PERFICIENT
vision. execution. value.

# OpenShift on Azure Workshop for the .NET Developer

# Workshop Agenda

1. Business Case?
2. The OpenShift Platform
3. Development Basics
4. Advanced Deployment Techniques
5. Managing Secrets and Environment Variables
6. Persistent Storage
7. Services and Routes
8. Azure DevOps ?
9. Advanced Techniques and Gotcha's ?

# Introductions

Steve Holstad
Azure Practice Director, Microsoft Cloud Solutions
steve.holstad@perficient.com

Tim McCarthy
Senior Solutions Architect, Microsoft Cloud Solutions
tim.mccarthy@perficient.com

Leroy Baynum
Senior Solutions Architect, Microsoft Cloud Solutions
leroy.baynum@perficient.com

# Intermodal Container

- An intermodal container is a large standardized shipping container, designed and built for intermodal freight transport.
- Can be used across different modes of transport – from ship to rail to truck – without unloading and reloading their cargo.
- Containers in IT have very similar properties

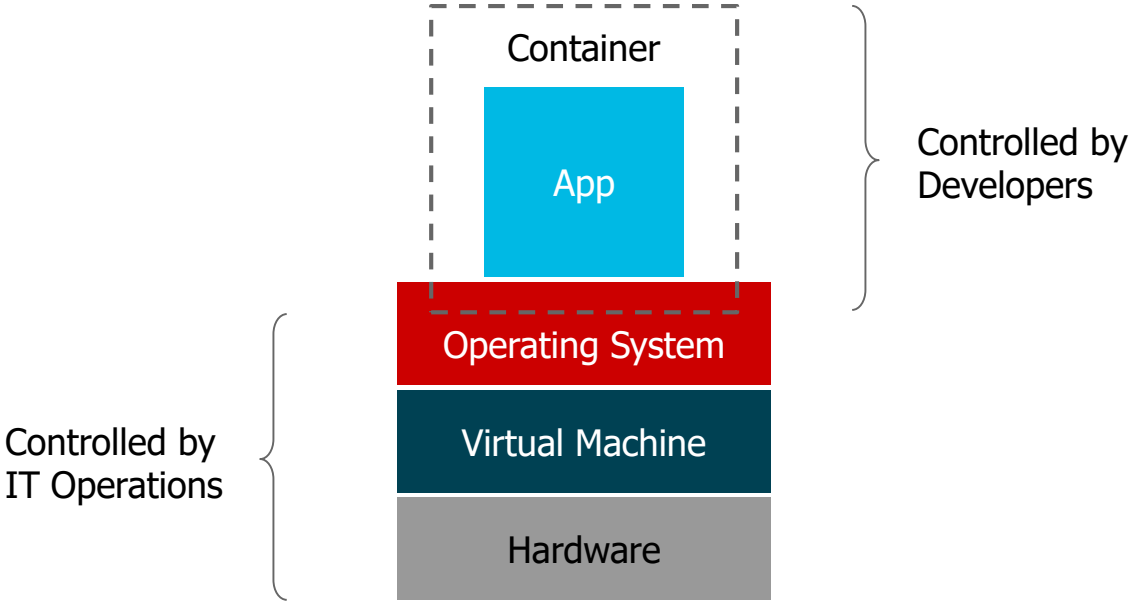

# Containers Overview


- Standard: Containers are an industry standard, so they could be portable anywhere
- Lightweight: Containers share the machine's OS system kernel and therefore do not require an OS per application, driving higher server efficiencies and reducing server and licensing costs
- Secure: Applications are safer in containers and Docker provides the strongest default isolation capabilities in the industry

A container is a standard unit of software that packages up code and all its dependencies so the application runs quickly and reliably from one computing environment to another. A Docker container image is a lightweight, standalone, executable package of software that includes everything needed to run an application: code, runtime, system tools, system libraries, and settings. Containerized software will always run the same, regardless of the infrastructure. Containers isolate software from its environment and ensure that it works uniformly despite differences for instance between development and staging.

# Containers Overview

**What are Containers in IT / DevOps**



# Containers Overview

## Why are containers good

- Creates abstraction layer between your app and where it runs
- Simplifies DevOps by providing a consistent environment for Developers and Operations by creating identical environments for development, testing, staging, and production
- Write once – Deploy anywhere

# Containers Overview

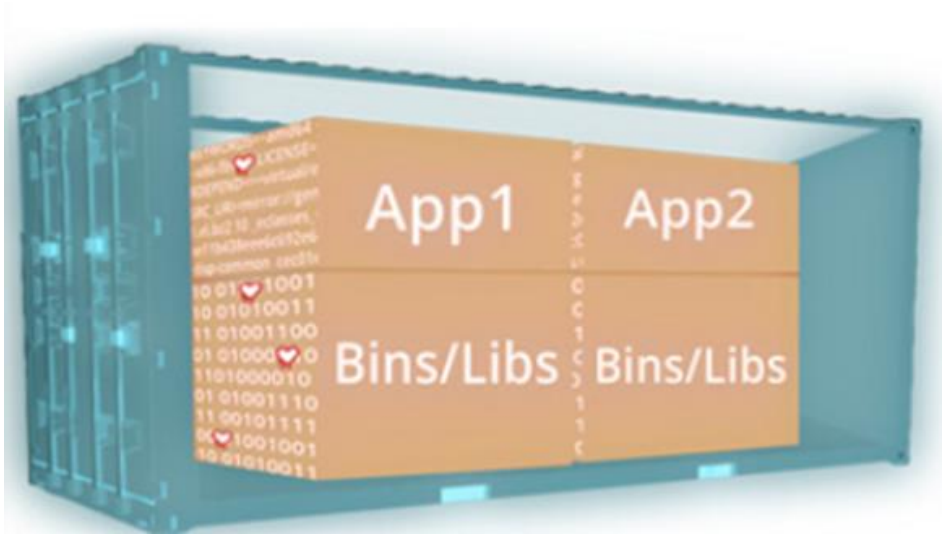
## What are containers? Infrastructure

- Sandboxed application processes that run on a shared Linux OS kernel or Windows kernel
- SE Linux and just recently Windows
- Simpler, lighter, and denser than virtual machines
- Portable across different environments

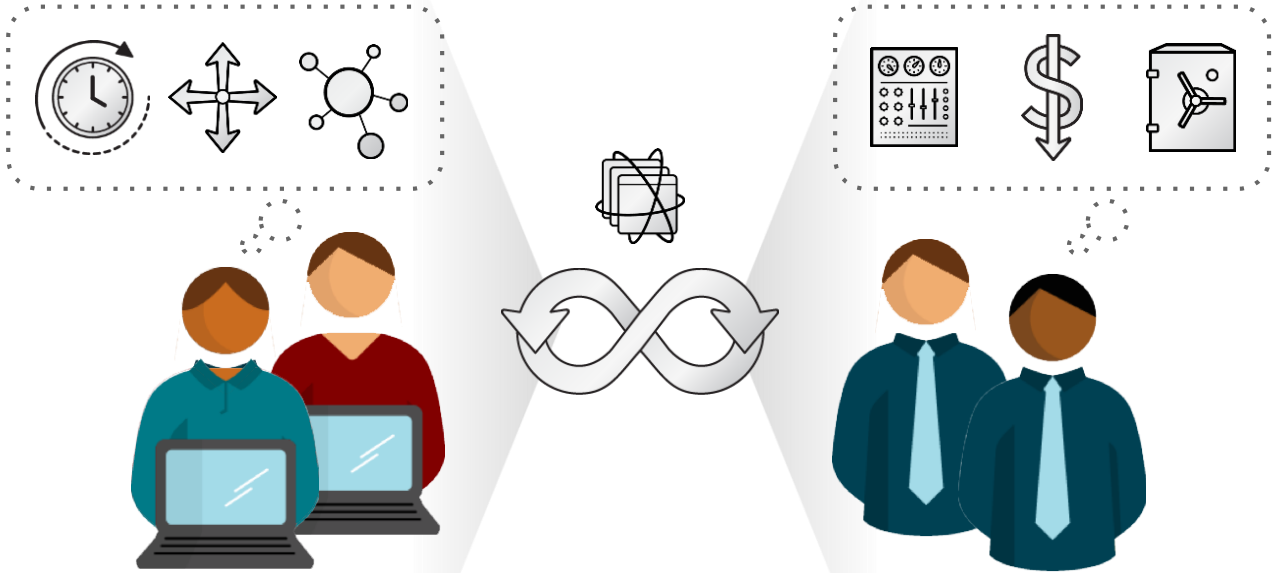
# Containers Overview

## What are containers? Applications

- Package the application and all of its dependencies
- Deploy to any environment in seconds and enable CI/CD
- Easily access and share containerized components



# Peace on Earth



# Orchestration

## Scheduling

Decide where to deploy containers.

