DEVOPS

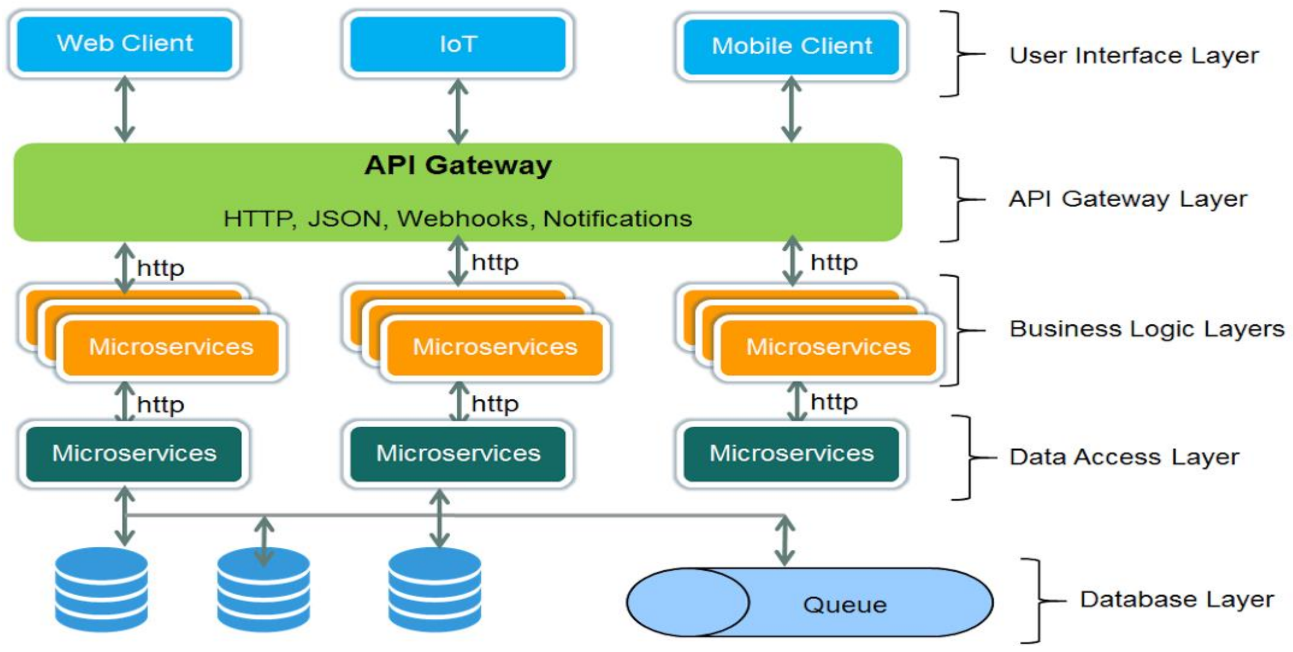
Training Material

INTRODUCTION TO
CLOUD NATIVE APPLICATION

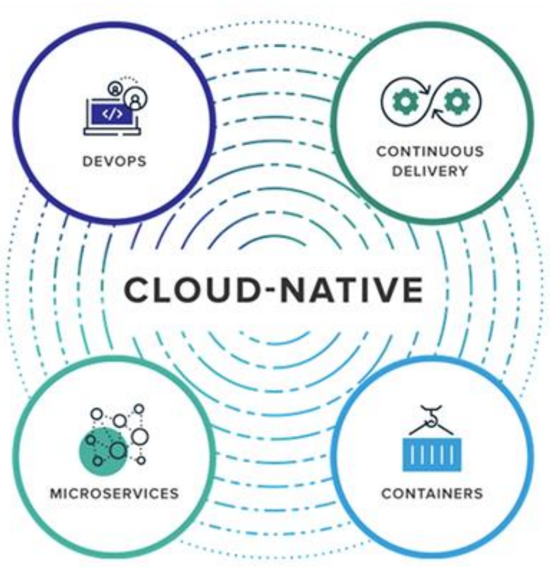
INTRODUCTION TO CLOUD-NATIVE APPLICATIONS

Cloud-Native Applications:

- **Cloud-native applications** are software applications designed and built specifically to run in cloud environments, taking full advantage of cloud computing's flexibility, scalability, and other features.
- These applications are typically **microservices-based**, highly **scalable**, and **resilient**, and they are optimized for deployment on cloud infrastructure (public, private, or hybrid).



