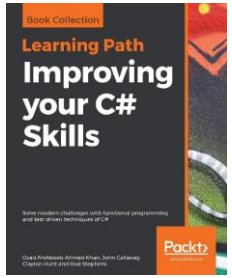
OvaisMehboob.com

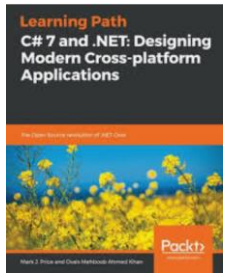
@ovaismehboob

linkedin.com/in/ovaismehboob/



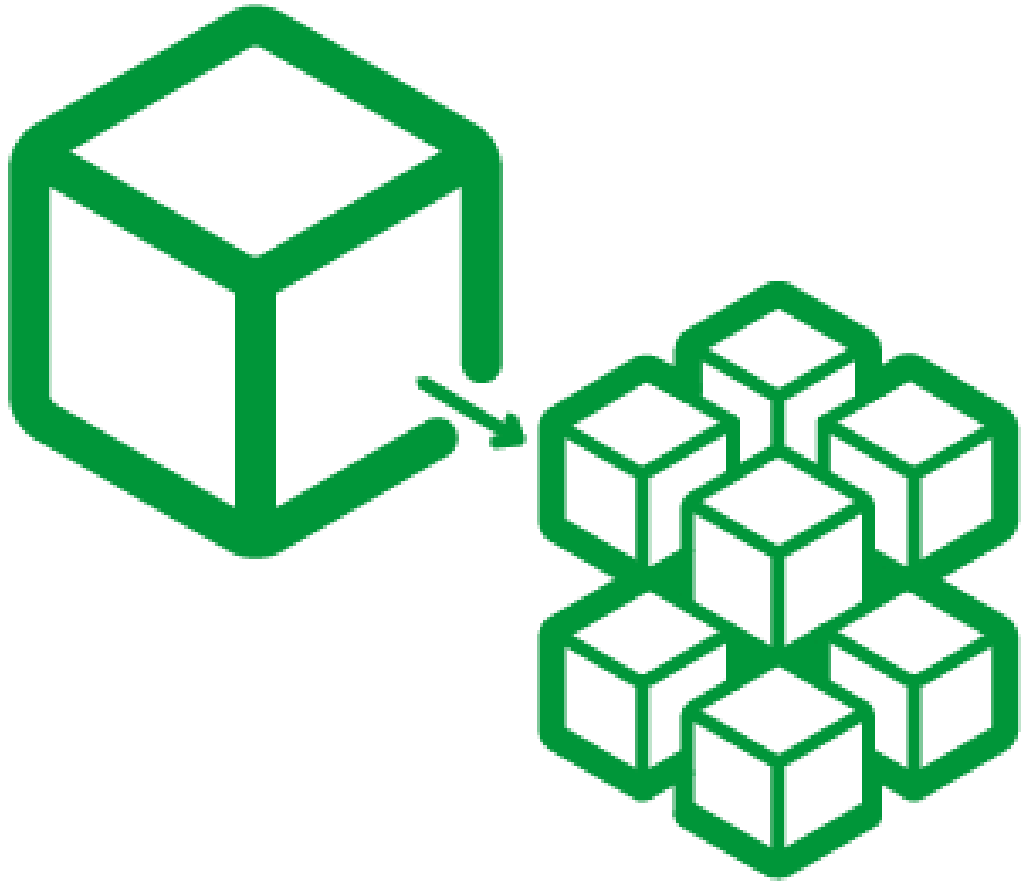






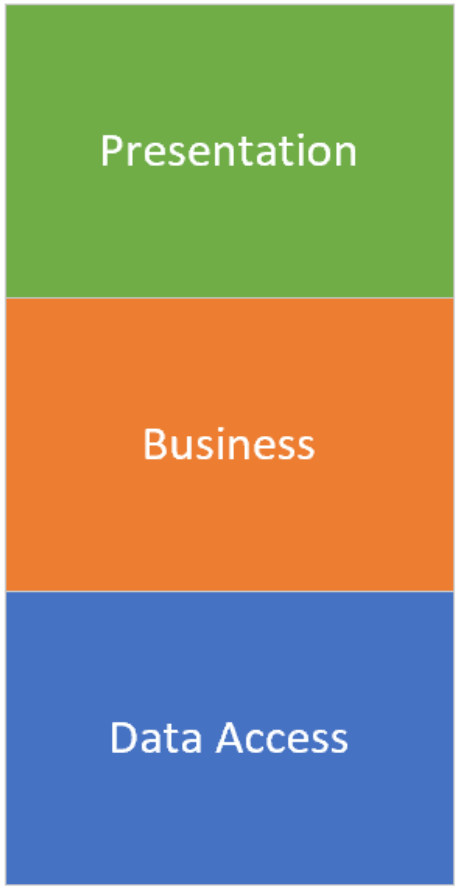
Microservices

- Independent, loosely coupled services modelled towards a business domain collaborate to make an application