## Security

Control who can do what

## Lifecycle and health

Keep containers running despite failures

## Scaling

Scale containers up and down

## Discovery

Find other containers on the network

## Persistence

Survive data beyond container lifecycle
.

## Monitoring

Visibility into running containers

## Aggregation

Compose apps from multiple containers

# kubernetes

## Service discovery and load balancing

No need to modify your application to use an unfamiliar service discovery mechanism. Kubernetes gives pods their own IP addresses and a single DNS name for a set of pods, and can load-balance across them.

## Storage orchestration

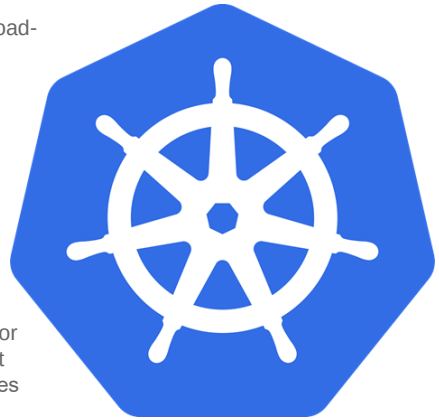
Automatically mount the storage system of your choice, whether from local storage, a public cloud provider such as Azure, GCP or AWS, or a network storage system such as NFS, iSCSI, Gluster, Ceph, Cinder, or Flocker.

## Automated rollouts and rollbacks

Kubernetes progressively rolls out changes to your application or its configuration, while monitoring application health to ensure it doesn't kill all your instances at the same time. If something goes wrong, Kubernetes will rollback the change for you. Take advantage of a growing ecosystem of deployment solutions.

## Batch execution

In addition to services, Kubernetes can manage your batch and CI workloads, replacing containers that fail, if desired.



## Automatic bin packing

Automatically places containers based on their resource requirements and other constraints, while not sacrificing availability. Mix critical and best-effort workloads in order to drive up utilization and save even more resources.

## Self-healing

Restarts containers that fail, replaces and reschedules containers when nodes die, kills containers that don't respond to your user-defined health check, and doesn't advertise them to clients until they are ready to serve.

## Secret and configuration management

Deploy and update secrets and application configuration without rebuilding your image and without exposing secrets in your stack configuration.

## Horizontal scaling

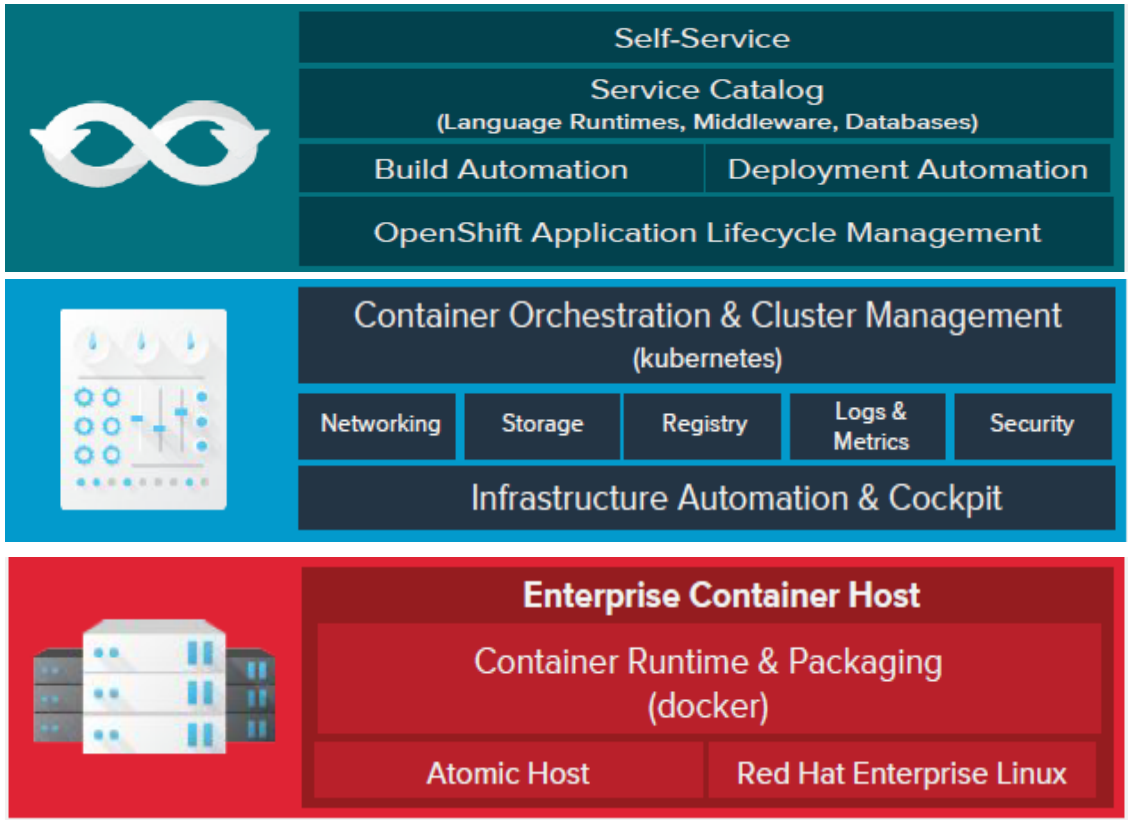
Scale your application up and down with a simple command, with a UI, or automatically based on CPU usage.

# kubernetes

- Developed by Google
- A collection of APIs for managing container-based workloads
- Open-source ORCHESTRATION framework
- Manages container lifecycle, health and scaling
- Many enterprise contributors such as: Google, Red Hat and Microsoft


Red Hat OpenShift Container Platform

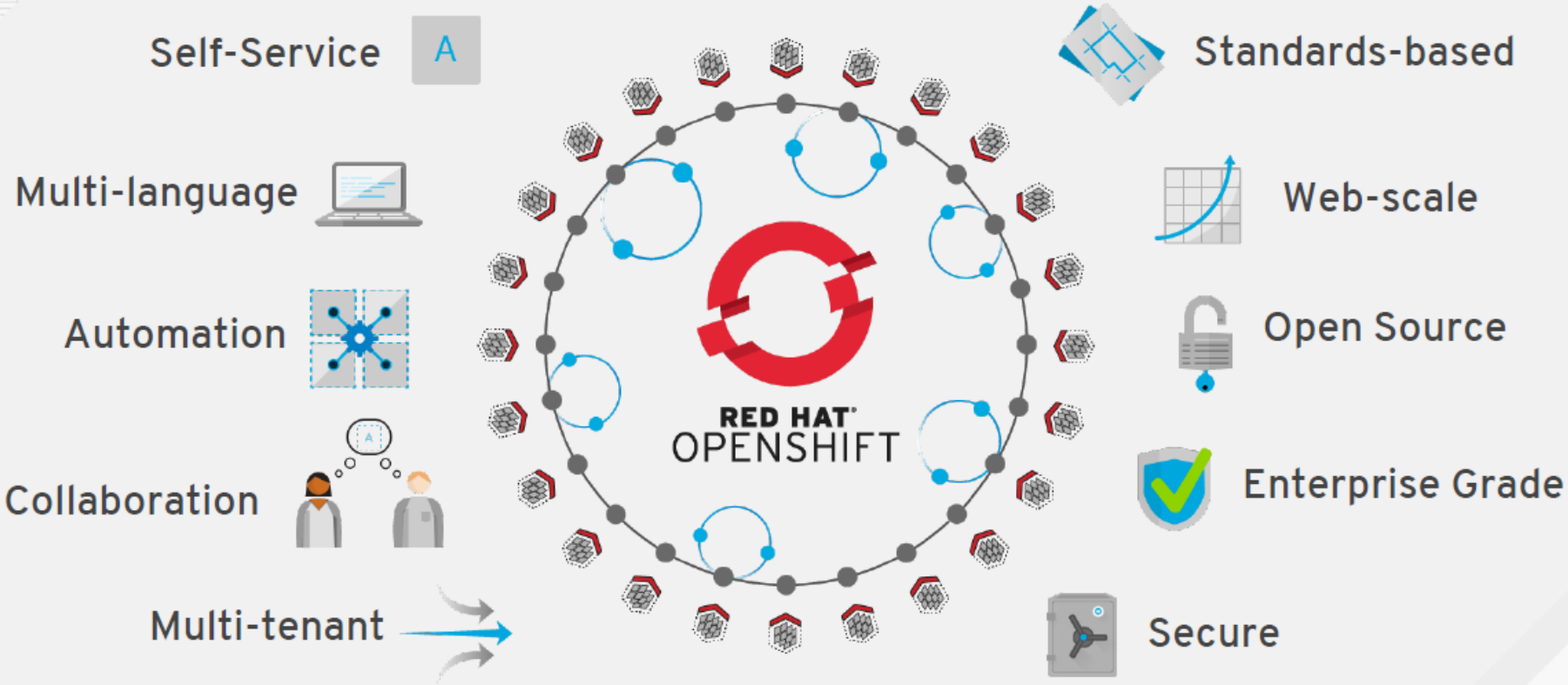
# OpenShift – The Agility Enabler!