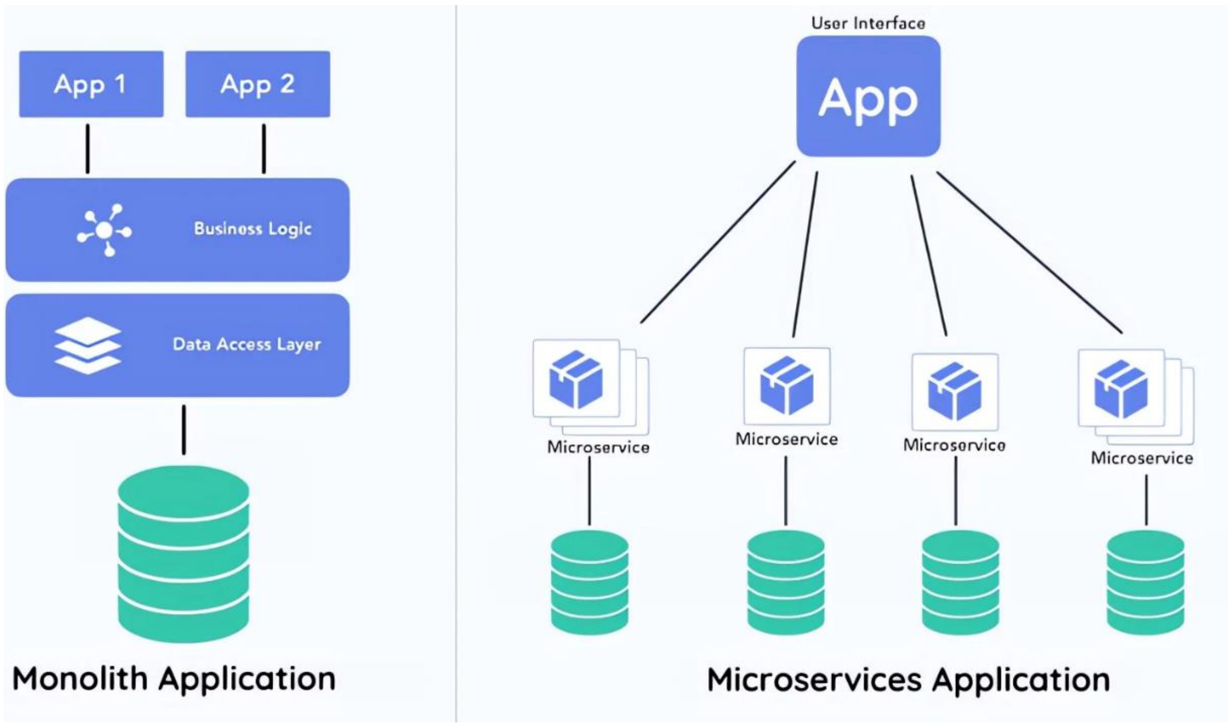
Four Pillers of Cloud Nativ Development:



1. <u>MICROSERVICES</u>:

Microservices break down an application into smaller, independent services that each perform a specific function. Each service can be developed, deployed, and scaled independently.