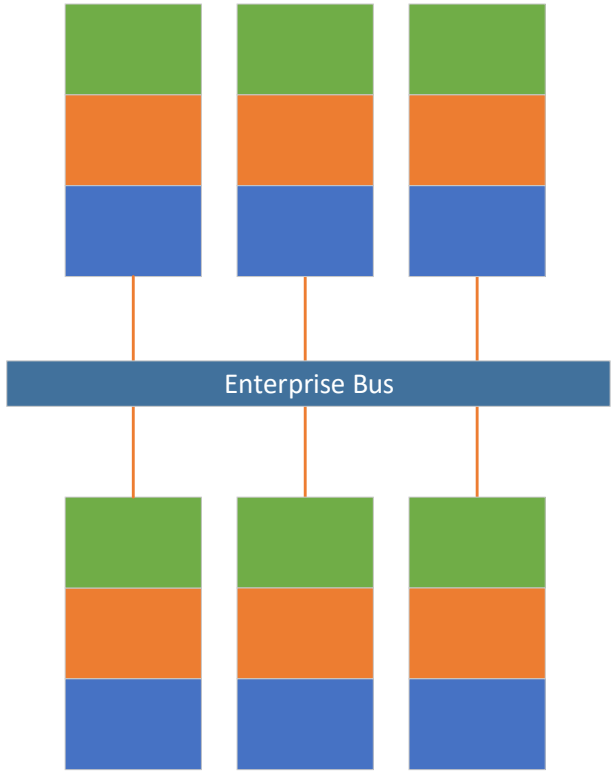


Evolution of Software Architecture

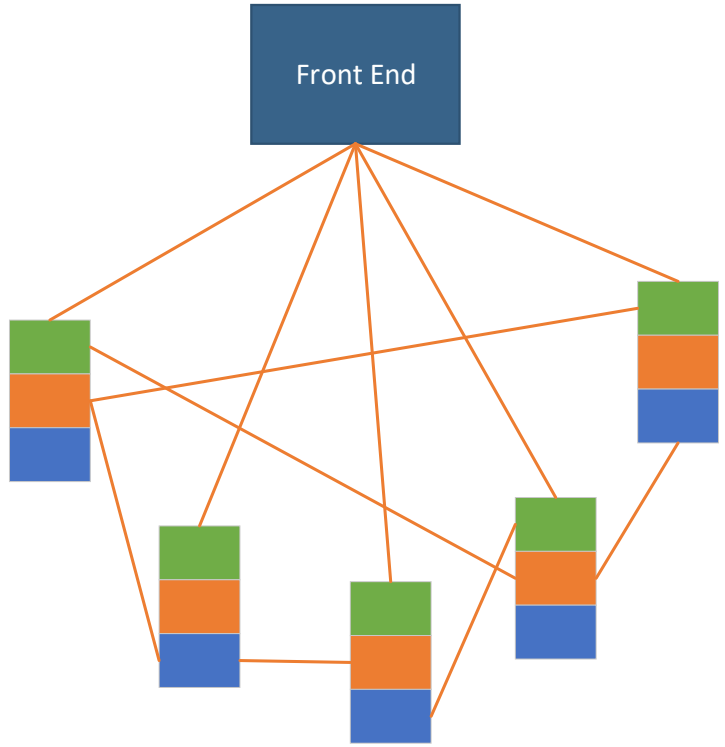
Monolith Architecture



Service Oriented Architecture



Microservices Architecture



Comparison with Traditional Application

Monolith App

VMs

Coarse-grained density of apps/services

Need to deploy the full application

Microservices App

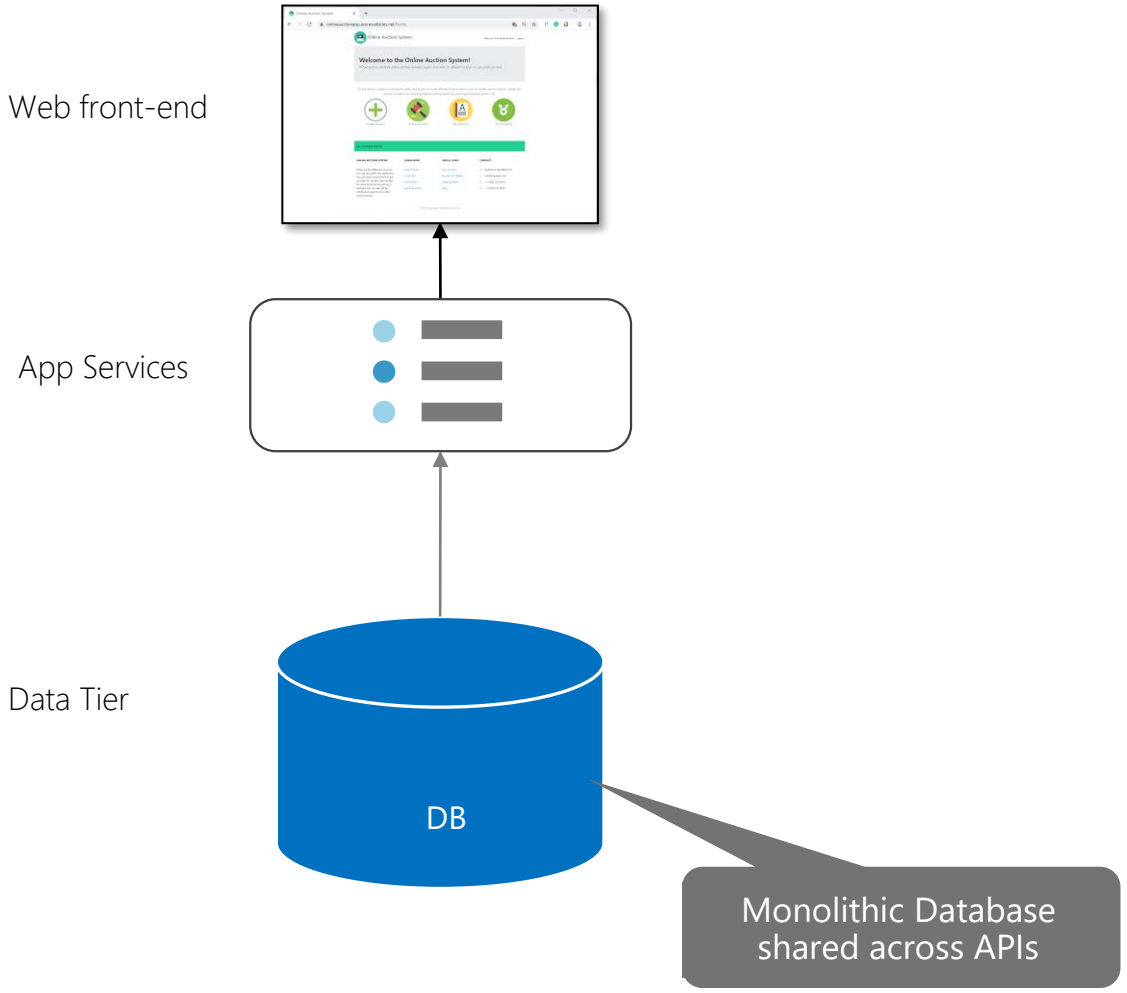
Deploy each service independently

VMs

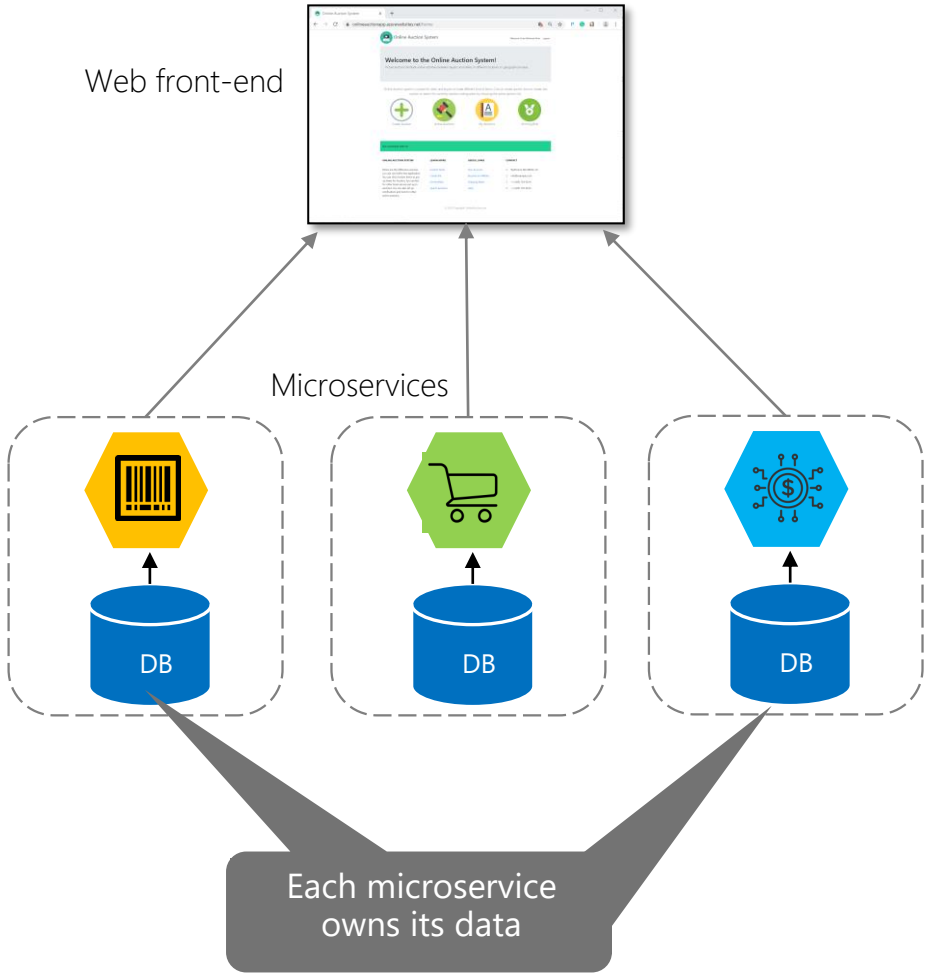
Fine-grained density of services

Data in Traditional vs Microservices Approach

- Single monolithic database
- Tiers of specific technologies

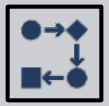


- Data/model is typically scoped to the microservice



Benefits of Microservices

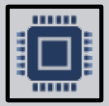
Business functionality encapsulated into small targeted services



Each service can evolve and deploy independently

Small team of developers can develop one service

Scale independently rather scaling out the whole application



Language agnostics and use mixed programming platforms

Challenges of Microservices

Single team of developers need to build expertise on new technologies

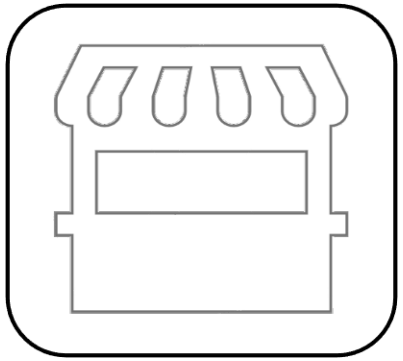
Manual deployment is cumbersome process

Need to use various patterns and communication channels for interaction between services

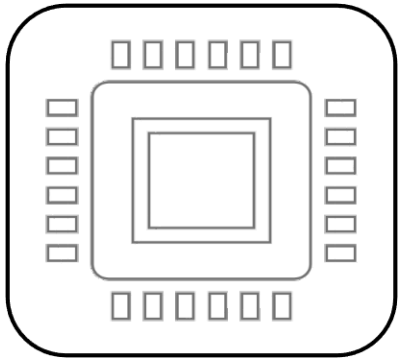
Distributed transaction is not supported, need to use certain patterns to implement data consistencies across services

Real-life Case Study: Online Auction System

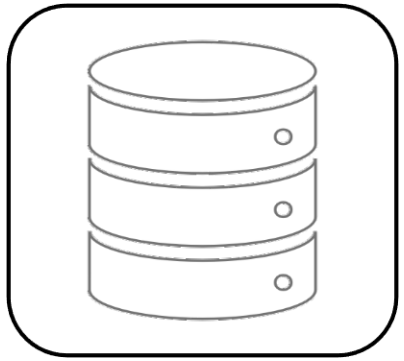
Online Auction System



Place for seller and buyer to trade different items

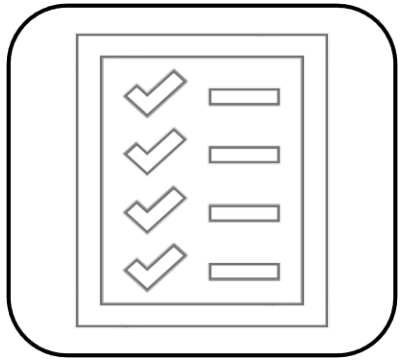


Set of microservices built on various OSS technologies



Web based front-end built on Angular

Used various Azure services to make it cloud native



Used Azure DevOps for complete application lifecycle management

Features

User can Register/Login to the system

Create new Auction items

Make bids to active Auction items

Each auction has a valid time period

Auto-auction awarding based on highest bid

User can make payment for winning bids

Demo

Online Auction System

Microservices Decomposition

Decomposition principles

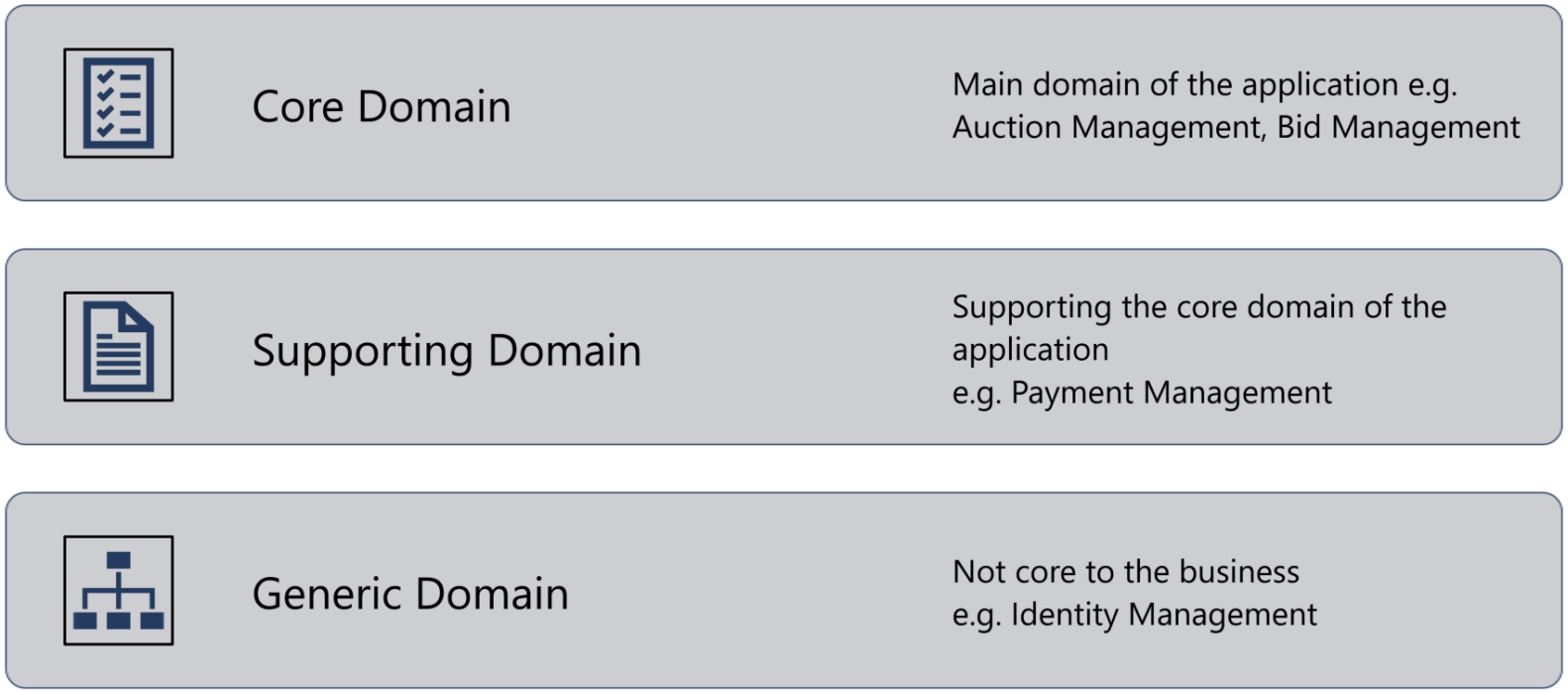
Services should be cohesive

Services must be loosely coupled

Service should be autonomous

Decompose by Subdomain

- Define services corresponding to the Domain Driven Design (DDD) principles
- DDD is a software development approach to map the technical implementation to the business domain

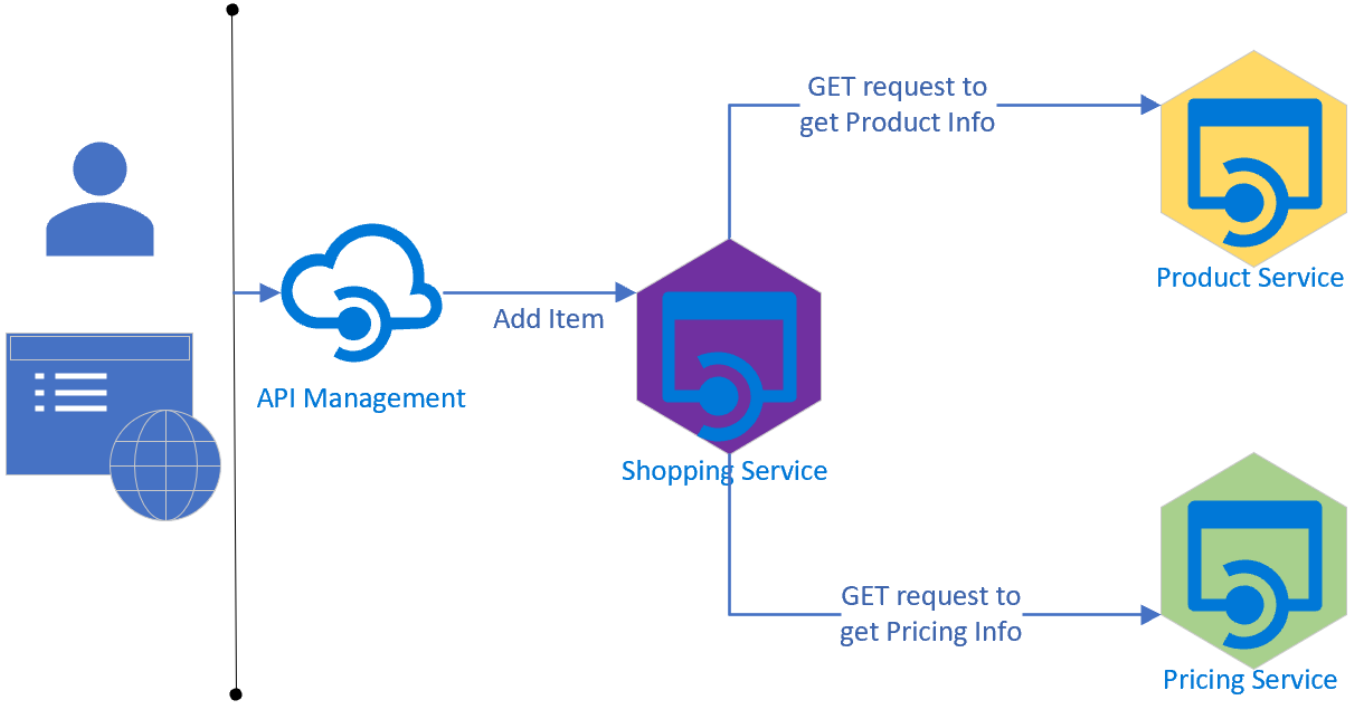


Decomposition based on DDD

Sub Domain	Processes	Service	Domain Category	Technology	Database
Auction Management	Create Auction Update Auction	Auction Service	Core	Node.JS	MySQL on Azure
Bid Management	Make Bid on Active Auctions	Bid Service	Core	Java Spring Boot	Mongo API in Cosmos DB
Payment Management	Make Payments	Payment Service	Supporting	.Net Core	Azure SQL DB
Identity Management	User Sign up/Sign In Profile Edit Password Reset	Identity Service	Generic	Azure AD B2C	-