Development Experience

Enterprise Kubernetes++ container orchestration

Trusted by Fortune Global 500 companies


Self-Service
Multi-language
Automation
Collaboration
Multi-tenant
RED HAT
OPENSHIFT
Standards-based
Web-scale
Open Source
Enterprise Grade
Secure

# OpenShift adds value to Kubernetes

| | |
|---|---|
| Multi-tenancy | Teams and Collaboration |
| Routing & Load Balancing | Quota Management |
| CI/CD Pipelines | Image Build Automation |
| Role-based Authorization | Container Isolation |
| Capacity Management | Vulnerability Scanning |
| Infrastructure Visibility | Chargeback |


So What?

# Market and Customer Situation

**Organizations must adapt to fast, ongoing change in market situations, customer demand, and competitors.**

**Containerized environments can provide the speed, innovation, and flexibility to become a digital business.**

# Customer Challenge

Building a hybrid, containerized environment can be time and resource consuming

## Expertise

Some organizations don't have the expertise or resources to build their own solution.
.

## Data Sovereignty

Maintaining data sovereignty in hybrid environments can also be challenging
.

## Security

Complex environments make managing security across infrastructure difficult.

You need a trusted, supported containerized hybrid infrastructure solution that lets you launch, scale, and adapt applications quickly and efficiently.

# Azure Red Hat OpenShift


Red Hat and Microsoft help you adopt container and hybrid cloud technologies more easily with Red Hat OpenShift Container Platform running on Microsoft Azure.

Build an enterprise-grade, containerized hybrid environment rapidly and easily. Red Hat and Microsoft deliver the consistency, support, and managed services you need to meet modern business needs.

# Azure Red Hat OpenShift

**Ensure a consistent application life-cycle experience across on-site and cloud environments.**

- Build and deploy applications where it makes the most sense for you.
- Eliminate inconsistencies using a single platform across environments.
- Speed application development and deployment.

**Combine Red Hat and Microsoft infrastructure into a scalable, reliable, and supported hybrid environment.**

- Take advantage of industry expertise.
- Simplify issue resolution with collaborative support.
- Deploy tested, integrated solutions.

**Focus on application development, not on container platform management.**

- Deploy a container environment faster.
- Offload platform management to the experts.
- Give developers self-service capabilities.
- Maintain security and regulatory compliance.

# Business Drivers for IT Leadership

- Adopting solutions that are certified, validated, and supported by enterprise service-level agreements (SLAs), services, and support
- Boosting innovation using existing staff skills and investments
- Accelerating time to value for new products and services
- Creating a global presence
- Scaling efficiently to meet growing demand
- Ensuring compliance through increased visibility and auditing
- Aligning your technology roadmap with business requirements

# Operations Drivers

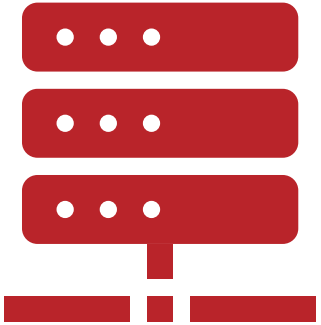
- Integrating new technologies into existing development pipelines and tools
- Ensuring that all applications meet deployment and security requirements
- Ensuring compliance across all operations and infrastructure
- Improving management and deployment efficiency with automation
- Implementing scalability on a global scale
- Implementing advanced monitoring, reporting, and traceability to improve visibility and understanding
- Ensuring users are authenticated for appropriate applications and resources

# Development Drivers

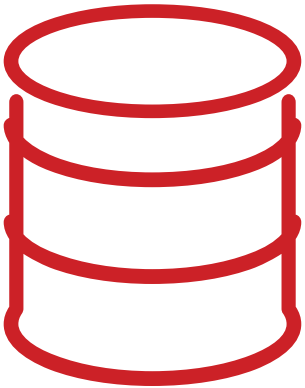
- Moving projects from development to production more quickly and easily
- Implementing microservices, web-scale applications, and containers
- Quickly deploying your desired tools from a central repository
- Gaining access to resources more quickly and easily, i.e. deploying and controlling containers on demand
- Implementing scalability on a global scale
- Adopting open application programming interfaces (APIs) and software development kits (SDKs)
- Increased flexibility in choice of languages and Open Sourced solutions
- Integrated development and deployment experience
- Abstract logging and configuration management to the container run time

# Today

**Client**



**Server**



**Data**

# The Problem

- Solution Size – How many developers can work on the solution at once
  - Team of 60 developers
  - Safe (Scaled Agile Framework)
- Developer Machine setup
- Application Feature Dependencies – one app feature depends on another both must be deployed in the same version of the app
- Right Scaling each piece of the application
  - Some parts of the application require more scale than others
- Platform dependencies
  - Cloud platform
  - Server platform
  - Logging solution

# The Problem

- Delivering an application into production is a non-trivial task
- lots of friction and many back-and-forth cycles between Developers and IT Operations.
- Painful deployments leads to poor quality of the delivered service
- Tendency to avoid the pain by deploying to production as seldom as possible.
- Results in larger deployments with more features being delivered at once and higher risk of things going wrong, which results in more pain, fewer deployments, … and story goes on.

# Top 10 Replies by Programmers

1. IT WORKS ON MY MACHINE!!!
2. Where were you when the program blew up?
3. Why do you want to do it that way?
4. You can't use that version on your system!
5. Somebody must have changed my code!
6. It works but it hasn't been tested.
7. I can't test everything!
8. You must have the wrong version!!!
9. It's never done that before!
10. That's weird!

