

Cloud-Native: Microservices, Kubernetes, Service Mesh, CI/CD

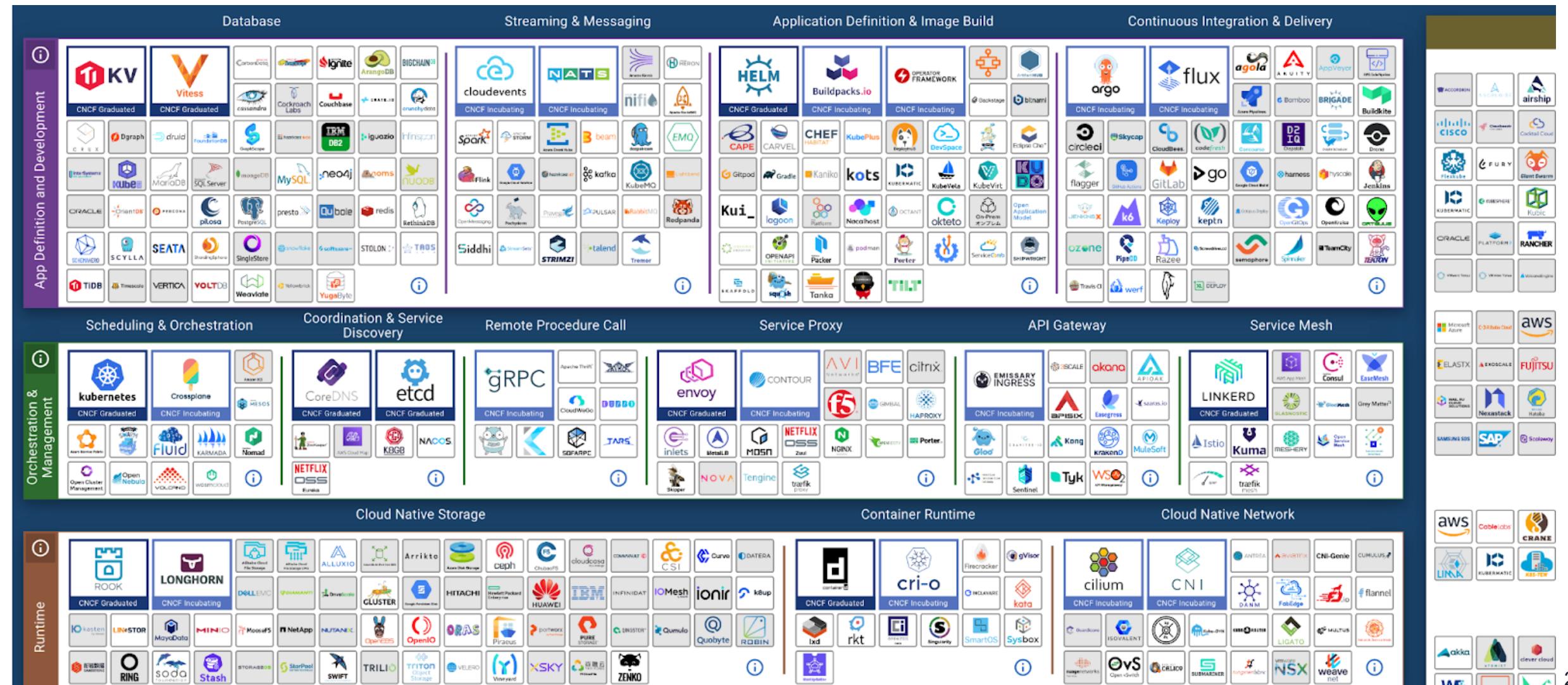
Design and Build Cloud Applications with Microservices Architecture, Kubernetes Deployments, Communications (Services Mesh), Backing Services (Databases, Caches and Message brokers), CI/CD pipelines and Monitoring & Observability with Patterns and Best Practices.



CNCF Landscape



CLOUD NATIVE COMPUTING FOUNDATION



CNCF Trial Map



CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape <https://github.com/cncf/landscape> has a growing number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator <https://www.cncf.io/training>

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider <http://cncf.io/kcsp>

C. Join CNCF's End User Community

For companies that don't offer cloud native services externally <http://cncf.io/enduser>

WHAT IS CLOUD NATIVE?

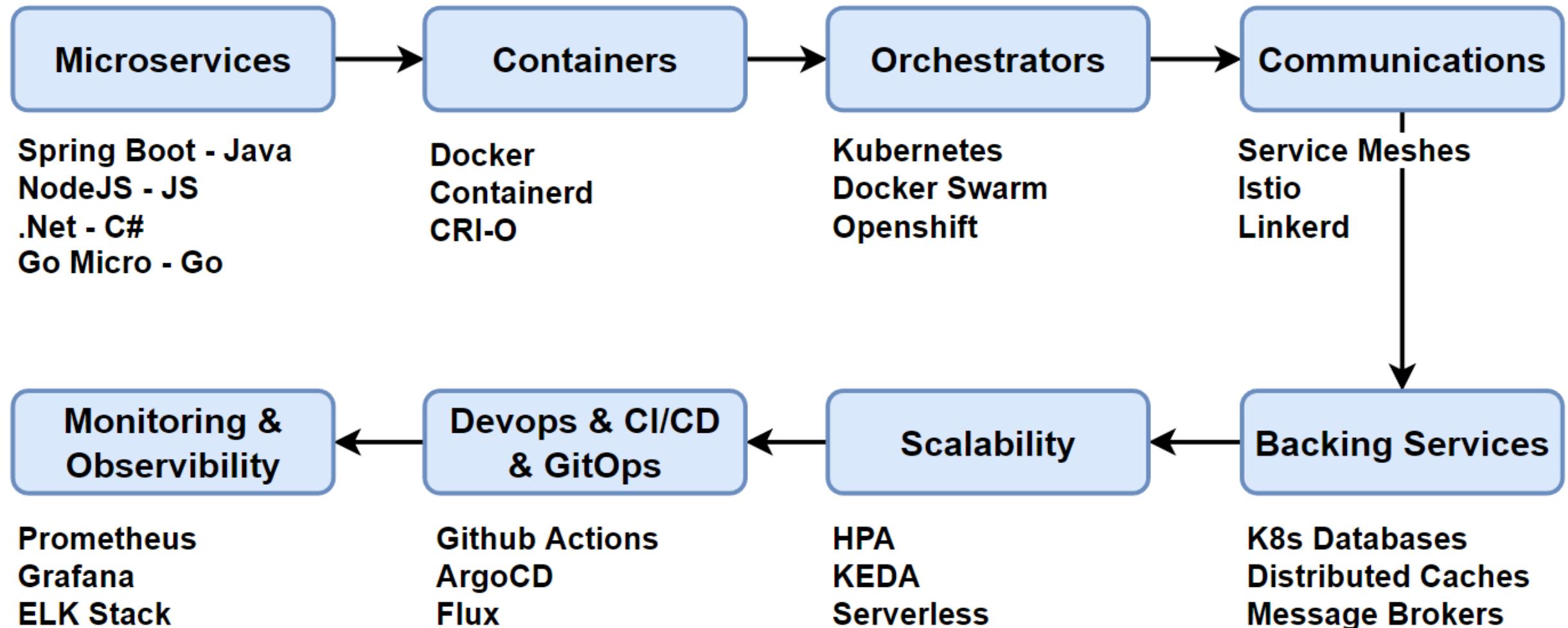
- **Operability:** Expose control of application/system lifecycle.
- **Observability:** Provide meaningful signals for observing state, health, and performance.
- **Elasticity:** Grow and shrink to fit in available resources and to meet fluctuating demand.
- **Resilience:** Fast automatic recovery from failures.
- **Agility:** Fast deployment, iteration, and reconfiguration.

www.cncf.io
info@cncf.io

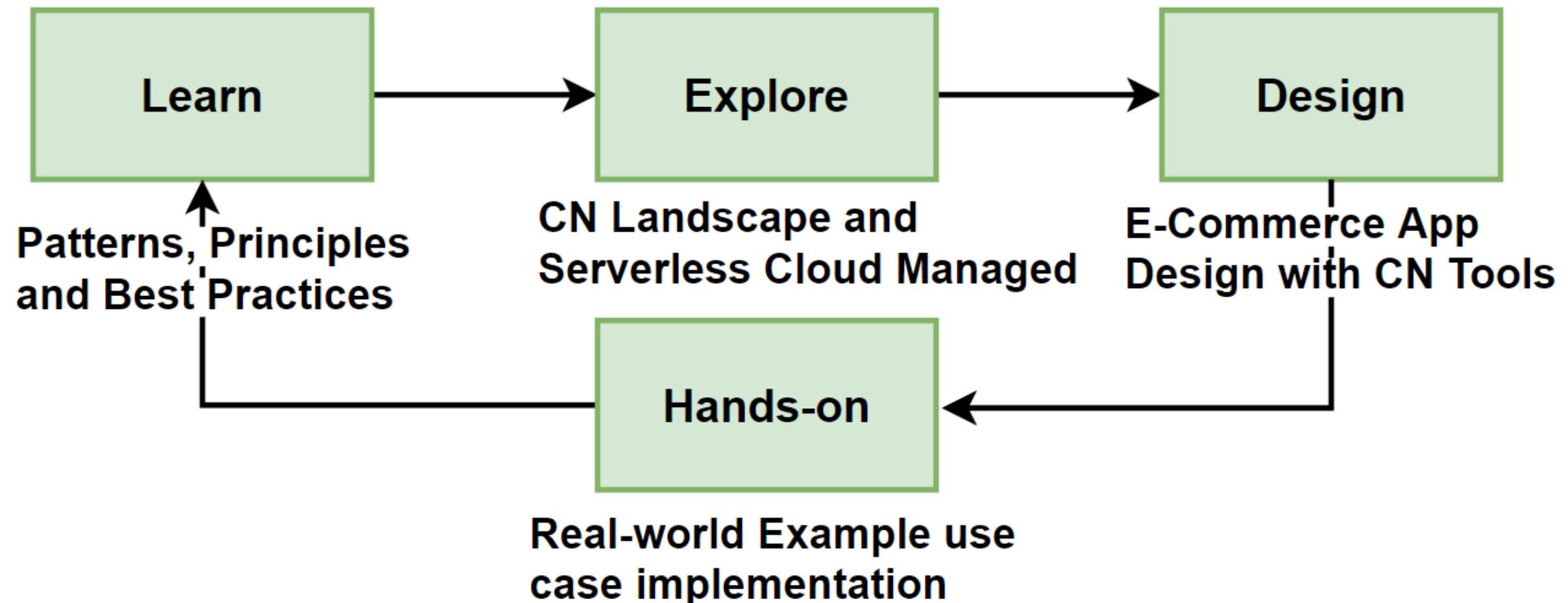


<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-trail-map/>

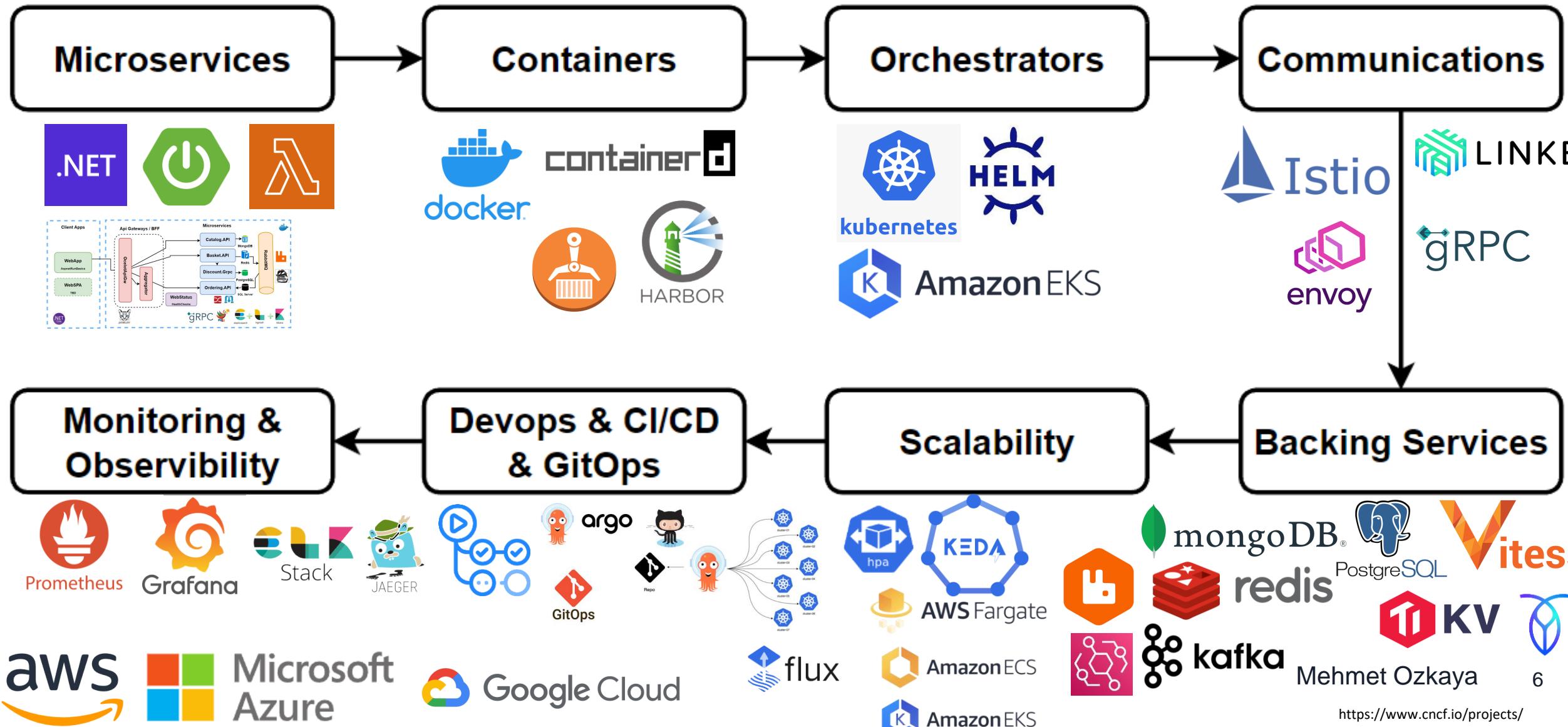
Cloud-Native Pillars Map – The Course Map



Way of Learning – The Course Flow



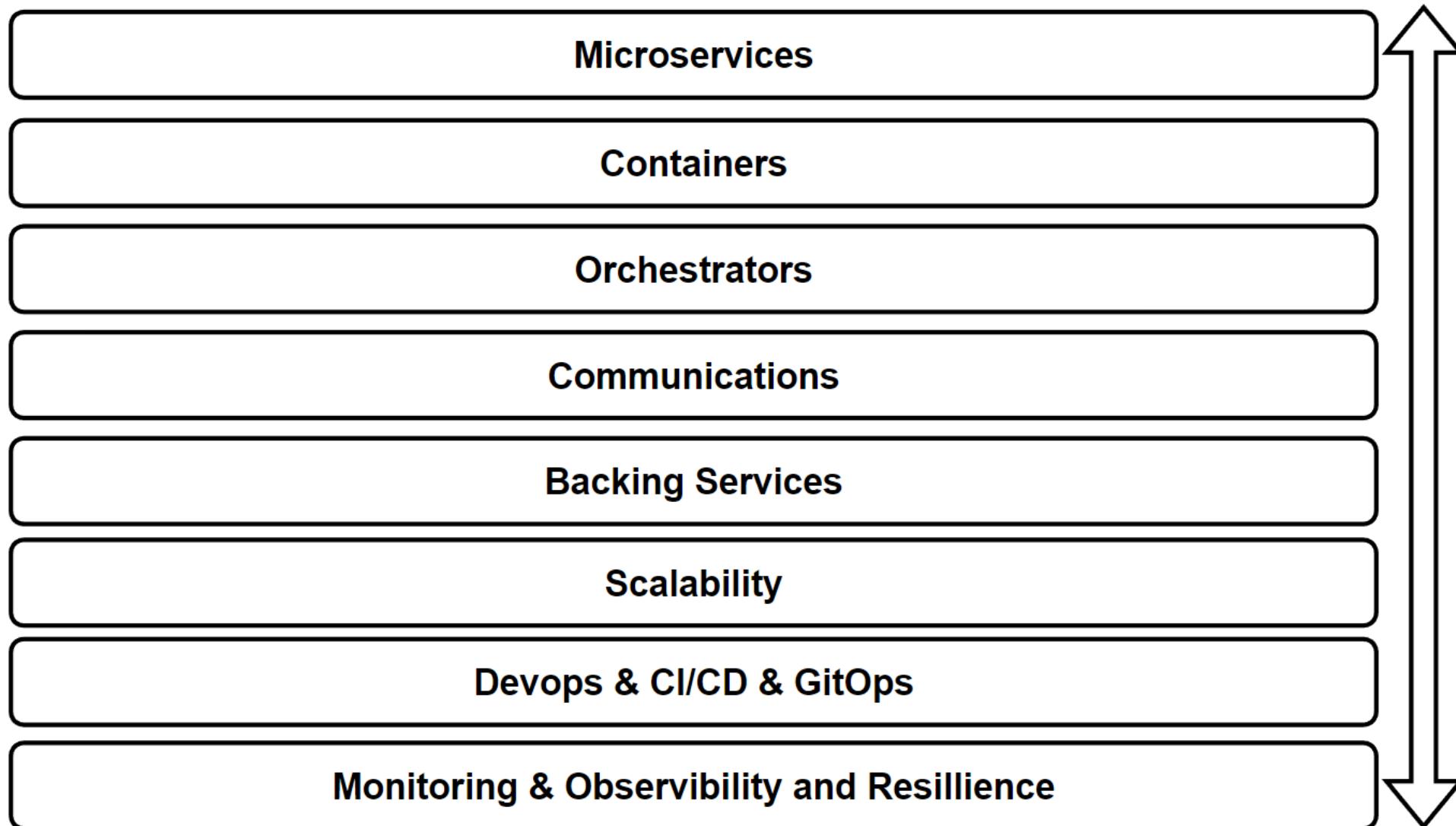
Cloud-Native Tools and Technologies that we will practice



Course Target

- To be a **decision-maker** as a **software developer/architect** in **cloud-native architecture boards**.
- Designed for **software developers and architects**
- **Hands-on Design and Development Activities**
- Apply **best practices** with **cloud-native microservices design patterns and principles**
- **Explore and Practice** cloud-native tools, understand **when** and **where to use** these tools
- Prepare for **Software Architecture Interviews**
- Prepare for **System Design Architecture Interview exams**
- **Solid understanding** of the **cloud-native ecosystem** and will be ready to **design, build, and deploy** your own **cloud-native applications**.

Cloud-Native Pillars



Cloud Types: Private/On-premises, Public and Hybrid Cloud

Private/On-premises Cloud

- Owned and operated by a single organization
- Deployed within the organization's data center
- Increased privacy, security, and control
- Best for strict compliance, security, and data privacy requirements

Public Cloud

- Provided by third-party service providers (AWS, Azure, Google Cloud)
- Shared computing resources among multiple users
- Cost-effective, scalable, and flexible
- Providers manage infrastructure, maintenance, and updates

Hybrid Cloud

- Combination of private and public cloud infrastructures
- Leverages the benefits of both cloud types
- Enables workload movement between environments for efficiency and cost optimization



What is Multi-Cloud Strategy ?

- Organization uses multiple cloud service providers to **meet its computing needs.**
- Distribute their workloads **across different cloud platforms.**
- Minimize the risk of **vendor lock-in** and providing **greater flexibility.**
- **Unique strengths** and **capabilities** of different providers, rather than being limited to a single cloud ecosystem.
- Organizations don't **want lock in to a single cloud provider**, distribute app and data among multiple cloud providers.
- **Advantages;** improved reliability, cost optimization, and the ability to leverage the best features and services from different providers.
- Organizations **can tailor their infrastructure** to better align with their specific needs, regulatory requirements, and performance goals.



Evolution of Cloud Platforms: Cloud Hosting Models: IaaS - CaaS - PaaS - FaaS - SaaS - Serverless

IaaS (Infrastructure as a Service)

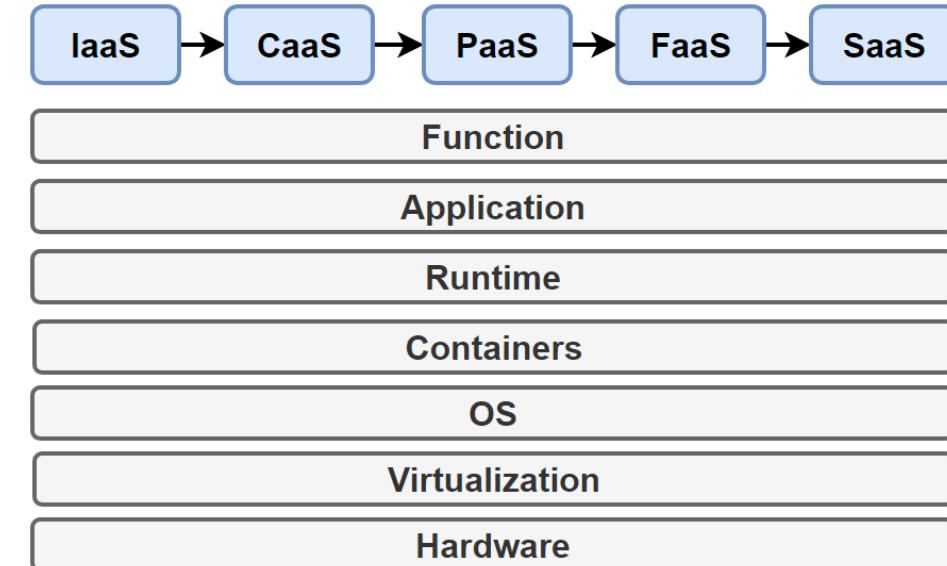
- Virtualized computing resources (VMs, storage, networking)
- Users control infrastructure, provider manages physical hardware
- Example: Amazon EC2, Microsoft Azure Virtual Machines
- Real-world: Hosting a web app on AWS EC2

CaaS (Container as a Service)

- Deploy, manage, and scale containerized applications
- Provider manages infrastructure and container orchestration
- Example: Google Kubernetes Engine, Amazon ECS, EKS, Fargate
- Real-world: Migrating a monolithic app to microservices using Google Kubernetes Engine

PaaS (Platform as a Service)

- Build, deploy, and manage applications without worrying about infrastructure
- Provider manages infrastructure, servers, networking, OS, etc.
- Example: Heroku, Microsoft Azure App Service
- Real-world: Developing a web app on Heroku



Hosting Models: IaaS - CaaS - PaaS - FaaS - SaaS - Serverless

FaaS (Function as a Service)

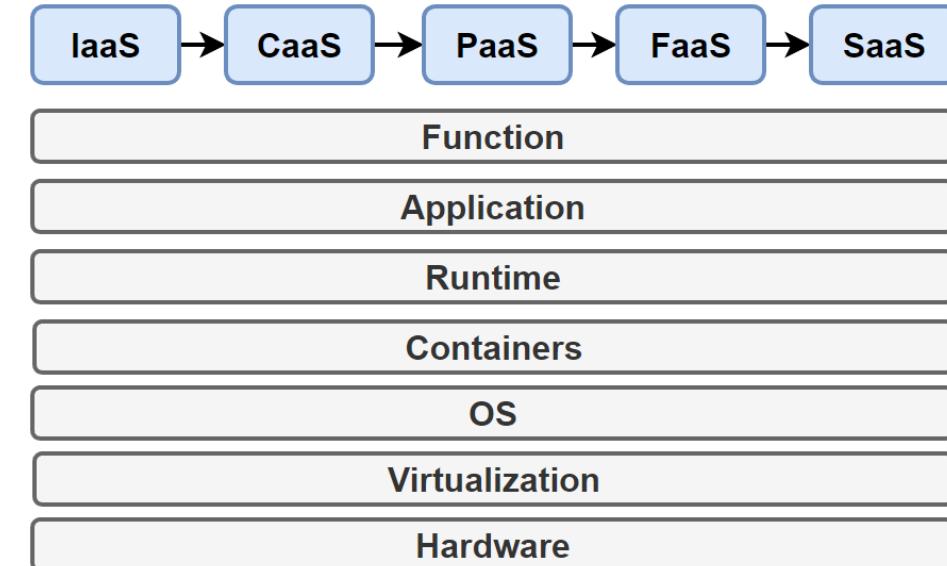
- Deploy individual functions, executed in response to events/triggers
- Provider manages infrastructure and auto-scales functions
- Example: AWS Lambda, Google Cloud Functions
- Real-world Example: Processing images using AWS Lambda

SaaS (Software as a Service)

- Software provided over the internet, eliminating need for installation
- Provider manages infrastructure, application, and data
- Example: Salesforce, Microsoft Office 365
- Real-world Example: Using Salesforce for CRM

Serverless

- Cloud provider manages infrastructure and auto-scales resources
- Pay only for resources consumed
- Example: Azure Functions with Azure Cosmos DB, AWS Lambda with Amazon DynamoDB
- FaaS is a specific implementation of serverless computing



Cloud-Native Application Architecture

- Designing, building, and running applications optimized for cloud computing environments. Scalable, resilient, and flexible applications for faster innovation and market responsiveness.

Microservices

- Small, loosely-coupled, independently deployable services. Improved agility, fault isolation, and continuous delivery.

Containers

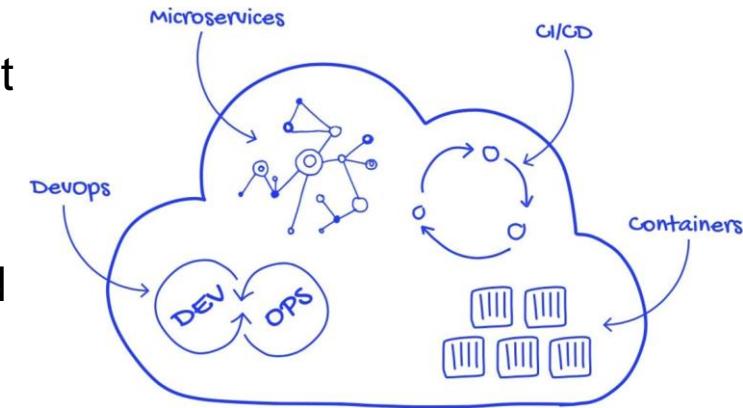
- Lightweight, portable runtime environments. Easier application management and deployment. Examples: Docker, containerd.

Orchestration

- Tools for managing containerized microservices. Automated scaling, rolling updates, and resource management. Example: Kubernetes.

Devops Practices

- Continuous Integration and Continuous Deployment (CI/CD), Infrastructure as Code (IaC). Streamlined development, testing, and deployment processes.



Cloud-Native Application Architecture - 2

Automation & CI/CD

- Infrastructure, deployment, scaling, and recovery processes. Uses CI/CD pipelines to automate the process of building, testing, and deploying code.

Scalability

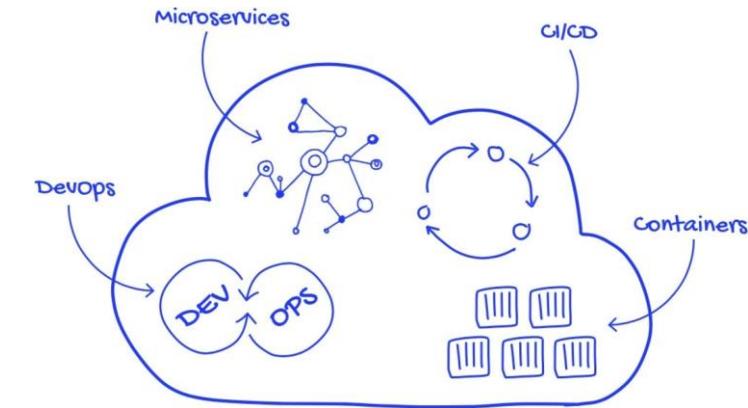
- Horizontal scaling for optimal resource usage and performance.

Resiliency

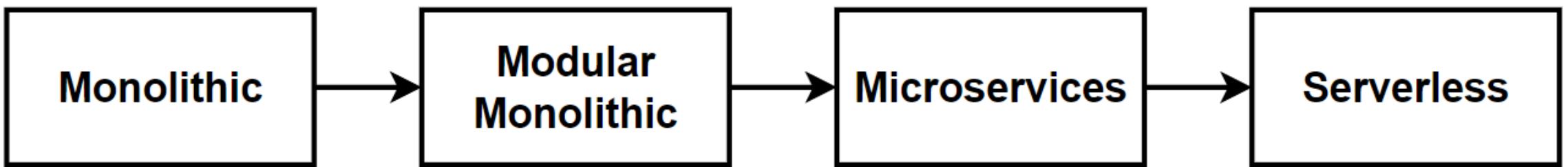
- Fault-tolerant and self-healing. Techniques: circuit breakers, retries, timeouts. High availability and minimized downtime.

Observability

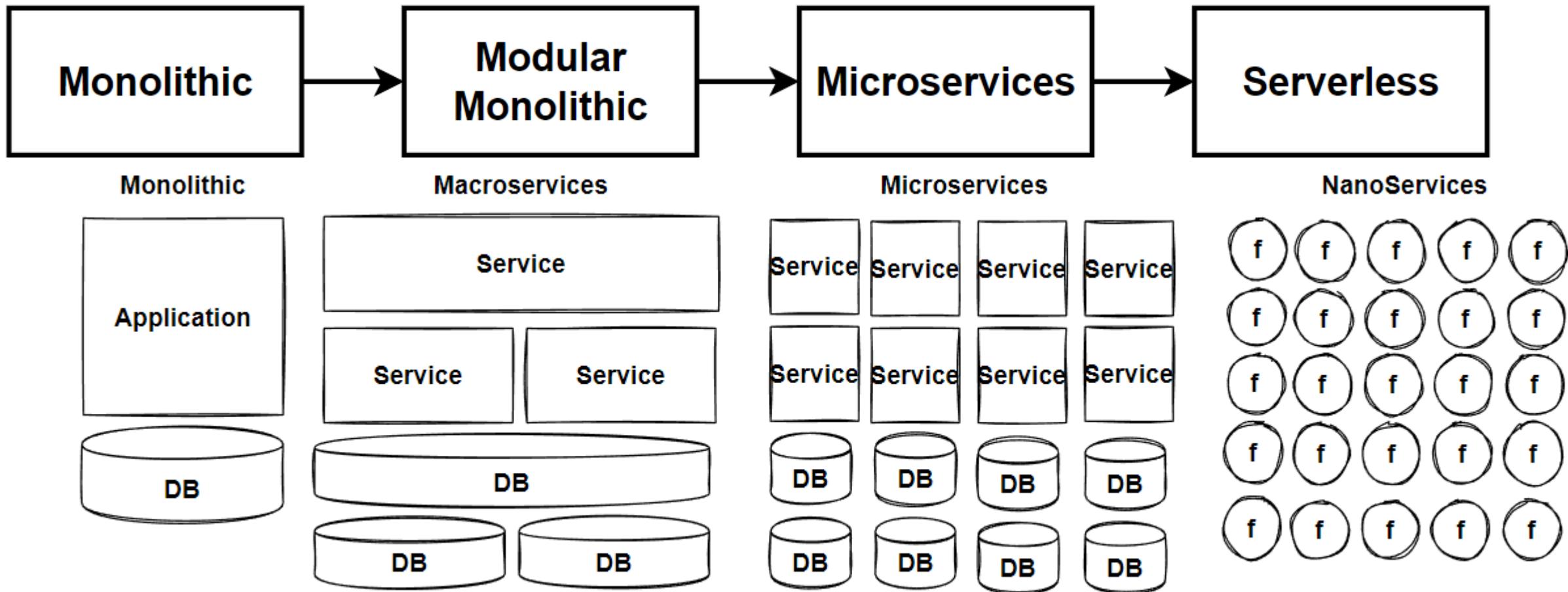
- Built-in monitoring, logging, and tracing. Insights into performance, health, and behavior. Tools: Prometheus, Fluentd, Jaeger.



Architecture Design Journey

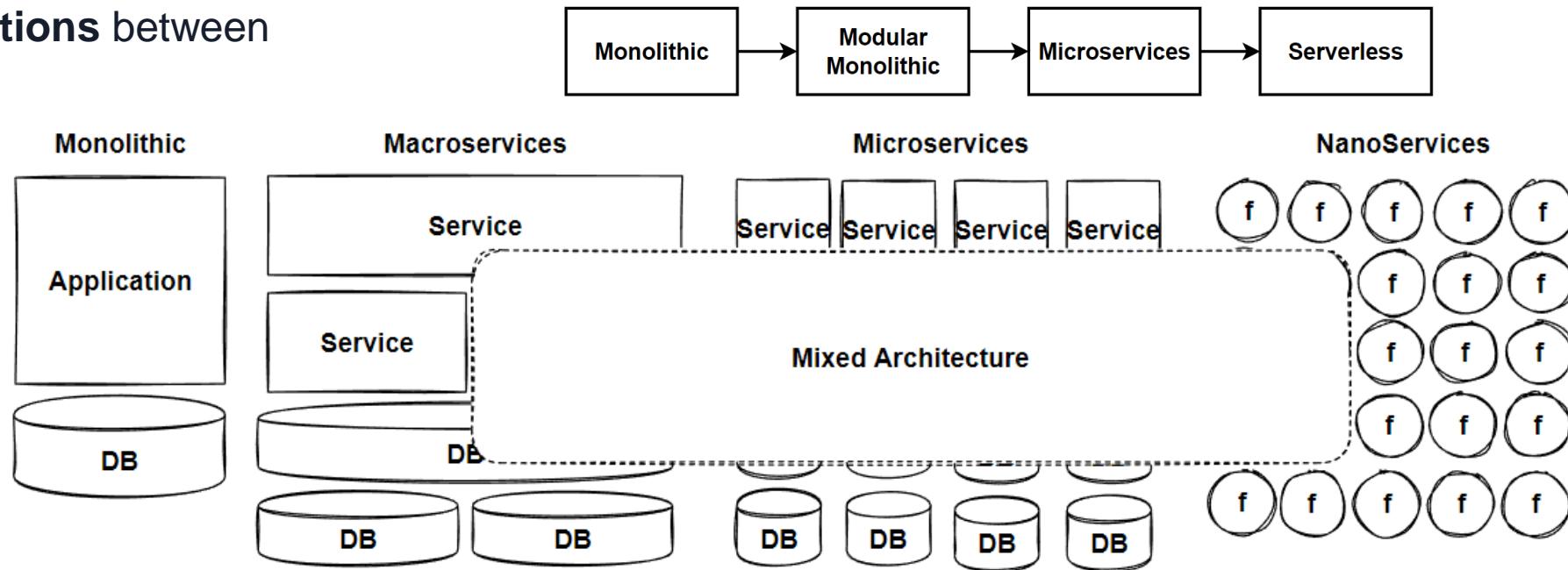


Macroservices to Nanoservices

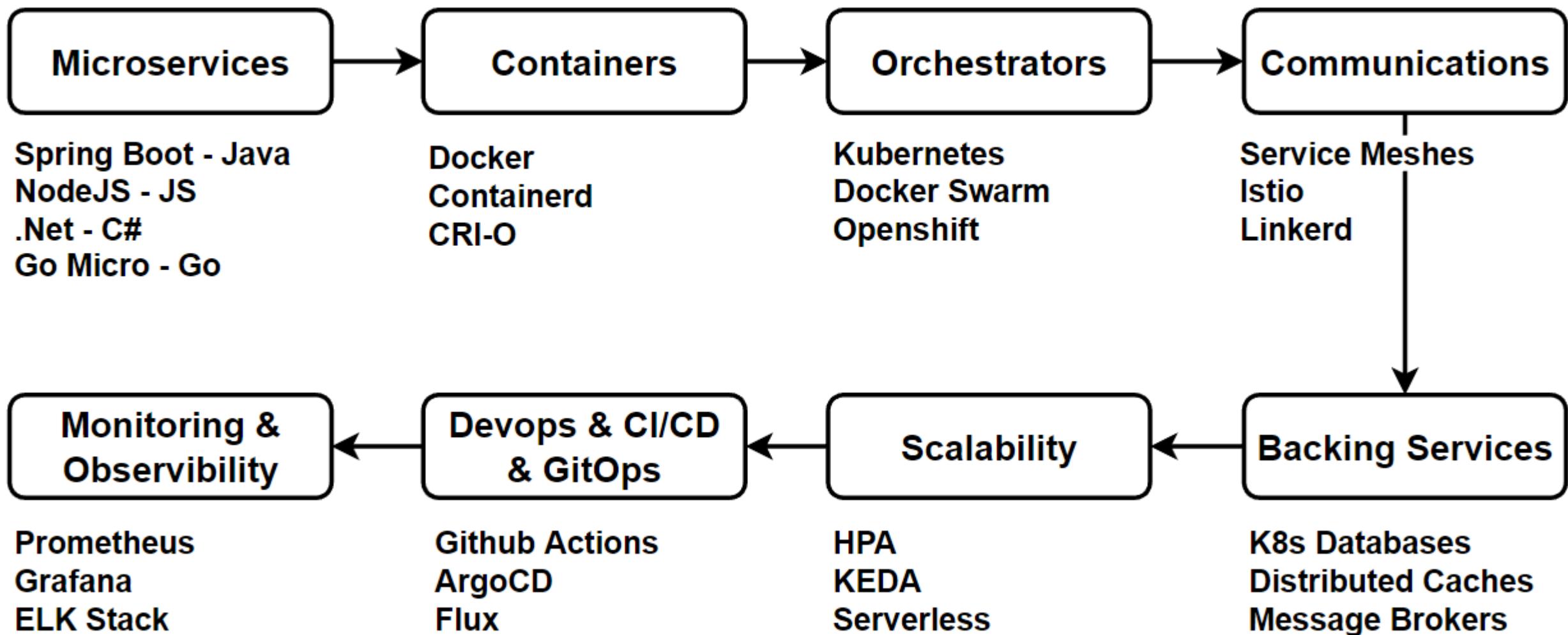


Which architecture approach we should choose ?

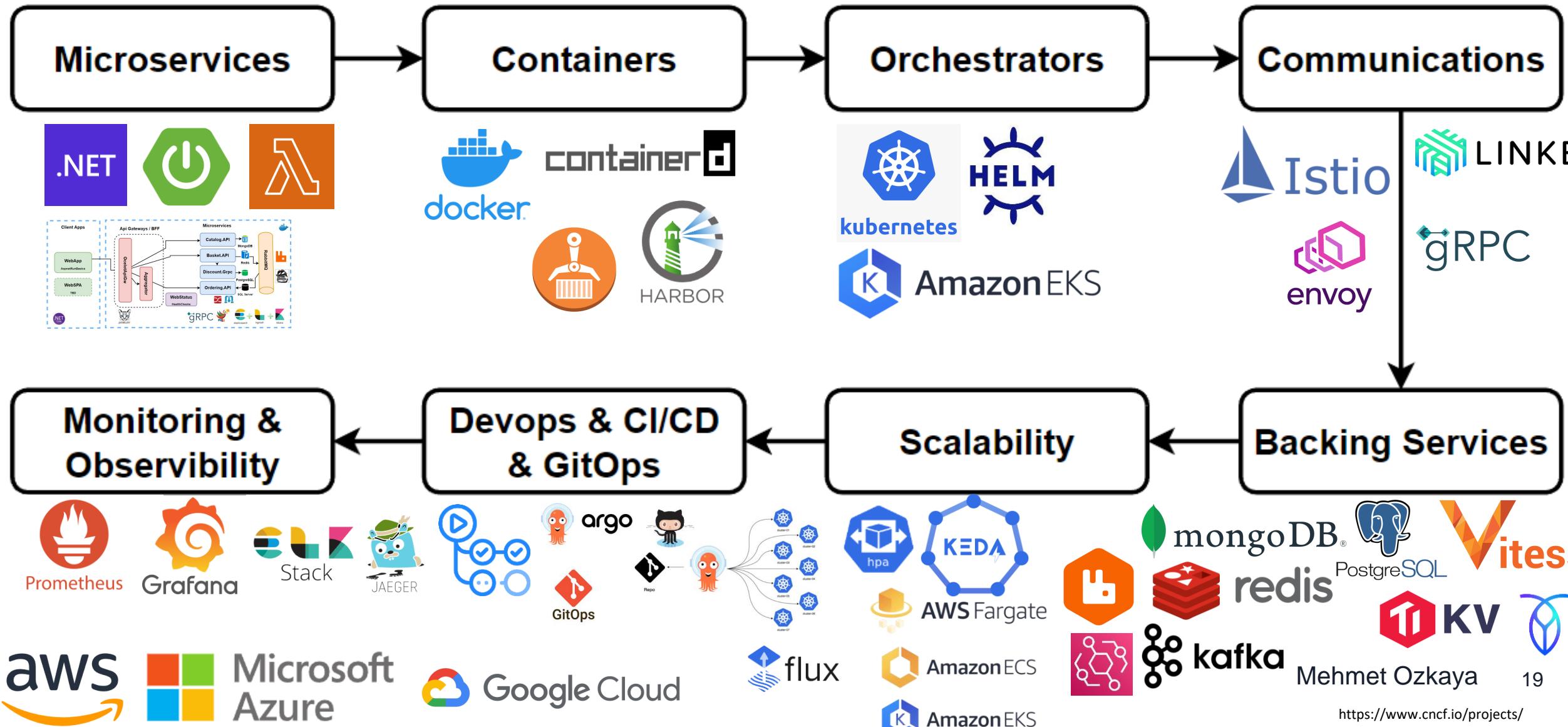
- It depends on your project
- Use **Mixed Architecture**
- Provide **transitions** between architectures



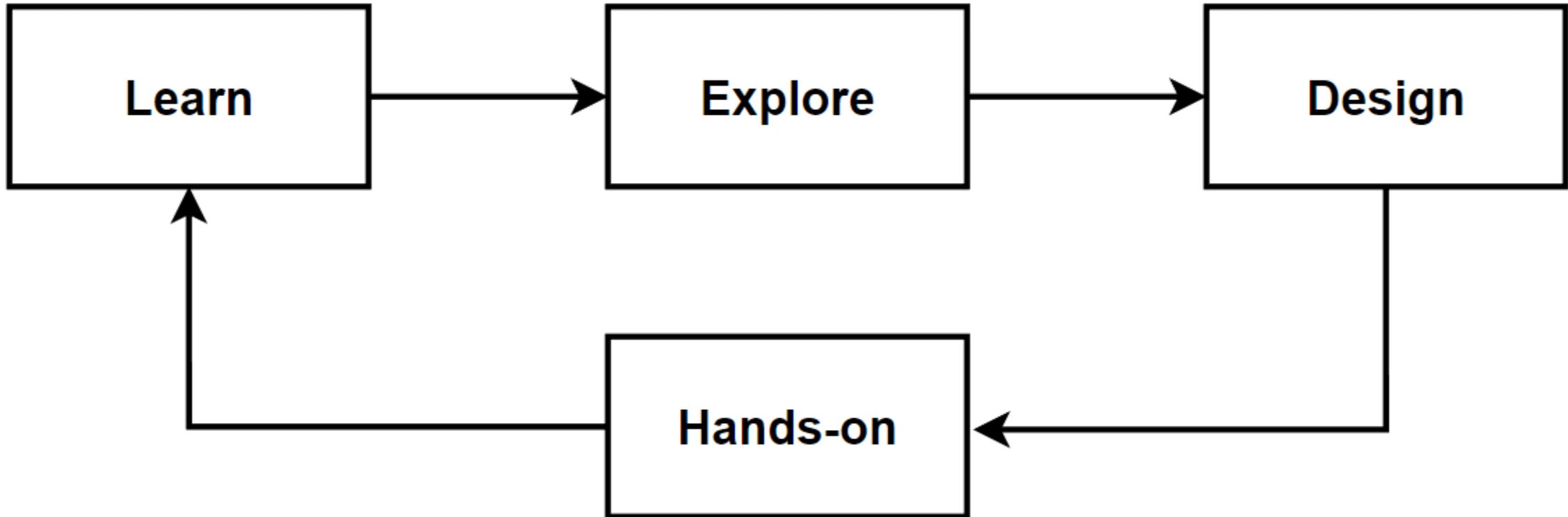
Cloud-Native Pillars Map – The Course Section Map



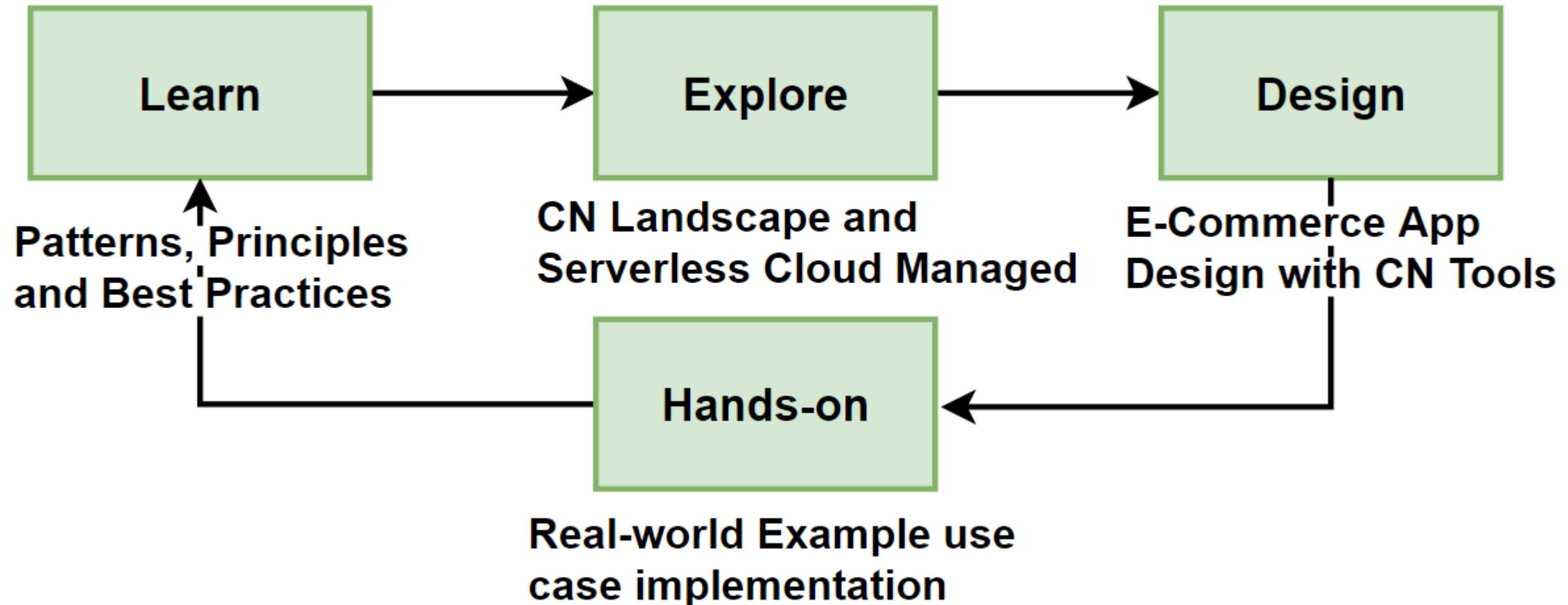
Cloud-Native Tools and Technologies that we will practice



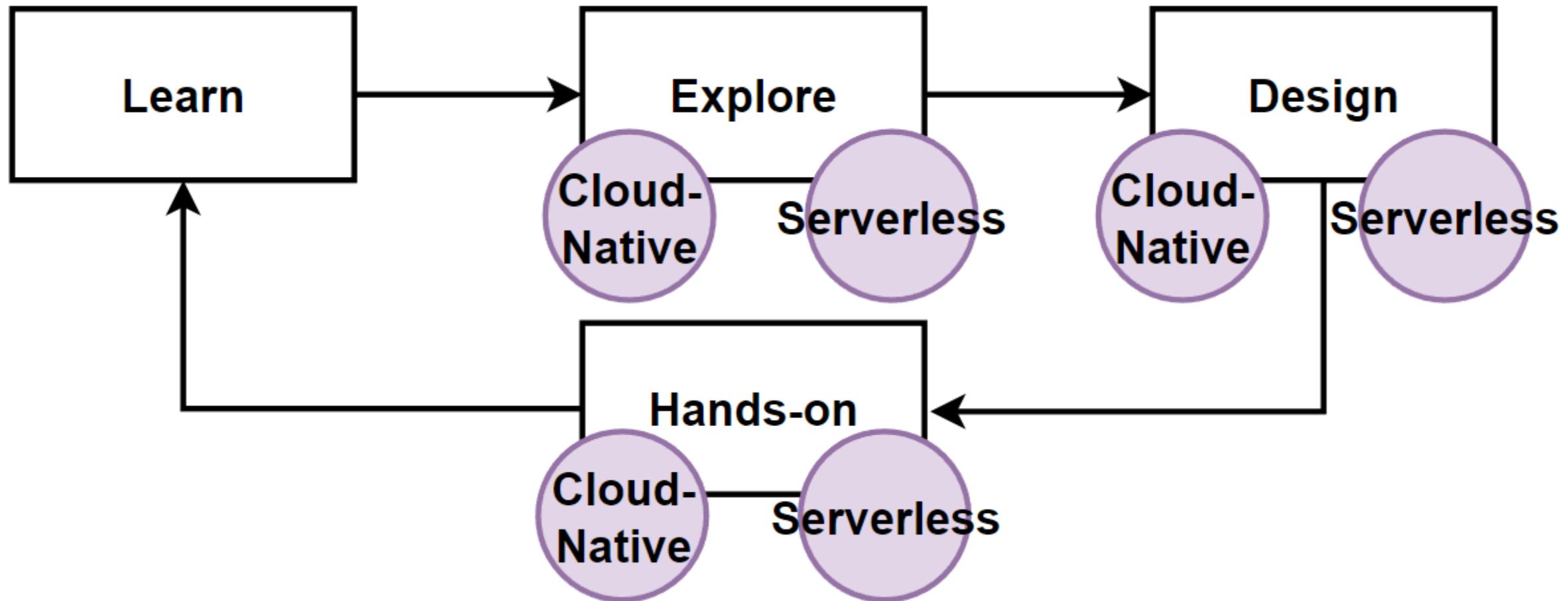
Way of Learning – The Course Flow



Way of Learning – The Course Flow



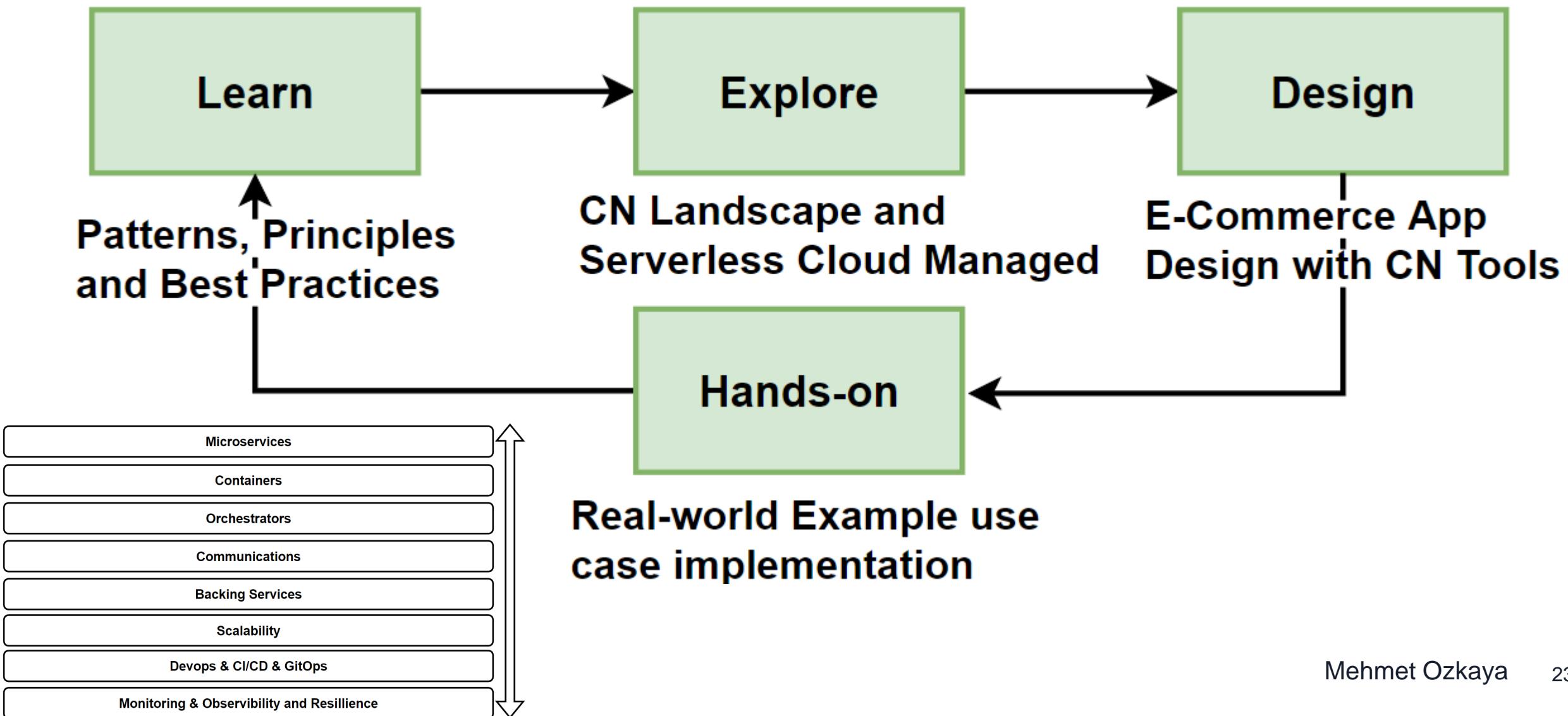
Way of Learning – Cloud-Native & Serverless Cloud Managed Tool



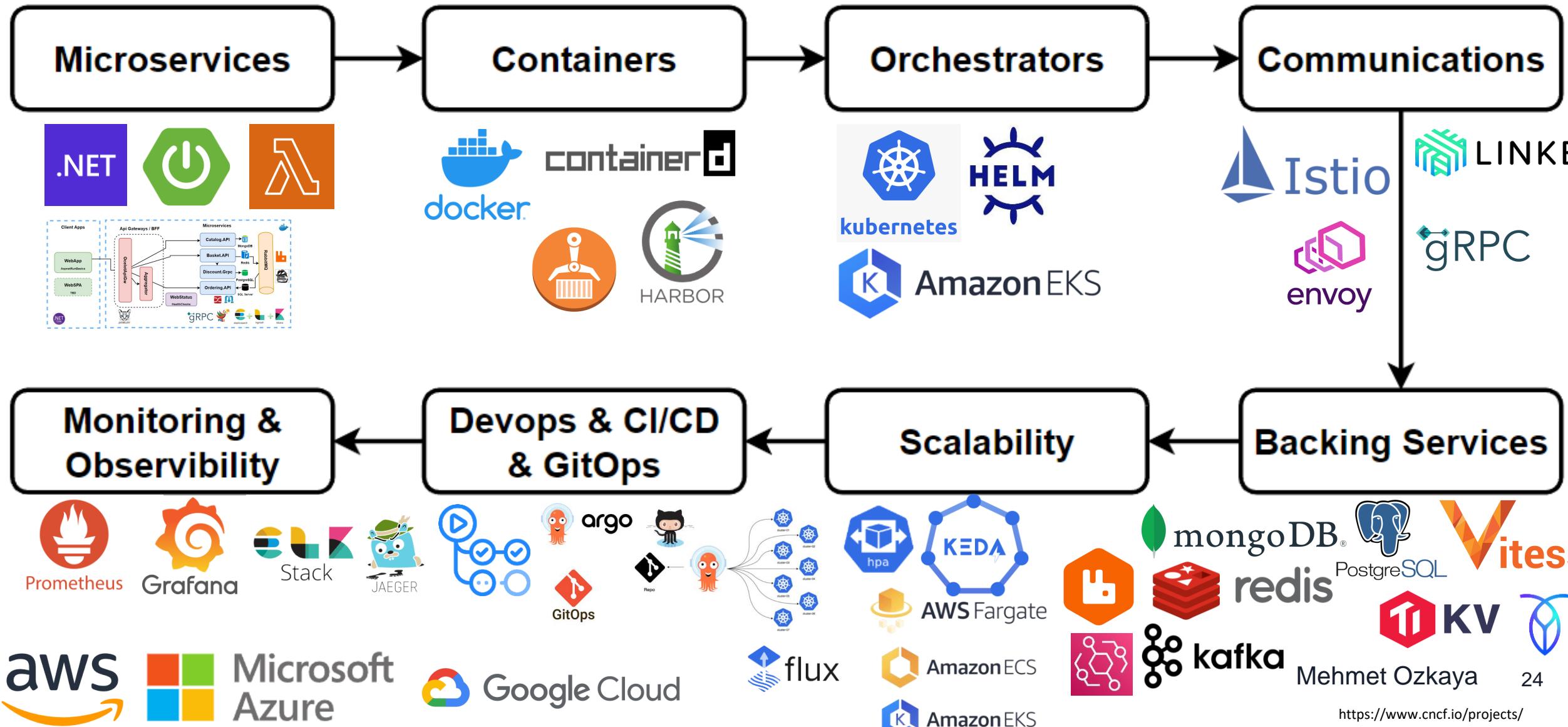
Examples of CN vs Serverless Cloud Tools:

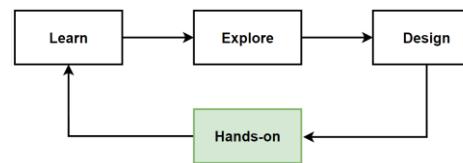
- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

Way of Learning – The Course Flow

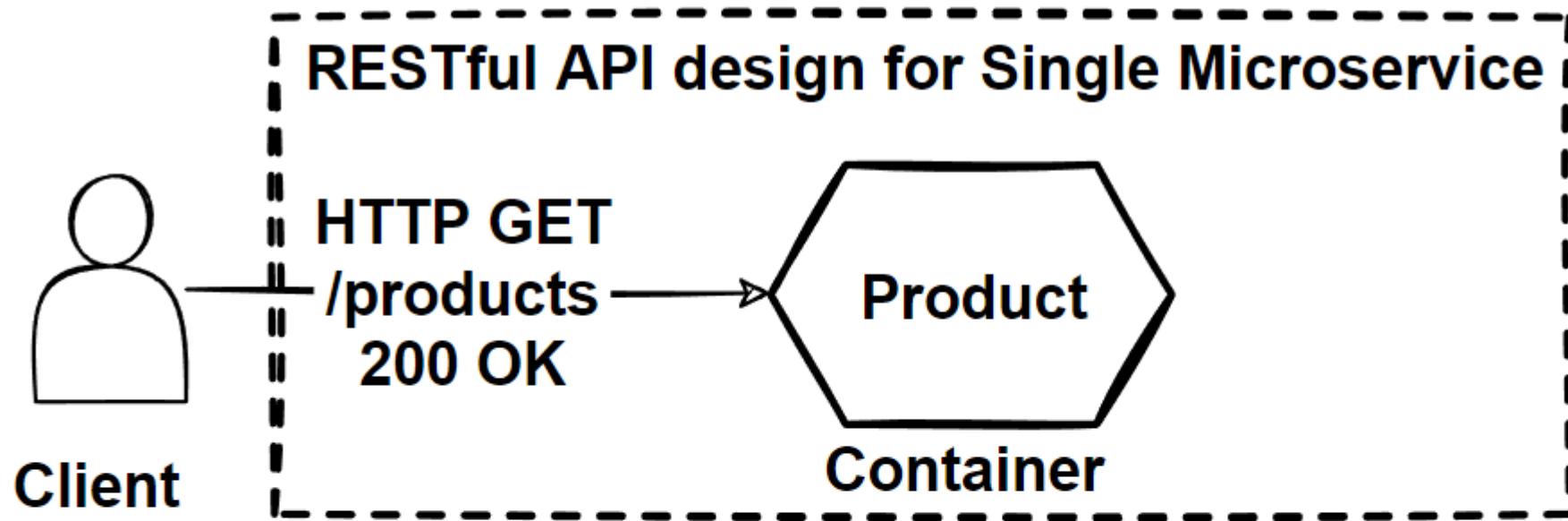


Cloud-Native Pillars – Hands-on Labs





Hands-on: Develop a RESTful Microservices with CRUD



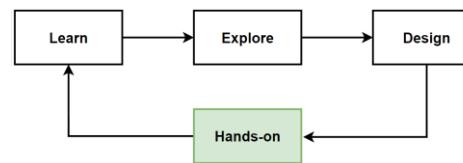
GET /api/products: Retrieves all products.

GET /api/products/{id}: Retrieves a specific product by ID.

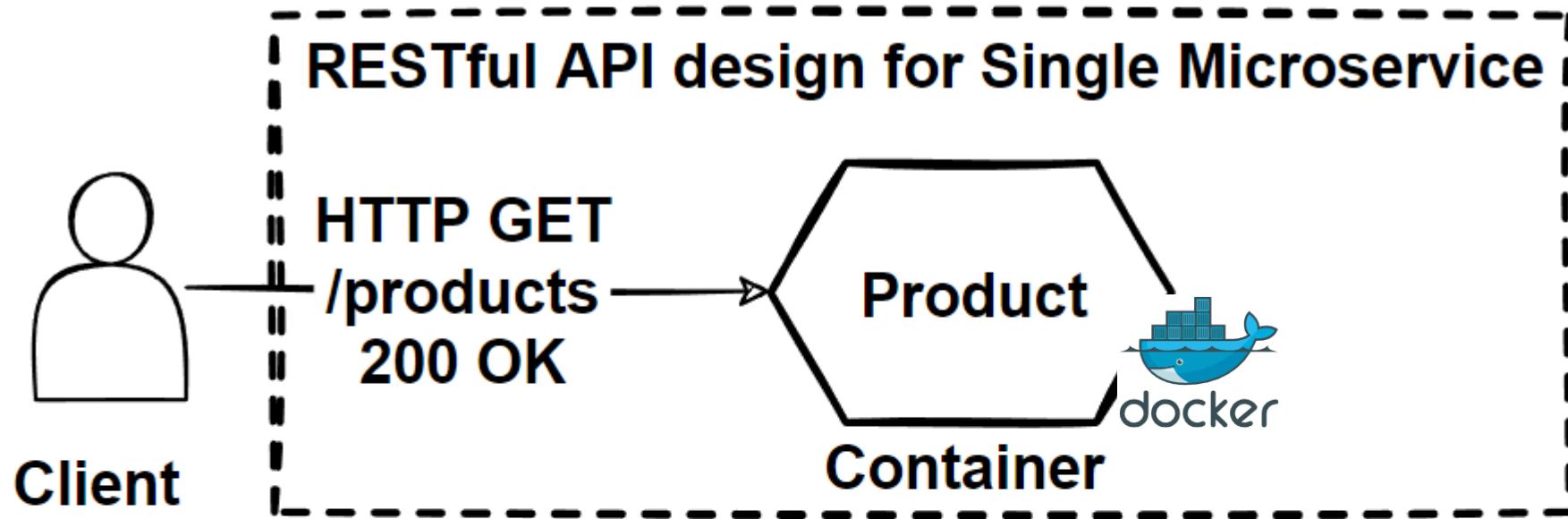
POST /api/products: Adds a new product.

PUT /api/products/{id}: Updates an existing product by ID.

DELETE /api/products/{id}: Deletes a specific product by ID.



Hands-on: Containerize .Net Microservices with Docker



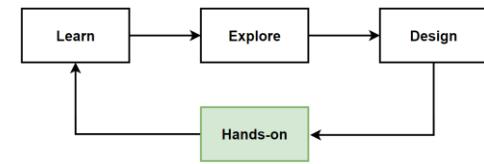
GET /api/products: Retrieves all products.

GET /api/products/{id}: Retrieves a specific product by ID.

POST /api/products: Adds a new product.

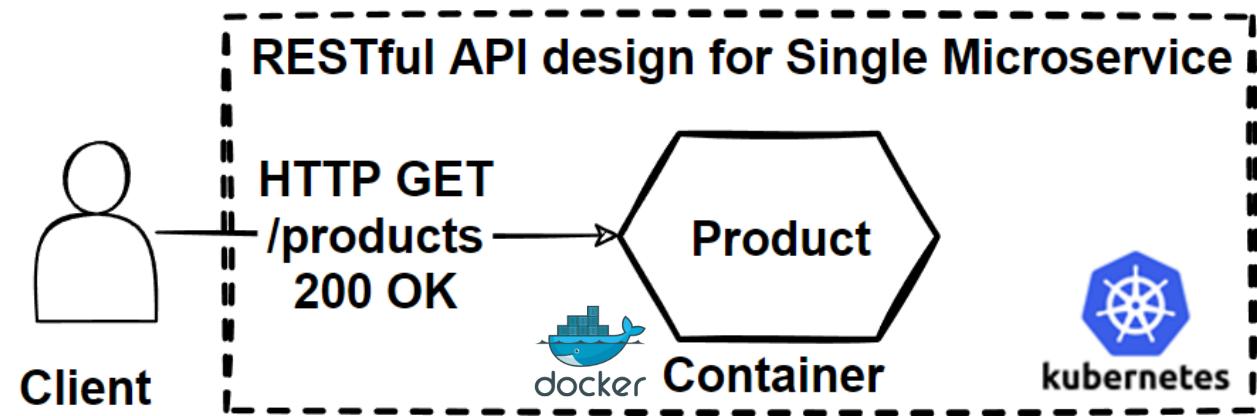
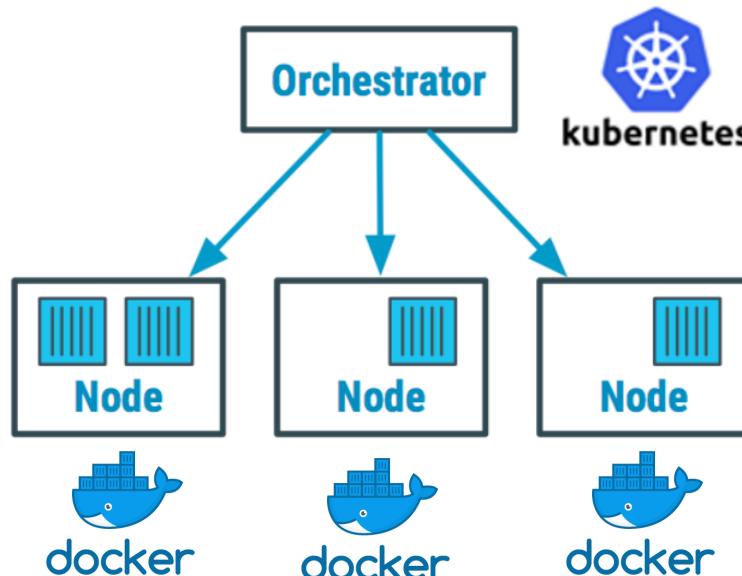
PUT /api/products/{id}: Updates an existing product by ID.

DELETE /api/products/{id}: Deletes a specific product by ID.

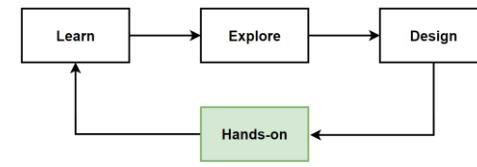


Hands-on: Deploy Microservices to Kubernetes

- Setting Up a Kubernetes Cluster
- Deploying Microservices on Kubernetes
- Docker Containers and Kubernetes Pods
- Deploying, Scaling, and Rolling Updates with Deployments
- Service Exposure and Ingress Controllers

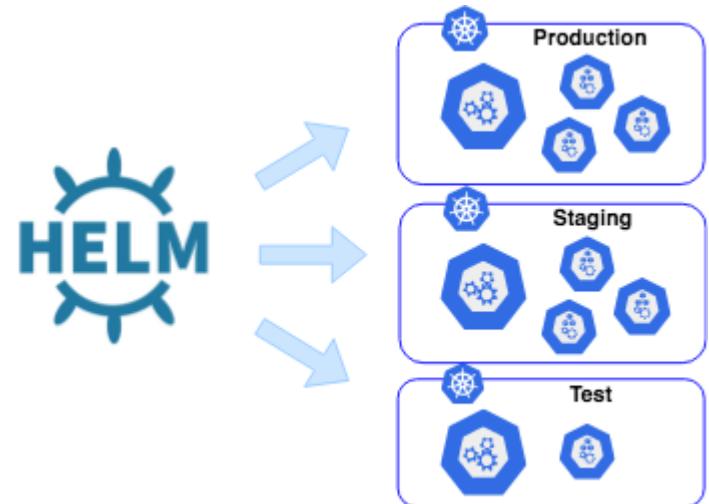
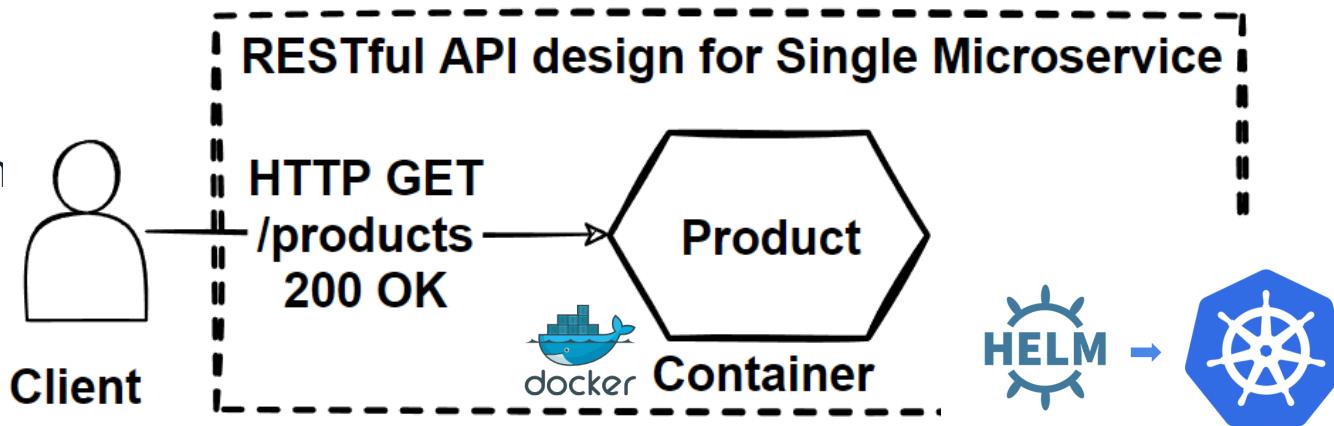


GET /api/products: Retrieves all products.
GET /api/products/{id}: Retrieves a specific product by ID.
POST /api/products: Adds a new product.
PUT /api/products/{id}: Updates an existing product by ID.
DELETE /api/products/{id}: Deletes a specific product by ID.

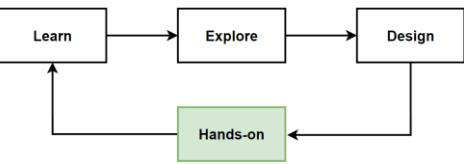


Hands-on: Deploy Microservices to Kubernetes with Helm Charts

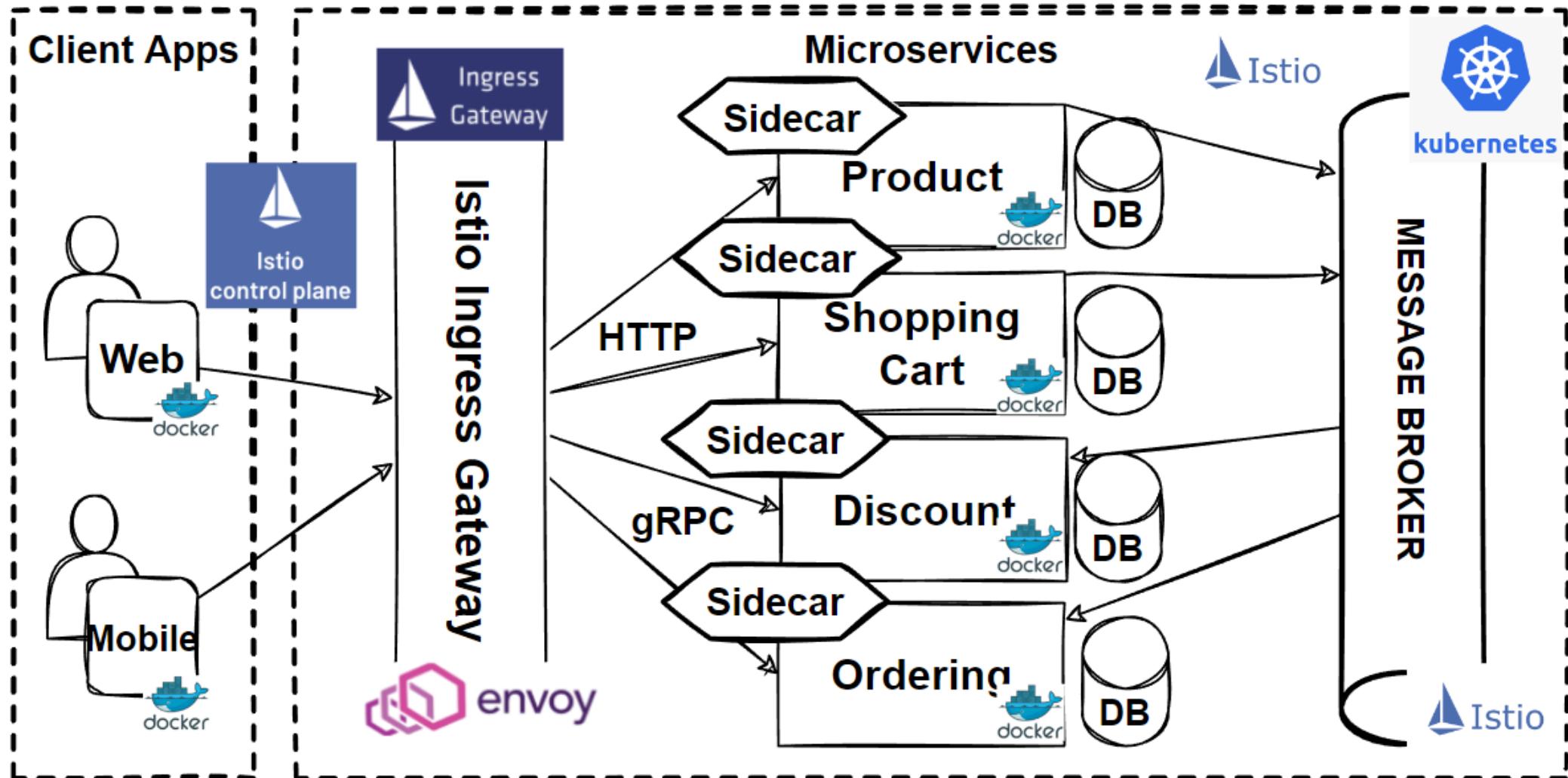
- Setting Up a Kubernetes Cluster
- Create helm chart for microservices
- Deploying Microservices on Kubernetes with helm
- Deploying, Scaling, and Rolling Updates with Deployments
- Service Exposure and Ingress Controllers

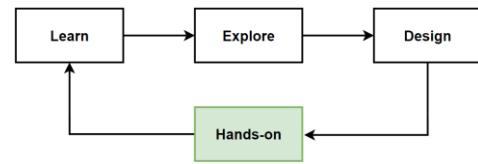


GET /api/products: Retrieves all products.
GET /api/products/{id}: Retrieves a specific product by ID.
POST /api/products: Adds a new product.
PUT /api/products/{id}: Updates an existing product by ID.
DELETE /api/products/{id}: Deletes a specific product by ID.