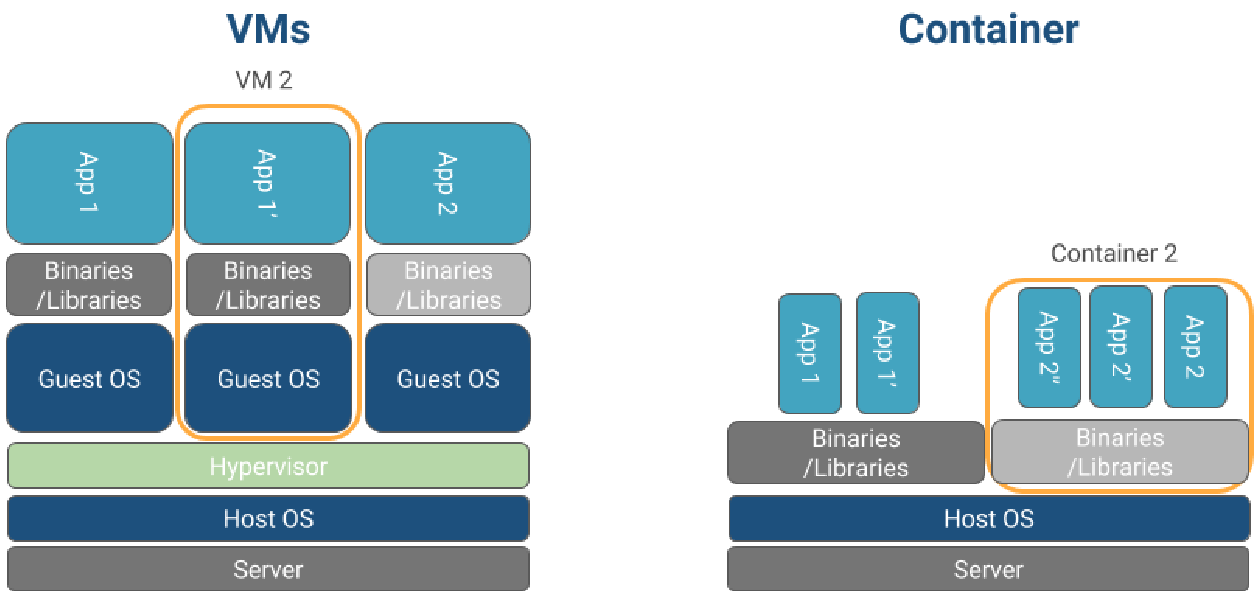
Example: Imagine an e-commerce app. Instead of having one big application for everything (product search, user authentication, payment processing), it is divided into microservices like one for searching products, another for handling payments, and a separate one for user authentication. These services can be deployed and updated independently, making the system more flexible.



2. <u>CONTAINERS</u>:

Containers package the code and all its dependencies into a lightweight, portable unit. This allows the application to run consistently across different environments, such as a developer's local machine, testing, and production systems.

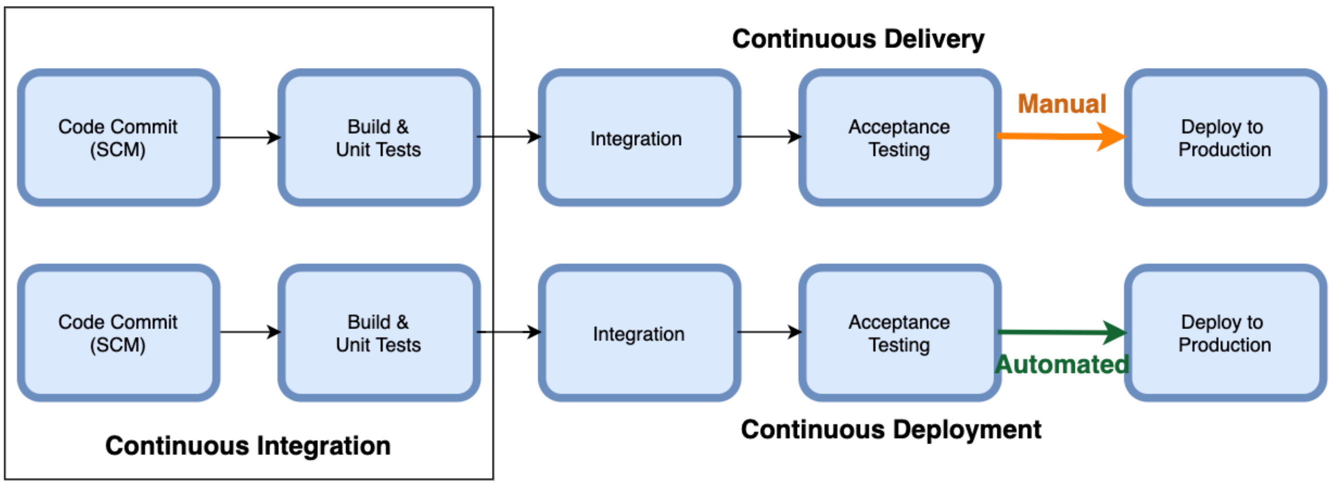
Example: If you're running a web app, you can use Docker to containerize your app. This means the same app can run the same way on your laptop, in a test environment, or in the cloud without issues. It isolates the app from the host system, ensuring it works in any environment.