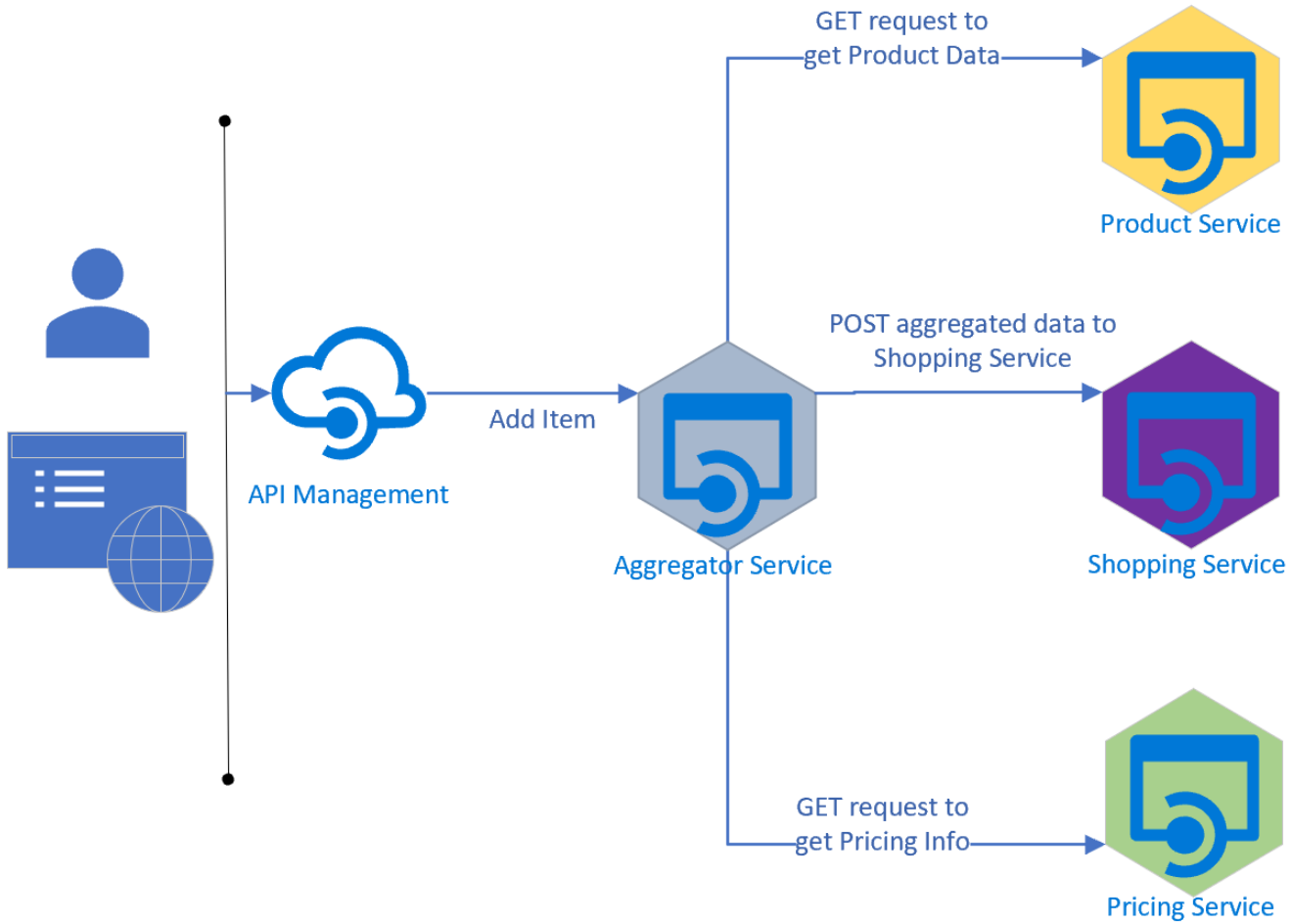
Distributed Database and Patterns

Direct HTTP call



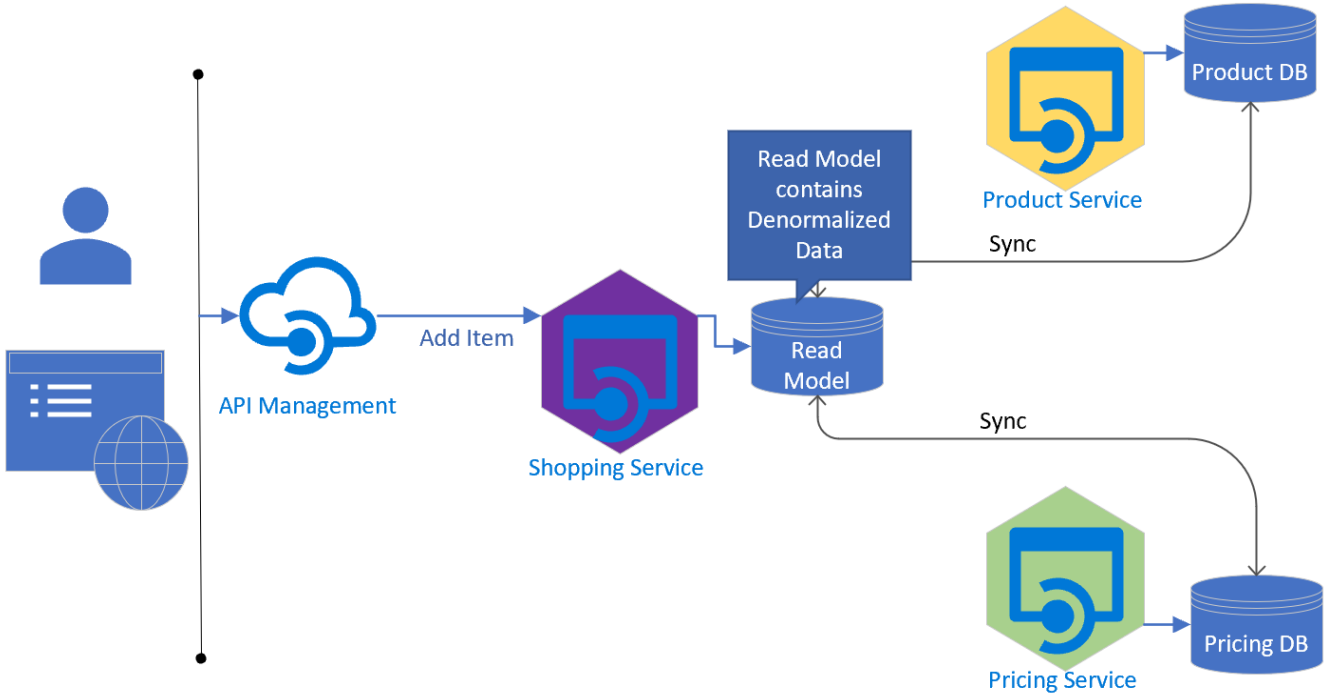
- Direct HTTP calls between microservices can become anti-pattern

Aggregator Pattern



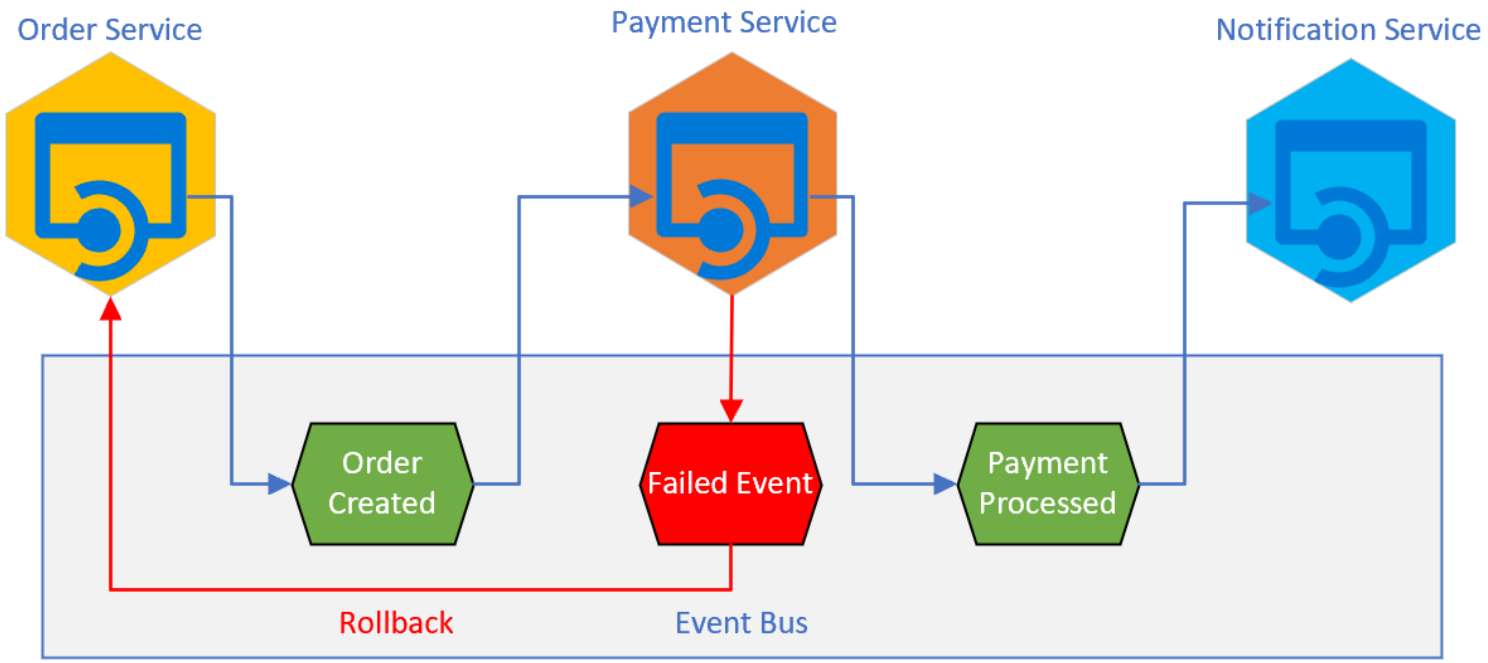
- Service that aggregates and orchestrates calls to multiple services and returns the aggregated result

CQRS – Command Query Responsibility Segregation



- Target Service holds the Read copy of the data
- Decreases coupling and improves response time

Saga Pattern for Distributed Transactions



- Distributed transactions are not possible
- Saga pattern can be implemented to enforce data consistency across services

Auction Service – DB Schema

Column	Data Type	Description
idAuction	int	Primary key, unique auto numerical value
Name	varchar	Name of the auction
Description	varchar	Description of the auction
StartingPrice	decimal	Starting price user can set while creating auction
AuctionDate	date	Date & time when auction was created
Status	int	Active, Completed
Image	mediumtext	Binary value of image
ActiveInHours	int	For how many hours the auction remains active. The number of hours added in AuctionDate to determine the cut off time
UserId	varchar	User who created the Auction
IsActive	tinyint	Active/Inactive
UserName	varchar	Name of the User who made the last bid
BidPrice	decimal	Last bid price value
IsPaymentMade	tinyint	Boolean value indicating if payment is made
BidUser	varchar	Id of the actual bid user
BidId	varchar	Id of the actual bid record