Hands-on: Deploy Microservices to Kubernetes with Service Mesh Istio and Envoy

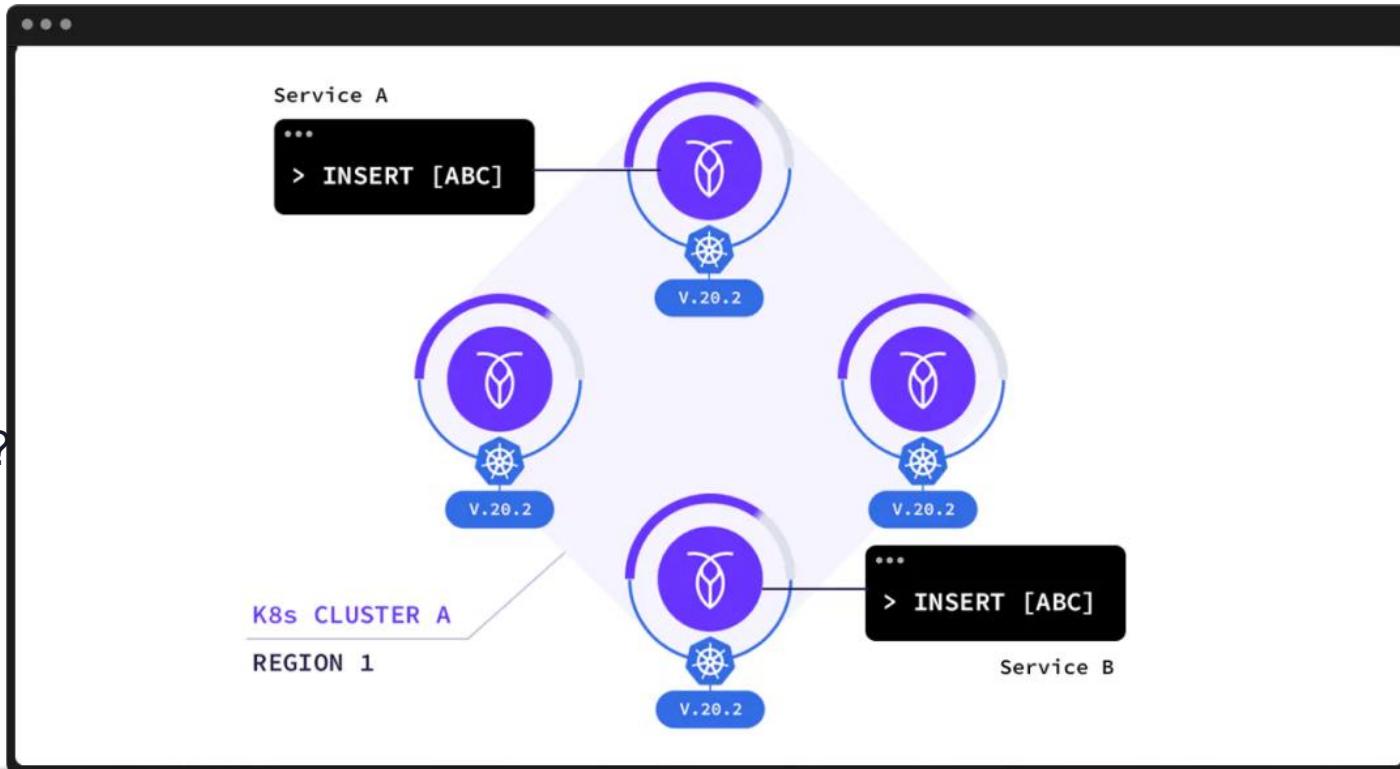




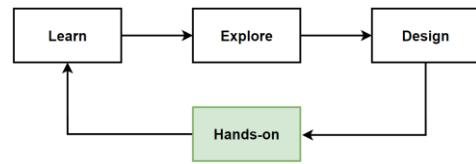
Hands-on: Deploy CockroachDB in a Single Kubernetes Cluster with Minikube – Task List

- Step 1. Start Kubernetes - minikube start
- Step 2. Start CockroachDB - Deploy with Kubernetes Operator
- Step 3. Use the built-in SQL client
- Step 4. Access the DB Console
- Step 5. Simulate node failure and node scales
- Step 6. Stop the cluster

- What is K8s Operator and Why use Operators ?
- CockroachDB Kubernetes Operator

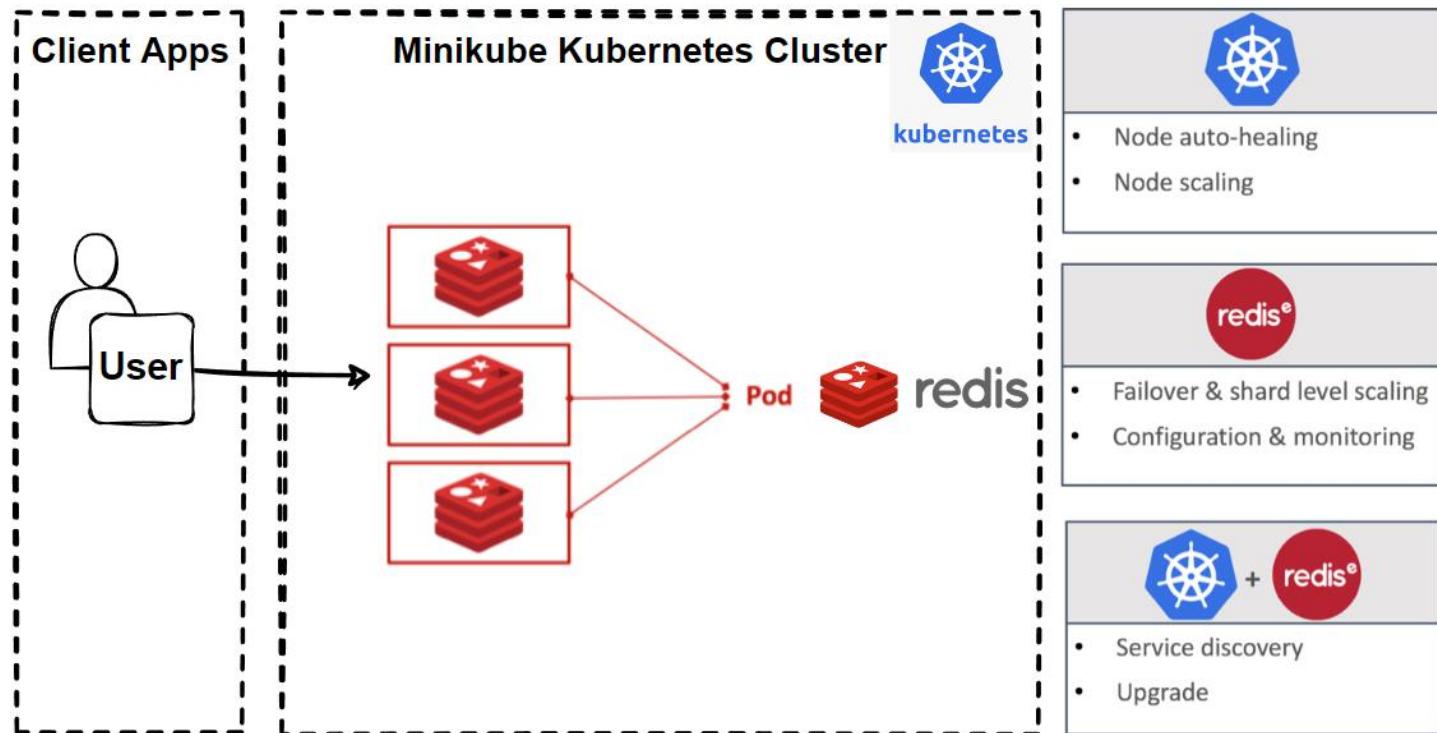


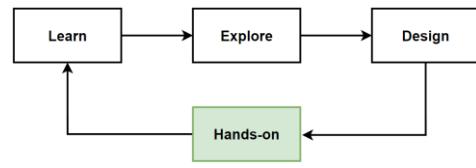
<https://www.cockroachlabs.com/product/kubernetes/>



Hands-on: Deploy Redis in a Single Kubernetes Cluster with Minikube – Task List

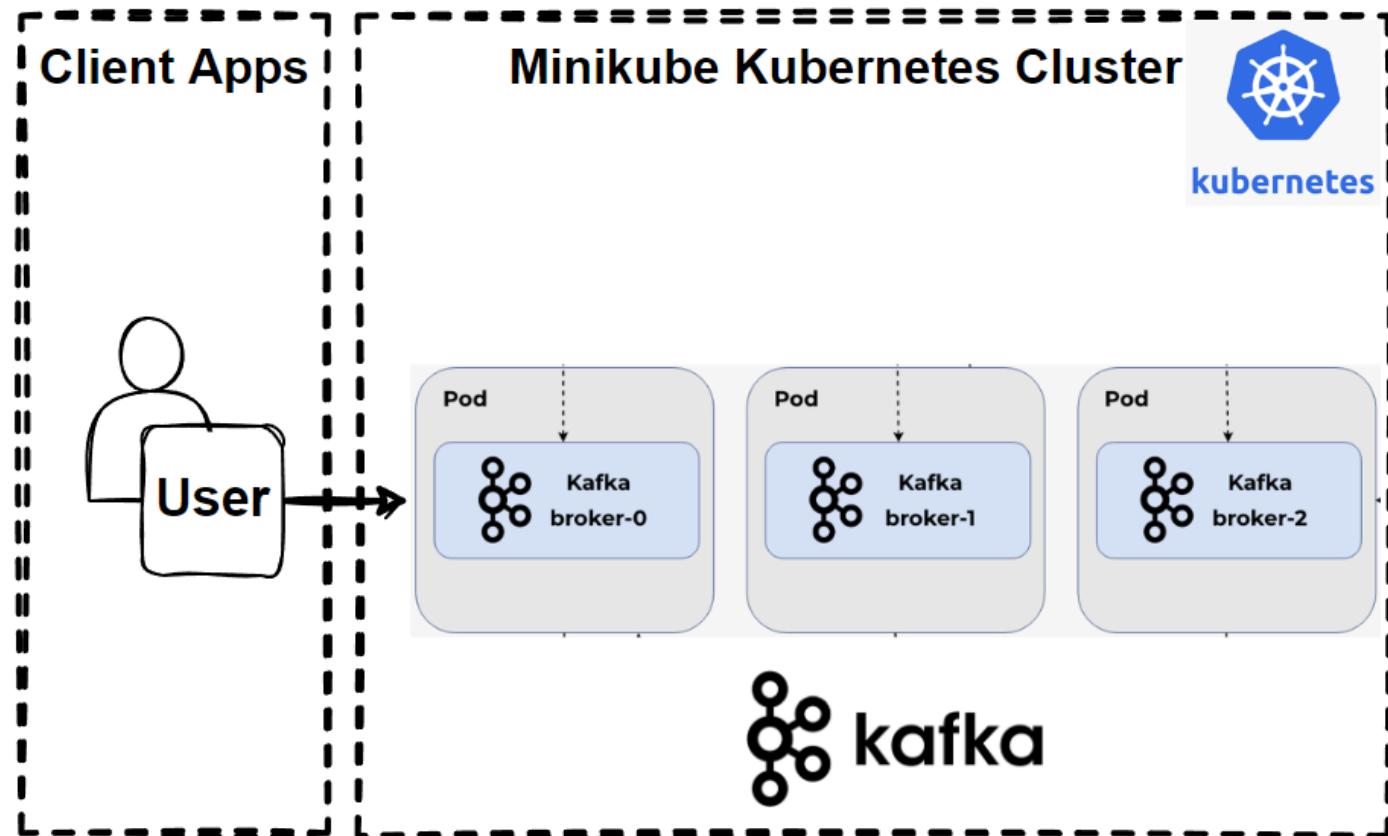
- Step 1. Start Kubernetes - minikube start
- Step 2. Start Redis - Deploy with Bitnami Helm Charts
- Step 3. Use the built-in Redis client
- Step 4. Simulate node failure and node scales
- Step 5. Stop the cluster

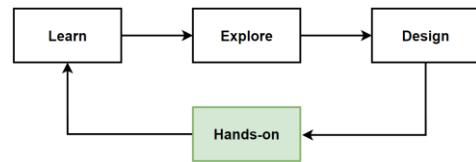




Hands-on: Deploy Kafka in a Single Kubernetes Cluster with Minikube – Task List

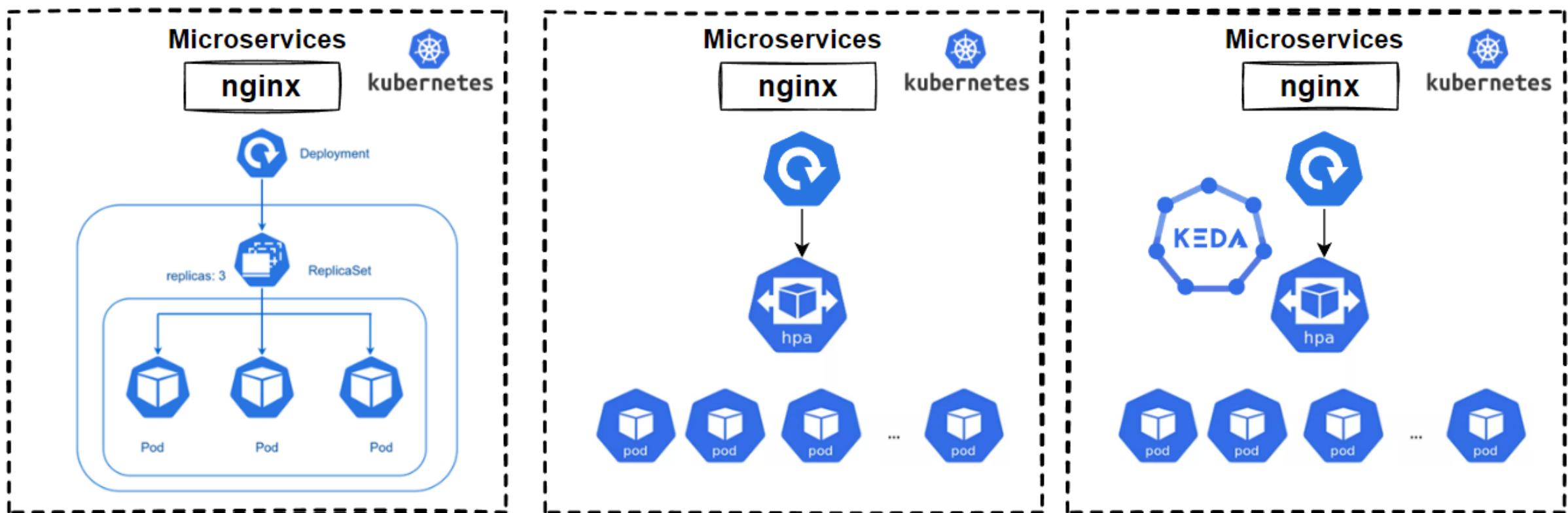
- Step 1. Start Kubernetes - minikube start
- Step 2. Start Redis - Deploy with Bitnami Helm Charts
- Step 3. Use the built-in Redis client
- Step 4. Simulate node failure and node scales
- Step 5. Stop the cluster

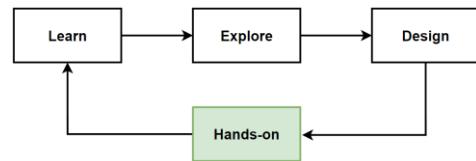




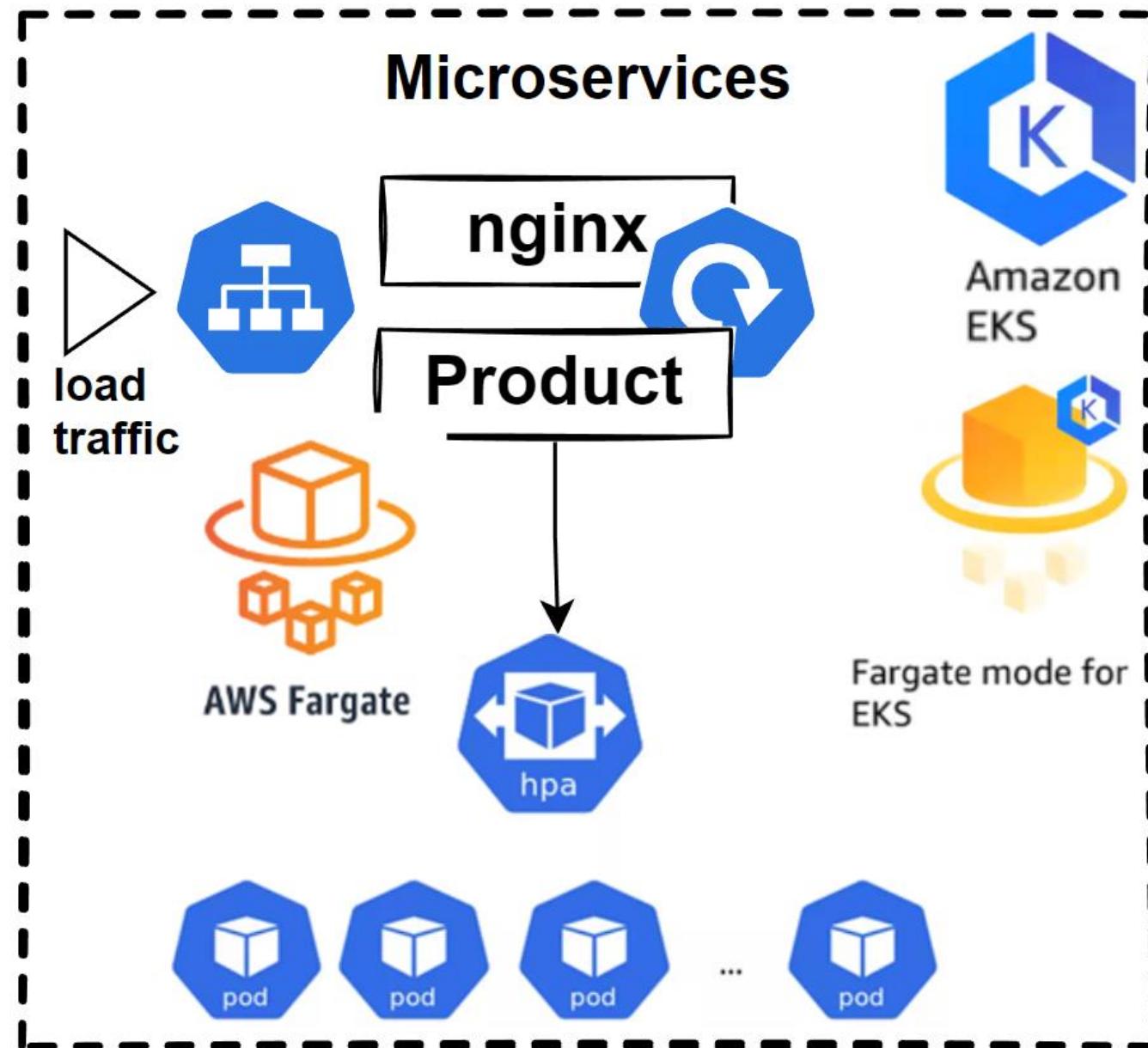
Hands-on: Scale Kubernetes Pods (VPA,HPA,KEDA) with Minikube – Task List

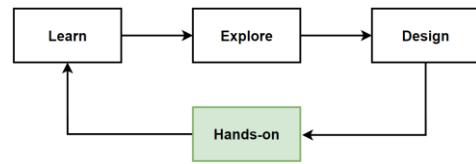
- Step 1. Manually Horizontal and Vertical scaling pods into Kubernetes Cluster with Minikube
- Step 2. Using HPA and VPA for scaling pods into Kubernetes Cluster with Minikube
- Step 3. Using KEDA for scaling pods into Kubernetes Cluster with Minikube



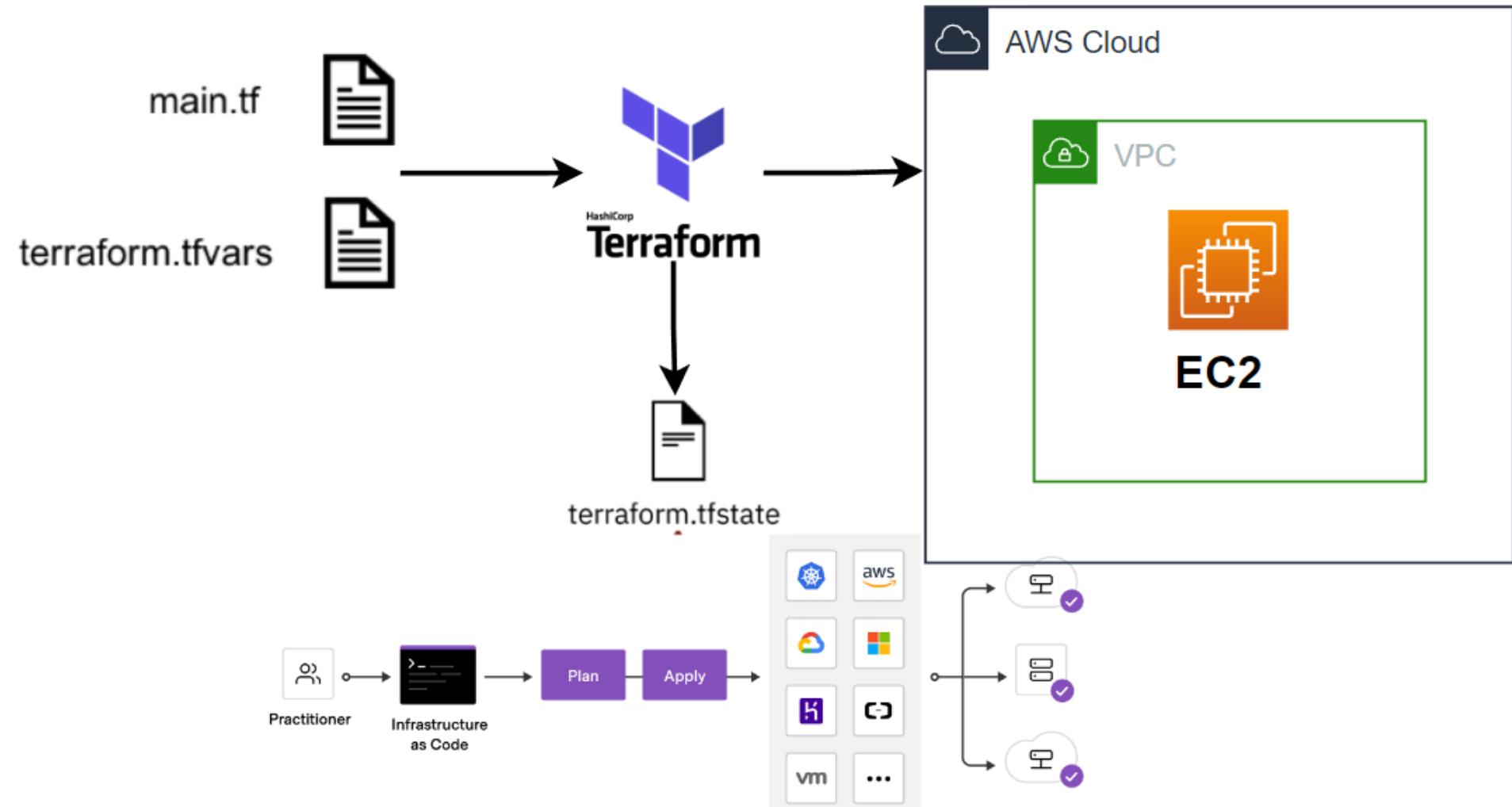


Hands-on: Deploy Amazon EKS Fargate – Architecture

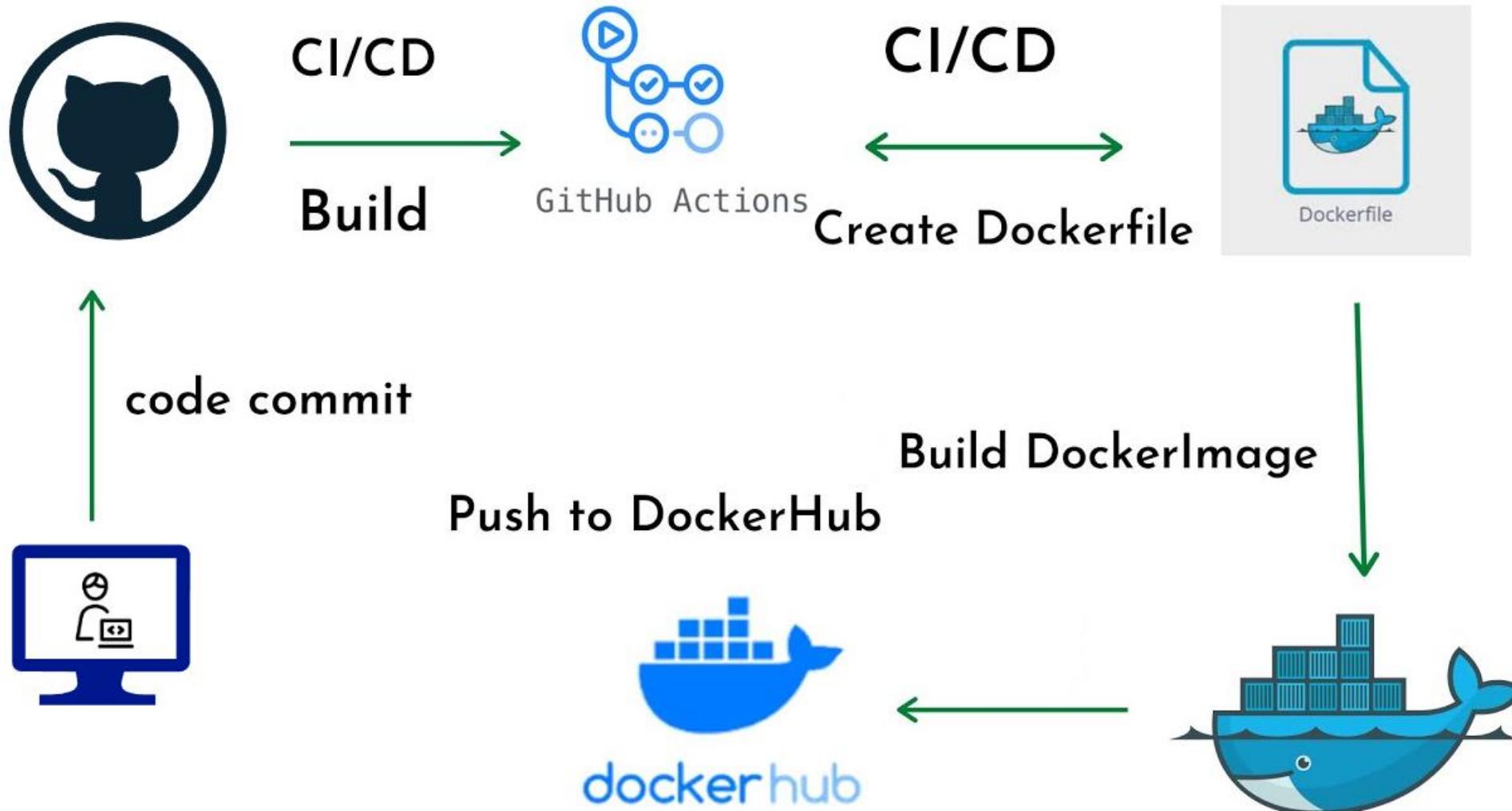
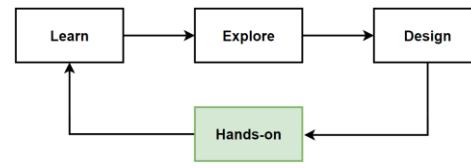


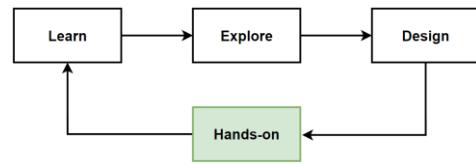


Hands-on: Terraform IaC provision AWS EC2 instance

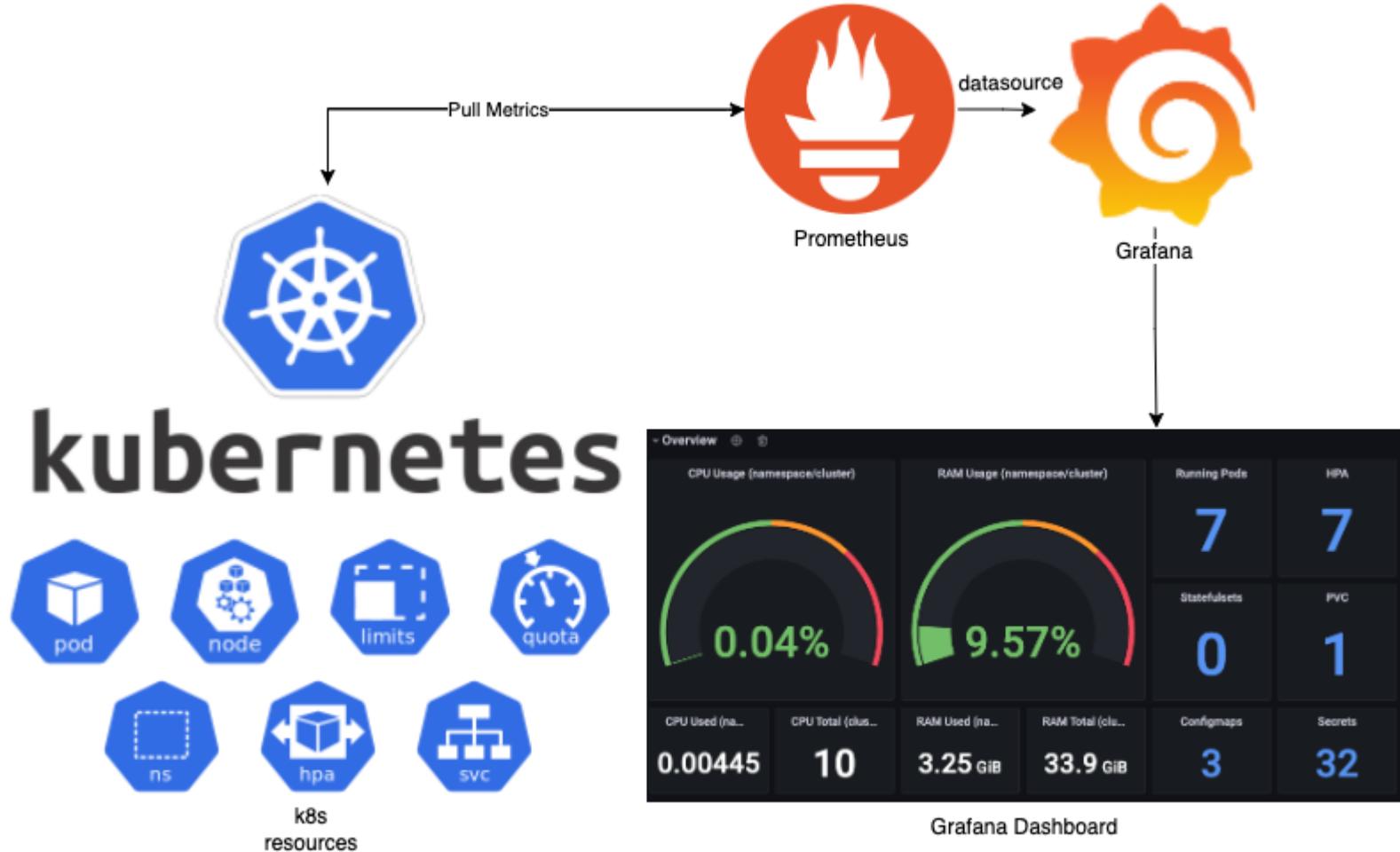


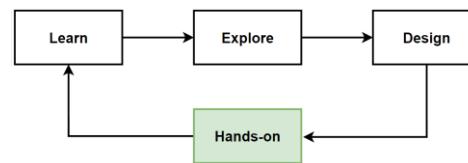
Hands-on: GitHub Actions CI/CD for Build & Push Docker Images to DockerHub





Hands-on: Kubernetes Monitoring with Prometheus and Grafana

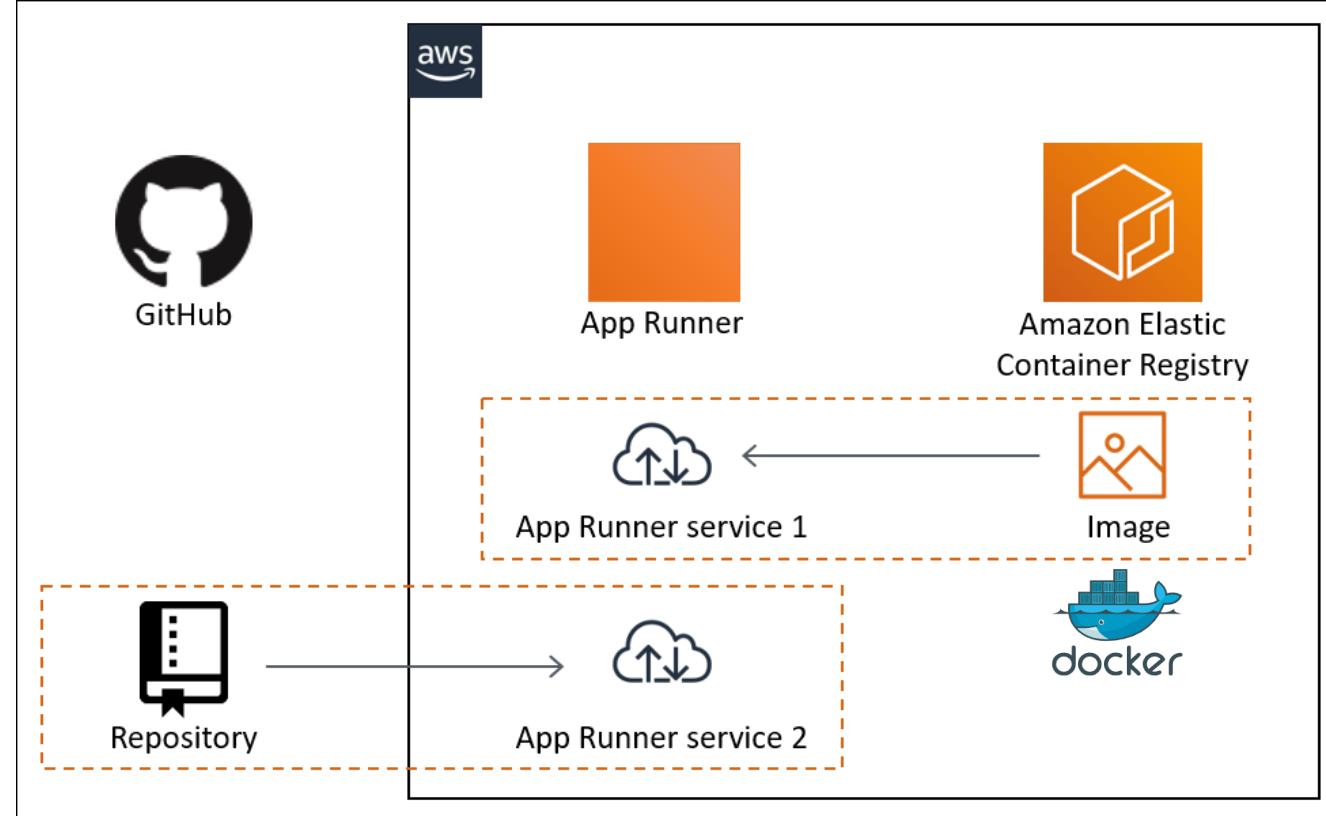




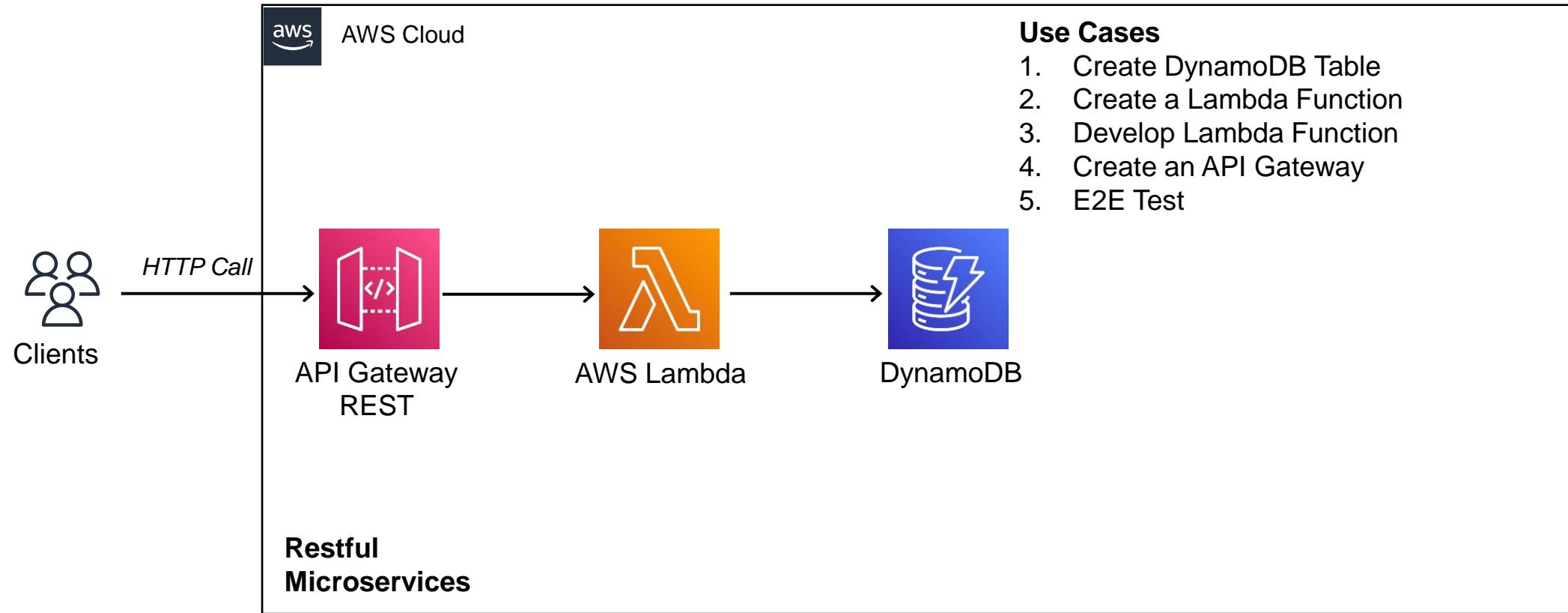
Hands-on: Deploy ProductService Container to AWS Apprunner

Todo List:

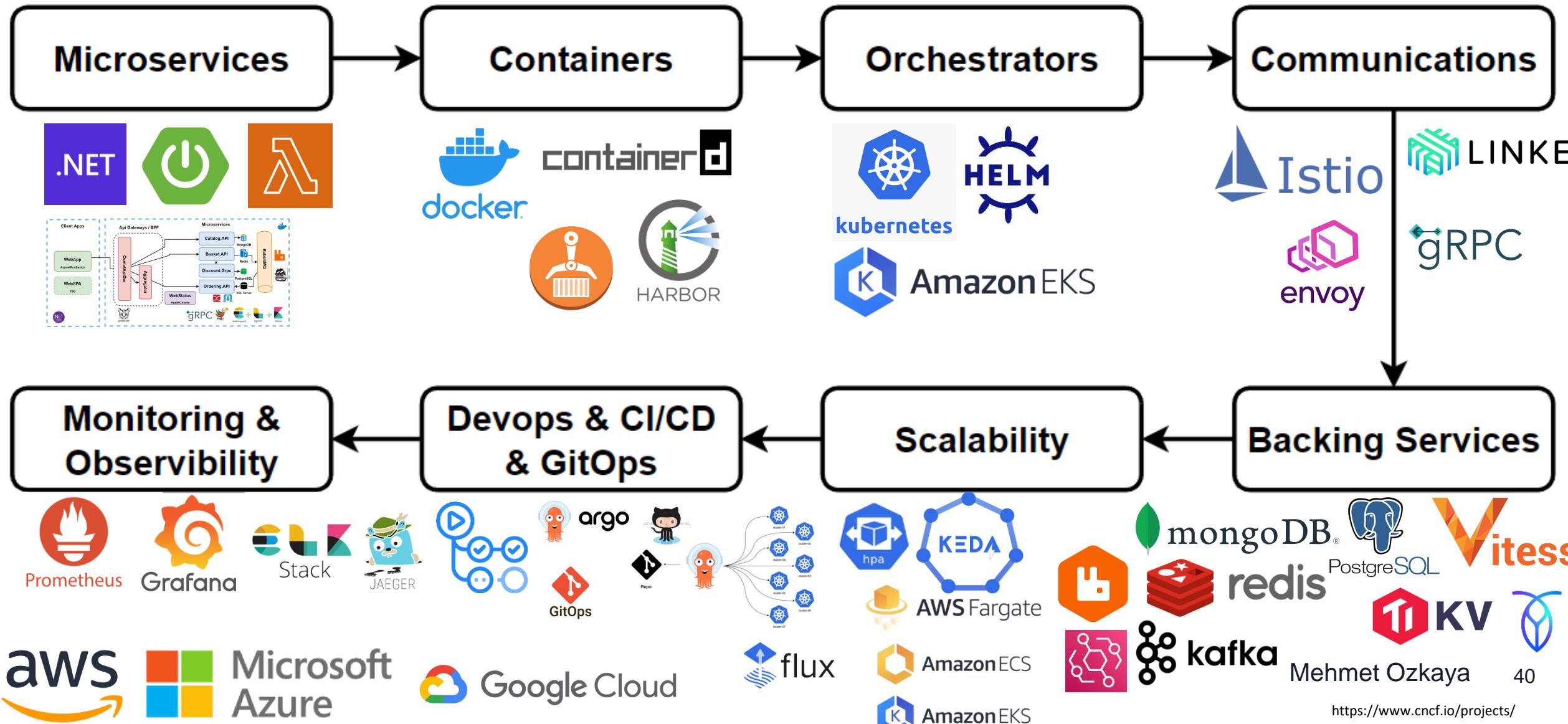
- Push your Docker container image to Amazon Elastic Container Registry (ECR)
- Deploy to AWS App Runner that pull image from ECR



Hands-on Lab: Building RESTful Microservices with AWS Lambda, API Gateway and DynamoDB



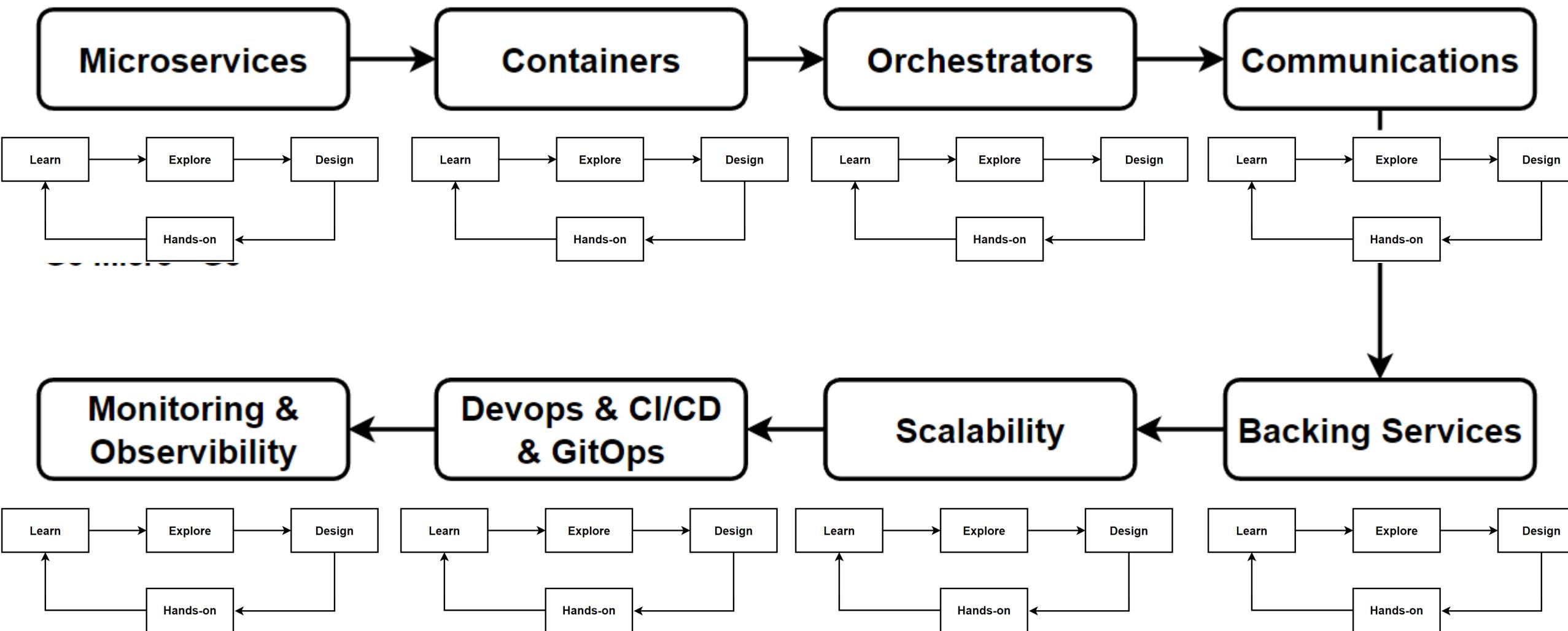
Cloud-Native Pillars – Hands-on Labs



How to Follow the Course

- I strongly recommended that you should take this course from **beginning to end**.
- If you already familiar **some cloud-native pillars**, jump into your target architectures and start to learn from that section.
- All **sections are independent** from each other and you can easily **switch on sections** with following different cloud-native pillars.
- Every section **starts with specific CN Pillar** like **Microservices, Container, Orchestrators** and **practice** this topics with learning new patterns and principles. Explore tools and design final architecture.
- If you have **prior knowledge** about Microservices, Containers and Orchestrators, **skip first sections** and **jump to** CN Communications and Backing Services.
- Follow the learning path which is **Learn -> Explore -> Design -> Hands on**
- **Skip definition** and goto **explore tools** or **design architecture** or directly jump to **Hands-on implementations**.
- This is **reference architecture course** that you can take any part of the course according to your Cloud-native architectural requirements.

Cloud-Native Pillars Map – The Course Section Map



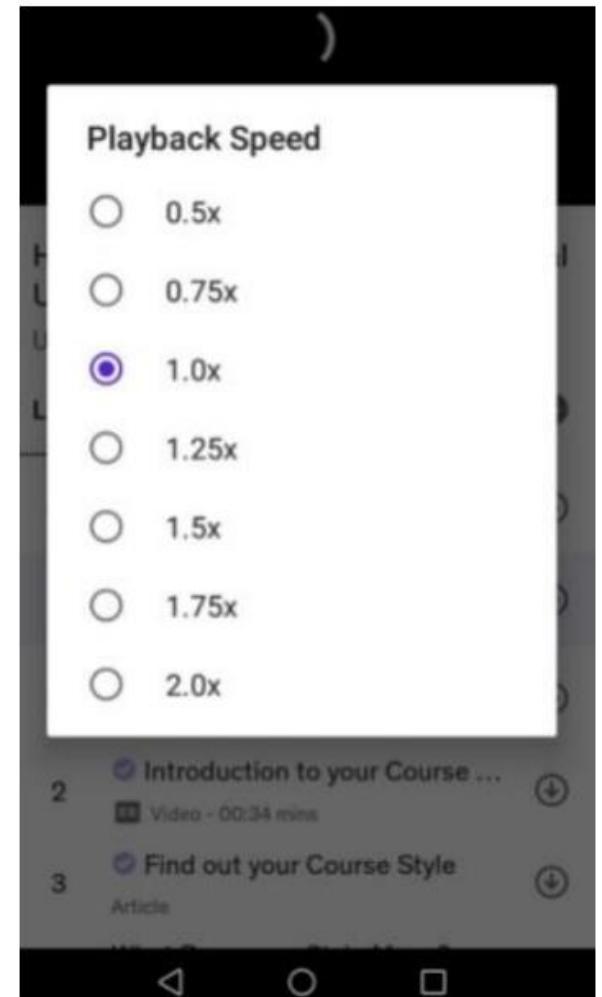
How to Follow the Course - 2

- **Increase Speed**

If you feel comfortable on any particular topic, please increase the video speed to avoid losing motivation of the course.

- **Put a Review**

Please put a comment and review the course, when you feel ready at any time of the course, this will help me a lot for further courses.



Course Slides

- **Powerpoint Slides**

Find full PowerPoint slides the link in the resource of this video.

- **Full Resources on GitHub (Codes and Slides)**

<https://github.com/mehmetozkaya/CloudNative>

Section 1: Introduction

1 / 9 | 39min

- 1. Introduction
- 2. Pre...

7min

Resources

Microservices Architecture on...

Introduction to Cloud-Native

What is Cloud-Native Architecture ?

Introduction to Cloud-Native Apps and Microservices

Understanding the fundamentals of cloud-native applications,
microservices architecture, and Kubernetes.

What is Cloud-Native ?

Cloud-Native

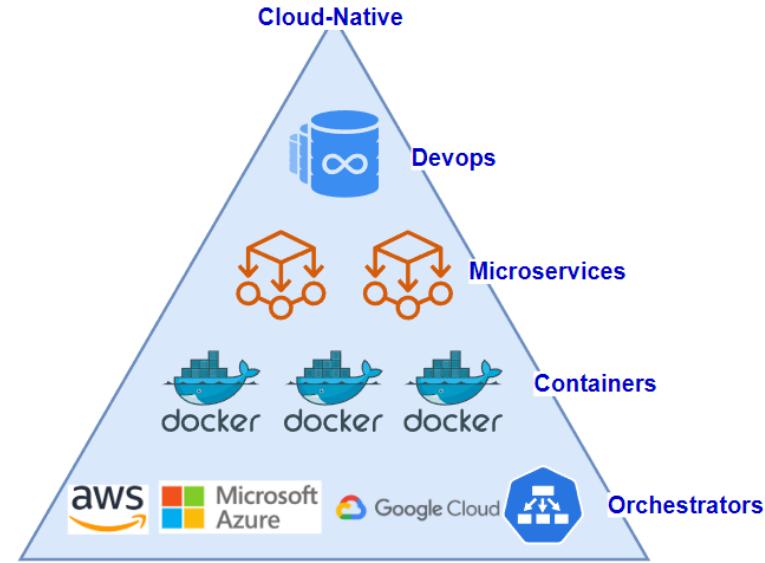
- Approach to building, running, and deploying modern applications
- Designed specifically for cloud environments
- Focus on scalability, resilience, and flexibility

Benefits of Cloud Native Applications

- Rapid innovation and response to market changes
- Modern tools and techniques for application development
- Frequent updates without impacting service delivery

Speed and Agility in Business

- Business systems as strategic tools for growth
- Users demand rapid responsiveness, innovative features, and zero downtime
- Cloud-native systems handle rapid change, large scale, and resilience



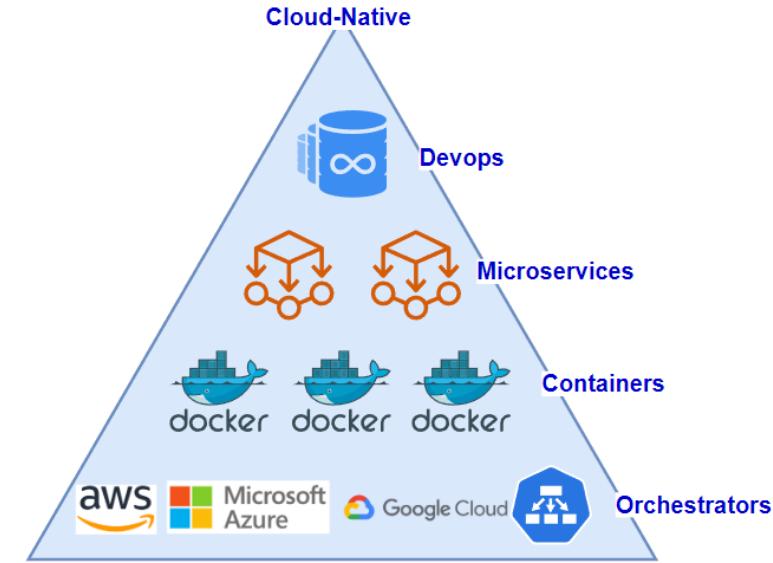
What is Cloud-Native ? - 2

Characteristics of Cloud Native Applications

- Loosely coupled components
- Optimized for cloud performance
- Use of managed services provided by cloud vendors
- Continuous delivery methodology

Successful Cloud Native Companies

- Examples: Netflix, Uber, Airbnb
- "Born in the cloud" companies
- Engineered to take full advantage of modern cloud infrastructure and delivery

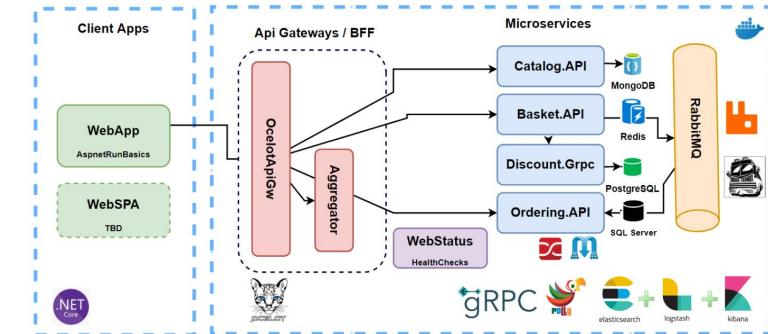
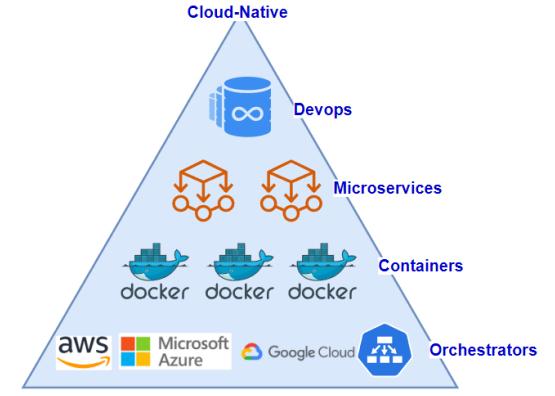


Cloud Native Architectural Style

- Composed of many independent microservices
- Rapid response to market conditions
- Ability to update small areas without full re-deployment
- Scalable services on-demand

What Is Cloud Native Architecture ?

- **Designing, building, and running applications optimized for cloud computing environments. Scalable, resilient, and flexible applications for faster innovation and market responsiveness.**
- Use a **microservices architectures** that is decomposed into several independent services integrate with each other **through APIs** and **event-based messaging**, and each perform a specific function.
- **Cloud native** applications has **dynamic scaling**.
- **Container orchestrator** handles resource management, load balancing, scheduling restarts after an internal failure, **provisioning and deploying containers** to server cluster nodes.
- Some of the building blocks of cloud native have become mainstream, according to a **2021 survey done by CNCF**.
- “**Kubernetes adoption among the ever-expanding cloud native community is approaching 100%**”



Essential components of Cloud Native Architecture

Microservices

- Small, loosely-coupled, independently deployable services. Better scalability, faster development cycles, and more efficient resource utilization.

Containers

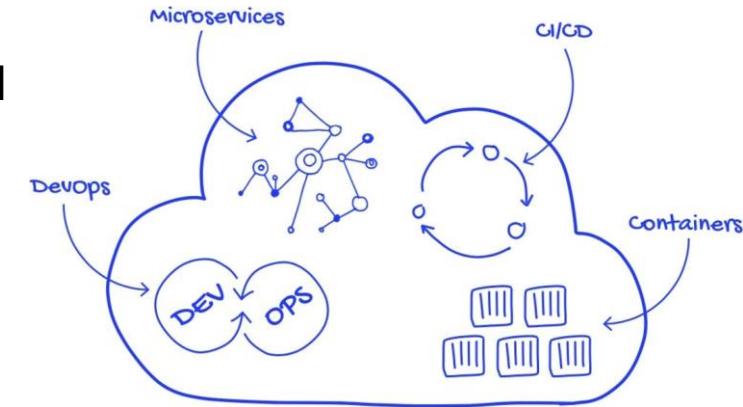
- Lightweight, portable runtime environments. Easier application management and deployment. Examples: Docker, containerd.

Orchestration

- Tools for managing containerized microservices. Automated scaling, rolling updates, and resource management. Example: Kubernetes.

Devops Practices

- Continuous Integration and Continuous Deployment (CI/CD), Infrastructure as Code (IaC). Streamlined development, testing, and deployment processes.



Essential components of Cloud Native Architecture - 2

Automation & CI/CD

- Infrastructure, deployment, scaling, and recovery processes. Uses CI/CD pipelines to automate the process of building, testing, and deploying code.

Scalability

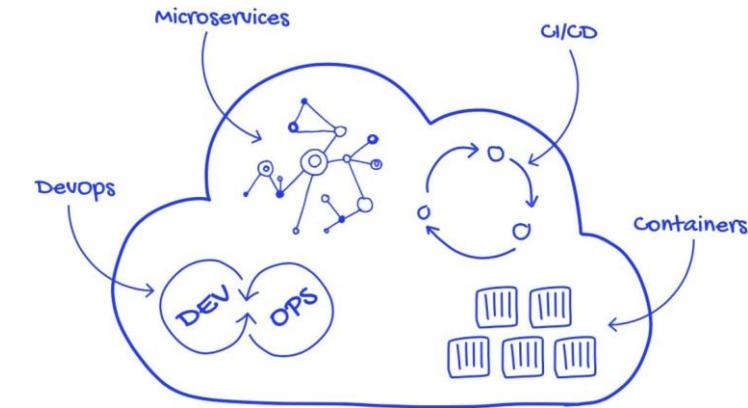
- Horizontal scaling for optimal resource usage and performance.

Resiliency

- Fault-tolerant and self-healing. Techniques: circuit breakers, retries, timeouts. High availability and minimized downtime.

Observability

- Built-in monitoring, logging, and tracing. Insights into performance, health, and behavior. Tools: Prometheus, Fluentd, Jaeger.



Cloud Native Architecture Design Principles

Designed as Loosely Coupled Microservices

- Develop applications as a collection of small, independent services that communicate using lightweight protocols, improving scalability and development efficiency.

Developed with Best-optimum Languages and Frameworks

- Utilize the most suitable language and framework for each microservice, allowing for a diverse and optimized application.

API-Centric Interaction and Collaboration

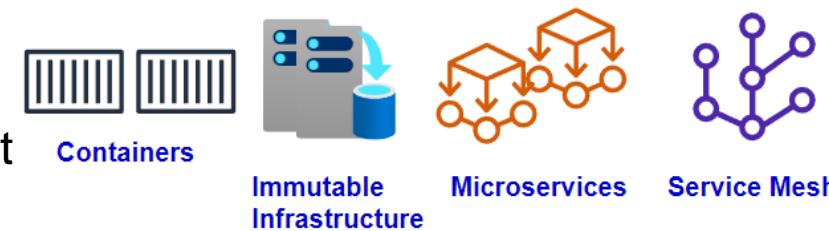
- Use lightweight APIs for communication between services, ensuring efficient collaboration and better performance.

Stateless and Massively Scalable

- Design applications to be stateless and scalable, storing state in external databases or entities for easy elasticity.

Elasticity and Dynamic Scaling

- Allow cloud-native applications to dynamically scale up or down according to resource requirements and usage spikes.



Cloud Native Architecture Design Principles - 2

Design for Resiliency

- Embrace failures by building applications that recover quickly and avoid downtime or data loss.

Polyglot Architecture

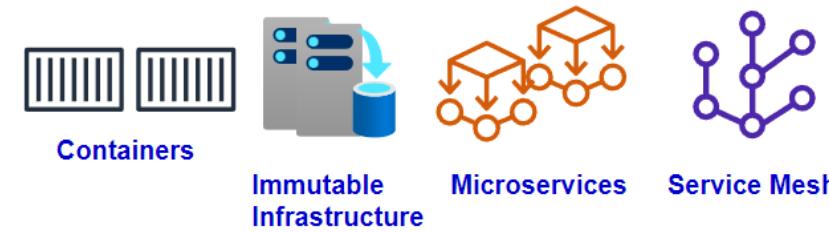
- Utilize the most appropriate language or technology for each component, considering team skillsets and time-to-market.

Packaged Lightweight Containers and Orchestration

- Package applications as lightweight containers using Docker and orchestrate deployment, scaling, and management with Kubernetes.

Immutable Infrastructure

- Servers for hosting cloud-native applications remain unchanged after deployment. By avoiding manual upgrades, immutable infrastructure makes cloud-native deployment a predictable process.



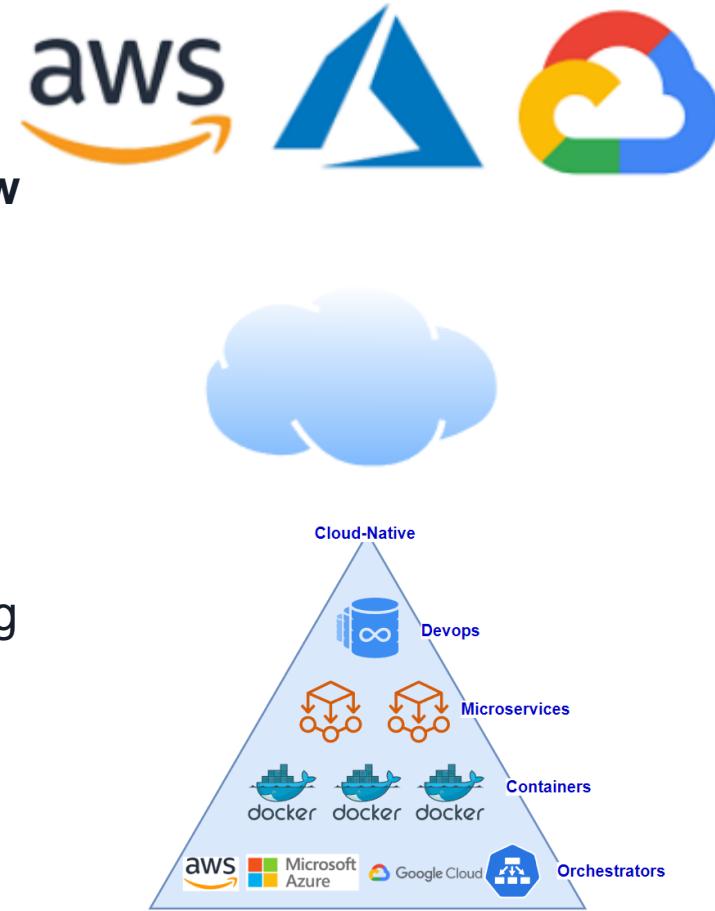
Benefits of Cloud-Native Architectures

- Easily **scale up or down based on demand** and allow apps to **scale horizontally**, making it easier to **handle increased workloads** and user traffic efficiently.
- **Rapid development, deployment, and iteration** of applications, allow businesses to **quickly respond to market changes** and customer needs, It release new features, and fix bugs more efficiently.
- Better **cost efficiency** through a **pay-as-you-go model** and optimized **resource utilization**.
- **Fault tolerance, and self-healing capabilities**, cloud-native architectures provide improved resilience and reliability, minimizing downtime.
- Optimal **resource allocation** and **utilization**, reducing infrastructure costs.
- Use the **most suitable languages, frameworks, and technologies** for different components, promoting innovation and adaptability.
- Easily moved between **different cloud providers** or environments, giving organizations more **flexibility** and **avoiding vendor lock-in**.



Challenges of Cloud-Native Architectures

- **Shift to microservices, containers, and orchestration tools** can introduce significant **complexity**, making it harder to manage and maintain applications.
- **Require a deep understanding** of various technologies, languages, and frameworks, which can be challenging for developers to acquire. Requires a **new set of skills and expertise**.
- Ensuring **data consistency, integrity, and security** across distributed microservices and cloud environments can be **challenging**.
- Distributed nature of cloud-native applications can introduce **new security vulnerabilities**, requiring organizations to invest in **robust security measures**.
- Specific cloud provider services or platforms can **lead to vendor lock-in**, making it difficult to switch providers or adopt a **multi-cloud strategy**.
- **Integrating cloud-native applications with existing legacy systems** can be complex and may require **significant effort**.
- **Monitoring and troubleshooting** distributed Cloud-Native Architectures may require **new tools, practices, and expertise**.



The Cloud Path of Legacy Applications – Modernization of Legacy Apps with Cloud-Native

- Organizations move to the cloud for the agility and speed applications. Able to set up thousands of servers (VMs) in the cloud in minutes. No single, one-size-fits-all strategy for migrating apps to the cloud.

Level 1: Infrastructure-Ready Applications in the Cloud

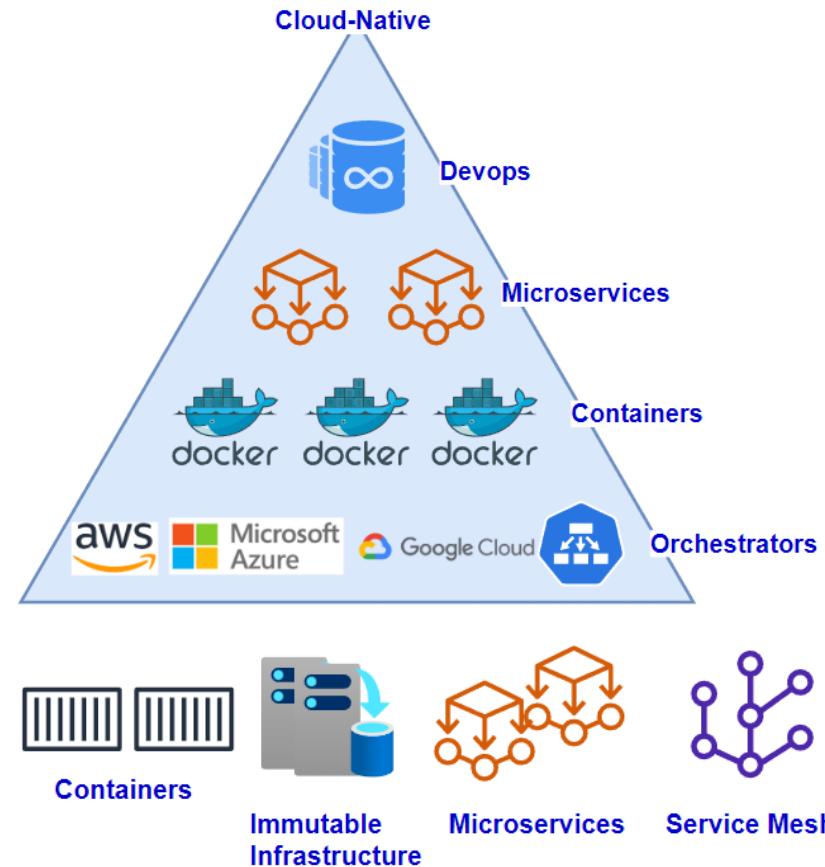
- Migrate or rehost existing on-premises applications to IaaS
- Minimal changes, retain original structure
- "Lift & Shift"

Level 2 - Cloud-Enhanced Applications

- Use modern cloud technologies (containers, cloud-managed services)
- Streamline DevOps processes
- Deploy containers on IaaS or PaaS
- Leverage cloud-managed services

Level 3 - Cloud-Native Applications

- Transition to PaaS computing platforms
- Implement cloud-native applications and microservices architecture
- Foster long-term agility and scalability
- May require new code or adapting applications



AWS Examples for Cloud Adoption to Cloud-Native Systems

- AWS and Azure offers set of services that we can use:

Level 1: Infrastructure-Ready Applications in the Cloud

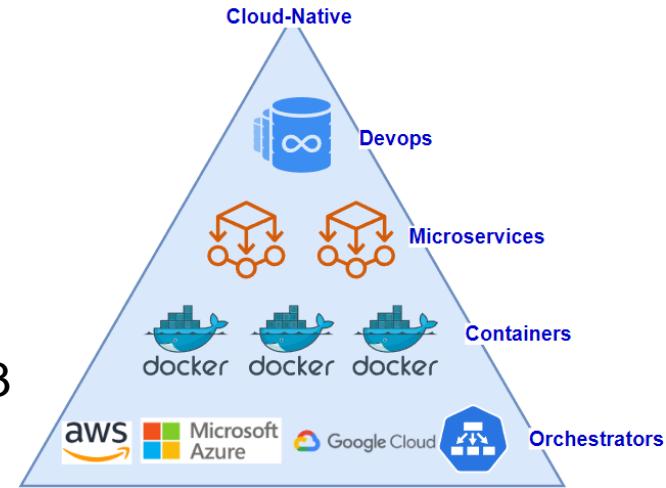
- AWS Cloud Adoption Framework (CAF) for Level 1

Level 2 - Cloud-Enhanced Applications

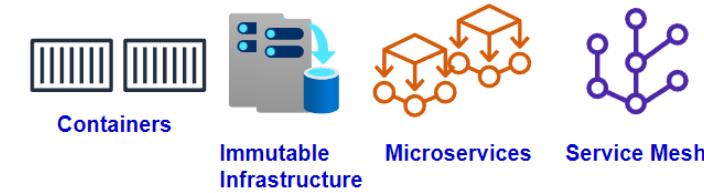
- AWS Application Migration Service & AWS Database Migration Service for Level 2

Level 3 - Cloud-Native Applications

- AWS EKS, ECS with Fargate, and AWS Serverless Services like Lambda for Level 3



- Choosing the Right Migration Approach
- Select a single approach or combine components from multiple approaches
- Applications can have a hybrid approach with on-premises and cloud components



Cloud-Native Fundamentals: The Conway's Law and 12 Factor App

Cloud-Native Fundamentals Patterns and Principles

Conway's Law

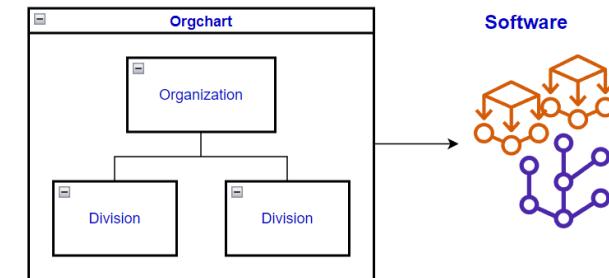
12 Factor App

The Conway's Law

- **Conway's Law** is an observation made by computer programmer **Melvin Conway** in 1967.
- **Design of a software system will reflect the organizational structure** of the team that built it.
- If an organization has **separate teams** for different parts of a software system, the resulting **software** will be **divided into corresponding parts** that communicate with each other.

Melvin E. Conway

- Any organization that designs a system (defined broadly) will produce a design whose structure is a **copy of the organization's communication structure**.
- The way teams within an organization are structured and communicate will have a direct **impact on the architecture** of the systems they build.



Martin Fowler

- It was originally described to me by saying that if a single team writes a compiler, it will be a **one-pass compiler**, but if the team is divided into two, then it will be a **two-pass compiler**.
- "Conway understood that software coupling is enabled and encouraged by **human communication**." -- Chris Ford

The impact of Conway's Law on Cloud-Native Microservices

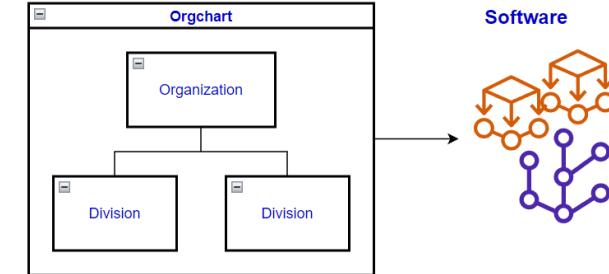
- Cloud-native apps rely on the principles of loose coupling and separation of concerns.
- Microservices are ideally correspond to the organization's structure and the way teams are set up.

Align Teams with Services

- Teams should be organized around specific microservices or functional domains to enable efficient communication and collaboration.
- Each team owns, develops, and maintains a single service or a group of closely related services.

Cross-functional Teams: Adapt Organizational structure

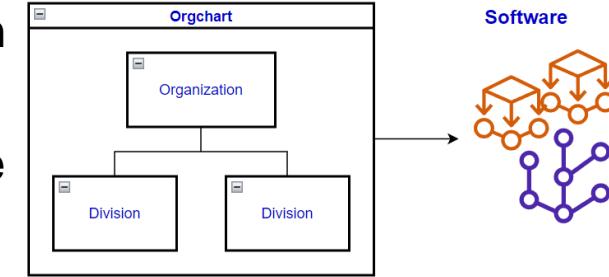
- Expertise to develop, deploy, and maintain a microservice. Promotes collaboration and reduces the need for excessive coordination between separate teams.
- Cross-functional teams working on individual microservices can lead to better-designed systems.



The impact of Conway's Law on Cloud-Native Microservices - 2

Communication and collaboration

- Services communicate through well-defined APIs, which allows for better collaboration between teams.
- Clear communication channels between services and teams, organizations build more resilient and adaptable systems.



Decentralize Decision-making

- Decentralized decision-making by empowering teams to choose the right tools, technologies
- Accelerates development, and is consistent with the polyglot nature of cloud-native architectures.

Promote Team Independence

- Minimize dependencies between teams by designing microservices with well-defined APIs and contracts.

Evolutionary design

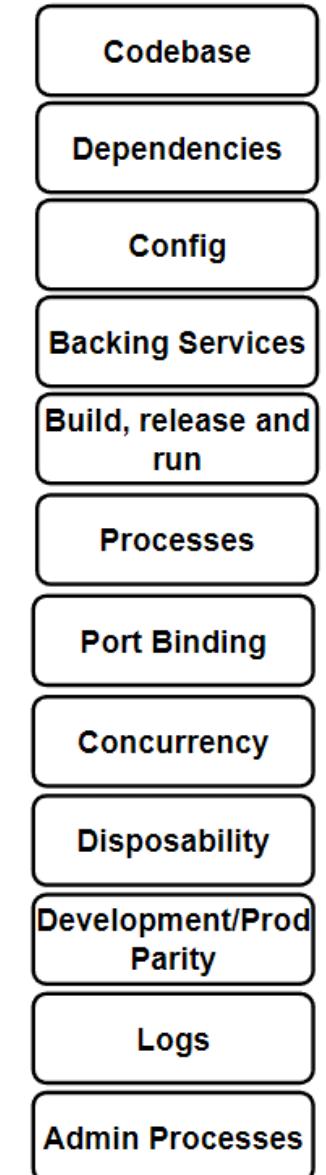
- Microservices architecture allows for the continuous evolution and improvement of a system.
- As teams and communication structures within an organization change over time.

12-Factors - The Twelve-Factor Application

- How would you design a Cloud-native architectures ? What are patterns, principles and best practices that we should follow when designing Cloud-native ?
- Twelve-Factor Application principles: set of principles and practices that developers follow to create cloud-native applications.

What is The Twelve-Factor Application ?

- Methodology for building modern, scalable, and maintainable software-as-a-service applications.
- Created by Heroku, to provide a set of best practices and guidelines for designing and developing apps that can be easily deployed, scaled, and managed in the cloud.
- Designed to work well with modern cloud-native architectures like microservices and container-based deployments.



12-Factors - The Twelve-Factors

1 Codebase

- Single code repository, and multiple deployments should be derived from that codebase.

2 Dependencies

- Dependencies should be Explicitly declared and isolate dependencies, enabling applications to have a controlled and predictable environment.

3 Config

- Store configuration information separately from the code, flexibility in deployment environments without changing the code.

4 Backing services

- Services treated as attached resources and attached and detached by the execution environment.

5 Build, release, run

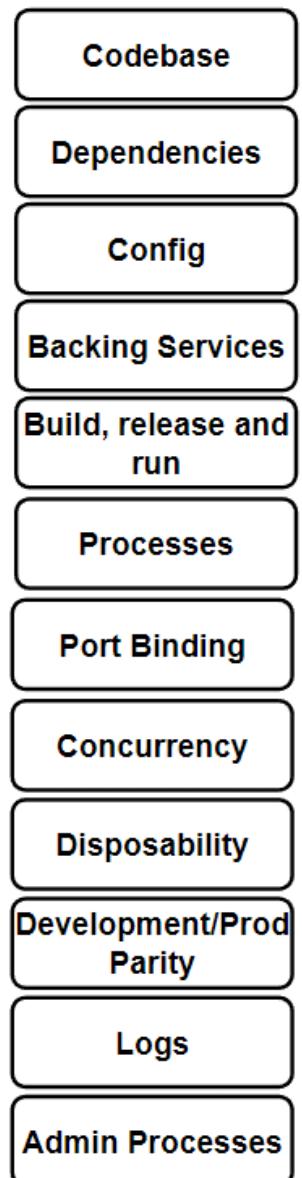
- Separate the build, release, and run stages to ensure a consistent and reliable deployment process.

6 Processes

- Deployed as one or more stateless processes with persisted data stored on backing service.

Mehmet Ozkaya

62



12-Factors - The Twelve-Factors - Continue

7 Port binding

- Self-contained and expose services via port binding that make themselves available to other services by specified ports.

8 Concurrency

- Design to scale horizontally through separate processes, improving load distribution and redundancy.

9 Disposability

- Start quickly and shut down gracefully, optimizing resource usage and minimizing downtime in dynamic cloud environments.

10 Dev/Prod parity

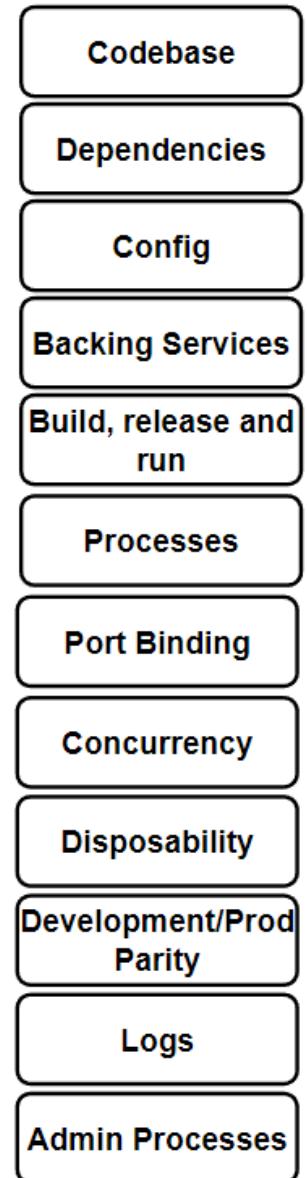
- All environments should be as similar as possible.

11 Logs

- Produce logs as event streams and leave the execution environment to aggregate.

12 Admin Processes

- Any needed admin tasks should be kept in source control and packaged with the application.



Additional Factors for Cloud-Native Apps

13 API First – Well defined APIs

- Expose APIs that make everything as a service. Design APIs that will be consumed by a front-end client, gateway, or another service.

14 Telemetry

- Deep visibility into your cluster and its behavior includes the collection of monitoring, domain-specific, and health/system data.

15 Authentication/ Authorization

- Implement identity server from the beginning of project, consider RBAC (role-based access control) features available in public clouds.

16 Resilience

- Fault-tolerance and Resilience to any exceptions with self-healing mechanisms.

Additional Factors

API First

Telemetry

Authentication /
Authorization

Resilience

I. Codebase - Twelve-Factor Application

Maintain a single codebase per microservice or application

- Easier to understand, develop, and deploy
- Simplifies dependency management and testing

Store in a separate repository with version control

- Supports collaboration among developers
- Allows tracking changes and maintaining history (e.g., Git, Subversion, Mercurial)

Deploy the same codebase to multiple environments

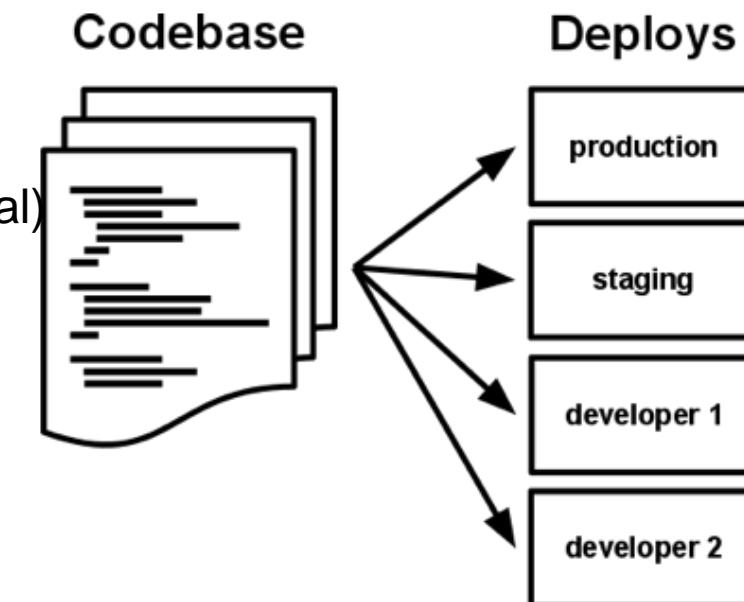
- Consistency across development, staging, and production environments
- Manage different configurations or backing services without changing code

Manage features or changes using branches or feature flags

- Keep branches and flags to a minimum for consistency
- Merge changes back into the main branch quickly

Benefits of a single codebase

- Fewer bugs and easier debugging
- Smoother development process overall



II. Dependencies - Twelve-Factor Application

Managing dependencies:

- Ensures applications are self-contained, portable, and maintainable
- Each microservice isolates and packages its own dependencies

Explicitly declare and isolate dependencies

- Use a manifest or configuration file to declare all dependencies
- Avoid conflicts and ensure components run with required versions

Use dependency management tools

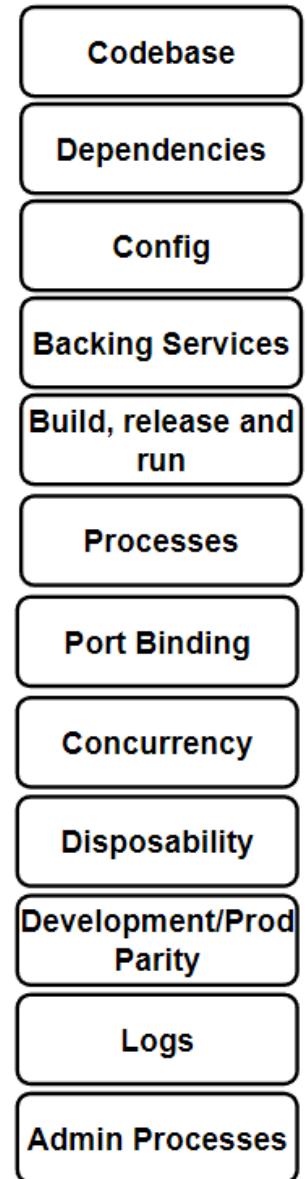
- Simplify dependency declaration and management (e.g., NPM for JavaScript, Maven for Java, pip for Python)
- Automate installation, versioning, and updating of dependencies

Package dependencies with the application

- Enhance portability and consistency across environments

Example: Java Dependency Management

- Popular tools: Maven and Gradle
- Declare dependencies in Maven's pom.xml file or Gradle's settings.gradle file
- Build tool ensures dependencies are installed, not the developer



III. Config - Twelve-Factor Application

Managing configuration data

- Configuration data (db connection string, api keys) could be vary between environments
- Separate configuration data from application code

Separation of concerns

- Maintain clear distinction between application logic and environment-specific data

Environment variables

- Easy to manage and modify across environments

Avoid committing sensitive data

- Prevent exposure or leakage by keeping sensitive data out of version control

Config management tools

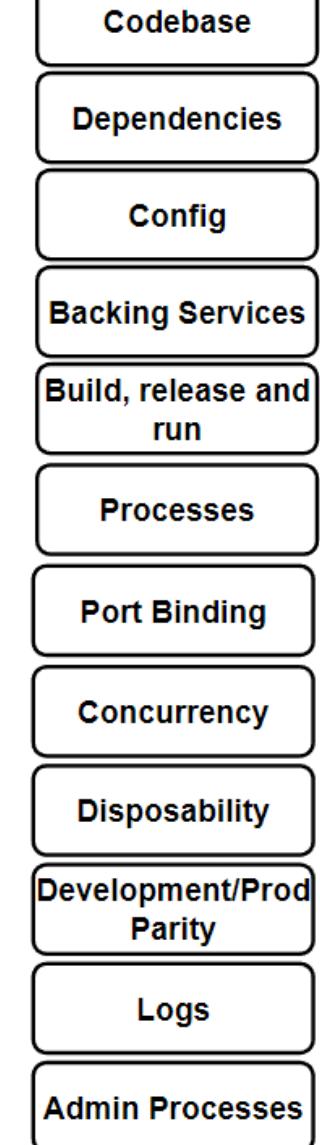
- Kubernetes ConfigMaps, HashiCorp Vault, AWS Parameter Store, etc.

Runtime configurability

- Adapt to different environments without rebuild or redeployment

Example: External Configuration Files

- Inject configuration values into the microservice's runtime environment
- Use independent configuration files (e.g., Kubernetes manifest file, docker-compose.yml file)



IV. Backing services - Twelve-Factor Application

Treating backing services as attached resources

- Backing services include databases, messaging systems, cache, file storage systems, or external APIs
- Expose backing services via addressable URLs and connect with APIs
- Decouple resources from the application for greater flexibility and adaptability

Key practices

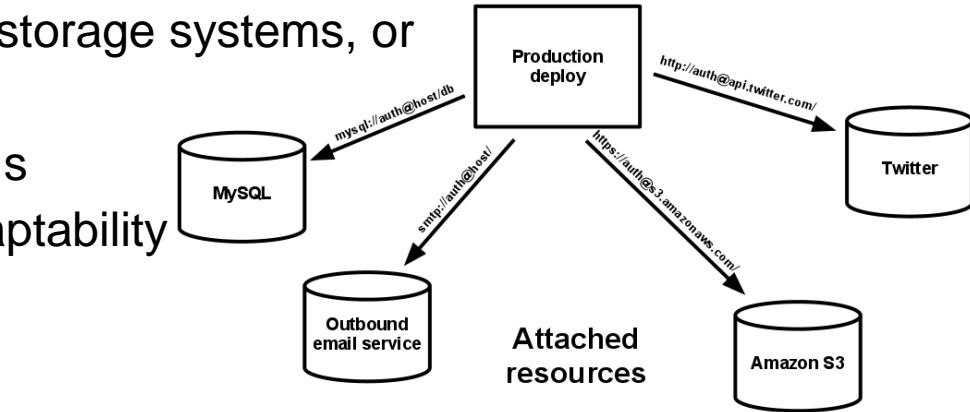
- Loosely couple application with backing services
- Switch between different service providers or instances without significant code changes
- Design application to be flexible and adaptable to changes in backing services
- Use abstractions or adapters for communication with different service providers

Example: Switching Database Systems

- Replace local PostgreSQL database with a cloud-hosted database by changing the URL and credentials without altering application code
- Cloud-Native Pillars: Backing services, will be covered in more detail in future sections

Use Case: Serverless Database Services

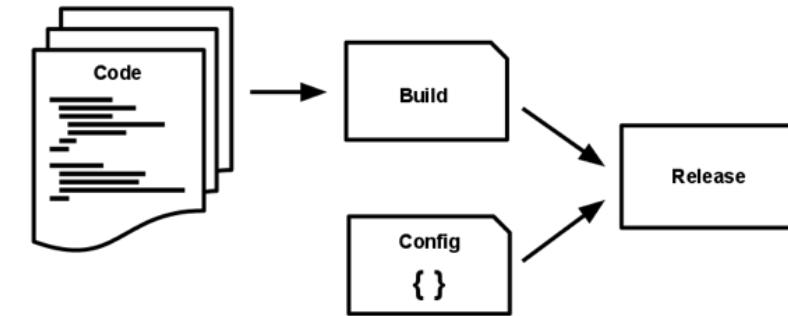
- Companies like Upstash provide managed Redis and Kafka clusters
- Pay-per-use pricing model, allowing developers to focus on more important tasks



V. Build, release, run - Twelve-Factor Application

Separating build, release, and run stages

- Promotes consistency, predictability, and reliability in the deployment process
- Enables rapid and confident deployment using CI/CD systems



Key concepts for each stage

- Build: Convert source code into an executable package or artifact (e.g., binary, container image)
 - Automate and make reproducible across environments
- Release: Combine build artifact with configuration data for the target environment
 - Create immutable and uniquely identifiable releases
 - Automate the process and support rollbacks
- Run: Launch the release in the target environment (e.g., development, staging, production)
 - Enable easy rollback to a previous release in case of issues

Example: Java Application Lifecycle

- Build: Package dependencies into a WAR or JAR file
- Release: Label releases with unique IDs for rollback and auditing
- Containerize application with tagged numbers (e.g., Docker images)
- Run: Launch application using cloud-based tooling like containers or serverless platforms (e.g., EKS, AKS, AWS Fargate, Azure Container Apps)

VI. Processes - Twelve-Factor Application

Stateless processes

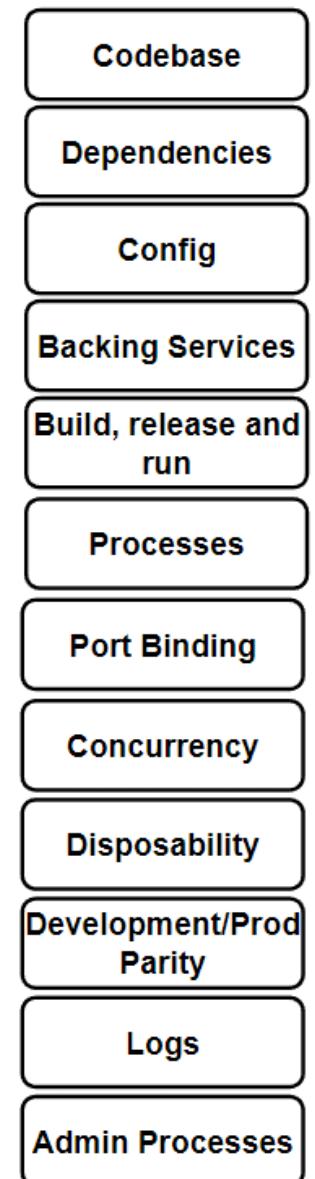
- Application should not rely on persistent state stored in memory or local storage
- Store required state information in a backing service (e.g., database, caching system)

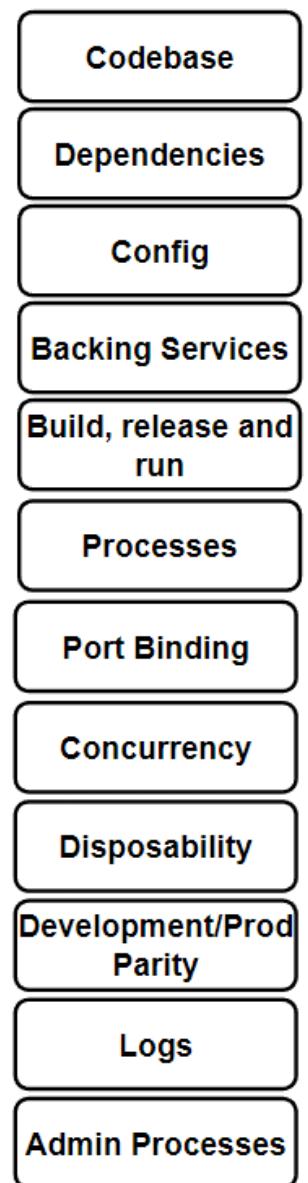
Key concepts

- Stateless processes enable horizontal scaling and independence of application instances
- Each process should be self-contained and not rely on shared resources (e.g., local storage, in-memory caches)
- Ensures that each process can be scaled or replaced without impacting others

Example: Cloud-native microservices architecture

- Run each microservice in its own process, isolated from others
- Externalize required state to a backing service (e.g., distributed cache, data store)





VII. Port binding - Twelve-Factor Application

Port binding concept

- Application exposes its service on a specific port for access by other applications/services
- Enables application to be a standalone, self-contained service

Key benefits

- More reliable than using domain names and IP addresses, which can change dynamically
- Easier to manage and avoid conflicts with port forwarding

Example: Cloud-native microservices architecture

- Each microservice should expose its interfaces and functionality on its own port
- Provides isolation from other microservices

VIII. Concurrency - Twelve-Factor Application

Concurrency principle

- Design applications to run as individual processes
- Handle multiple tasks or requests concurrently
- Efficiently utilize resources and scale horizontally

Scaling out vs. scaling up

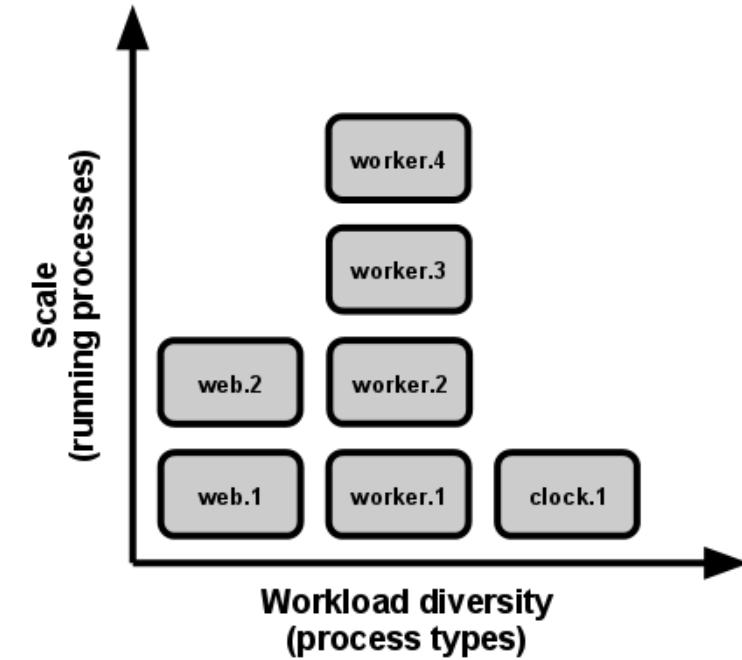
- Focus on horizontal scaling (scaling out) instead of increasing resources on a single machine (scaling up)
- Distribute workload across multiple identical processes or instances

Example: Cloud-native microservices architecture

- Break application into smaller, independent processes
- Start, terminate, and replicate processes independently for easier horizontal scaling

Tools for managing concurrency

- Kubernetes: Horizontal Pod Autoscaler adjusts the number of pods based on observed CPU utilization or custom metrics
- Facilitates elastic scaling and workload adjustments



IX. Disposability - Twelve-Factor Application

Disposability principle

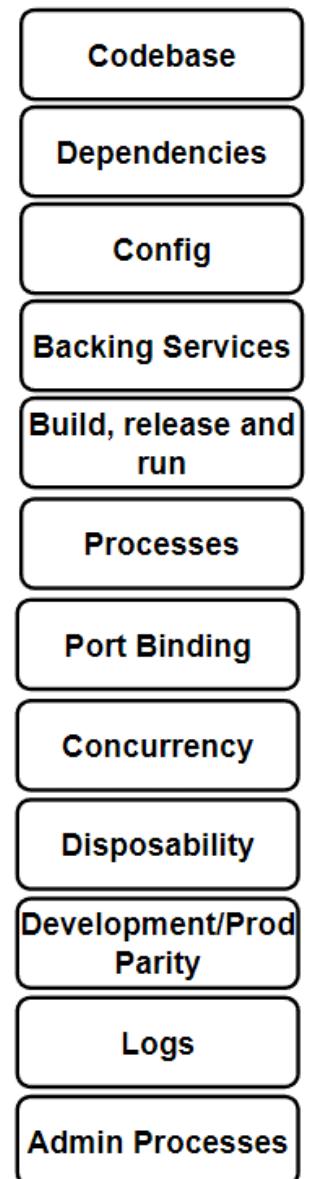
- Building apps that can quickly start, gracefully stop, and robustly handle unexpected terminations, ensuring resiliency and adaptability in cloud.