3. **<u>CONTINUOUS DELIVERY/CONTINUOUS INTEGRATION (CD/CI)</u>:**

CI/CD is the practice of automatically testing and deploying changes to the application. With CI, developers integrate their changes frequently, and with CD, those changes are automatically deployed to production or a staging environment.

Example: When developers push updates to a GitHub repository, a CI tool like Jenkins automatically runs tests to check for errors. If the tests pass, the changes are deployed to a staging or production environment without manual intervention.



4. DEVOPS CULTURE:

DevOps is the practice of unifying development and operations teams to work together to automate and streamline processes. The goal is to reduce the time between writing code and deploying it to production while ensuring high quality.

Example: In a DevOps environment, developers and operations teams work closely to ensure the application is easy to deploy and monitor. For example, if an application is experiencing high traffic, the operations team can scale it up by adding more containers, and the developers can push code updates without disrupting service.

Cloud-Native vs. Traditional Applications

Aspect	Cloud-Native Applications	Traditional Applications
Architecture	Built with microservices (small, independent services)	Monolithic (single, large, interconnected application)
Deployment	Deployed in containers, easy to scale and update	Deployed on physical servers or virtual machines, harder to scale
Flexibility	Highly flexible, can be changed and updated easily	Less flexible, requires downtime for updates or changes
Scalability	Easily scalable, can add/remove resources automatically (e.g., cloud)	Scaling is manual, requires additional hardware or virtual machines
Resilience	Designed for failure; self-healing and can recover from issues automatically	Susceptible to failures; requires manual intervention to fix issues
Development Speed	Faster development due to smaller teams working on independent services	Slower development, as changes affect the entire application
Infrastructure	Runs on cloud platforms, utilizing cloud services (e.g., AWS, Azure)	Runs on on-premises servers or static environments

Aspect	Cloud-Native Applications	Traditional Applications
Maintenance	Continuous monitoring and updates, automated testing, and deployment	Requires manual updates and maintenance
Example	E-commerce app with separate services for payment, search, and orders	Classic monolithic e-commerce app with everything in one codebase

Real-World Examples of Cloud-Native Applications:

Netflix as a Cloud-Native Application

Scenario:

Netflix is one of the most popular video streaming platforms in the world, serving millions of users daily across various devices. To maintain a seamless user experience and handle massive traffic spikes, Netflix adopted **cloud-native architecture**. This approach allows Netflix to deliver content efficiently, scale rapidly, and recover from failures without impacting users.

The Problem Netflix Faced

1. **Scalability Issues**:
   - As Netflix grew in popularity, its monolithic architecture struggled to handle massive and unpredictable traffic spikes, such as during the release of a new show or movie.
   - Scaling their monolithic system was time-consuming and required significant manual intervention.
2. **System Downtime**:
   - Any failure in the monolithic architecture could lead to the entire platform being unavailable, causing interruptions in service for millions of users.
3. **Development Bottlenecks**:
   - A single codebase made it difficult for multiple teams to work on different features simultaneously, slowing down the release of new updates and features.
4. **Global Reach Challenges**:
   - Delivering high-quality video content to users worldwide required managing data centers across multiple regions and ensuring minimal latency.

The Cloud-Native Solution

Netflix adopted cloud-native principles to solve these challenges. Here's how:

1. **Microservices Architecture**:
   - Netflix broke its monolithic application into smaller, independent microservices, each responsible for a specific function, such as user authentication, recommendation engine, and video streaming.
   - **Result**: Each microservice could be developed, deployed, and scaled independently, allowing faster feature releases and updates.
2. **Containers and Kubernetes**:
   - Netflix packaged its microservices into containers using Docker. These containers are orchestrated using **Kubernetes**, which manages the deployment, scaling, and resilience of services automatically.
   - **Result**: Containers ensured consistency across development and production environments,

while Kubernetes handled traffic spikes by scaling services up or down as needed.