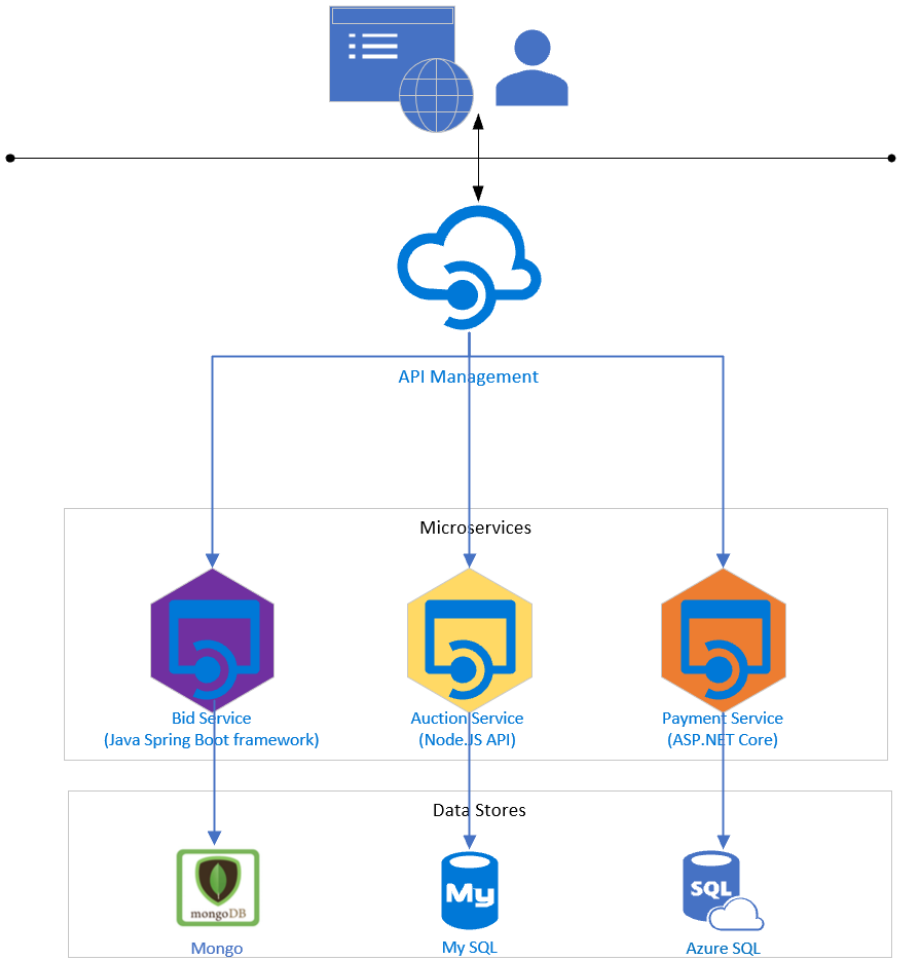
Bid Service – DB Schema

Column	Data Type	Description
bidId	String	Auto number
auctionId	String	Auction table primary key
bidAmount	Number	Amount value of the bid made
userId	String	ID of the User who made the bid
bidDate	Date	Date when the bid is made

Payment Service – DB Schema

Column	Data Type	Description
Id	int	Unique Id of Payment transaction
CreditCardNo	nvarchar	Credit Card number
Name	nvarchar	Name of the credit card holder
IdAuction	int	Auction Id from Auction table
BidUser	nvarchar	User who made the bid
Month	int	Month of credit card expiry date
Year	int	Year of credit card expiry date
PaymentStatus	int	Payment status of Credit card
PaymentDate	datetime2	Date of payment

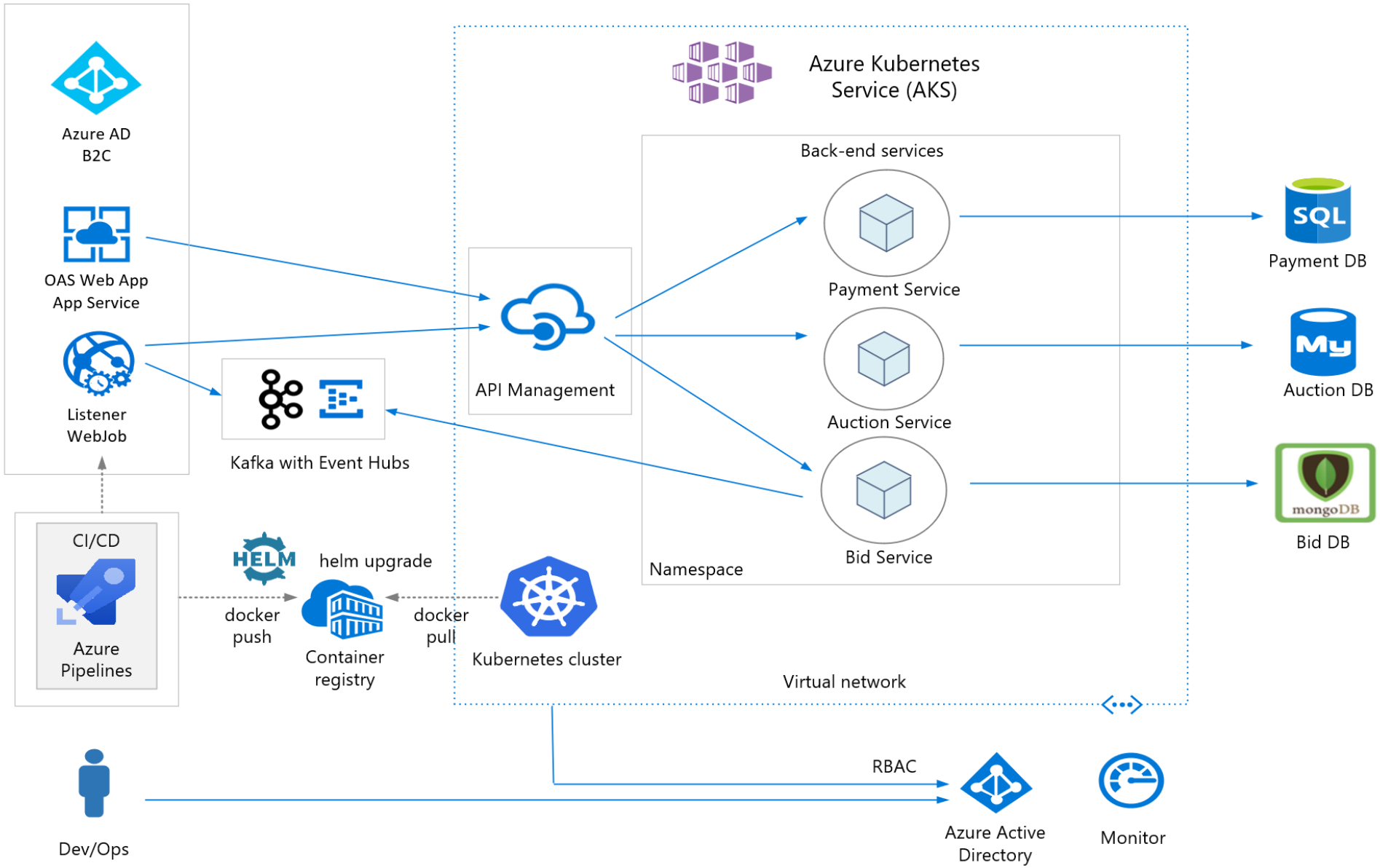
Polyglot Persistence



- An architecture that supports multiple technologies
- Each service implements its own data store of choice
- Leads to better performance, scalability and cost effectiveness
- Introduces challenges with data consistency, fragmentation and management

Online Auction System Architecture

Architecture



How to pick technology when building Microservices?

Microservices Chassis Pattern

- Build microservices on technologies that supports cross-cutting concerns
- Providing Microservices chassis framework that includes externalized configuration, logging, health checks, metrics and distributed tracing
- Technologies support microservices chassis pattern
  - **.NET Core**
  - **Java Spring Boot framework** and Java Spring Cloud
  - **Node.JS – Express JS**, Molecular, Seneca
  - Go kit
  - Micro
  - Gizmo

Communication

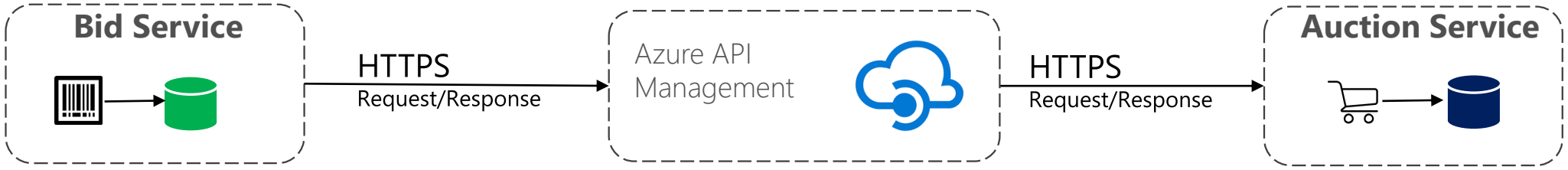
Two approaches

Request/Response (synchronous) Communication

Publish/Subscribe (asynchronous) Communication

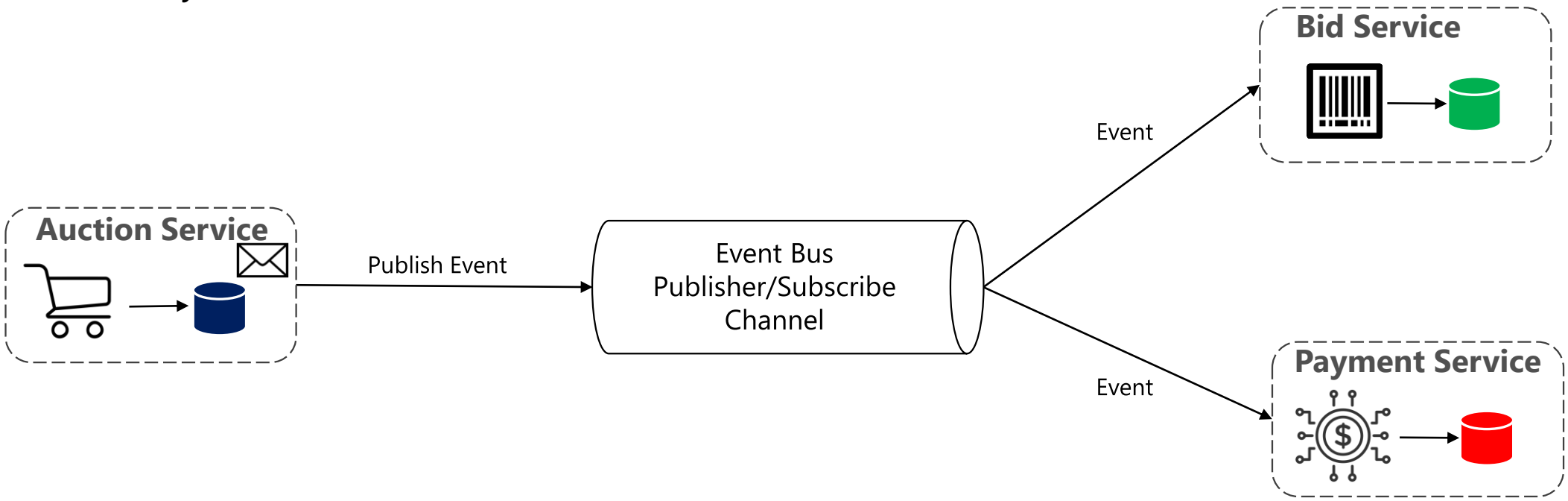
Request/Response Communication

- Direct communication over HTTP/HTTPS
- Simple to implement
- Calls are synchronous