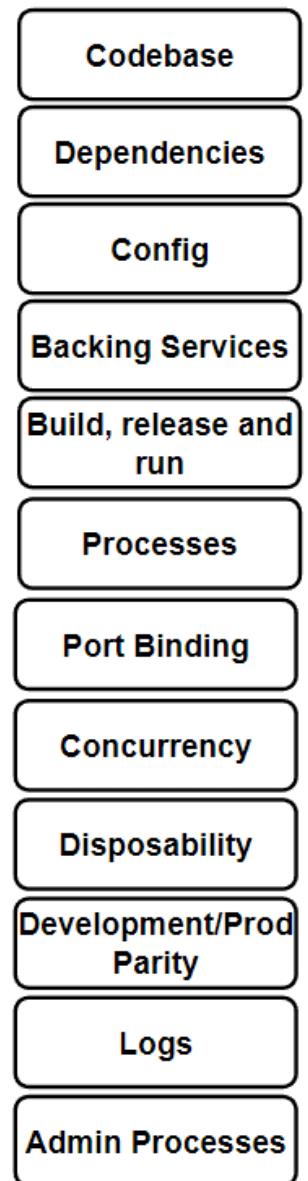
Key aspects of Disposability

- **Fast startup:** Minimize time to become available and responsive
- **Graceful shutdown:** Finish ongoing tasks, release resources, and perform cleanup before exiting
- **Disposable processes:** Treat instances as disposable resources that can be replaced or removed

Example: Cloud-native Microservices Architecture

- Docker containers and orchestrators inherently satisfy disposability requirements
- Rapid scaling, deployment, release, and recovery rely on fast startup and graceful shutdown
- Processes must start and stop rapidly to handle high traffic and dynamic workloads
- Minimize downtime and denied requests during application startup





X. Dev/prod parity - Twelve-Factor Application

Dev/prod parity principle

- Maintaining consistency between development, staging, and production environments to minimize errors and bugs, streamline development, and ensure consistent application.
- The development environment should be as similar as possible to the production environment.

Challenges caused by inconsistent environments

- Reduce likelihood of errors and bugs arising from environmental differences
- Difficulties replicating issues or testing new features
- Delays in development and increased risk of errors or downtime during deployment

Utilize containerization tools for consistency

- Docker and similar tools provide a uniform environment for running code
- Package application, dependencies, configuration, and resources into a single, self-contained unit

XI. Logs - Twelve-Factor Application

Logs principle

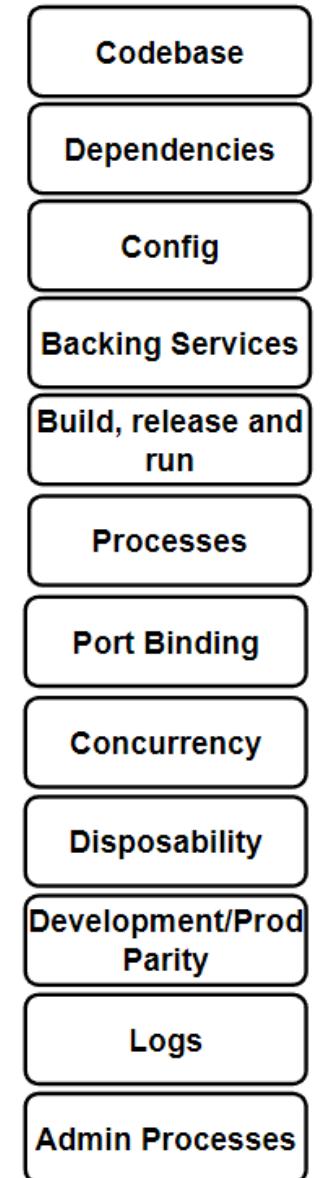
- Efficient log handling is crucial for cloud-native applications.
- Treat log data as continuous streams for various consumers
- Segregate routing of log data from processing, various consumers can focus on specific logs

Microservices log management

- Manage logs as event streams using an event aggregator
- Direct log data to data-mining and log management tools (e.g. ELK, Azure Monitor, Splunk)

Event stream model for logs

- View logs as time-ordered sequences of events produced by an application
- Output log entries to stdout (standard output) and stderr (standard error)
- Use tools like ELK stack (Elasticsearch, Logstash, Kibana) or Splunk for log capture, processing, and analysis



XII. Admin processes - Twelve-Factor Application

Admin process management

- Should be managed separately, run in the same environment as the application, and executed as one-off tasks to ensure smooth operation, maintenance, and greater resiliency for cloud-native applications.

One-off task execution

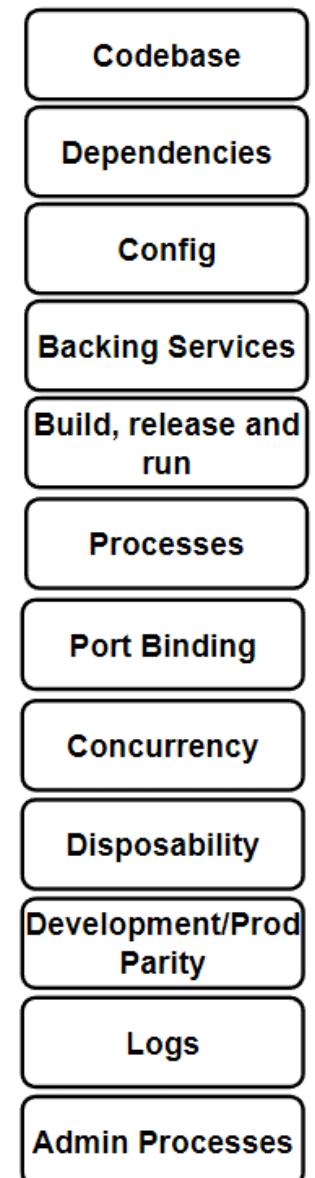
- Execute admin processes separate from the application's regular processes
- Prevent interference with normal application operation

Consistent environment for admin processes

- Run admin tasks in the same environment as the application
- Use the same codebase and configuration to reduce inconsistencies or errors

Examples in cloud-native migration of databases

- Run admin processes as Kubernetes tasks
- Focus microservices on business logic
- Enable safe debugging and admin of production applications
- Enhance resiliency for cloud-native applications



Cloud-Native Deep Dive - Landscape, TrialMap and Pillars

Cloud-Native Deep Dive: Cloud-Native Landscape, TrialMap, Pillars

What is CNCF ?

What Projects are driving with CNCF ?

What are CN Pillars that we can follow during the course ?

Cloud Native Computing Foundation (CNCF)

What is Cloud Native Computing Foundation (CNCF)

- CNCF fosters collaboration, supports cloud-native open-source projects
- Non-profit organization under the Linux Foundation
- Promotes and advances adoption of cloud-native technologies
- Collaborates with major corporations like Google, IBM, Intel, Box, Cisco, and VMware



Role of CNCF

- Creating sustainable ecosystems for cloud-native software
- Improving developer experience
- Hosting influential open-source projects (e.g., Kubernetes, Prometheus, Helm, Envoy, gRPC)
- Promoting open-source ecosystem to avoid vendor-lock-in

Why CNCF is needed

- Companies need to adopt a software-centric approach
- Examples: Airbnb's impact on the hospitality industry
- Cloud-native approaches allow for faster, more agile business software applications
- Facilitates collaboration between IT and business teams for better services and competitiveness

Quick Tour on Cloud Native Computing Foundation (CNCF)

Cloud Native Computing Foundation (CNCF) website

- <https://www.cncf.io/>

Quick Tour of CNCF

- About
- Projects
- Case Studies



Cloud Native Computing Foundation (CNCF) Landscape

CNCF Landscape

- Interactive Overview of open-source projects, products, and services
- Helps understand the cloud-native ecosystem and accelerate adoption
- Helping developers and organizations navigate the ecosystem, identify the right tools, and accelerate cloud-native adoption.

Cloud Native Landscape Categories

- App Definition & Development
 - Tools for data storage, building, and deploying applications
- Orchestration & Management
 - Tools for orchestrating and managing containers, applications, and resources
- Runtime
 - Tools for running containers in cloud-native environments
- Provisioning
 - Tools for foundational infrastructure and supporting technologies
- Observability & Analysis
 - Monitoring tools and alerting systems
- Platforms
 - Bundles multiple functionalities and tools for easy cloud-native adoption



Visit the CNCF Landscape:

- <https://landscape.cncf.io/>

Complicated with CNCF Projects ? - Funny Break

CNCF Landscape

- Deep Dive into CNCF Landscape Categories and see sub categories and projects.
- Most probably we have only familiar with Kubernetes, Docker, and so on.

Funny meme about Cloud Native Landscape Categories

- <https://twitter.com/memenetes/status/1637861860039876624>



<https://twitter.com/memenetes/status/1637861860039876624>



Not Get Overwhelmed with Cloud-Native Tools

Focus on Our Goals

- Have a clear understanding of what we are trying to achieve.
 - Learn by problem and solve this problem with cloud-native tools for our use cases.

Start with Core Concepts

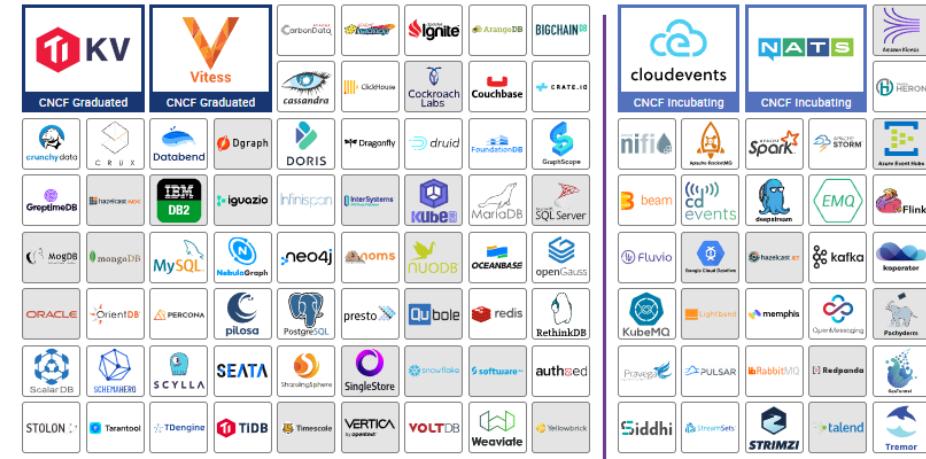
- Before exploring specific tools, have a good understanding of the core concepts such as containers, orchestrators, microservices, and CI/CD.

Adopt Incrementally

- Start with a few fundamental tools and get comfortable with them.
 - Gradually integrate more tools as you understand how they fit into your workflow.

Use the CNCF Trail Map

- CNCF provides a Trail Map which is a recommended path for adopting cloud-native technologies.



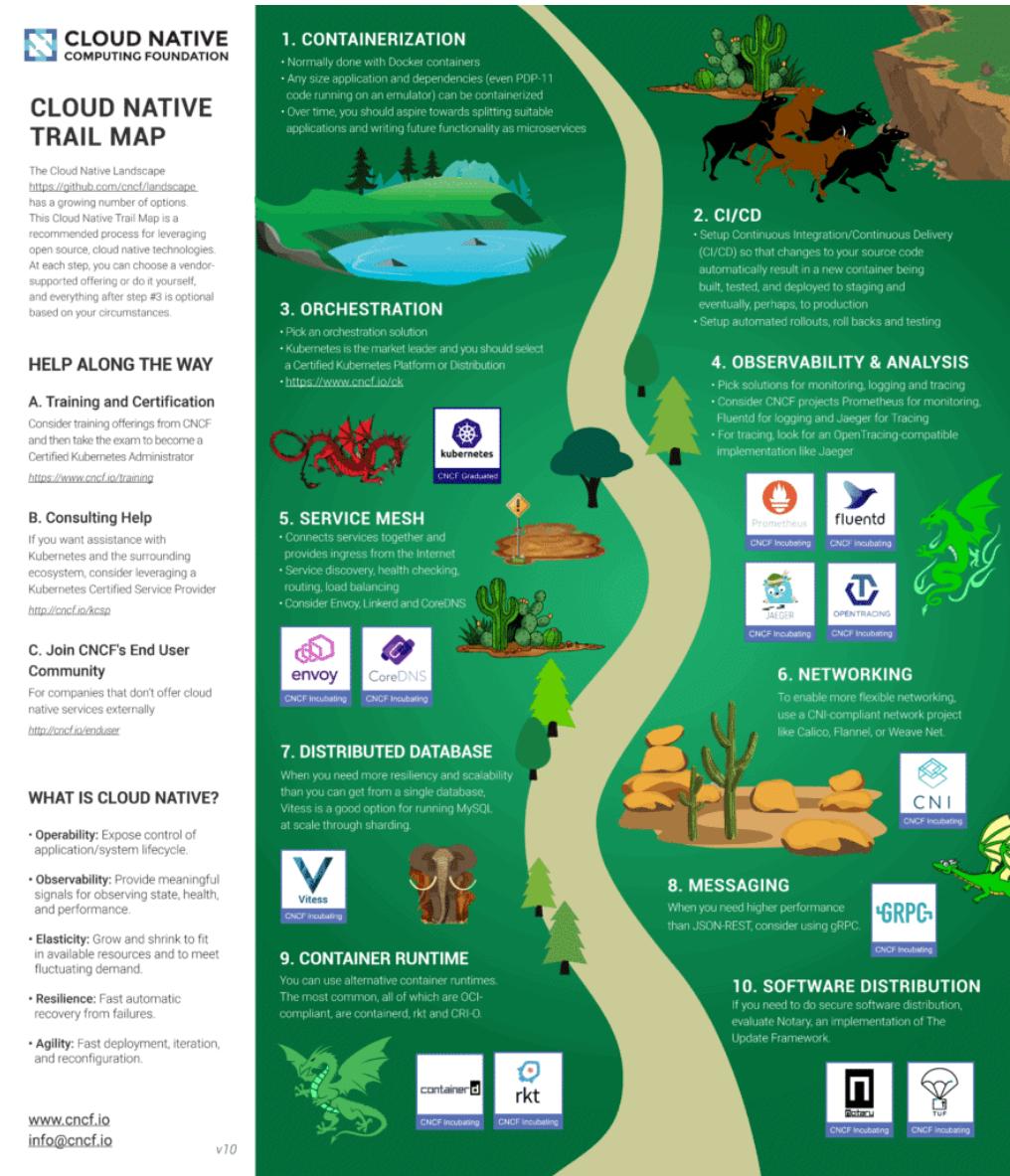
CNCF Trial Map

CNCF Cloud Native Trail Map

- Graphical guide created by the CNCF to help organizations and individuals navigate their cloud-native journey
- Recommended path for adopting cloud-native technologies, outlining the key steps, projects, and tools
- Useful starting point for those who are new to the cloud-native ecosystem or those looking to adopt cloud-native best practices.

Open Trail Map

- https://raw.githubusercontent.com/cncf/trailmap/master/CNCF_TrailMap_latest.png



<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>

The Four Pillars of Cloud-Native Applications (Minimum Required)

Microservices

- Small, loosely-coupled, independently deployable services
- Faster development, improved scalability, efficient resource utilization

Containers

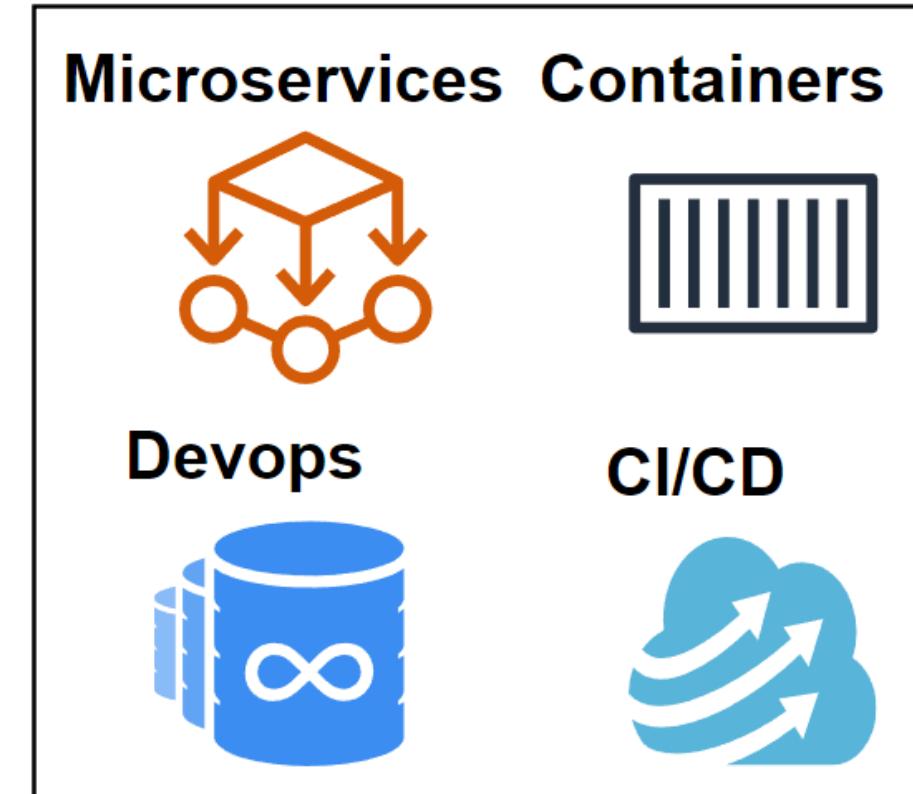
- Package and deploy individual microservices
- Lightweight, portable, consistent runtime environments

DevOps

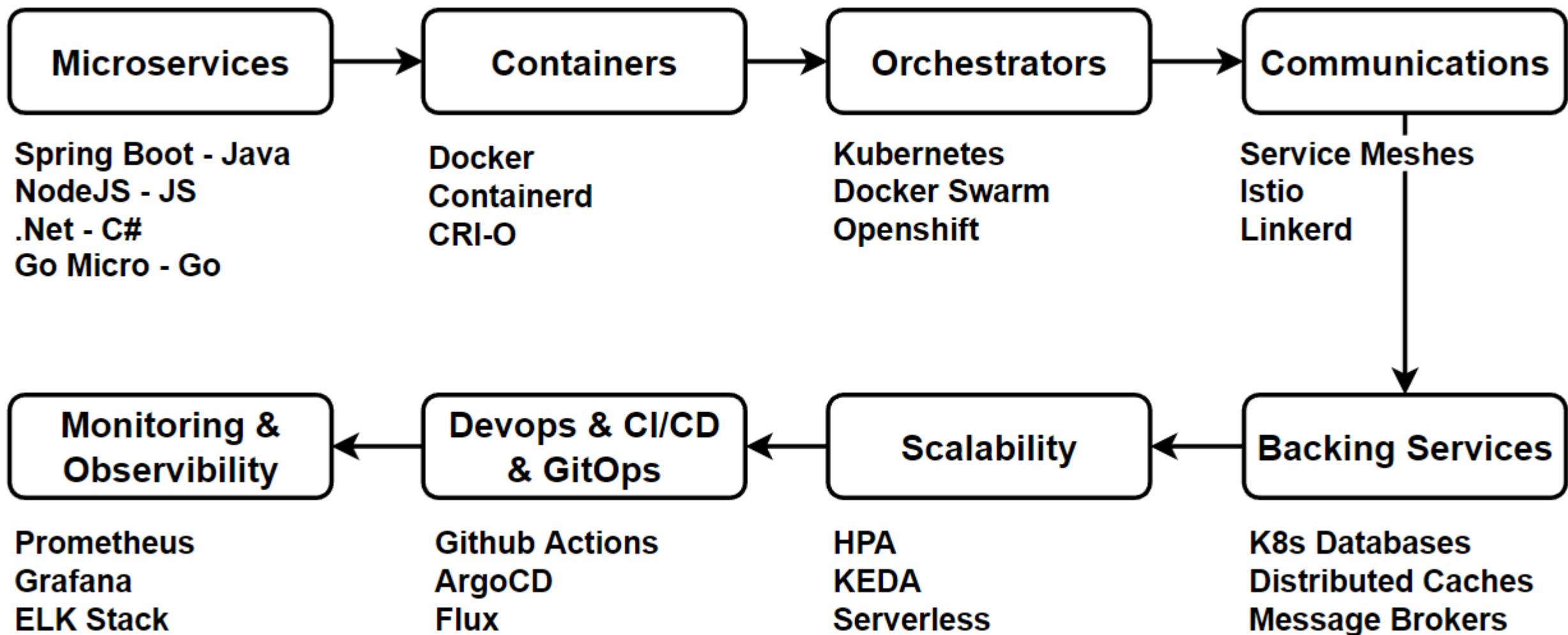
- Bridge the gap between development and operations teams
- Foster collaboration, shared responsibility
- Automation, Infrastructure as Code (IaC)

CI/CD

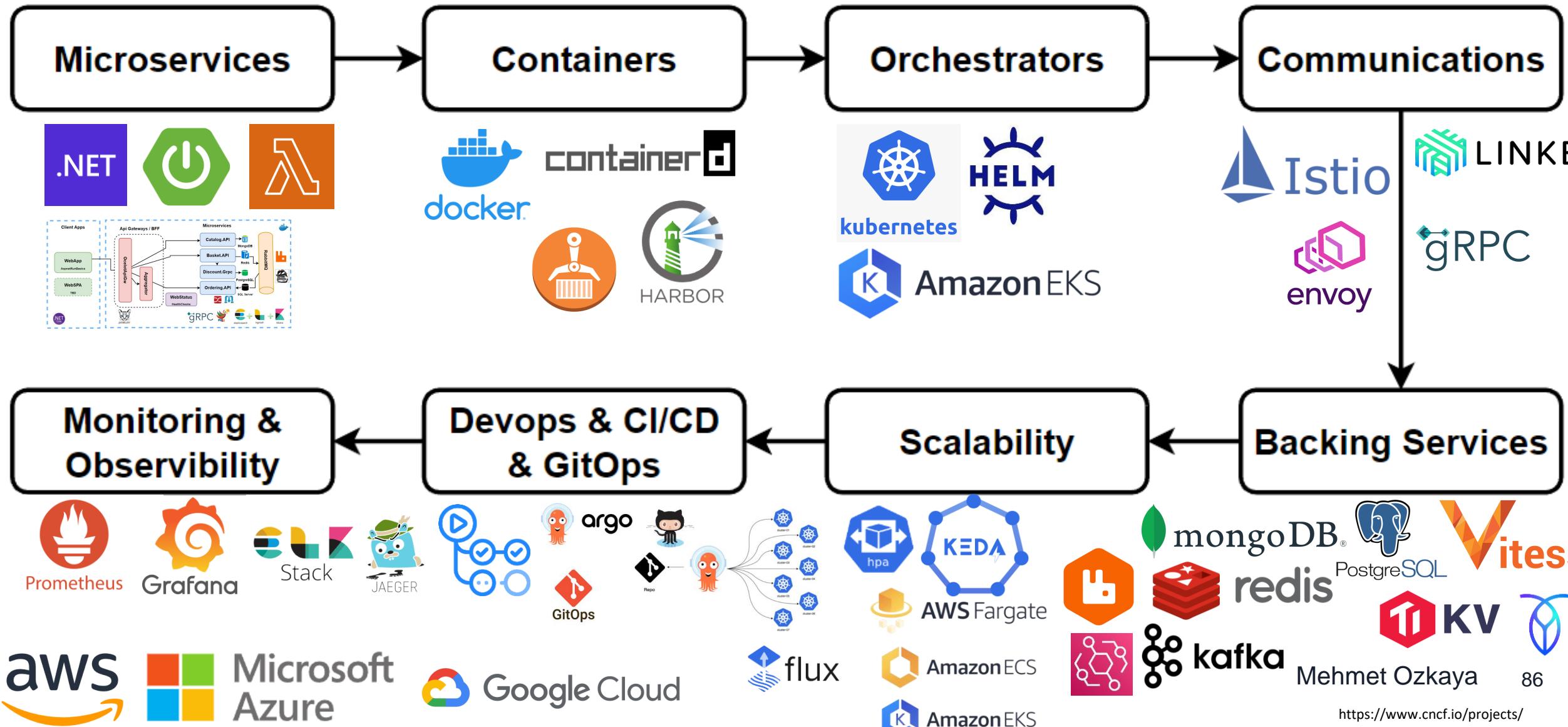
- Continuous Integration and Continuous Delivery pipelines
- Automate build, test, and deployment processes
- Faster, more reliable software releases



Cloud-Native Pillars Map – The Course Section Map



Cloud-Native Tools and Technologies that we will practice



Cloud-Native Pillar1: Microservices

What are Microservices ?

Why we should use Microservices Architecture for Cloud-Native Applications ?

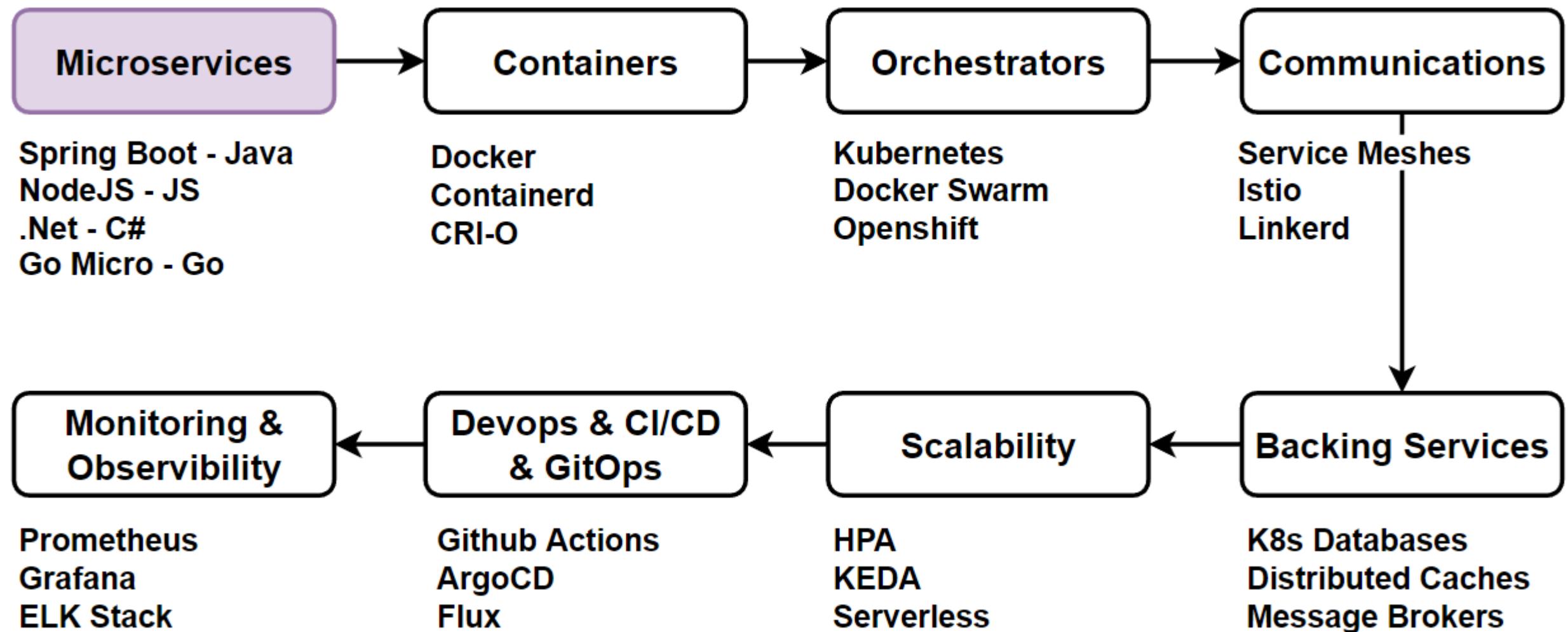
Benefits and Challenges of Microservices Architecture.

Design our E-Commerce application with Microservices Architecture

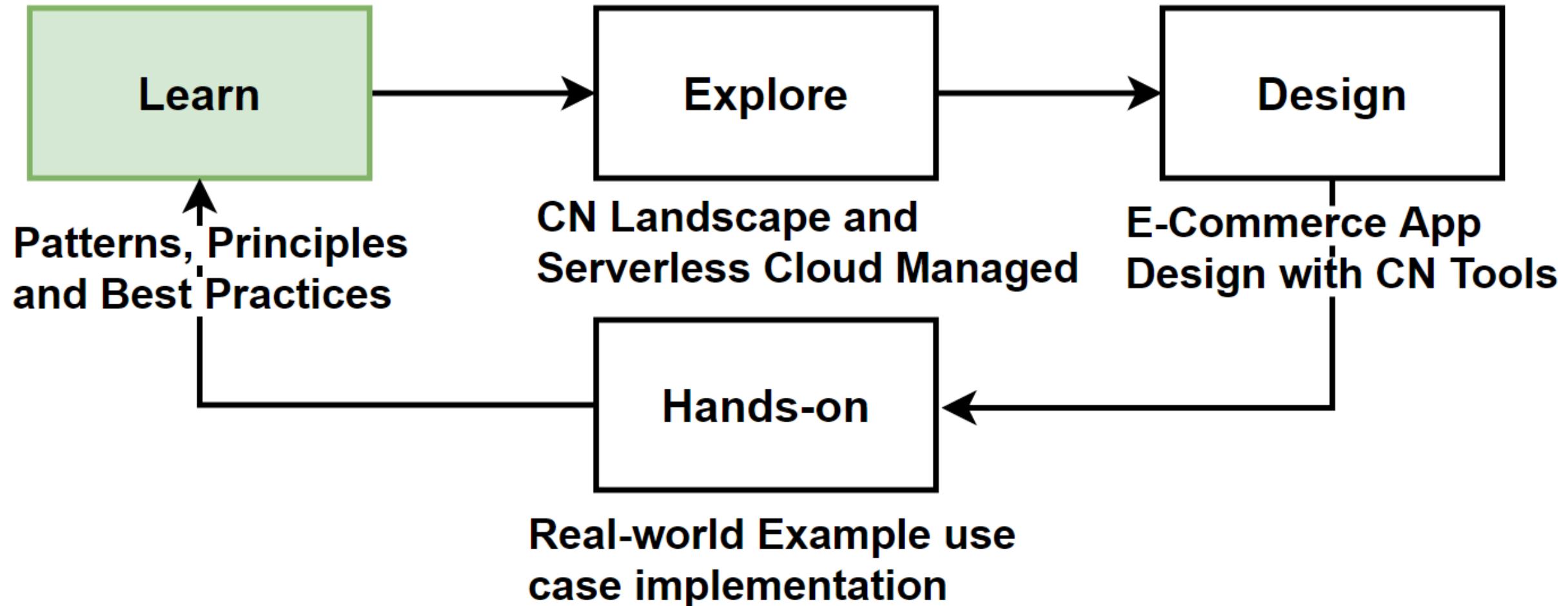
Implement Hands-on Development of Microservices

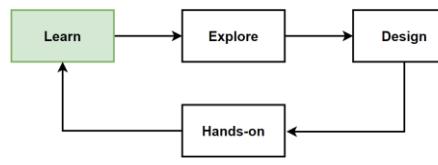
Mehmet Ozkaya

Cloud-Native Pillars Map – The Course Section Map



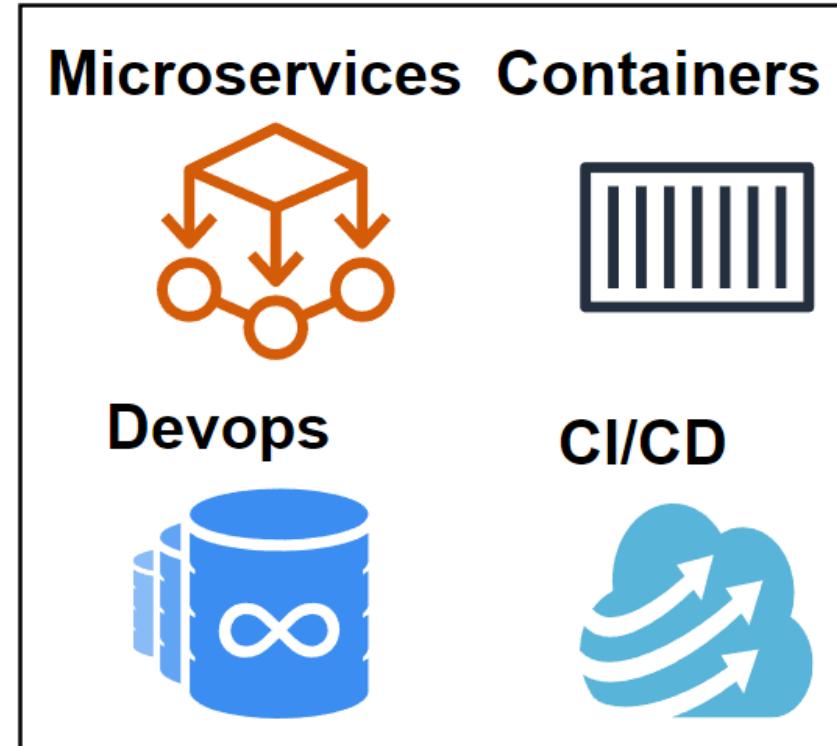
Way of Learning – The Course Flow

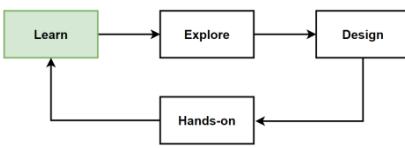




Learn: The First Pillar - Microservices

- What are Microservices ?
- What is Microservices Architecture ?
- Microservices Characteristics
- Benefits of Microservices Architecture
- Challenges of Microservices Architecture
- When to Use Microservices Architecture - Best Practices
- When Not to Use Microservices - Anti-Patterns of Microservices
- Monolithic vs Microservices Architecture Comparison
- The Database-per-Service Pattern - Polygot Persistence
- Design the Architecture - Microservices Architecture - First Iteration





Where «Microservices» in 12-Factor App and CN Trial Map

Twelve-Factor App

- Codebase
- Dependencies
- Config
- Backing Services
- Build, release and run
- Processes
- Port Binding
- Concurrency
- Disposability
- Development/Prod Parity
- Logs
- Admin Processes



CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape <https://github.com/cncf/landscape> has a growing number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator <https://www.cncf.io/training>

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider <http://cncf.io/ksp>

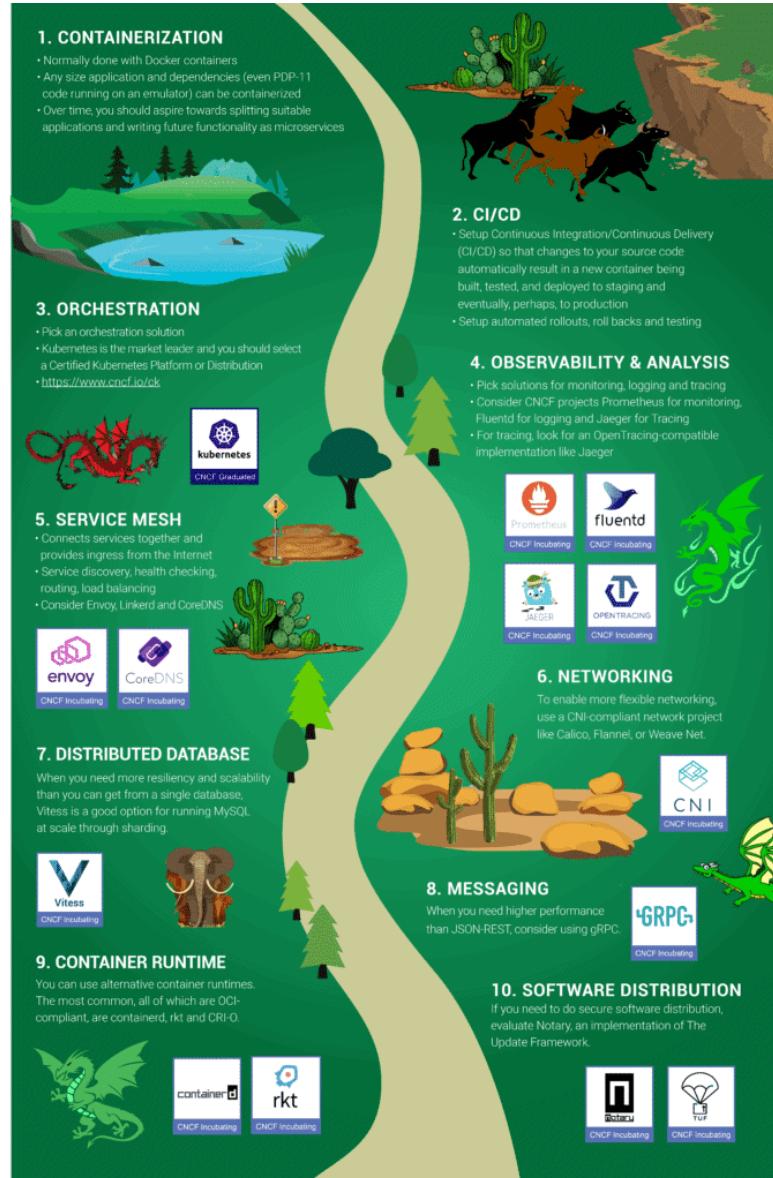
C. Join CNCF's End User Community

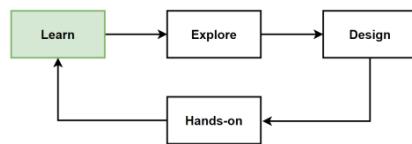
For companies that don't offer cloud native services externally <http://cncf.io/enduser>

WHAT IS CLOUD NATIVE?

- **Operability:** Expose control of application/system lifecycle.
- **Observability:** Provide meaningful signals for observing state, health, and performance.
- **Elasticity:** Grow and shrink to fit in available resources and to meet fluctuating demand.
- **Resilience:** Fast automatic recovery from failures.
- **Agility:** Fast deployment, iteration, and reconfiguration.

www.cncf.io
info@cncf.io





12-Factor App - Microservices

Codebase

- Separate process for each microservice
- Better scalability, fault tolerance, resilience

Configuration

- Configuration stored in the environment
- Unique configuration needs for each microservice

Processes

- Separate process for each microservice
- Better scalability, fault tolerance, resilience

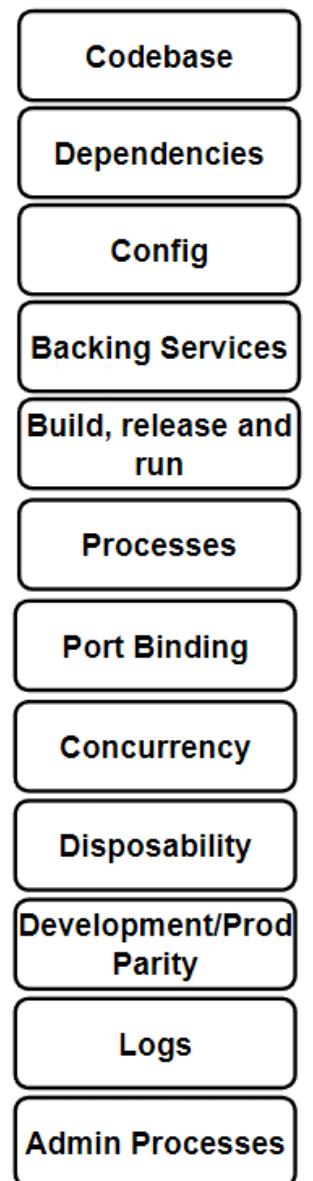
Port binding

- Each microservice binds to a specific port
- Ensures isolation and independent scalability

Disposability

- Quick and easy start-stop processes
- Improved scalability, fault tolerance, resilience

Twelve-Factor App



Cloud-native Trial Map - Microservices

Containers

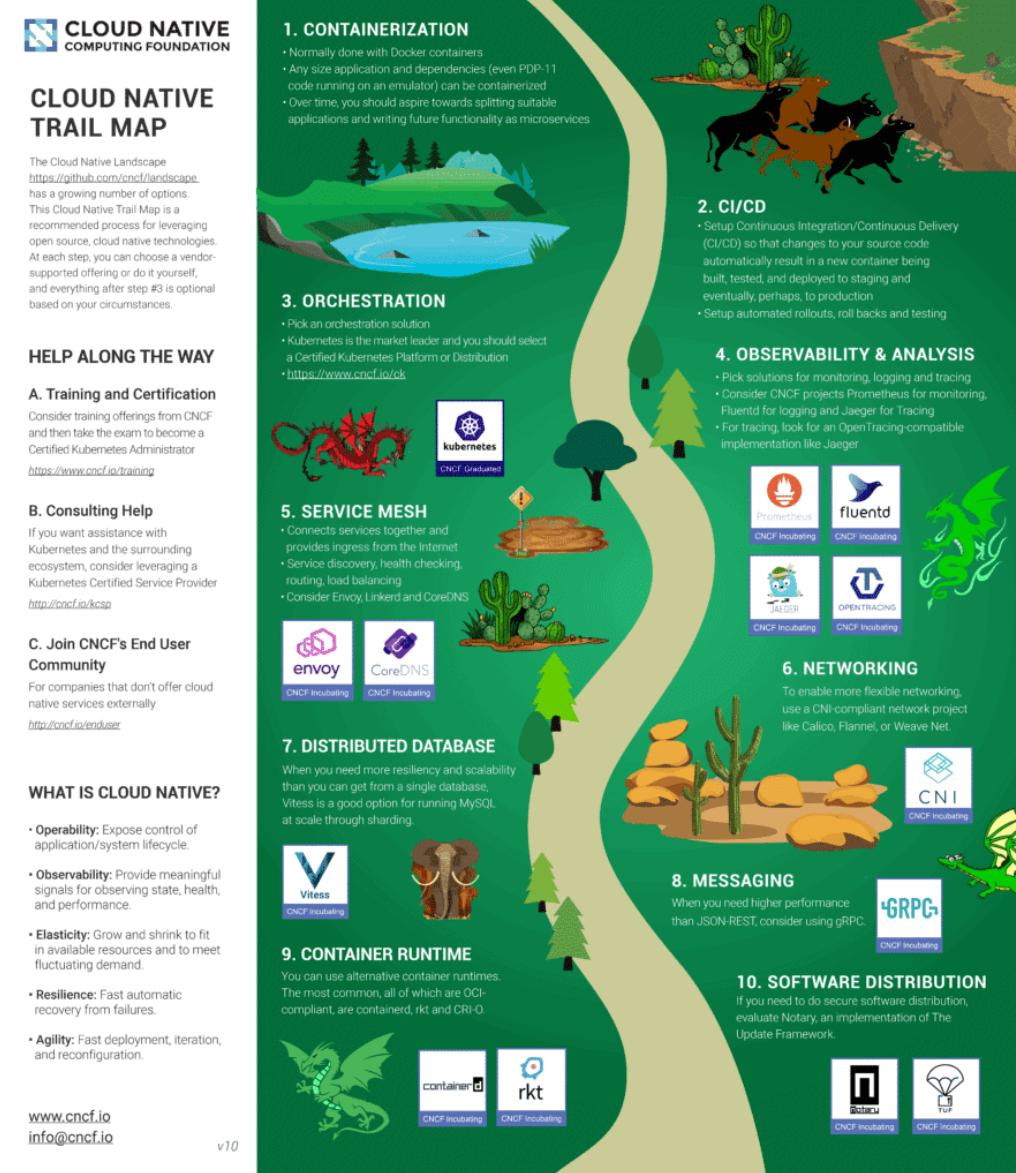
- Lightweight, isolated runtime environment for each microservice
- Enhanced scalability, fault tolerance, isolation

Orchestration

- Tools like Kubernetes for managing and scaling microservices
- Handles load balancing, service discovery, automated scaling

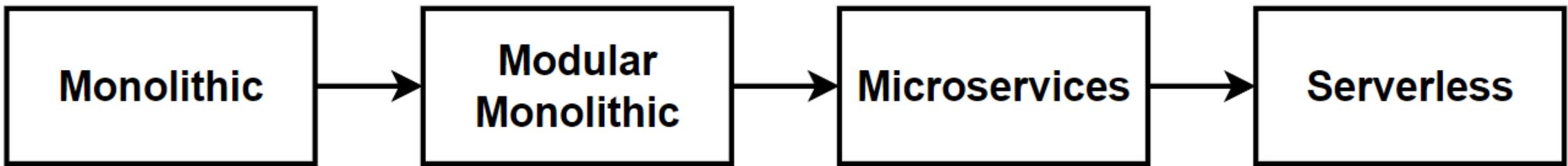
Continuous Delivery

- Rapid, automated deployments of new or updated microservices
- Easier to iterate and improve architecture over time

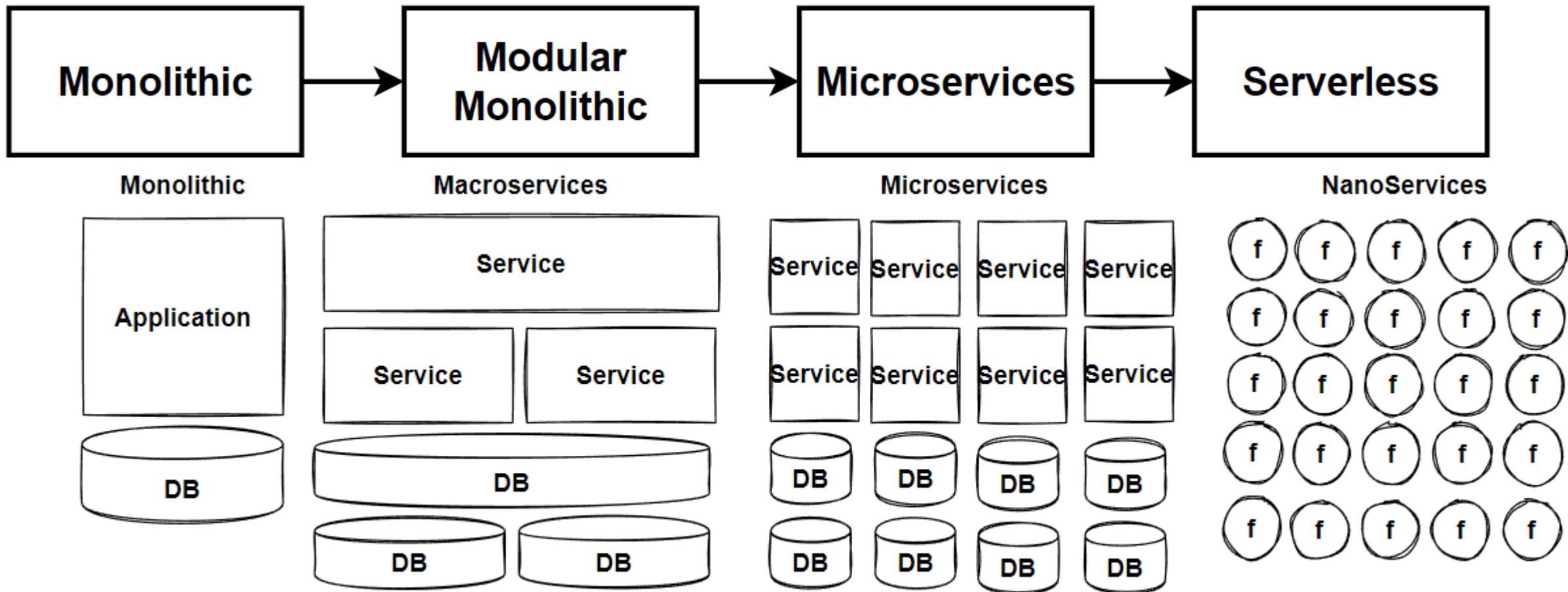


<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>

Architecture Design Journey

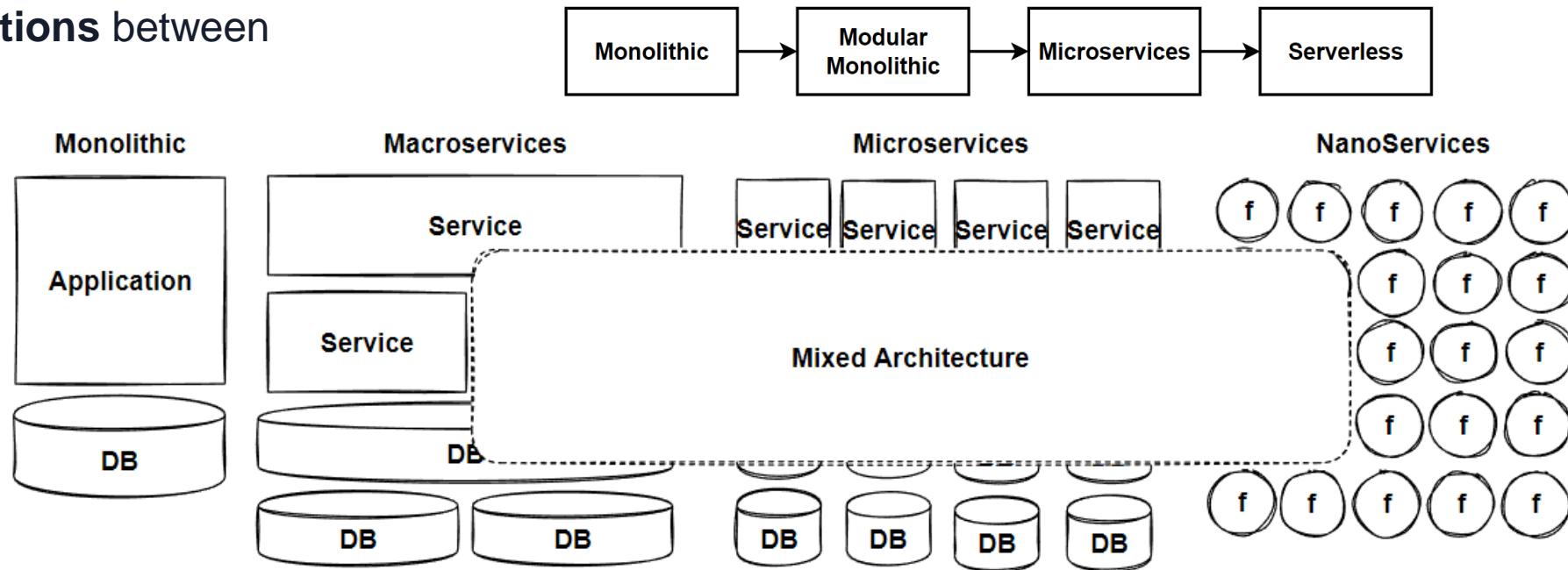


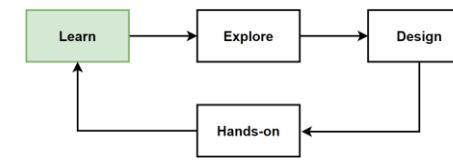
Macroservices to Nanoservices



Which architecture approach we should choose ?

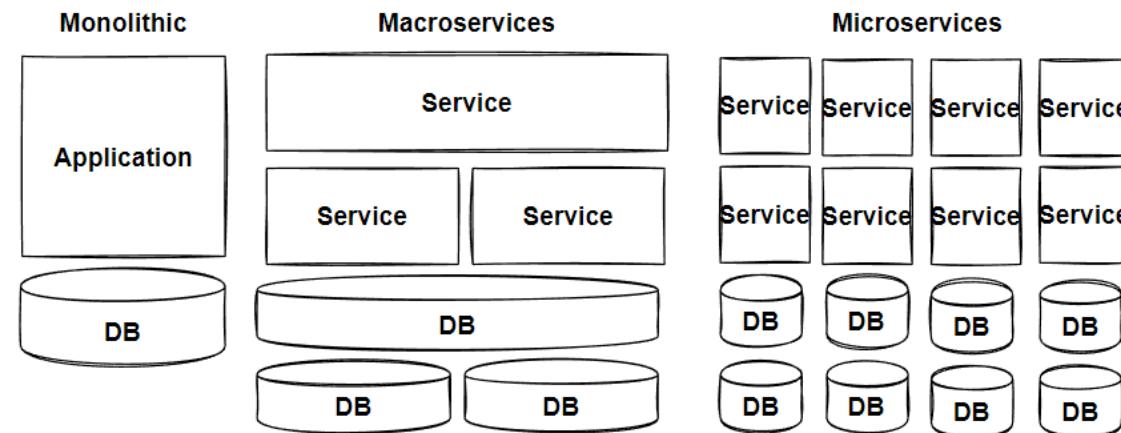
- It depends on your project
- Use **Mixed Architecture**
- Provide **transitions** between architectures

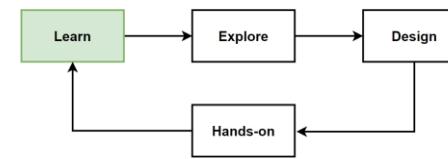




What are Microservices ?

- Microservices are **small, independent, and loosely coupled** services that can work together.
- Each service is a **separate codebase**, which can be managed by a small development team.
- Microservices **communicate** with each other by using **well-defined APIs**.
- Microservices can be **deployed independently and autonomously**.
- Microservices can work with many different technology stacks which is **technology agnostic**.
- Microservices has its **own database** that is not shared with other services.



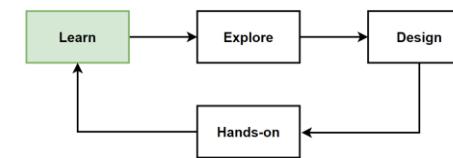


What is Microservices Architecture ?

- From Martin Fowlers Microservices article;

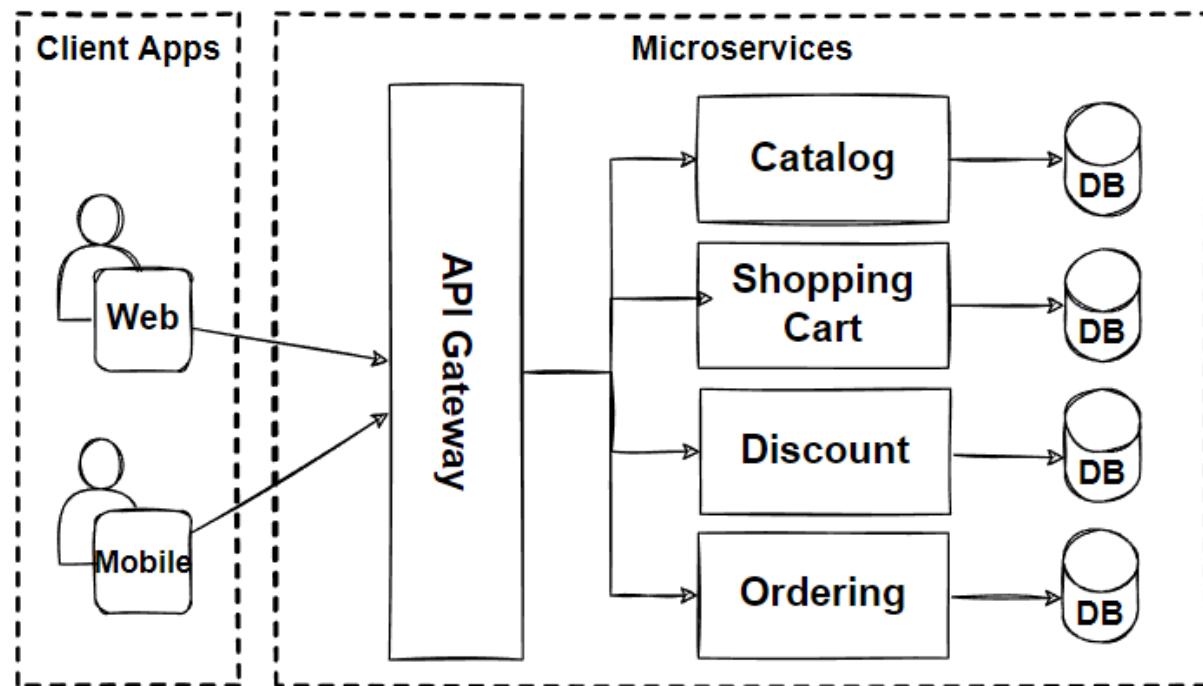
*The microservice architectural style is an approach to developing a single application as a **suite of small services**, each running in **its own process** and **communicating** with lightweight mechanisms, often an **HTTP or gRPC API**.*

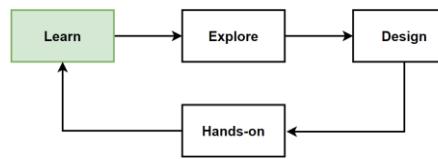
- Microservices are **built around business capabilities** and **independently deployable** by fully automated deployment process.
- Microservices architecture **decomposes** an application into **small independent services** that communicate over **well-defined APIs**. Services are owned by **small, self-contained teams**.
- Microservices architecture is a **cloud native architectural approach** in which services composed of many **loosely coupled** and **independently deployable** smaller components.
- Microservices have their **own technology stack**, communicate to each other over a combination of **REST APIs**, are organized by **business capability**, with the **bounded contexts**.
- Following **Single Responsibility Principle** that referring separating responsibilities as per services.



Microservices Characteristics

- From Martin Fowlers Microservices article;
- Componentization via Services
- Organized around Business Capabilities
- Products not Projects
- Smart endpoints and dumb pipes
- Decentralized Governance
- Decentralized Data Management
- Infrastructure Automation
- Design for failure





Benefits of Microservices Architecture

- Agility, Innovation and Time-to-market**

Microservices architectures make applications easier to scale and faster to develop, enabling innovation and accelerating time-to-market for new features.

- Flexible Scalability**

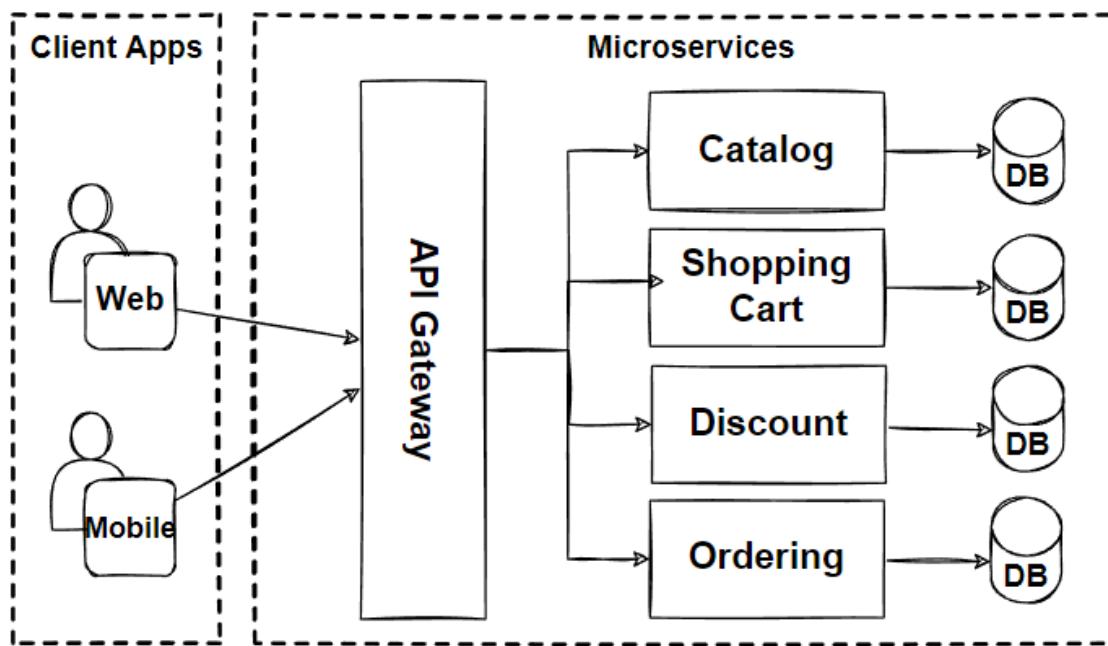
Microservices can be scaled independently, so you scale out sub-services that require less resources, without scaling out the entire application.

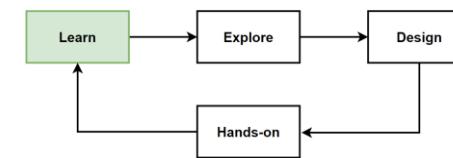
- Small, focused teams**

Microservices should be small enough that a single feature team can build, test, and deploy it.

- Small and separated code base**

Microservices are not sharing code or data stores with other services, it minimizes dependencies, and that makes easier to adding new features.





Benefits of Microservices Architecture -2

- **Easy Deployment**

Microservices enable continuous integration and continuous delivery, making it easy to try out new ideas and to roll back if something doesn't work.

- **Technology agnostic, Right tool for the job**

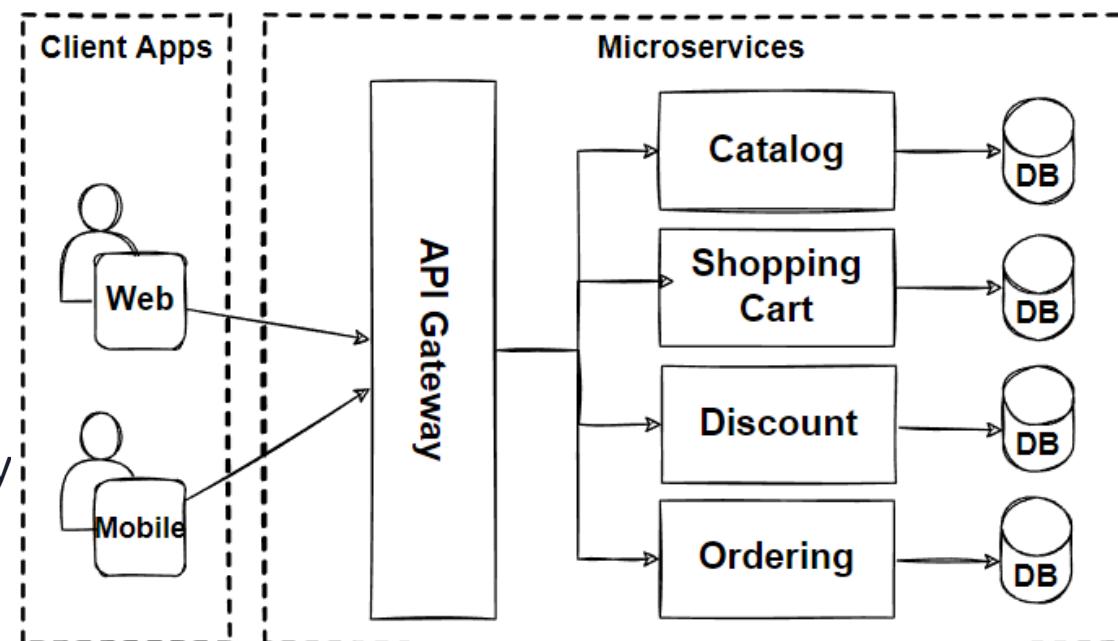
Small teams can pick the technology that best fits their microservice and using a mix of technology stacks on their services.

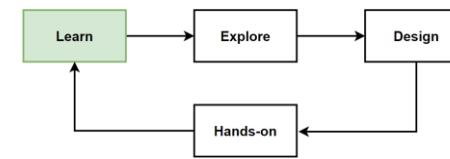
- **Resilience and Fault isolation**

Microservices are fault tolerated and handle faults correctly for example by implementing retry and circuit breaking patterns.

- **Data isolation**

Databases are separated with each other according to microservices design. Easier to perform schema updates, because only a single database is affected.





Challenges of Microservices Architecture

- Complexity**

Each service is simpler, but the entire system is more complex. Deployments and Communications can be complicated for hundreds of microservices.

- Network problems and latency**

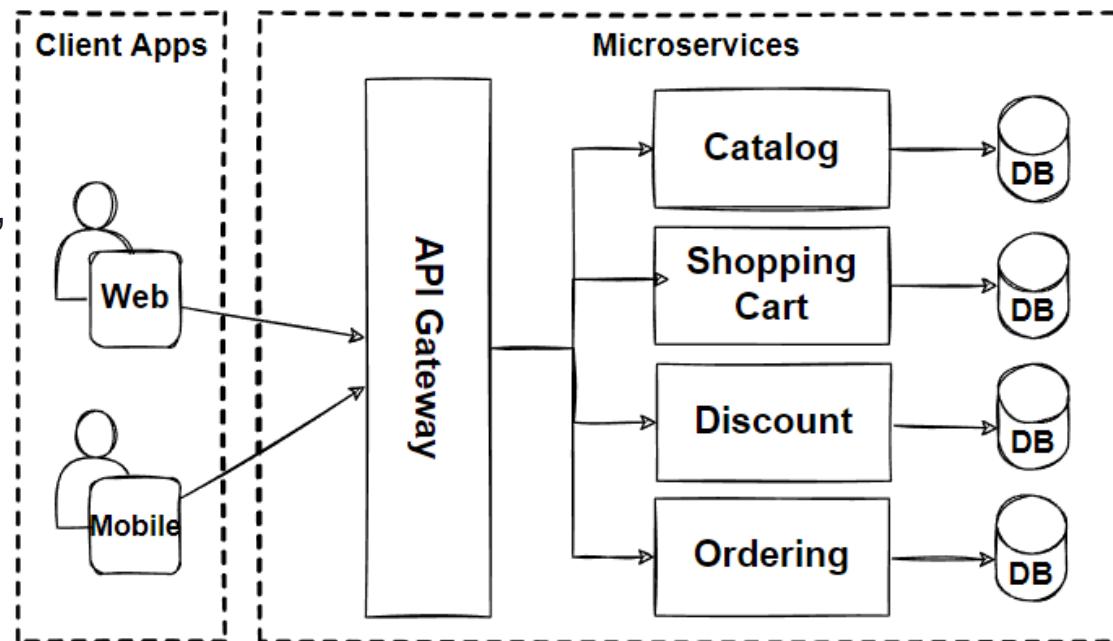
Microservice communicate with inter-service communication, we should manage network problems. Chain of services increase latency problems and become chatty API calls.

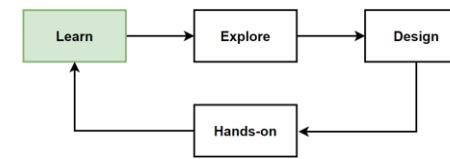
- Development and testing**

Hard to develop and testing these E2E processes in microservices architectures if we compare to monolithic ones.

- Data integrity**

Microservice has its own data persistence. Data consistency can be a challenge. Follow eventual consistency where possible.





Challenges of Microservices Architecture -2

- Deployment**

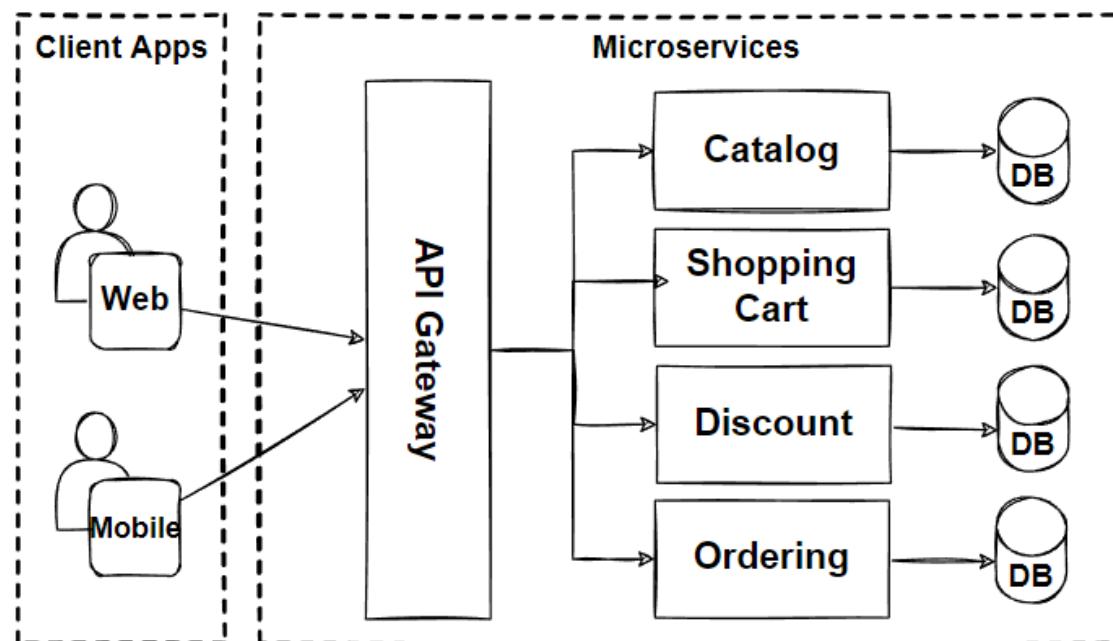
Deployments are challenging. Require to invest in quite a lot of devops automation processes and tools. The complexity of microservices becomes overwhelming for human deployment.

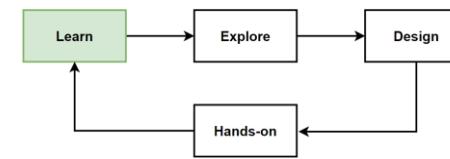
- Logging & Monitoring**

Distributed systems are required to centralized logs to bring everything together. Centralized view of the system to monitor sources of problems.

- Debugging**

Debugging through local IDE isn't an option anymore. It won't work across dozens or hundreds of services.





When to Use Microservices Architecture

- Make Sure You Have a “Really Good Reason” for Implementing Microservices**

Check if your application can do without microservices. When your application requires agility to time-to-market with zero-down time deployments and updated independently that needs more flexibility.

- Iterate With Small Changes and Keep the Single-Process Monolith as Your “Default”**

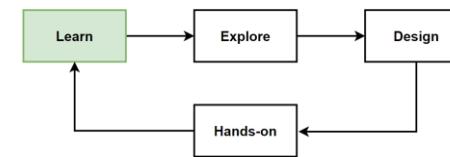
Sam Newman and Martin Fowler offers Monolithic-First approach. Single-process monolithic application comes with simple deployment topology. Iterate and refactor with turning a single module from the monolith into a microservices one by one.

- Required to Independently Deploy New Functionality with Zero Downtime**

When an organization needs to make a change to functionality and deploy that functionality without affecting rest of the system.

- Required to Independently Scale a Portion of Application**

Microservice has its own data persistence. Data consistency can be a challenge. Follow eventual consistency where possible.



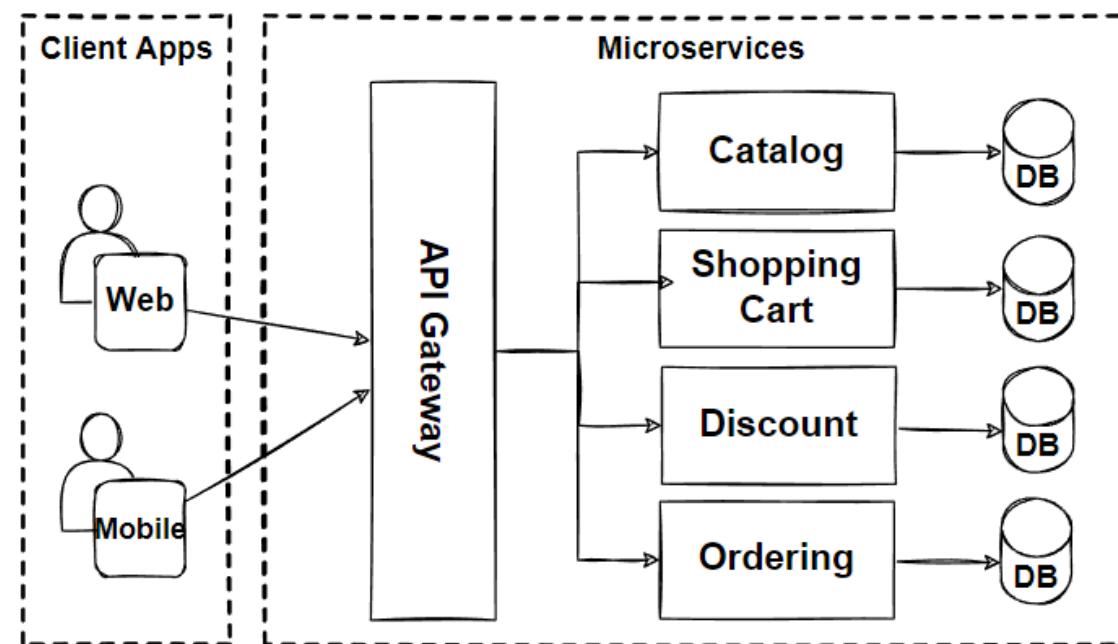
When to Use Microservices Architecture -2

- Data Partitioning with different Database Technologies**

Microservices are extremely useful when an organization needs to store and scale data with different use cases. Teams can choose the appropriate technology for the services they will develop over time.

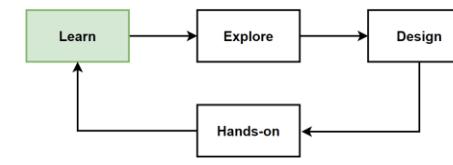
- Autonomous Teams with Organizational Upgrade**

Microservices will help to evolve and upgrade your teams and organizations. Organizations need to distribute responsibility into teams, where each team makes decisions and develops software autonomously.



When **Not** to Use Microservices

Anti-Patterns of Microservices



- **Don't do Distributed Monolith**

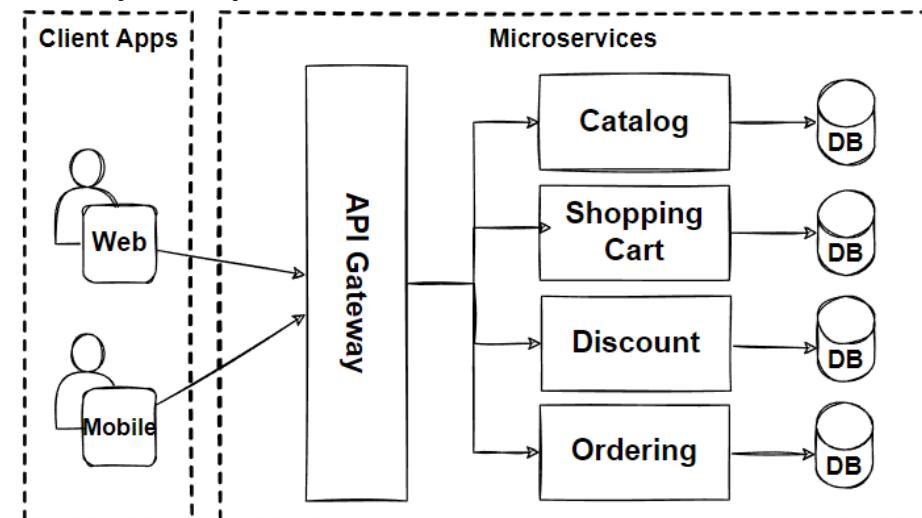
Make sure that you decompose your services properly and respecting the decoupling rule like applying bounded context and business capabilities principles.

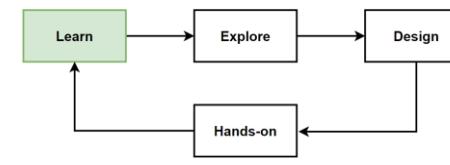
- Distributed Monolith is the worst case because you increase complexity of your architecture without getting any benefit of microservices.

- **Don't do microservices without DevOps or cloud services**

Microservices are embrace the distributed cloud-native approaches. And you can only maximize benefits of microservices with following these cloud-native principles.

- CI/CD pipeline with devops automations
- Proper deployment and monitoring tools
- Managed cloud services to support your infrastructure
- Key enabling technologies and tools like Containers, Docker, and Kubernetes
- Following asnyc communications using Messaging and event streaming services





When **Not** to Use Microservices

- Limited Team sizes, Small Teams**

If you don't have a team size that cannot handle the microservice workloads, This will only result in the delay of delivery.

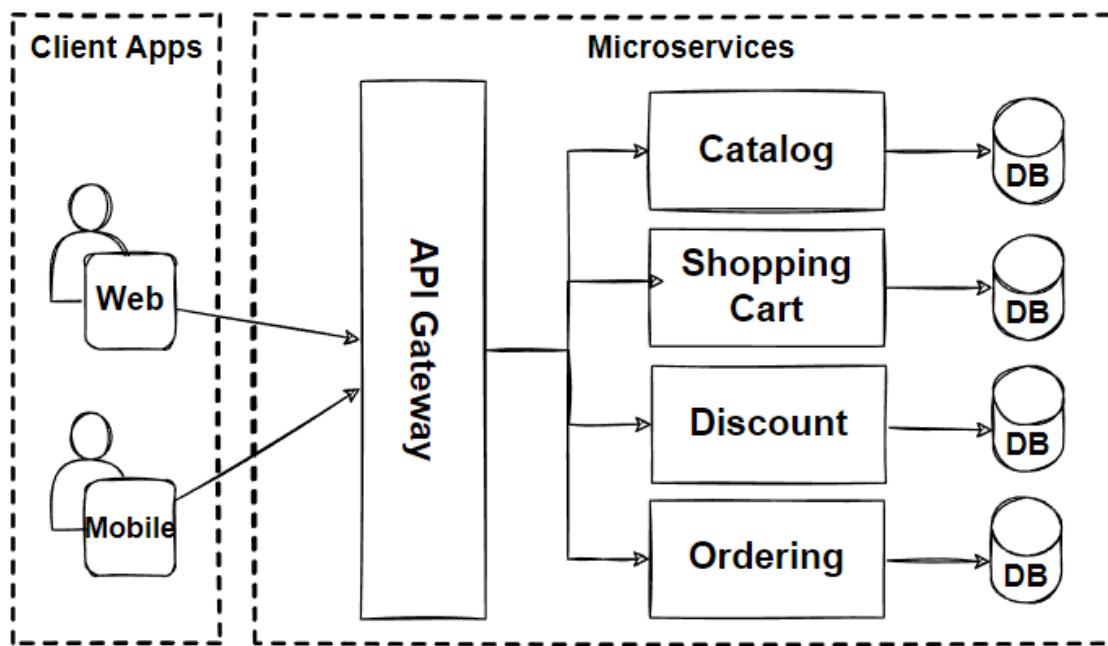
- For a small team, a microservice architecture can be hard to justify, because team is required just to handle the deployment and management of the microservices themselves.

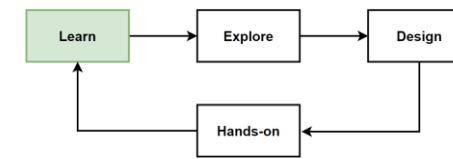
- Brand new products or startups**

If you are working on a new startup or brand new product which require significant change when developing and iterating your product, then you should not start with microservices.

- Microservices are so expensive when you re-design your business domains. Even if you do become successful enough to require a highly scalable architecture.

- The Shared Database anti-pattern**





Monolithic vs Microservices Architecture Comparison

■ Application Architecture

Monolith has a simple straightforward structure of one undivided unit. Microservices have a complex structure that consists of various heterogeneous services and databases.

■ Scalability

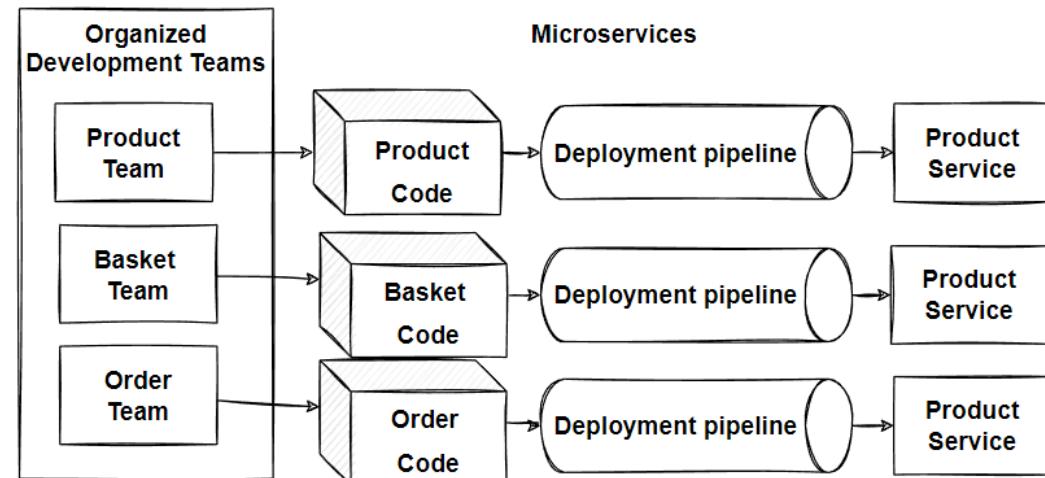
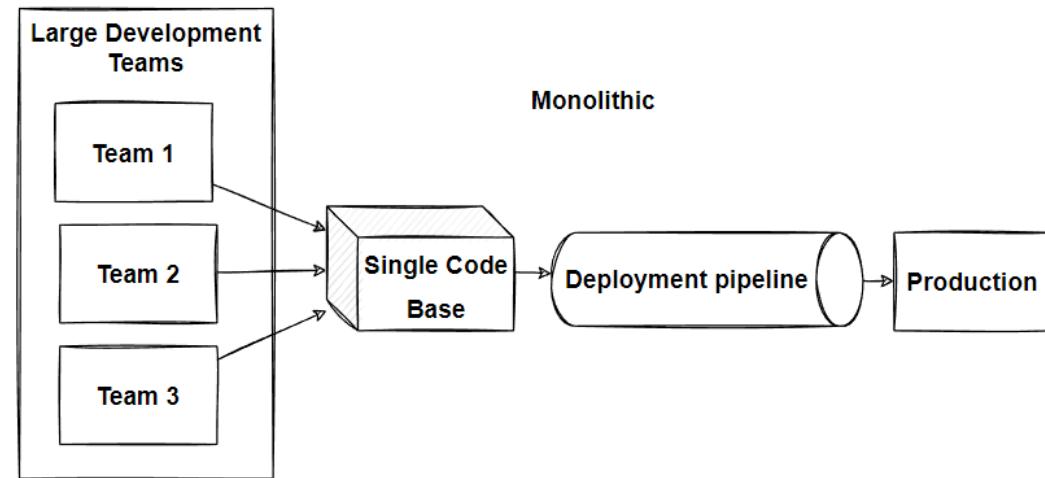
Monolithic application is scaled as a whole single unit, but microservices can be scaled unevenly. encourages companies to migrate their applications to microservices.

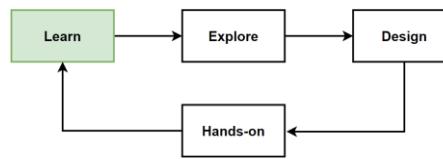
■ Deployment

Monolithic application provides fast and easy deployment of the whole system. Microservices provides zero-downtime deployment and CI/CD automation.

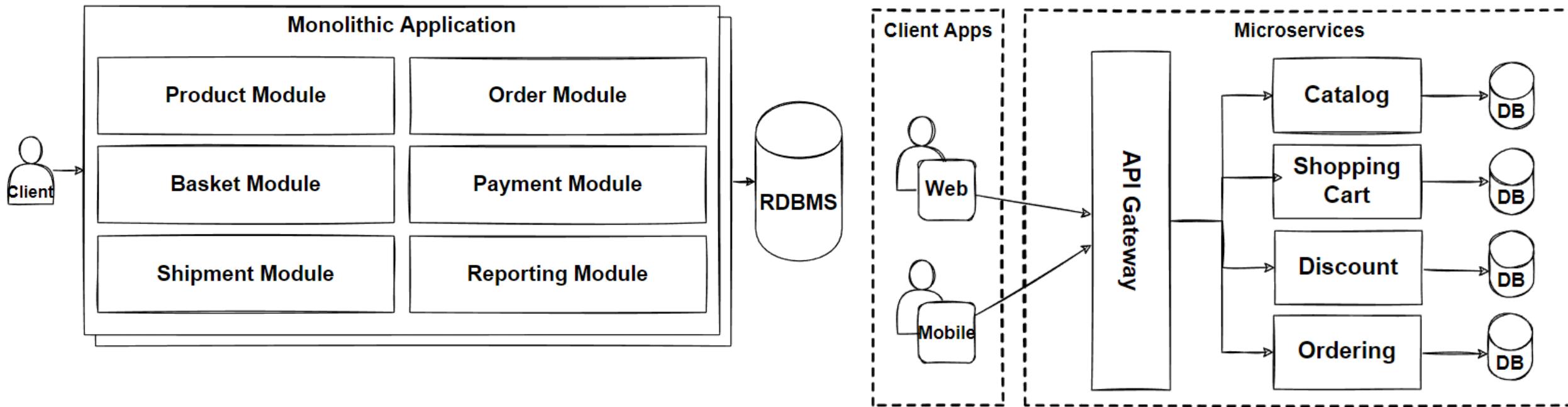
■ Development team

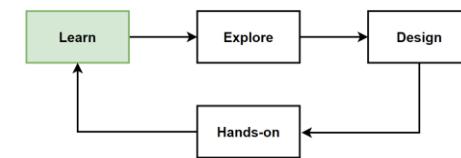
If your team doesn't have experience with microservices and container systems, building a microservices-based application will be difficult.