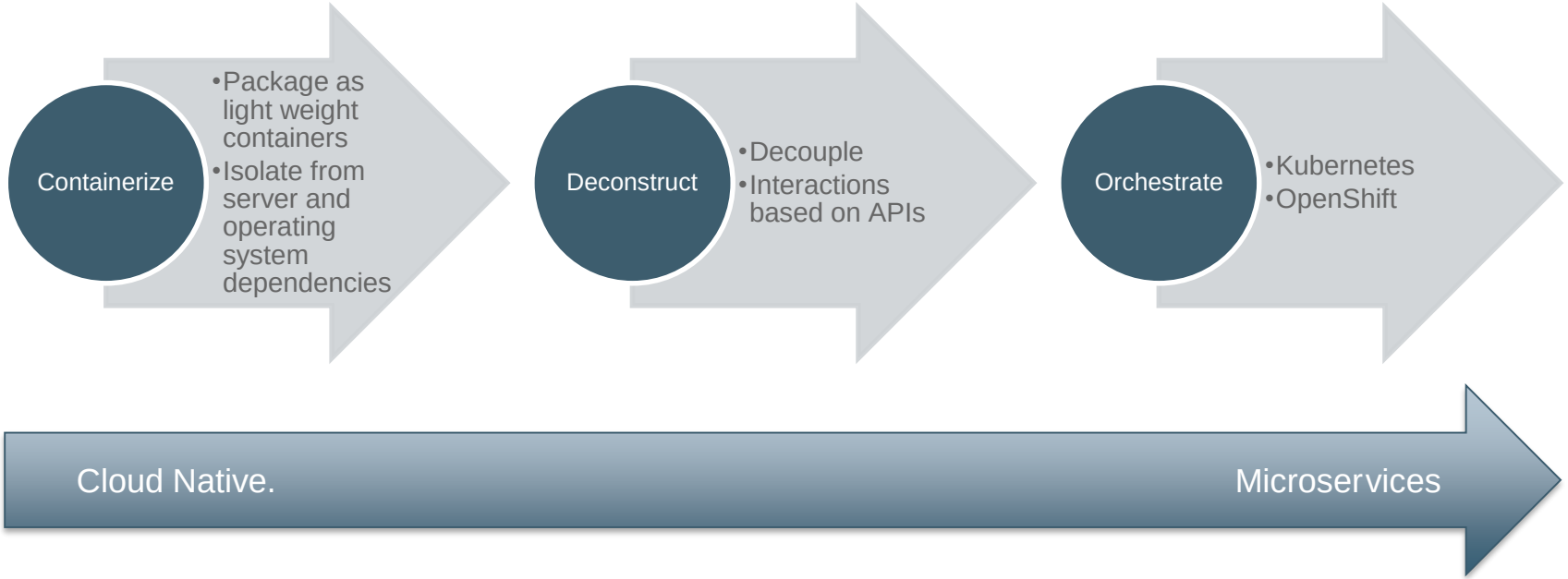
# The Solution

## Build all your Applications in a Cloud Native Way


What is Cloud Native?

According to Red Hat:
Cloud-native applications are a collection of small, independent, and loosely coupled services. They are designed to deliver well recognized business value, like the ability to rapidly incorporate user feedback for continuous improvement. In short, cloud-native app development is a way to speed up how you build new applications, optimize existing ones, and connect them all together. Its goal is to deliver apps users want at the pace a business needs.

# Modernize


Containerize
•Package as light weight containers
•Isolate from server and operating system dependencies
Deconstruct
•Decouple
•Interactions based on APIs
Orchestrate
•Kubernetes
•OpenShift
Cloud Native.
Microservices

# Why is Cloud Native Good?

- Containers give you portability. Portable apps give you flexibility in hosting
  - Isolates your development languages from the rest of the system.
  - Cloud hosting flexibility
  - On-Premises
  - Multi-Cloud
- Enforces an environment of automation
  - Repeatable deployments
  - Easy developer setup
  - Consistency between Prod and Pre-Prod environments
- Offloads logging to Container Run-Time
- Supports a Polyglot development

# Monolith vs Microservices - The Hype

**Monolith**
- Bad
- Old
- Can’t scale
- Risky deployment

**Microservices**
- Good
- Solves all Scale issues
- More Reliable
- Faster development
- Modern
- Only way to do DevOps

# Monolith vs Microservices – The Reality

## Monolith

- Best for smaller apps. Can scale quite large
- 100% support of DevOps
- Simpler than Microservices
- Can take advantage of Containers

## Microservices

- Good – for large systems
- Good for Scaling stateless services
- Increases resiliency
- Allows for more teams on a single system without violating agile principals
- Increased complexity
- Good Fit for Containers
- Can help with large data issues by distributing the data in to each service

# Twelve-Factor Apps

## History of Twelve-Factor Apps

- The **Twelve-Factor App methodology** is a methodology for building software as a service applications. These best practices are designed to enable applications to be built with portability and resilience when deployed to the web.
- The methodology was drafted by developers at Heroku and was first presented by Adam Wiggins circa 2011
- The **Twelve-Factors** provide a good set of principals to follow with developing Microservice based applications

# 12 Factor Apps

## Introduction

In the modern era, software is commonly delivered as a service: called web apps, or software-as-a-service. The twelve-factor app is a methodology for building software-as-a-service apps that:

- Use declarative formats for setup automation, to minimize time and cost for new developers joining the project
- Have a clean contract with the underlying operating system, offering maximum portability between execution environments
- Are suitable for deployment on modern cloud platforms, obviating the need for servers and systems administration
- Minimize divergence between development and production, enabling continuous deployment for maximum agility
- And can scale up without significant changes to tooling, architecture, or development practices.

The twelve-factor methodology can be applied to apps written in any programming language, and which use any combination of backing services (database, queue, memory cache, etc).

# 12 Factor Apps

## I. Codebase

- One codebase tracked in revision control, many deploys
- If there are multiple codebases, it's not an app – it's a distributed system. Each component in a distributed system is an app, and each can individually comply with twelve-factor.
- Multiple apps sharing the same code is a violation of twelve-factor. The solution here is to factor shared code into libraries which can be included through the dependency manager.
- There is only one codebase per app, but there will be many deploys of the app. A deploy is a running instance of the app. This is typically a production site, and one or more staging sites.
- The codebase is the same across all deploys, although different versions may be active in each deploy.

# 12 Factor Apps

## II. Dependencies

- **A twelve-factor app never relies on implicit existence of system-wide packages**. It declares all dependencies, completely and exactly, via a dependency declaration manifest.
- One benefit of explicit dependency declaration is that it simplifies setup for developers new to the app. The new developer can check out the app's codebase onto their development machine, requiring only the language runtime and dependency manager installed as prerequisites. Think Nuget and NPM.
- If the app needs to shell out to a system tool, that tool should be vendored into the app.

# 12 Factor Apps

## III. Config

- **Store config in the environment -** An app's config is everything that is likely to vary between deploys (staging, production, developer environments, etc). This includes:
  - Resource handles to the database, Memcached, and other backing services
  - Credentials to external services such as Azure or Twitter
  - Per-deploy values such as the canonical hostname for the deploy
- The twelve-factor app stores config in environment variables - Env vars are easy to change between deploys without changing any code; unlike config files, there is little chance of them being checked into the code repo accidentally; and unlike custom config files, or other config mechanisms, they are a language- and OS-agnostic standard.
- Sometimes apps batch config into named groups (often called "environments") named after specific deploys, such as the development, test, and production environments in Rails. This method does not scale cleanly: as more deploys of the app are created, new environment names are necessary.

# 12 Factor Apps

## IV. Backing services

- **Treat backing services as attached resources**
- A backing service is any service the app consumes over the network as part of its normal operation. I.E. Database, Storage, or Queue
- The code for a twelve-factor app makes no distinction between local and third party services.
- A deploy of the twelve-factor app should be able to swap out a local SQL database with one managed by a third party (such as Azure SQL) without any changes to the app's code.

# 12 Factor Apps

## V. Build, release, run

- **Strictly separate build, release and run  stages**
- Build – compiles the code, transpiles, compresses
- Release - The release stage takes the build produced by the build stage and combines it with the deploy's current config. The resulting release contains both the build and the config and is ready for immediate execution in the execution environment.
- Run - The run stage (also known as "runtime") runs the app in the execution environment, by launching some set of the app's processes against a selected release.

# 12 Factor Apps

## VI. Processes

- **Execute the app as one or more stateless processes**
- Do not use Sticky Sessions
- Do not do JIT compiling
- Do not depend on the local file system. If you need to persist information or files use a backing system