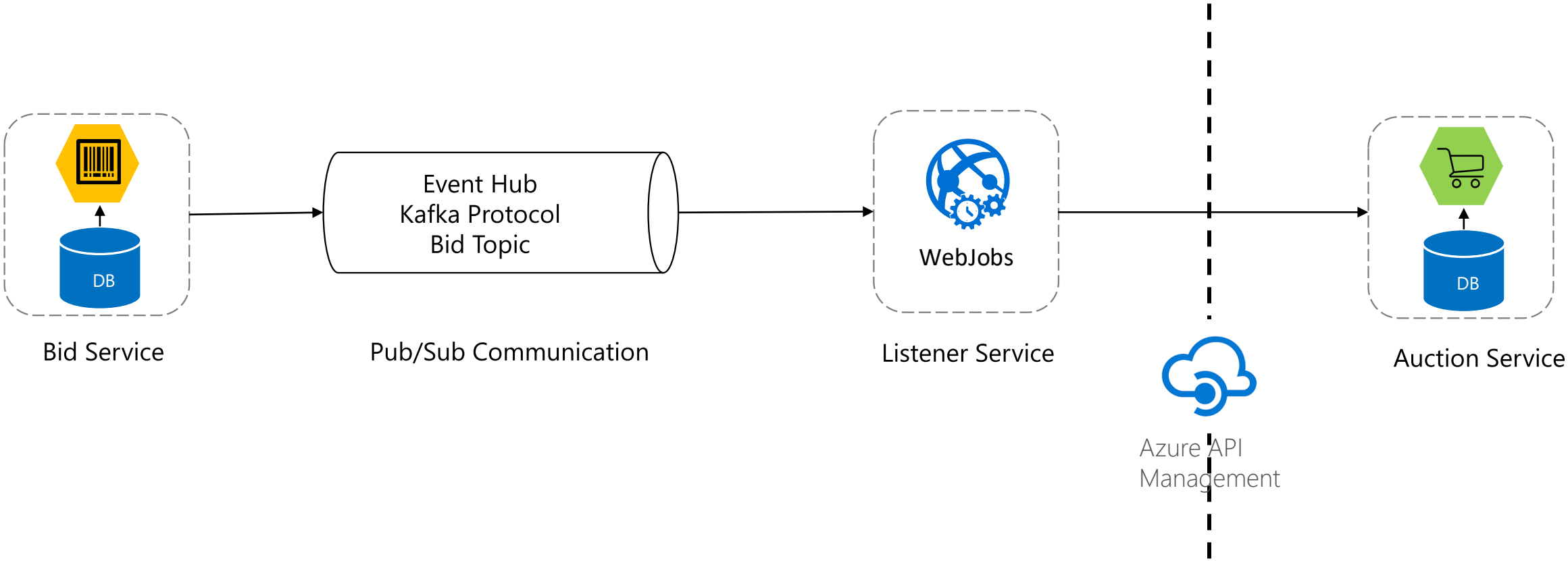


Pub/Sub Communication

- Event based messaging
- Calls are asynchronous



Kafka with Event Hubs

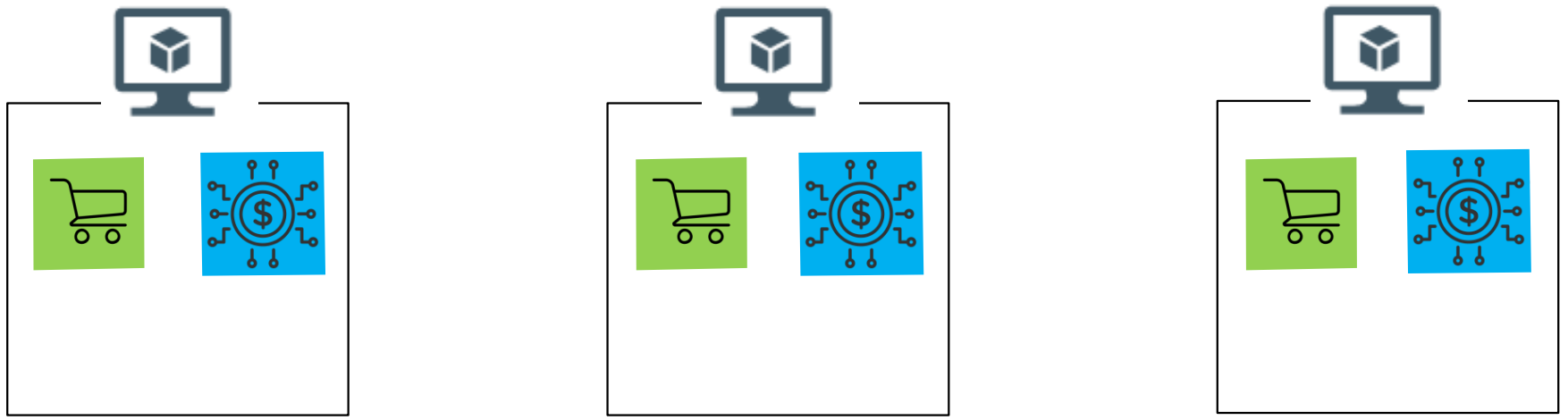


Deployment Patterns and Containerizing Microservices

Deployment Patterns

- Multiple Service Instances Per Host
- Service Instance Per Host
- Service Instance Per VM
- **Service Instance Per Container**