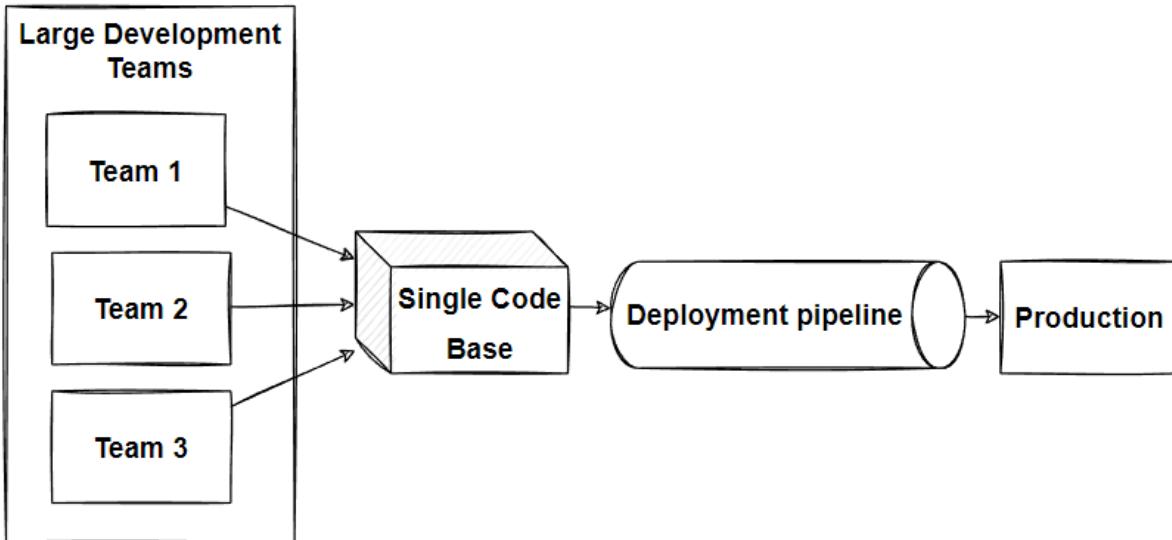
Architecture Comparison



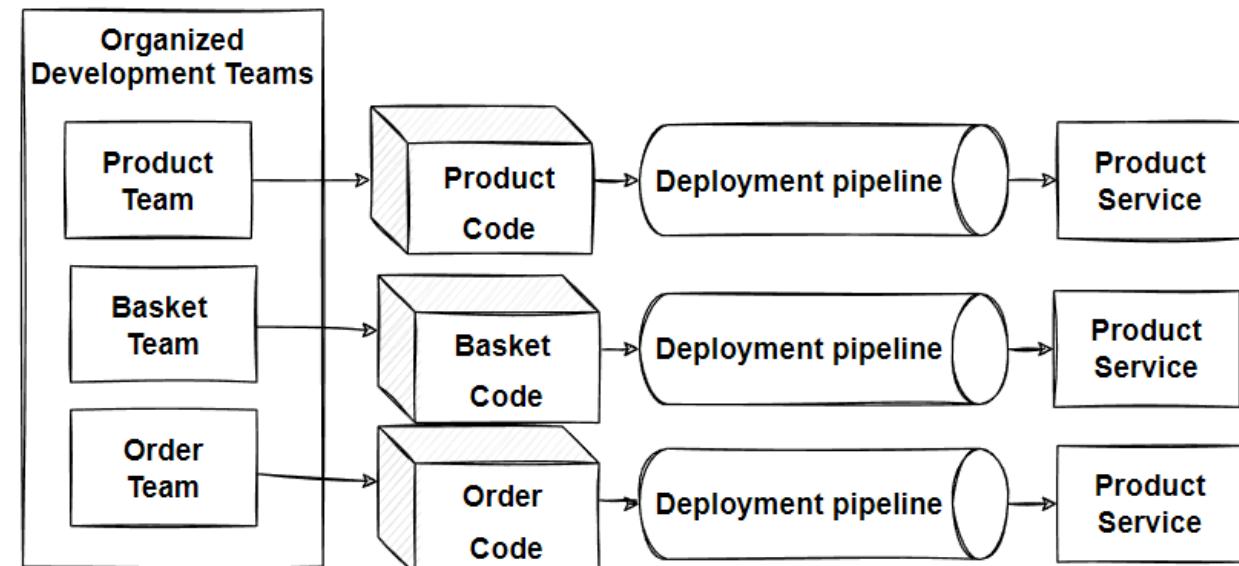


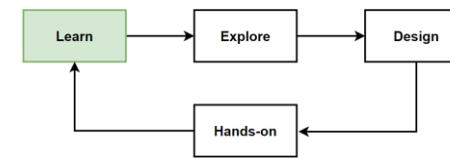
Deployment Comparison

Monolithic



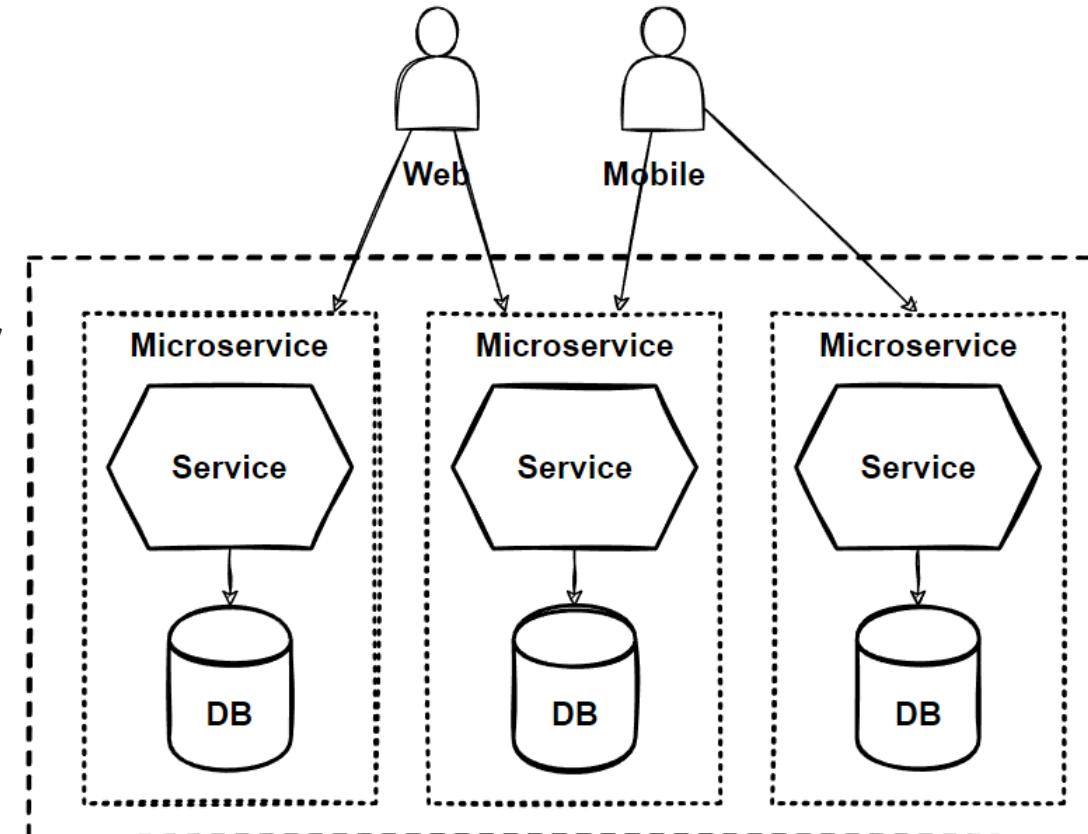
Microservices





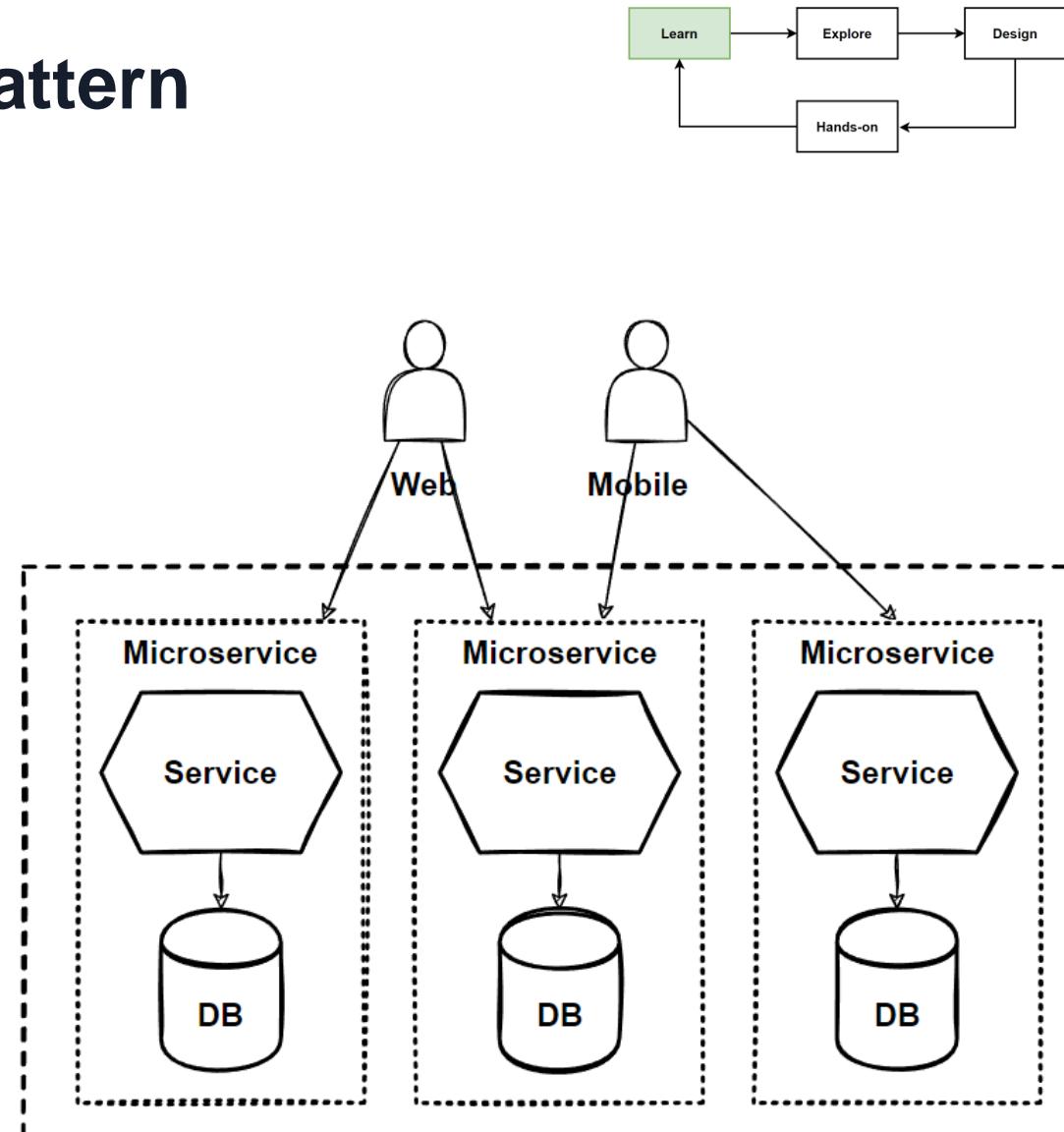
The Database-per-Service Pattern

- Core characteristic of the microservices architecture is the **loose coupling of services**. every service should have its own **databases**, it can be **polyglot persistence** among to microservices.
- E-commerce application. We will have **Product - Ordering** and **SC** microservices that **each services data in their own databases**. Any changes to one database **don't impact other microservices**.
- The **service's database can't be accessed** directly by other microservices. Each service's persistent data can only be accessed via **Rest APIs**.

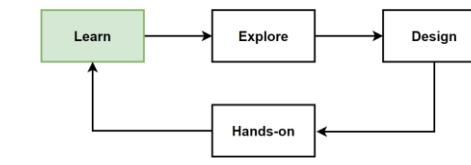


Benefits of the Database-per-Service Pattern with Polygot Persistence

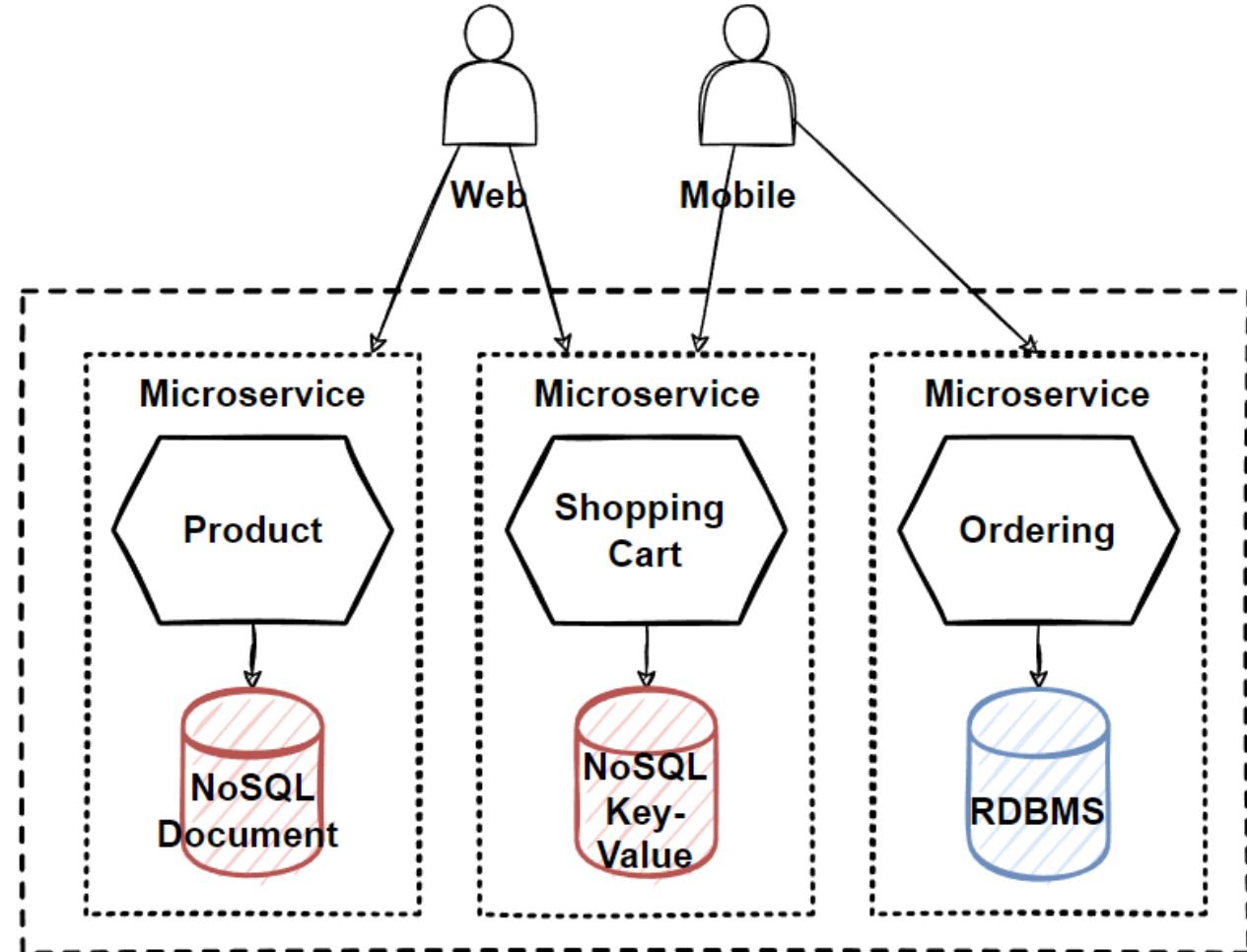
- **Data schema changes** made easy without impacting other microservices.
- Each **database can scale independently**.
- Microservices domain data is **encapsulated** within the service.
- If one of the database server is down, this will **not affect to other services**.
- **Polyglot data persistence** gives ability to select the **best optimized storage** needs per microservices.



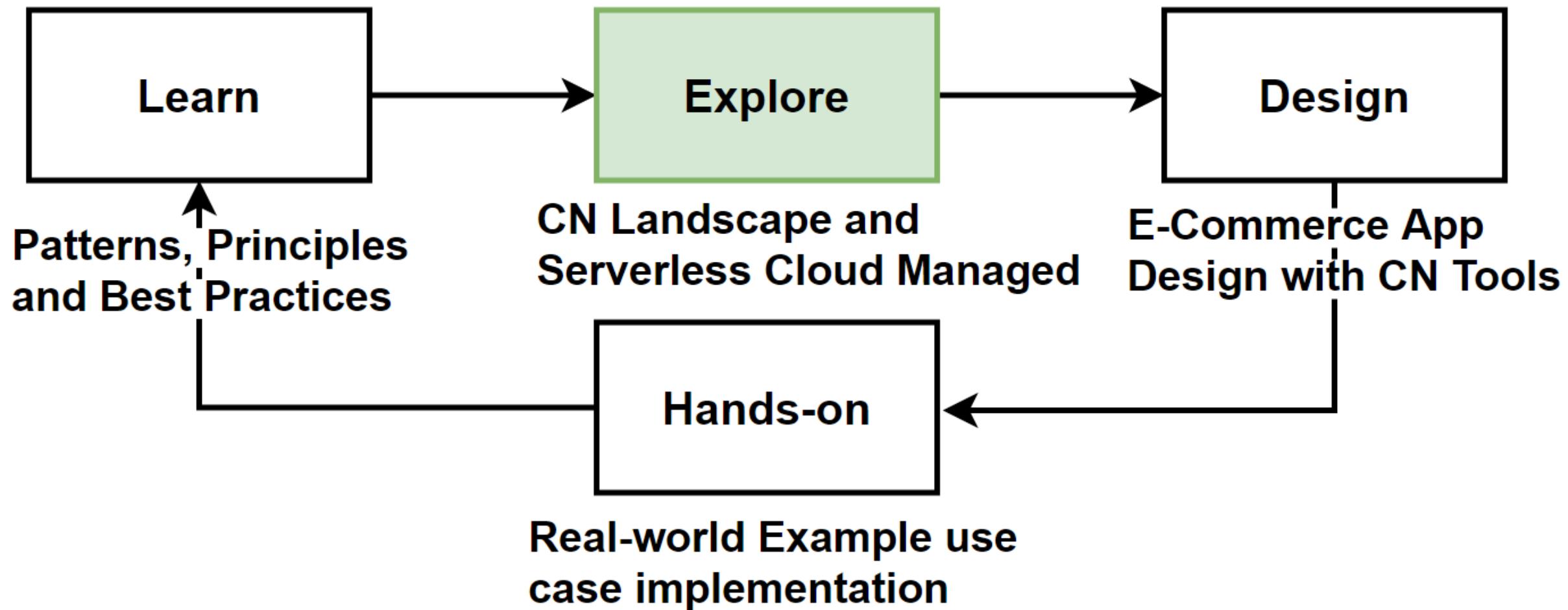
E-Commerce with Database-per-Service Pattern and Polygot Persistence



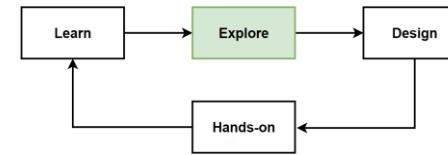
- Product service using **NoSQL document database** for storing catalog related data.
- Shopping cart service using a **distributed cache** that supports its simple, **key-value data store**.
- Ordering service using a **relational database** to handle the rich relational structure.
- **NoSQL databases** able to massive scale and high availability, and also schemaless structure give flexibility.



Way of Learning – The Course Flow



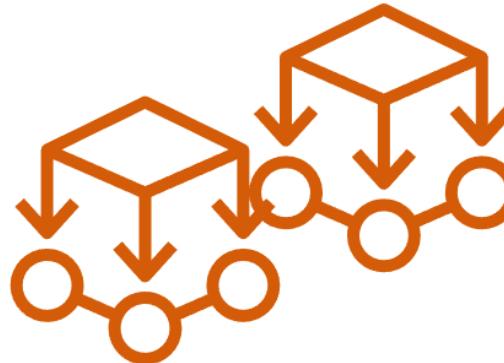
Microservices Languages and Frameworks



- Microservices can be developed using a variety of languages and frameworks
 - The choice depends on the requirements, team expertise, and factors like performance, scalability, and cloud-native support

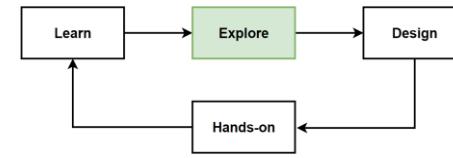
Java – Spring Boot

- Java is a popular choice for enterprise-grade applications
 - Spring Boot provides a rapid application development platform for building stand-alone, production-ready Spring applications
 - Spring Boot's convention-over-configuration simplifies development



JavaScript – Node.js

- Node.js allows server-side JavaScript development
 - Its event-driven, non-blocking I/O model is well-suited for microservices
 - Express.js is a simple and minimalistic framework often used with Node.js



Microservices Languages and Frameworks - 2

Python – Flask, Django

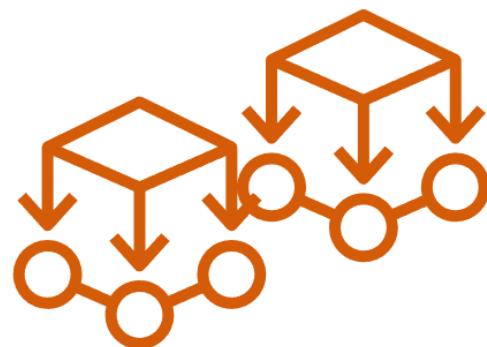
- Python's simplicity make it good for microservices, especially in data-intensive apps
- Flask is a lightweight framework suitable for simple microservices
- Django offers more features and is better suited for larger applications

Go (Golang) – GoMicro

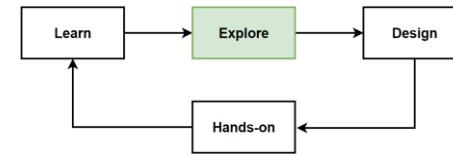
- Go is a statically-typed, compiled language developed by Google
- Go's efficiency, and excellent concurrency support make it increasingly popular for services
- The standard library in Go is sufficient for building microservices without additional frameworks

C# – .NET

- C# is a modern, object-oriented programming language that is widely used for building web applications, and microservices.
- .NET is a cross-platform, open-source framework for building modern, cloud-native applications, including microservices.
- .NET provides a wide range of libraries and tools for building microservices
- .NET is also highly scalable, with support for containerization and orchestration using tools like Docker and Kubernetes.



Microservices



Microservices Frameworks – Java Frameworks

Spring Boot

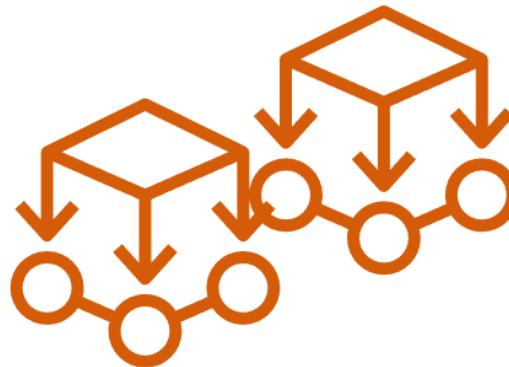
- Industry-standard Java microservices framework with powerful features like dependency injection, security, and transaction management.
- Aided by Spring Cloud, it offers resilience and flexibility. Spring Boot applications can be quickly started using Spring Initializr and packaged as a JAR.

Quarkus

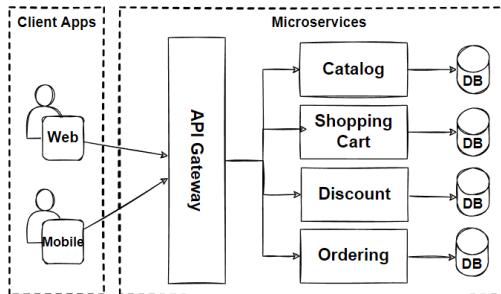
- A lightweight framework by Red Hat, perfect for building serverless and containerized microservices with quick startup times and low memory footprint.
- Offers both reactive and imperative programming models.

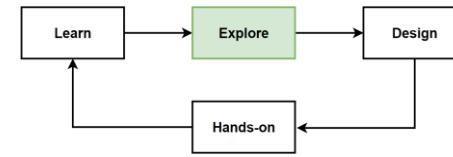
Micronaut

- A modern, JVM-based framework for creating modular, testable microservice applications.
- Like Quarkus, it also provides fast startup times and low memory footprint, along with first-class support for reactive programming.



Microservices

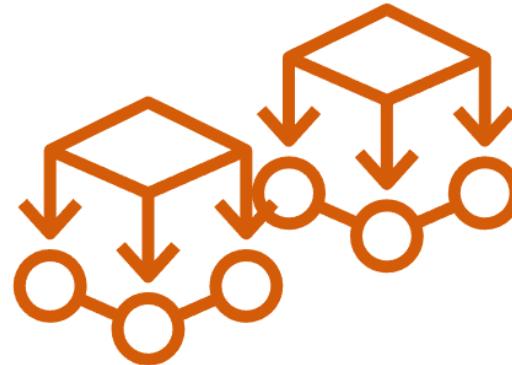




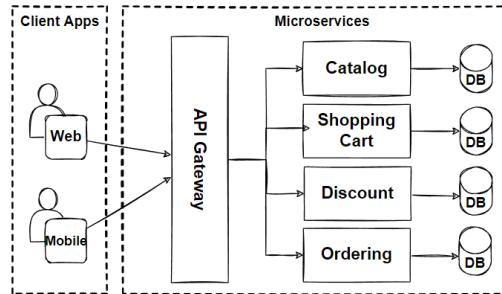
Microservices Frameworks – Go Frameworks

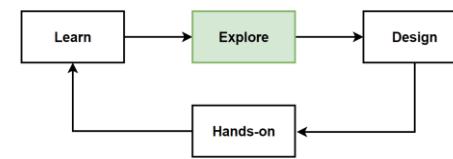
GoMicro

- A lightweight Go framework that offers built-in support for service discovery, load balancing
- Different communication protocols like HTTP, gRPC, and NATS.
- It is suitable for creating REST and gRPC protocols, allowing seamless inter-service communication.



Microservices





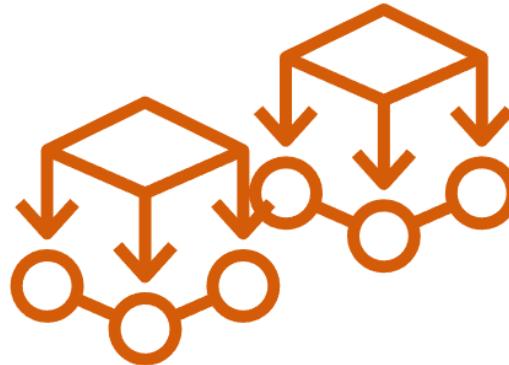
Microservices Frameworks – Python Frameworks

Django

- A feature-rich Python web framework with built-in support for models, views, templates, security, and user authentication.
- Perfect for creating scalable and maintainable microservices.

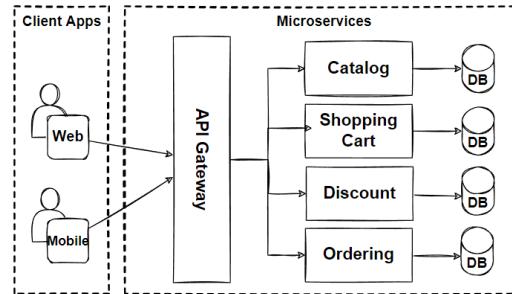
Flask

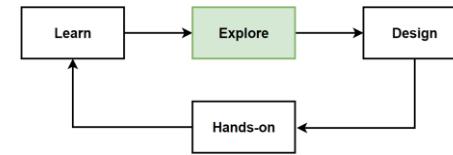
- A minimalist Python web framework well-suited for building focused microservices. Offers built-in support for request handling, views, and templates.



Microservices

django





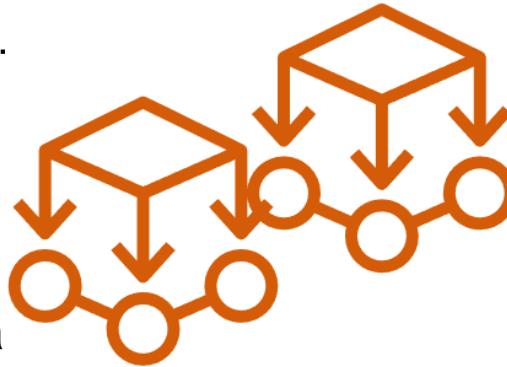
Microservices Frameworks – JavaScript Frameworks

Node.js

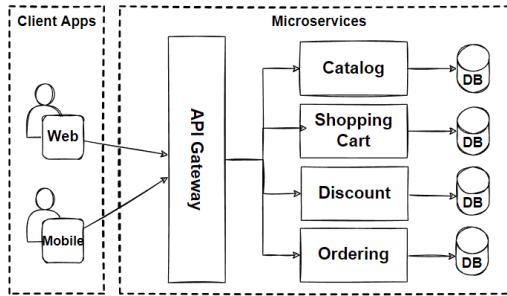
- A versatile server-side JavaScript runtime with support for event-driven programming, non-blocking I/O, and asynchronous programming.
- It comes with a variety of libraries and frameworks, including Express and Koa.

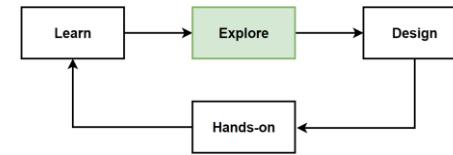
Express.js

- A minimal and flexible Node.js web application framework, ideal for building APIs quickly and efficiently. Part of the MEAN stack.



Microservices

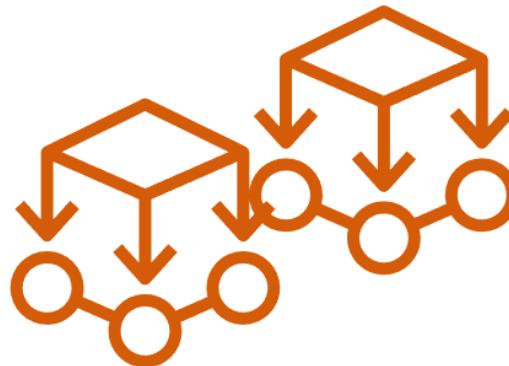




Microservices Frameworks – .NET Frameworks

ASP.NET

- A popular framework for building web applications and microservices with .NET Core.
- It offers built-in support for MVC, Web API, Razor Pages, dependency injection, logging, and caching.
- It supports containerization and orchestration using Docker and Kubernetes.

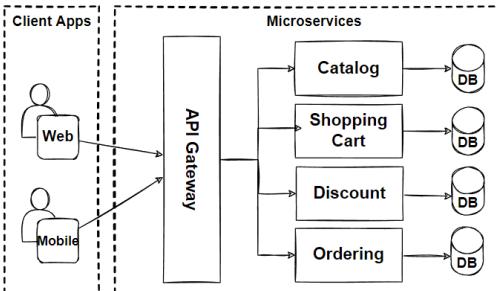


.NET is heavily investing cloud-native development

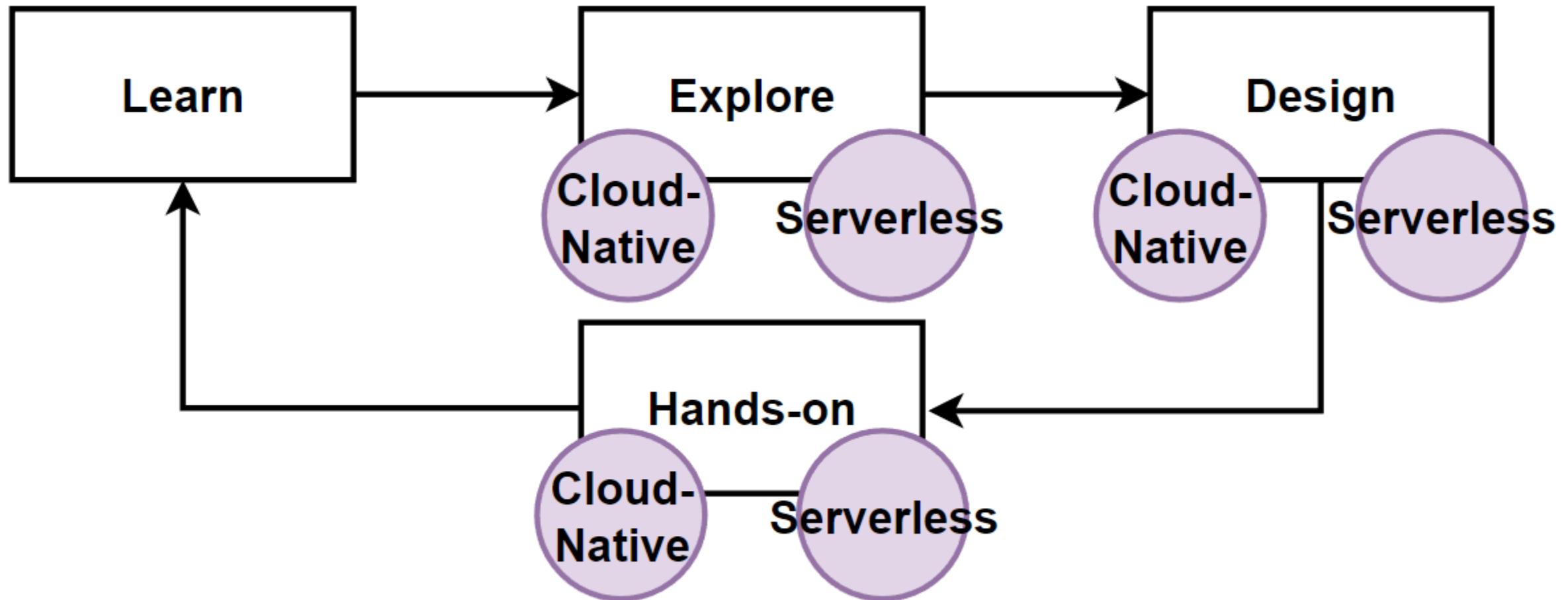
- .NET is a cross-platform, open-source framework for building modern, cloud-native applications, including microservices.
- .NET provides a wide range of libraries and tools for building microservices
- .NET is also highly scalable, with support for containerization and orchestration using tools like Docker and Kubernetes.



Microservices



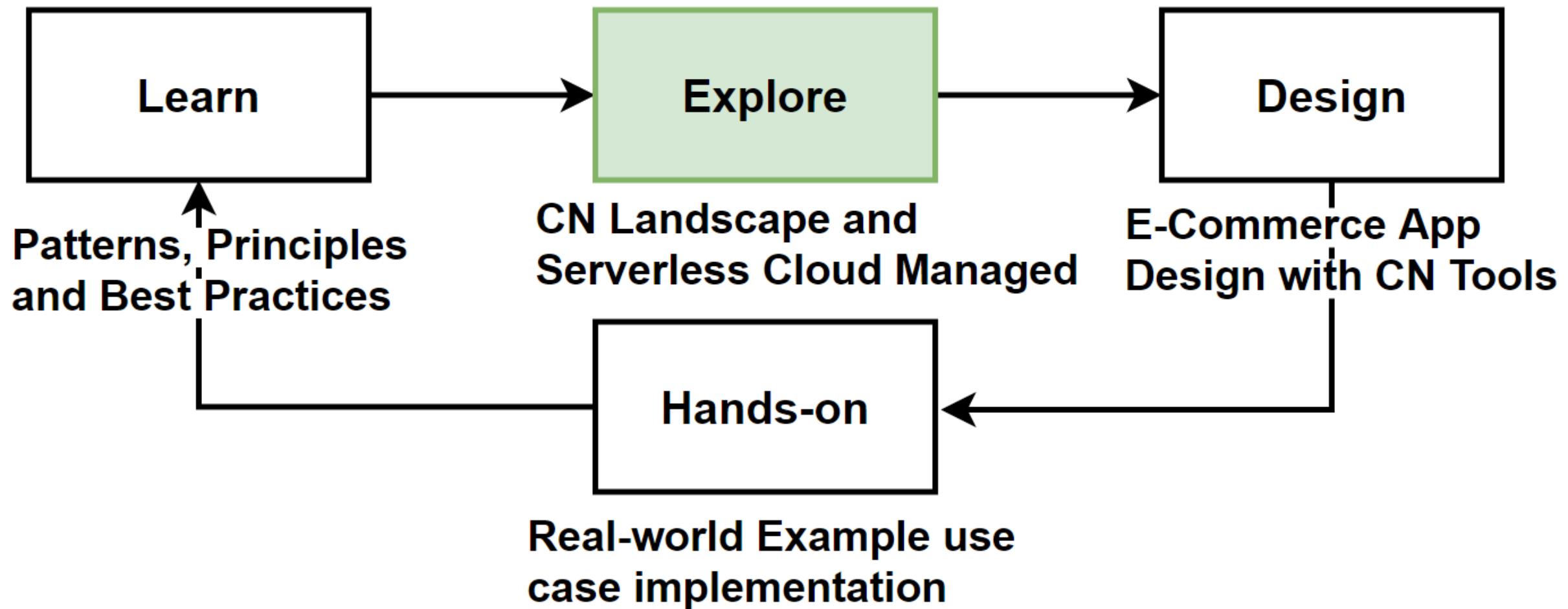
Way of Learning – Cloud-Native & Serverless Cloud Managed Tool

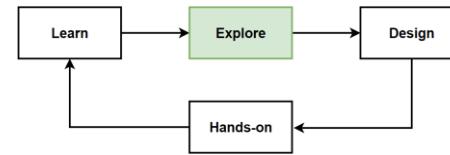


Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

Explore: Cloud Managed and Serverless Microservices Frameworks

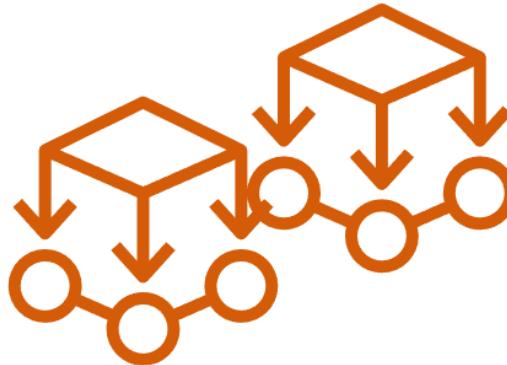




Cloud Serverless Functions for Microservices

What are Serverless Functions?

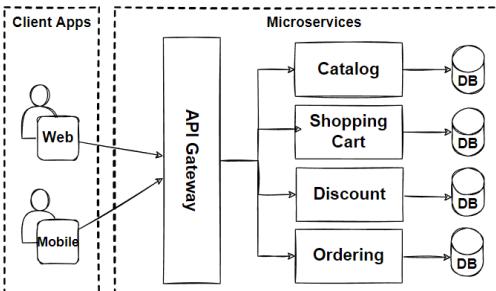
- Serverless functions are single-purpose, programmatic functions hosted on managed infrastructure.
- They are invoked via the internet, are fully scalable, and have built-in fault tolerance.
- These are ideal for variable workloads, removing the overhead of maintaining a full-time server.

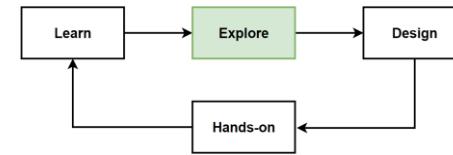


Benefits in Microservices Development

- Aligns with the microservices philosophy of small, loosely coupled, and independently deployable units.
- Each microservice can be a serverless function, independently scalable and deployable.
- Developers can focus on writing business logic, not on infrastructure management.

Microservices

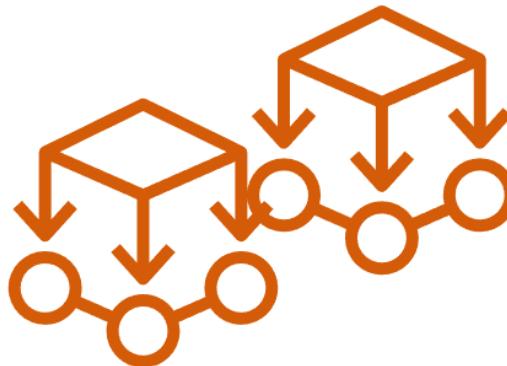




Cloud Serverless Functions for Microservices - 2

AWS Lambda

- Amazon's serverless platform, ideal for running code in response to events like database changes or message arrival. Executes code only when needed and scales automatically.
- Event-driven, compute-on-demand service
- Individual microservices for data processing, calculations, user requests handling
- Key features: Executes code only when needed, automatic scaling

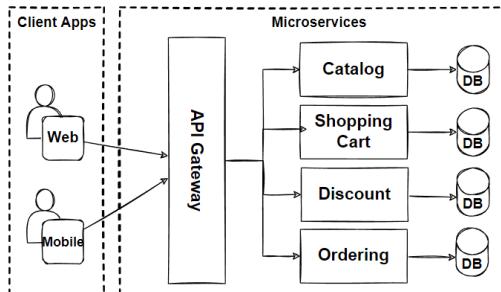


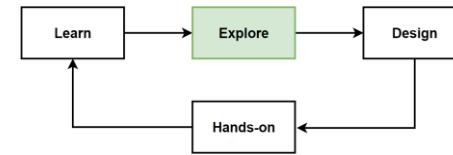
Azure Functions

- Microsoft's serverless offering, providing event-driven, compute-on-demand similar to AWS Lambda. Ideal for handling HTTP requests, data processing, or running background tasks.
- Serverless computing service similar to AWS Lambda
- Individual microservices for HTTP requests handling, data processing, background tasks
- Key features: Event-driven, compute-on-demand experience



Microservices

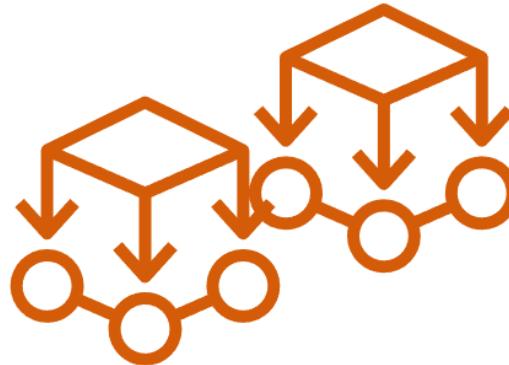




Cloud Serverless Functions for Microservices - 3

Google Cloud Functions

- Google's serverless event-driven platform. Can be used for handling user requests, data processing, or running machine learning models.
- Serverless event-driven computing service
- Individual microservices for user requests handling, data processing, machine learning models running
- Key features: Similar functionality to AWS Lambda and Azure Functions

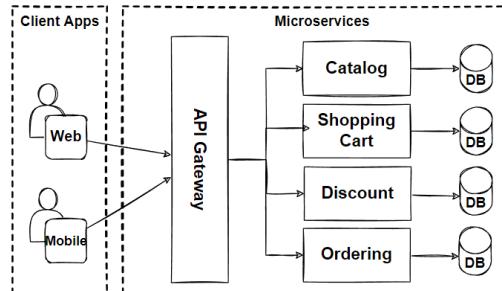


Vercel Edge Functions

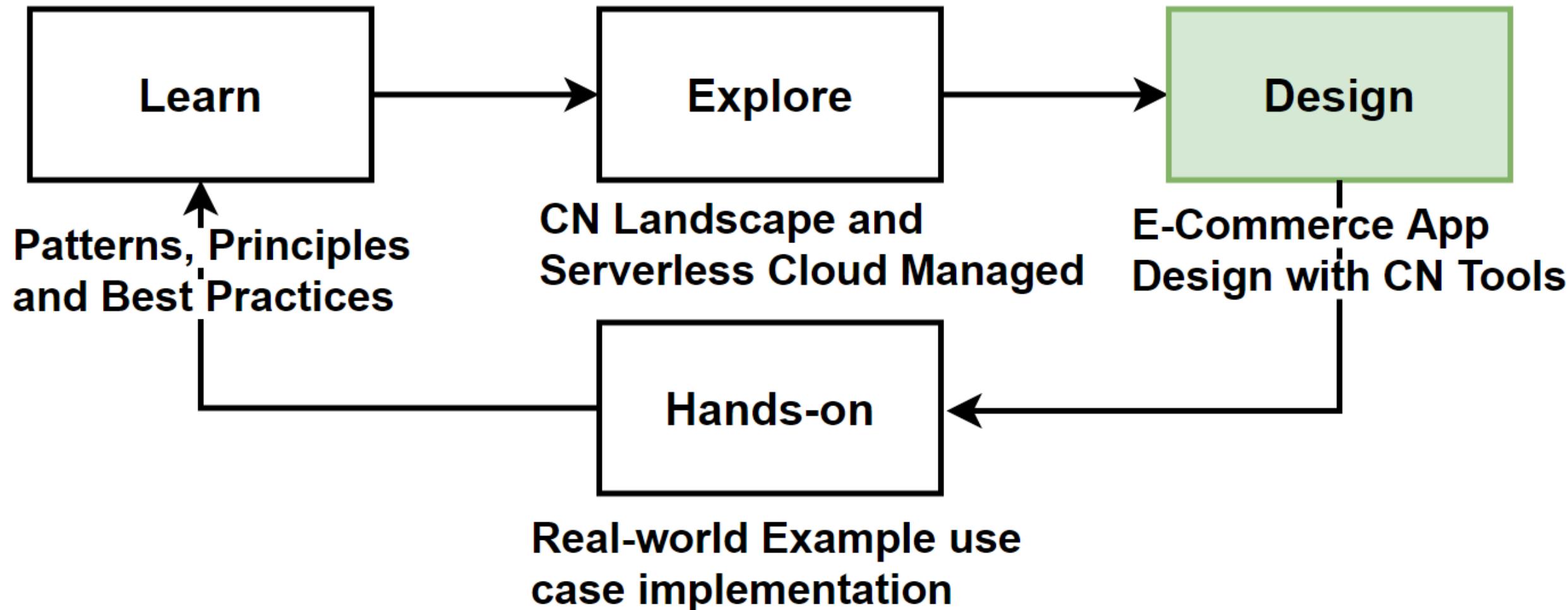
- Combines excellent developer experience with a focus on end-user performance.
- Focus on best developer experience and end-user performance
- Ideal platform for front-end teams



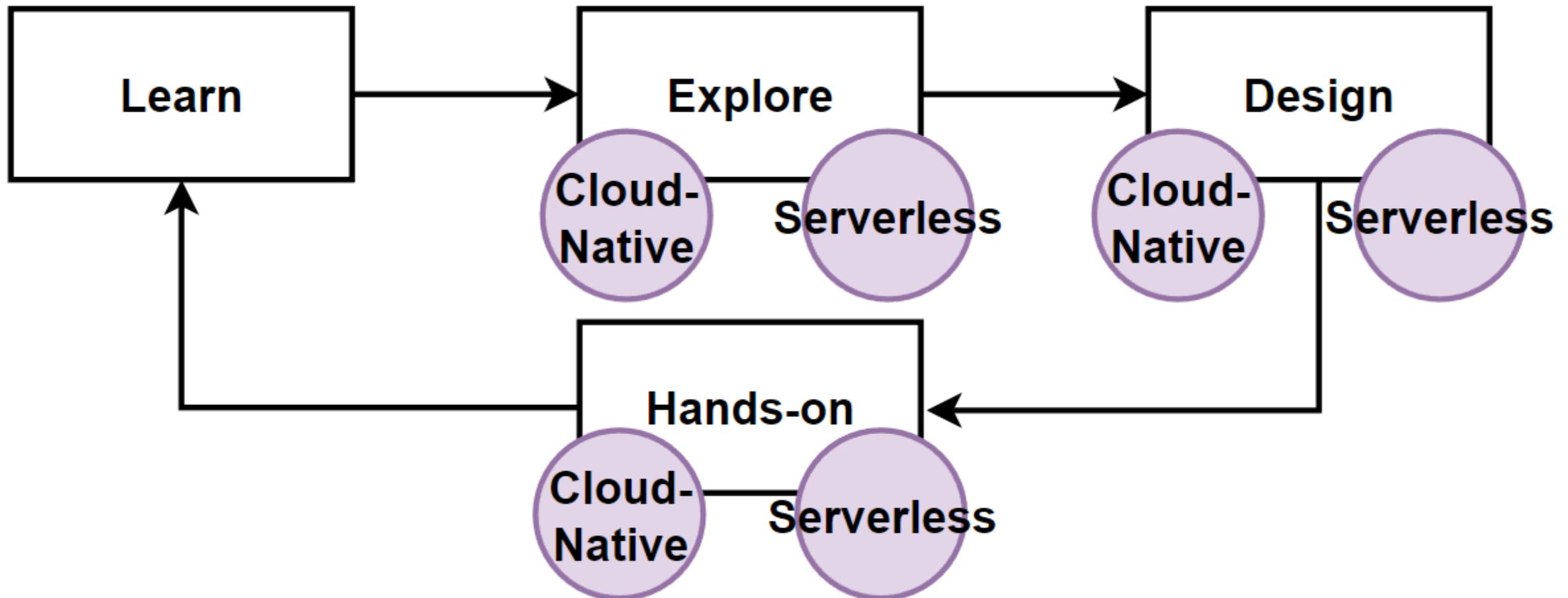
Microservices



Way of Learning – The Course Flow

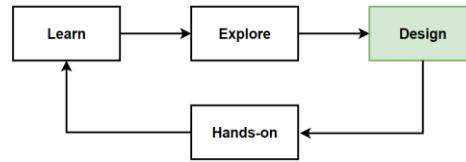


Way of Learning – Cloud-Native & Serverless Cloud Managed



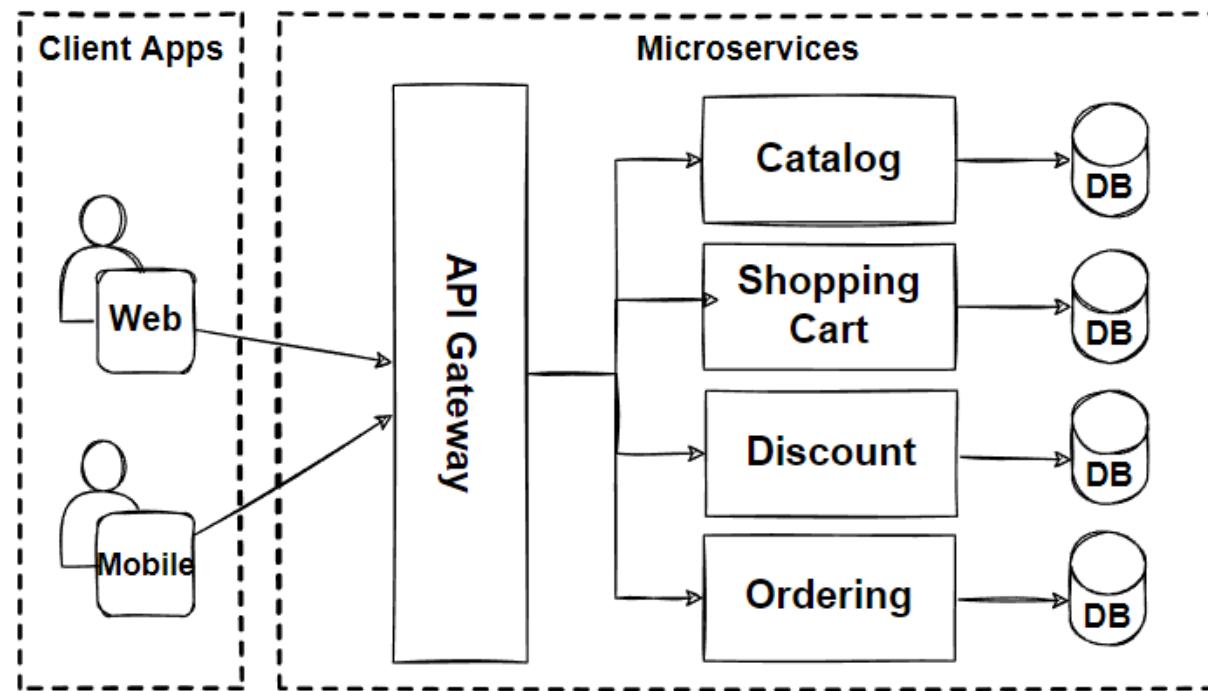
Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Next Step - Understand Domain

- Understand E-Commerce Domain
- Analysis and Decompose E-Commerce Microservices

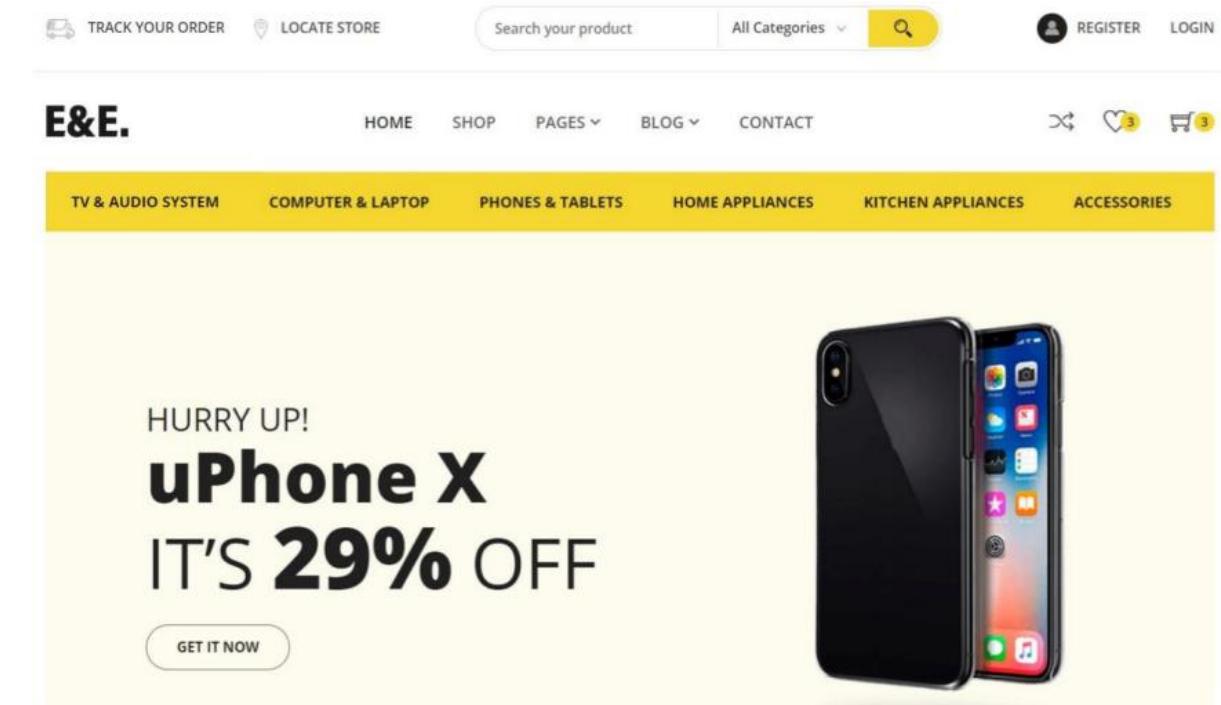


Understand E-Commerce Domain

- Our Domain: E-Commerce
- Understand Domain and Decompose small pieces
 - Use Cases
 - Functional Requirements

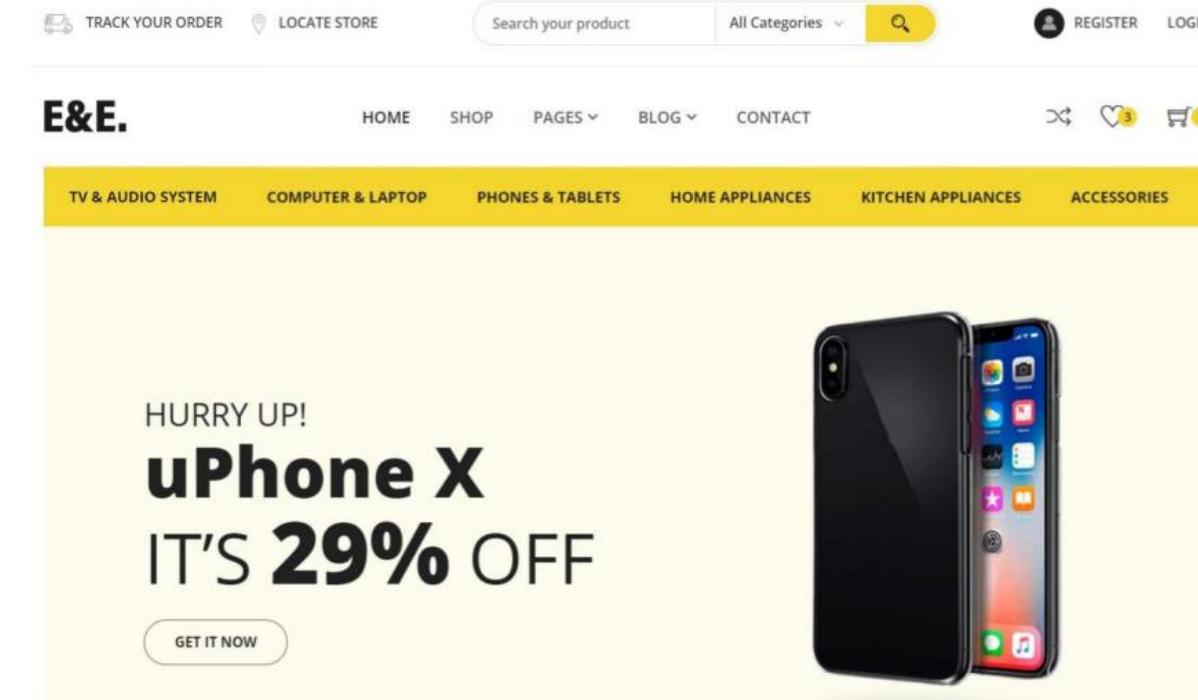
Identify steps:

- Requirements and Modelling
- Identify User Stories
- Identify the Nouns in the user stories
- Identify the Verbs in the user stories



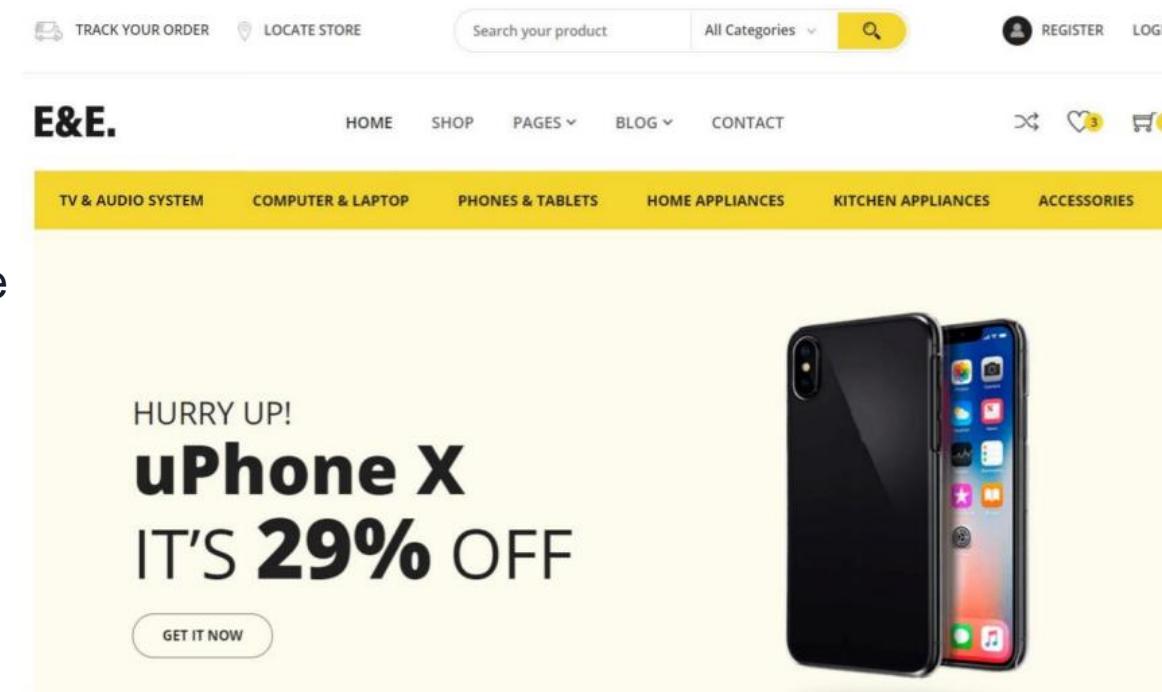
Understand E-Commerce Domain: Functional Requirements

- List products
- Filter products as per brand and categories
- Put products into the shopping cart
- Apply coupon for discounts and see the total cost all for all of the items in shopping cart
- Checkout the shopping cart and create an order
- List my old orders and order items history



Understand E-Commerce Domain: User Stories (Use Cases)

- As a user I want to list products
- As a user I want to filter products as per brand and categories
- As a user I want to put products into the shopping cart so that I can check out quickly later
- As a user I want to apply coupon for discounts and see the total cost all for all of the items that are in my cart
- As a user I want to checkout the shopping cart and create an order
- As a user I want to list my old orders and order items history
- As a user I want to login the system as a user and the system should remember my shopping cart items

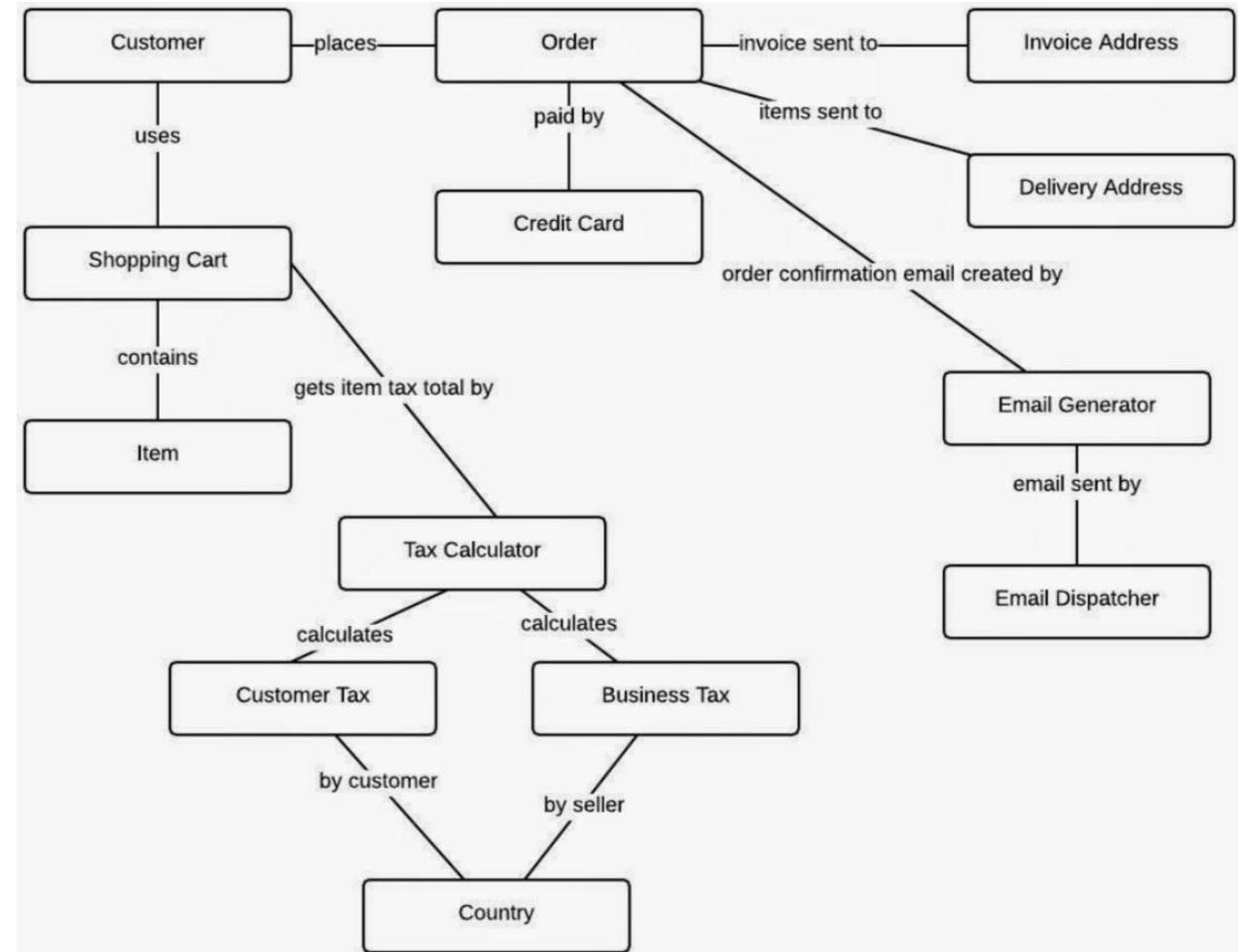


Analysis E-Commerce Domain - Nouns and Verbs

- As a user I want to **list products**
- As a user I want to be able to **filter products** as per **brand** and **categories**
- As a user I want to **see** the supplier of **product** in the product detail screen with all characteristics of product
- As a user I want to be able to **put products** that I want to **purchase** in to the **shopping cart** so I can check out
- As a user I want to **see** the total cost all for all of the **items** that are in my **cart** so that I see if I can afford to buy
- As a user I want to **see** the total cost of each **item** in the **shopping cart** so that I can re-check the price for items
- As a user I want to be able to **specify** the **address** of where all of the products are going to be sent to
- As a user I want to be able to **add** a note to the **delivery address** so that I can provide special instructions
- As a user I want to be able to **specify** my **credit card** information during **check out** so I can pay for the items
- As a user I want system to **tell** me how many items are in **stock** so that I know how many items I can purchase
- As a user I want to **receive order** confirmation email with **order** number so that I have proof of purchase
- As a user I want to **list** my old **orders** and **order items history**
- As a user I want to **login** the system as a **user** and the system should remember my shopping cart items

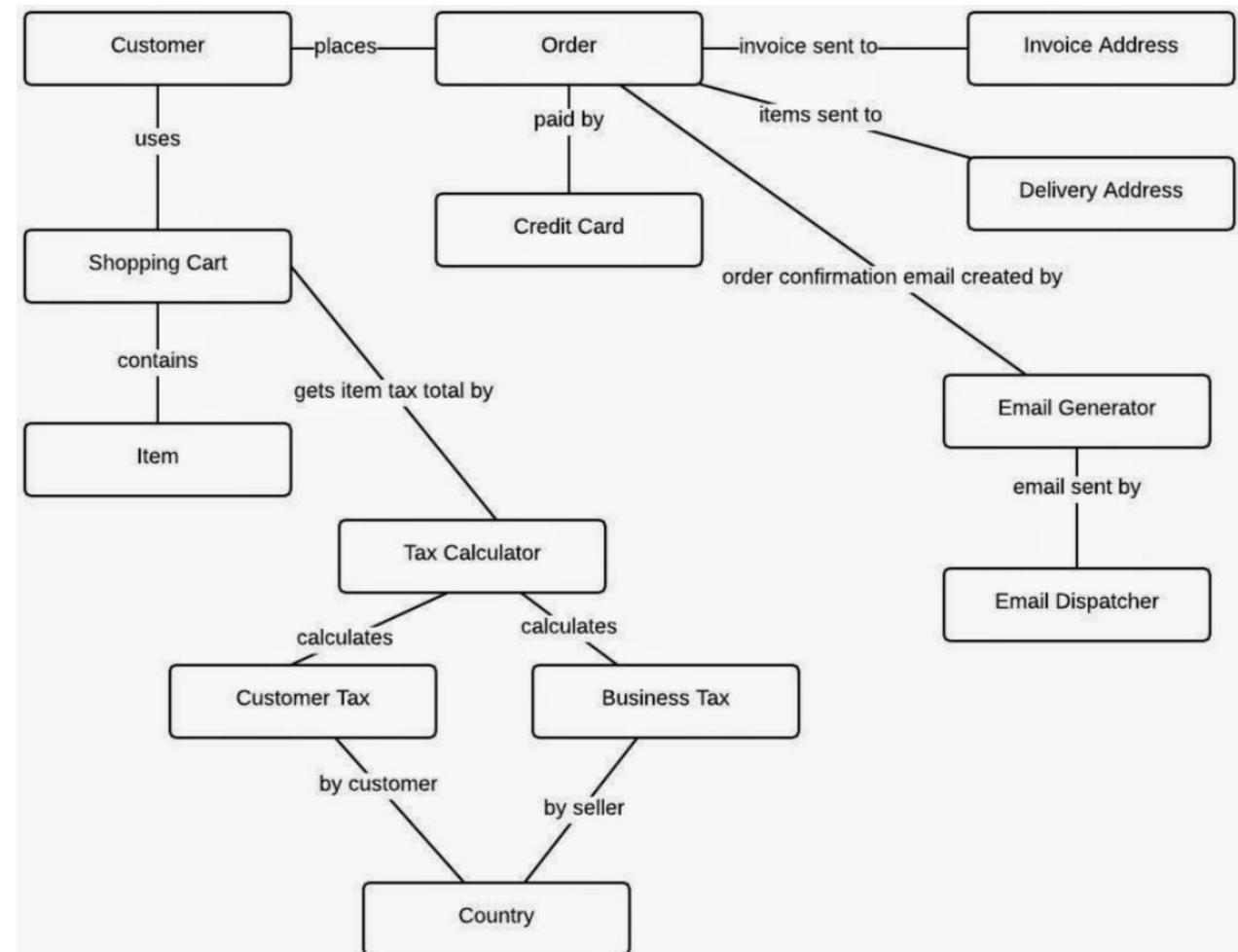
Analysis E-Commerce Domain - Nouns

- Customer
- Order
- Order Details
- Product
- Shopping Cart
- Shopping Cart Items
- Supplier
- User
- Address
- Brand
- Category

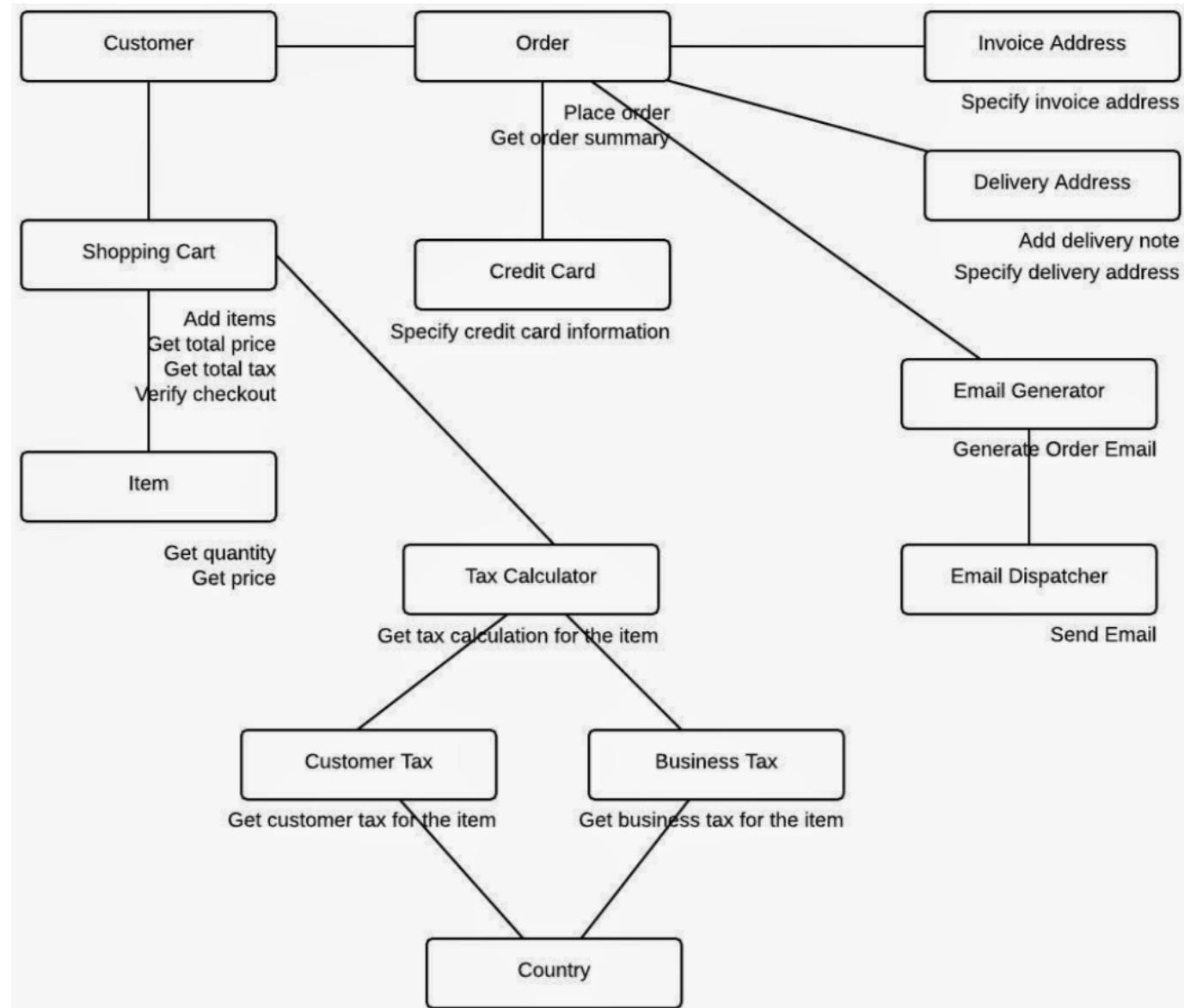


Analysis E-Commerce Domain - Verbs

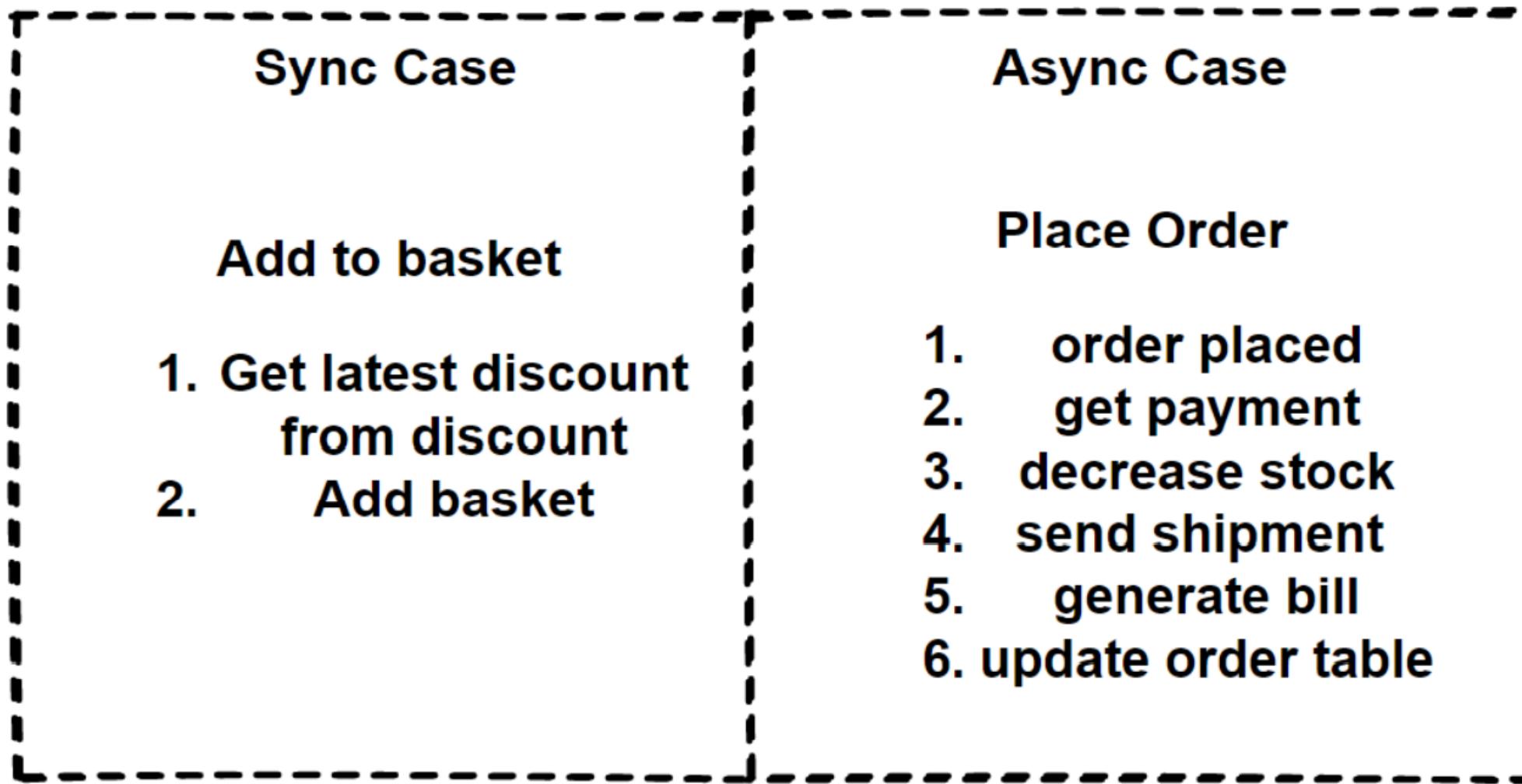
- List products applying to paging
- Filter products by brand, category and supplier
- See product all information in the details screen
- Put products in to the shopping cart
- See total cost for all of the items
- See total cost for each item
- Checkout order with purchase steps
- Specify delivery address
- Specify delivery note for delivery address
- Specify credit card information
- Pay for the items
- Tell me how many items are in stock
- Receive order confirmation email
- List the order and details history
- Login the system and remember the shopping cart items



Object Responsibility Diagram



2 Main Use Case of Our E-Commerce Application



Identifying and Decomposing Microservices for E-Commerce Domain

Main Microservices

Users

Product

Customers

Shopping Cart

Discount

Orders

Order Transactional Microservices

Orders

Payment

Inventory

Shipping

Billing

Notification

Intelligence Microservices

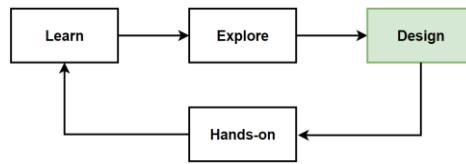
Identity

Marketing

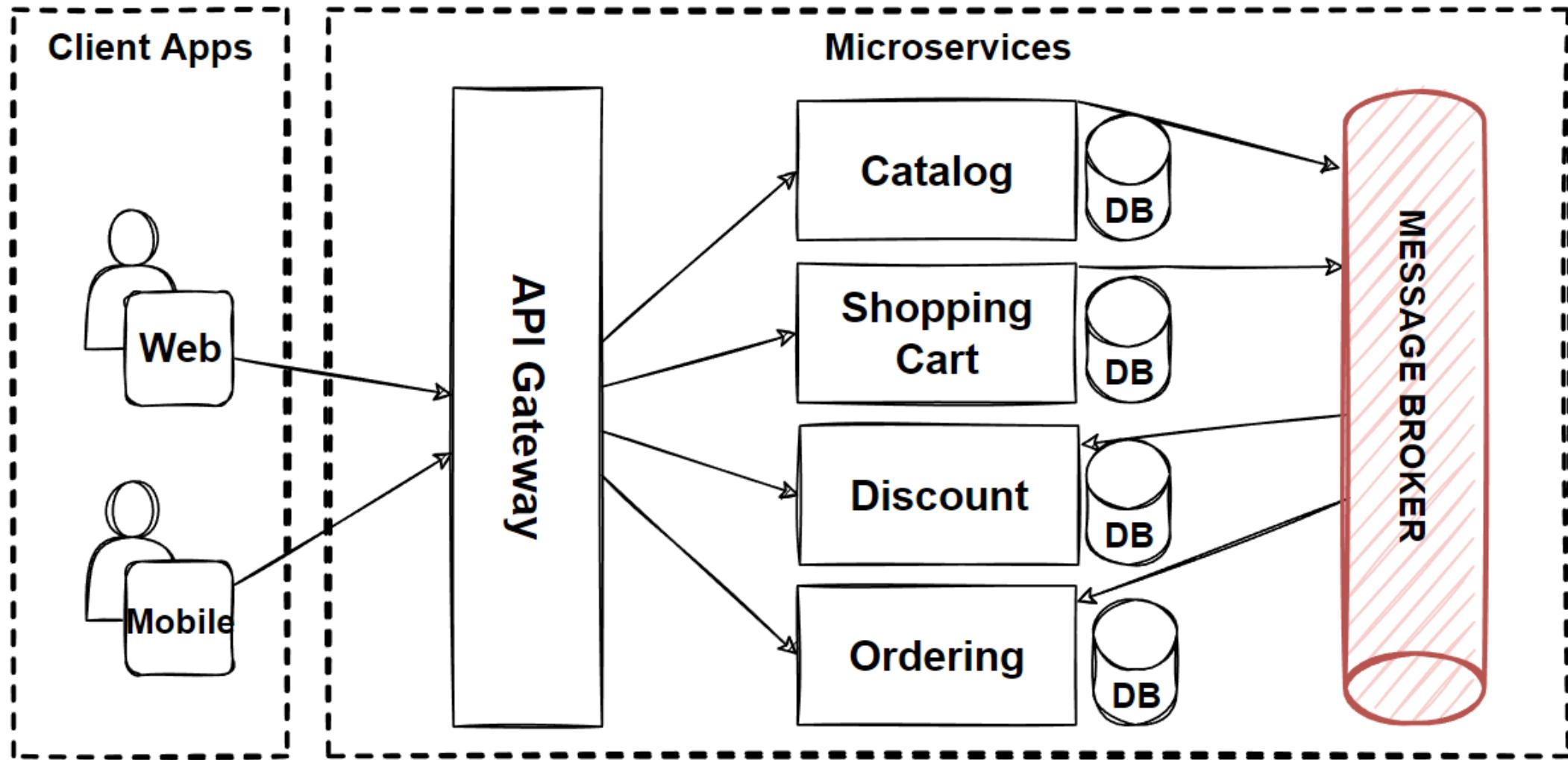
Location

Rating

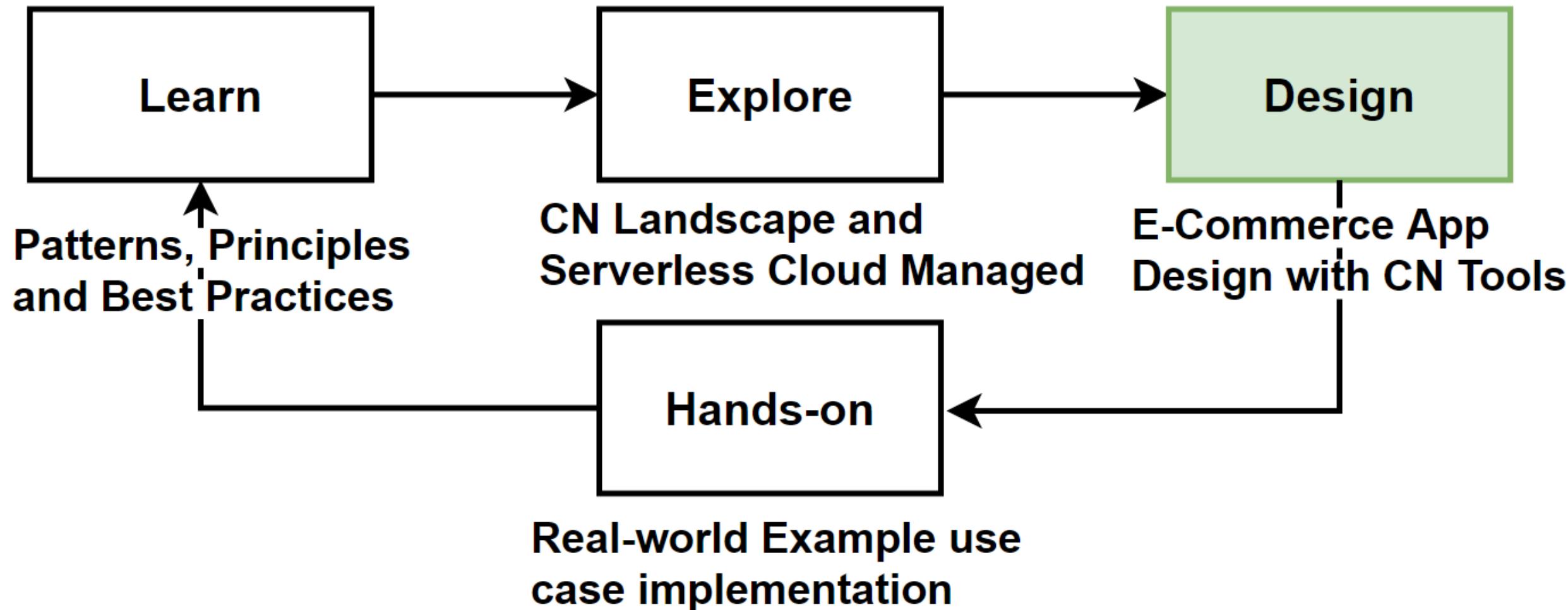
Recommendation



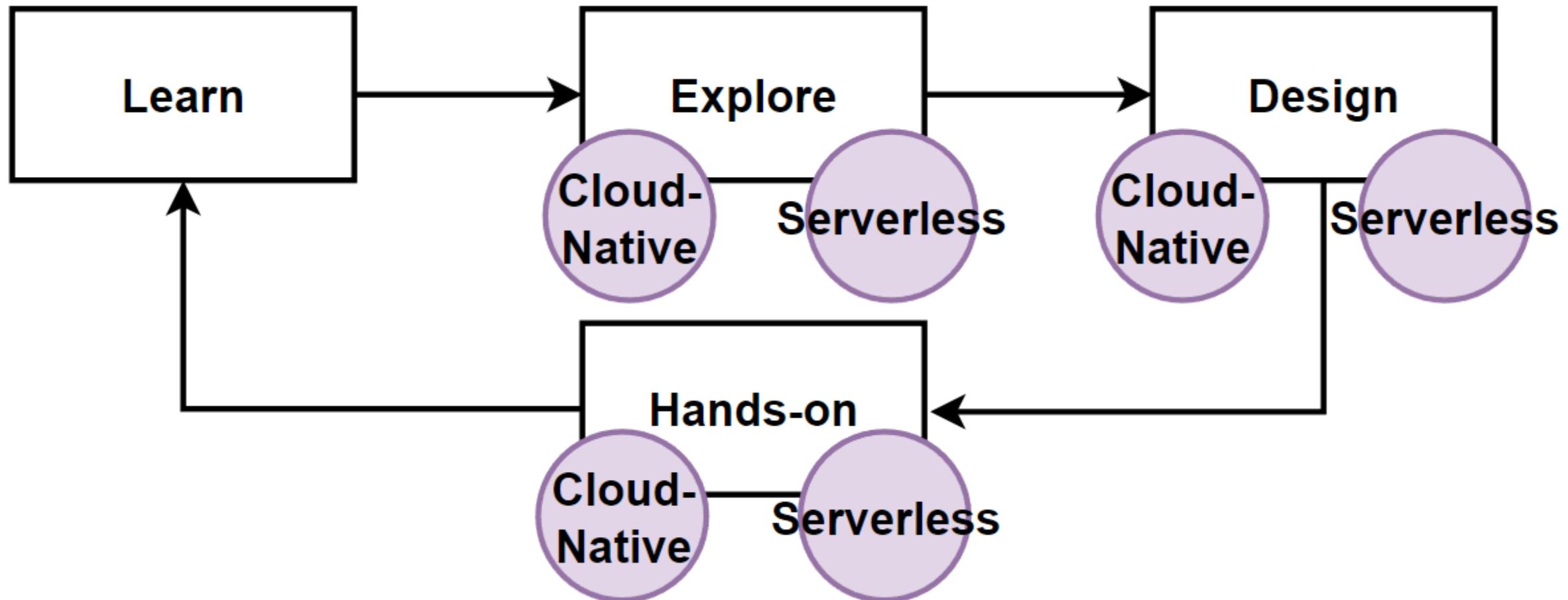
Design: Cloud-Native E-commerce Microservices