3. **Elastic Cloud Infrastructure**:
    - Netflix migrated its infrastructure to **AWS (Amazon Web Services)**. They use auto-scaling groups to dynamically adjust resources based on user demand.
    - **Result**: During high-demand events like new show releases, Netflix seamlessly scales up its resources to maintain service quality.
4. **Resilience Through Chaos Engineering**:
    - Netflix developed a tool called **Chaos Monkey**, which deliberately causes failures in their infrastructure to test the system's resilience.
    - **Result**: The system is built to recover automatically from failures, ensuring high availability and uninterrupted service.
5. **Global Content Delivery**:
    - Netflix uses **CDNs (Content Delivery Networks)** to cache and deliver content closer to users' locations.
    - **Result**: Reduced latency and improved streaming quality for users worldwide.

Outcome: Problem Solved

1. **High Scalability**:
    - Netflix can now handle millions of simultaneous users without interruptions. For example, during peak times like the release of "Stranger Things," Netflix scales up automatically to handle the surge.
2. **Increased Availability**:
    - Even if one microservice fails, the rest of the system continues functioning, ensuring uninterrupted service.
3. **Faster Development**:
    - Independent teams work on different microservices, allowing Netflix to introduce new features and improvements rapidly.
4. **Global Reach**:
    - Users across the world enjoy smooth streaming, thanks to CDNs and cloud infrastructure that minimize latency.

Design strategies of IBM Cloud Native Application

- The **IBM CLOUD CONTAINER REGISTRY** is a secure, private repository to store and manage Docker images.
- It plays a key role in designing and building cloud-native applications by providing a centralized platform for container image management.

Overview of IBM Cloud Container Registry

- It is a **Docker-compatible registry** that integrates seamlessly with Kubernetes and other IBM Cloud services.
- Provides **image vulnerability scanning** to enhance security.
- Facilitates version control and allows sharing of container images across teams.

Steps to Use IBM Cloud Container Registry

Step 1: Set Up the IBM Cloud CLI and Plug-ins

1. Install the IBM Cloud CLI on your system.
2. Add the **Container Registry CLI plug-in** by running:

```
ibmcloud plugin install container-registry
```

3. Log in to your IBM Cloud account:

```
ibmcloud login
```

Step 2: Create a Namespace

A **namespace** is a logical grouping for your images. Create one using:

```
ibmcloud cr namespace-add <your-namespace>
```

Example:

```
ibmcloud cr namespace-add my-cloud-native-app
```

Step 3: Build a Docker Image

Write a Dockerfile for your microservice. Example for a Node.js app:

```
FROM node:16-alpine

WORKDIR /app
```

```
COPY package*.json ./

RUN npm install

COPY . .

EXPOSE 3000

CMD ["node", "app.js"]
```

Build the Docker image locally:

```
docker build -t <your-namespace>/<image-name>:<tag> .
```

Example:

```
docker build -t my-cloud-native-app/backend-service:v1 .
```

Step 4: Push the Image to IBM Cloud Container Registry

Tag the image for the registry:

```
docker tag <local-image> <region>.icr.io/<namespace>/<image-name>:<tag>
```

Example:

```
docker tag my-cloud-native-app/backend-service:v1 us.icr.io/my-cloud-native-app/backend-service:v1
```

Push the image to the registry:

```
docker push <region>.icr.io/<namespace>/<image-name>:<tag>
```

Example:

```
docker push us.icr.io/my-cloud-native-app/backend-service:v1
```

Step 5: Use Images in Kubernetes Deployments

In your Kubernetes manifest (e.g., deployment.yaml), specify the image from the registry:

```
apiVersion: apps/v1

kind: Deployment

metadata:

  name: backend-service

spec:

  replicas: 3

  selector:
    matchLabels:
      app: backend-service
  template:
    metadata:
      labels:
        app: backend-service
    spec:
      containers:
      - name: backend-service
        image: us.icr.io/my-cloud-native-app/backend-service:v1
        ports:
        - containerPort: 3000
```