Multiple Service Instances Per Host

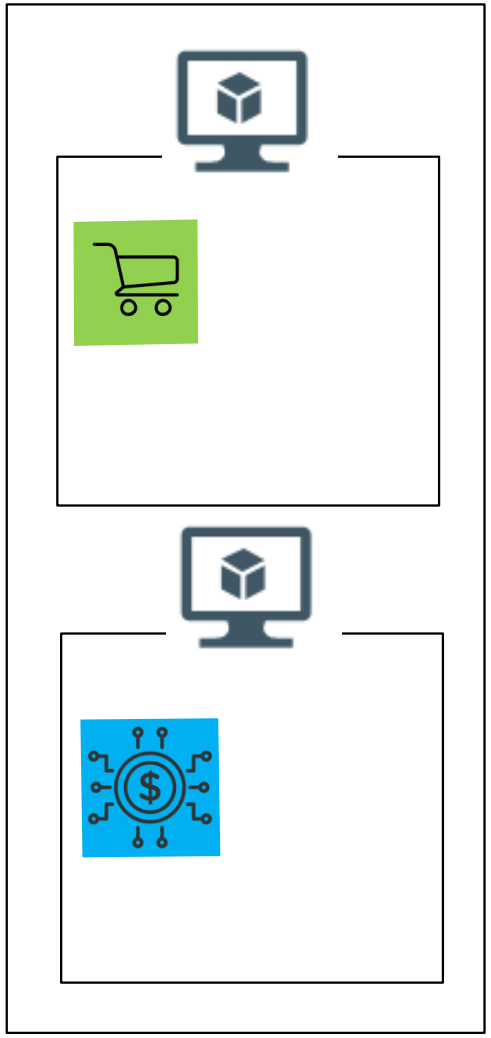


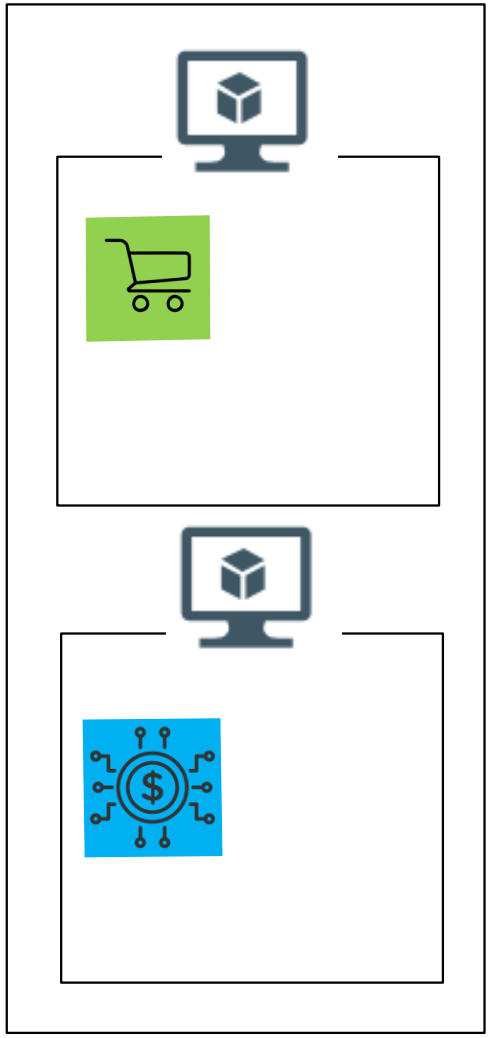
Could be physical host as well

Service Instance Per Host

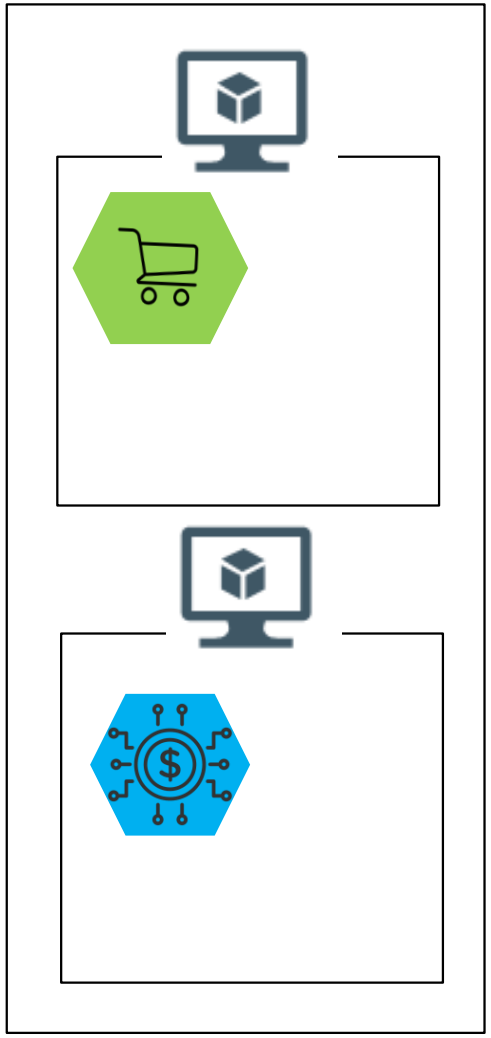
Could be physical host as well

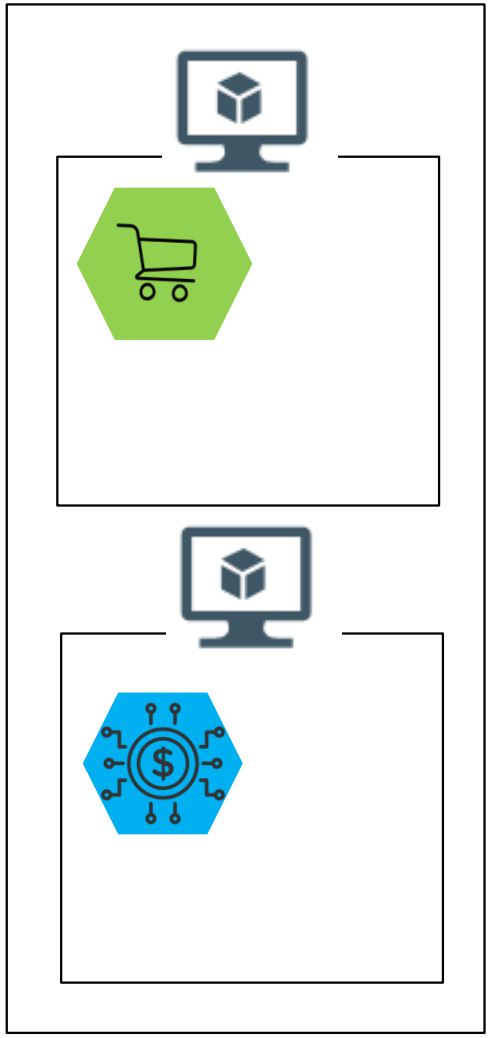
Service Instance Per VM



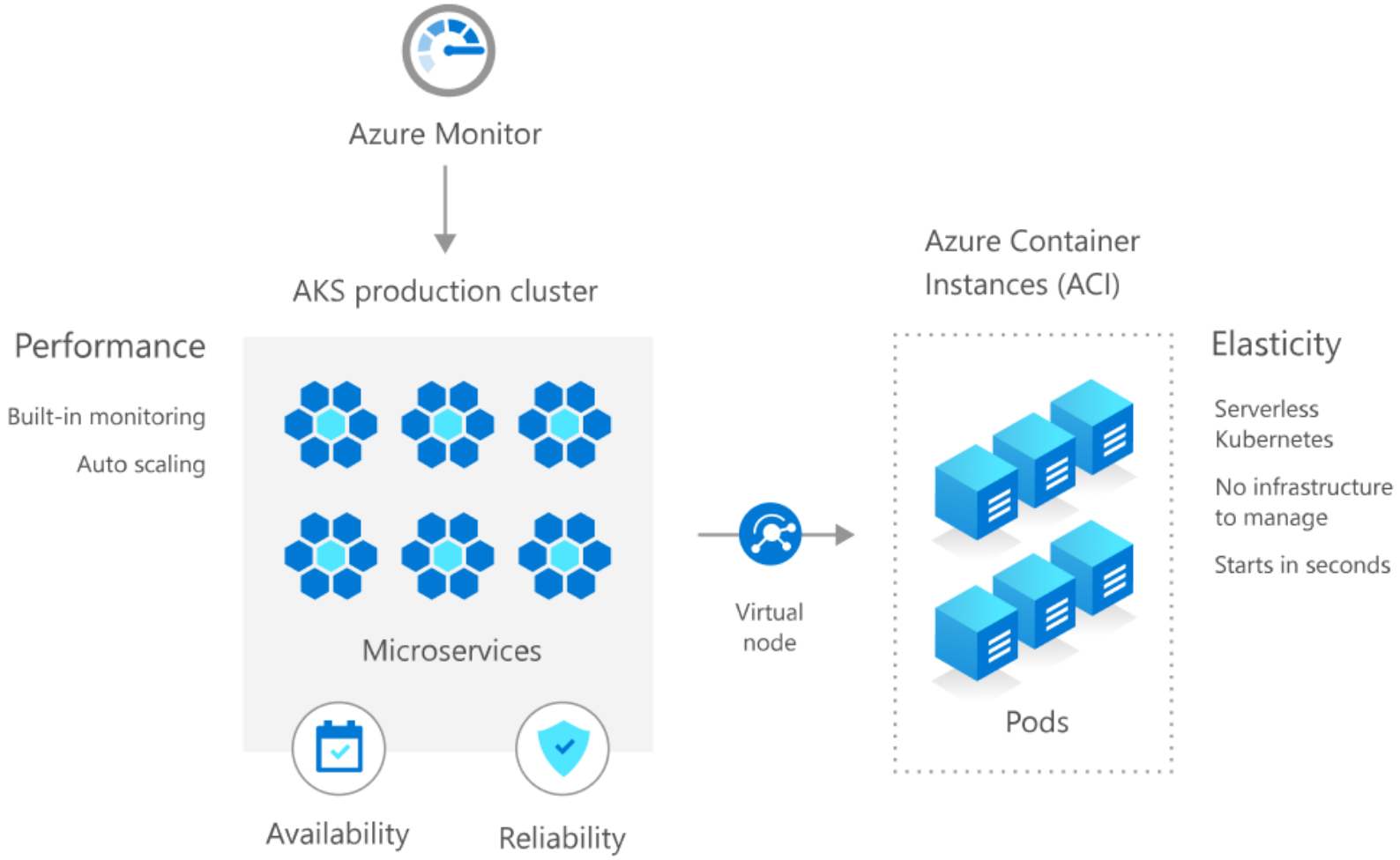


Service Instance Per Container

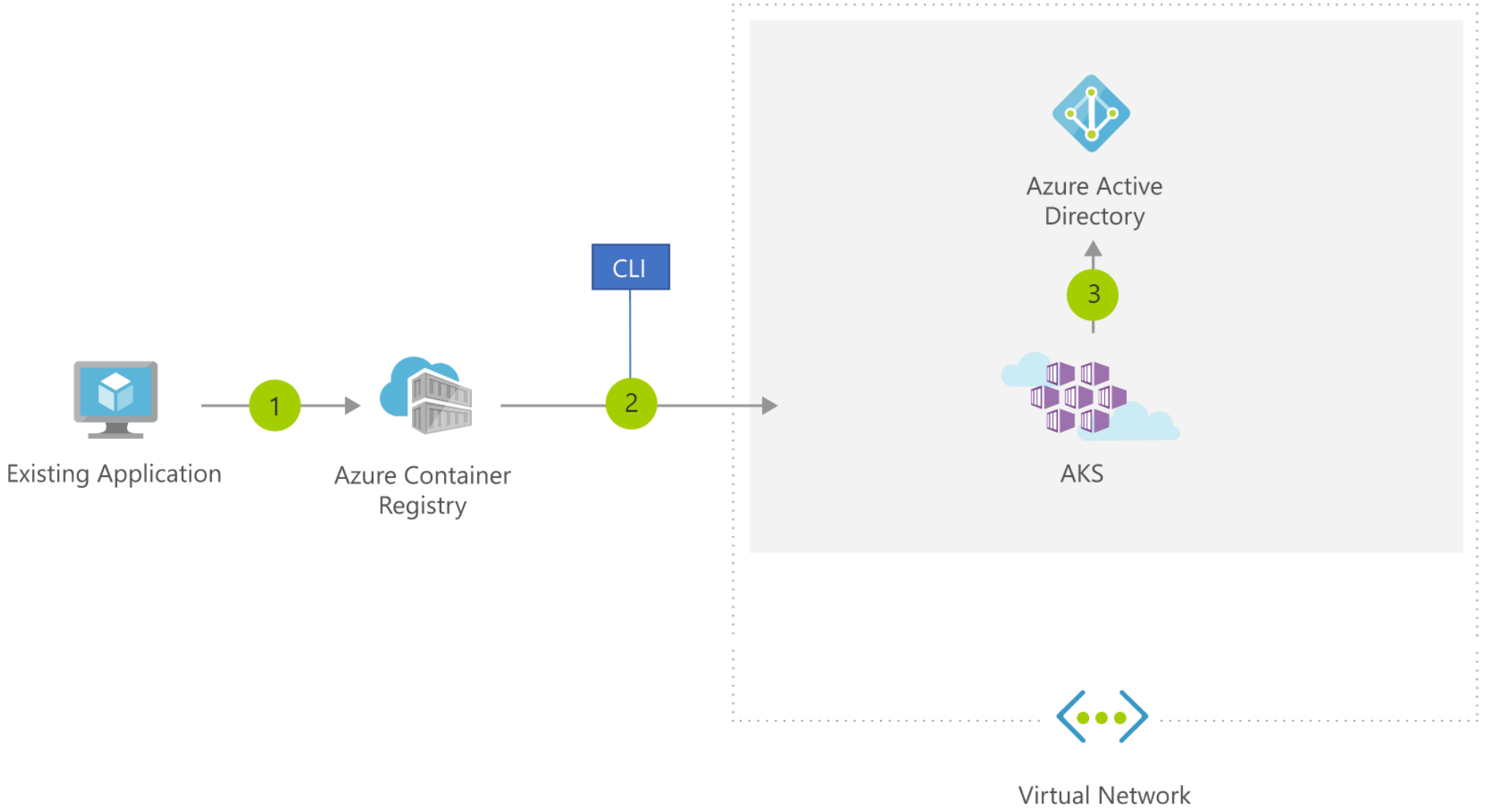




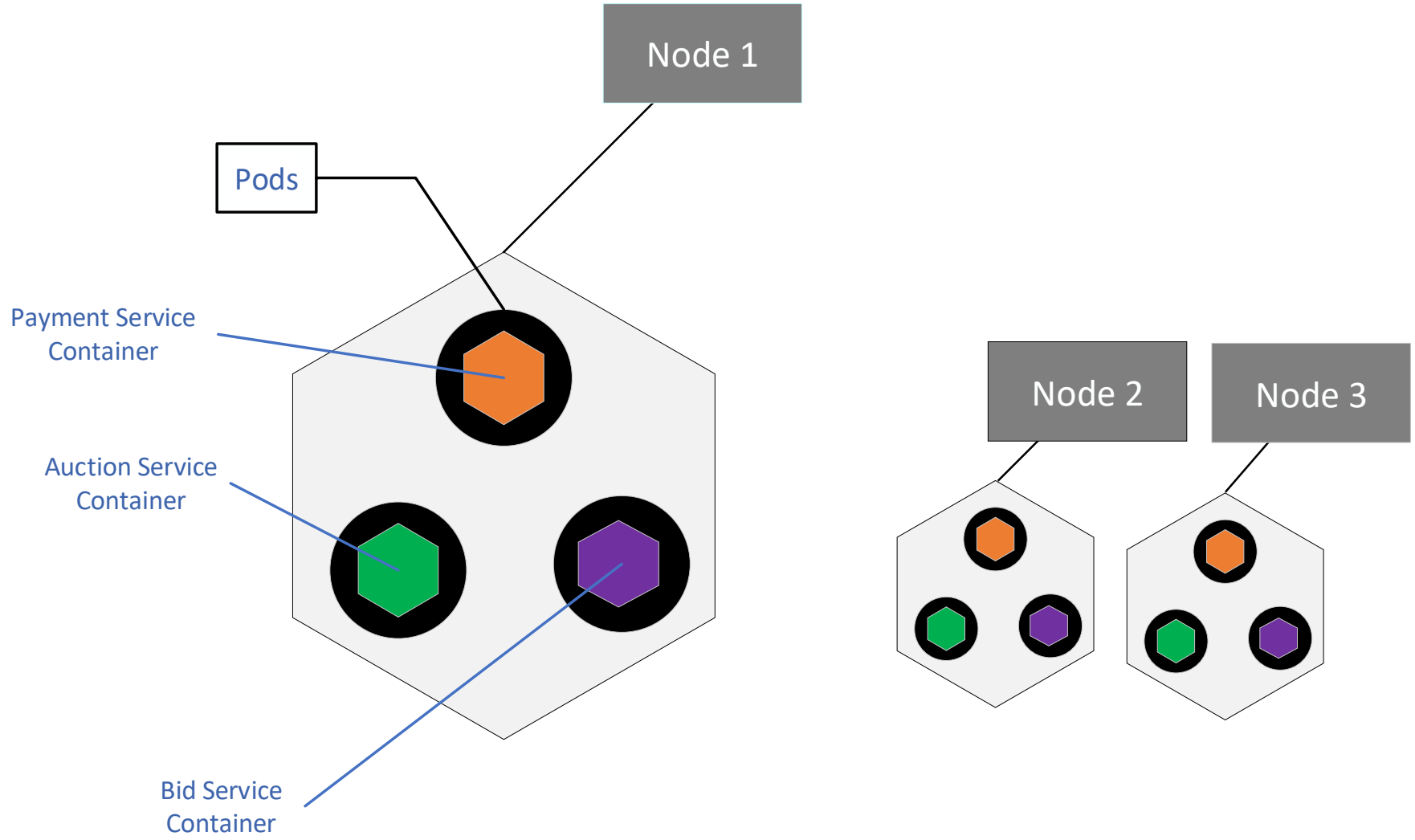
Azure Kubernetes Services



End-to-end AKS Workflow

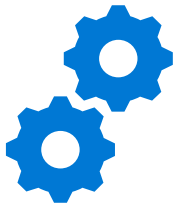


Kubernetes Cluster - Nodes, Pods and Services

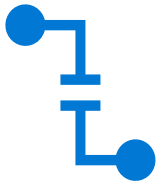


API Gateway

Problem

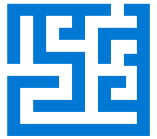


Number of endpoints



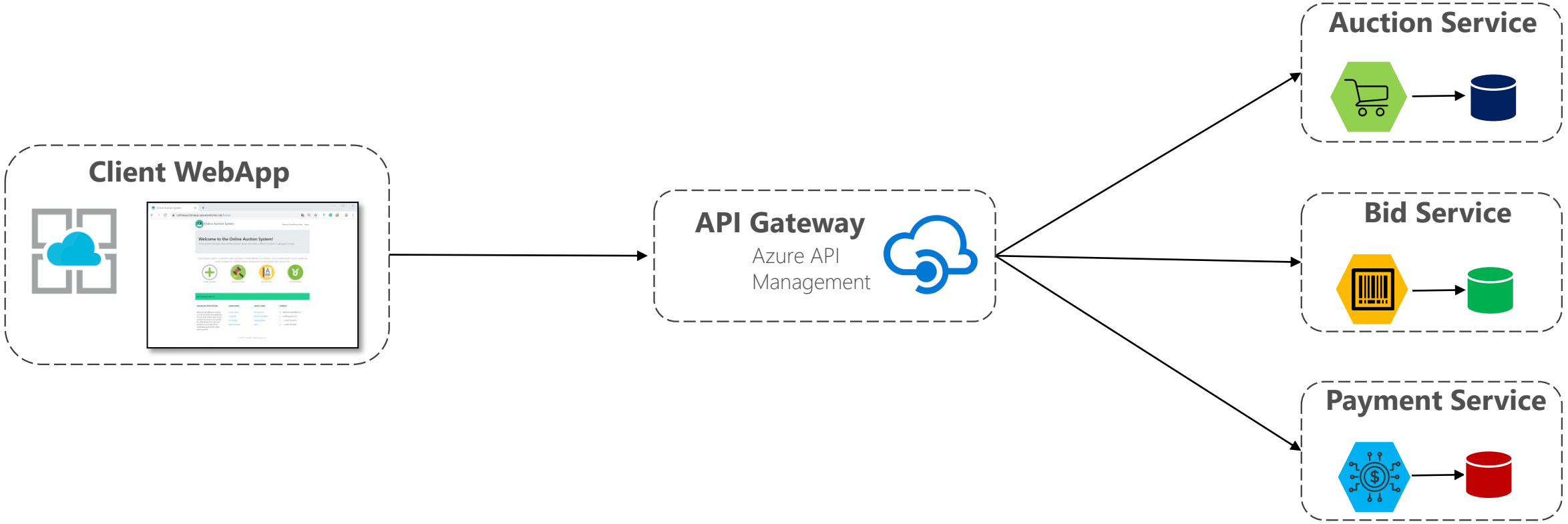
Client becomes tightly-coupled to microservices