Way of Learning – The Course Flow



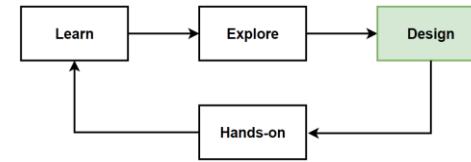
Way of Learning – Cloud-Native & Serverless Cloud Managed



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

Choose Microservices Language and Frameworks: Cloud-Native Microservices E-commerce App



Frontend SPAs

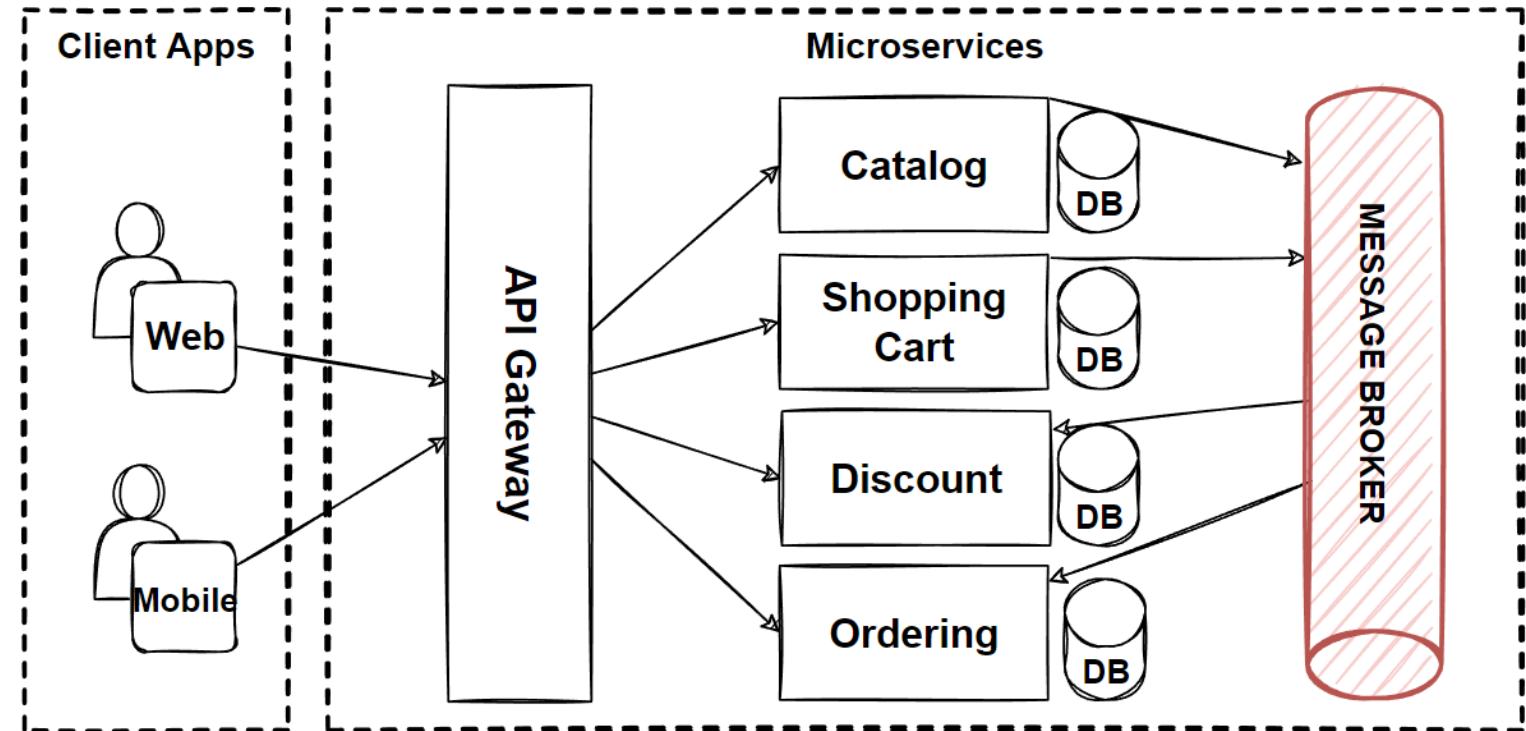
- Angular
- Vue
- React

Backend Microservices

- Java – Spring Boot
- .Net – Asp.net
- JS – NodeJS
- Python – Django, Flask

Database

- MongoDB – NoSQL Document DB
- Redis – NoSQL Key-Value Pair
- Postgres – Relational
- SQL Server – Relational

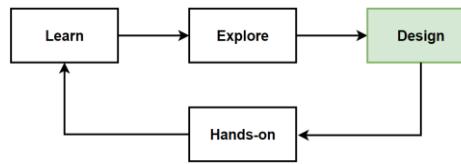


API Gateways

- Kong Gateway
- Tyk API Gateway
- KubeGateway

Message Brokers Event Bus

- Apache Kafka
- RabbitMQ
- Redis Event Bus



Choose Microservices Language and Frameworks: Serverless Microservices E-commerce App

Frontend SPAs

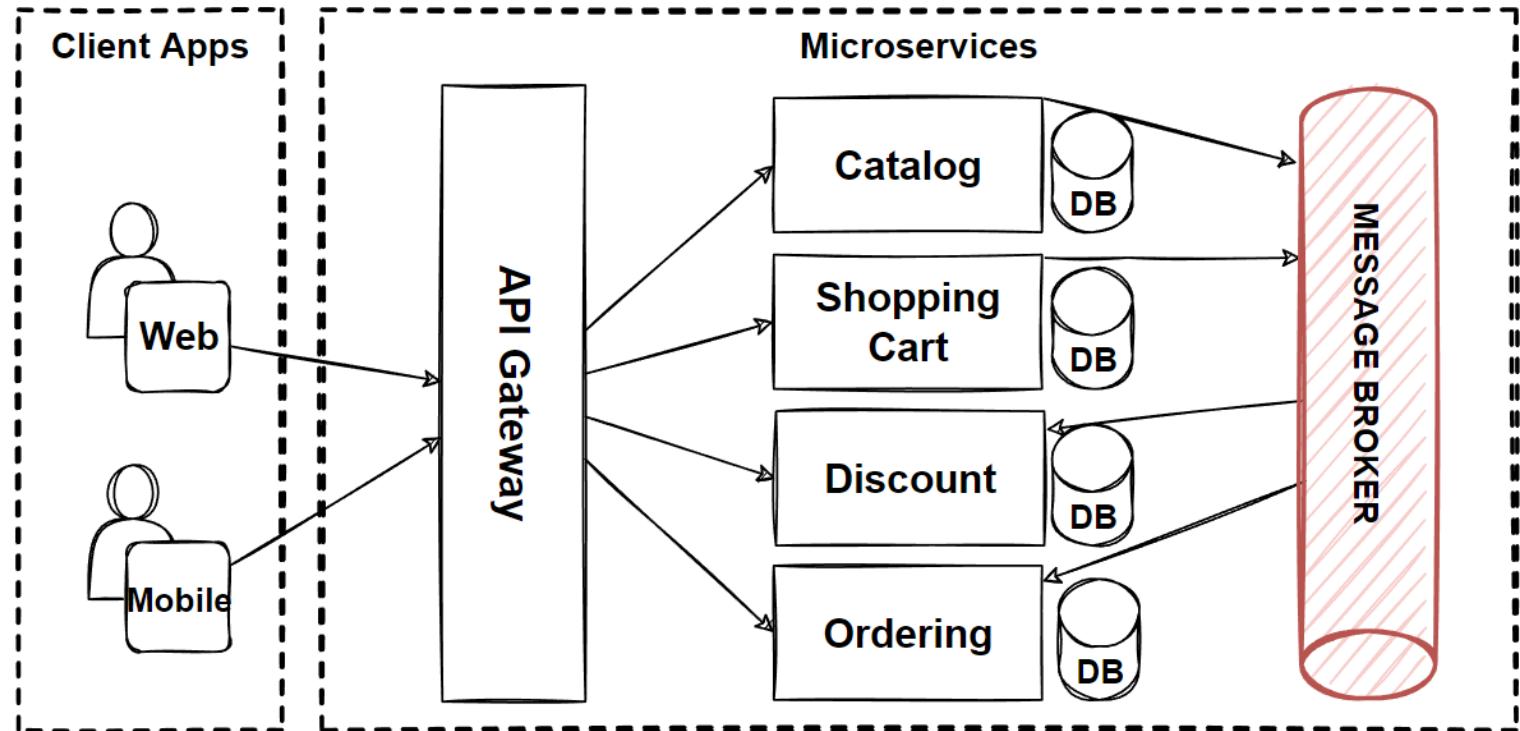
- Vercel NextJS
- Amazon Amplify

Backend Microservices

- AWS Lambda
- Azure Functions
- Google Cloud Functions
- Vercel Edge

Database

- Amazon DynamoDB
- Azure CosmosDB
- Upstash Redis



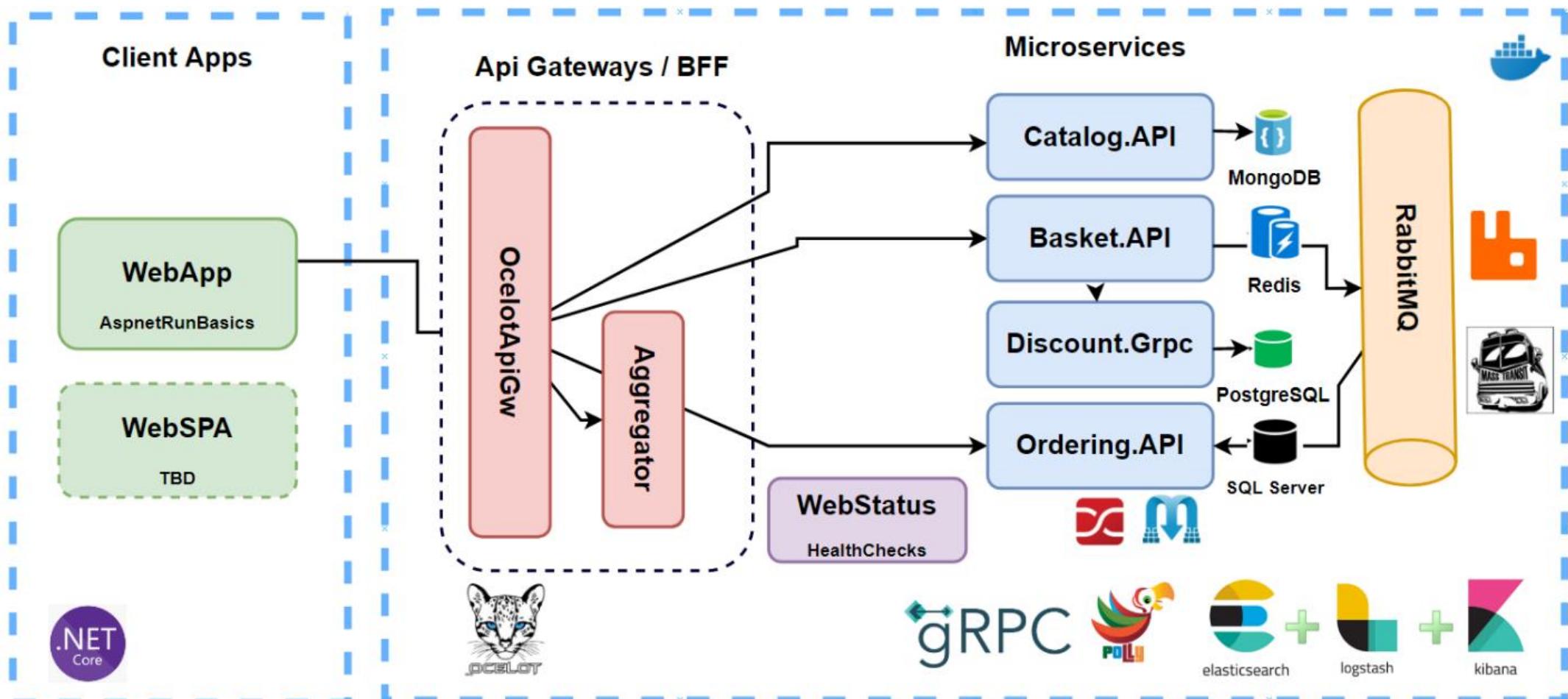
API Gateways

- Amazon API Gateway
- Azure Gateway

Message Brokers Event Bus

- Amazon EventBridge, SNS
- Azure Service Bus

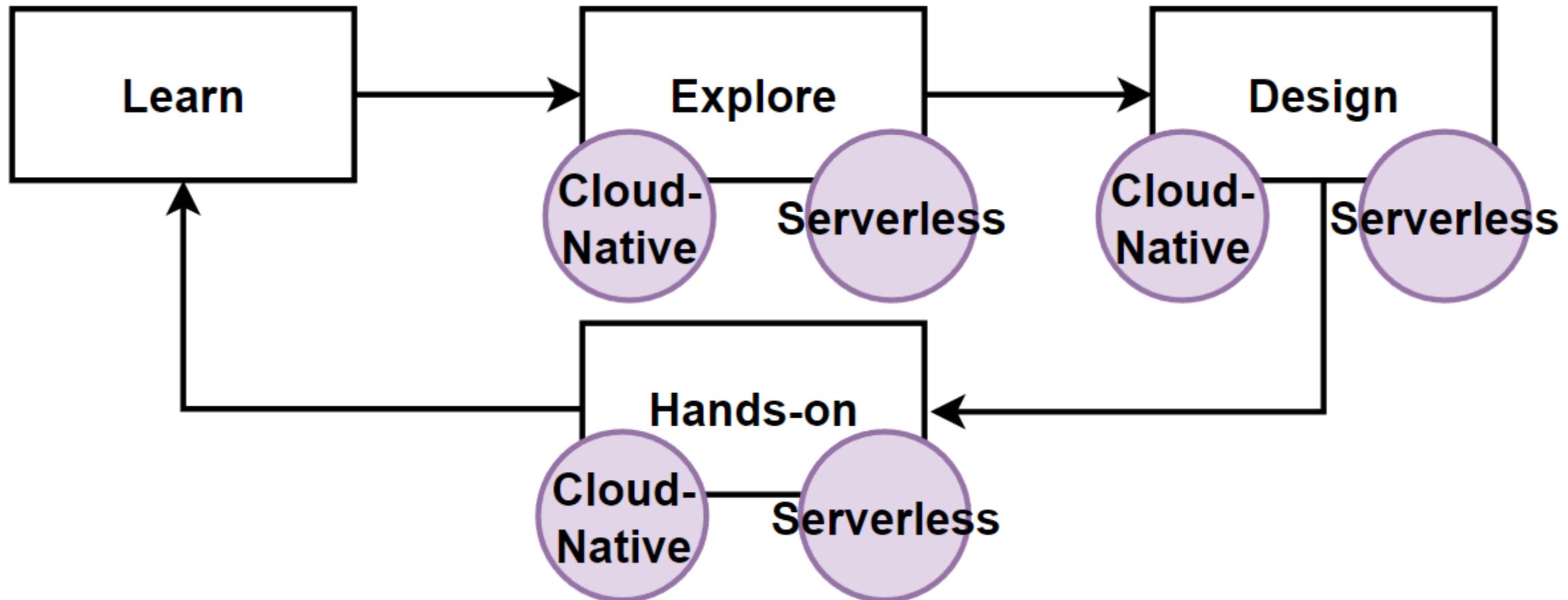
Reference Project: .Net Microservices - Cloud-Native E-commerce



DEMO: Code review of Microservices Architecture .NET Implementation

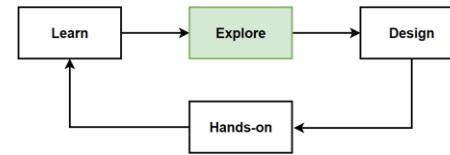
- <https://github.com/aspnetrun/run-aspnetcore-microservices>
- <https://github1s.com/aspnetrun/run-aspnetcore-microservices>

Way of Learning – Cloud-Native & Serverless Cloud Managed



Examples of CN vs Serverless Cloud Tools:

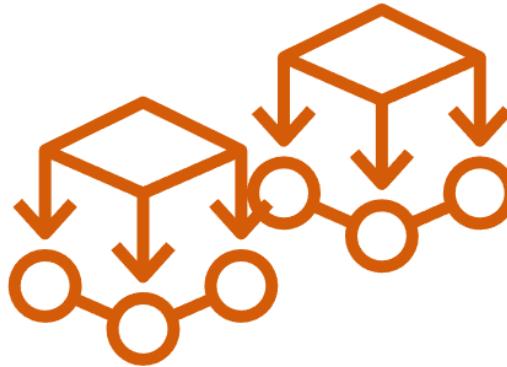
- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Cloud Serverless Functions for Microservices

What are Serverless Functions?

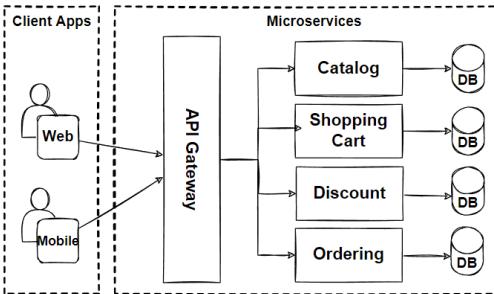
- Serverless functions are single-purpose, programmatic functions hosted on managed infrastructure.
- They are invoked via the internet, are fully scalable, and have built-in fault tolerance.
- These are ideal for variable workloads, removing the overhead of maintaining a full-time server.



Benefits in Microservices Development

- Aligns with the microservices philosophy of small, loosely coupled, and independently deployable units.
- Each microservice can be a serverless function, independently scalable and deployable.
- Developers can focus on writing business logic, not on infrastructure management.

Microservices



AWS Event-driven Serverless Services

Compute



AWS Lambda

Databases



Amazon DynamoDB

API Management



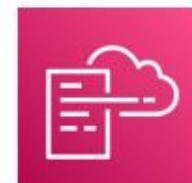
Amazon API Gateway

Messaging/Application Integrations



Amazon EventBridge

IaC



AWS
CloudFormation

Monitoring



Amazon
CloudWatch



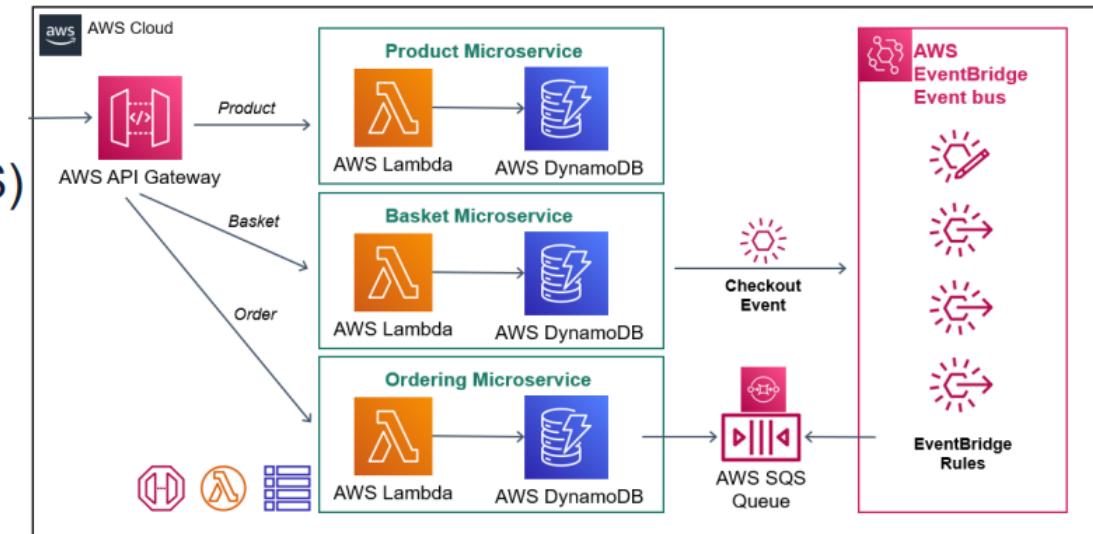
Amazon Simple Queue
Service (Amazon SQS)



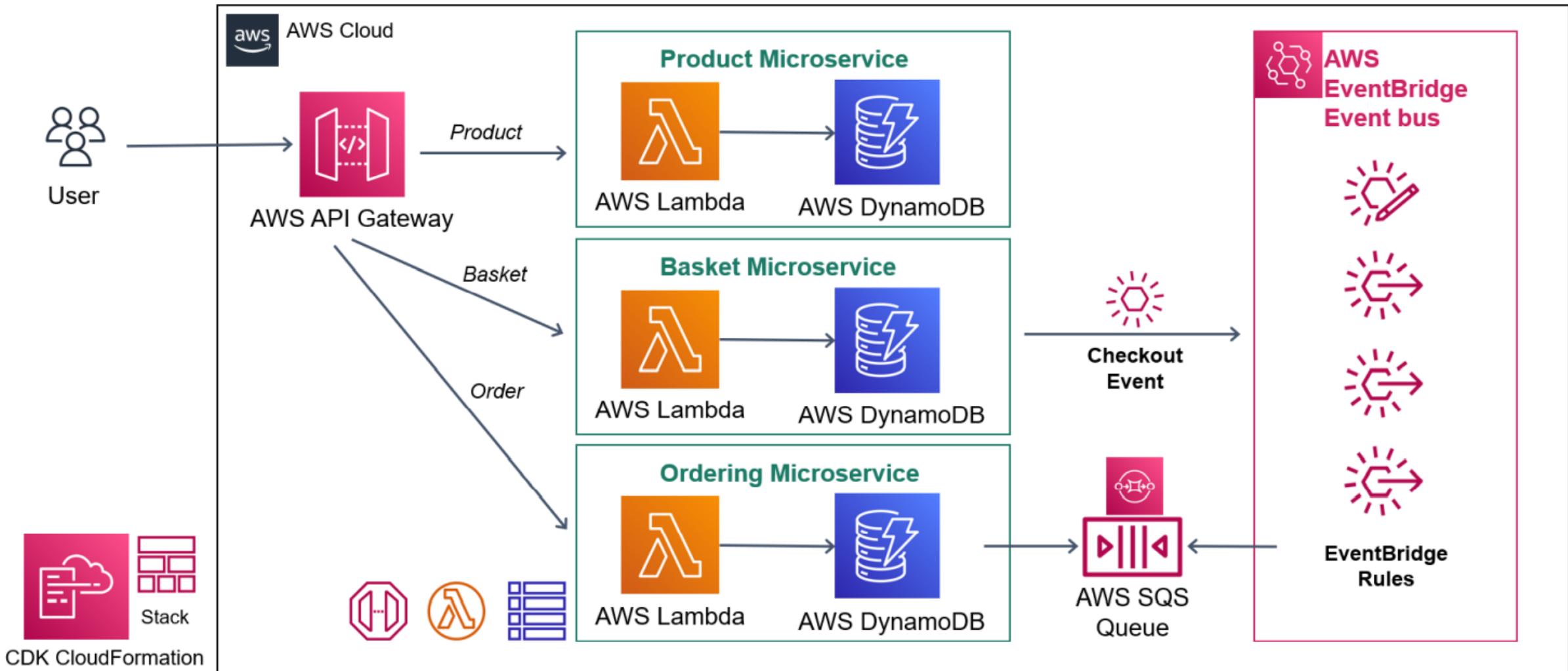
AWS CDK
Stack

AWS Event-driven Serverless Microservices

- REST API and CRUD endpoints (AWS Lambda, API Gateway)
- Data persistence (AWS DynamoDB)
- Decouple Microservices with event (AWS EventBridge)
- Message Queues for cross-service communication (AWS SQS)
- Cloud stack development with IaC (AWS CloudFormation CDK)
- Event-driven Serverless Microservices
- Follow the **Serverless Design Patterns** and **Best Practices**
- Developing **E-commerce Event-driven Microservices** application

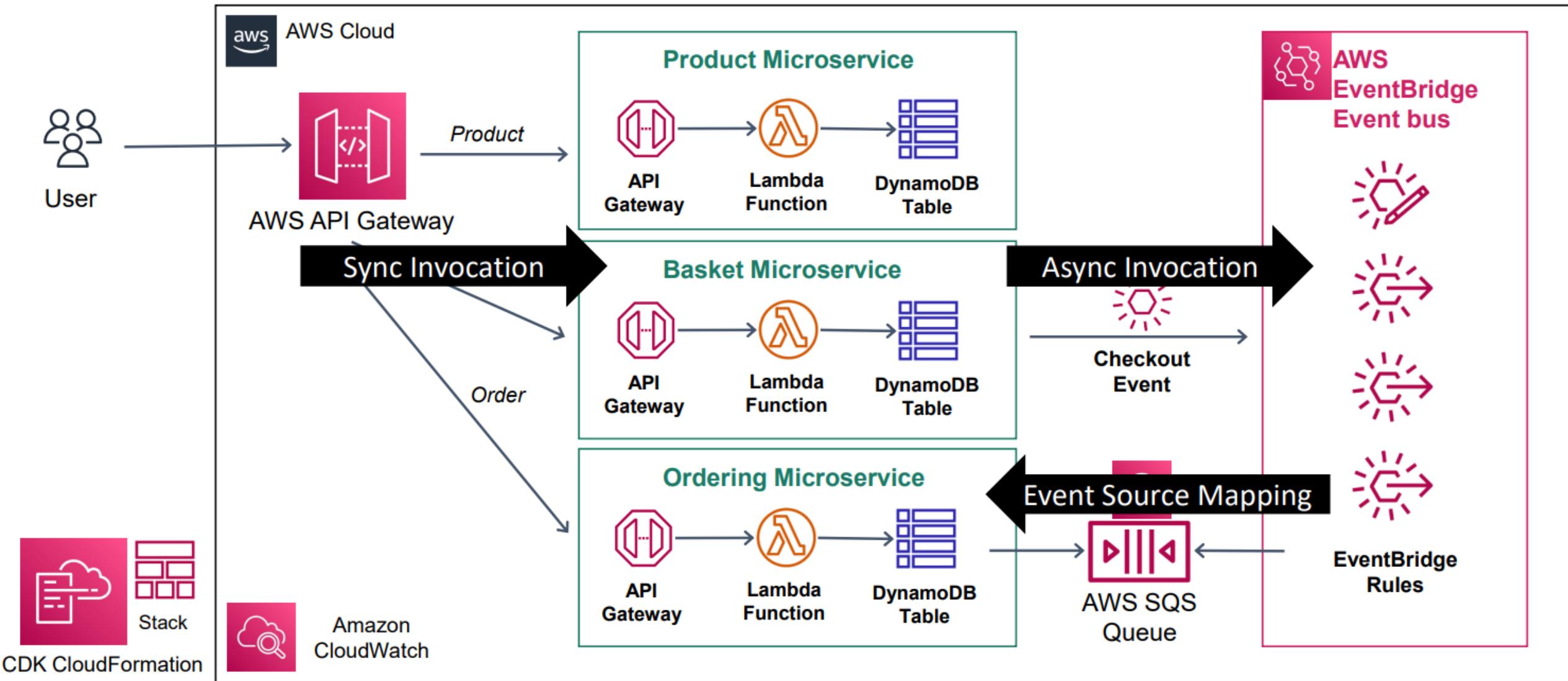


Reference Project: AWS Serverless E-Commerce Microservices



- <https://github.com/awsrun/aws-microservices>
- <https://github1s.com/awsrun/aws-microservices>

AWS Serverless E-Commerce Microservices Communications



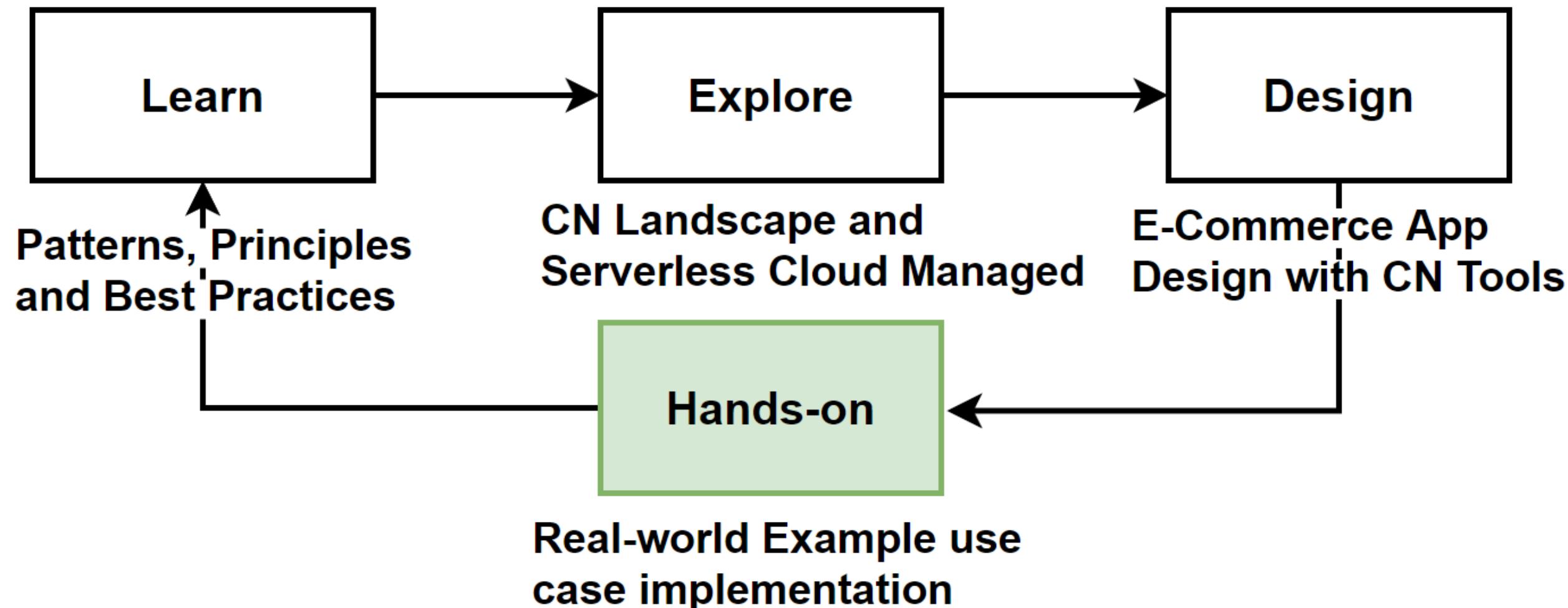
- <https://github.com/awsrun/aws-microservices>
- <https://github1s.com/awsrun/aws-microservices>

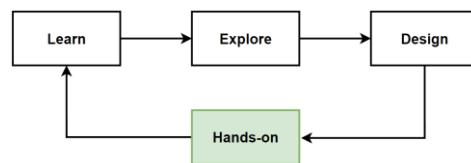
Hands-on: Develop a RESTful Microservices with CRUD endpoints

RESTful API design for Single Microservice

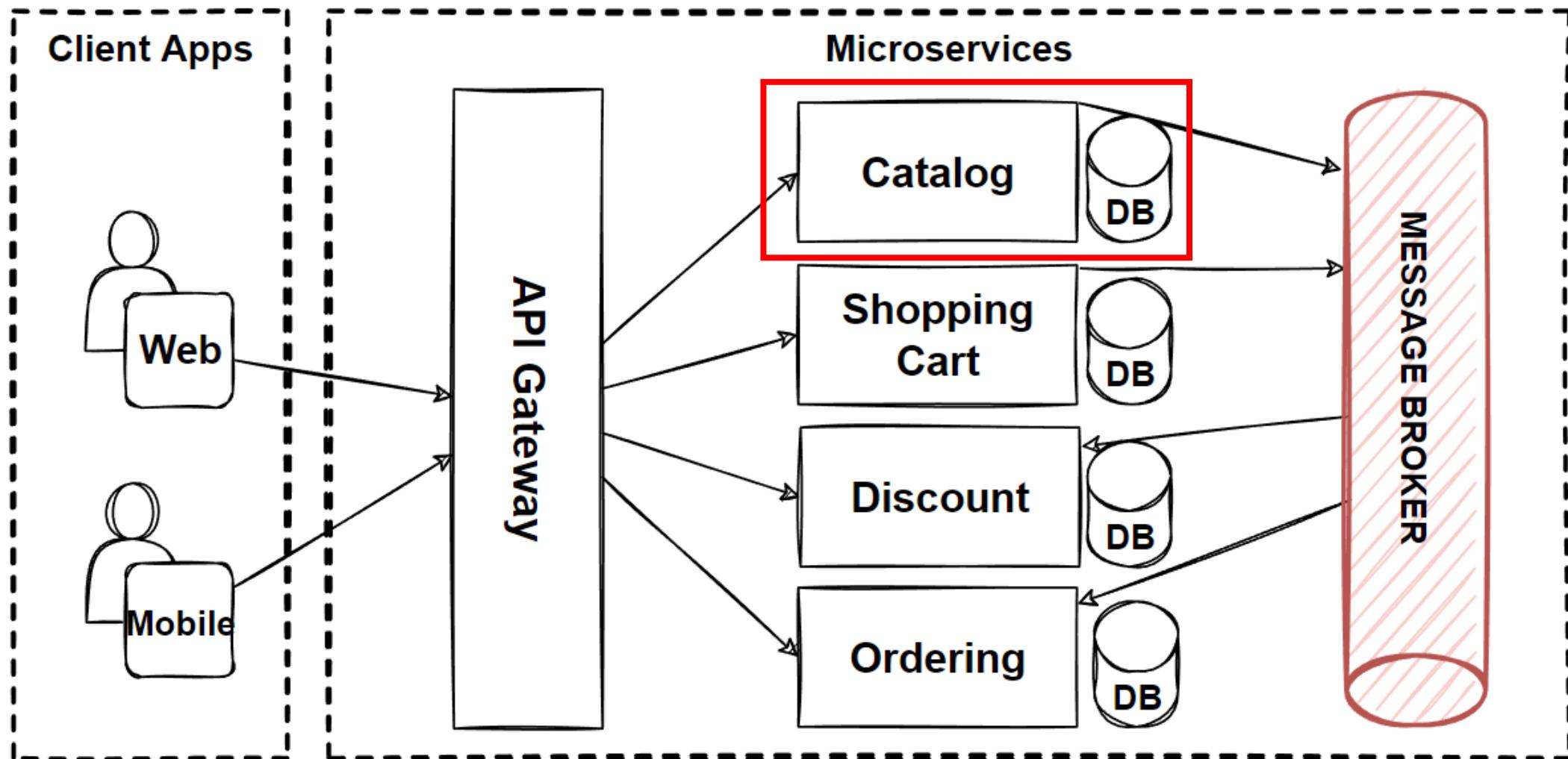
REST API and CRUD endpoints with using Microservices

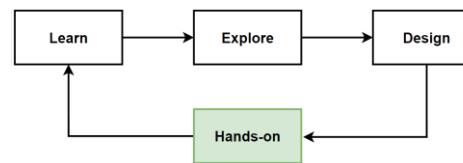
Way of Learning – The Course Flow



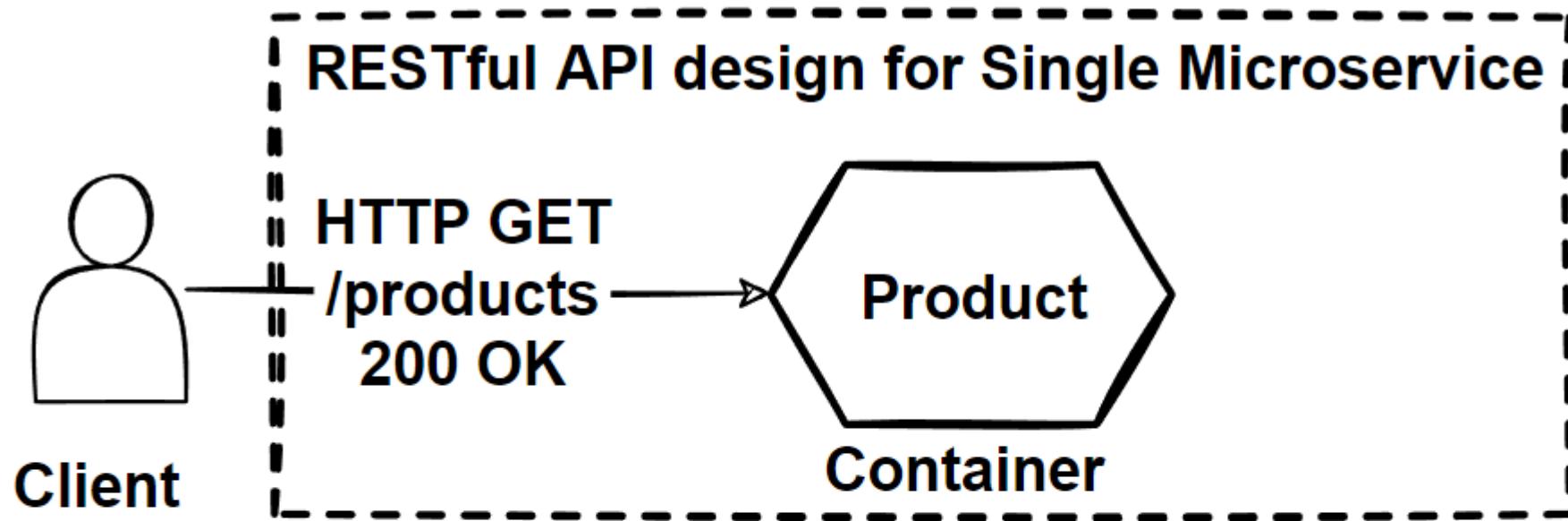


Design: Cloud-Native E-commerce Microservices





Hands-on: Develop a RESTful Microservices with CRUD



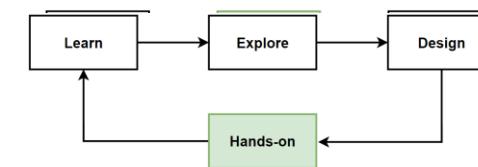
GET /api/products: Retrieves all products.

GET /api/products/{id}: Retrieves a specific product by ID.

POST /api/products: Adds a new product.

PUT /api/products/{id}: Updates an existing product by ID.

DELETE /api/products/{id}: Deletes a specific product by ID.



Microservices Languages and Frameworks

Java – Spring Boot

- Java is a popular choice for enterprise-grade applications
- Spring Boot's convention-over-configuration simplifies development

JavaScript – Node.js

- Node.js allows server-side JavaScript development
- Its event-driven, non-blocking I/O model is well-suited for microservices

Python – Flask, Django

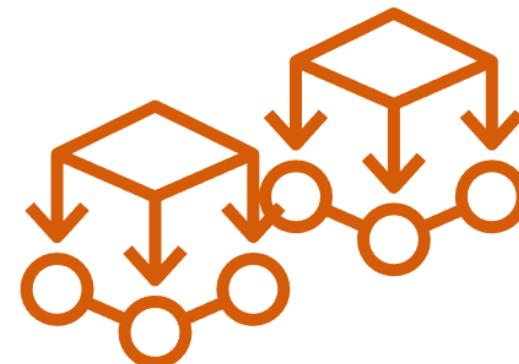
- Python's simplicity make it good for microservices, especially in data-intensive apps
- Flask is a lightweight framework suitable for simple microservices

Go (Golang) – GoMicro

- Go is a statically-typed, compiled language developed by Google
- Go's efficiency, and excellent concurrency support make it increasingly popular for services

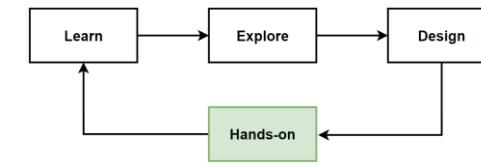
C# – .NET

- C# is a modern, object-oriented programming language that is widely used for building web applications, and microservices.
- .NET is a cross-platform, open-source framework for building modern, cloud-native applications, including microservices.

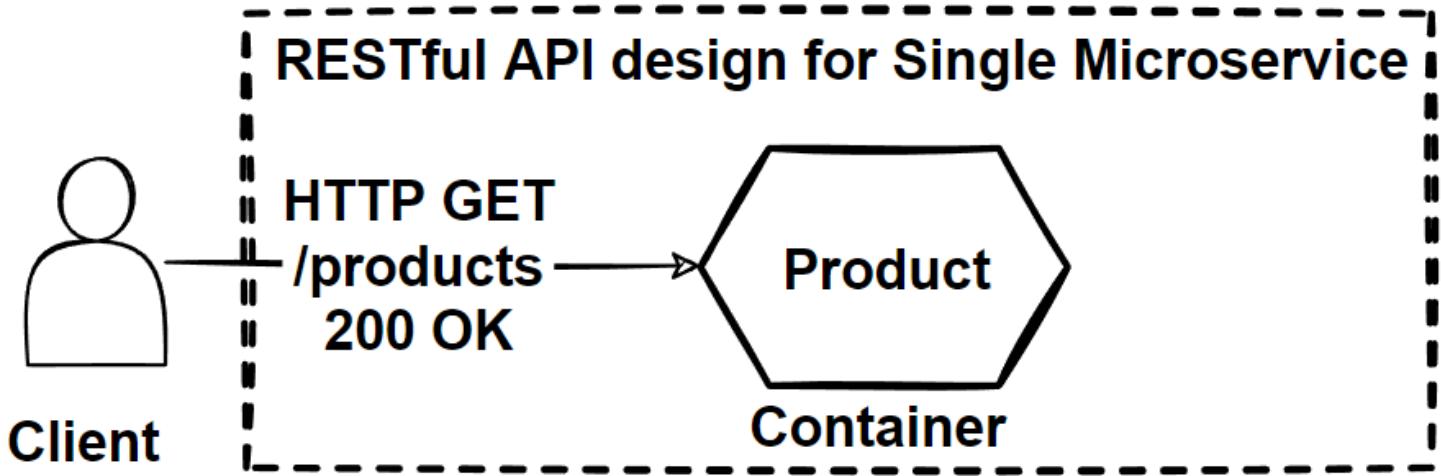


Microservices

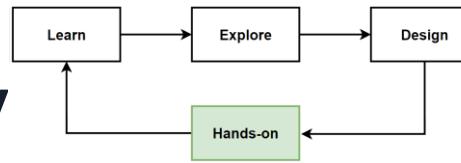
Which Programming Languages and Frameworks for Microservices Developments ?



- Java – Spring Boot
- JavaScript – Node.js
- Python – Flask, Django
- Go (Golang) – GoMicro
- C# – .NET

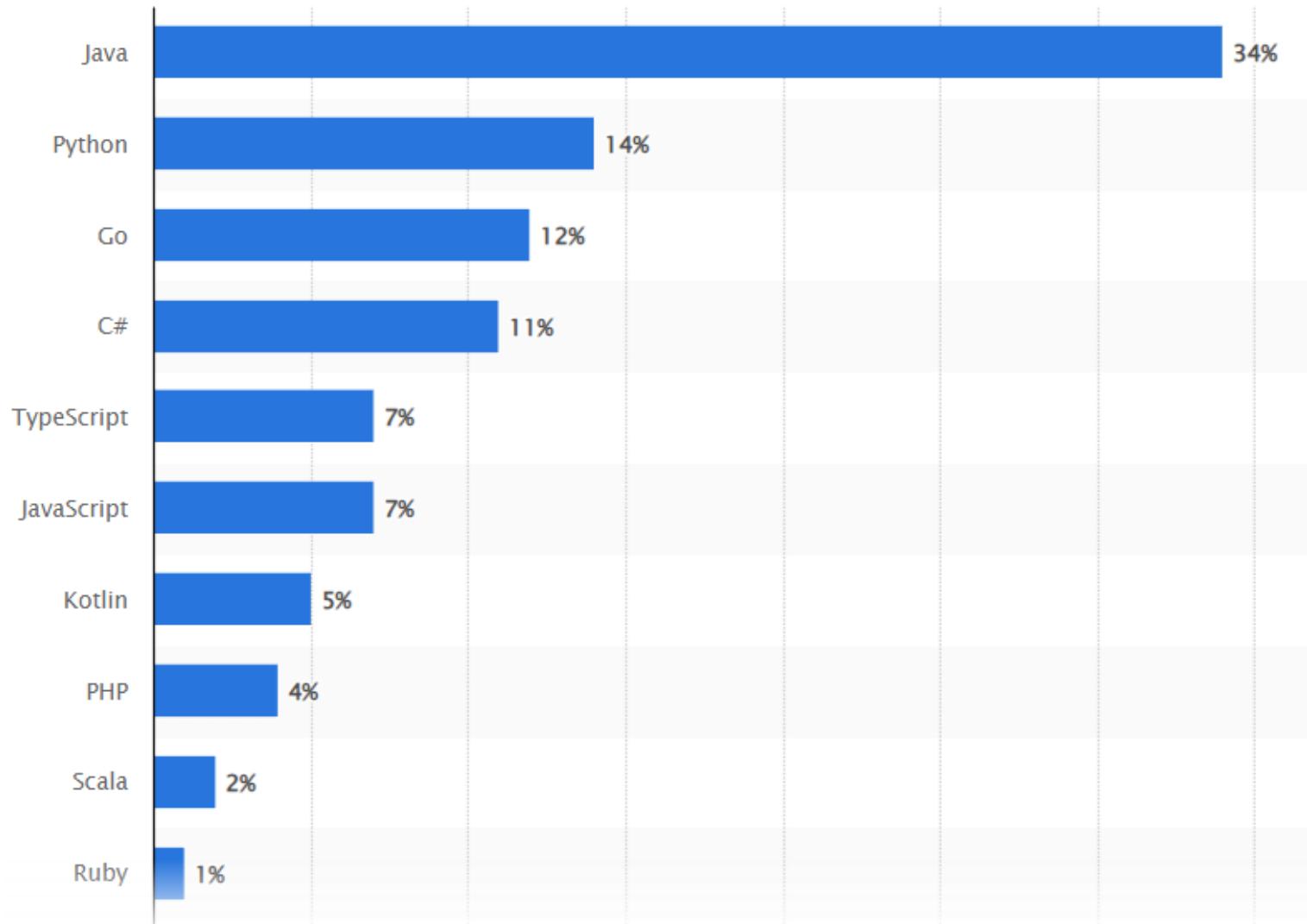


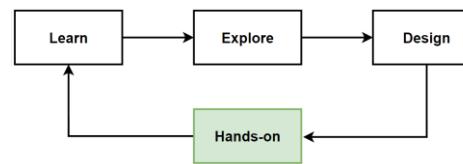
GET /api/products: Retrieves all products.
GET /api/products/{id}: Retrieves a specific product by ID.
POST /api/products: Adds a new product.
PUT /api/products/{id}: Updates an existing product by ID.
DELETE /api/products/{id}: Deletes a specific product by ID.



Statista - Primary programming languages among microservices dev worldwide in 2022

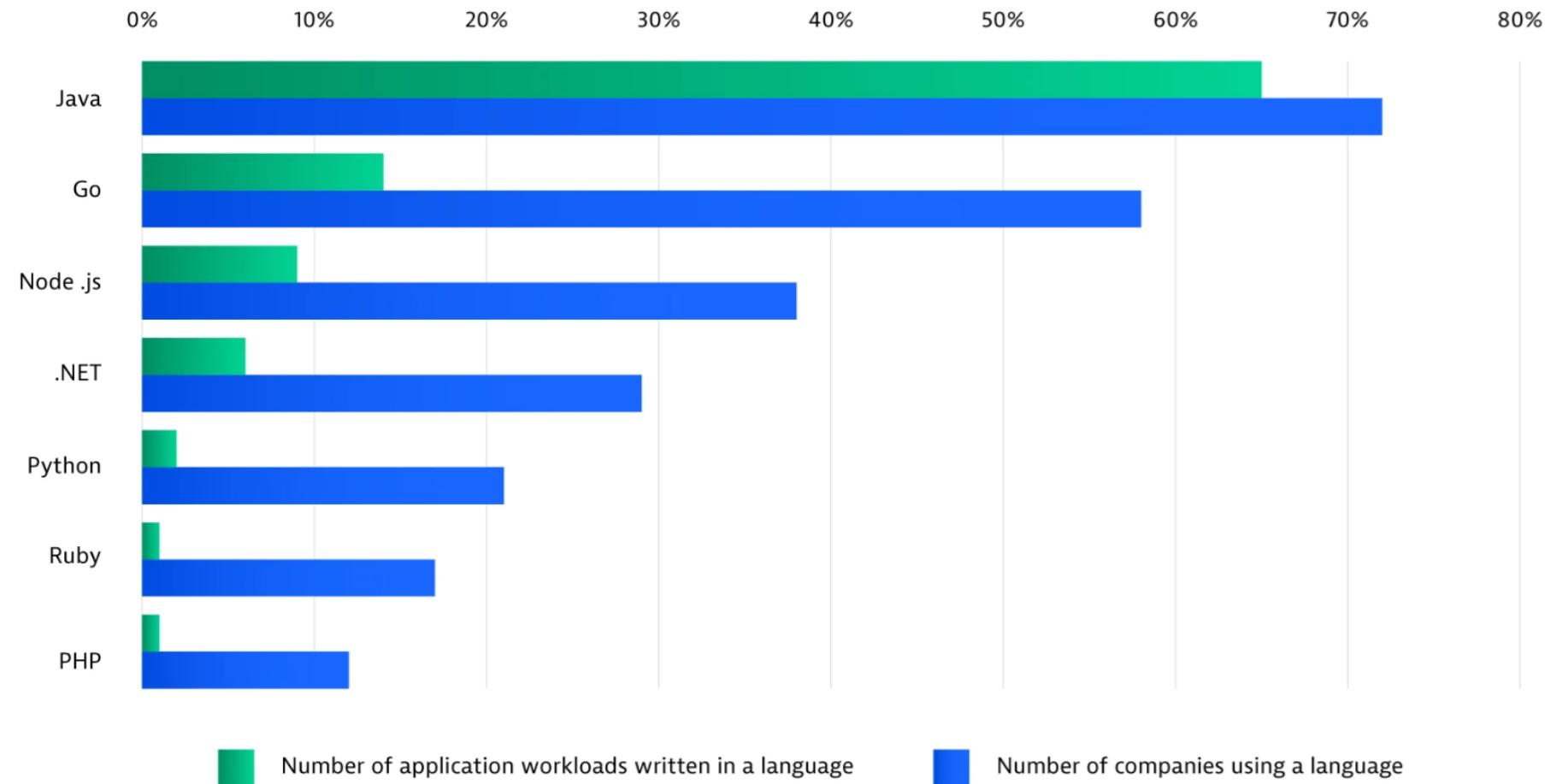
- Java – Spring Boot
- Python – Flask, Django
- Go (Golang) – GoMicro
- C# – .NET





Dynatrace - Top Kubernetes Programming Languages 2023

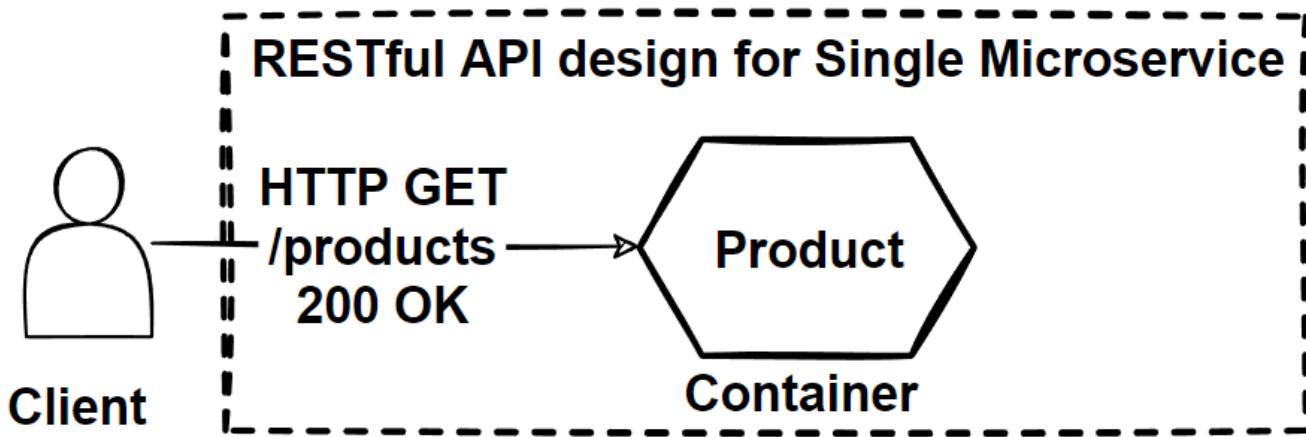
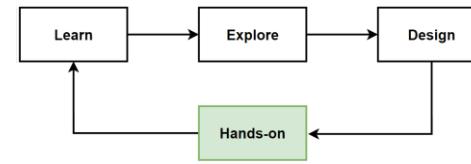
- Java – Spring Boot
- Go (Golang) – GoMicro
- JavaScript – Node.js
- C# – .NET



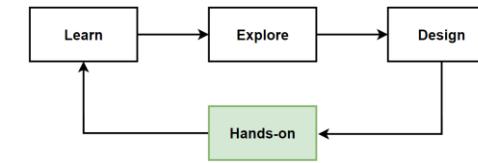
<https://www.dynatrace.com/news/blog/kubernetes-in-the-wild-2023/>

Which Programming Languages and Frameworks for Microservices Developments ?

- Java – Spring Boot
 - JavaScript – Node.js
 - Python – Flask, Django
 - Go (Golang) – GoMicro
 - C# – .NET
-
- Pick any of these languages and framework which is more familiar for you
 - Containerize with Docker and deploy to Kubernetes
 - Doesn't matter which language you develop a microservices since you can containerize it.



GET /api/products: Retrieves all products.
GET /api/products/{id}: Retrieves a specific product by ID.
POST /api/products: Adds a new product.
PUT /api/products/{id}: Updates an existing product by ID.
DELETE /api/products/{id}: Deletes a specific product by ID.



.NET for Microservices Development

Why .NET for Microservices

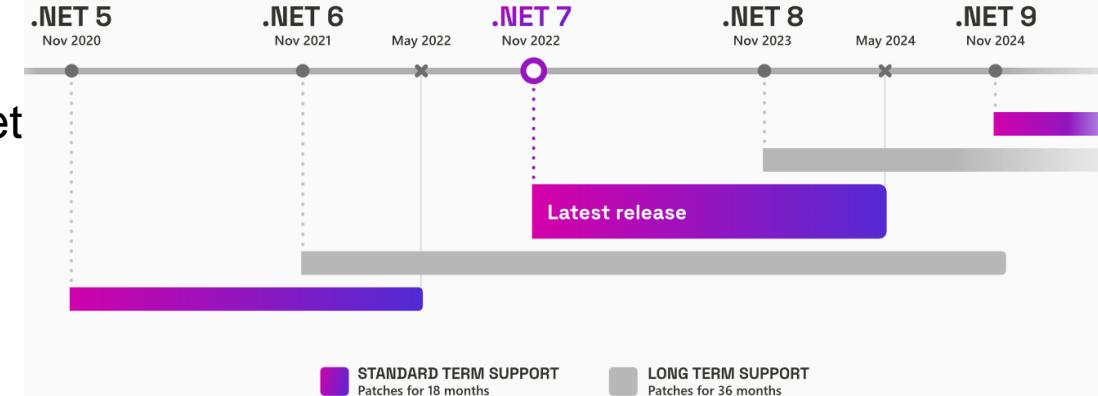
- Free, open-source and cross-platform
- Robust and versatile framework
- Microsoft heavily invest cloud-native development tools with .net
- Rich ecosystem and strong community support
- Ideal for building scalable, performant microservices

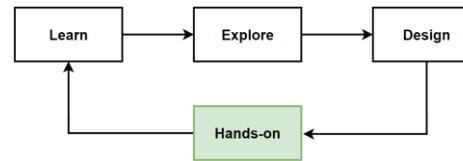
Cross-Platform Support

- Support for Windows, Linux, and macOS
- Build and deploy microservices across various platforms and cloud providers
- Versatile choice for diverse environments

High Performance

- Consistent improvement in .NET's performance
- Scores well in benchmark tests compared to other platforms
- JIT and AOT compilation features ensure fast startup and execution times
- [Link: TechEmpower performance benchmarks](#)
- [Native AOT deployment boosts performance on Linux machines](#)





.NET for Microservices Development - 2

Easy Containerization

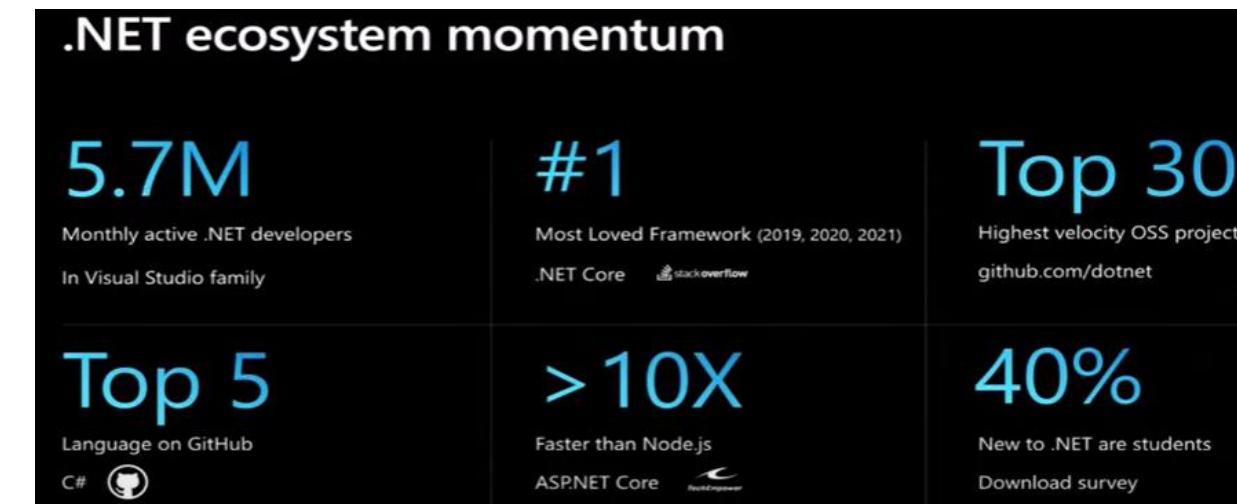
- Simplified container customization and deployment with «[dotnet publish command](#)»
- Seamless integration with Docker, Podman, Containerd
- Integrated deployment process into Azure services
- Easy setup for CI/CD with GitHub Actions

Built-In Support for Microservices

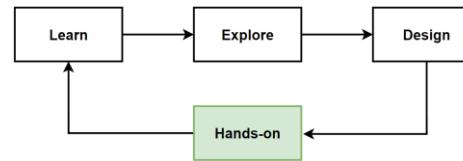
- ASP.NET Core for building web applications and APIs
- Integration with container and orchestration tools
- Horizontal and vertical scalability

Rich Ecosystem & Strong Community Support

- Wide array of libraries, frameworks, and tools
- Active developer community contributing to .NET's growth and development



- Over 5.7 million developers use .NET monthly
- Over 28,000 contributions from 10,000 community members for .NET 7
- An ideal choice for cloud-native microservices
- Simplified containerization and seamless integration with Cloud



Install Prerequisites

Download .NET

- [Download .NET \(Linux, macOS, and Windows\)](#)

Download Visual Studio Code

- [Download Visual Studio Code](#)

Download C# Extension for Visual Studio Code

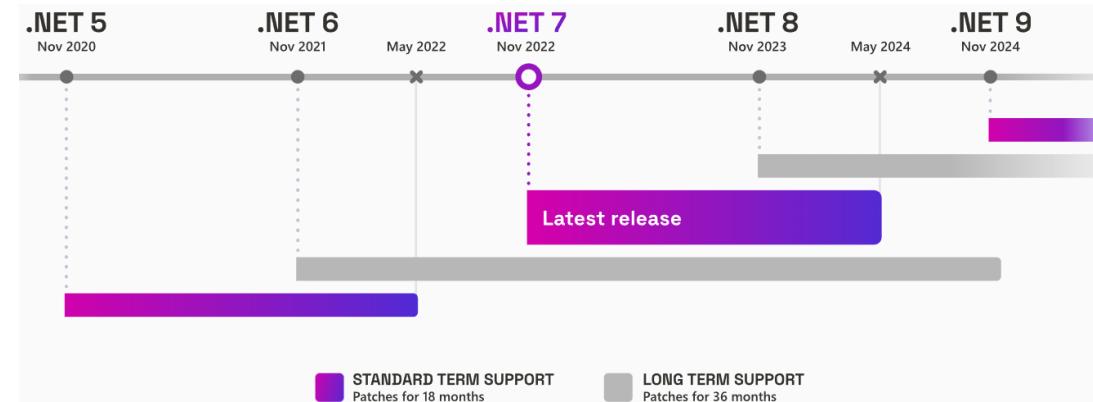
- [C# Extension for vscode](#)

Create Root Folder for CloudNative Course

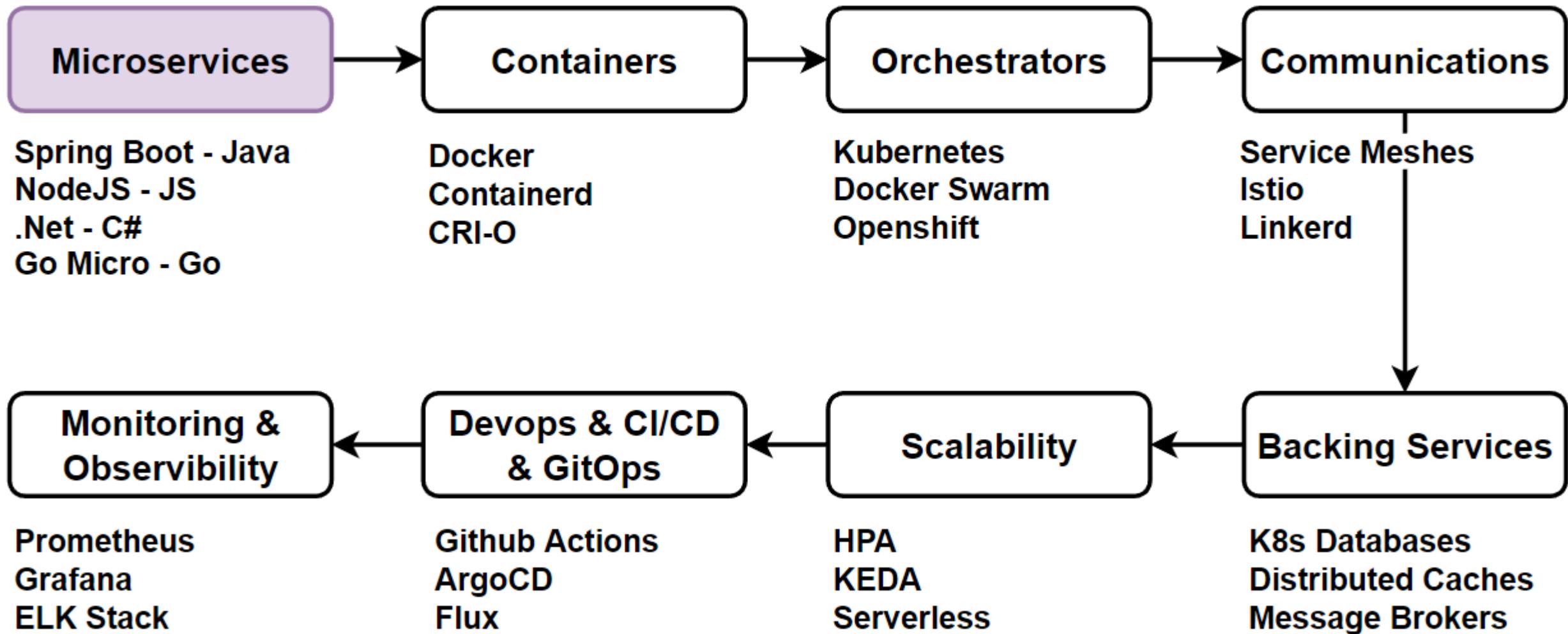
- Create Pillar folders: microservices, containers, orchestrators..

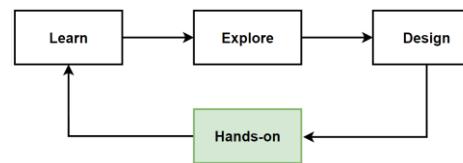
GitHub Source Codes for CloudNative Course

- <https://github.com/mehmetozkaya/CloudNative>

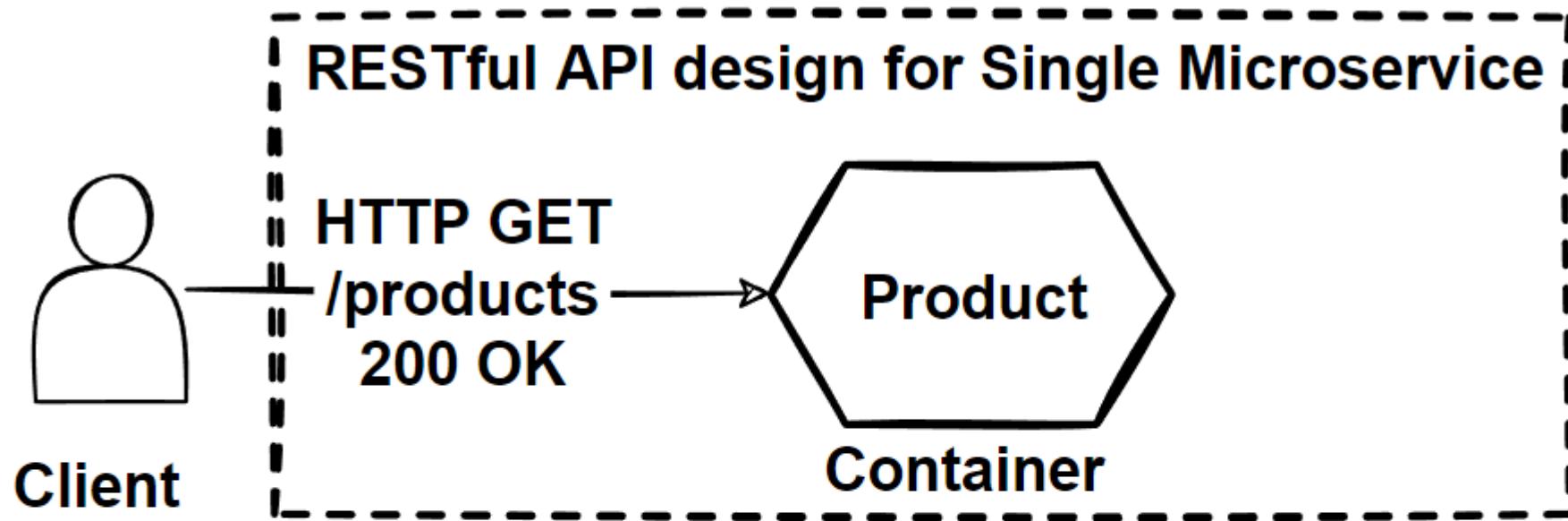


Cloud-Native Pillars Map – The Course Section Map





Hands-on: Develop a RESTful Microservices with CRUD



GET /api/products: Retrieves all products.

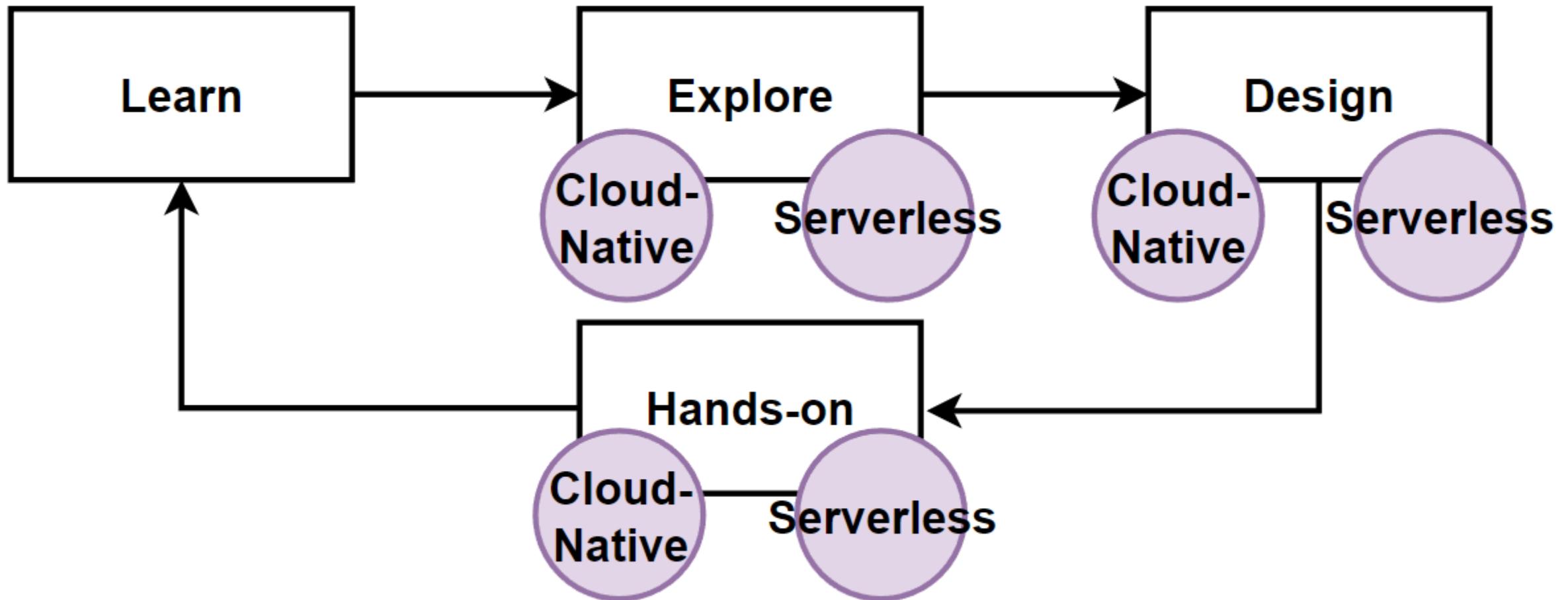
GET /api/products/{id}: Retrieves a specific product by ID.

POST /api/products: Adds a new product.

PUT /api/products/{id}: Updates an existing product by ID.

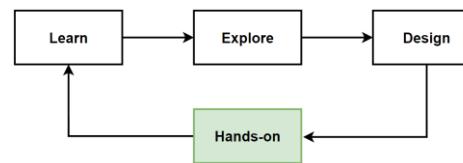
DELETE /api/products/{id}: Deletes a specific product by ID.

Way of Learning – Cloud-Native & Serverless Cloud Managed



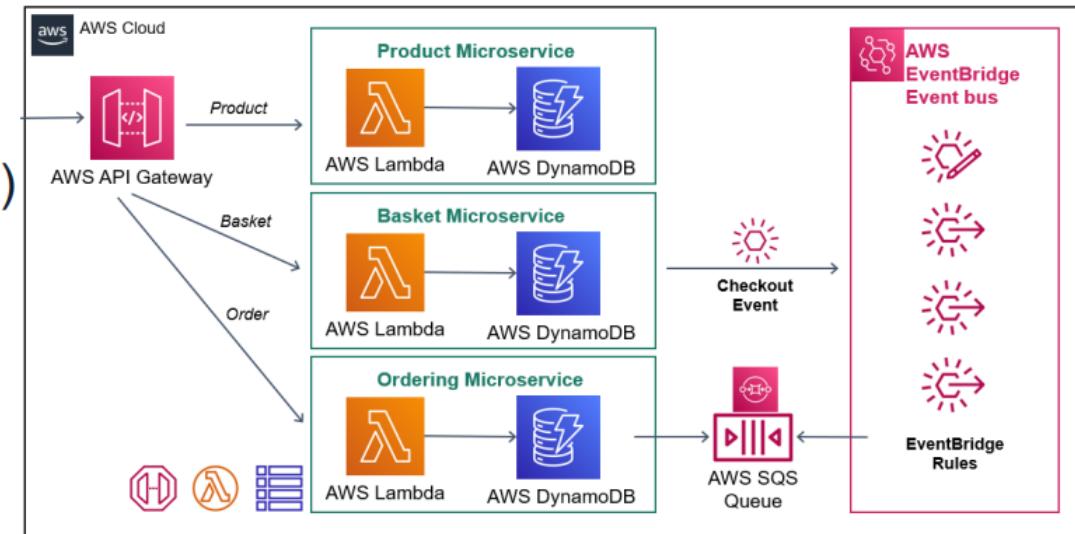
Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

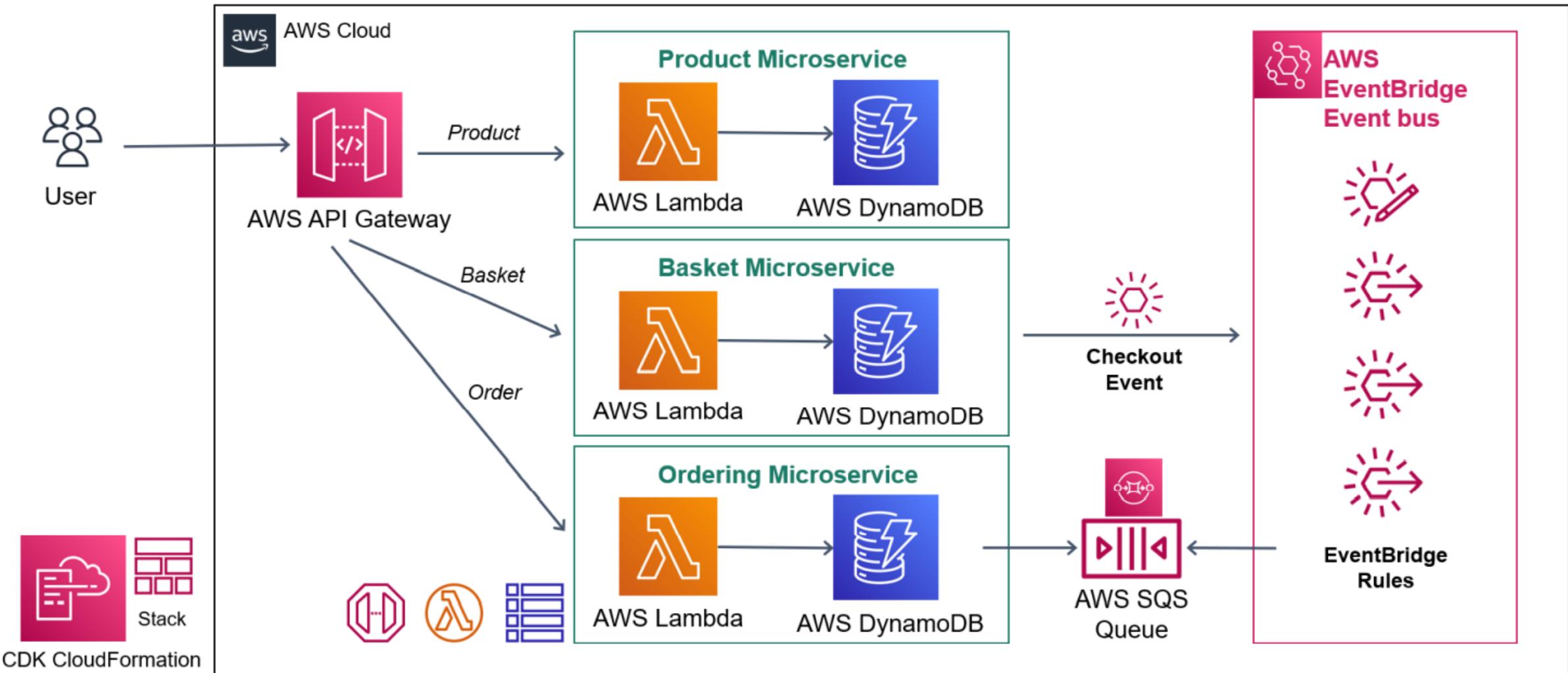


AWS Event-driven Serverless Microservices

- REST API and CRUD endpoints (AWS Lambda, API Gateway)
- Data persistence (AWS DynamoDB)
- Decouple Microservices with event (AWS EventBridge)
- Message Queues for cross-service communication (AWS SQS)
- Cloud stack development with IaC (AWS CloudFormation CDK)
- Event-driven Serverless Microservices
- Follow the **Serverless Design Patterns** and **Best Practices**
- Developing **E-commerce Event-driven Microservices** application

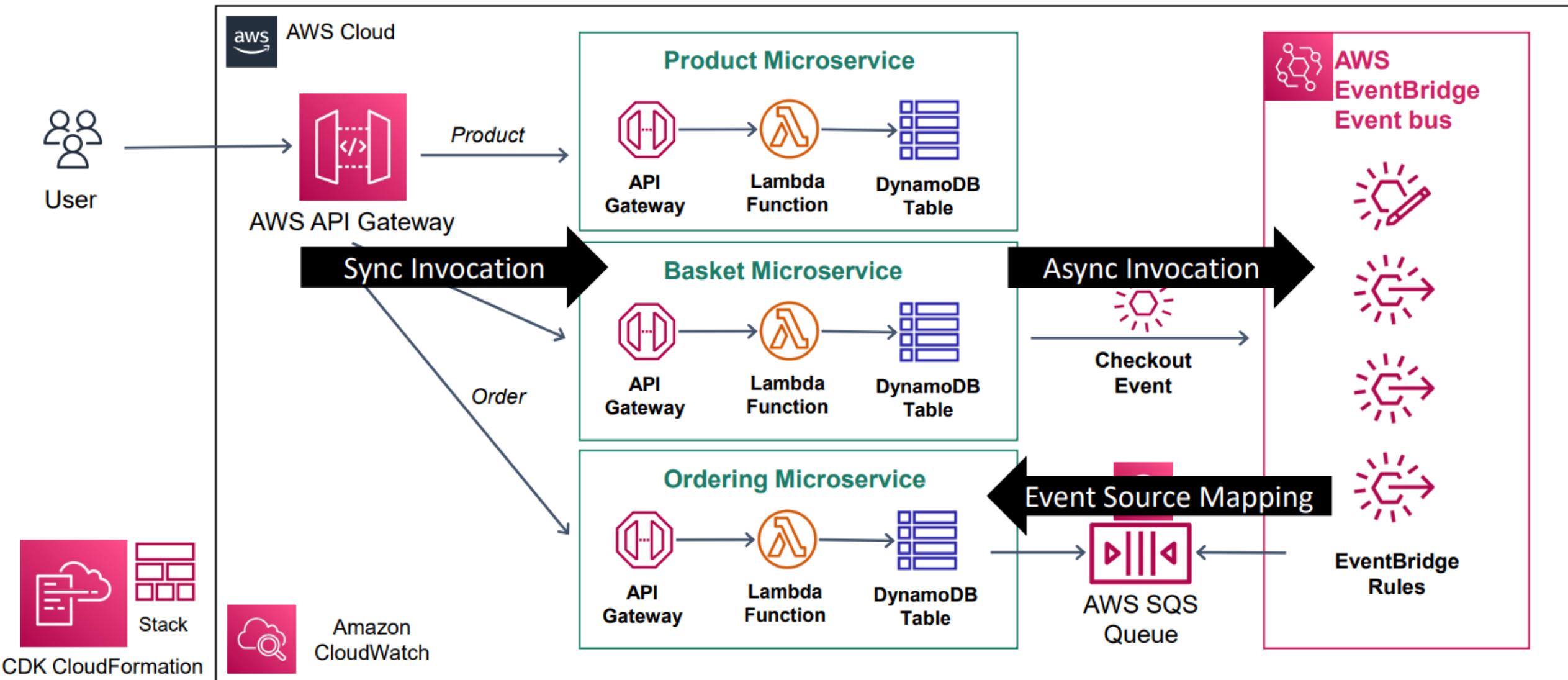


Reference Project: AWS Serverless E-Commerce Microservices

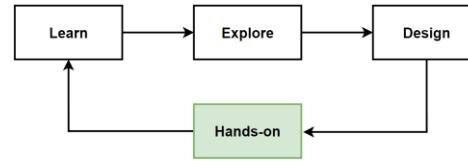


- <https://github.com/awsrun/aws-microservices>
- <https://github1s.com/awsrun/aws-microservices>

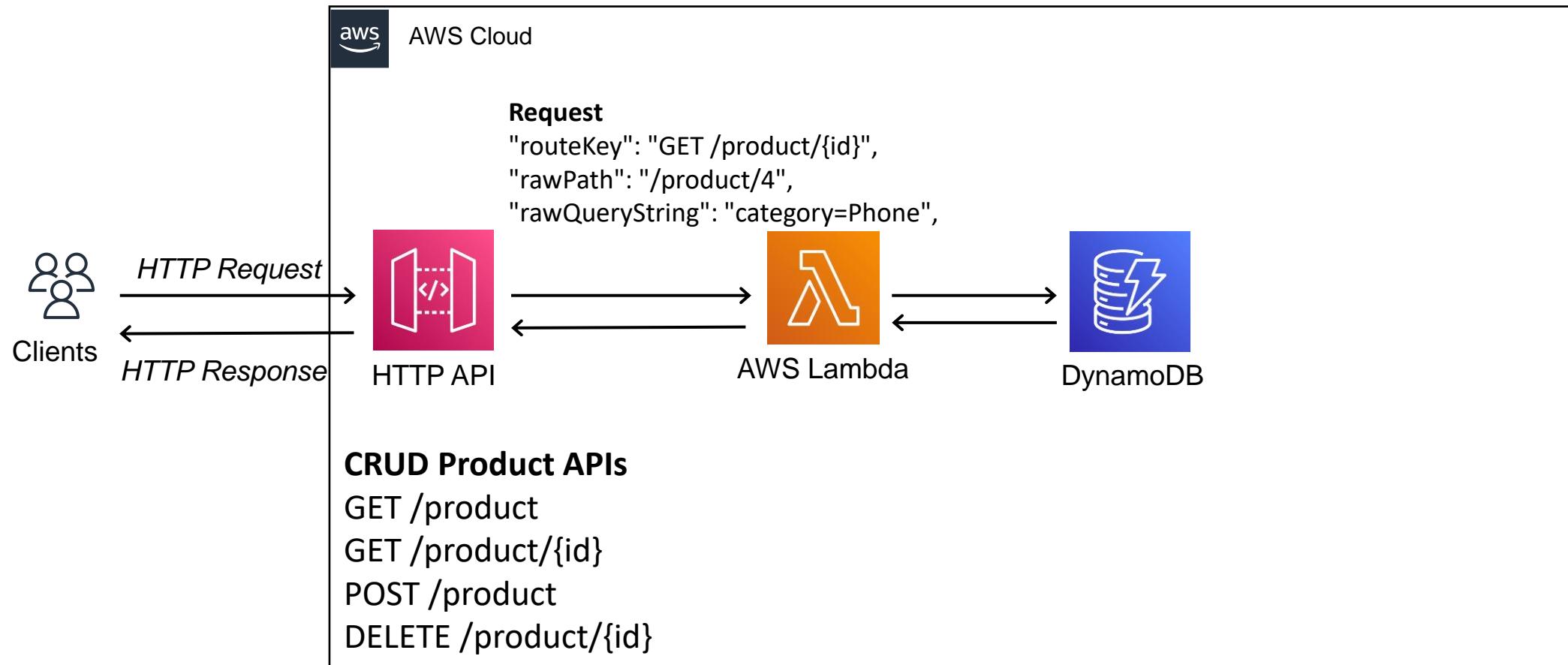
AWS Serverless E-Commerce Microservices Communications



- <https://github.com/awsrun/aws-microservices>
- <https://github1s.com/awsrun/aws-microservices>



Hands-on Lab: Build CRUD Microservice with HTTP API and AWS Lambda & DynamoDB



Cloud-Native Pillar2: Containers

What are Containers ?