# 12 Factor Apps

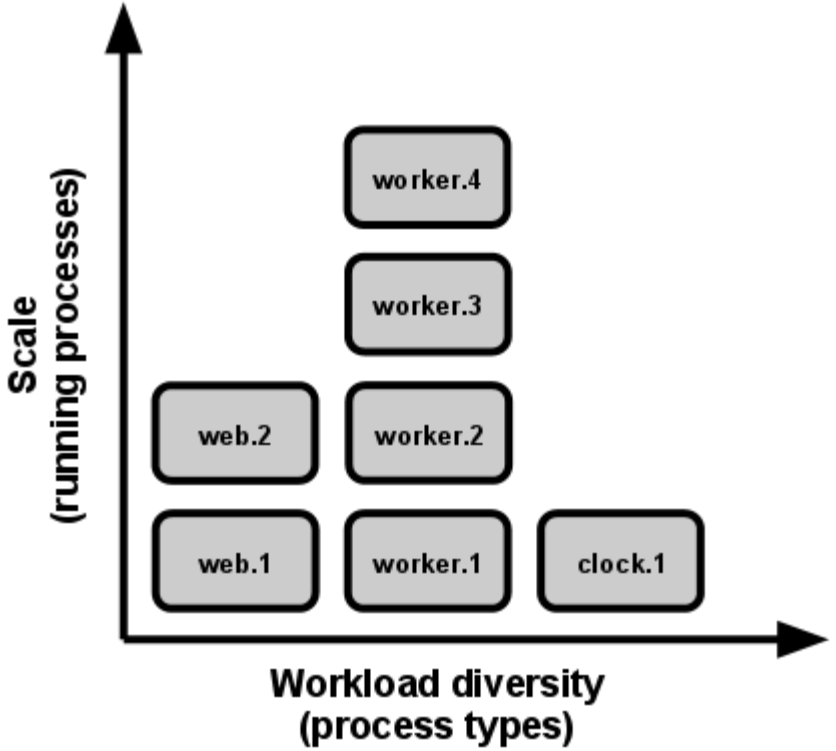
## VII. Port binding

- **Export services via port binding**
- The twelve-factor app is completely self-contained and does not rely on runtime injection of a webserver into the execution environment to create a web-facing service. The web app exports HTTP as a service by binding to a port, and listening to requests coming in on that port.
- Port-binding approach means that one app can become the backing service for another app, by providing the URL to the backing app as a resource handle in the config for the consuming app.

# 12 Factor Apps

## VIII. Concurrency

- **Scale out via the process model**
- Today the best know solution is Containers
- Stateless service model makes scaling simple



# 12 Factor Apps

## IX. Disposability

- **Maximize robustness with fast startup and graceful shutdown**
- Event driven and or web processes should complete requests before shutting down.
- Worker roles or Jobs should use queue to insure job completion. They should also be idempotent

# 12 Factor Apps

## X. Dev/Prod parity

- **Keep development, staging, and production as similar as possible**
- Make the time gap small: a developer may write code and have it deployed hours or even just minutes later.
- Make the personnel gap small: developers who wrote code are closely involved in deploying it and watching its behavior in production.
- Make the tools gap small: keep development and production as similar as possible.

# 12 Factor Apps

## XI. Logs

- **Treat logs as event streams**
- A twelve-factor app never concerns itself with routing or storage of its output stream. It should not attempt to write to or manage logfiles. Instead, **each running process writes its event stream, unbuffered, to stdout**
- Tools like Prometheus, or Azure Monitor can be used to inspect, visualize and analyze the health and operation of the over all application / system
- Using these tools allows for:
  - Finding specific events in the past.
  - Large-scale graphing of trends (such as requests per minute).
  - Active alerting according to user-defined heuristics (such as an alert when the quantity of errors per minute exceeds a certain threshold).

# 12 Factor Apps

## XII. Admin processes

- **Run admin/management tasks as one-off processes**
- One-off admin processes should be run in an identical environment as the regular long-running processes of the app. They run against a release, using the same codebase and config as any process run against that release. Admin code must ship with application code to avoid synchronization issues.


# OpenShift on Azure eShop Demo

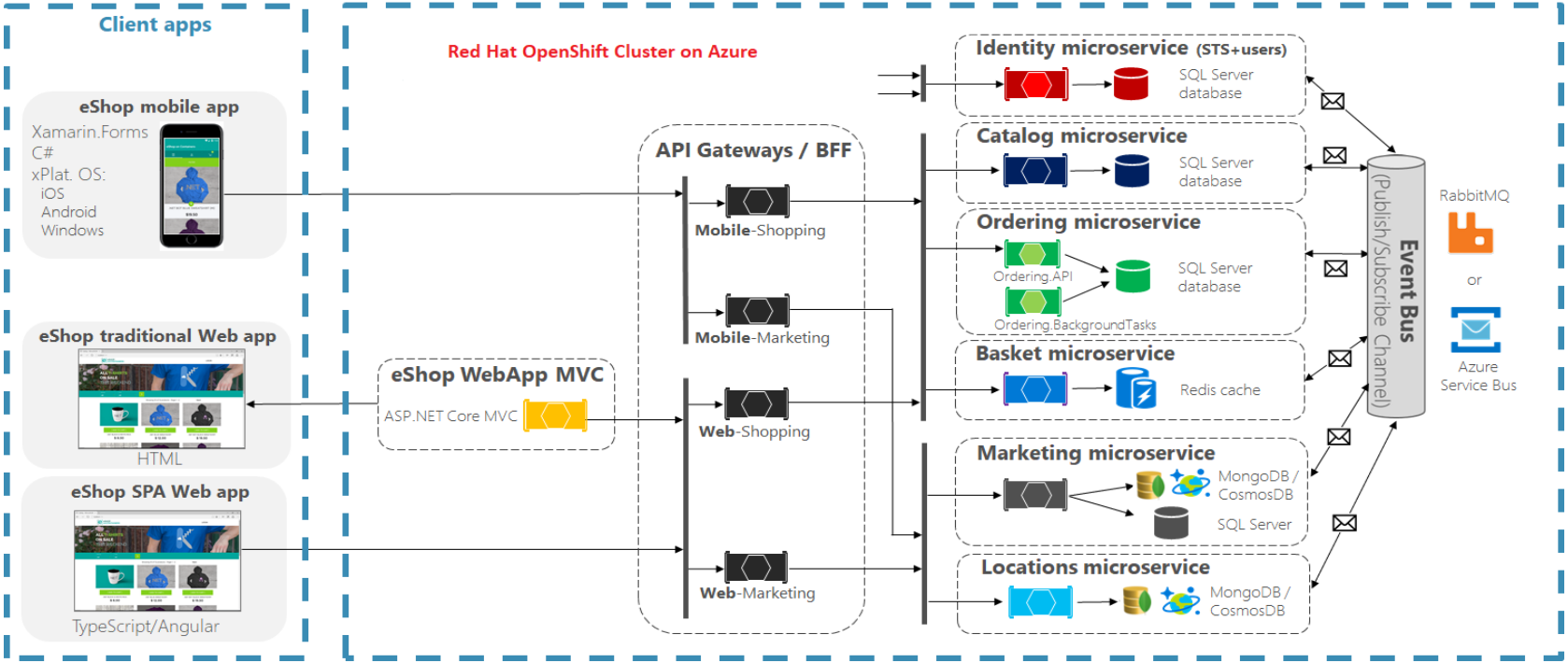
## Enterprise solution demo

eShop on Containers Microservices reference application deployed to OpenShift on Azure