Consumers may need specific format of message



Client code becomes complex

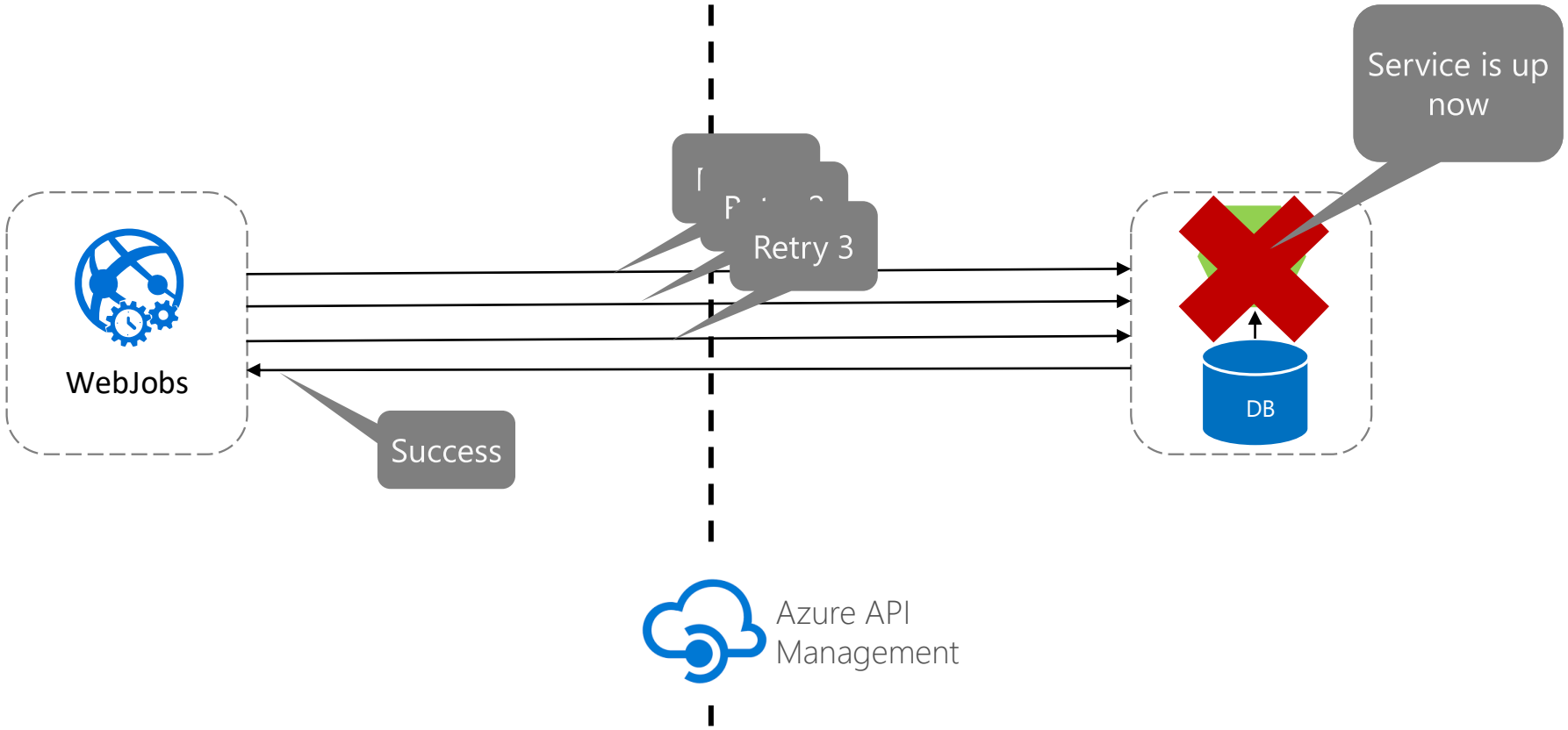
API Management



Resiliency

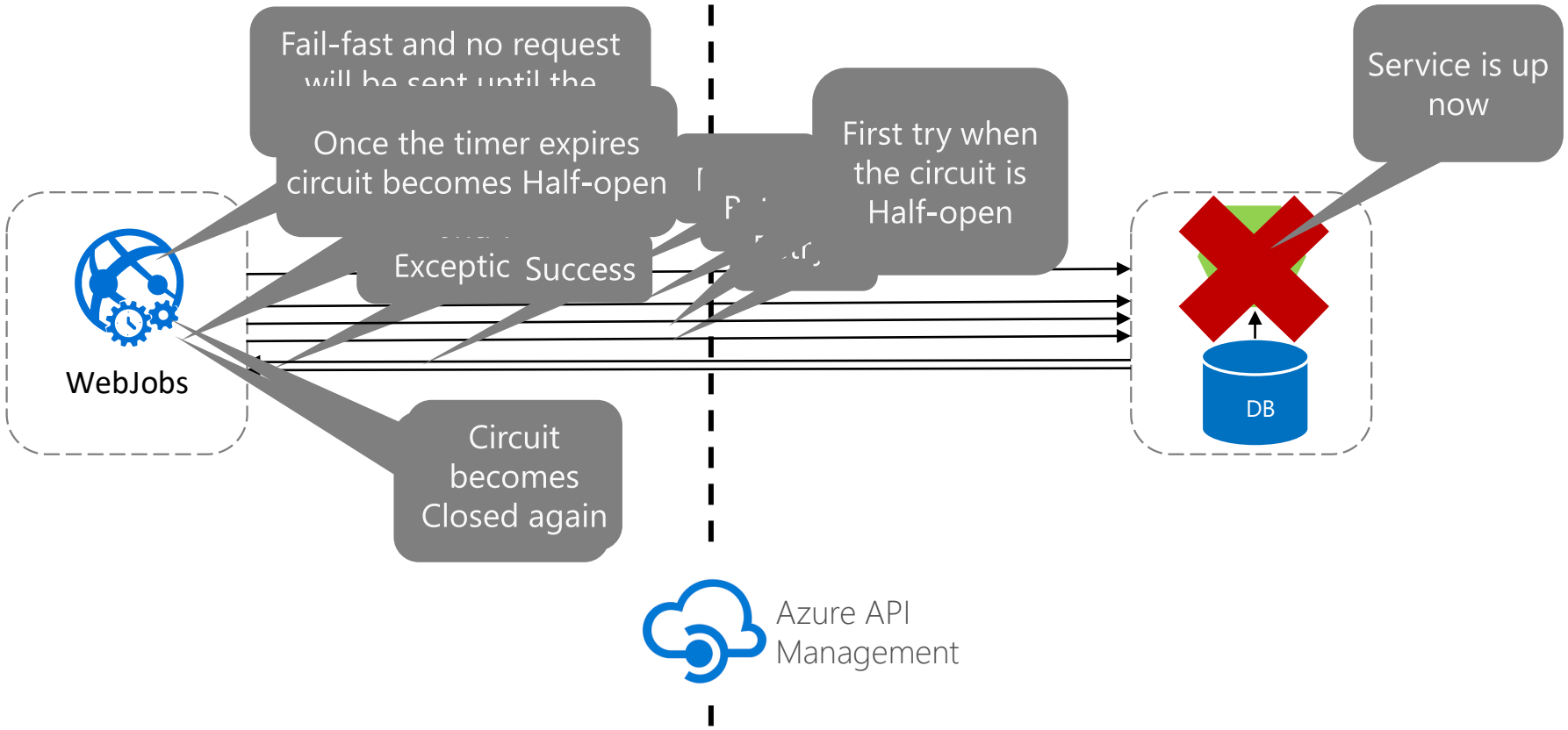
Retry Pattern

- It is used when we need to retry the faulted service for some number of times to get a valid response



Retry Pattern with Circuit Breaker

- The circuit breaker pattern is used in conjunction with a retry pattern



Resiliency in Listener Service

- Used Retry and Circuit Breaker patterns

Application Security

Application Security

- Azure AD B2C
- Microsoft Authentication Libraries (MSAL)
- Configured user flows for Registration, Profile Editing and Password Reset

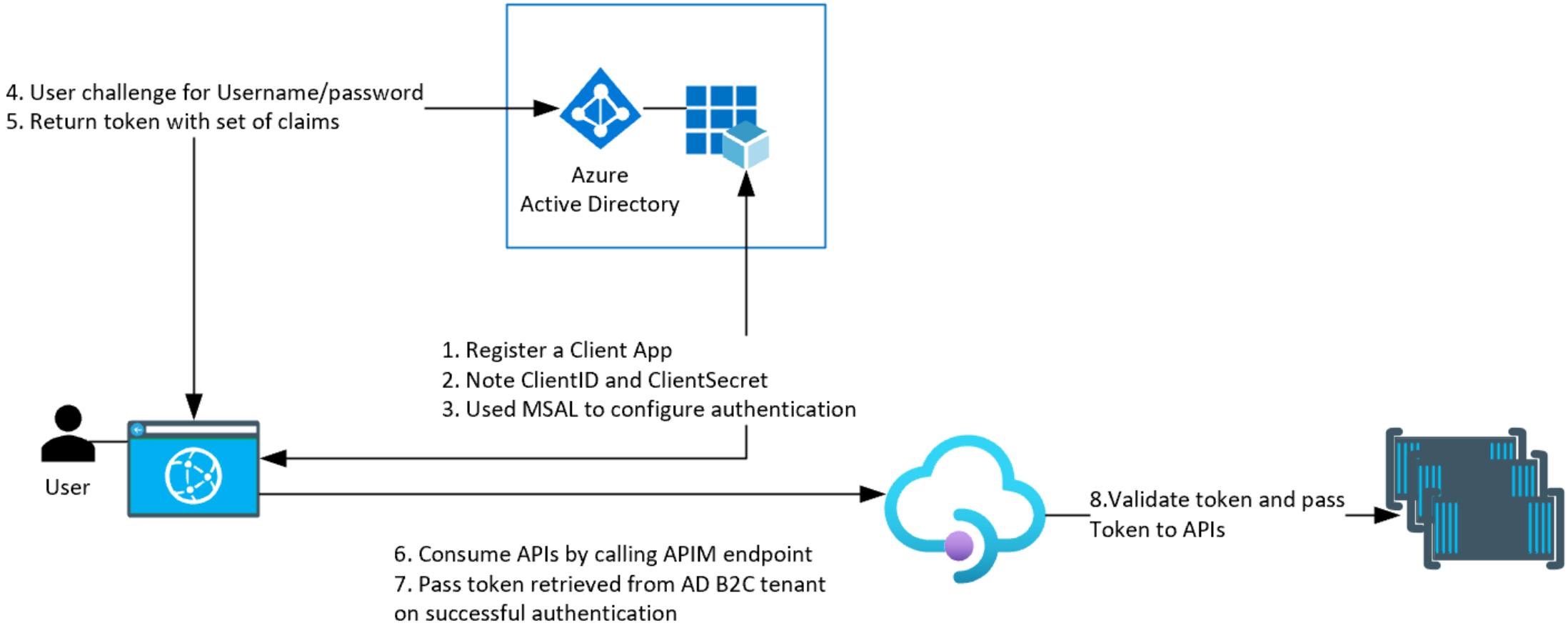
Secure an Azure APIM with Azure AD B2C

- Configure inbound policy in Azure APIM

```
<inbound>
    <base />
    <validate-jwt header-name="Authorization" failed-validation-httpcode="401">
        <openid-config url="https://oasmicroservics.b2clogin.com/oasmicroservics.onmicrosoft.com/v2.0/.well-known/openid-cc
        <audiences>
            <audience>9[redacted]65</audience>
        </audiences>
        <issuers>
            <issuer>https://oasmicroservics.b2clogin.com/[redacted]7/v2.0/</issuer>
        </issuers>
    </validate-jwt>
</inbound>
```

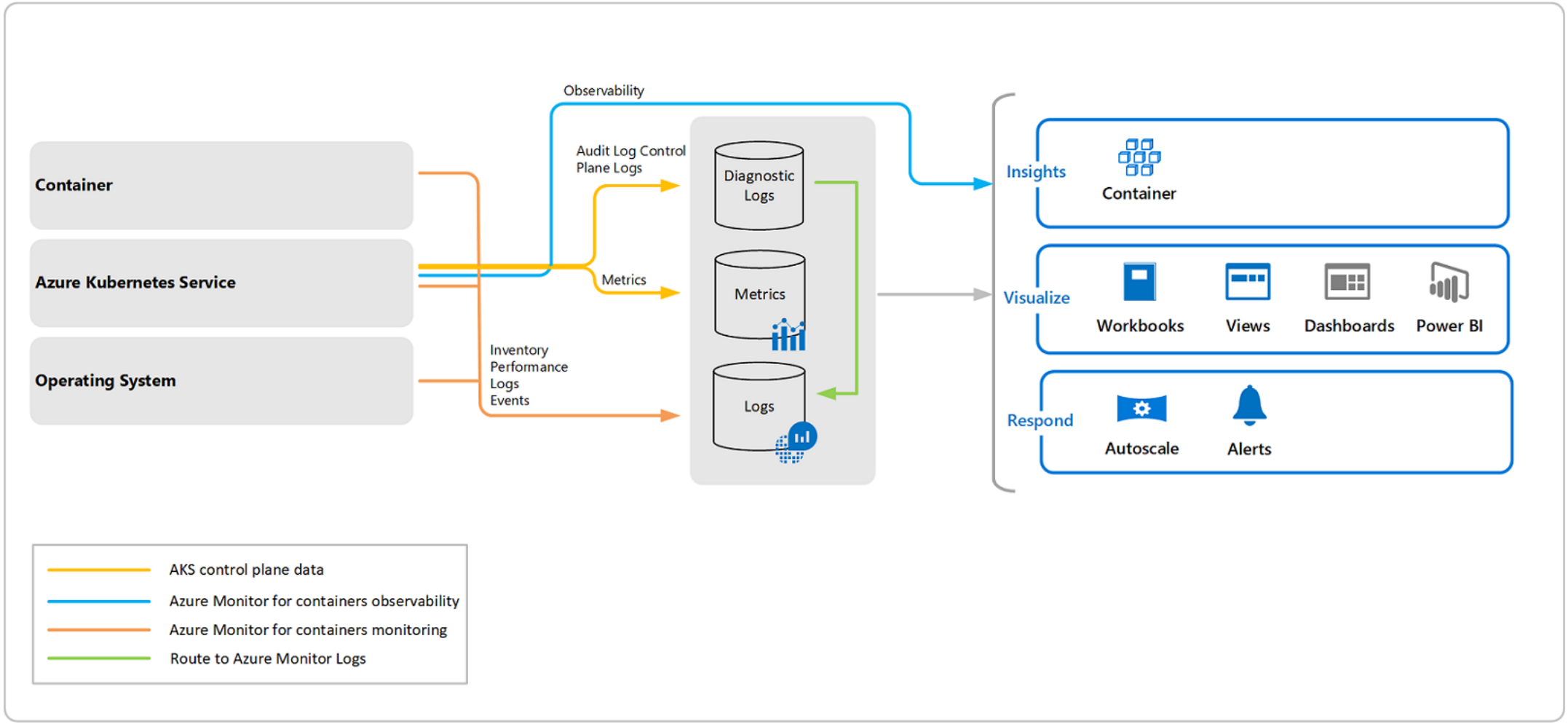
- Pass Bearer token from Web App to APIM