Why we should use Containers for Cloud-Native Applications ?

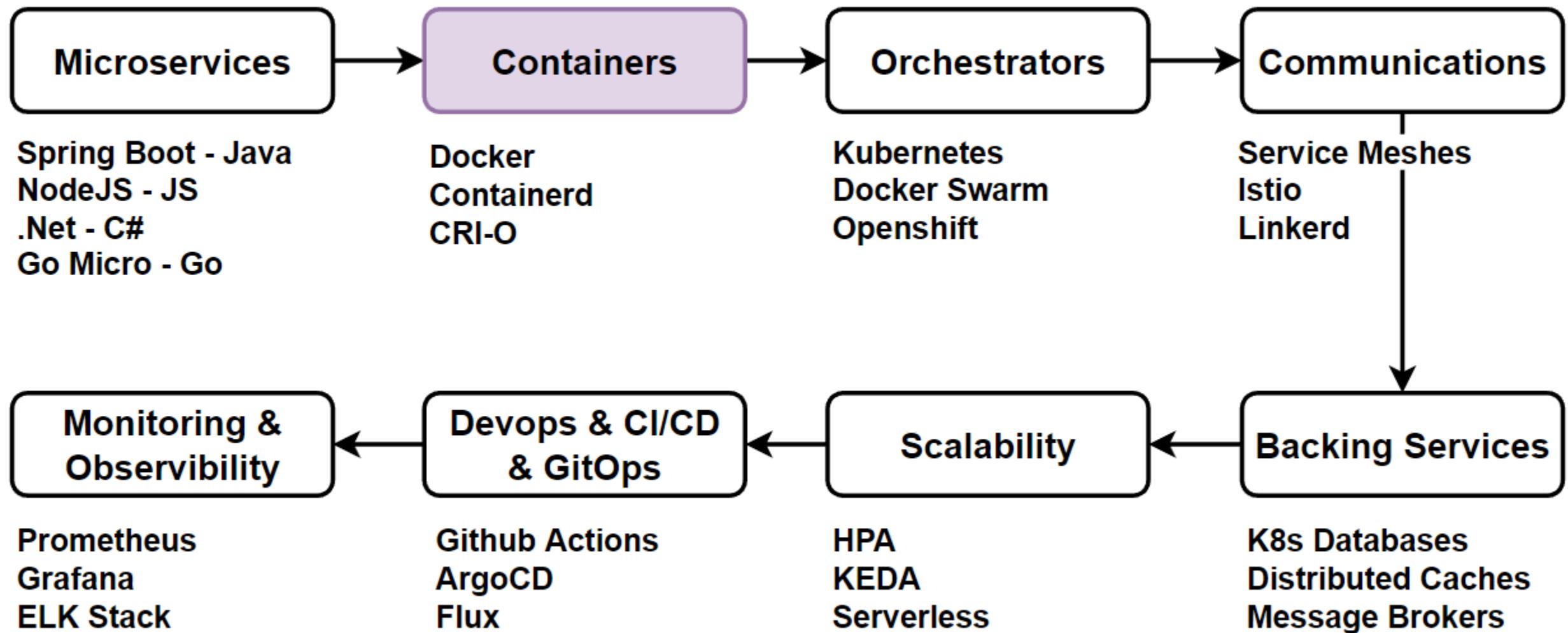
Benefits and Challenges of Containers

Design our E-Commerce application with Containers

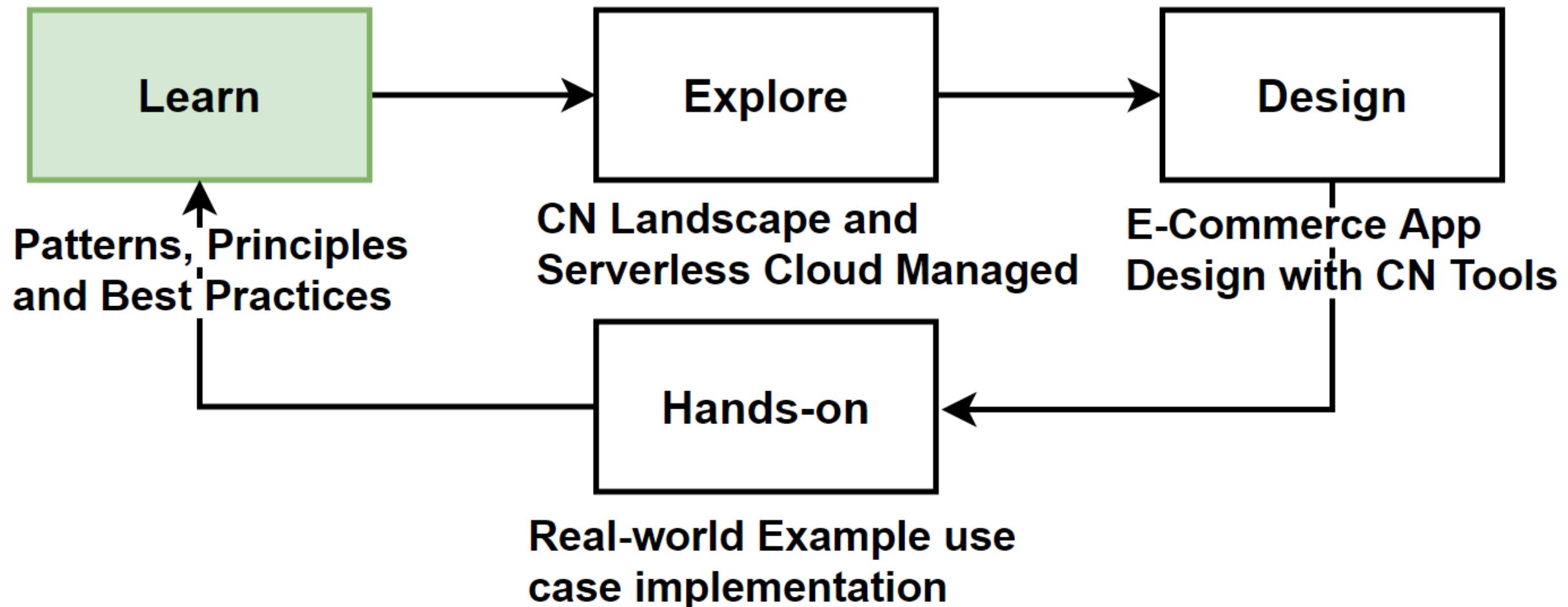
Implement Hands-on Development of Containers

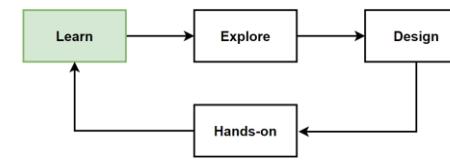
Mehmet Ozkaya

Cloud-Native Pillars Map – The Course Section Map



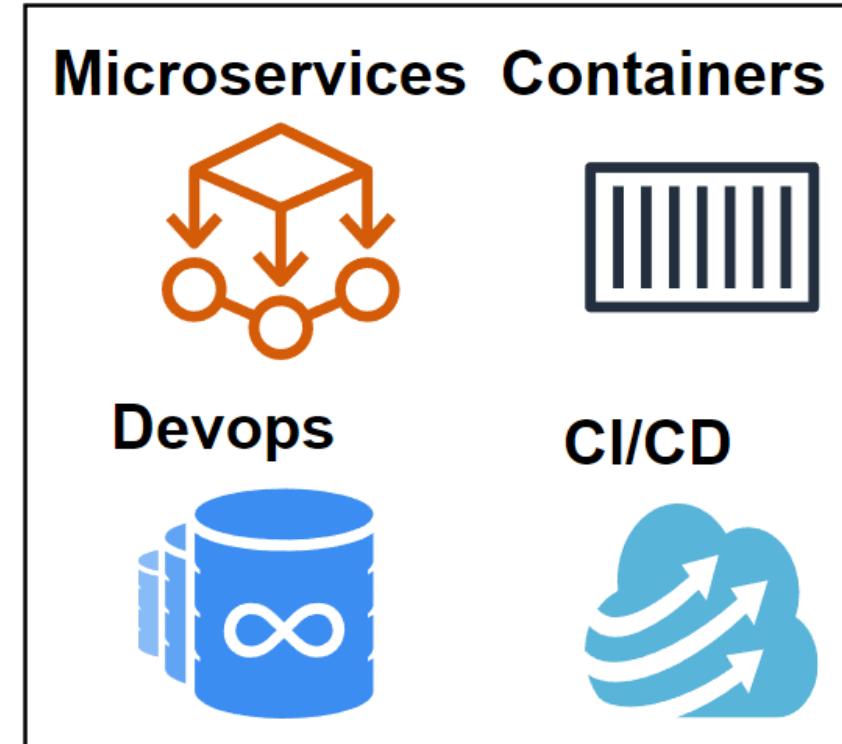
Way of Learning – The Course Flow

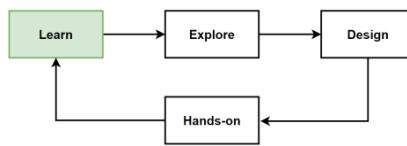




Learn: The Second Pillar - Containers

- Microservices Deployments with Containers and Orchestrators
- What are Containers ?
- Benefits and Challenges of Containers
- When to Use Containers - Best Practices
- Container Runtimes
- Container Registries
- What is Docker ?
- Docker Containers, Images, and Registries
- Why we use them for microservices deployments ?
- Design the Architecture – Containers
- Hands-on Labs with Containers





Where «Containers» in 12-Factor App and CN Trial Map

Twelve-Factor App

- Codebase
- Dependencies
- Config
- Backing Services
- Build, release and run
- Processes
- Port Binding
- Concurrency
- Disposability
- Development/Prod Parity
- Logs
- Admin Processes



CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape <https://github.com/cncf/landscape> has a growing number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator <https://www.cncf.io/training>

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider <http://cncf.io/ksp>

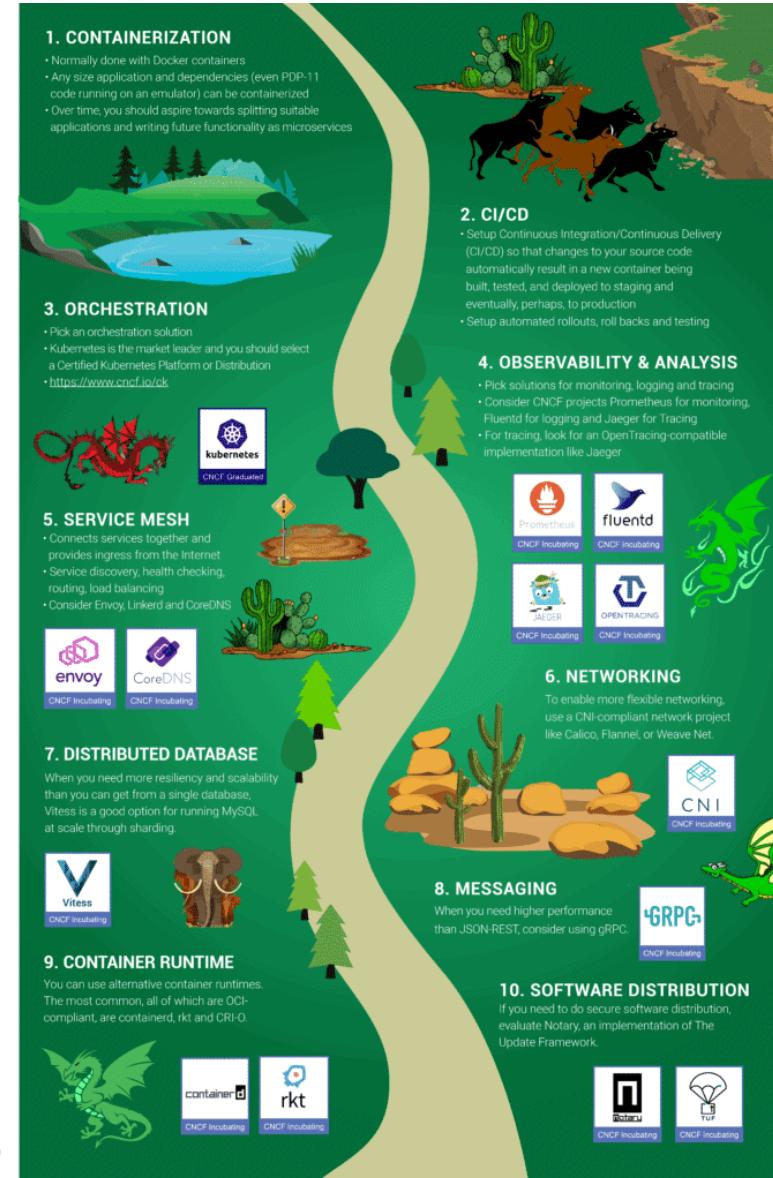
C. Join CNCF's End User Community

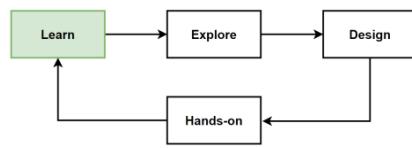
For companies that don't offer cloud native services externally <http://cncf.io/enduser>

WHAT IS CLOUD NATIVE?

- **Operability:** Expose control of application/system lifecycle.
- **Observability:** Provide meaningful signals for observing state, health, and performance.
- **Elasticity:** Grow and shrink to fit in available resources and to meet fluctuating demand.
- **Resilience:** Fast automatic recovery from failures.
- **Agility:** Fast deployment, iteration, and reconfiguration.

www.cncf.io
info@cncf.io





12-Factor App - Containers

Codebase

- Single codebase tracked in revision control, many deploys
- Containers package application & dependencies into an immutable unit
- Allows for consistent deployment across environments

Dependencies

- Declare and isolate dependencies explicitly
- Containers use Dockerfile or similar manifest
- Helps manage dependencies and ensure consistency

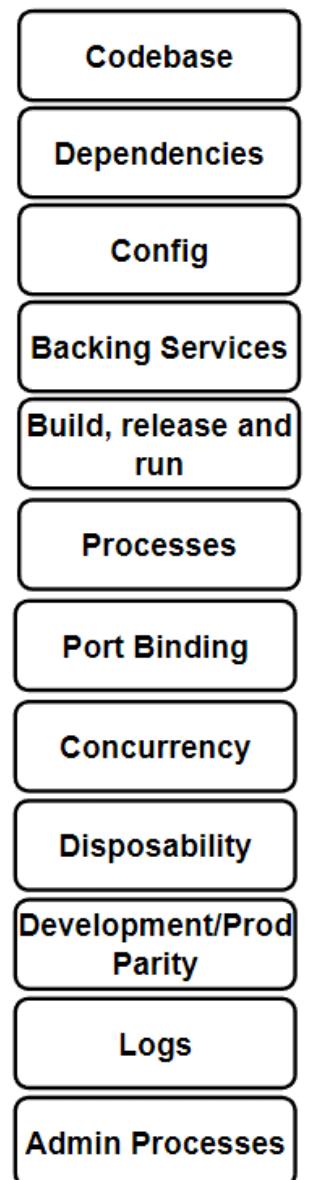
Config

- Store config in the environment
- Containers use environment variables for configuration
- Avoids code modification or container rebuilding

Build, Release, Run

- Separate build and run stages
- Containers build an image then run instances of the image

Twelve-Factor App



Cloud-native Trial Map - Containers

Containerization

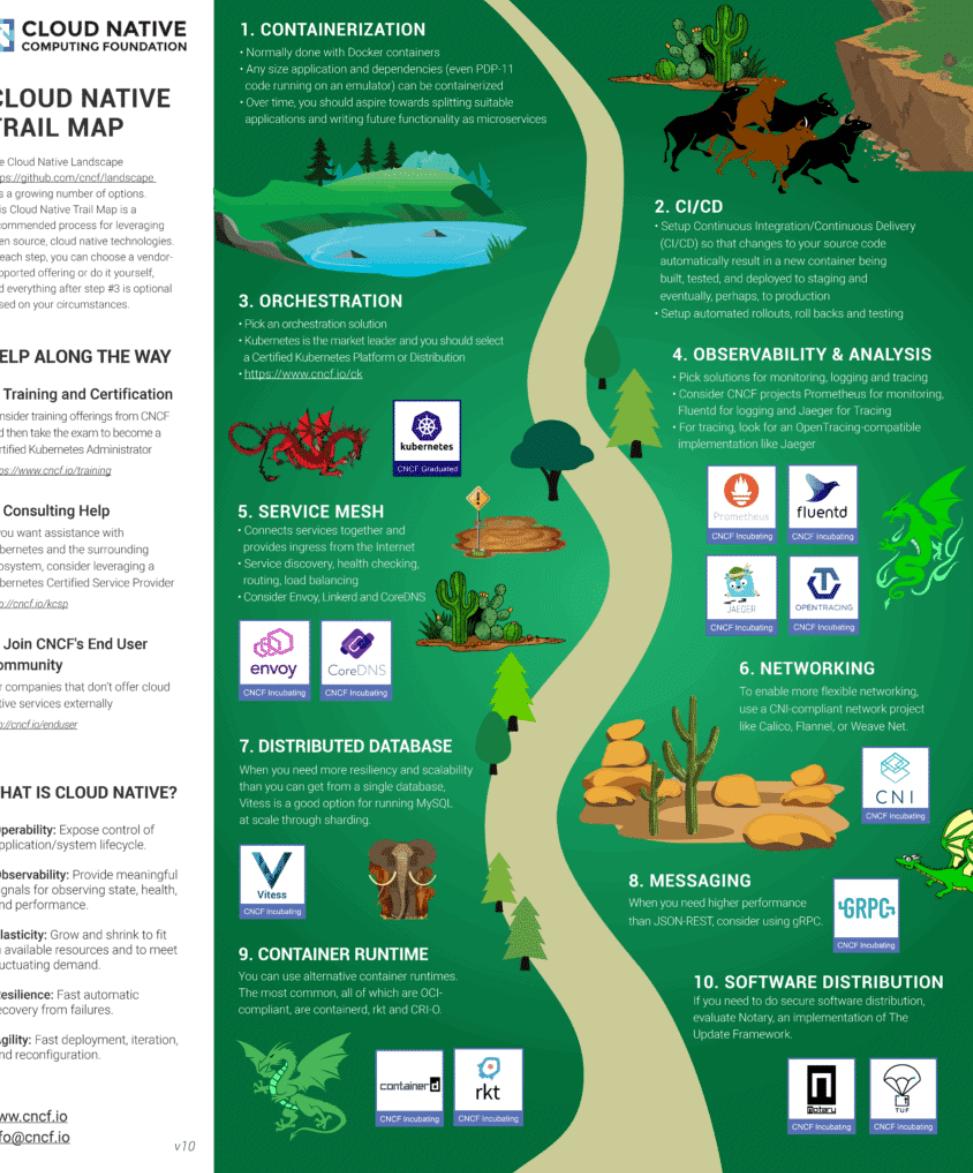
- Lightweight, portable units that package applications and their dependencies together
- Consistent and reproducible environment across different stages

Orchestration

- Tools like Kubernetes for managing and scaling containers
- Containers run efficiently and reliably in a distributed environment

Container Registry

- Centralized repository for storing and distributing container images
- Docker Hub, Google Container Registry, and Amazon Elastic Container Registry (ECR)
- Secure and efficient management and deployment of container images across environments



<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>

Evolution of Cloud Platforms: Cloud Hosting Models: IaaS - CaaS - PaaS - FaaS - SaaS - Serverless

IaaS (Infrastructure as a Service)

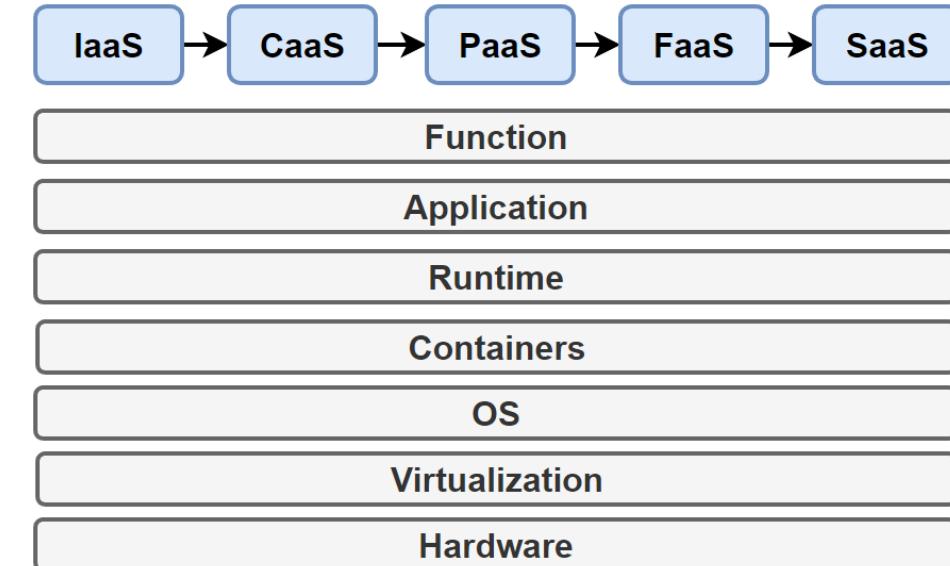
- Virtualized computing resources (VMs, storage, networking)
- Users control infrastructure, provider manages physical hardware
- Example: Amazon EC2, Microsoft Azure Virtual Machines
- Real-world: Hosting a web app on AWS EC2

CaaS (Container as a Service)

- Deploy, manage, and scale containerized applications
- Provider manages infrastructure and container orchestration
- Example: Google Kubernetes Engine, Amazon ECS, EKS, Fargate
- Real-world: Migrating a monolithic app to microservices using Google Kubernetes Engine

PaaS (Platform as a Service)

- Build, deploy, and manage applications without worrying about infrastructure
- Provider manages infrastructure, servers, networking, OS, etc.
- Example: Heroku, Microsoft Azure App Service
- Real-world: Developing a web app on Heroku



Hosting Models: IaaS - CaaS - PaaS - FaaS - SaaS - Serverless

FaaS (Function as a Service)

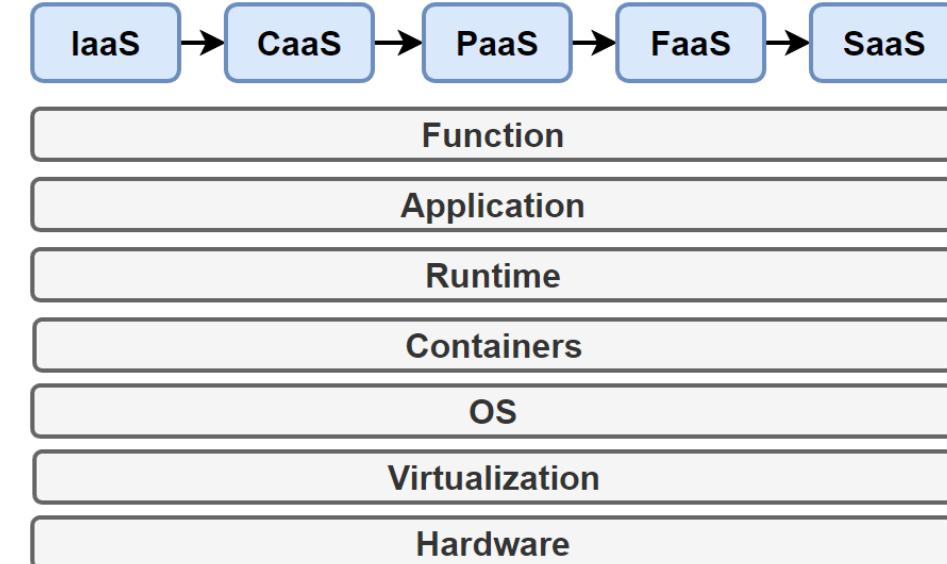
- Deploy individual functions, executed in response to events/triggers
- Provider manages infrastructure and auto-scales functions
- Example: AWS Lambda, Google Cloud Functions
- Real-world Example: Processing images using AWS Lambda

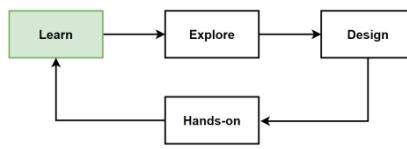
SaaS (Software as a Service)

- Software provided over the internet, eliminating need for installation
- Provider manages infrastructure, application, and data
- Example: Salesforce, Microsoft Office 365
- Real-world Example: Using Salesforce for CRM

Serverless

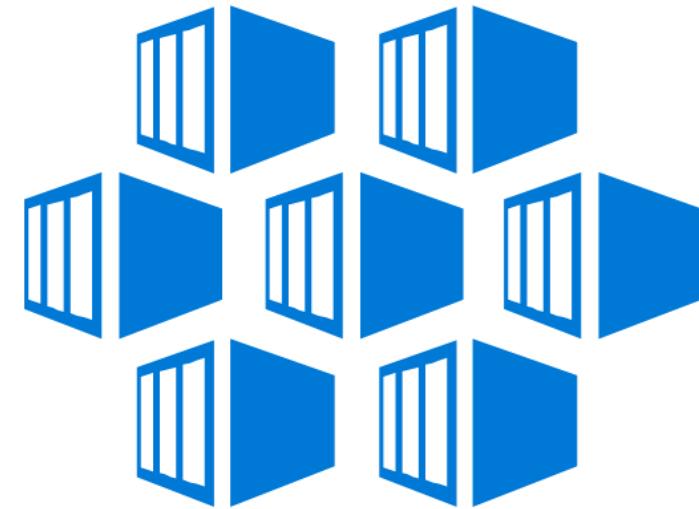
- Cloud provider manages infrastructure and auto-scales resources
- Pay only for resources consumed
- Example: Azure Functions with Azure Cosmos DB, AWS Lambda with Amazon DynamoDB
- FaaS is a specific implementation of serverless computing

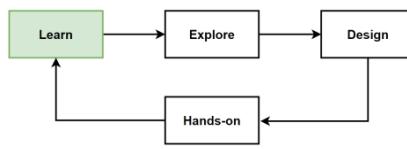




What are Containers ?

- Monolithic applications are deployed as a **single unit** and **deploy whole application** in one time.
- This caused **temporarily un-available** time of application, **needs to rollback** the **whole deployment** process. **Can't split modules** and scale independently.
- It solved in **microservices** architecture with **Containers** and **Orchestrators**.
- **Containers** are **package** and **distribute software applications**, makes them easy to **deploy** and **run** on **different environments**.
- Allow developers to **package an application** and its **dependencies** into a **single, self-contained** unit container images, **easily shipped** and **run** on any computer that has a container runtime.
- **Containers** are often used in the **deployment of microservices**, **independent components** can be developed, tested, and deployed separately.
- **Each microservice** is typically a **self-contained unit**, communicates with other microservices through well-defined interfaces.

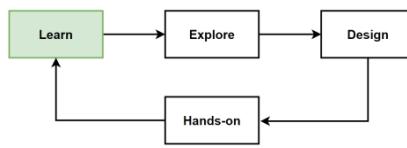




What are Containers ? - 2

- **Containers** provide to **decouple applications** with their own **os, dependencies and libraries**, perfect match to microservices deployments.
- Microservices can **deployed separately** in a **container**. Each microservice can **deploy independently** with containers.
- Since **deployed microservices separate container**, we can **scale** as per their **volume of traffics**.
- **Changes** can be **applied independently** while other container stay not changed.
- **New features** can be **applied** and **rollback** very easy with **container deployments**.
- **Docker** is defacto standard for **containerization of microservices**.





Advantages of Containers

- **Isolation**

Each microservice runs in its own container, provides isolation from the other microservices and the host operating system.

- **Scalability**

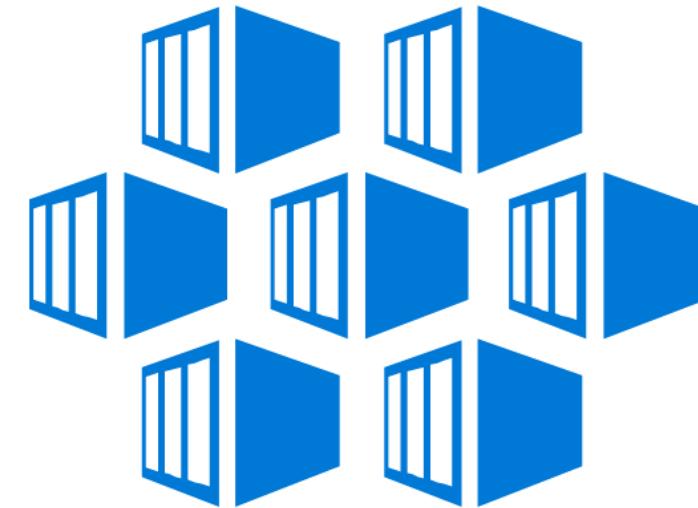
Containers make it easy to scale microservices horizontally by simply running more instances of a containerized microservice.

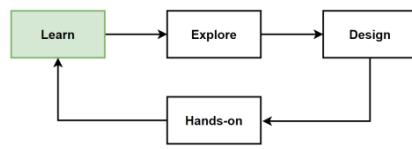
- **Portability**

Containers allow microservices to be easily deployed and run on any computer or cloud platform that supports container runtime.

- **Resiliency**

Microservices are intended to be independently deployable and scalable. Using containers in the deployment of microservices that can be quickly started and stopped, increase the resiliency of the application.





Why use Containers for developing Cloud-native Microservices ?

Consistency & Portability

- Containers package microservice with its dependencies
- Ensures service runs consistently across all environments
- Eliminates "it works on my machine" issues

Isolation

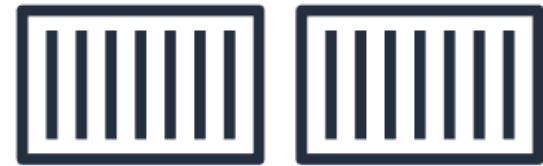
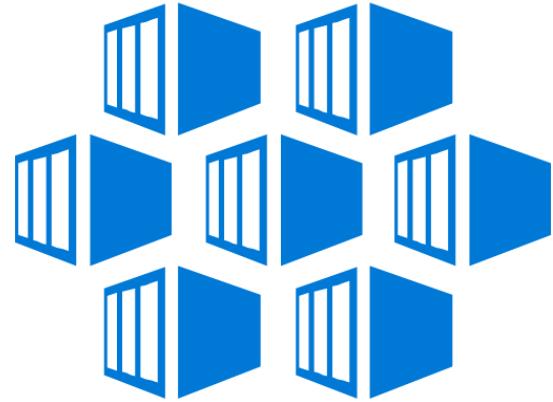
- Containers run in isolated environments
- Provides process and resource isolation between services
- Each microservice has its own dependencies, libraries, and runtime

Scalability

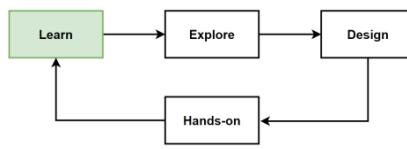
- Containers can be easily scaled horizontally
- Simplifies handling increased load or demand without affecting other architecture parts

Faster Development & Deployment

- Containers provide a consistent environment for development and deployment
- Reduces risk of delays due to dependency conflicts or environment differences



Containers



Why use Containers for developing Cloud-native Microservices ? - 2

Polyglot Environments

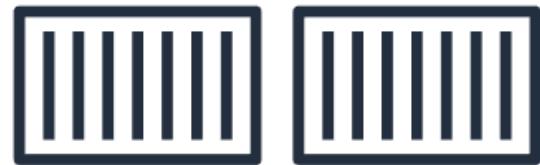
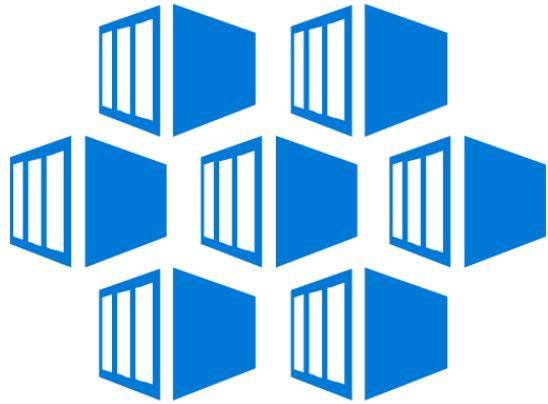
- Containers manage applications with different languages, frameworks, or dependencies
- Isolated containers allow running multiple services with varying dependencies without conflicts

Microservices Migration

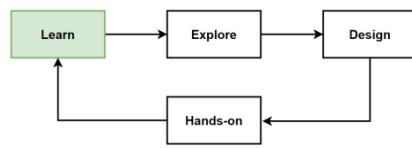
- Containers facilitate the transition from monolithic to microservices architectures
- Provides a consistent and isolated environment for each service

Hybrid or Multi-cloud Deployments

- Containers simplify deployment and management across multiple cloud providers
- Portability allows easy movement of services between platforms without significant changes



Containers



Best Practices Using Containers in Cloud-native Apps

One Process per Container

- Follow single-responsibility principle
- Run only one primary process per container

Use Small Base Images

- Choose small, minimal base images (Alpine Linux, distroless)
- Reduces attack surface, build times, improves startup performance

Keep Containers Stateless

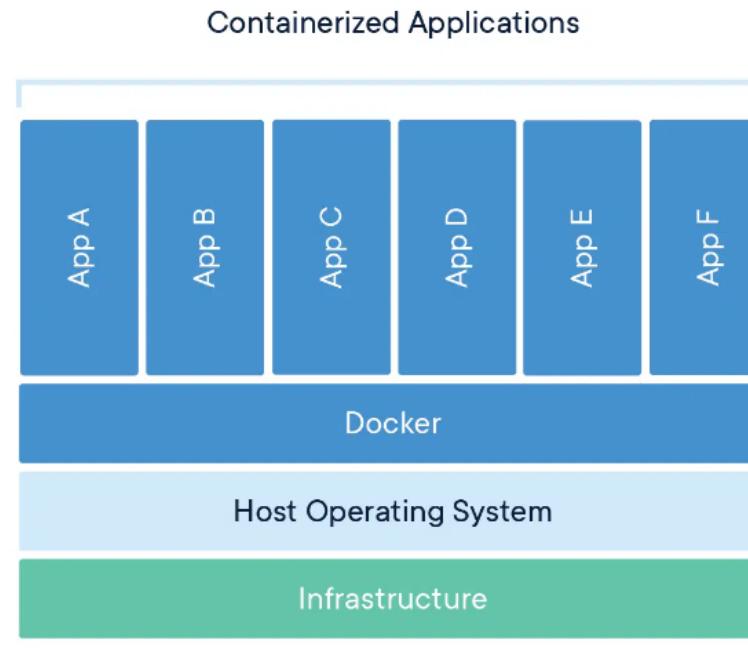
- Design containers to be stateless
- Easy replacement, scaling, updating without data loss
- Store stateful data in external services (databases, caches, object stores)

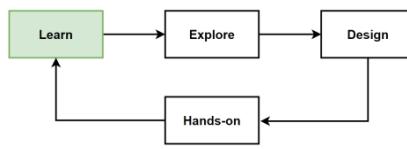
Use Environment Variables for Configuration

- Keep configuration settings external to the container
- Easy modification of settings without rebuilding container image

Label Container Images

- Use labels to provide metadata (version, build date, description)
- Helps with tracking, organization, auditing





Best Practices Using Containers in Cloud-native Apps - 2

Version Container Images

- Use semantic versioning for container images
- Store them in a container registry

Implement Health Checks

- Add health checks to containers
- Allows orchestration system to restart or replace unhealthy instances

Use Container Orchestration

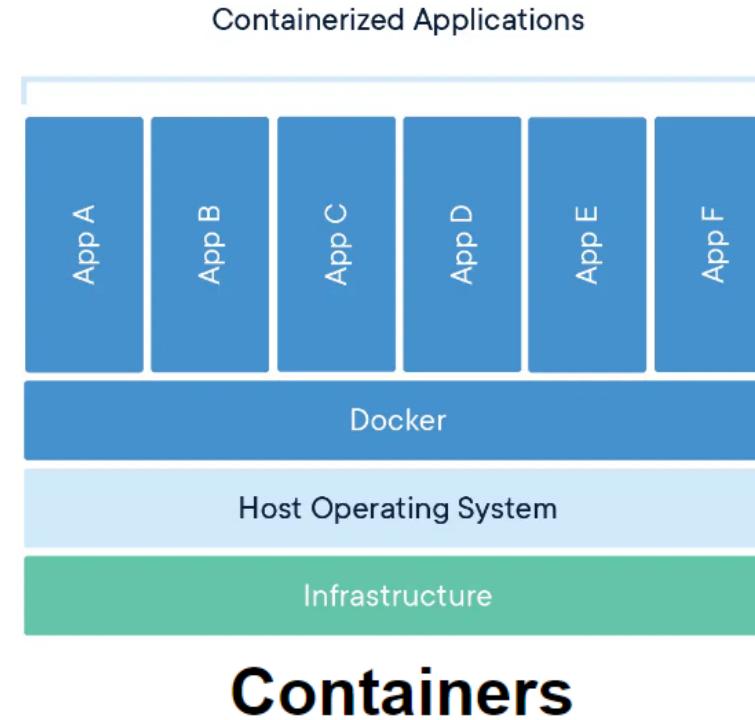
- Use orchestration platform (Kubernetes, Amazon ECS) for automation
- Simplifies management of complex applications and improves resilience

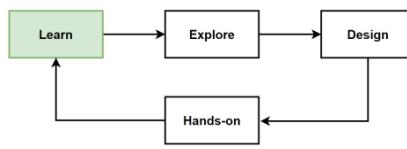
Monitor and Log

- Implement monitoring, logging, tracing for containerized services
- Use tools (Prometheus, Grafana, ELK stack) for metrics and logs

Continuous Integration and Deployment

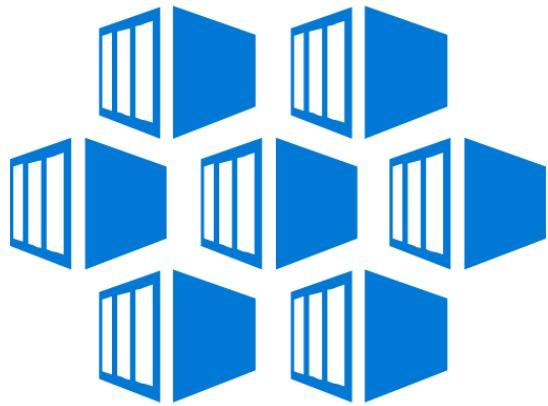
- Automate building, testing, deploying with CI/CD pipelines
- Keeps application up-to-date, minimizes risk of human errors



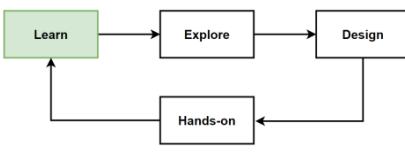


How Do Containers Work?

- Containers are a form of lightweight virtualization. Allow multiple isolated apps to run on the same host. They share the host's kernel and resources while running in isolation.
- **Container Image**
A self-contained package that includes the application, runtime, libraries, and settings. Constructed from a base image (usually a minimal OS) and a set of instructions.
- **Container Runtime**
Software that manages containers, providing an isolated environment and managing resources, lifecycle, and interactions with the host system.
- **Image Layers**
These are created for each instruction in a Dockerfile. Layers can be cached and reused, reducing build time and storage needs.
- **Namespaces**
Containers utilize Linux namespaces for process isolation. Creates an environment where a process can't see or interact with resources outside of its namespace.
- **Networking**
Containers communicate with each other and the host system through virtual network interfaces, which are created by container runtimes.

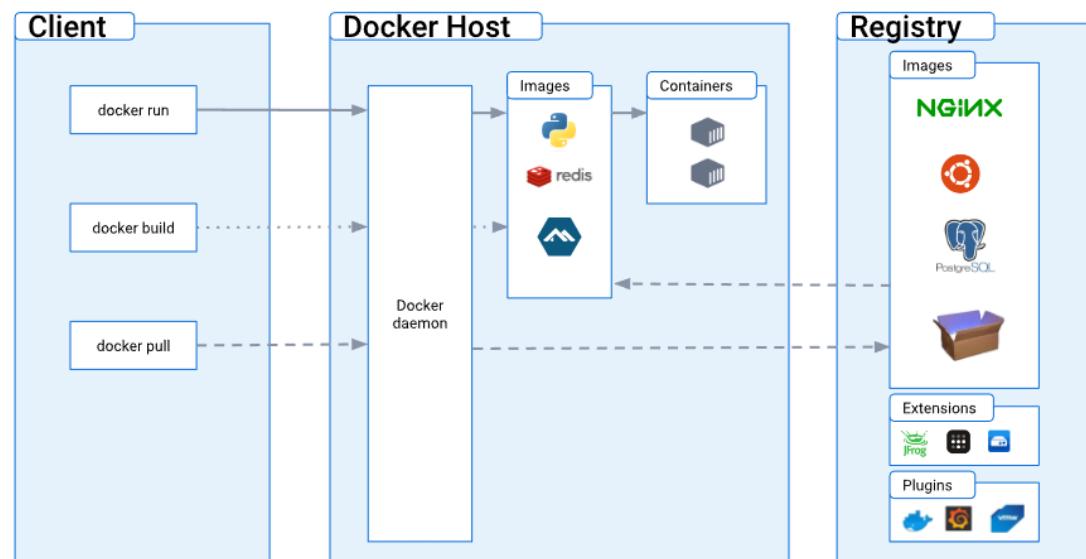


Containers

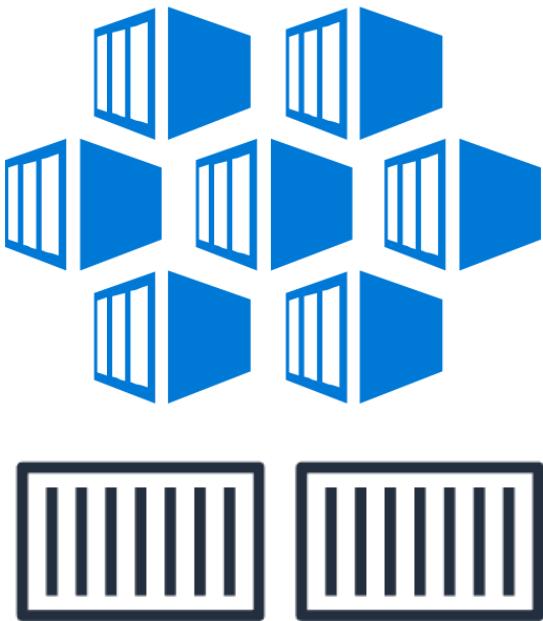


Running a Container

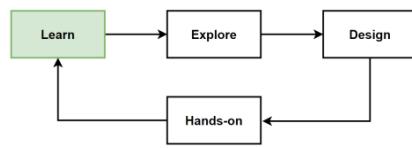
- When a container runs, the **container runtime** creates an **isolated environment** using **namespaces**, **cgroups**, and a union file system.
- The application starts within this environment, **utilizing the resources** and **dependencies** provided by the **container image**.
- Despite **sharing the host's kernel** and **resources**, the container remains separate from other containers and the host itself.



<https://docs.docker.com/get-started/overview/>

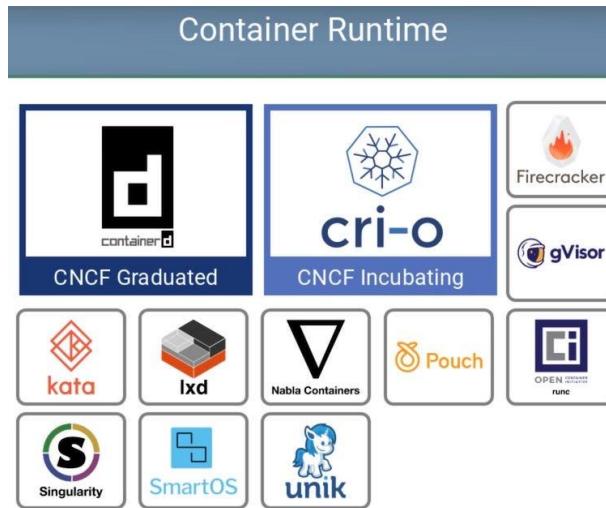


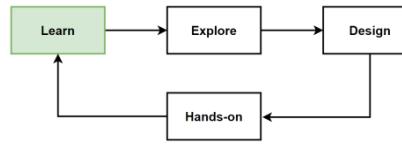
Containers



What is Container Runtime ?

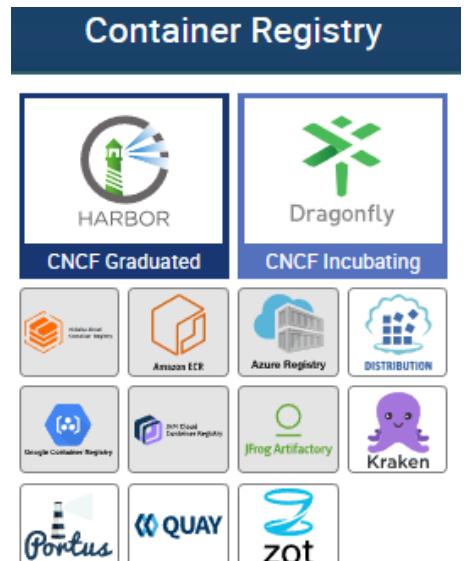
- Container runtime executes and manages containers that provides the necessary environment to run containerized applications handles tasks; creation, starting, stopping, and monitoring of containers.
- **Docker**
The most known container runtime synonymous with container technology. Uses its own runtime, containerd, to manage containers.
- **containerd**
Docker's container runtime, also operable independently. It's lightweight and focuses on core container execution functionalities.
- **CRI-O**
Lightweight container runtime designed specifically for Kubernetes. Implements the Kubernetes Container Runtime Interface (CRI), allowing Kubernetes to use any OCI-compliant runtime.
- **runc**
low-level container runtime based on the Open Container Initiative (OCI) runtime specification. It's the default runtime used by containerd and can also be used standalone for running containers.

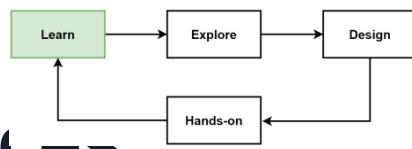




What is a Container Registry?

- **Centralized repositories** for storing and managing container images.
- Facilitates **pushing, pulling, and sharing container images**.
- Ensures **secure storage** and **easy access** to images for deployment.
- Some **popular container registries** are: Docker Hub, Google Container Registry (GCR), Amazon Elastic Container Registry (ECR), Azure Container Registry (ACR)
- You can find a vast array of images **created by other users** and organizations, many of which **are open-source** and **free to use**.
- Option to **create private registries** using cloud services or by hosting your own registry server.





Container Registries in Deploying Cloud-native Microservices

Image Building

- Dockerfiles define the construction of Docker images for microservices.
- Includes base image, application code, and dependencies.

Image Pushing

- Post-construction, images are pushed to a container registry.
- Typically part of a Continuous Integration/Continuous Deployment (CI/CD) pipeline.

Image Pulling

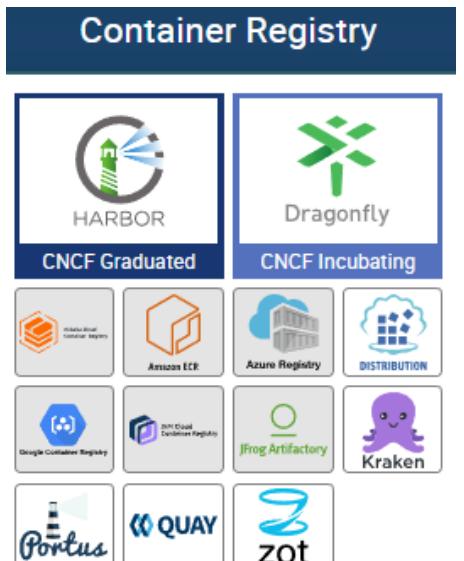
- During deployment or update, the container orchestration system (like Kubernetes) retrieves the image from the registry.

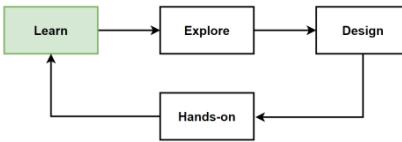
Versioning and Rollback

- Images can be tagged with version numbers for easy rollback.

Access Control

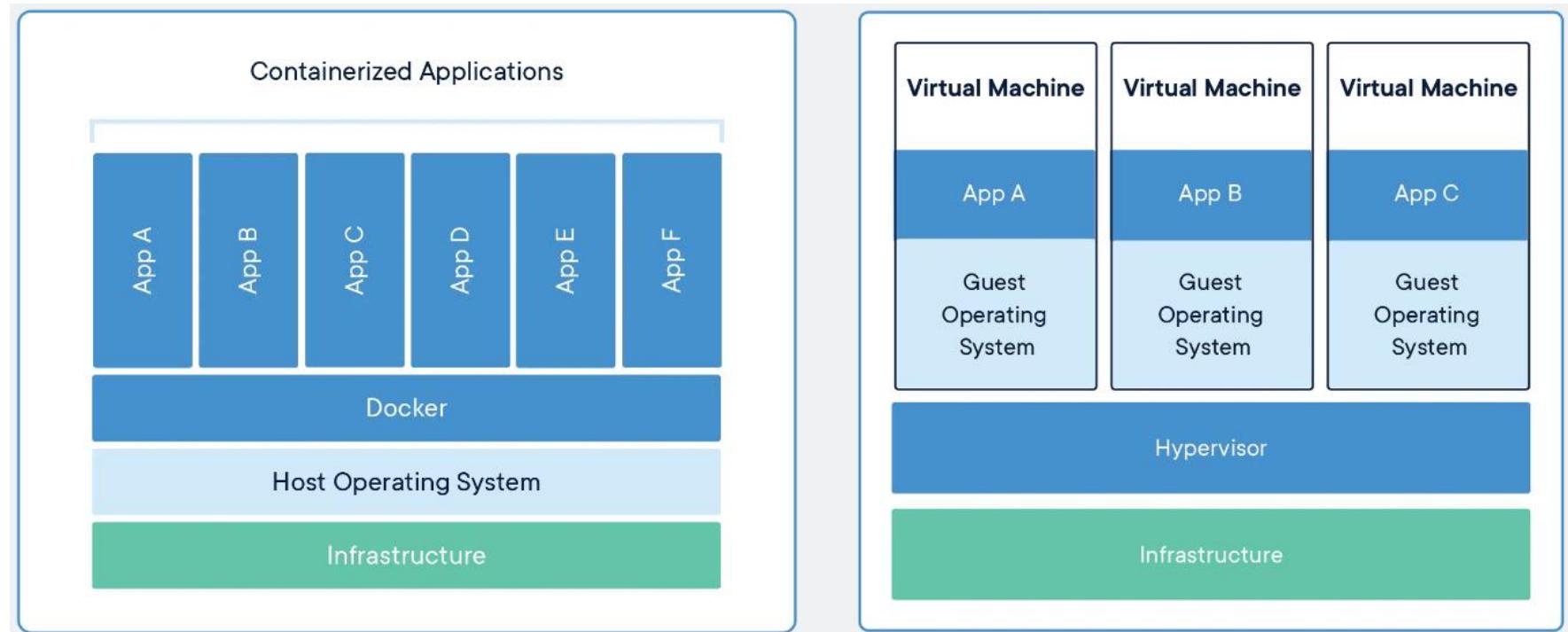
- Private registries often provide access control features for secure and authorized access to images.

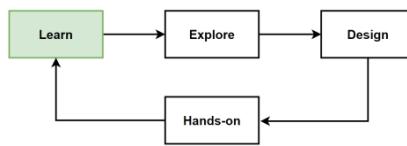




Comparison of VM and Container Runtimes

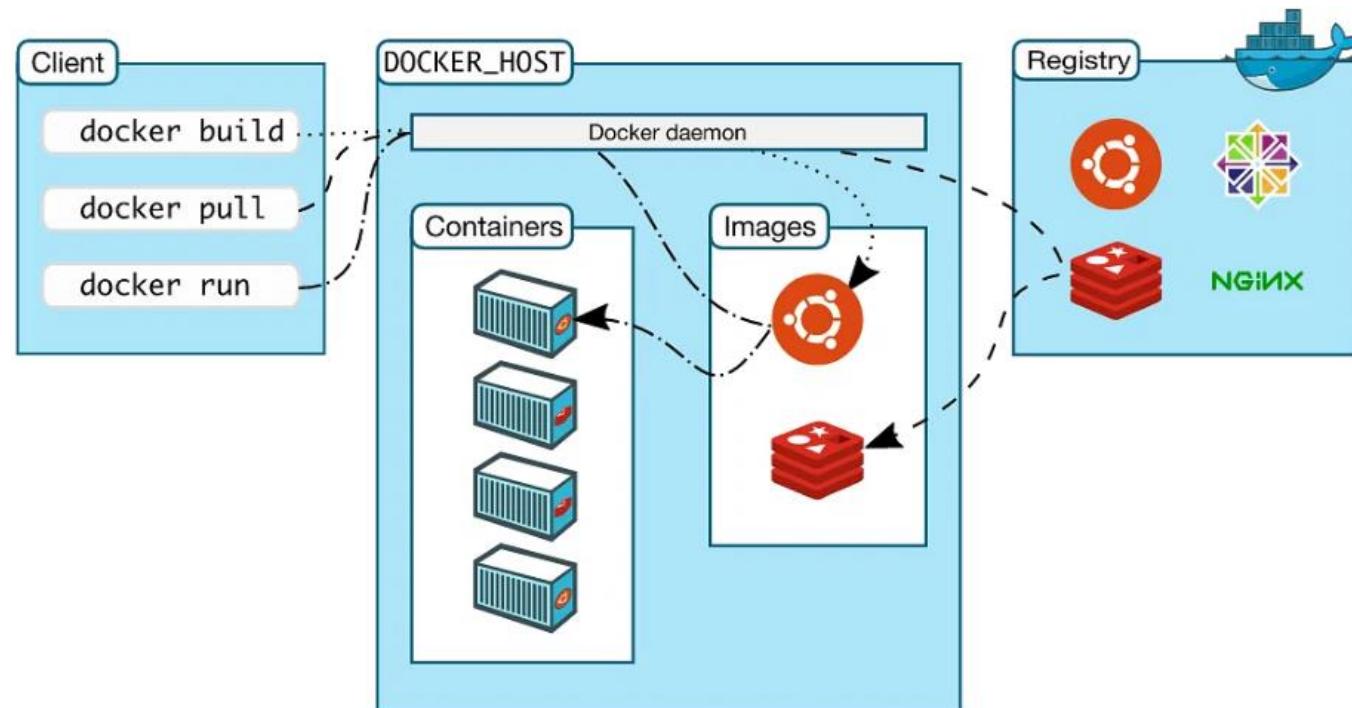
- **Microservices** can run on **virtual machines (VMs)** or **containers**.
- **Containers share** the host machine's **kernel, processor, and memory** with other containers, making them more **lightweight**.
- VMs run a **full guest operating system** with **virtual access** to host resources, resulting in **higher resource consumption**.





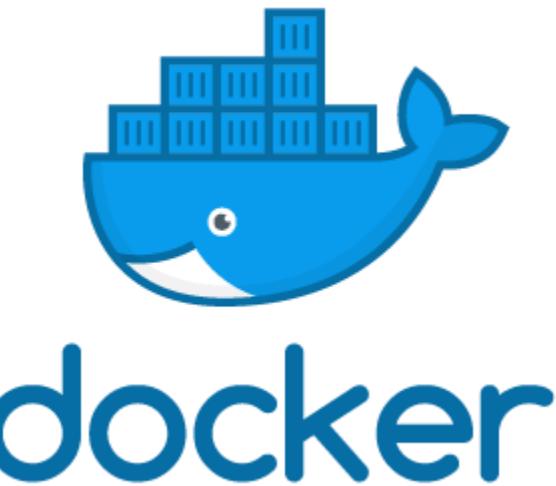
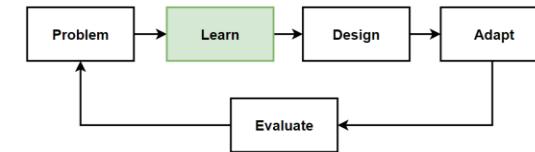
Containerization: Running Microservices in Containers

- Containerization is the process of **converting microservices to run on containers**.
- Docker is the most **popular platform for containerization**.
- The **code, its dependencies, and runtime** are packaged into a **container image**.
- **Images** are **stored in a container registry**, which acts like a repository.
- When needed, the image is transformed into a **running container instance** on any computer with a container runtime engine installed.



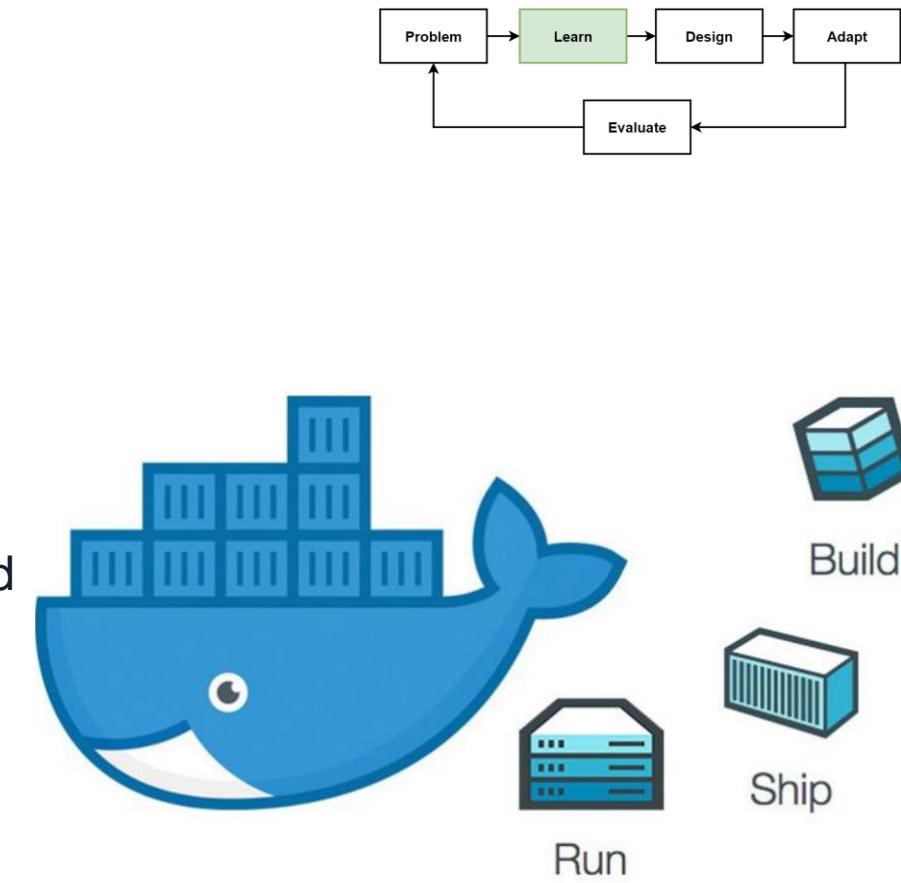
What is Docker ?

- Docker is an **open platform** for **developing, shipping, and running** applications.
- **Separate your applications from your infrastructure** so you can deliver software quickly.
- Advantages of Docker's methodologies for **shipping, testing, and deploying** code **quickly**.
- **Significantly reduce** the delay between **writing code** and **running** it in production.
- Automating the deployment of applications as **portable, self-sufficient containers** that can run on the **cloud or on-premises**.
- **Docker containers** can run anywhere, in your local computer to the cloud.
- **Docker image** containers can run **natively** on **Linux and Windows**.
- **Docker** is defacto standard for **containerization** of **microservices**.

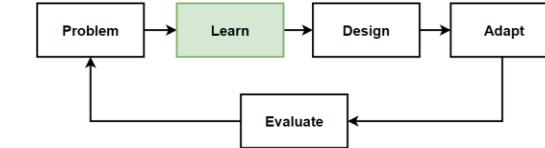


Docker Containers, Images, and Registries

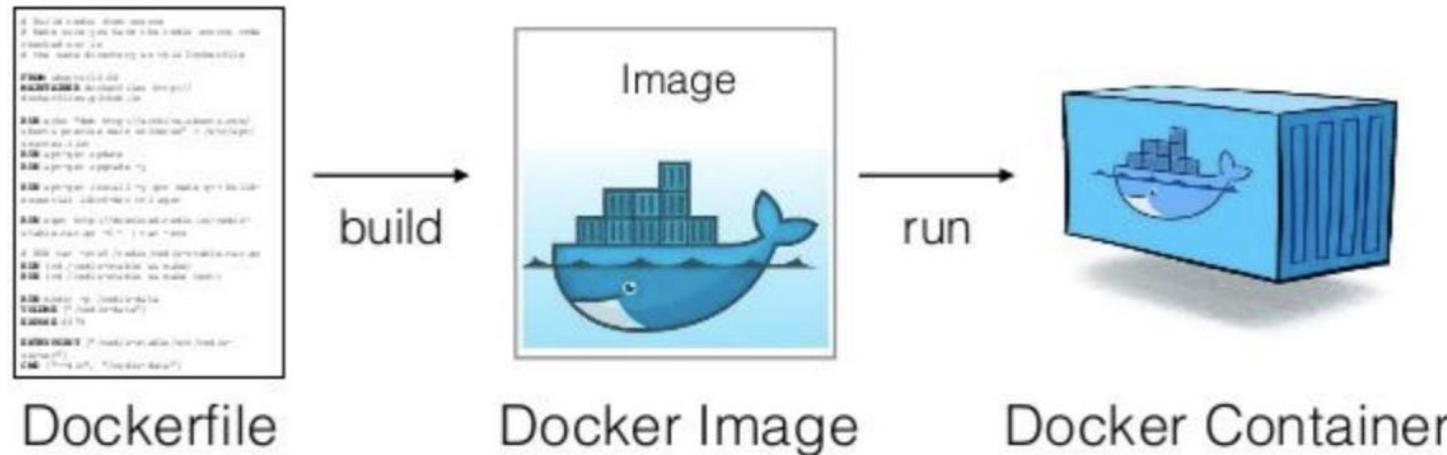
- Developer **develops** and **packages** application **with its dependencies** into a **container image**, that is a **static representation** of the application **with its configuration** and **dependencies**.
- To run the application, the application's image is **instantiated to create** a **container**, which will be running on the Docker host.
- Store **images in a registry**, which is a **library of images** and is needed when deploying to production orchestrators.
- **Docker images are stores** a public **registry** like **Docker Hub**, **Azure Container Registry**.
- Developer **creates container** in local and **push the images** the **Docker Registry**.
- Developer **download existing image** from **registry** and create container from image in local environment.



Application Containerization with Docker



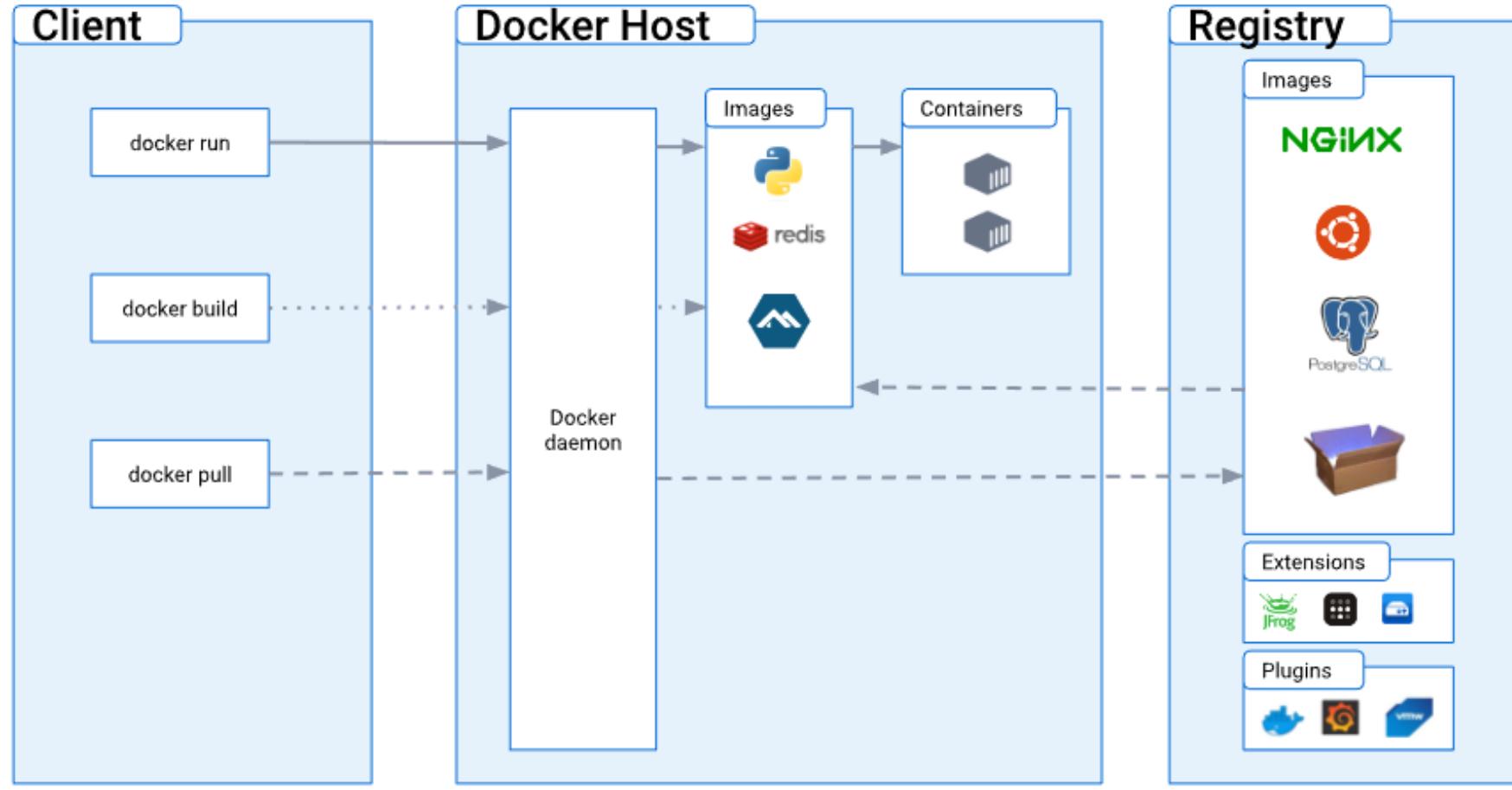
1. Write Dockerfile for our application.
 2. Build application with this docker file and creates the docker images.
 3. Run this images on any machine and creates running docker container from docker image.



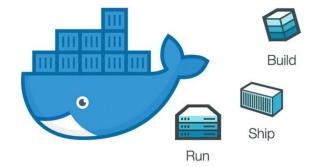
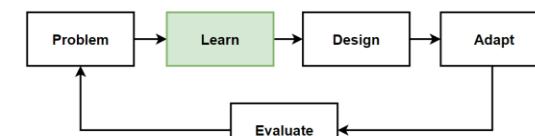
- **Orchestrating** whole **microservices** application with **Docker** and **Kubernetes**.

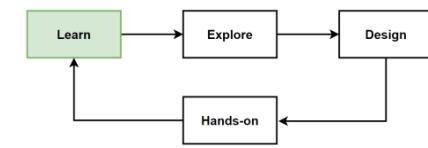
Docker Architecture

- Docker Daemon, Docker Client, Docker API, Docker Images, Docker Registries, Docker Containers, Docker Networks, Docker Volumes.



<https://docs.docker.com/get-started/overview/>





Docker Architecture

Docker Daemon

- The Docker daemon (dockerd) is the server component running on the host machine.
- It listens for API requests from the Docker client.
- The daemon manages Docker objects: images, containers, networks, and volumes.

Docker Client

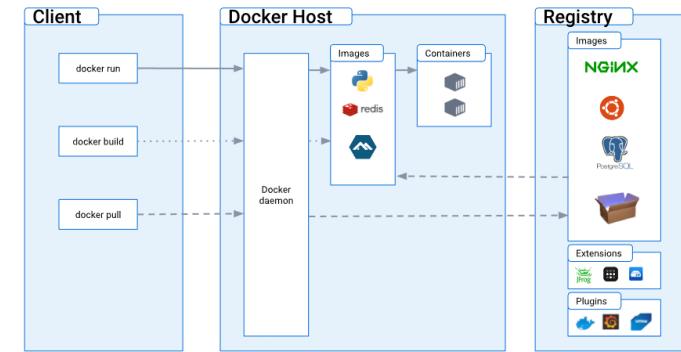
- The Docker client (docker) is the command-line interface (CLI).
- It communicates with the Docker daemon via the Docker API.
- Users issue commands such as 'docker build', 'docker run', and 'docker ps'

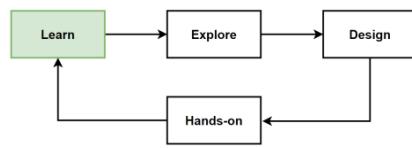
Docker API

- The Docker API allows the Docker client to communicate with the Docker daemon.
- It facilitates the creation, management, and monitoring of Docker objects.
- It can be interacted with directly or through third-party tools.

Docker Images

- Docker images are the building blocks of containers.
- They include everything needed to run an application: code, runtime, system libraries.
- Images are stored in container registries.





Docker Architecture - 2

Docker Registries

- Docker registries store and distribute Docker images.
- Docker Hub is the default public registry, but private registries or your own can be used.
- The Docker client communicates with registries to download or upload images.

Docker Containers

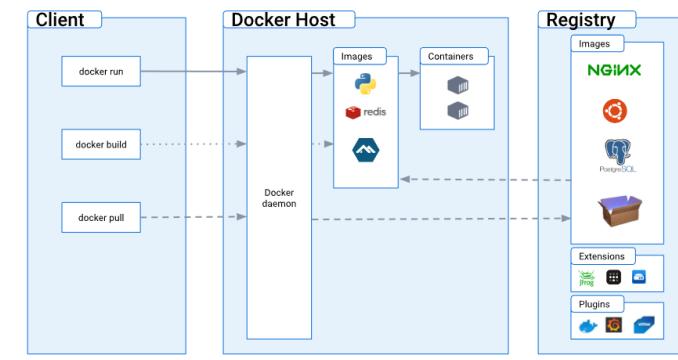
- A Docker container is a running instance of a Docker image.
- Containers are isolated from the host system and other containers.

Docker Networks

- Docker networks provide the communication infrastructure for containers.
- They support various network types, each with different isolation and performance characteristics.

Docker Volumes

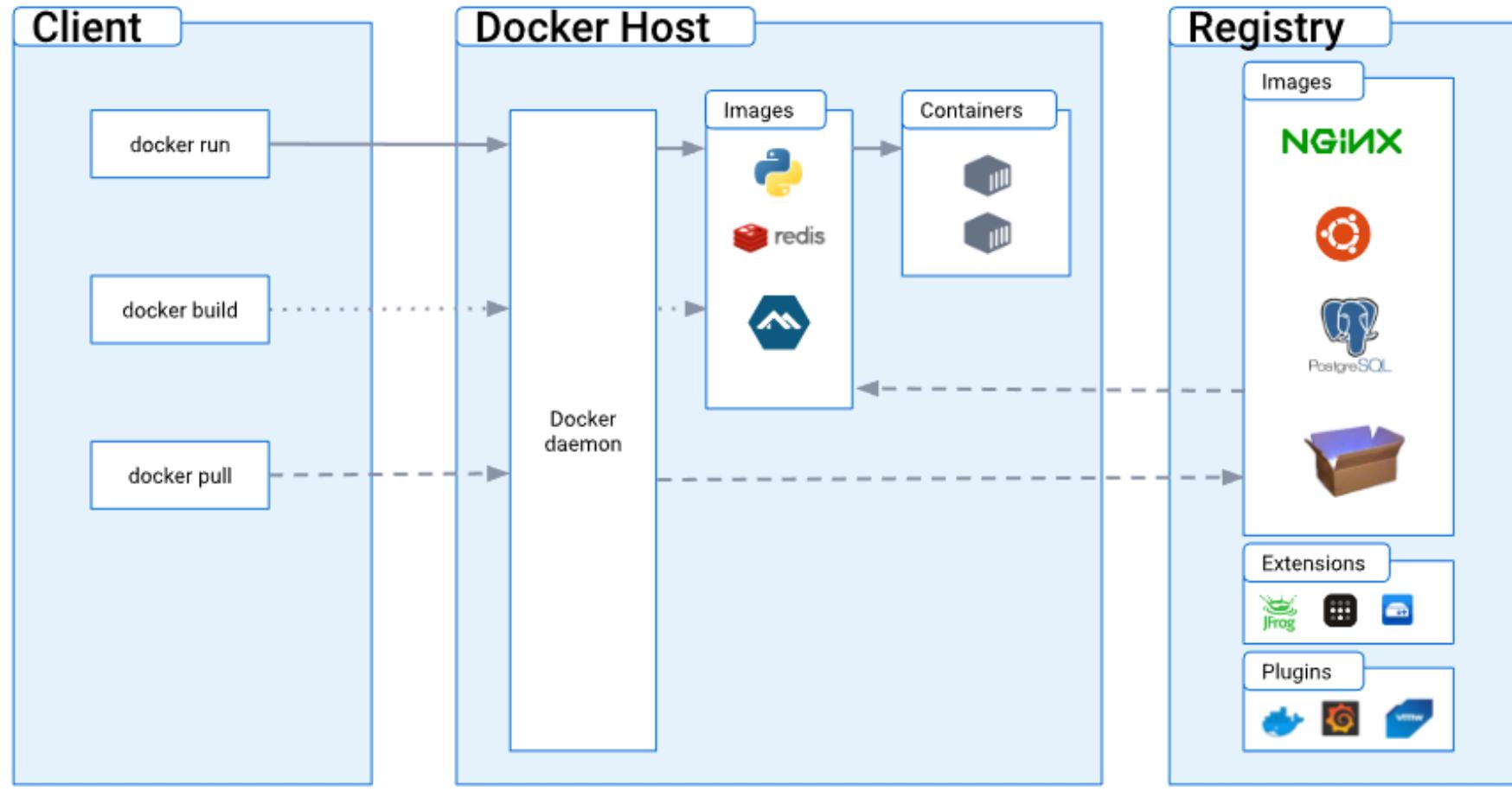
- Docker volumes store and manage data in containers.
- They enable data persistence and sharing between containers and the host system.



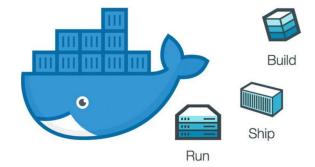
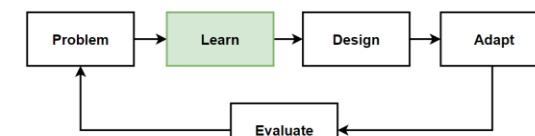
<https://docs.docker.com/get-started/overview/>

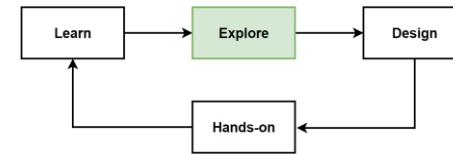
Docker Architecture

- Docker Daemon, Docker Client, Docker API, Docker Images, Docker Registries, Docker Containers, Docker Networks, Docker Volumes.



<https://docs.docker.com/get-started/overview/>



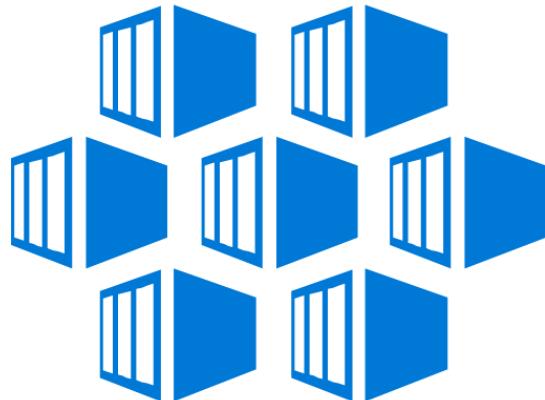


Explore: Container tools: Container Runtimes, Container Registries, Container Deployment Options

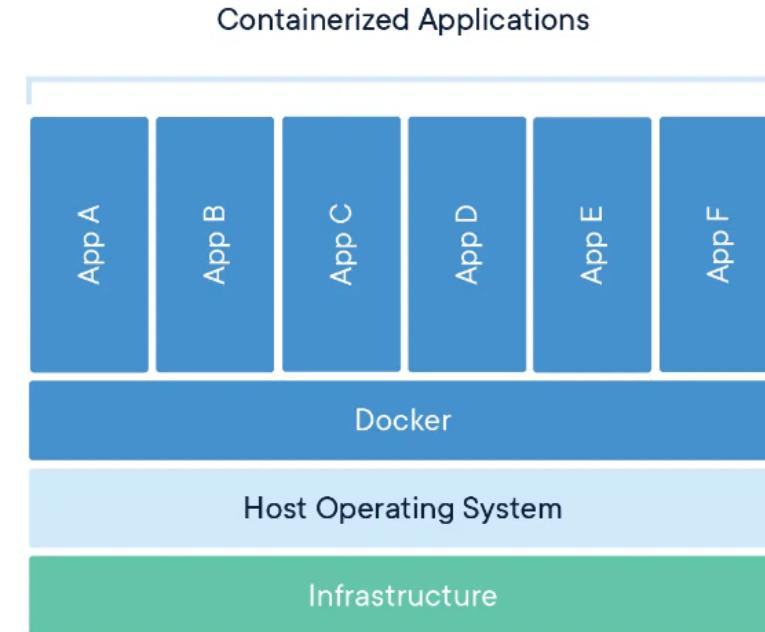
- Container tools: Container Runtimes, Container Registries, Container Deployment Options

Exploring Container Tools

- Container Runtimes
- Container Registries
- Container Deployment Options

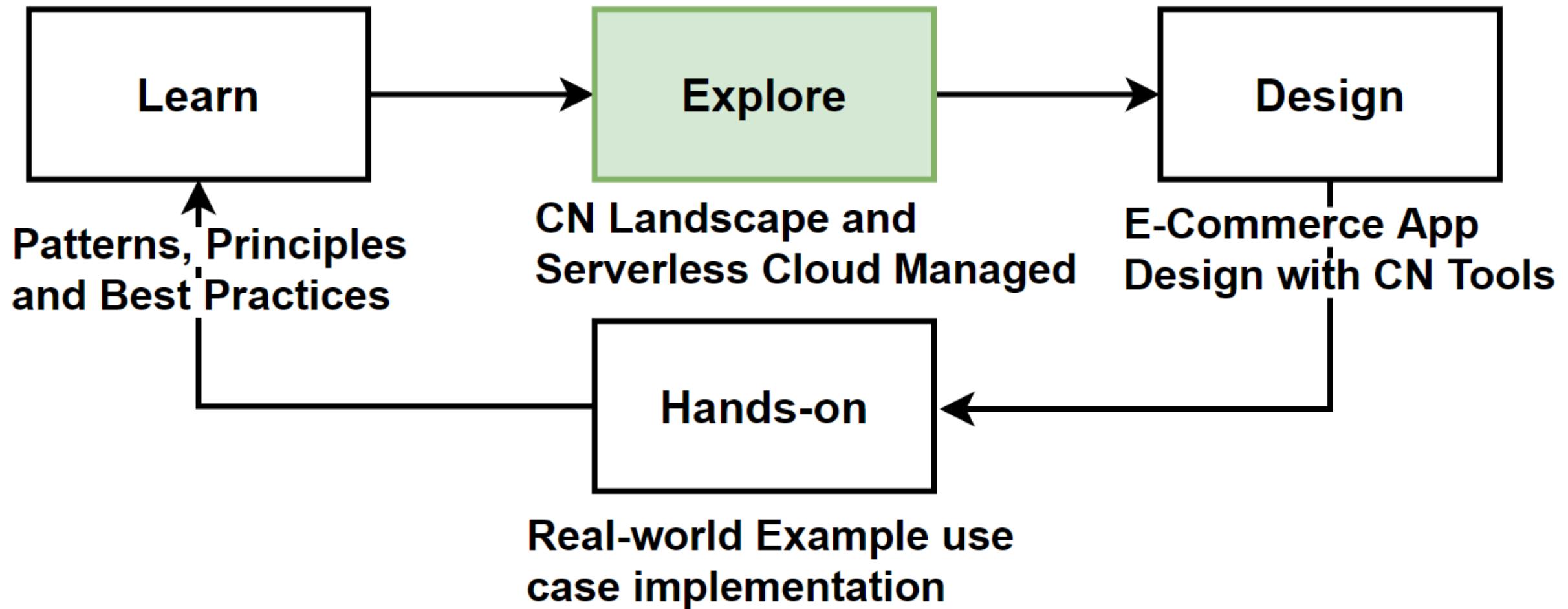


Containers

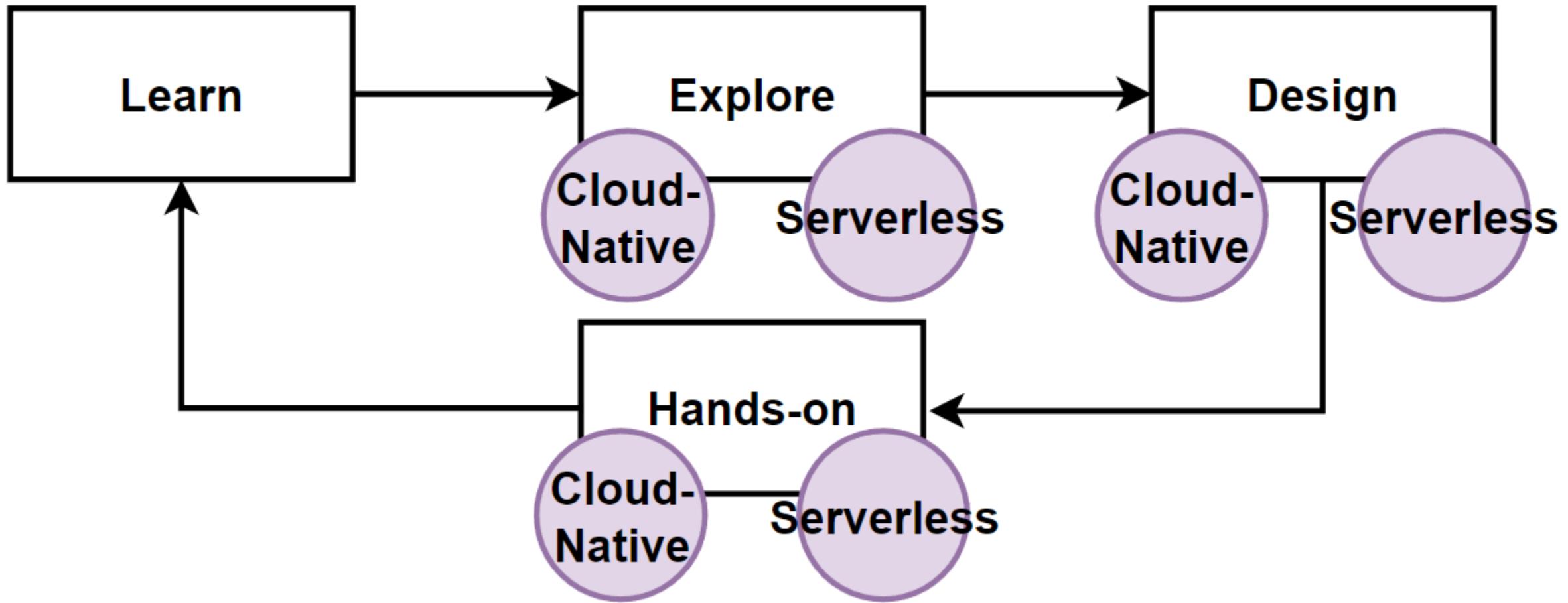


<https://landscape.cncf.io/>

Way of Learning – The Course Flow

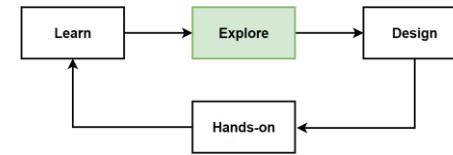


Way of Learning – Cloud-Native & Serverless Cloud Managed



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

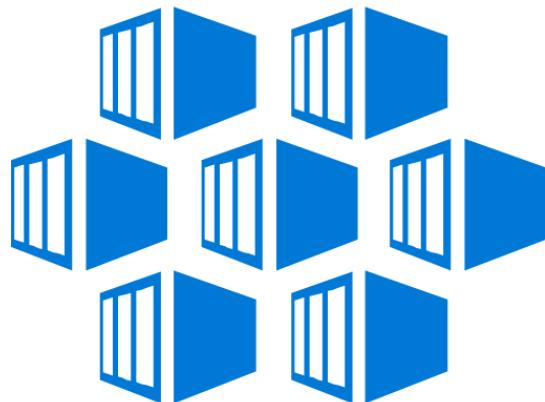


Explore: Container tools: Container Runtimes, Container Registries, Container Deployment Options

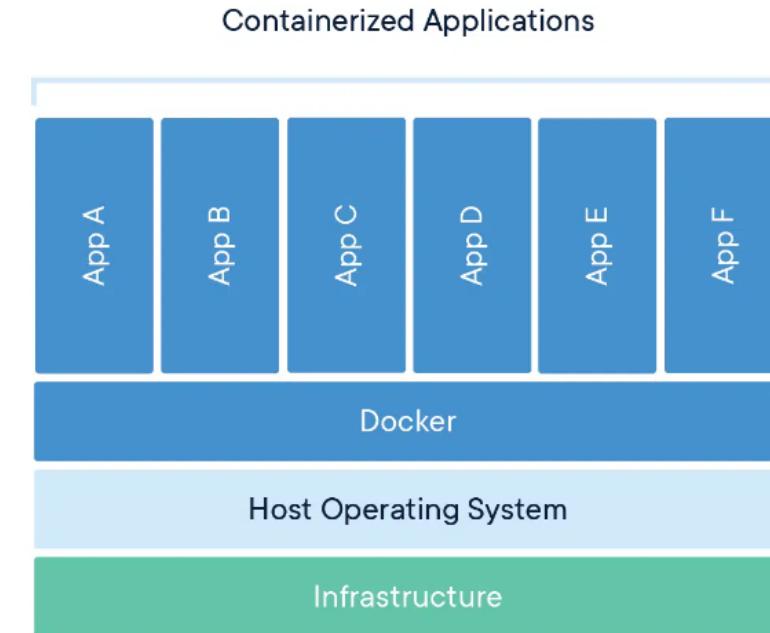
- Container tools: Container Runtimes, Container Registries, Container Deployment Options

Exploring Container Tools

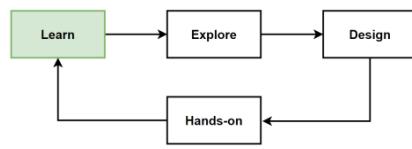
- Container Runtimes
- Container Registries
- Container Deployment Options



Containers

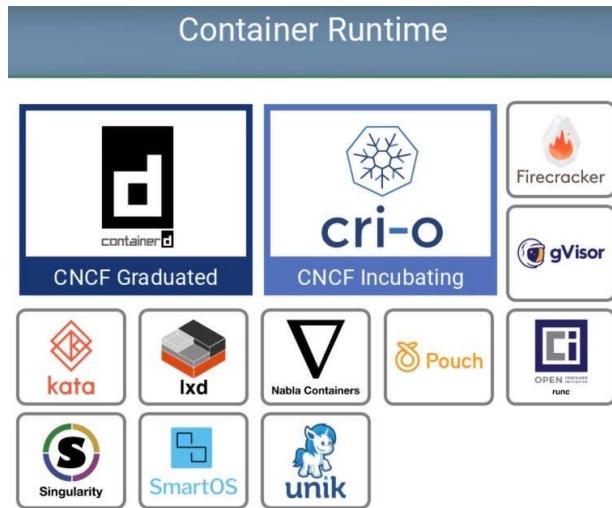


<https://landscape.cncf.io/>



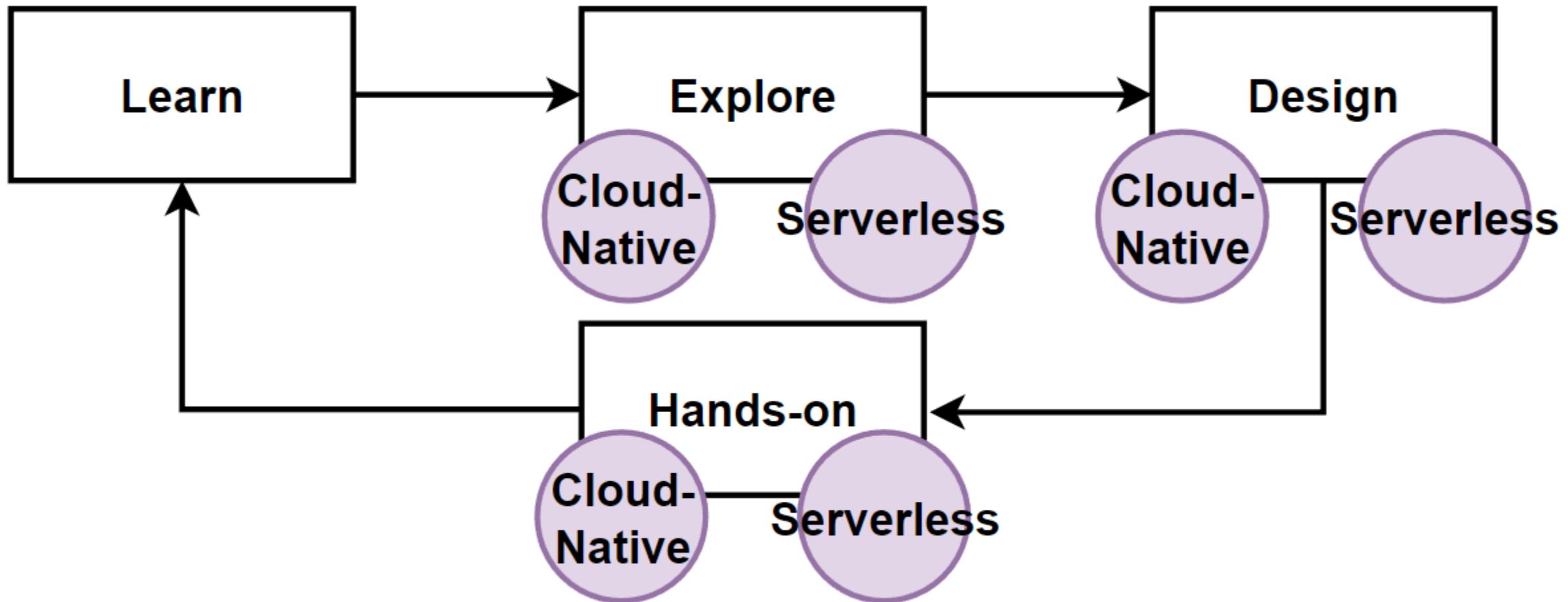
Explore: Container Runtimes

- A container runtime executes and manages containers that provides the necessary environment to run containerized applications handles tasks; creation, starting, stopping, and monitoring of containers.
- Docker**
The most known container runtime synonymous with container technology. Uses its own runtime, containerd, to manage containers.
- containerd**
Docker's container runtime, also operable independently. It's lightweight and focuses on core container execution functionalities.
- CRI-O**
Lightweight container runtime designed specifically for Kubernetes. Implements the Kubernetes Container Runtime Interface (CRI), allowing Kubernetes to use any OCI-compliant runtime.
- runc**
low-level container runtime based on the Open Container Initiative (OCI) runtime specification. It's the default runtime used by containerd and can also be used standalone for running containers.