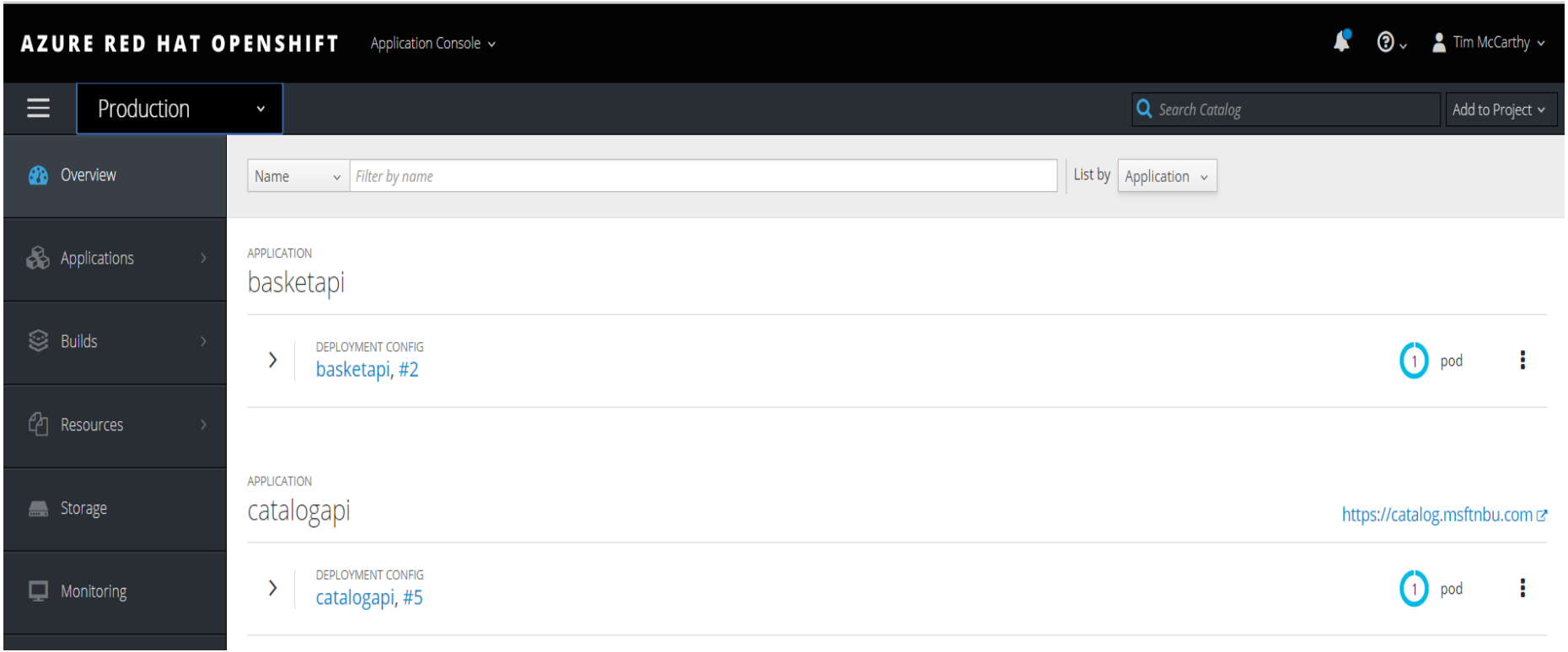
## Demo Details

- Deployed to ARO (Azure Red Hat OpenShift)
- Production grade routes, certificates and domains
- 100% Azure DevOps automation through Builds and Release Pipelines
- 19 Services / Pods – provides real world complexity
- Integration with Azure PaaS services through Open Service Broker
- Allows us to fully demonstrate the features of OpenShift

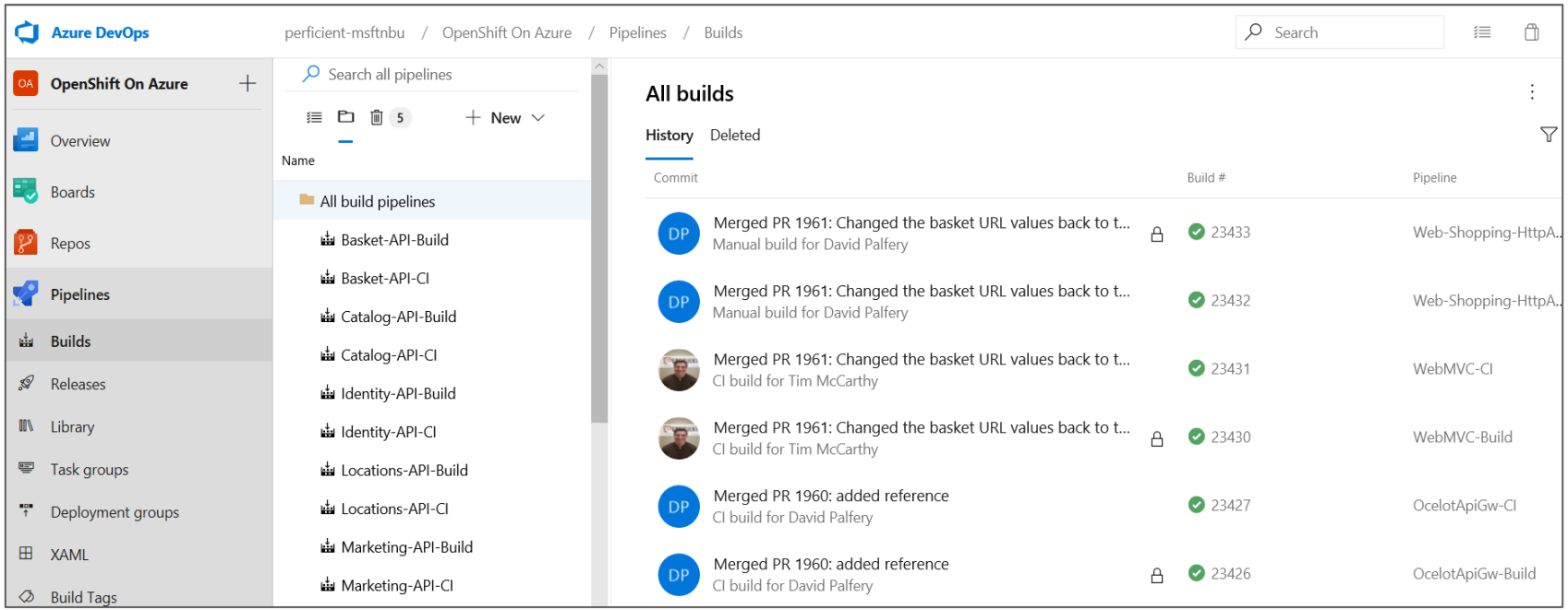
# App Architecture - eShop



# OpenShift Cluster Architecture



# Azure DevOps – CI/CD Pipelines




Development Basics

# OpenShift Core Concepts

## Overview

- **Containers** and **images** are the building blocks for deploying your applications.
- **Pods** and **services** allow for containers to communicate with each other and proxy connections.
- **Projects** and **users** provide the space and means for communities to organize and manage their content together.
- **Builds** and **image streams** allow you to build working images and react to new images.
- **Deployments** add expanded support for the software development and deployment lifecycle.
- **Routes** announce your service to the world.
- **Templates** allow for many objects to be created at once based on customized parameters.

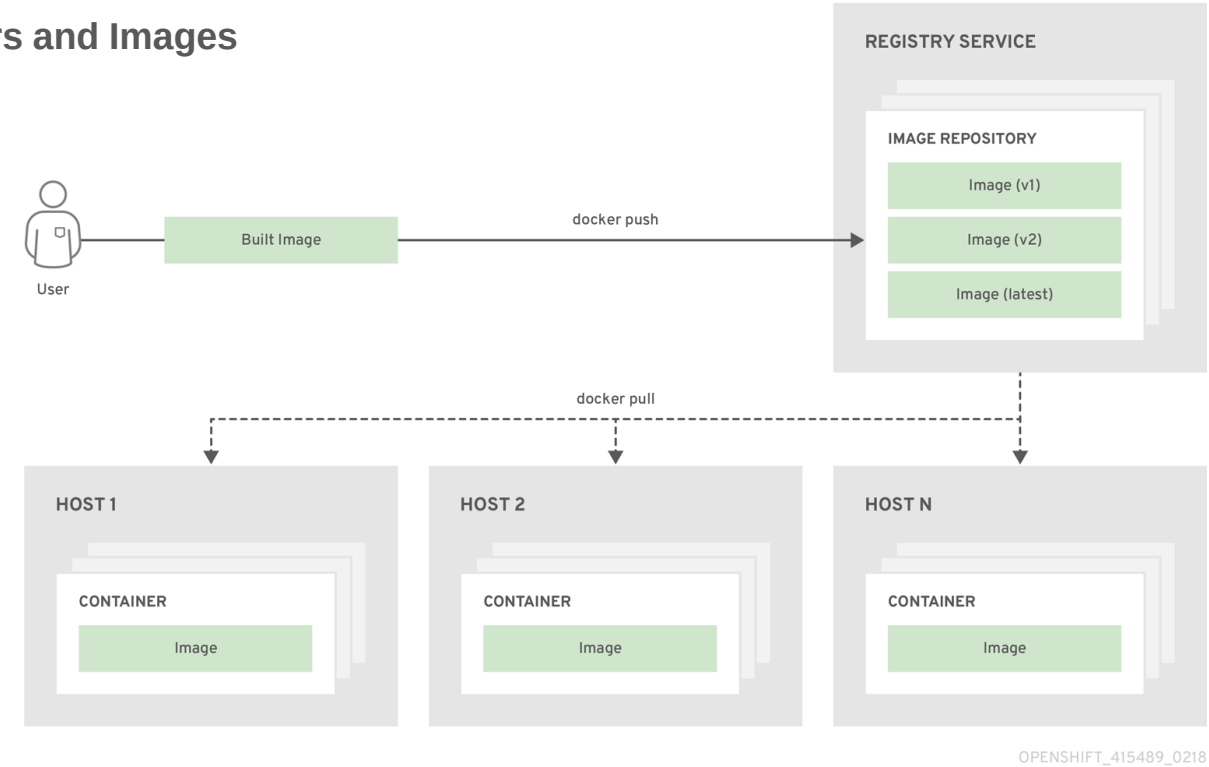
# OpenShift Core Concepts

## Containers and Images

- **Docker Containers** - Lightweight mechanisms for isolating running processes so that they are limited to interacting with only their designated resources.
- **Docker Images** - A Docker image is a binary that includes all of the requirements for running a single Docker container, as well as metadata describing its needs and capabilities.
- **Docker Registries** - A Docker registry is a service for storing and retrieving Docker images.
  - Docker image repositories - Each image repository contains one or more tagged images.
  - Docker Hub
  - Red Hat provides a Docker registry at registry.access.redhat.com