End-to-end Authentication Flow

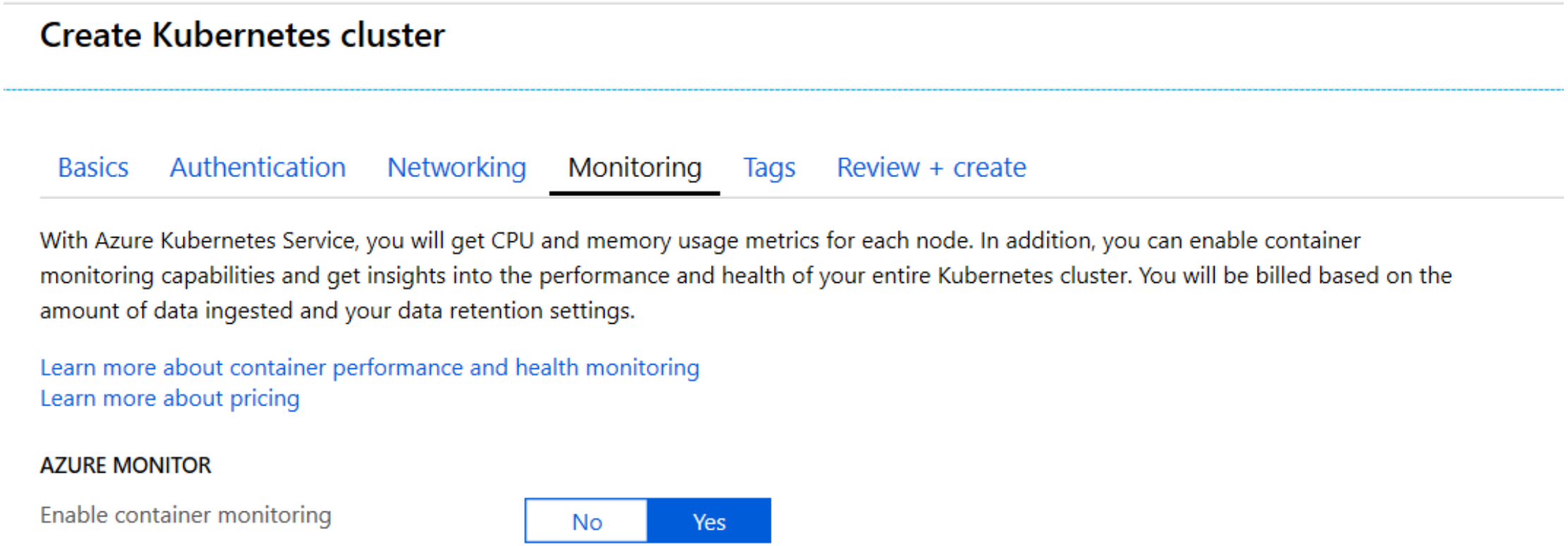


Monitoring

Azure Monitor



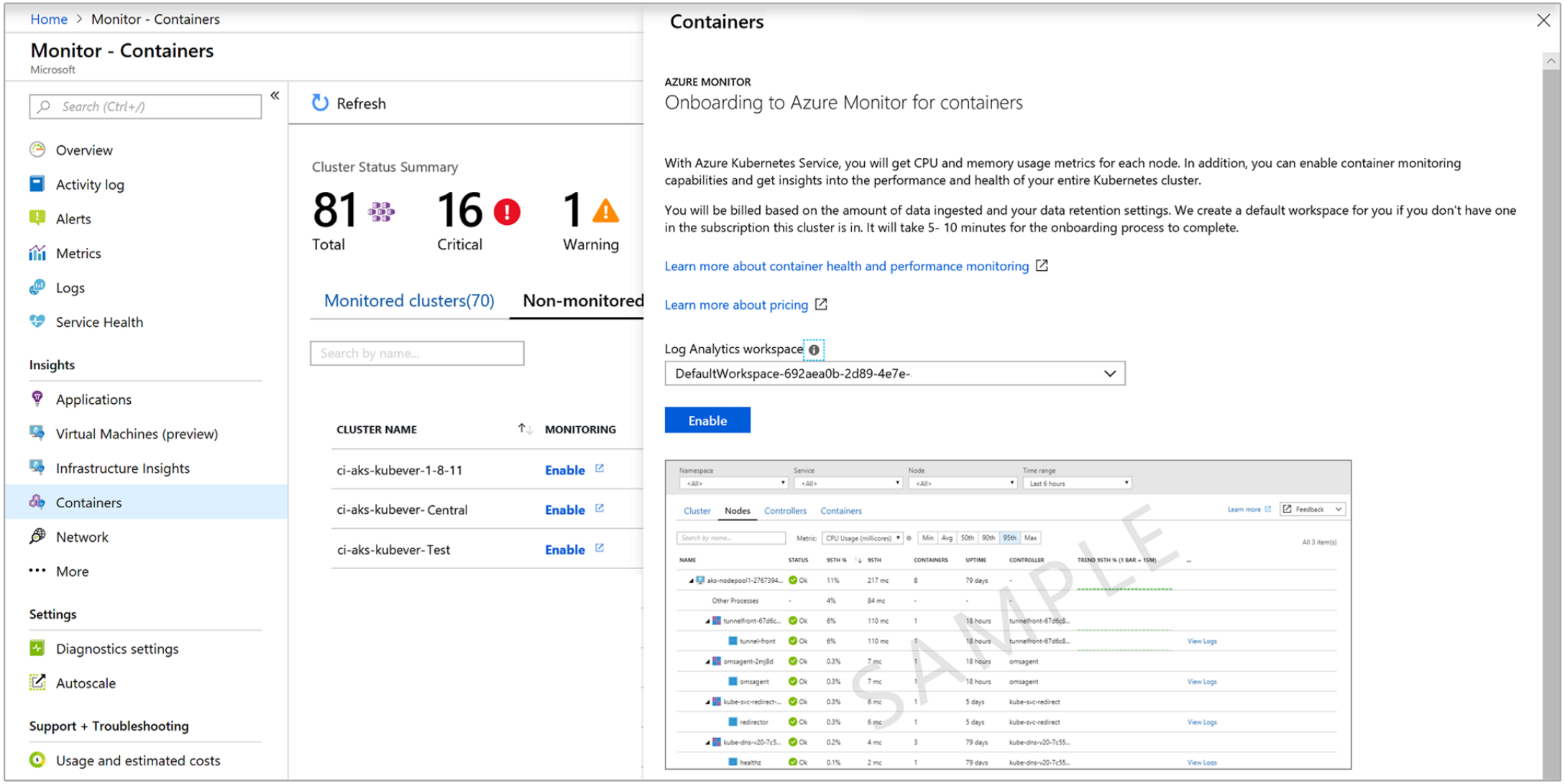
Enable Monitoring in AKS - Portal



Enable monitoring on existing AKS Cluster

az aks **enable-addons -a monitoring** -n oasAKSCluster -g OSSRG

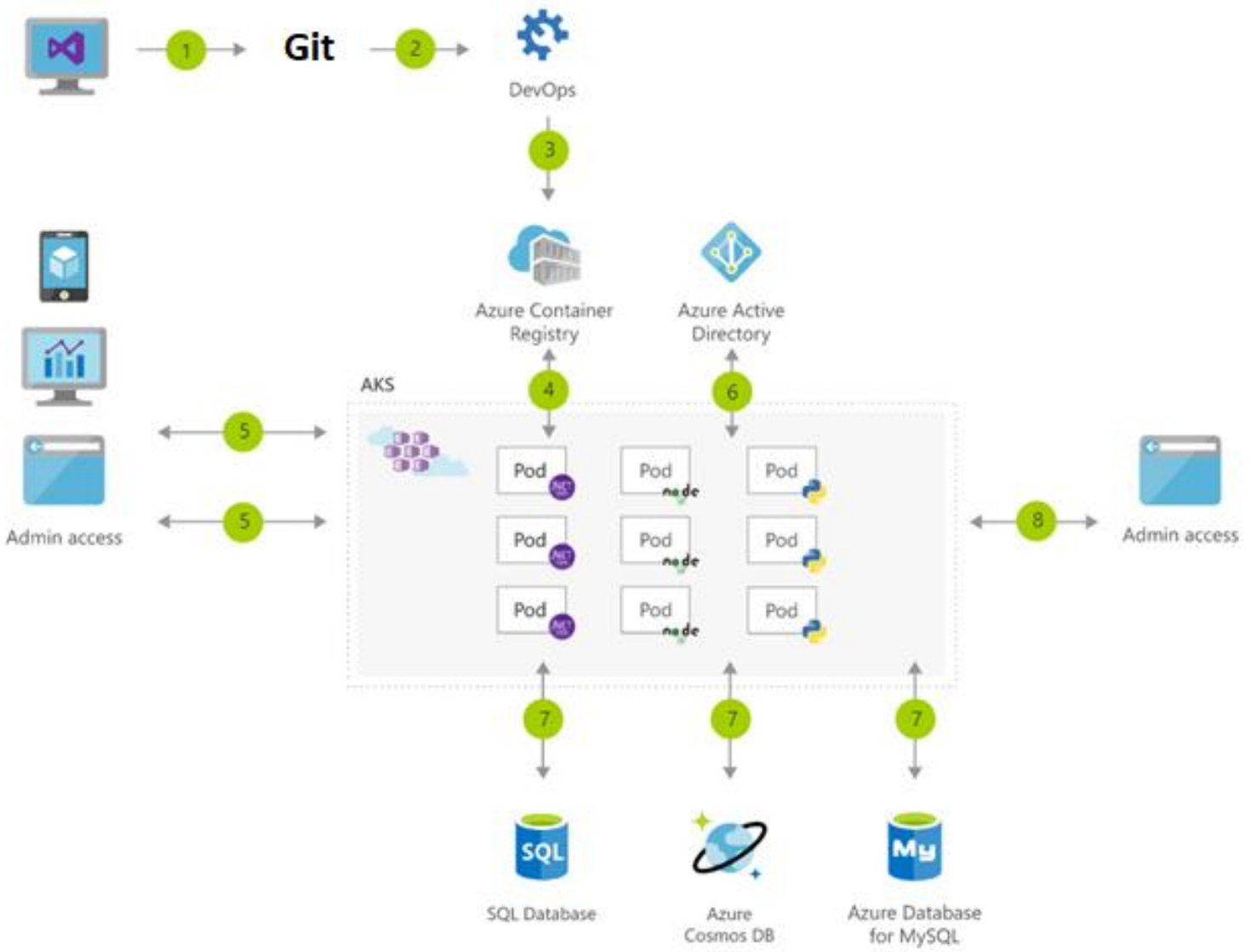
Enable Monitoring for AKS cluster from Azure Monitor



Monitoring Demo

DevOps

Simplify the deployment using Azure DevOps



Session Summary

- Learn the importance of Microservices Architecture
- Showcase a real-life case study of an Online Auction System built on Microservices Architecture
- Learn advanced concepts and patterns used in the application
- Learn what Microsoft Azure provides to make it cloud-native
- Learn how to use Azure DevOps for CI/CD

Q & A

Ovais Mehboob Ahmed Khan

 @ovaismehboob

 linkedin.com/in/ovaismehboob/