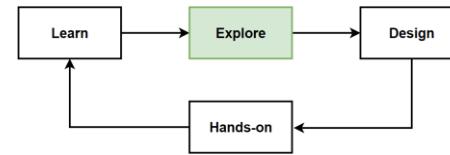
<https://landscape.cncf.io/>

Way of Learning – Cloud-Native & Serverless Cloud Managed



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Cloud Serverless Functions for Serverless Runtimes

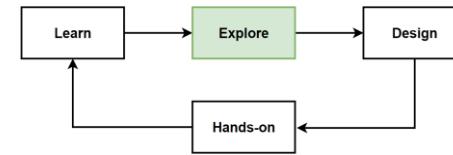
AWS Lambda

- Amazon's serverless platform, ideal for running code in response to events like database changes or message arrival. Executes code only when needed and scales automatically.
- Event-driven, compute-on-demand service
- [AWS Lambda Runtimes: NodeJS, Python, Java, .NET](#)

Azure Functions

- Microsoft's serverless offering, providing event-driven, compute-on-demand similar to AWS Lambda. Ideal for handling HTTP requests, data processing, or running background tasks.
- Serverless computing service similar to AWS Lambda



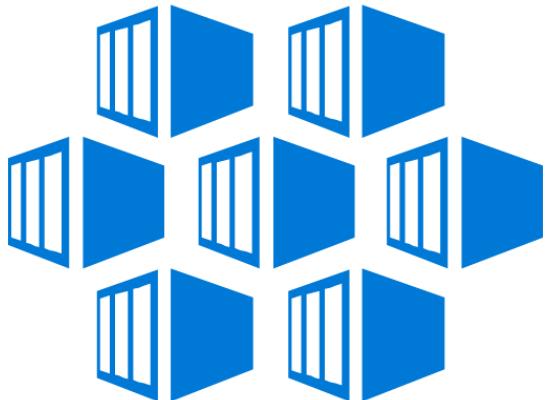


Explore: Container tools: Container Runtimes, Container Registries, Container Deployment Options

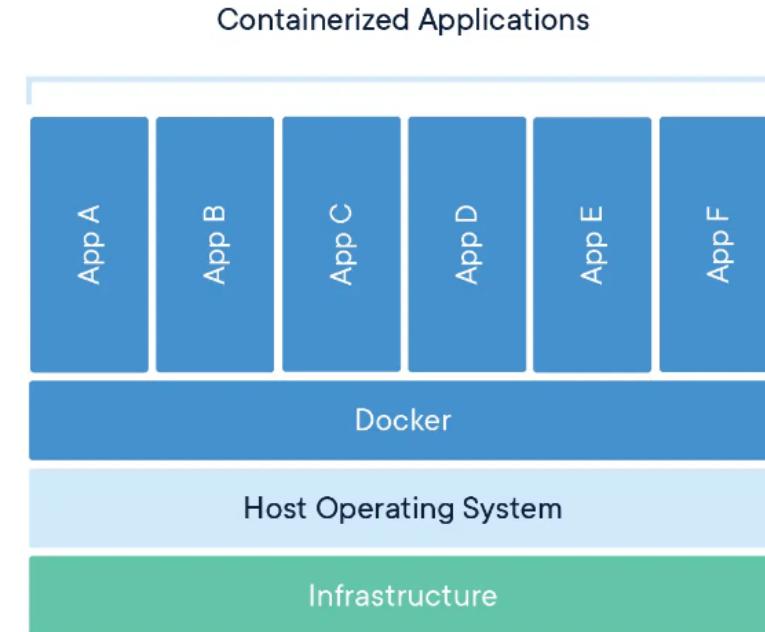
- Container tools: Container Runtimes, Container Registries, Container Deployment Options

Exploring Container Tools

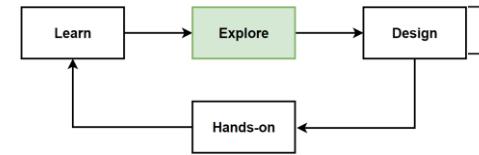
- Container Runtimes
- Container Registries
- Container Deployment Options



Containers

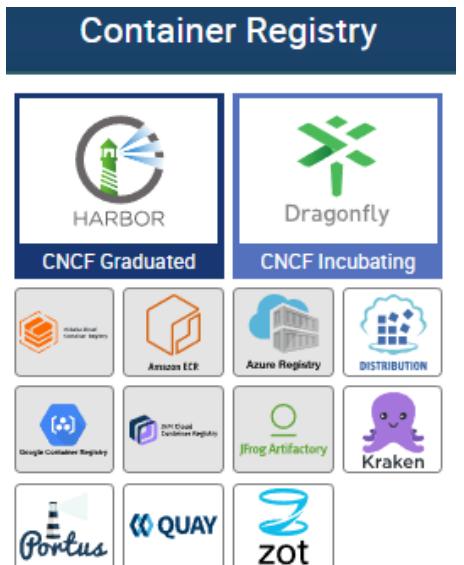


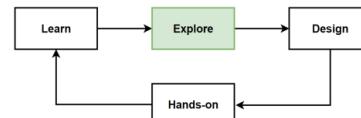
<https://landscape.cncf.io/>



Explore: Container Registries – Cloud-Native CNCF

- Container registries are centralized repositories for storing and sharing container images.
- Conjunction with container runtimes and orchestration tools to deploy and manage containerized applications.
- **Docker Hub**
Default public registry for Docker. Hosts community images, supports public/private repositories. Integrated with Docker CLI tools.
- **Harbor**
Open-source container image registry. Role-based access control, vulnerability scanning, image signing. A graduated project within the CNCF landscape
- **Quay.io (by Red Hat)**
Secure private container registry. Supports Docker and OCI images. Features include automated builds, vulnerability scanning, RBAC
- **JFrog Artifactory**
Universal binary repository manager. Supports Docker containers and Helm chart repositories. Secure, high availability registries for Docker images





Explore: Container Registries – Serverless Cloud Managed

- **Google Container Registry (GCR)**

Google Cloud's private container registry. Supports secure Docker image storage. Seamlessly integrated with Google Kubernetes Engine (GKE)

- **Amazon Elastic Container Registry (ECR)**

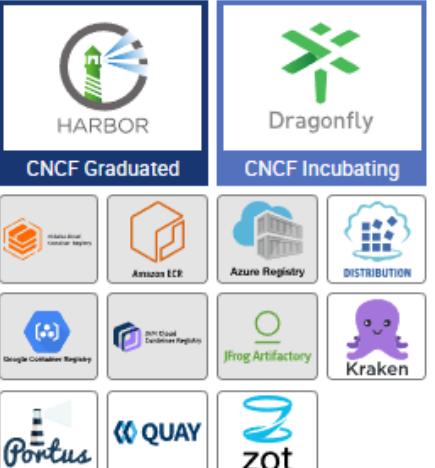
Fully-managed Docker container registry by Amazon. Easy storage, management, deployment of Docker images. Integrated with Amazon ECS and AWS Fargate

- **Azure Container Registry (ACR)**

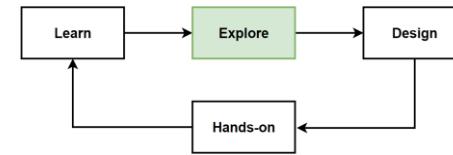
Microsoft's managed Docker registry service. Stores and manages container images centrally. Integrated with Azure Kubernetes Service (AKS), Azure DevOps, etc.



Container Registry



<https://landscape.cncf.io/>

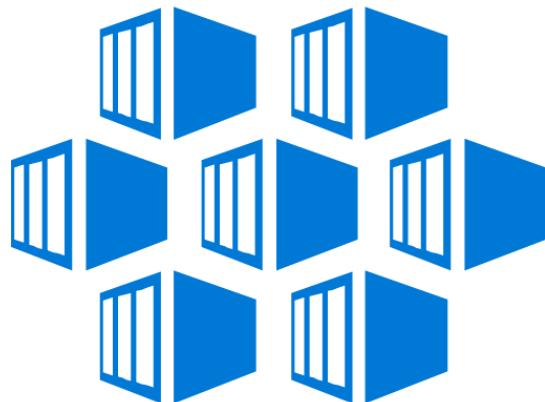


Explore: Container tools: Container Runtimes, Container Registries, Container Deployment Options

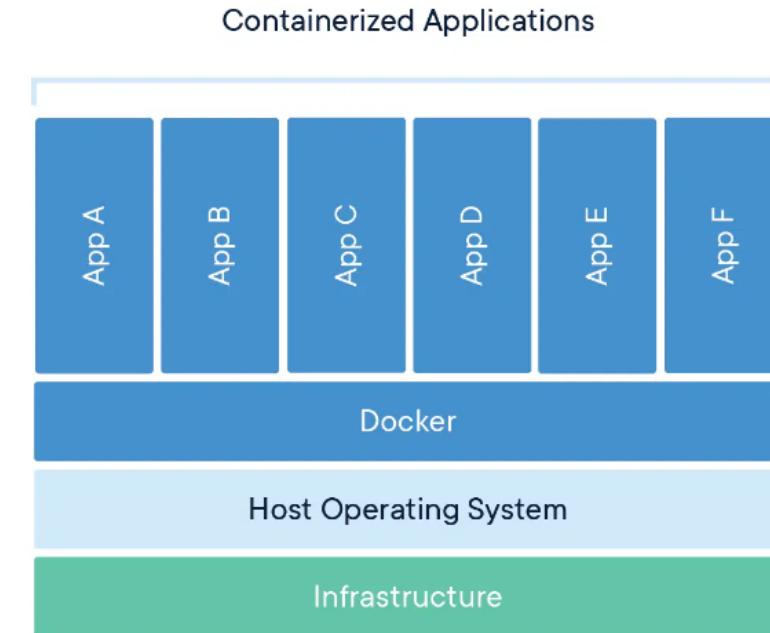
- Container tools: Container Runtimes, Container Registries, Container Deployment Options

Exploring Container Tools

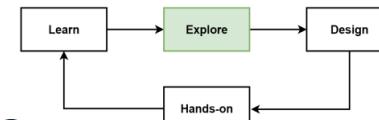
- Container Runtimes
- Container Registries
- Container Deployment Options



Containers



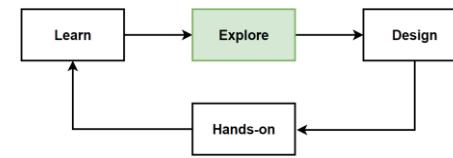
<https://landscape.cncf.io/>



Explore: Container Deployment – Serverless Cloud Management

- PaaS tools for a single web application container deployments
- **AWS Elastic Beanstalk**
Enables quick deployment and management of cloud applications without having to learn about the underlying infrastructure. Handles capacity provisioning, load balancing, scaling, and application health monitoring.
- **Google App Engine**
Support the development and hosting of web applications in Google-managed data centers. Applications are sandboxed, run, and scaled automatically across multiple servers.
- **Microsoft Azure App Service**
Fully managed PaaS that combines various Azure services into a single platform.
- **Salesforce Heroku**
Early and much-loved PaaS. Acquired by the SaaS giant Salesforce in 2010.
- **Cloud providers offers serverless container deployment options**
AWS Fargate, Azure Container Apps, and Google Cloud Run products. Abstraction of managed kubernetes services on cloud that can deploy and run containers without thinking any server configurations.

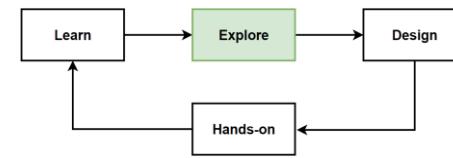




Cloud Container Services: Abstraction Level of Container Deployment Options

- **App Deployments > PaaS Deployments > Managed Kubernetes > Serverless Kubernetes > Serverless Functions (FaaS)**
- Abstraction of managed services on cloud that can deploy and run containers without thinking any server configurations.
- The way you increase the abstraction level, you don't need to think infrastructure but accept less flexible of configuring your infrastructure.
- **App Deployments**
If we have web app and would like to deploy on App Services
- **PaaS Deployments**
If we have single container, deploy to Container Instances
- **Kubernetes - Managed Kubernetes**
If we have multiple container that need to orchestrate we use Cloud Managed K8s
- **Serverless Kubernetes**
If we don't want to manage Kubernetes, we use Serverless Kubernetes
- **Serverless Functions**
If we don't want to manage containers at all, we use serverless functions

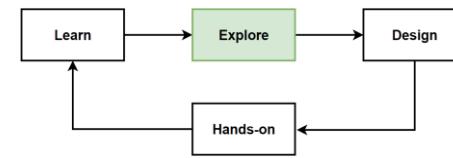




AWS Container Services: Abstraction Level of AWS Deployment Options

- **AWS Elastic Beanstalk > Amazon ECS > Amazon EKS > AWS Fargate > AWS Lambda**
- Abstraction of managed services on cloud that can deploy and run containers without thinking any server configurations.
- **AWS Elastic Beanstalk**
Deploy single web app containers without thinking scalability.
- **Amazon Elastic Container Service (ECS)**
Powerful container orchestration tool to manage a cluster of Amazon EC2 instances.
- **Amazon Kubernetes Service (EKS)**
Containerized orchestration tool for container applications managed by Kubernetes on the AWS cloud.
- **Amazon Fargate**
Enables administrators to run container clusters in the cloud without having to worry about the management of the underlying infrastructure.
- **AWS Lambda**
Serverless computing tool that lets you run code without the need to provision and manage servers.
Upload code as a container image and automatically provisions.

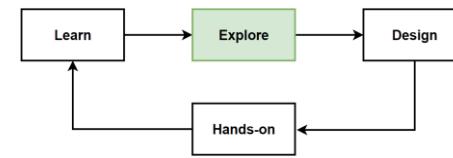




Azure Container Services: Abstraction Level of Azure Deployment Options

- **Azure App Service > Azure Container Instances > AKS > Azure Container Apps > Azure Functions**
- Abstraction of managed services on cloud that can deploy and run containers without thinking any server configurations. Build and deploy microservices using serverless containers.
- **Azure App Service**
Deploy web apps or APIs using containers in a PaaS environment.
- **Azure Container Instances**
Create individual containers in the cloud without any higher-level management services.
- **Azure Kubernetes Service (AKS)**
Scale and orchestrate Windows & Linux containers using Kubernetes.
- **Azure Container Apps**
Run your container workloads without managing servers, orchestration, or infrastructure and leverage native support for Dapr and KEDA.
- **Azure Functions**
Execute event-driven serverless code functions with an end-to-end development experience.

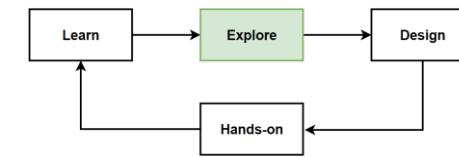




Google Cloud Container Services: Abstraction Level of Google Cloud Deployment Options

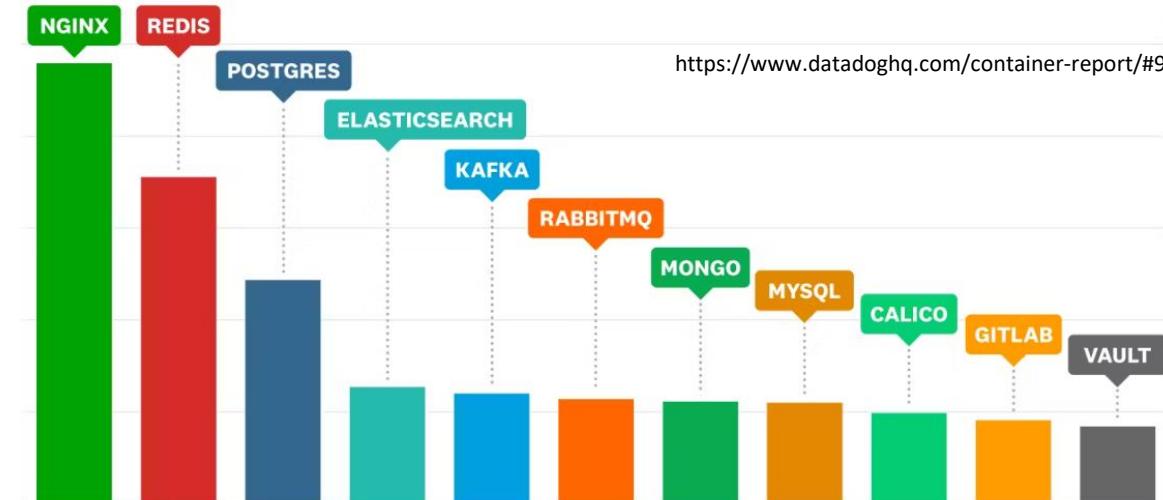
- **Google Cloud Run > Google Anthos > Google Kubernetes Engine (GKE) > Cloud Functions**
- Abstraction of managed services on cloud that can deploy and run containers without thinking any server configurations. Build and deploy microservices using serverless containers.
- **Google Cloud Run**
Serverless computing management platform for your container resources.
- **Google Anthos**
Hybrid and cloud-agnostic container environment management platform.
- **Google Kubernetes Engine (GKE)**
Managed Kubernetes service that lets you run Kubernetes clusters on Google Cloud infrastructure
- **Cloud Functions**
Run your code in the cloud with no servers or containers to manage with our scalable, pay-as-you-go functions as a service (FaaS) product.

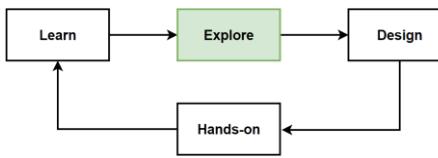




The Most Popular Container Images: Redis, Postgres, ElasticSearch, Kafka, MongoDB

- [Datadog Report: The Most Popular Container Images: Redis, Postgres, ElasticSearch, Kafka, MongoDB](https://www.datadoghq.com/container-report/#9)
- **NGINX**
NGINX provides caching, load balancing, and proxying capabilities to nearly 50 percent of organizations that use containers.
- **Redis**
Redis is an in-memory data structure store that can be used as a database, cache, and message broker.
- **PostgreSQL**
PostgreSQL is a powerful, open-source object-relational database system.
- **Elasticsearch**
Distributed, RESTful search and analytics engine capable of addressing a growing number of use cases.
- **Kafka**
Distributed streaming platform that's used for building real-time data pipelines and streaming apps.
- **RabbitMQ**
RabbitMQ supports decoupled architecture in microservice-based applications.

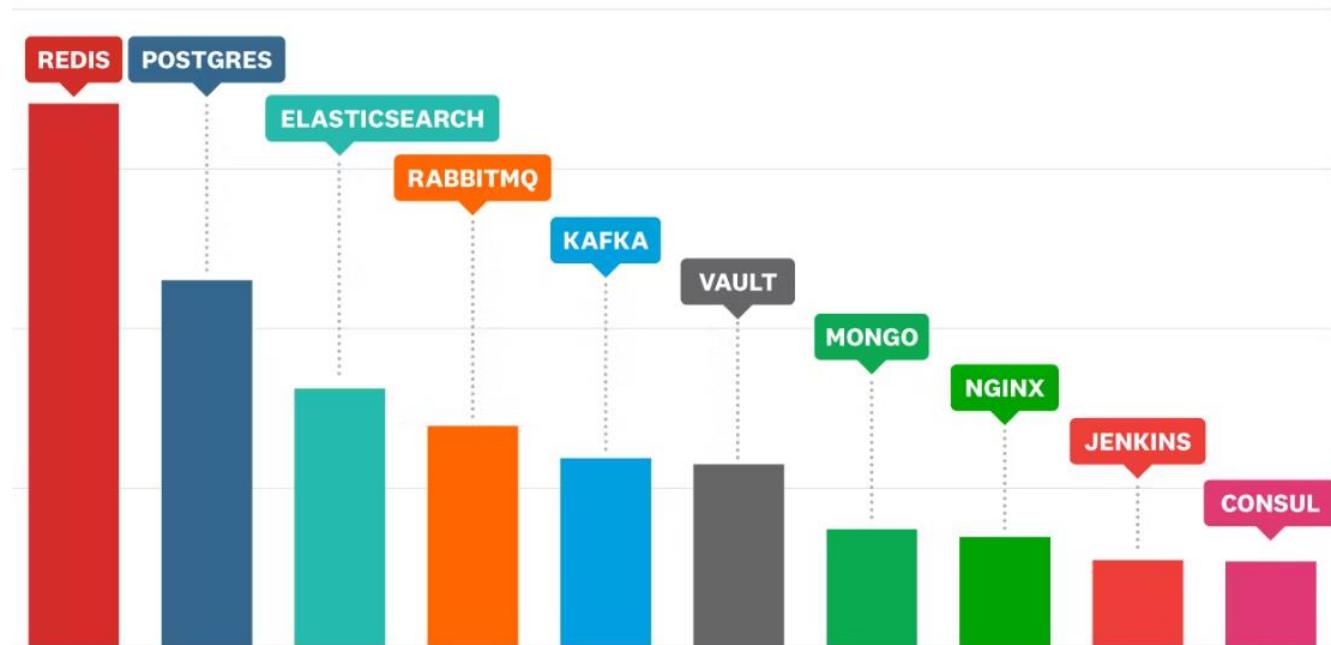




Kubernetes StatefulSets: Redis, Postgres, Elasticsearch, RabbitMQ, and Kafka

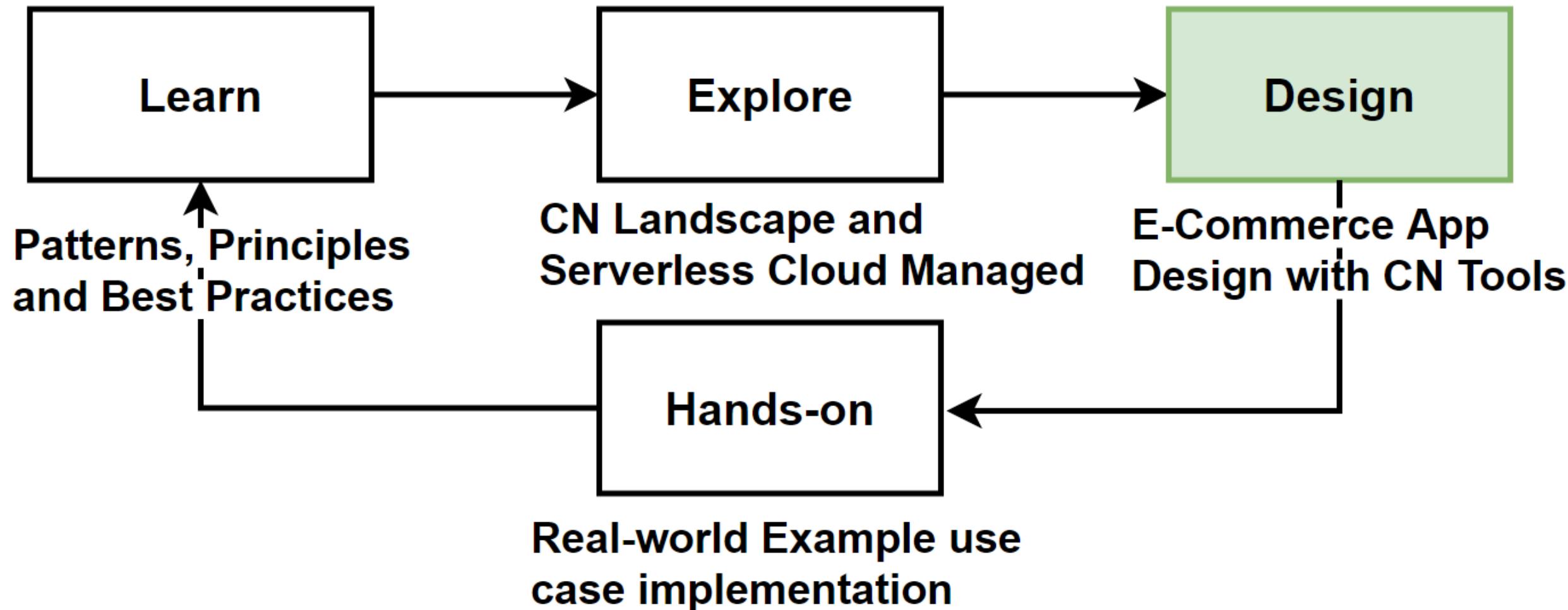
- [Datadog Report: The Most Popular Container Images: Redis, Postgres, ElasticSearch, Kafka, MongoDB](#)
- Kubernetes StatefulSets: Redis, Postgres, Elasticsearch, RabbitMQ, and Kafka were the most commonly deployed images.

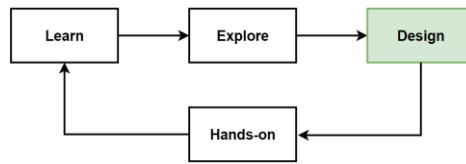
Top Container Images Running in Kubernetes StatefulSets



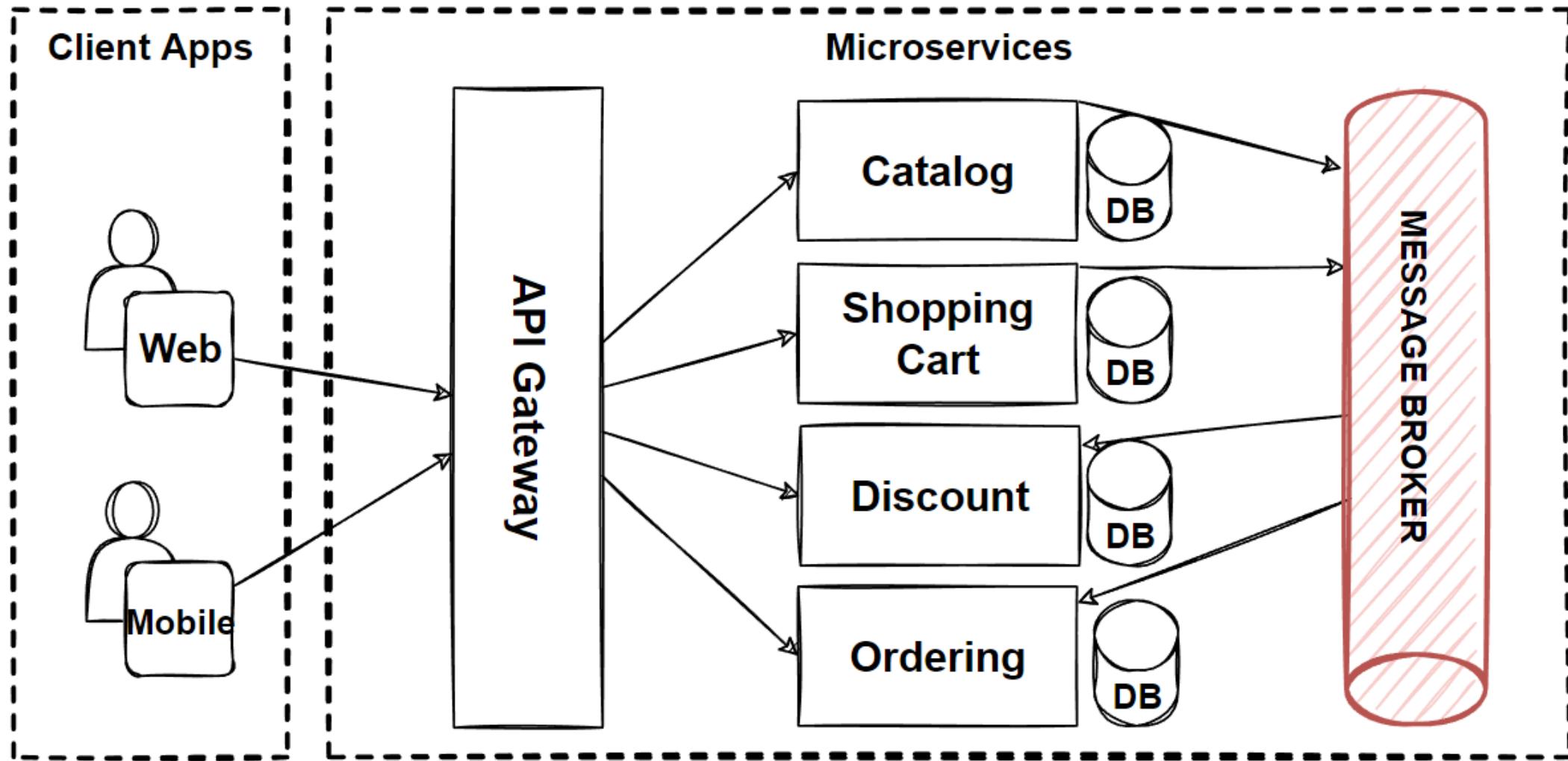
<https://www.datadoghq.com/container-report/#9>

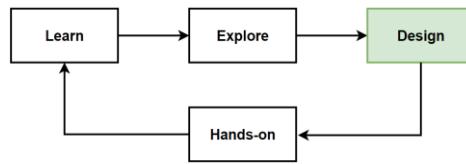
Way of Learning – The Course Flow



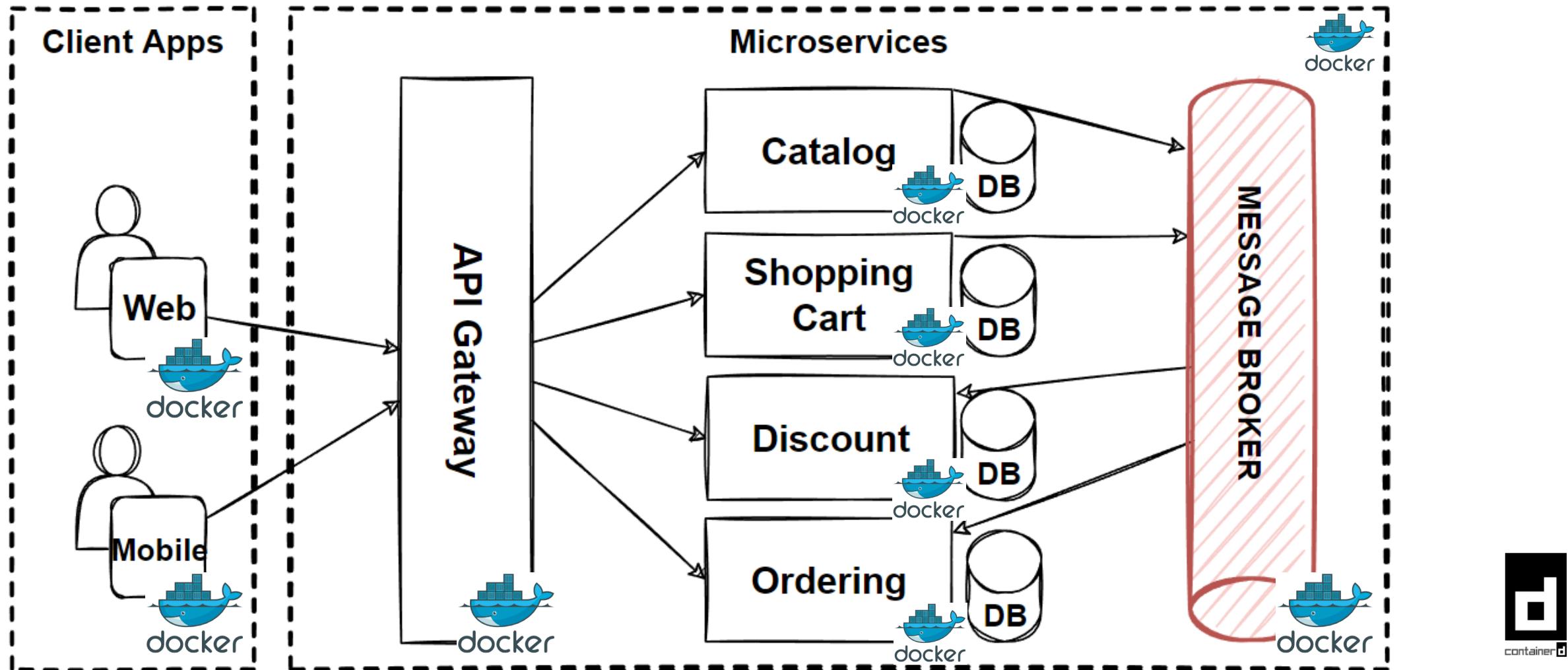


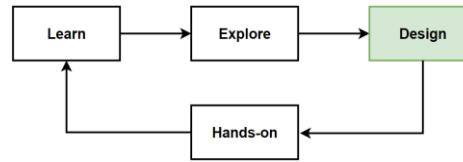
Design: Cloud-Native E-commerce Microservices





Design: Cloud-Native E-commerce w/ Containers





Containerization process with Docker

Create Dockerfiles

- For each microservice, create a Dockerfile
- Contains instructions for Docker to build an image for the microservice
- Base image (Python or Node.js), the application code, any dependencies

Build Docker Images

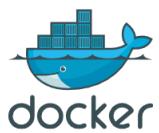
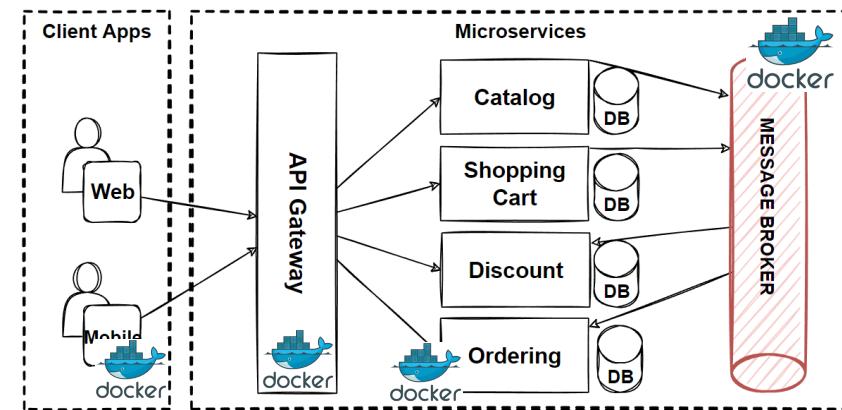
- Build Docker images for each of your microservices
- Images contain everything needed to run a particular microservice

Push Images to a Registry

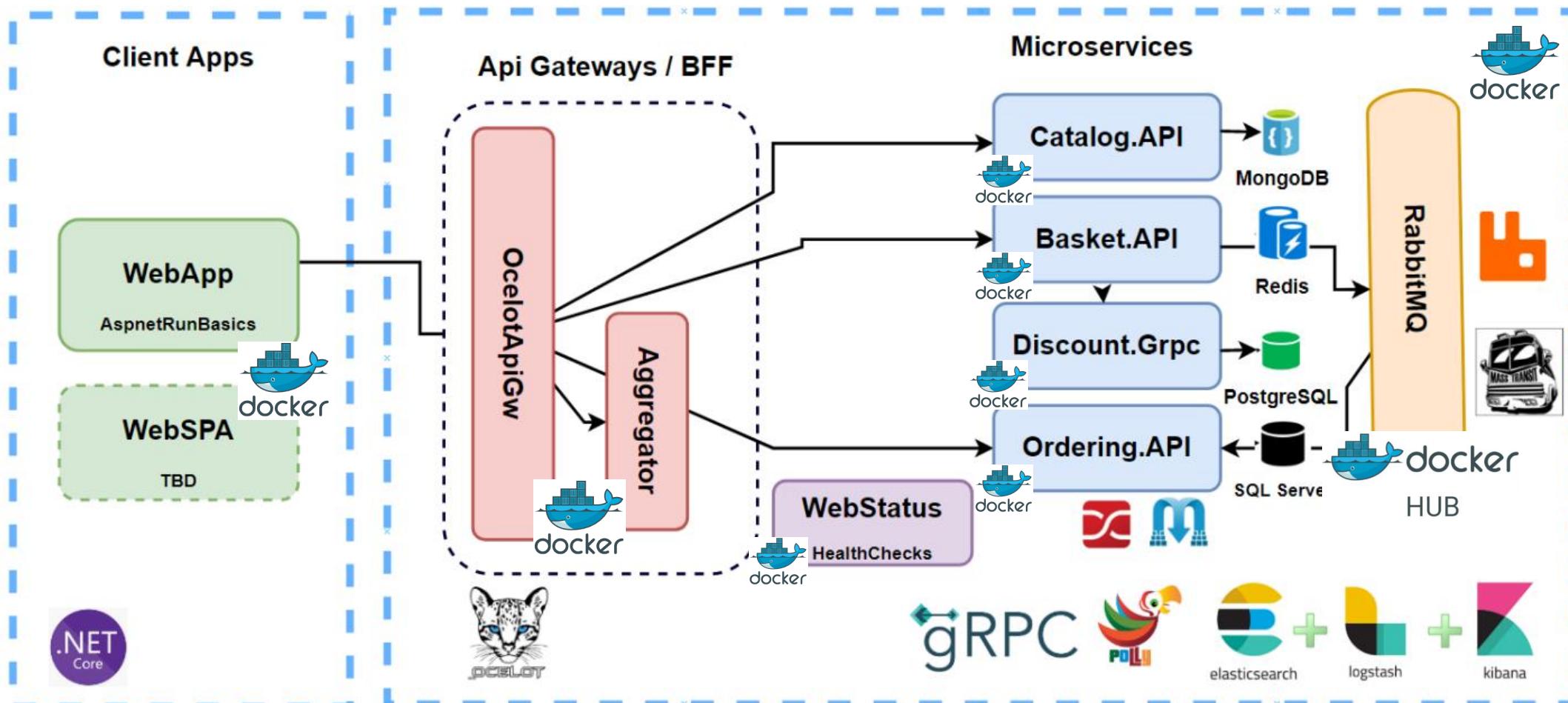
- Push images to a container registry like Docker Hub, Google Container Registry, or a private registry
- This makes the images available for deployment.

Deploy Containers

- Pull the images from the registry and start running containers.
- Each container runs an instance of a microservice.



Reference Project: .Net Microservices - Cloud-Native E-commerce



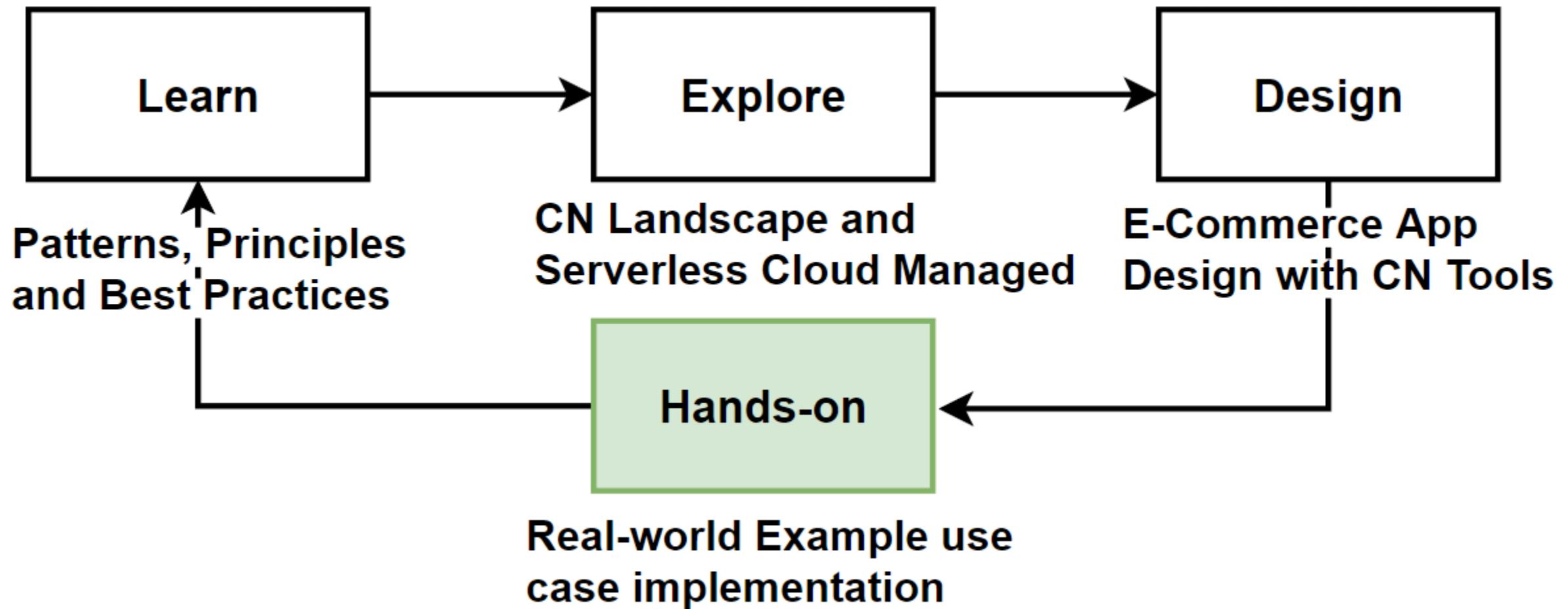
DEMO: Code review of Microservices Architecture .NET Implementation

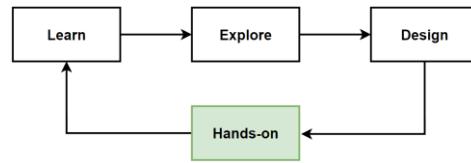
- <https://github.com/aspnetrun/run-aspnetcore-microservices>
- <https://github1s.com/aspnetrun/run-aspnetcore-microservices>

Hands-on: Containerize .Net Microservices with Docker

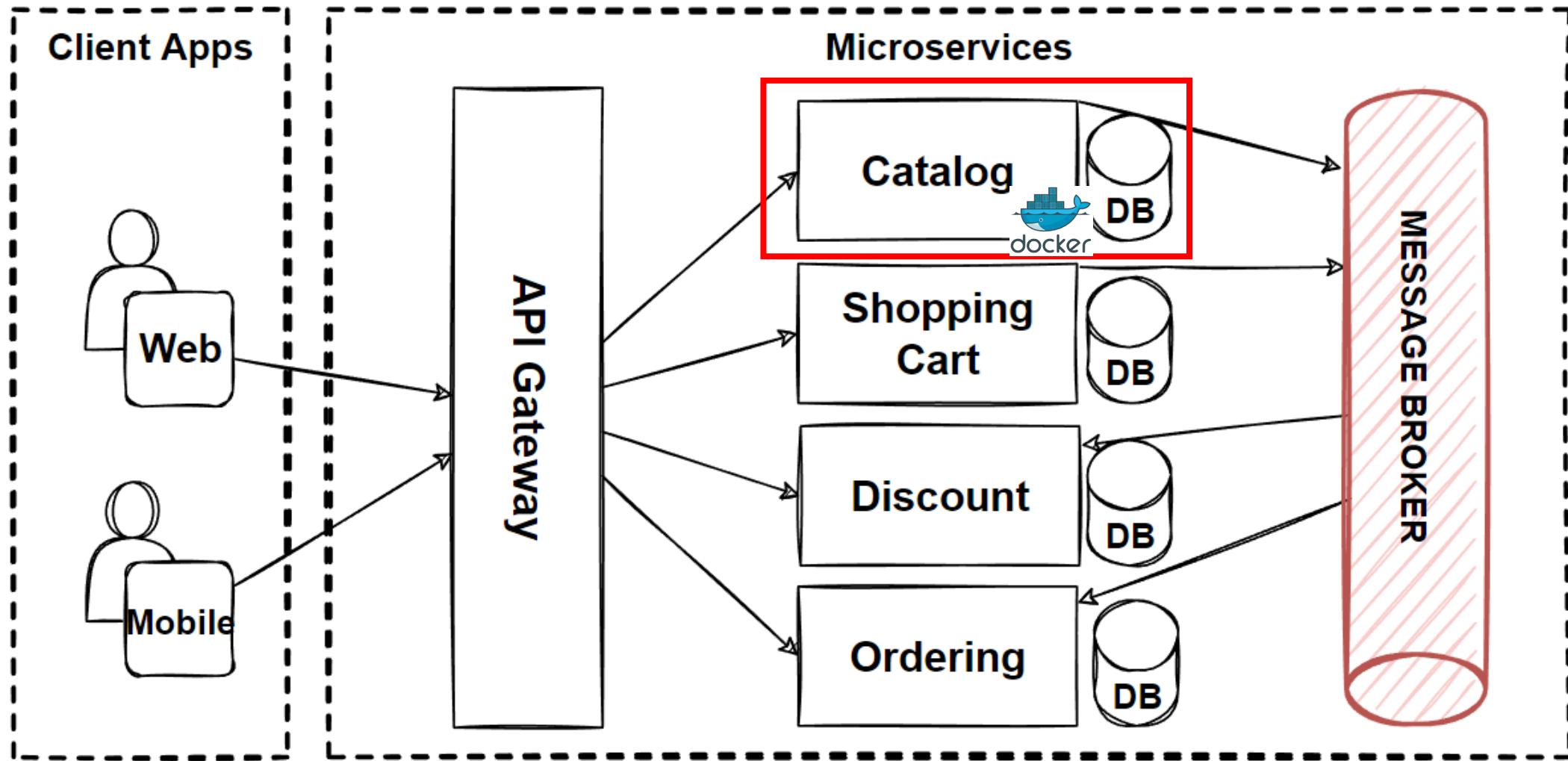
Leveraging docker for deploying, scaling, and managing containerized applications

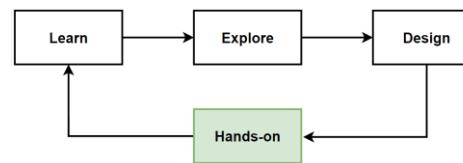
Way of Learning – The Course Flow



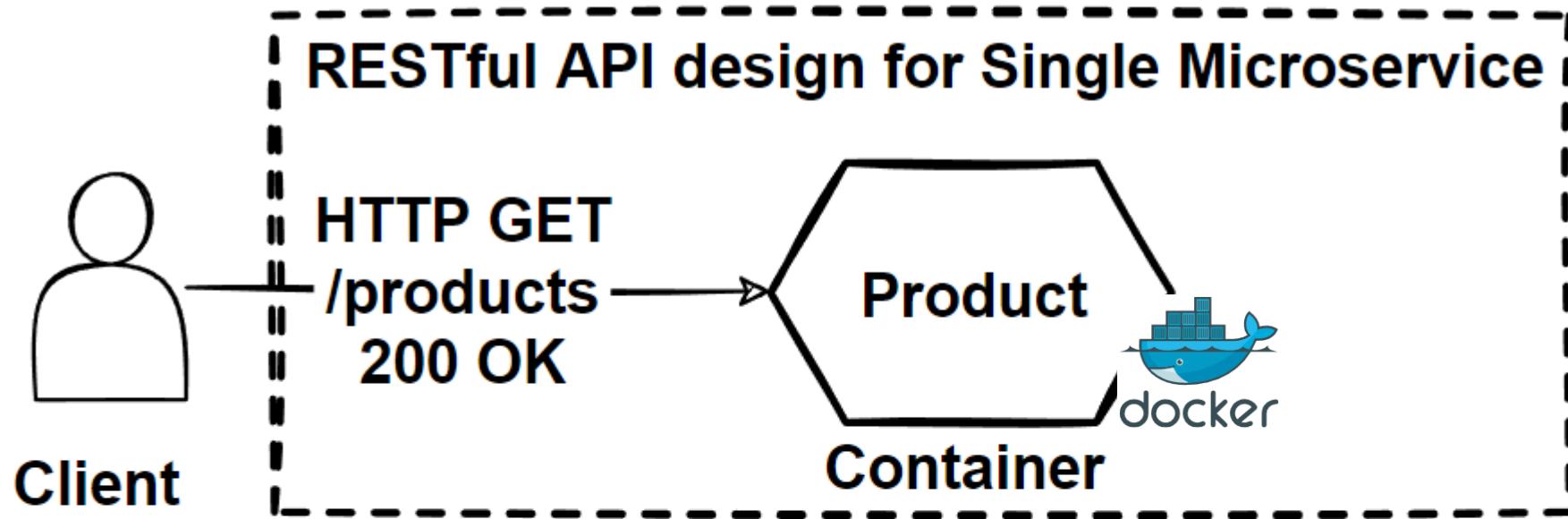


Design: Cloud-Native E-commerce Microservices





Hands-on: Containerize .Net Microservices with Docker



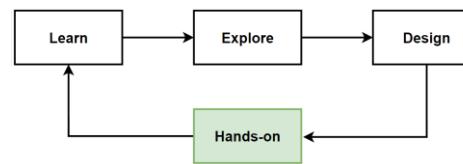
GET /api/products: Retrieves all products.

GET /api/products/{id}: Retrieves a specific product by ID.

POST /api/products: Adds a new product.

PUT /api/products/{id}: Updates an existing product by ID.

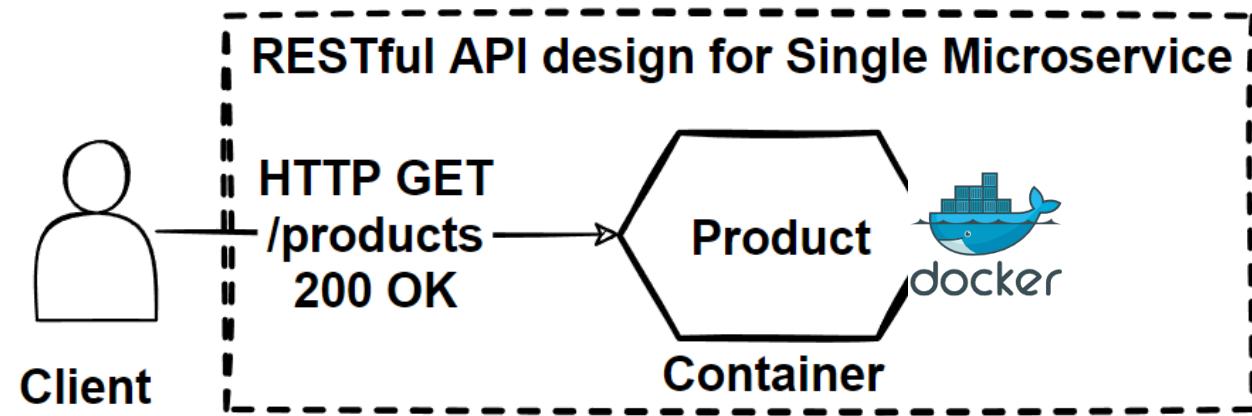
DELETE /api/products/{id}: Deletes a specific product by ID.



Hands-on: Containerize .Net Microservices with Docker

Todo List:

- Write Dockerfile
- Build Docker Image
- Run Docker Container
- Test running docker container on local docker env
- Tag Docker Image
- Publish image to a Registry: Docker Hub
- Deploy to Cloud: AWS, Azure

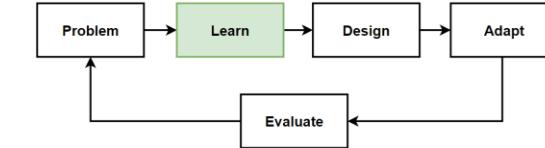


GET /api/products: Retrieves all products.
GET /api/products/{id}: Retrieves a specific product by ID.
POST /api/products: Adds a new product.
PUT /api/products/{id}: Updates an existing product by ID.
DELETE /api/products/{id}: Deletes a specific product by ID.

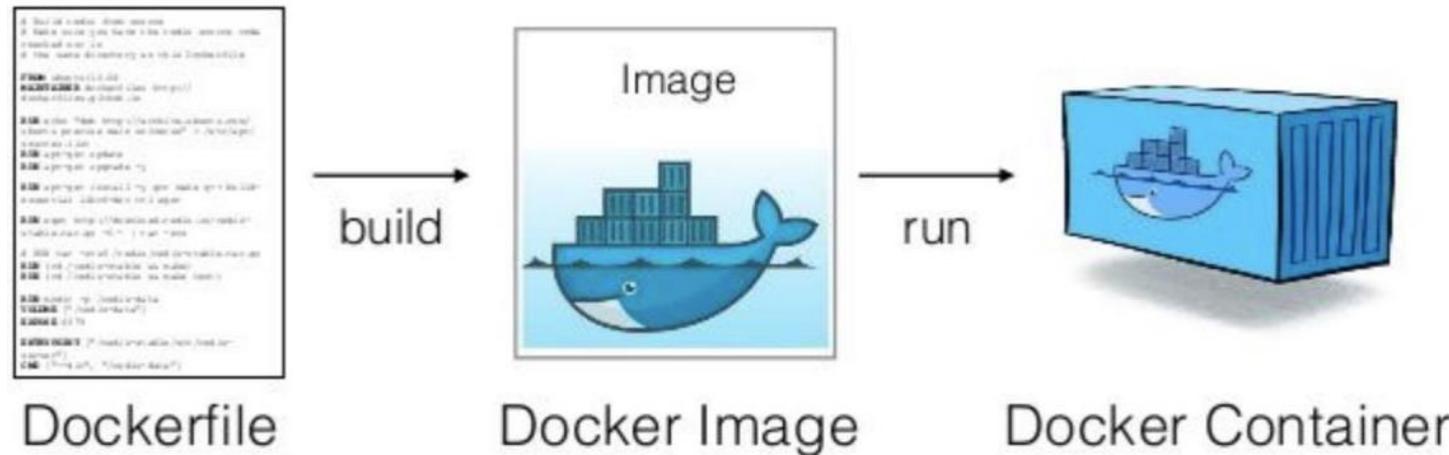
Download All Hands-on Project Codes

- <https://github.com/mehmetozkaya/CloudNative>

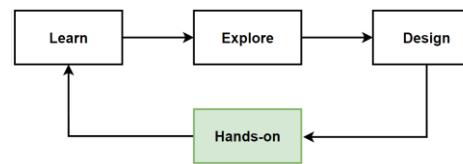
Application Containerization with Docker



1. Write Dockerfile for our application.
 2. Build application with this docker file and creates the docker images.
 3. Run this images on any machine and creates running docker container from docker image.



- **Orchestrating whole microservices application with Docker and Kubernetes.**



Install Prerequisites

Install Docker Desktop

- To containerize .NET apps, download and install the Docker Desktop.
- <https://www.docker.com/products/docker-desktop/>

Start Docker

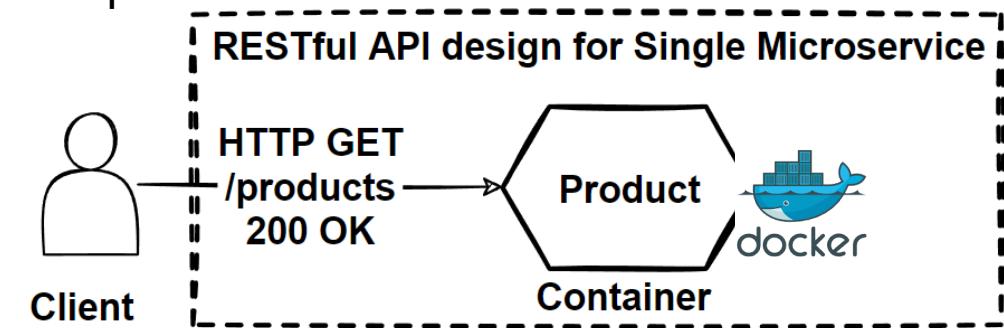
- Open Docker Desktop App

Check Docker Command

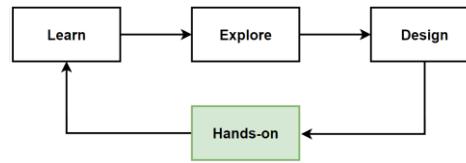
- docker –version

Download All Hands-on Project Codes

- <https://github.com/mehmetozkaya/CloudNative>



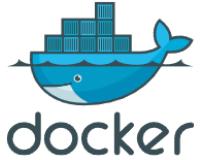
GET /api/products: Retrieves all products.
GET /api/products/{id}: Retrieves a specific product by ID.
POST /api/products: Adds a new product.
PUT /api/products/{id}: Updates an existing product by ID.
DELETE /api/products/{id}: Deletes a specific product by ID.



Deploy Container to Cloud: AWS Apprunner, Google Cloud Run, Azure Container Instance

Deploy Single Container to Cloud

- Single application or microservice deployment on the cloud
- No need for managed Kubernetes or serverless orchestration services
- Focus on cloud services that allow running single containers with abstracted infrastructure
- Google Cloud Run, AWS App Runner, and Azure Container Instances are all managed services that allow you to run containerized applications



Google Cloud Run

- Fully managed compute platform for running stateless containers
- Automatic scaling based on incoming traffic
- Pay only for compute resources used during request processing
- Built on open-source Knative project for complete control

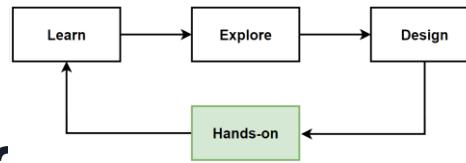


AWS App Runner

- Fully managed service for building, deploying, and scaling containerized applications
- Infrastructure is abstracted away, allowing focus on application code
- Automatic build, deployment, scaling, and health monitoring of applications



AWS App Runner



Deploy Container to Cloud: AWS Apprunner, Google Cloud Run, Azure Container Instance 2

Azure Container Instances (ACI)

- Managed service for running containers directly on Azure cloud
- No need to manage virtual machines or Kubernetes clusters
- Designed for fast and easy container deployment with automatic scaling, patching, and management



Comparison Main Features

- **Ease of use:** Simplifies deployment and management of containerized applications
- **Scaling:** Automatic scaling based on demand
- **Pricing:** Pay-as-you-go model, paying only for used compute resources
- **Integration with cloud services:** Each service integrates well with its respective cloud platform's services

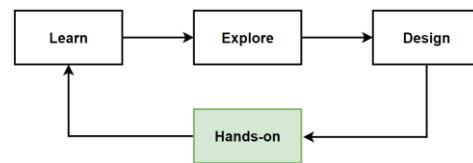


"Scale to Zero" Feature

- Cost-effective for single container deployments
- Ideal for side projects or minimum viable products (MVPs)
- Currently offered by Google Cloud Run, soon to be available in AWS App Runner
- [GitHub Issue for AWS Apprunner](#)



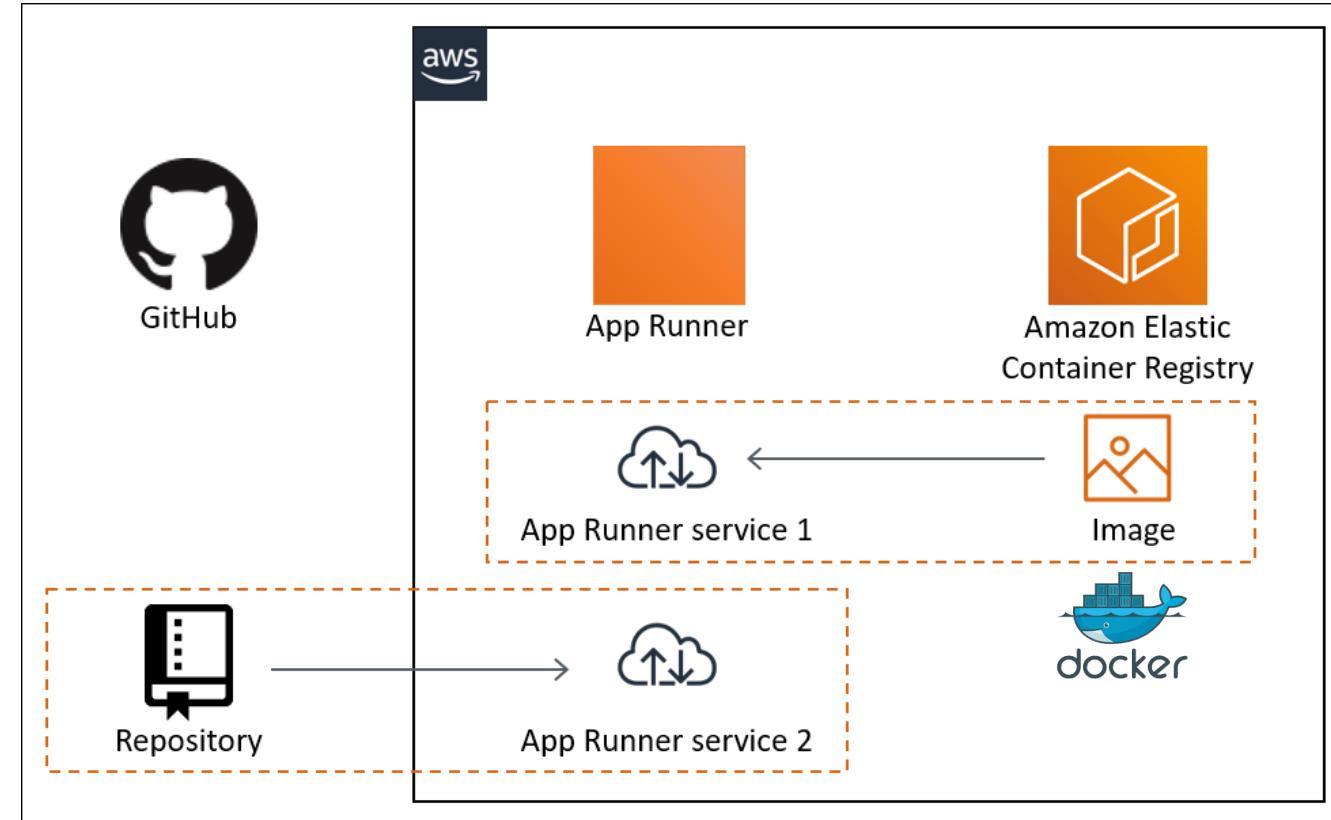
AWS App Runner

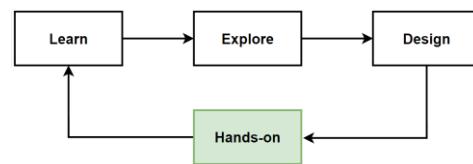


Deploy ProductService Container to AWS Apprunner

Todo List:

- Push your Docker container image to Amazon Elastic Container Registry (ECR)
- Deploy to AWS App Runner that pull image from ECR

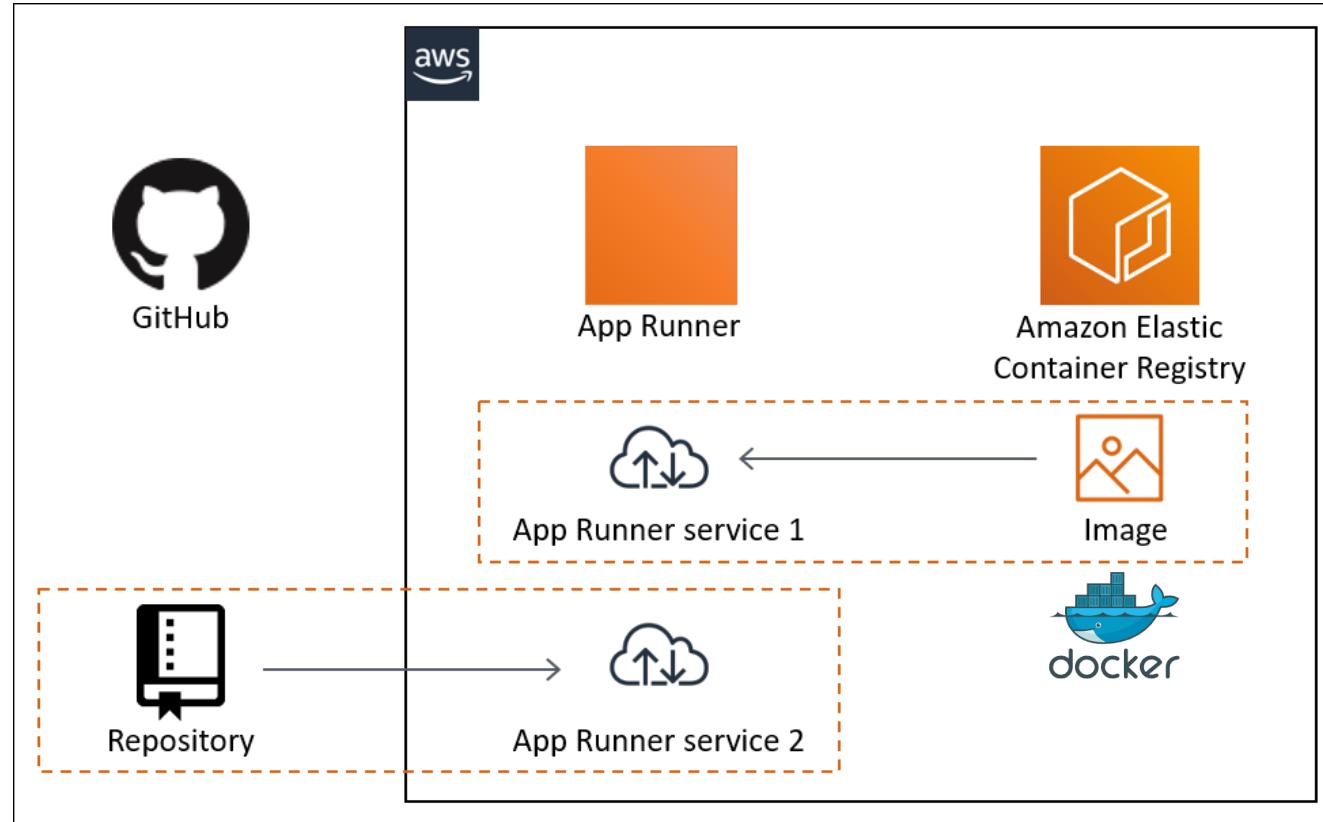




Prepare your AWS Environment

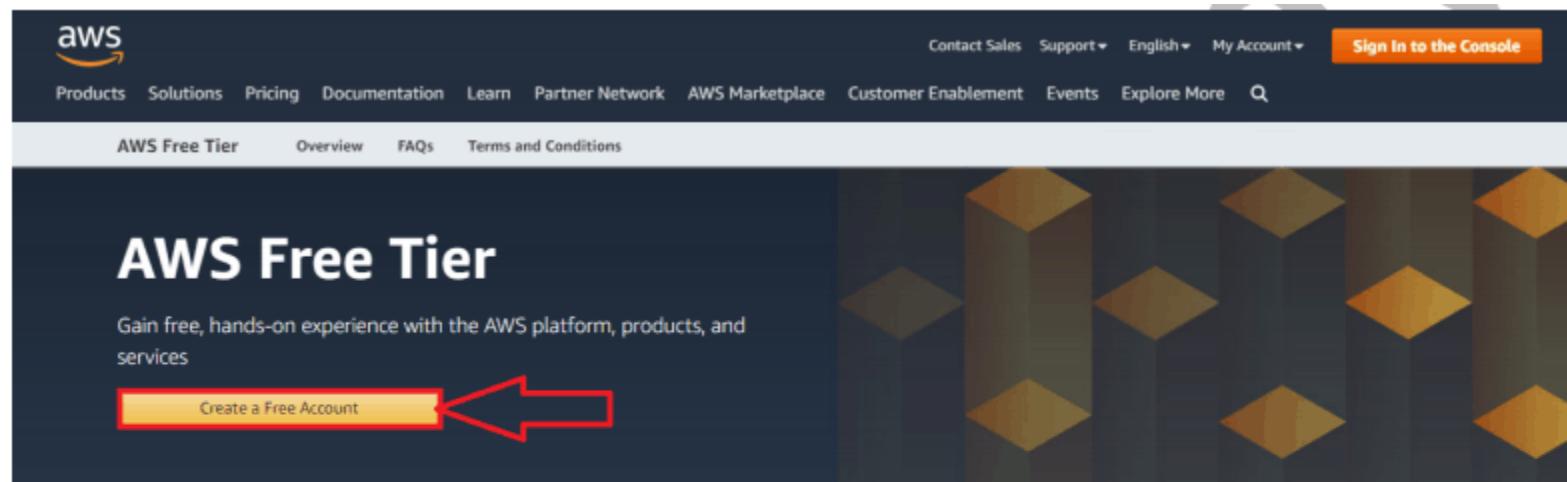
Activate:

- AWS Account
- IAM User
- Access Key Programmatic Access
- AWS CLI



Create Free Tier AWS Account

- [Clik Here for AWS Create Free Tier Account](#)
- [Follow the Steps to Activate Free Tier Account](#)
- More than 100 products for building applications.
- Most of the services allowed good amount of usage depending on the product
- For example **25 GB DynamoDb, 1 million Lambda request, 1 TB Amazon CloudFront ..**



The screenshot shows the AWS Free Tier landing page. At the top, there's a navigation bar with links like Products, Solutions, Pricing, Documentation, Learn, Partner Network, AWS Marketplace, Customer Enablement, Events, Explore More, and a search bar. Below the navigation, there are links for AWS Free Tier, Overview, FAQs, and Terms and Conditions. The main title "AWS Free Tier" is prominently displayed. A sub-section titled "Types of offers" describes three types: "Always free", "12 months free", and "Trials". Each type has a corresponding icon and a brief description. A large yellow button labeled "Create a Free Account" is highlighted with a red arrow pointing to it.

AWS Free Tier

Gain free, hands-on experience with the AWS platform, products, and services

[Create a Free Account](#)

Types of offers

Explore more than 60 products and start building on AWS using the free tier. Three different types of free offers are available depending on the product used. See below for details on each product.

Always free

These free tier offers do not expire and are available to all AWS customers

12 months free

Enjoy these offers for 12-months following your initial sign-up date to AWS

Trials

Short-term free trial offers start from the date you activate a particular service

Select a Support Plan

- AWS support offers a selection of plans to meet your business needs.
- Select **Basic Plan** support for free account
- created AWS Free Tier Account
- Wait for account activation
- If you face any problem during this process, you can [follow this page](#)

Select a Support Plan

AWS offers a selection of support plans to meet your needs. Choose the support plan that best aligns with your AWS usage. [Learn more](#)

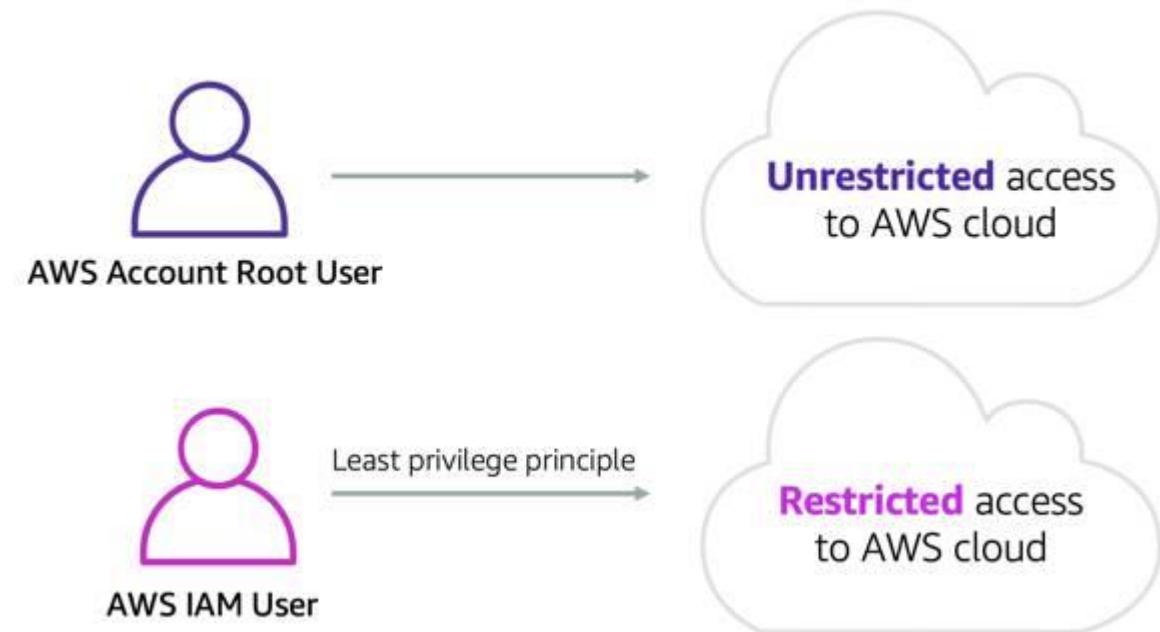
Support Plan	Cost	Description
Basic Plan	Free	<ul style="list-style-type: none">• Included with all accounts• 24x7 self-service access to AWS resources• For account and billing issues only• Access to Personal Health Dashboard & Trusted Advisor
Developer Plan	From \$29/month	<ul style="list-style-type: none">• For early adoption, testing and development• Email access to AWS Support during business hours• 1 primary contact can open an unlimited number of support cases• 12-hour response time for nonproduction systems
Business Plan	From \$100/month	<ul style="list-style-type: none">• For production workloads & business-critical dependencies• 24/7 chat, phone, and email access to AWS Support• Unlimited contacts can open an unlimited number of support cases• 1-hour response time for production systems

Need Enterprise level support?
Contact your account manager for additional information on running business and mission critical-workloads on AWS (starting at \$15,000/month). [Learn more](#)

© 2020 Amazon Internet Services Private Ltd. or its affiliates. All rights reserved.
[Privacy Policy](#) | [Terms of Use](#) | [Sign Out](#)

Security Best Practices of AWS Accounts

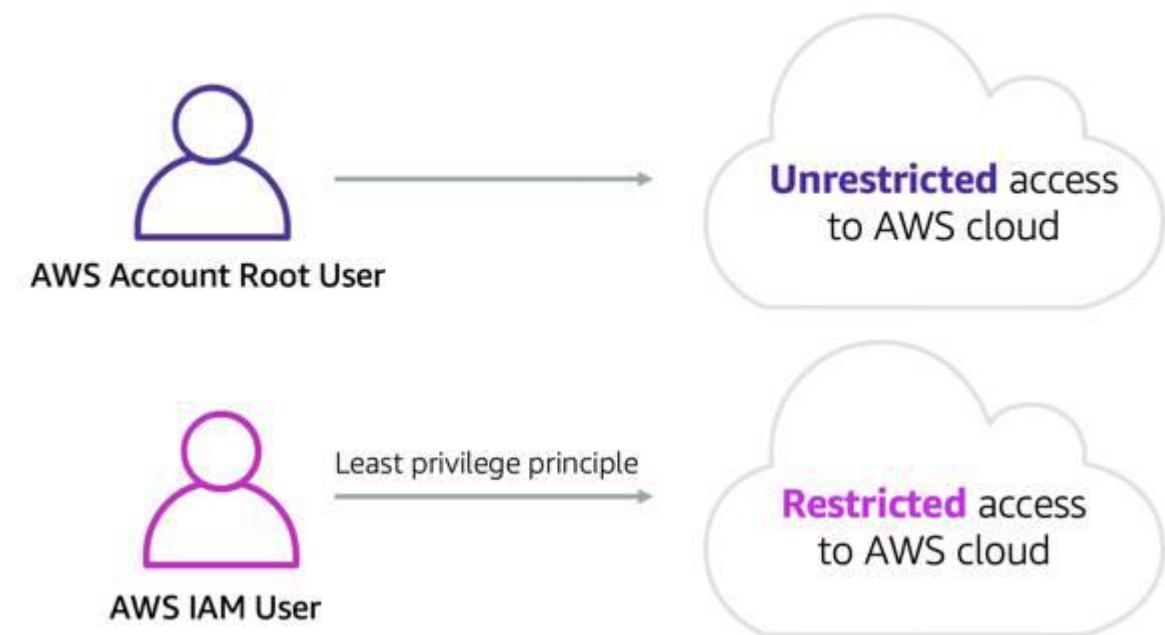
- AWS account has **2 main user type**
 - Root User Account
 - IAM User Account
- **Root user account** has full power of your AWS account and it has **unrestricted access** over to your AWS cloud account
- **IAM User Account** is **sub-users** under the root user account and **define policies** over this account and **restrict** over to your AWS cloud account
- **Security Best Practice:** After active our account, first thing we should do create "**IAM User**" under root account for our **daily usage** of AWS Console
- [Security best practices in IAM](#)



<https://trailhead.salesforce.com/en/content/learn/modules/aws-identity-and-access-management/set-iam-policies>

Create IAM User Account and Configure for Programmatic and Console Access

- Create **user-specific AWS account** under root user account that they have own login and passwords
- Define **Programmatic and Console Access**
- **Programmatic access is required** for our course, because we will use all interactions with AWS resources like **AWS Console, AWS CLI, AWS CDK and AWS SDK**.
- Follow articles below;
- [Authenticating using IAM user credentials](#)
- [Create IAM User Account and Configure for Programmatic and Console Access](#)
 - **Don't forget to allow Programmatic Access**



<https://trailhead.salesforce.com/en/content/learn/modules/aws-identity-and-access-management/set-iam-policies>

AWS Access Types - Programmatic / Management Console Access

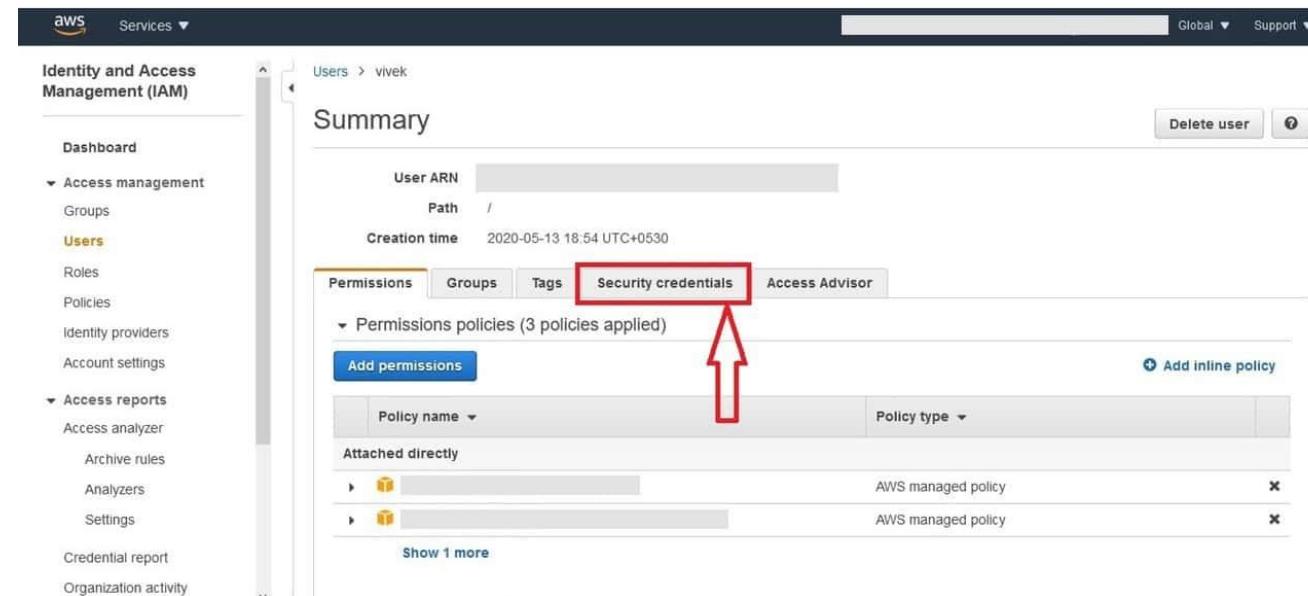
- When creating a user, we have an option about **AWS Access Types**

- **Programmatic Access**
- **AWS Management Console Access**

- **Programmatic Access**

Enables **access key ID** and **secret access key** for the **AWS API, CLI, SDK**, and other development tools.