# OpenShift Core Concepts

## Containers and Images



# OpenShift Core Concepts

## Pods and Services

- **Pods** - one or more containers deployed together on one host, and the smallest compute unit that can be defined, deployed, and managed.
  - Pods have a lifecycle – Waiting, Running and Terminated
  - Pods are immutable
    - Can be changed by updating the definition and then restarting
  - Stateless – by default no information is persisted when a Pod is restarted
    - We can use Persisted Volumes to store data – more on that later
- **Services** – Serves as an internal load balancer.
  - Pods can be added and removed from a service without the consumers knowing
  - A service uses a label selector to find all the containers running that provide a certain network service on a certain port.
- **Labels -** Labels are used to organize, group, or select API objects.
  - This makes it possible for services to reference groups of pods, even treating pods with potentially different Docker containers as related entities.
  - Labels are simple key-value pairs,

# OpenShift Core Concepts

## Projects and Users

- **Users** – an OpenShift user object represents the actor which may be granted permissions in the system in order to perform any interaction
  - Regular users – most interactive OpenShift users; created automatically in the system upon first login (or via the API)
  - System users – most are created automatically when the infrastructure is defined; used mainly to enable the infrastructure to interact with the API securely
  - Service accounts – special system users associated with projects
  - *Every user must authenticate in order to access OpenShift*
- **Namespaces** – a (Kubernetes) namespace provides the mechanism to scope resources within a cluster
  - Help to avoid basic naming collisions
  - Delegate management authority to trusted users
  - Allow for the ability to limit community resource consumption
- **Projects** – a project is a Kubernetes namespace with additional annotations, becoming the central vehicle by which access to resources for regular users is managed
  - Each project scopes its own set of: *objects*, *policies*, *constraints*, and *service accounts*

# OpenShift Core Concepts

## Builds and Image Streams

- **Builds** – a process used to transform input parameters or source code into a runnable container image
  - Leverages Kubernetes architecture: creates Docker containers from build images and pushes them to an internal Docker registry
  - Extensive *build strategies*: Docker build, Source-to-Image, or custom builds
- **Image Streams** – comprised of one or more Docker image identified by *tags*, presenting a single, virtual view of related images
  - Like that of a Docker image repository

# OpenShift Core Concepts

## Deployments

- **Replication Controllers** – a core Kubernetes object responsible for ensuring…
  - The correct number of replicas desired are running
  - Securing the pod definition for the creation of a replicated pod
  - Operating a *selector* used to identify managed pods
- **Deployments** – a replication controller based on a defined template (deployment configuration)
- **Deployment Configurations** – a template consisting of…
  - A replication controller template
  - The default replica count
  - A deployment strategy (used for deployment execution)
  - A set of triggers which define how deployments are created automatically

# OpenShift Core Concepts

## Routes

- **Overview** – a route is a method to *expose* a service
  - Defined routes and identified endpoints are consumed by routers to provide external connectivity
- **Routers** – ingress points for all external traffic destined for OpenShift services
  - Host-name mapping, load balancing
- **Route Host Names –** the *edge* association used to route traffic
- **Route Types**
  - **Unsecured** – simplest way to configure routes
  - **Secured** – provide the ability use TLS & serve certificates
    - **Edge** – TLS served by front-end of router, must be configured into the route
    - **Passthrough** – no key or certificate required
    - **Re-encryption** – full path of connection is encrypted
- **Path Based Routes**

# OpenShift Core Concepts

## Templates

- **Overview** – a set of related object definitions to be created together
- **Parameters** – reference parameters to be substituted during object creation

# OpenShift Web Console

## Overview

- Web Based UI for Management of your cluster
- Useful for generating YAML files

# OpenShift Web Console

## Demo and Lab

- **Deploy .NET Core Hello World App**

# Advanced Deployment Techniques

# Advanced Deployment Techniques

## Developer CLI Operations

- **Basic CLI operations** – navigate between projects, describing & explaining OpenShift objects
- **Application Modification operations** – query for objects, annotate them, and delete them
- **Build & Deployment operations** – build management, build start & cancellation
- **Advanced operations** – ‘oc apply’, patching, ‘oc adm’

```
$ oc new-build --name=data-seeder dotnet:2.2 --binary=true
$ oc start-build data-seeder --from-dir=bin/Release/netcoreapp2.2/publish --follow
$ oc new-app data-seeder
```

```
$ oc apply -f .\mongo-deployment-config.json
```

# Advanced Deployment Techniques

## OpenShift Templates: When / How to Use Them

```
$ oc process openshift//mongodb-persistent \
    -p MONGODB_USER=forecastsuser \
    -p MONGODB_PASSWORD=forecastpassword \
    -p MONGODB_DATABASE=forecastdb \
    -p MONGODB_ADMIN_PASSWORD=forecastpassword | oc create -f -
```

# Advanced Deployment Techniques

## Demo

- **Deploy the Nginx container**
- **Deploy the front-end SampleApp (Nginx + React application)**

# Managing Secrets and Environment Variables

# Managing Secrets & Environment Variables

## Introduction

- **Challenge of multiple environments**
- **The 3rd rule of the 12 Factor Methodology**
  - "Store config in the environment" sound familiar?
- **Provisioning techniques**
- **Environment variables**
- **Passing secrets**
- **Demo** – techniques for deployment, secret reference
- **Lab** – configure environment variables & secrets, deploy SampleApp

# Managing Secrets & Environment Variables

## Managing Environment Variables

```
$ oc set env <object-selection> --list [<common-options>]
```

```
$ oc set env pod/p1 --list
```

```
$ oc set env <object-selection> KEY_1=VAL_1 ... KEY_N=VAL_N [<set-env-options>] [<common-options>]
```

```
$ oc set env dc/d1 ENV1- ENV2-
```

```
$ oc set env dc/data-seeder MongoDbSettings__HostName=mongodb.workshop-demo.svc.cluster.local
```

# Managing Secrets & Environment Variables

## Provisioning Techniques – Secrets

- Secret object type in OpenShift
- Decouple sensitive content
- Mounting secrets
- Performing actions on behalf of Pods

```
apiVersion: v1
kind: Secret
metadata:
  name: test-secret
  namespace: my-namespace
type: Opaque (1)
data: (2)
  username: dmFsdWUtMQ0K (3)
  password: dmFsdWUtMg0KDQo=
stringData: (4)
  hostname: myapp.mydomain.com (5)
```

```
# Single Key Secret
$ oc create secret generic my-single-secret --from-literal=key1=supersecret

# Multiple Key Secrets
$ oc create secret generic my-multiple-secret \
    --from-literal=key1=supersecret \
    --from-literal=key2=topsecret
```

# Managing Secrets & Environment Variables

## Provisioning Techniques – ConfigMaps

- The ConfigMap object
- Decouple configuration artifacts
- Configuration injection & agnosticism

```
kind: ConfigMap
apiVersion: v1
metadata:
  creationTimestamp: 2016-02-18T19:14:38Z
  name: example-config
  namespace: default
data: 1
  example.property.1: hello
  example.property.2: world
  example.property.file: |-
    property.1=value-1
    property.2=value-2
    property.3=value-3
```

```
$ oc create configmap <configmap_name> [options]
```

# Managing Secret & Environment Variables

**Consuming secrets as environment variables**

- **Important for your apps!**
- **All config values are ultimately environment variables**
- **Typically at the DeploymentConfig level**

```
$ oc set env dc/workshop-api --from secret/mongodb
```