Deploy to an IBM Cloud Kubernetes Service cluster:

```
kubectl apply -f deployment.yaml
```

Step 6: Enable Vulnerability Scanning

Scan your image for vulnerabilities to ensure it's secure:

```
ibmcloud cr va <region>.icr.io/<namespace>/<image-name>:<tag>
```

View results to address any issues:

```
ibmcloud cr va-report <image-id>
```

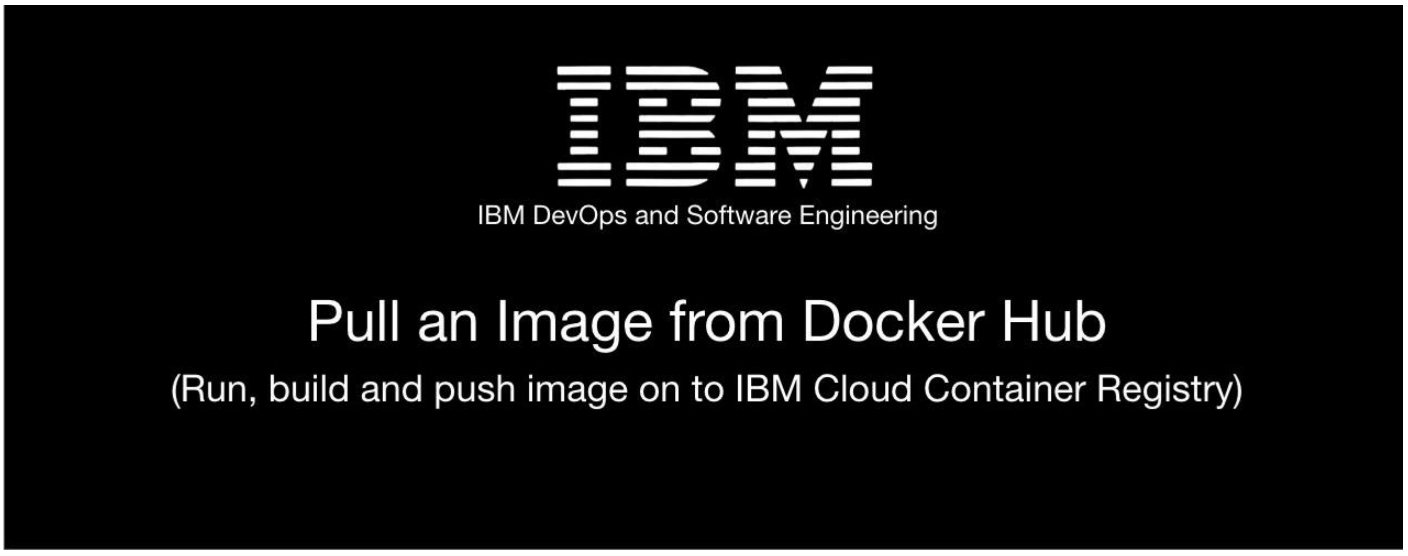
Step 7: Automate CI/CD Pipeline

Integrate IBM Container Registry with IBM Continuous Delivery to automate builds and deployments:

- **Build Stage**: Use tools like Jenkins or Tekton Pipelines to automate Docker image builds.
- **Push Stage**: Automate pushing images to the registry.
- **Deploy Stage**: Automate deploying images to Kubernetes clusters.

Benefits of Using IBM Cloud Container Registry

1. **Security**:
   - Automatically scans images for vulnerabilities.
   - Stores images in a secure environment with encryption.
2. **Region-Specific Repositories**:
   - Store images closer to your deployments to reduce latency.
   - Regions include us.icr.io, eu.icr.io, and more.
3. **Integration**:
   - Seamlessly integrates with IBM Kubernetes Service, OpenShift, and CI/CD tools.
4. **Team Collaboration**:
   - Share images across teams with controlled access using IAM policies.
5. **Scalability**:
   - Supports storing and managing multiple images for microservices-based applications.


IBM

IBM DevOps and Software Engineering

Pull an Image from Docker Hub

(Run, build and push image on to IBM Cloud Container Registry)