# Managing Secret & Environment Variables

## Demo & Lab

**Creating Environment Variables and Secrets**

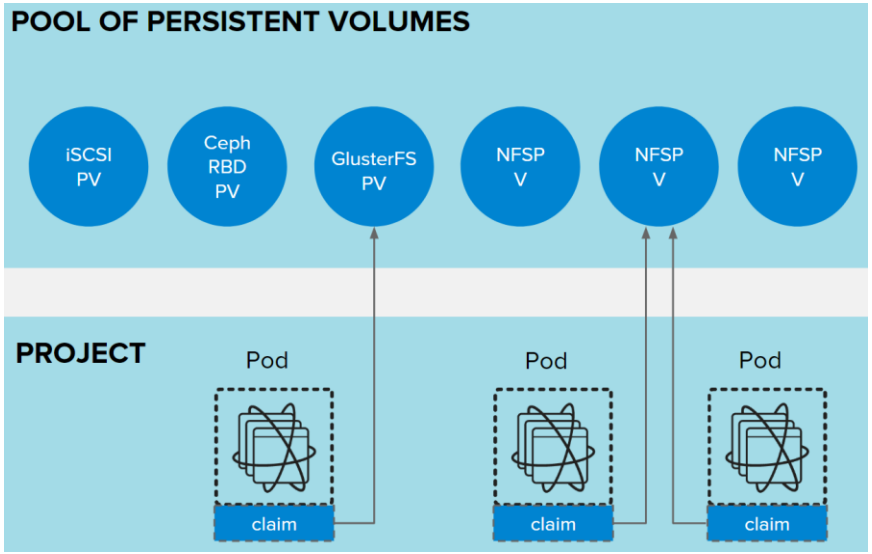

Persistent Storage

# Persistent Storage

## Overview

- **Kubernetes**
  - Persistent Volumes
  - Persistent Volume Claims
- **Adding storage to a container**
- **Demo** – Creating a persistent claim, adding it to a MongoDB Server container
- **Lab** – Deploy SampleApp's data tier to a MongoDB container

# An Abstraction




- **Persistent Volumes (PVs)**
- **Provisioned by an administrator**
  - Statically or dynamically
- **Roles** – admins describe; users request
- **Storage assignment**
  - Requested size, access mode, labels, & type

# Persistent Storage

## Persistent Volume Specificities

- Each PV contains a *spec* and a *status*

```yaml
apiVersion: v1
kind: PersistentVolume
metadata:
  name: pv0001 (1)
spec:
  capacity:
    storage: 5Gi (2)
  accessModes:
    - ReadWriteOnce (3)
  persistentVolumeReclaimPolicy: Retain (4)
  ...
status:
  ...
```

# Persistent Storage

## Persistent Volume Claims Specificities

- Like PVs, each PVC also contains a spec and status

```
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: myclaim 1
spec:
  accessModes:
    - ReadWriteOnce 2
  resources:
    requests:
      storage: 8Gi 3
  storageClassName: gold 4
status:
  ...
```

# Persistent Storage

## Adding Storage to a Container

- Mounting volumes to the host and into the Pod
- The ‘StorageClass’ *gotcha*

```
kind: Pod
apiVersion: v1
metadata:
  name: mypod
spec:
  containers:
    - name: myfrontend
      image: dockerfile/nginx
      volumeMounts:
      - mountPath: "/var/www/html"  (1)
        name: mypd  (2)
  volumes:
    - name: mypd
      persistentVolumeClaim:
        claimName: myclaim  (3)
```

# Persistent Storage

## Expanding PVCs with a File System

- **A two-step process:**
  - Expand volume objects in the cloud provider
  - Expand the filesystem on the actual node
- **The 'allowVolumeExpansion' *gotcha***
- **Recovering from failures**

```yaml
kind: PersistentVolumeClaim
apiVersion: v1
metadata:
  name: ebs
spec:
  storageClass: "storageClassWithFlagSet"
  accessModes:
    - ReadWriteOnce
  resources:
    requests:
      storage: 8Gi (1)
```

```
$ oc describe pvc <pvc_name>
```

# Persistent Storage

## Demo & Lab

- **Create a persistent claim** – then, add it to a MongoDB Server container application
- **Deploy the SampleApp** – then, connect its data tier (MongoDB container)


Services and Routes

# Services & Routes

## Overview

- **Kubernetes tie-in**
  - Pod inter-communication (Internal Network)
  - Multiple pods; multiple services
- ***A hypothetical example…***
- **Exposing an *app* to the Internet (External web)**
- **Demo** – Creating a persistent claim, adding it to a MongoDB Server container
- **Lab** – Deploy SampleApp's data tier to a MongoDB container

# Services & Routes

## Pod Inter-communication

- **A quick re-cap on Pods**
- **Public routes** – exposing an app to an external network
- **Kubernetes services as internal load-balancers**
- ***Backing* Pods**

```
apiVersion: v1
kind: Pod
metadata:
  annotations: { ... }
  labels:                                    1
    deployment: docker-registry-1
    deploymentconfig: docker-registry
    docker-registry: default
  generateName: docker-registry-1-           2
spec:
  containers:                                3
  - env:                                     4
    - name: OPENSHIFT_CA_DATA
      value: ...
    - name: OPENSHIFT_CERT_DATA
      value: ...
    - name: OPENSHIFT_INSECURE
      value: "false"
    - name: OPENSHIFT_KEY_DATA
```

# Services & Routes

## Expose an app to the External Web

- Services are REST objects
  - The internal *Docker-registry*
- Created automagically with “oc new-app” cli command
- Creating a project & service
- Expose a service to create a route

```
apiVersion: v1
kind: Service
metadata:
  name: docker-registry          (1)
spec:
  selector:                      (2)
    docker-registry: default
  clusterIP: 172.30.136.123      (3)
  ports:
  - nodePort: 0
    port: 5000                   (4)
    protocol: TCP
    targetPort: 5000             (5)
```

# Services & Routes

## Pod Deployment Tidbit

- The internal *Docker-registry*
- ‘openshift/origin-docker-registry’

```
      - name: OPENSHIFT_KEY_DATA
        value: ...
      - name: OPENSHIFT_MASTER
        value: https://master.example.com:8443
      image: openshift/origin-docker-registry:v0.6.
2 5
      imagePullPolicy: IfNotPresent
      name: registry
      ports:                                        6
      - containerPort: 5000
        protocol: TCP
      resources: {}
      securityContext: { ... }                      7
      volumeMounts:                                 8
      - mountPath: /registry
        name: registry-storage
      - mountPath: /var/run/secrets/kubernetes.io/s
erviceaccount
```

# Services & Routes

## Demo & Lab

- **Services** – test service connectivity from an app's terminal window (to another app)
- **Create a public route** – walkthrough & steps
- **SampleApp services & routes** – deploy services and routes so that the SampleApp can talk across tiers (data, front-end, .NET API)


Azure DevOps

# Azure DevOps

## Overview

- **Repeatable deployments** – deploying from Azure DevOps (or any ALM platform)
- **Connecting to OpenShift**
  - eShop build & release pipelines

# Azure DevOps

## Deploy from Azure DevOps

- **Azure Pipelines**
  - Builds
  - Releases
- **Builds**
  - Agent queue
  - Task Groups
  - Build patterns (restore, build, publish)
  - OpenShift CLI integration

# Azure DevOps

## Setting up the OpenShift environment

- **Hand-off from Azure DevOps to OpenShift**
  - OpenShift CLI commands
- **Release**
  - Docker image creation
  - Application deployment
  - Service creation
  - Route creation
  - YAML for CI / CD

# Azure DevOps

## Demo

- **Automated builds from Azure DevOps**
- **Entire build and release pipeline**

# Advanced Techniques and Gotcha’s

Tim McCarthy
Senior Solutions Architect, Microsoft Cloud Solutions
tim.mccarthy@perficient.com

Leroy Baynum
Senior Solutions Architect, Microsoft Cloud Solutions
leroy.baynum@perficient.com

Steve Holstad
Azure Practice Director
steve.holstad@perficient.com

Adam Ward
Portfolio Specialist – Microsoft
Adam.Ward@perficient.com

Otto Konopa
Portfolio Specialist – Emerging Solutions
otto.konopa@perficient.com

# Questions?