- Make sure that you have created your access keys for programmatic access.



Access keys for CLI, SDK, & API access

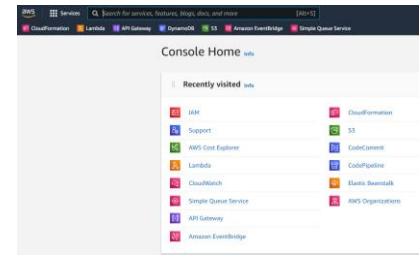
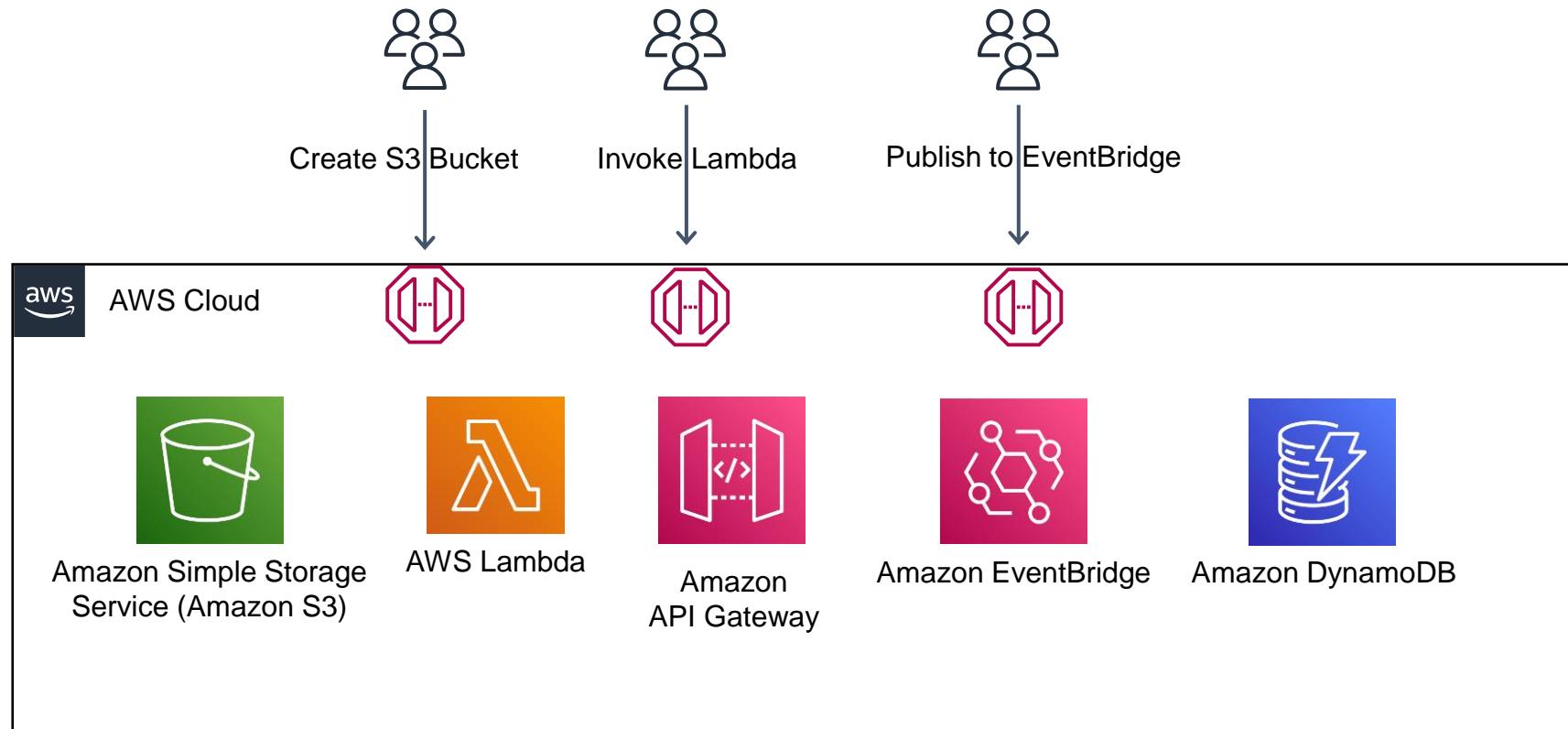
Use access keys to make programmatic calls to AWS from the AWS Command Line Interface (AWS CLI), Tools for Windows PowerShell, the AWS SDKs, or direct AWS API calls. If you lose or forget your secret key, you cannot retrieve it. Instead, create a new access key and make the old key inactive. Learn more

[Create access key](#)

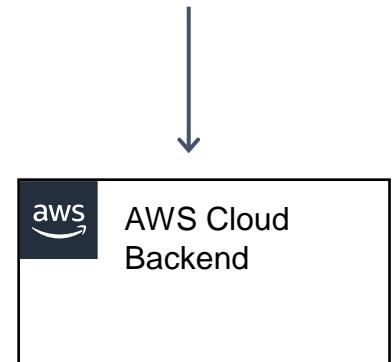
Access key ID	Status	Created	Last used	Actions
AKIAIM5S2HZOCSL6KA3Q	Active	2019-01-04 16:26 PST	N/A	Make inactive Delete

AWS Access with APIs

- AWS expose APIs that we can invoke to **create** and **manage** aws services



AWS Management Console
FrontEnd



Invoke AWS APIs with Different Ways

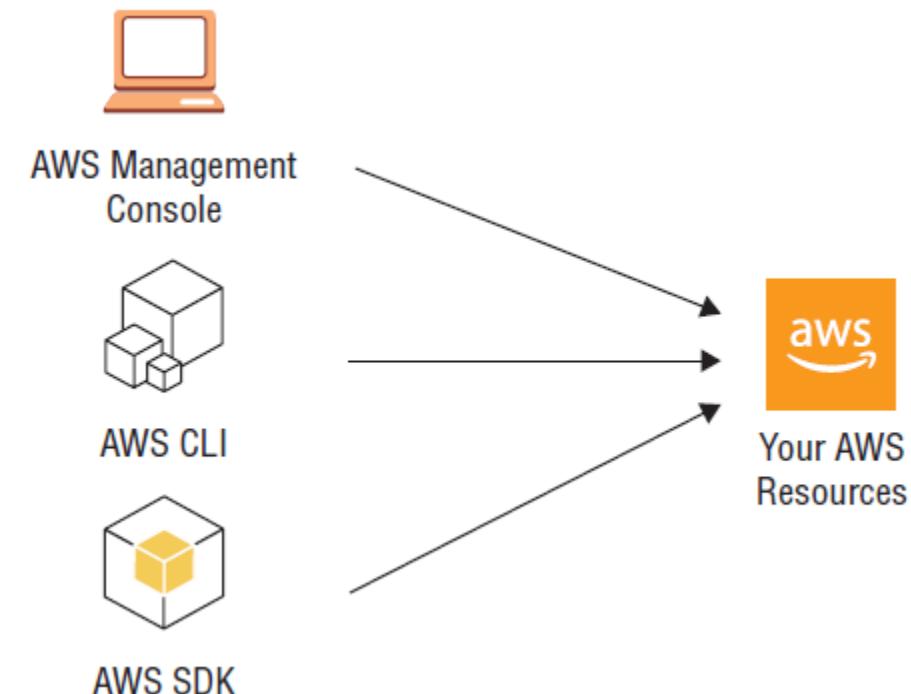
- **AWS Management Console Access**

You can think as a web application allows us to manage AWS resources for particular AWS accounts.

- **Programmatic Access**

Gives us to manage AWS resources from our development environments and manage by writing codes.

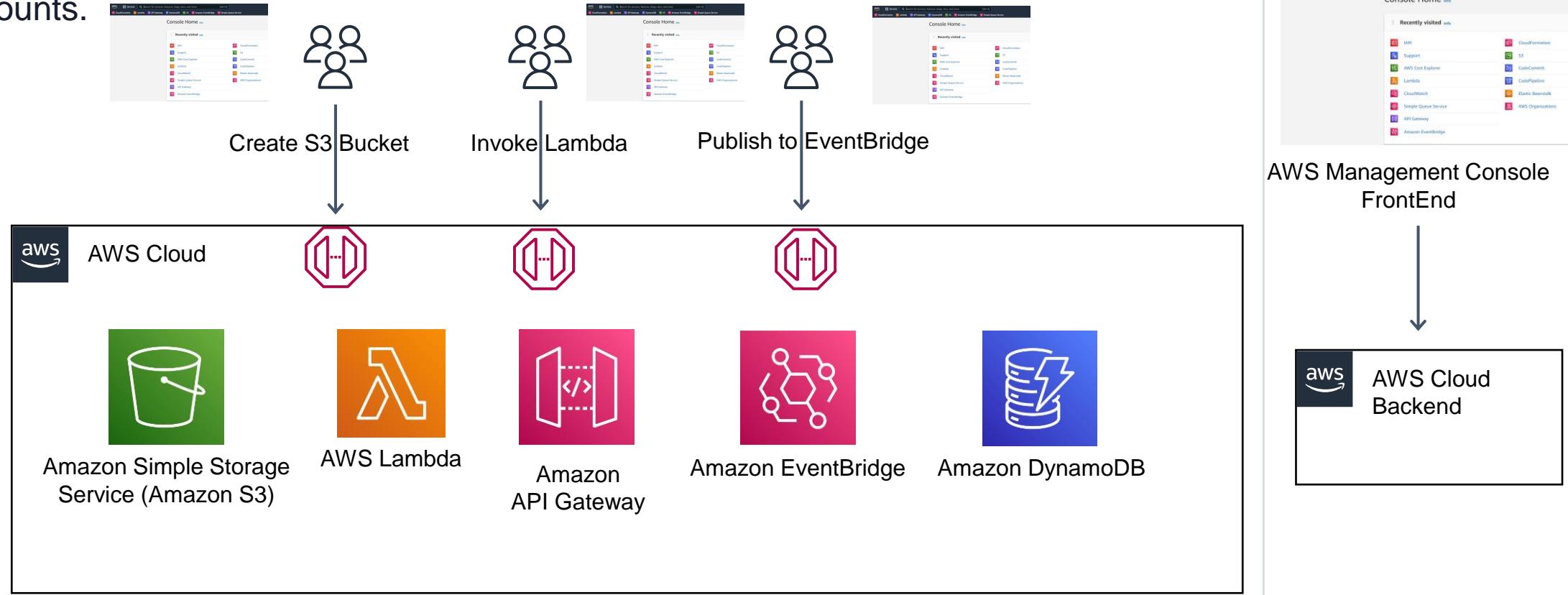
- AWS CLI
- AWS SDK
- AWS Cloud Formation - IaC
 - AWS SAM
 - AWS CDK



Invoke AWS APIs with Different Ways; AWS Management Console

▪ AWS Management Console Access

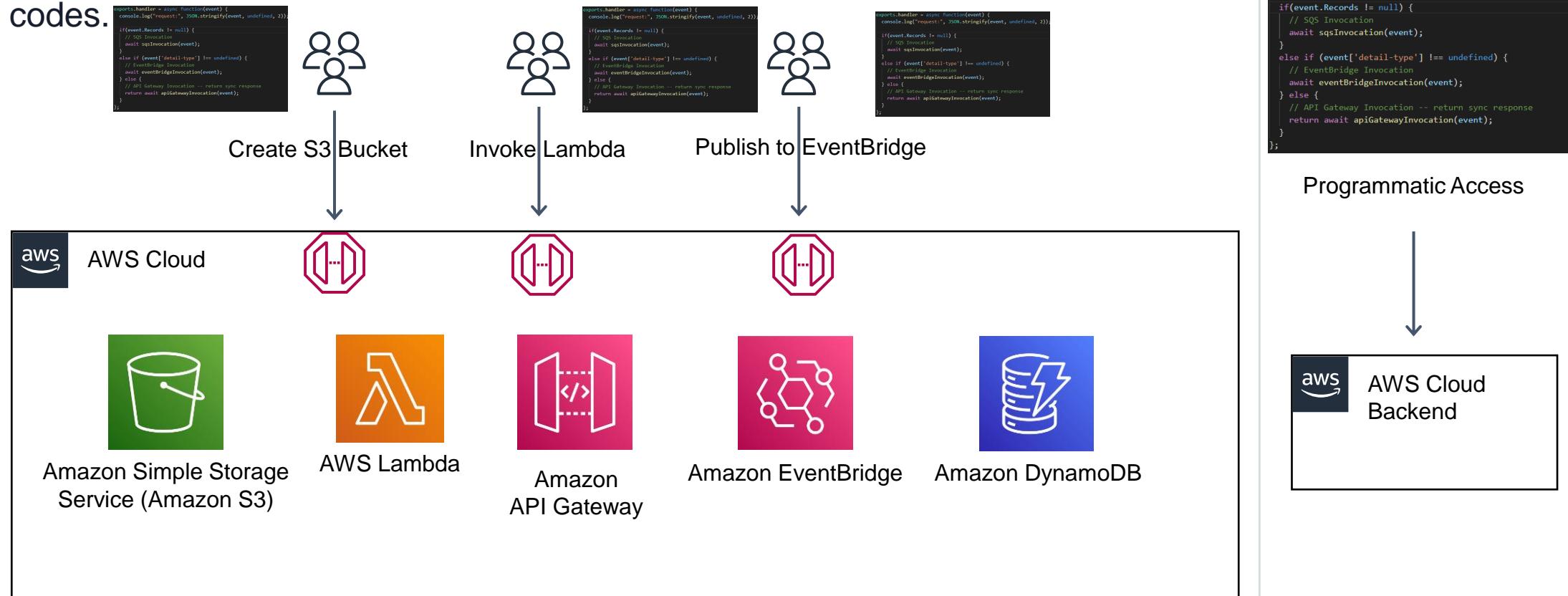
You can think as a web application allows us to manage AWS resources for particular AWS accounts.



Invoke AWS APIs with Different Ways; Programmatic Access

▪ Programmatic Access

Gives us to manage AWS resources from our development environments and manage by writing codes.



Programmatic Access

- **Programmatic Access**

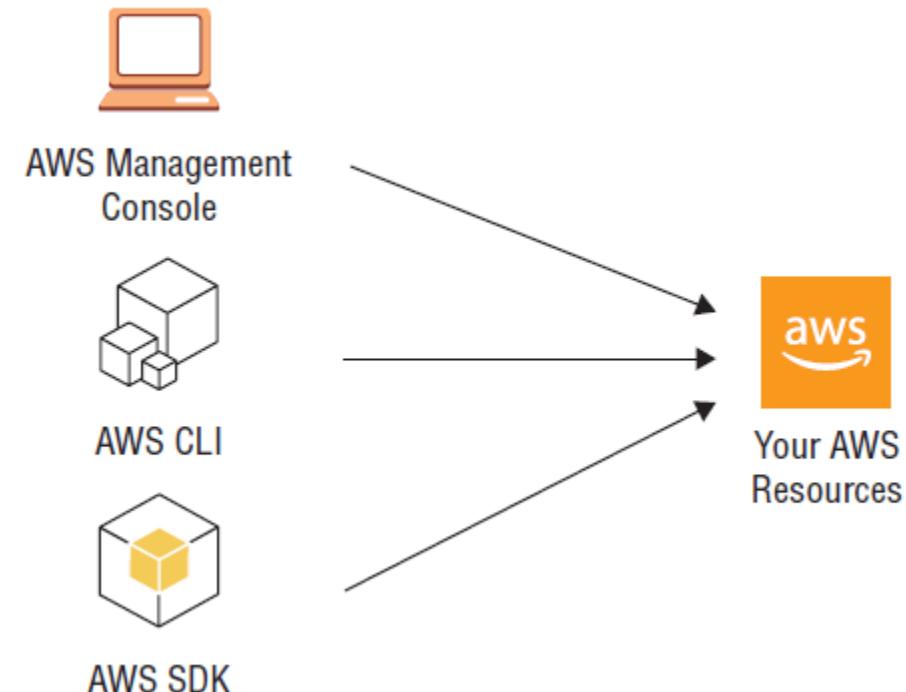
Gives us to manage AWS resources from our development environments and manage by writing codes.

- AWS CLI
- AWS SDK
- AWS Cloud Formation - IaC
 - AWS SAM
 - AWS CDK

- **AWS Command Line Interface (CLI)**

Unified tool to manage your AWS services.

- Control multiple AWS services from the command line and automate them through scripts.
- `$ aws ec2 describe-instances`
- `$ aws ec2 start-instances --instance-ids i-1348636c`



Download and Configure AWS CLI

- **Download and install AWS CLI**

<https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html>

- **Check AWS CLI**

```
aws –version
```

- **Configure AWS CLI**

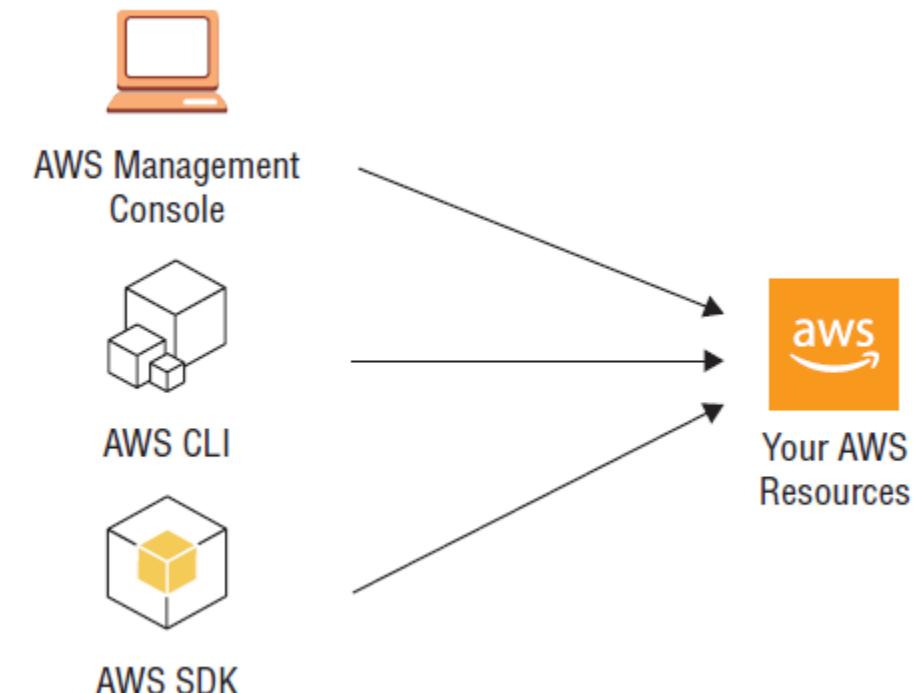
[Authenticating using IAM user credentials](#)

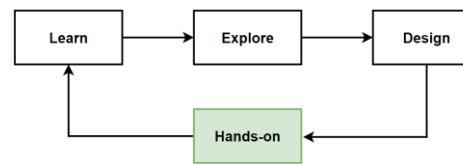
- **aws configure**

- Access Key ID
- Secret Access Key
- AWS Region
- Output format

- [aws configure commands](#)

- **aws configure list**

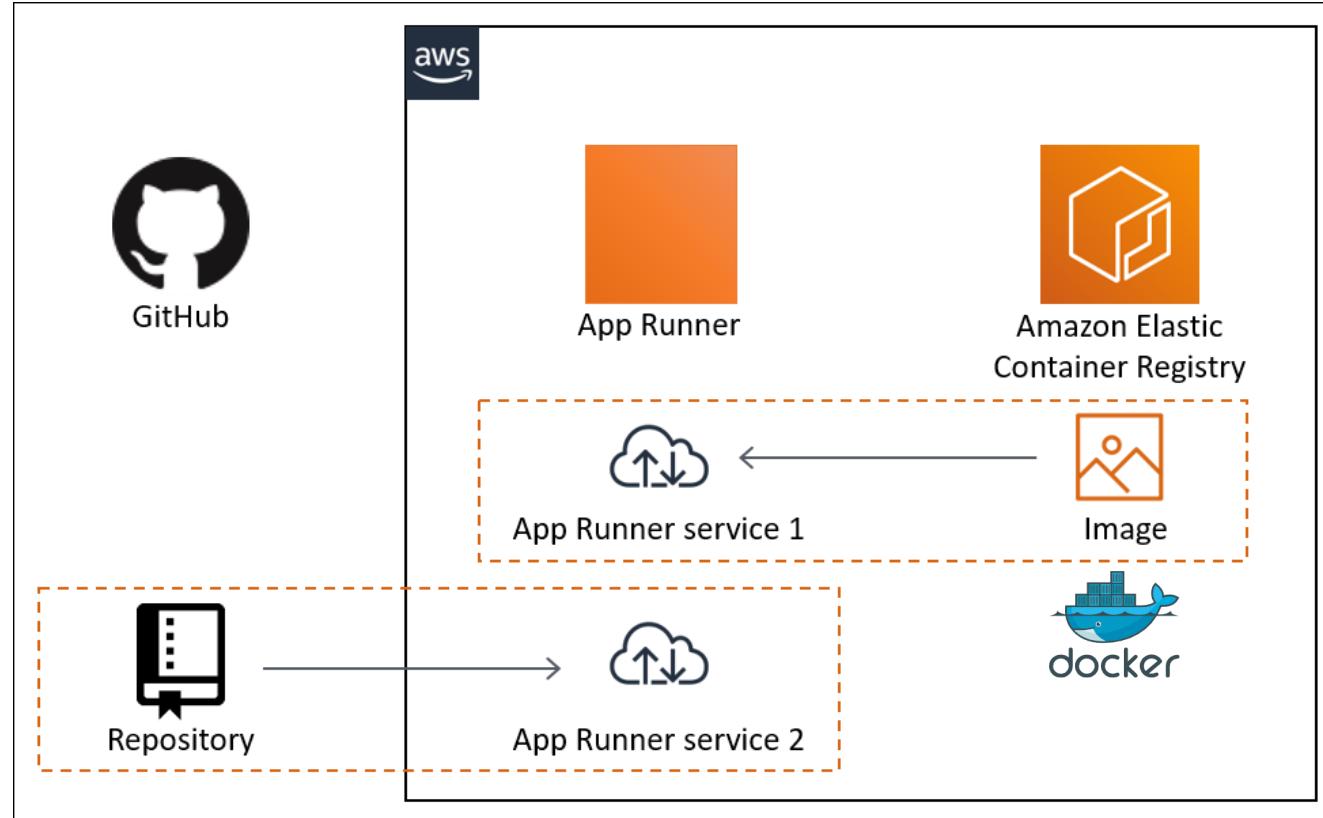




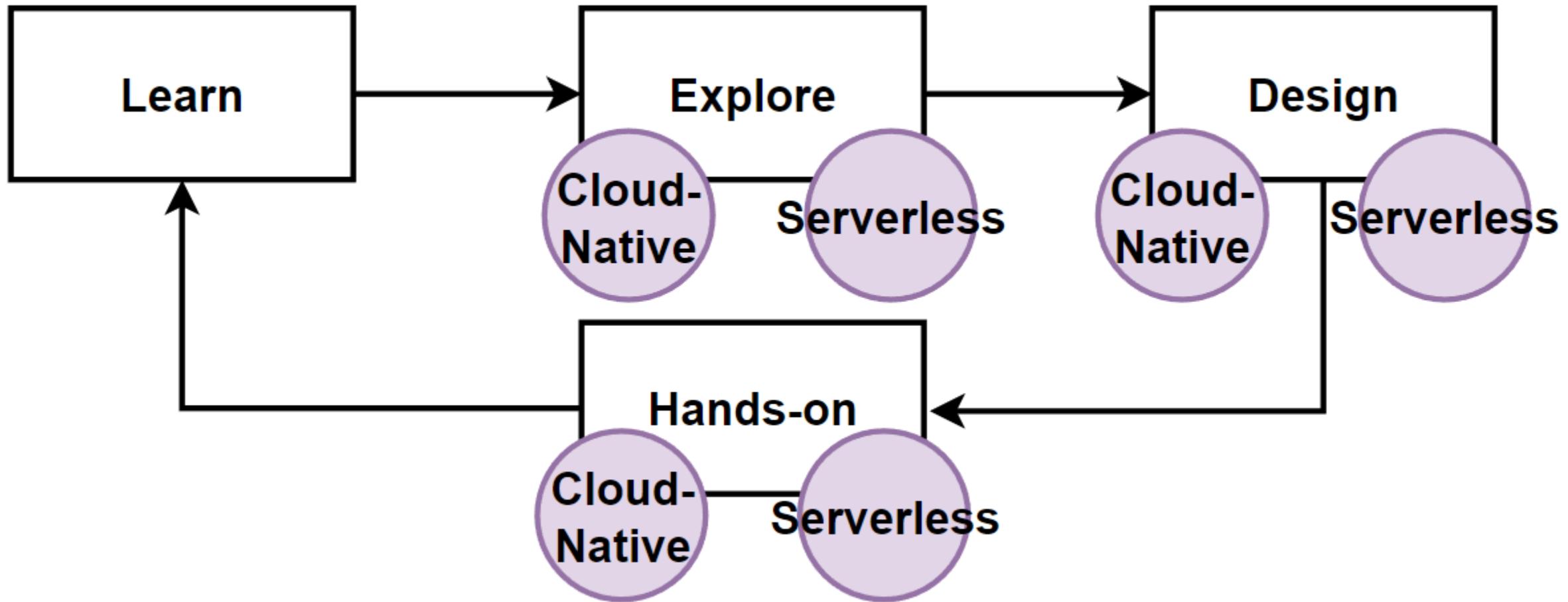
Hands-on: Deploy ProductService Container to AWS Apprunner

Todo List:

- Push your Docker container image to Amazon Elastic Container Registry (ECR)
- Deploy to AWS App Runner that pull image from ECR

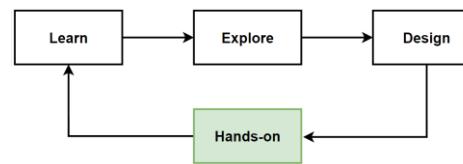


Way of Learning – Cloud-Native & Serverless Cloud Managed



Examples of CN vs Serverless Cloud Tools:

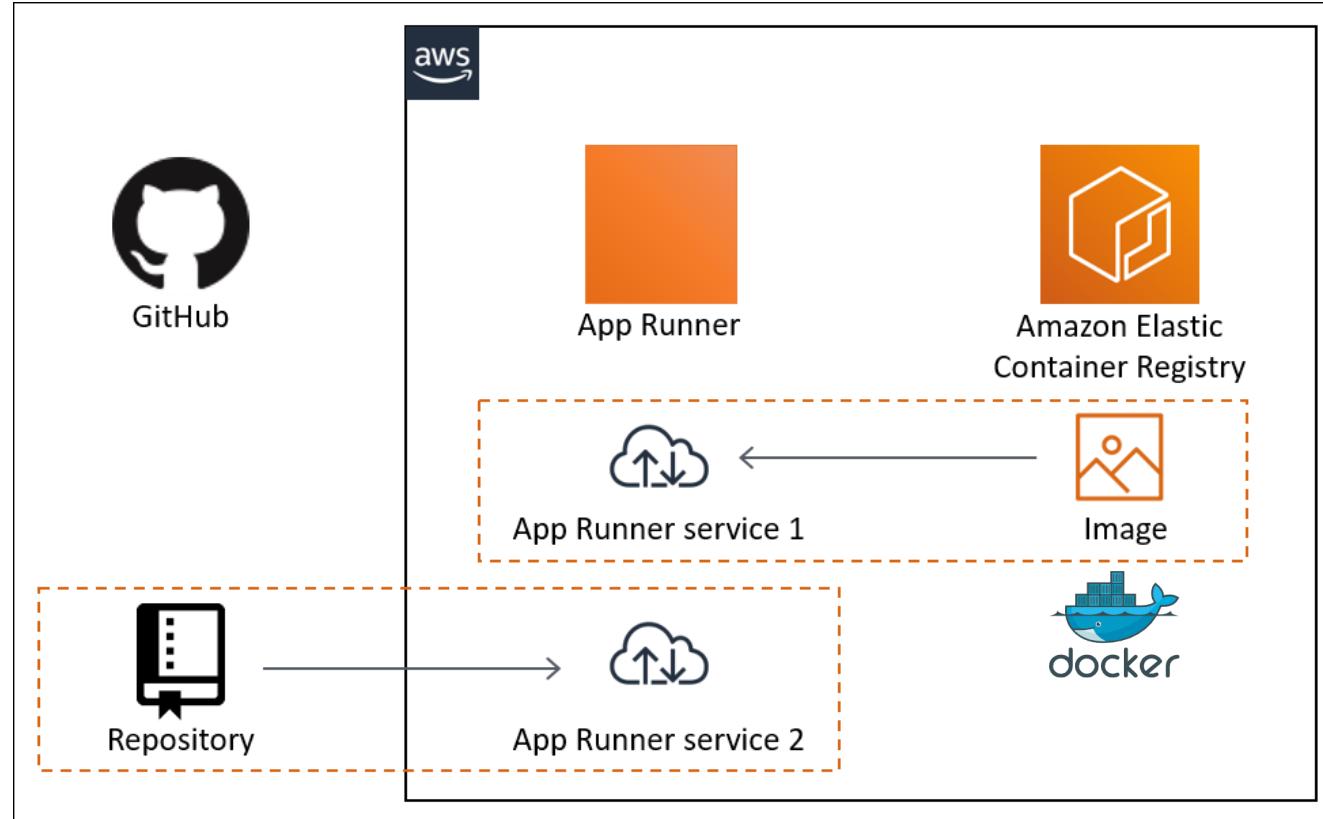
- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

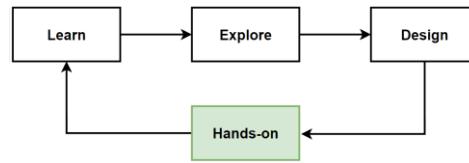


Hands-on: Deploy ProductService Container to AWS Apprunner

Todo List:

- Push your Docker container image to Amazon Elastic Container Registry (ECR)
- Deploy to AWS App Runner that pull image from ECR

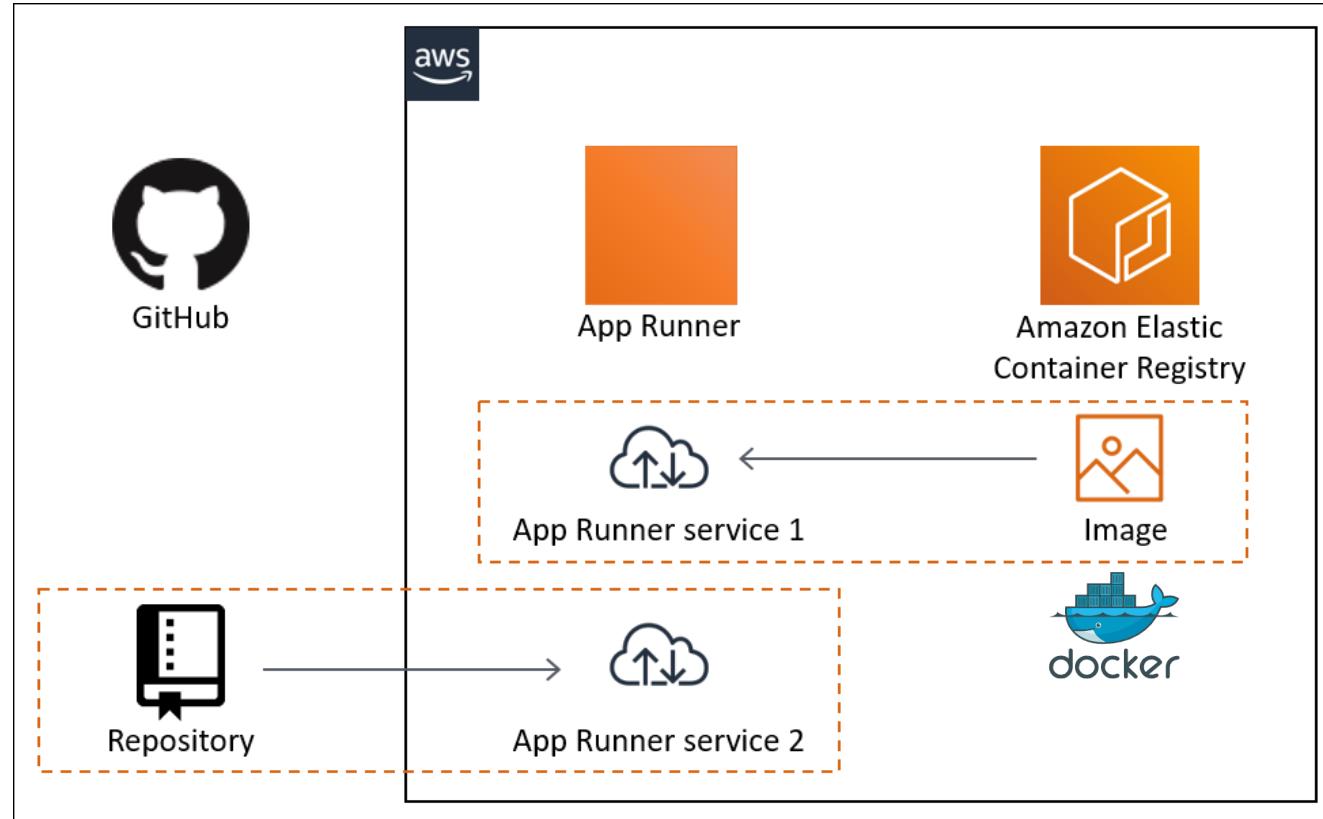


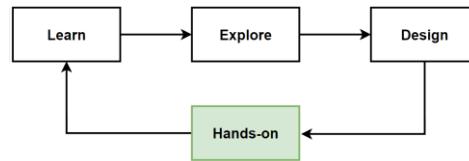


Pushing Docker Image to Amazon Elastic Container Registry (ECR)

Todo List:

- Step 1: Create an Amazon ECR repository
- Step 2: Authenticate Docker to your Amazon ECR registry
- Step 3: Build Docker image
- Step 4: Tag Docker image
- Step 5: Push your Docker image to Amazon ECR

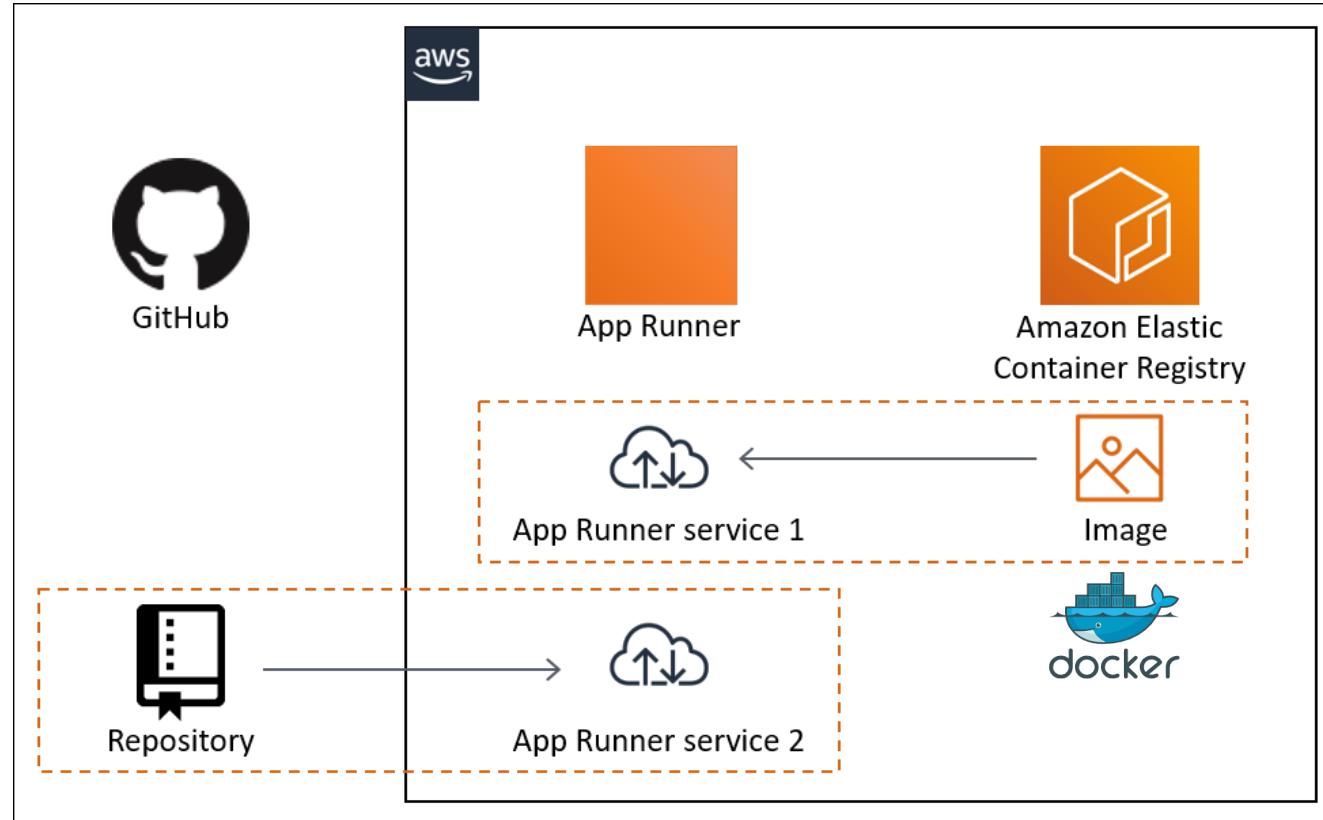


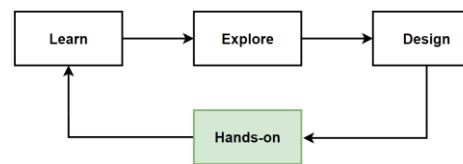


Hands-on: Deploy ProductService Container to AWS Apprunner

Todo List:

- Push your Docker container image to Amazon Elastic Container Registry (ECR)
- Deploy to AWS App Runner that pull image from ECR

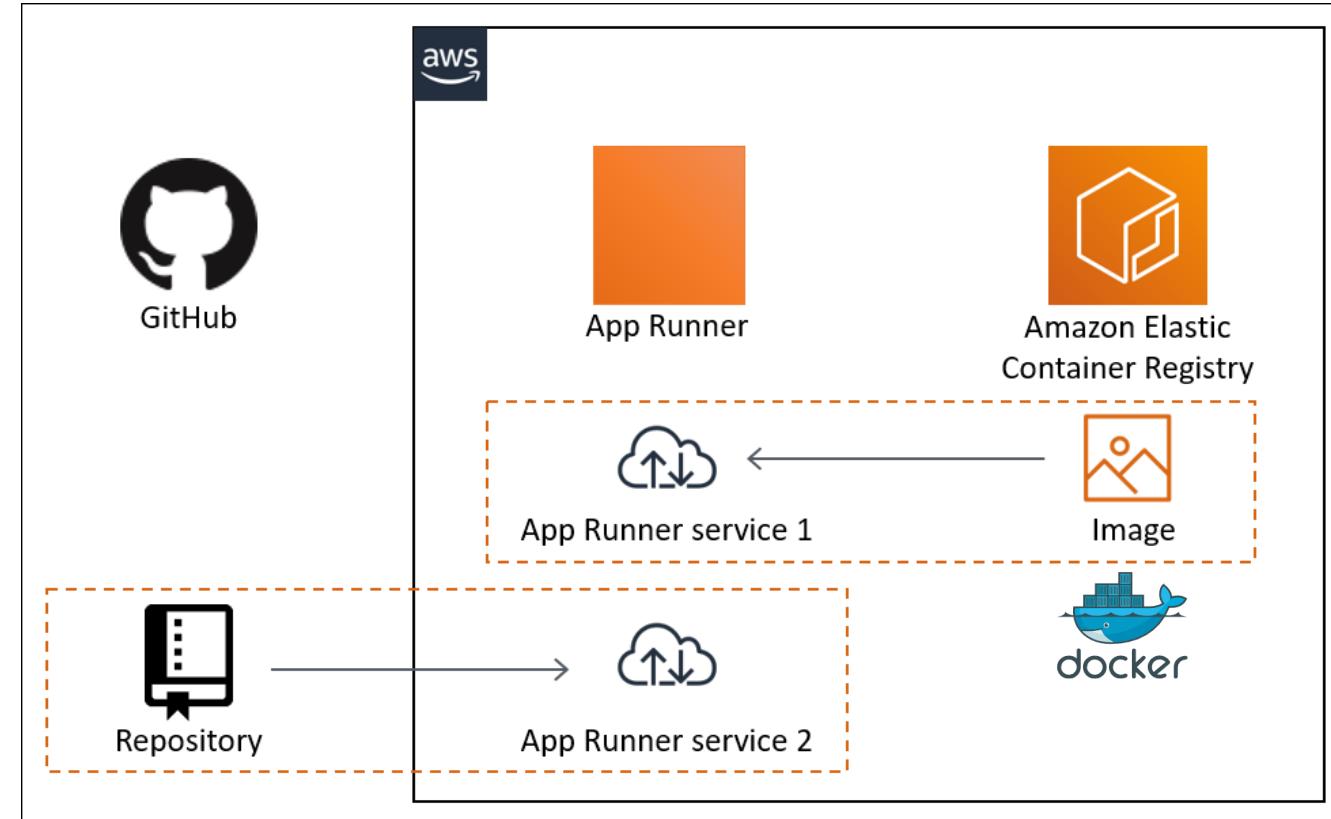


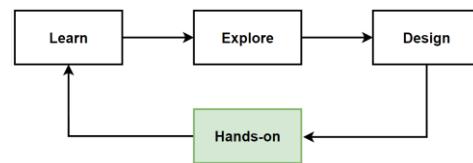


Deploy to AWS App Runner that pull image from ECR

Todo List:

- Step 1: Create a new service in AWS App Runner
- Step 2: Configure the container image
- Step 3: Configure the deployment settings
- Step 4: Review your configuration and create the service
- Step 5: Deploy

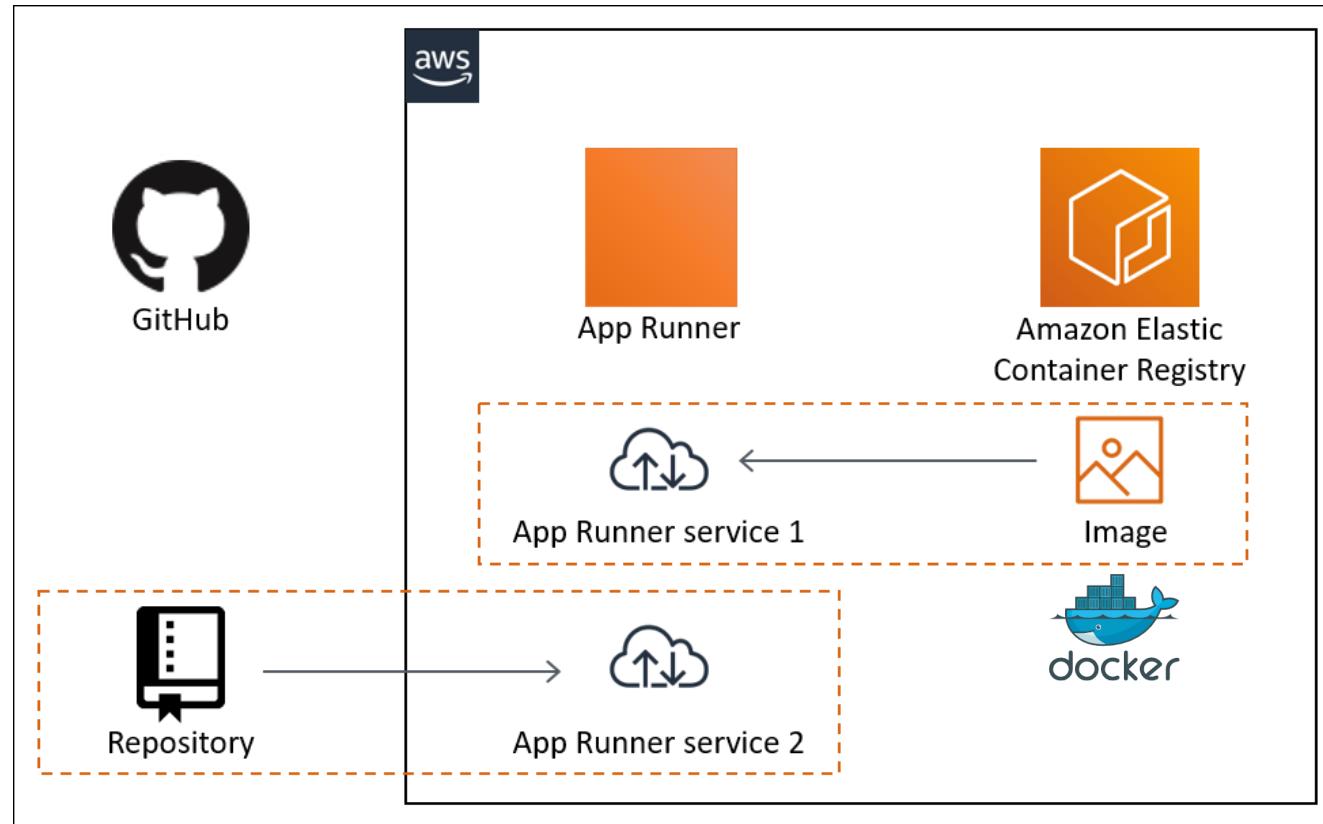




Clear AWS Resources

Remove:

- AWS App Runner
- AWS ECR and delete images
- Role – AppRunnerECRAccessRole



Cloud-Native Pillar3: Container Orchestrators

What are Container Orchestrators ?

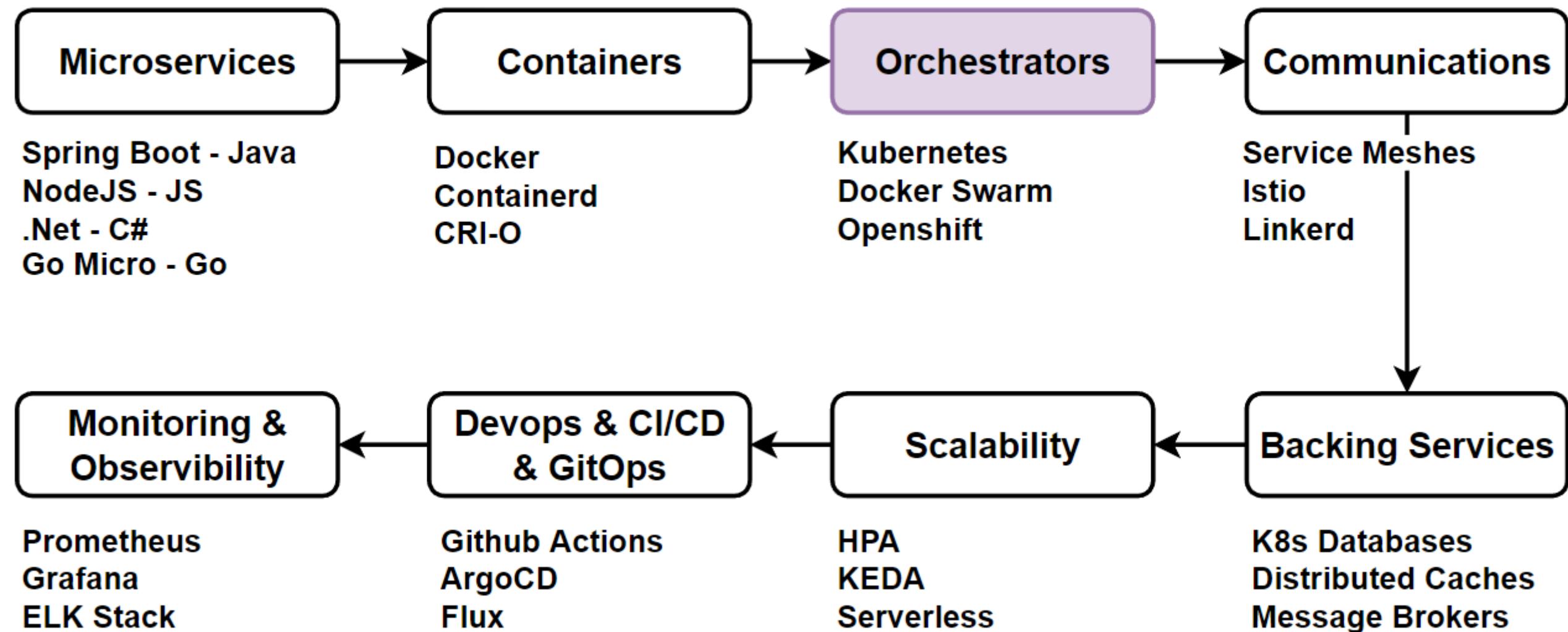
Why we should use Container Orchestrators for Cloud-Native Applications ?

Benefits and Challenges of Container Orchestrators

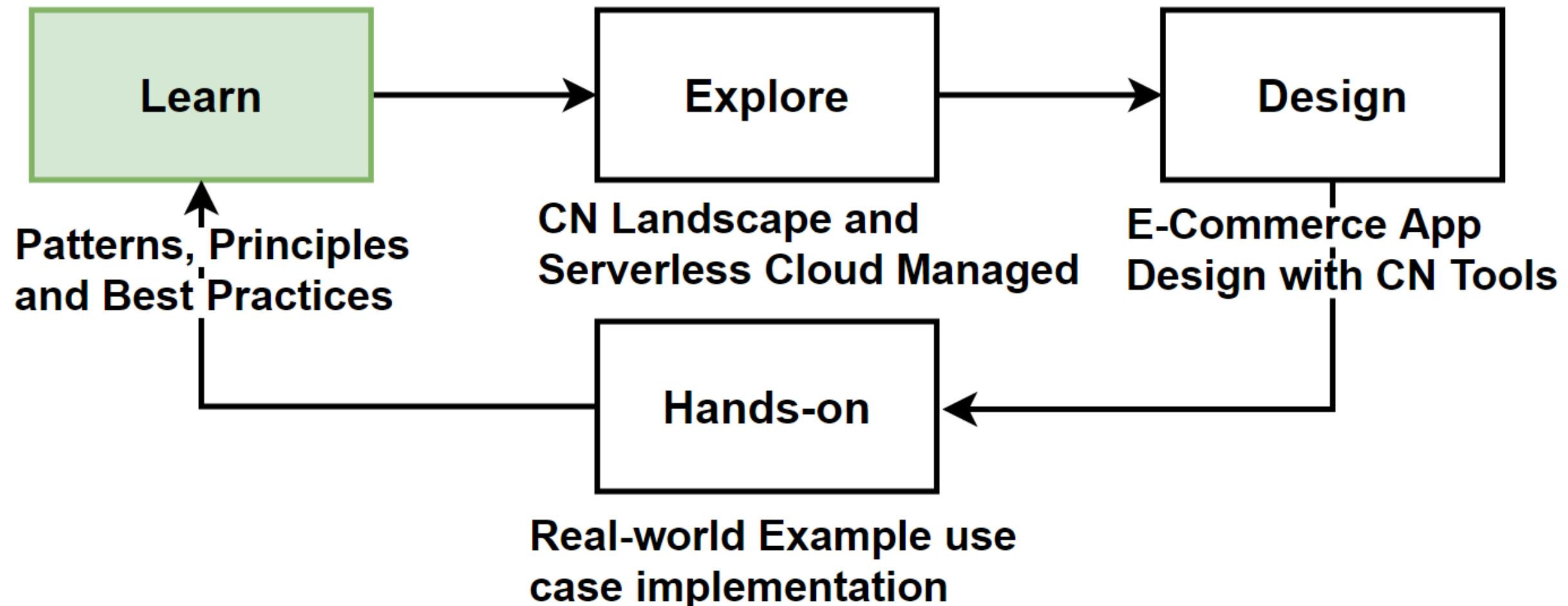
Design our E-Commerce application with Container Orchestrators

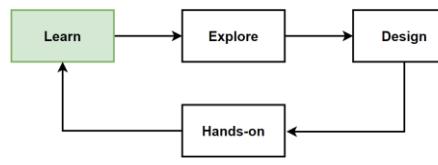
Implement Hands-on Development of Container Orchestrators

Cloud-Native Pillars Map – The Course Section Map



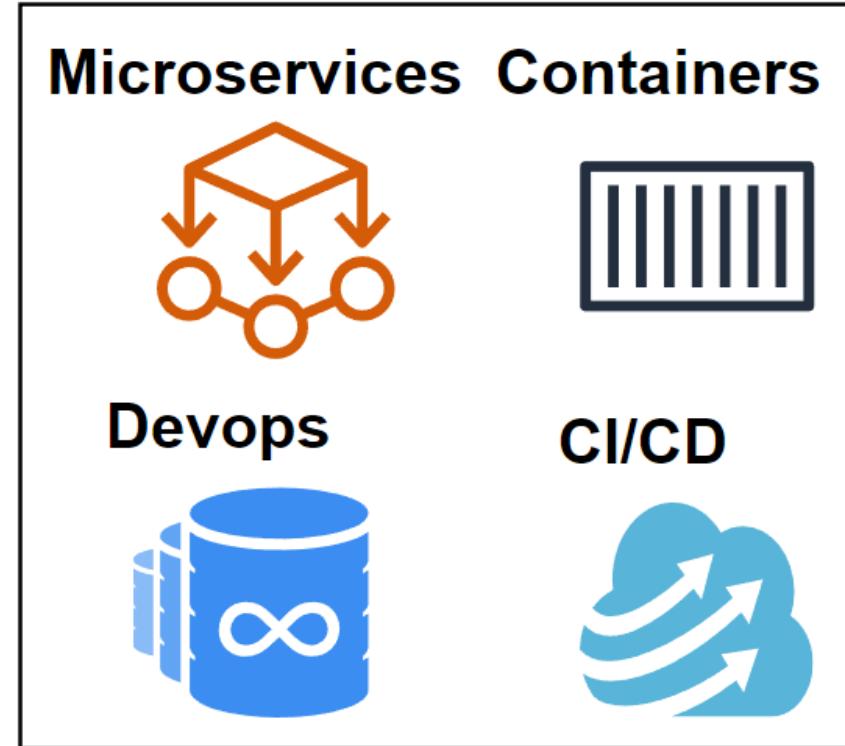
Way of Learning – The Course Flow

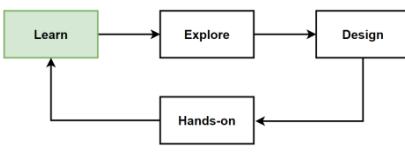




Learn: The Third Pillar – Container Orchestrators

- Microservices Deployments with Containers and Orchestrators
- What are Container Orchestrators ?
- Benefits and Challenges of Container Orchestrators
- When to Use Container Orchestrators - Best Practices
- What is Kubernetes and why we use them for microservices deployments
- Kubernetes Architecture
- Kubernetes Components
- Helm Charts with Kubernetes for Microservices Deployments
- Design our E-Commerce application with Container Orchestrators
- Implement Hands-on Development of Container Orchestrators





Where «Orchestrators» in 12-Factor App and CN Trial Map

Twelve-Factor App

- Codebase
- Dependencies
- Config
- Backing Services
- Build, release and run
- Processes
- Port Binding
- Concurrency
- Disposability
- Development/Prod Parity
- Logs
- Admin Processes



CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape <https://github.com/cncf/landscape> has a growing number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator <https://www.cncf.io/training>

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider <http://cncf.io/kcsp>

C. Join CNCF's End User Community

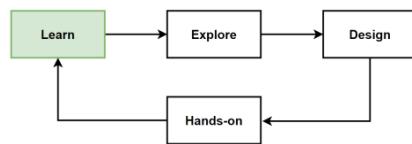
For companies that don't offer cloud native services externally <http://cncf.io/enduser>

WHAT IS CLOUD NATIVE?

- **Operability:** Expose control of application/system lifecycle.
- **Observability:** Provide meaningful signals for observing state, health, and performance.
- **Elasticity:** Grow and shrink to fit in available resources and to meet fluctuating demand.
- **Resilience:** Fast automatic recovery from failures.
- **Agility:** Fast deployment, iteration, and reconfiguration.

www.cncf.io
info@cncf.io





12-Factor App – Container Orchestrators

Codebase

- Orchestrators work with container images, which ensure a consistent codebase across environments.

Backing Services

- Manage the connections and configuration of backing services like databases, caches, or message brokers.

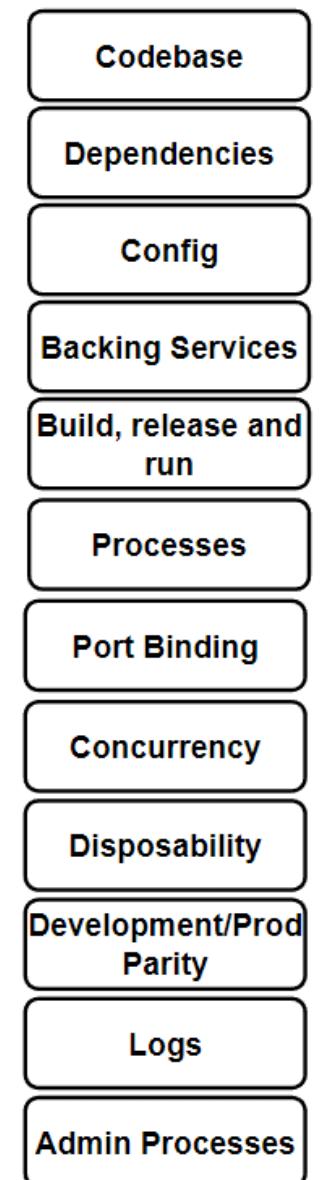
Build, release, run

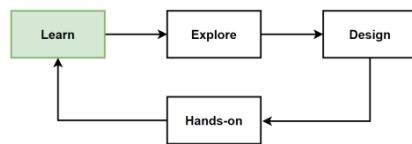
- Automate the deployment of new releases, allowing a clear separation between the build, release, and run stages.

Processes

- Manage container instances as stateless processes, enabling horizontal scaling and fault tolerance.

Twelve-Factor App





12-Factor App – Container Orchestrators

Concurrency

- Scale applications by managing multiple instances of the same container, allowing the application to handle concurrency efficiently.

Disposability

- Support the fast startup and graceful shutdown of container instances, ensuring disposability and fault tolerance.

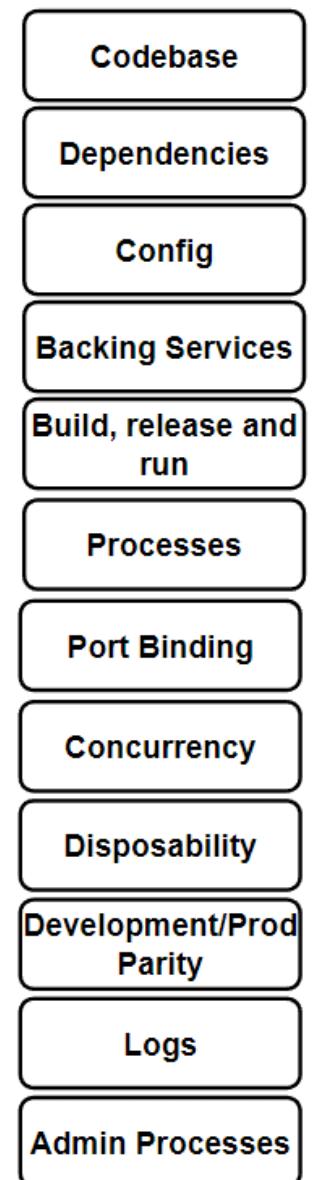
Dev/prod parity

- Enable consistent environments across development, staging, and production, reducing discrepancies and ensuring parity.

Admin processes

- Run one-off admin tasks or maintenance jobs as separate container instances, maintaining the separation of concerns.

Twelve-Factor App



Cloud-native Trial Map – Orchestrators

Containerization

- Work with container images to manage and deploy applications in a consistent and portable manner.

CI/CD

- Integrate with CI/CD pipelines to automate the deployment, scaling, and management of applications, ensuring a streamlined and efficient delivery process.

Orchestration & Application Definition

- Kubernetes, Docker Swarm, or Amazon ECS, which automate deployment, scaling, and management of containerized applications.

Observability & Analysis

- Expose metrics and logs from applications, making it easier to monitor, analyze, and troubleshoot issues in a cloud-native environment.



<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>

Cloud-native Trial Map – Orchestrators -2

Service Mesh

- Integrated with container orchestrators to provide enhanced networking, security, and observability features for microservices communication.

Networking & Load Balancing

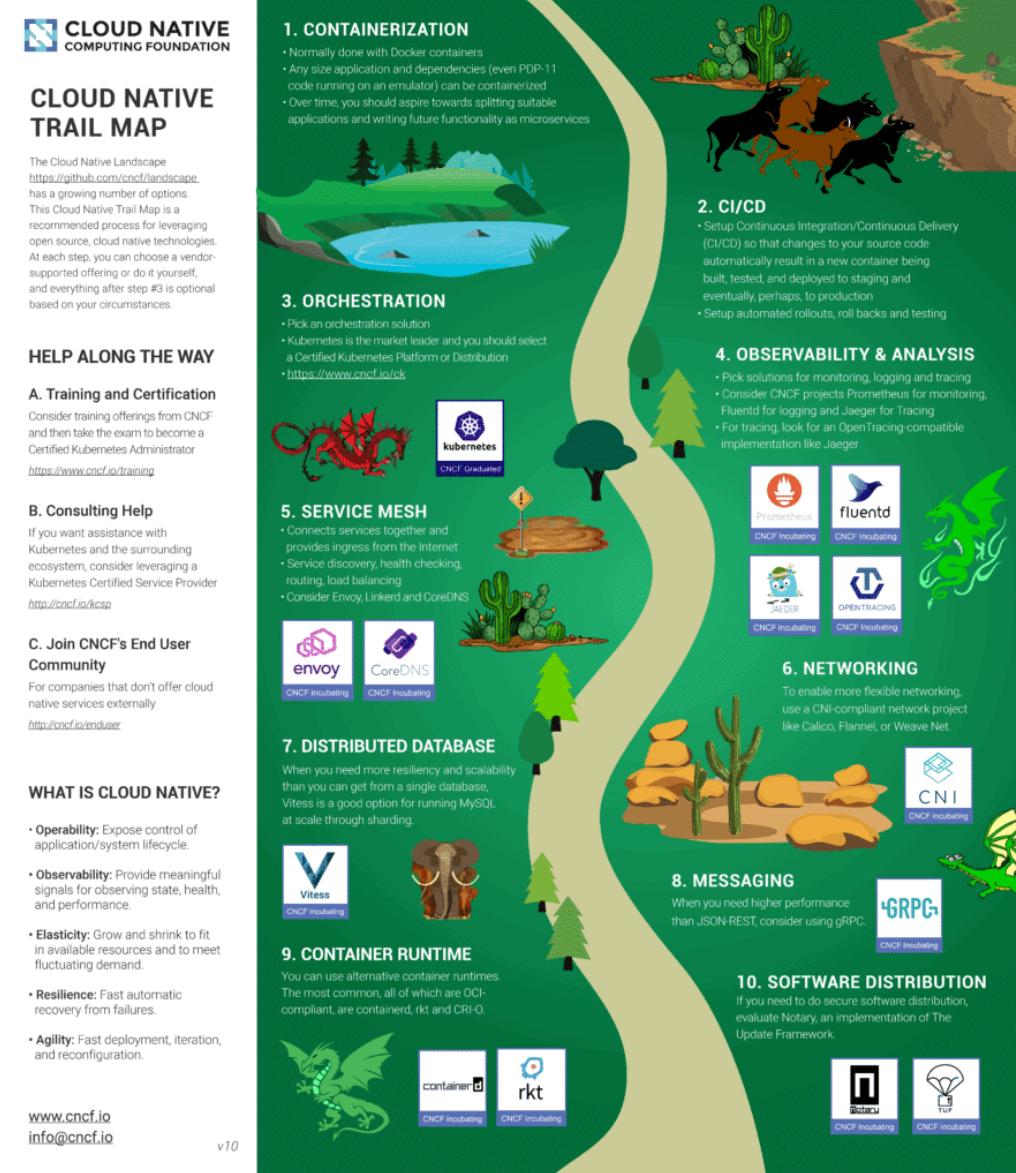
- Manage the networking aspects of applications, including service discovery, load balancing, and routing.

Scaling & High Availability

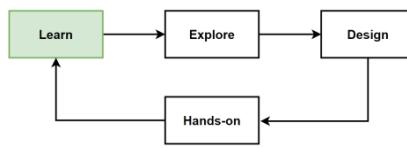
- Automatically scale applications based on demand and ensure high availability by managing container instances and distributing workloads.

Configuration & Secret Management

- Manage application configuration and secrets, allowing for secure and dynamic configuration updates.

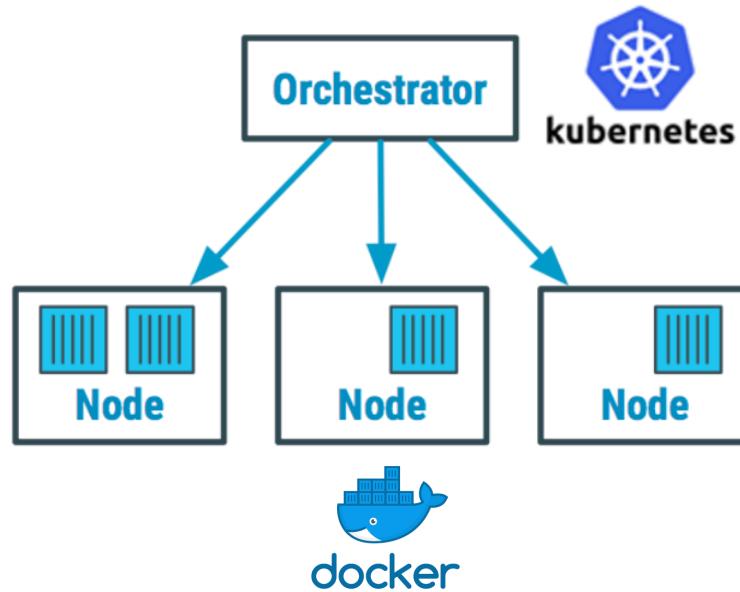


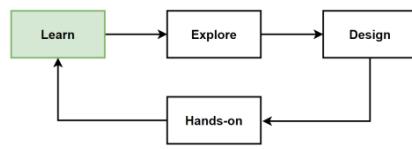
<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>



Why need Orchestrator for Containers ?

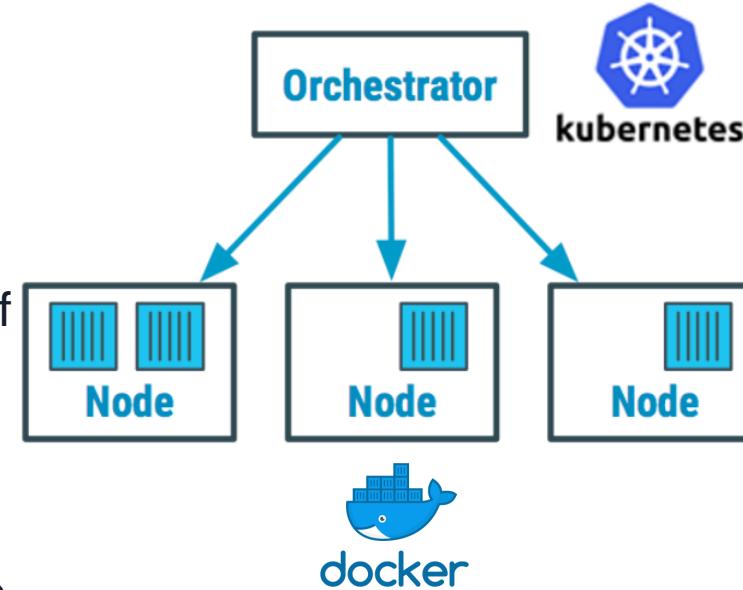
- Why we **need to orchestrate our containers** ?
- Think to developed and containerize our microservices and ready for shipping and deployment. Why we **need an orchestrator** for our **containers** ?
- **Deployment requirements of microservices**, think that have **hundreds of microservices**, we should ask;
 - How container instances be provisioned into cluster of multiple machines ?
 - After deployment, how will containers discover and communicate with each other ?
 - How can containers scale in or out on-demand and peek traffic ?
 - How do you monitor the health of each container ?
 - How do you protect containers against hardware and software failures ?
 - How do upgrade containers for a live application with zero downtime ?
- All these questions handled by **Container Orchestrators** that automate all these concerns.
- These task **can not be managed by manually** administrated for thousands of independently deployed containers.
- **Containerized services require automated management.**

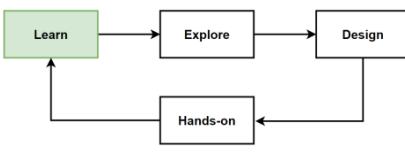




What is Container Orchestrator ?

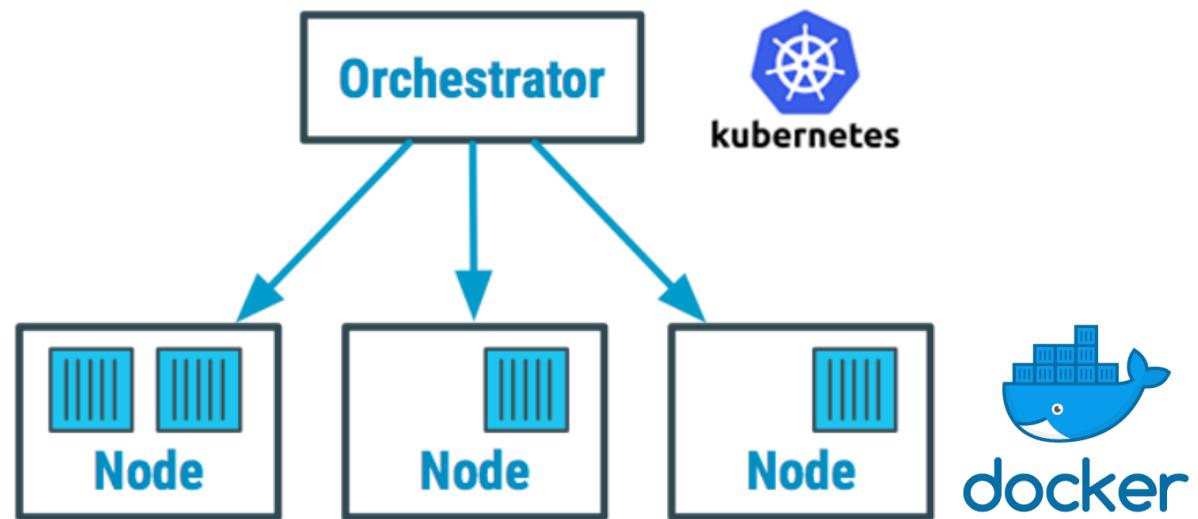
- Container orchestrators are **manage** and **automate** the **deployment**, **scaling**, and management of containerized applications.
- Enable to **run** and **manage** **microservices-based applications** in an efficient and scalable way by **abstracting the underlying infrastructure** and handling tasks such as **resource allocation**, **load balancing**, and **monitoring**.
- Microservices architectures, **containers need to orchestrate** to manage lots of container in your application cluster.
- **Orchestrators automates** the **deployment**, **scaling**, and **operational concerns** of containerized workloads across clusters.
- Container orchestrator **automates the deployment and management** of these microservices, making it **easier to scale and update application** as needed.
- There are several popular **container orchestrators** available, including **Kubernetes**, **Docker Swarm**, and **Apache Mesos**.

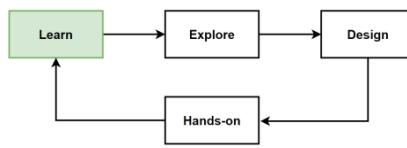




Application Containerization and Orchestration

1. Package each of **microservices** into a **container image** and push it to a **container registry**.
2. Then use the **container orchestrator** to **deploy** the **microservices** to a cluster of nodes.
3. **Orchestrator** will handle tasks such as **scheduling containers** onto nodes, **monitoring** the health of the containers, and providing **self-healing** capabilities if a container fails.





Benefits of Using Container Orchestration

- **Automation**

orchestrator can automate the deployment and management of your microservices.

- **Scalability**

orchestrator can automatically scale your microservices up or down as needed meet changing demand.

- **High availability**

orchestrator can provide self-healing capabilities to ensure that your microservices remain available even if individual containers fail.

- **Improved resource utilization**

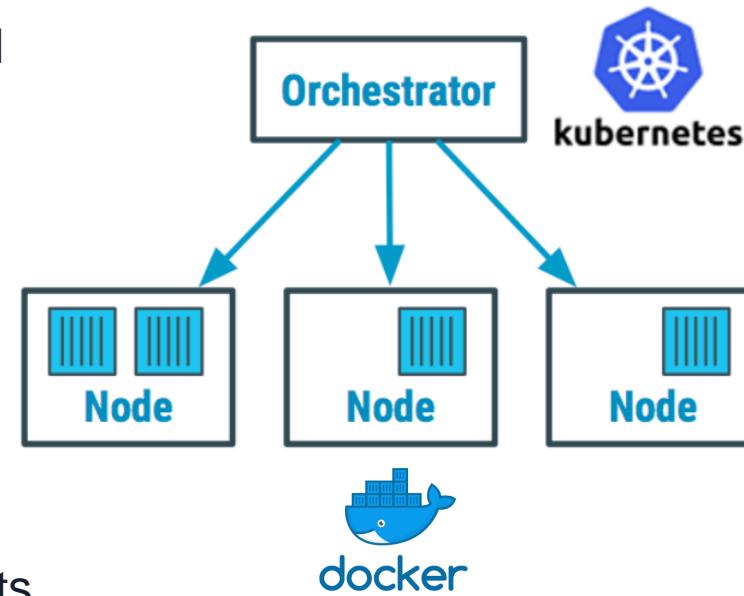
Orchestrator can optimize the use of resources in your cluster.

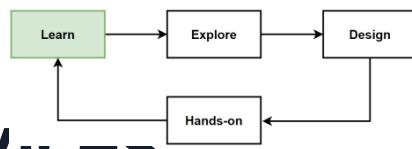
- **Portability**

Containerized applications can be easily moved between different environments and platforms.

- **Security**

orchestrators provide security features such as role-based access control and network segmentation.



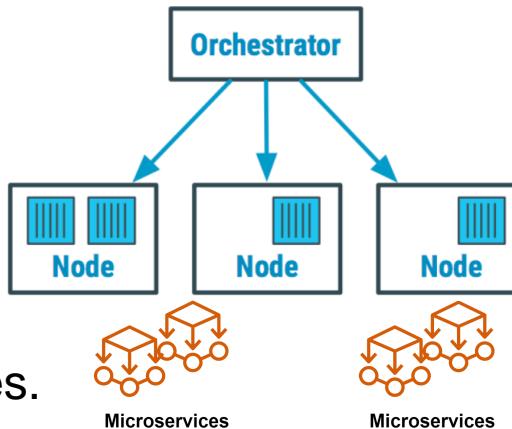


Container Orchestrators Usage for Cloud-Native Microservices

- Simplify the complexities of deploying, scaling, and maintaining containerized apps.
- Provide resource utilization for managing microservices and ensure their continuous operation.

Continuous Deployment and Rollback

- Automate the deployment and management of containerized applications.
- Automation facilitates the development, testing, and deployment of microservices.
- Updating or rolling back versions of microservices without downtime, enabling continuous deployment.

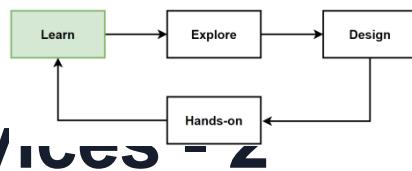


Management of Microservices

- Microservices involves developing an app as a collection of small, loosely coupled services.
- Orchestrators provide the necessary tools to manage these numerous containers efficiently.

Scalability

- Microservices scaled independently based on demand.
- Orchestrators like Kubernetes enable automatic scaling of services, leading to efficient resource usage and improved application performance.



Container Orchestrators Usage for Cloud-Native Microservices

High Availability and Resiliency

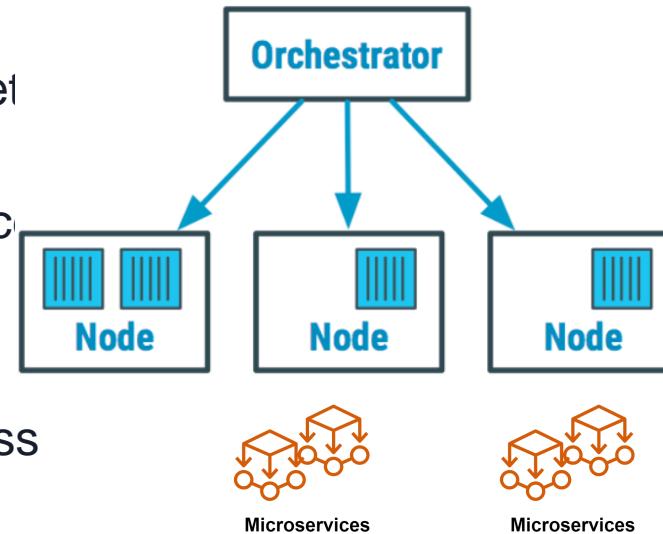
- Orchestrators monitor container health, restart failed containers, and reschedule them on available nodes.
- This self-healing capability significantly improves the reliability and uptime of applications.

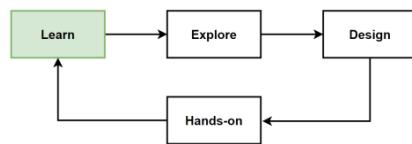
Service Discovery and Load Balancing

- Orchestrators provide service discovery mechanisms, facilitating communication between microservices.
- They offer load balancing to distribute network traffic among a set of service instances, improving response times and system resilience.

Networking and Security

- Orchestrators manage the networking aspects of microservices, including IP address allocation, routing, and network isolation.
- They enforce security policies, manage access control, and ensure compliance across the application stack.

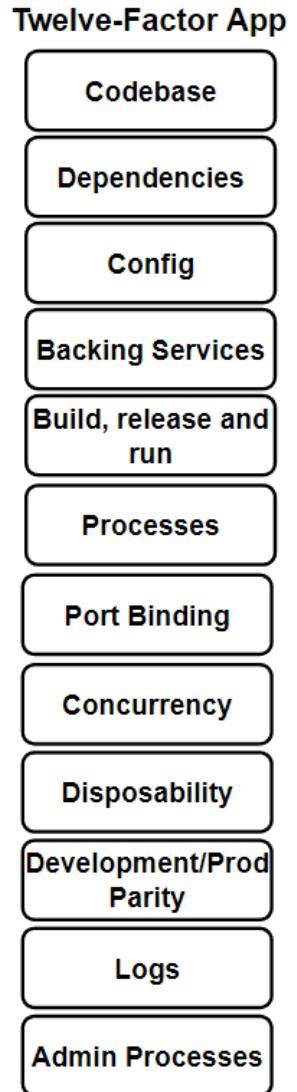
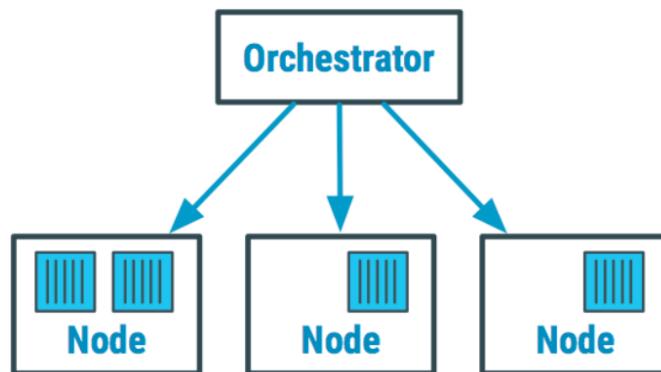


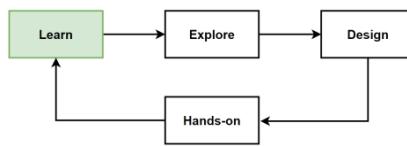


Orchestrators and Twelve-Factor Application Principles

Embracing Twelve-Factor Application Principles

- Orchestrators embrace the disposability and concurrency principles from the Twelve-Factor Application methodology.
- **Factor #8:** Services scale out across a large number of small identical processes (copies) as opposed to scaling-up a single large instance.
- **Factor #9:** Service instances should be disposable, favoring fast startups to increase scalability opportunities and graceful shutdowns to leave the system in a correct state.





Best Practices for Using Container Orchestrators

- **Declarative Approach**

Define app desired state in configuration files, to ensure consistency, simplify management, and enable version control.

- **Leverage Autoscaling**

Utilize orchestrator's autoscaling capabilities to automatically handle demand fluctuations, optimizing performance and resource efficiency.

- **Rolling Updates and Rollbacks**

Minimize downtime with rolling updates during microservices upgrades.

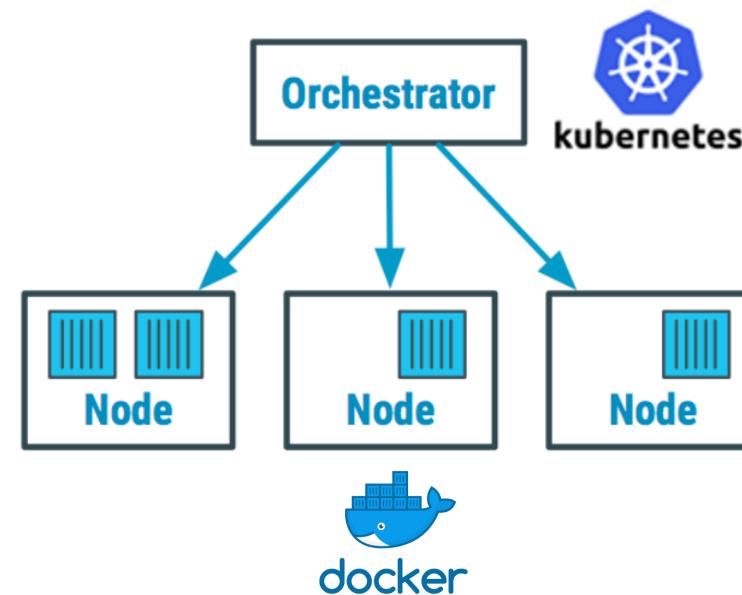
Configure rollbacks to revert to a previous application version if updates fails.

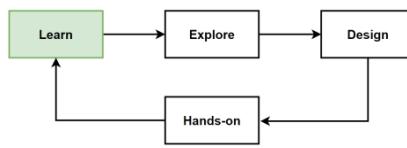
- **Isolate and Secure Environments**

Use namespaces, network policies, and access controls to isolate different environments (e.g., development, staging, production)

- **Resource Limits and Quotas**

Set resource (CPU, memory, etc.) limits and quotas for containers and namespaces to prevent resource starvation and ensure fair resource usage.





Best Practices for Using Container Orchestrators - 2

- Monitor and Observe Applications**

Implement monitoring and observability tools to collect metrics, logs, and traces from containers and apps, facilitating performance insights, bottleneck identification, and issue troubleshooting.

- Manage Storage Efficiently**

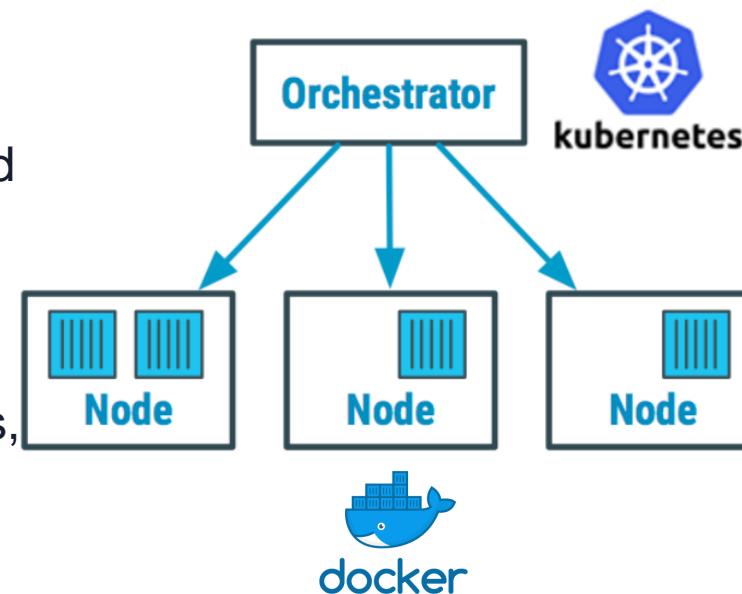
Use orchestrator's storage management features for stateful applications to provision, attach, and manage storage volumes, ensuring data persistence and efficient storage usage.

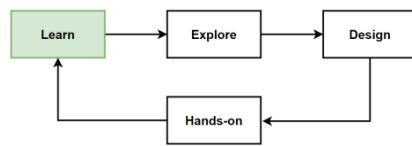
- Implement a CI/CD Pipeline**

Integrate your orchestrator with a continuous integration and continuous deployment pipeline to automate the building, testing, and deployment process, promoting fast and reliable application delivery.

- Implement Health Checks**

Configure health checks for your microservices, enabling the orchestrator to monitor their health and automatically restart or reschedule them if necessary.





How Container Orchestrators Work

Scheduling

- Orchestrators allocate containers to appropriate nodes within a cluster
- Factors like resource availability, affinity rules, and constraints are considered

Cluster Management

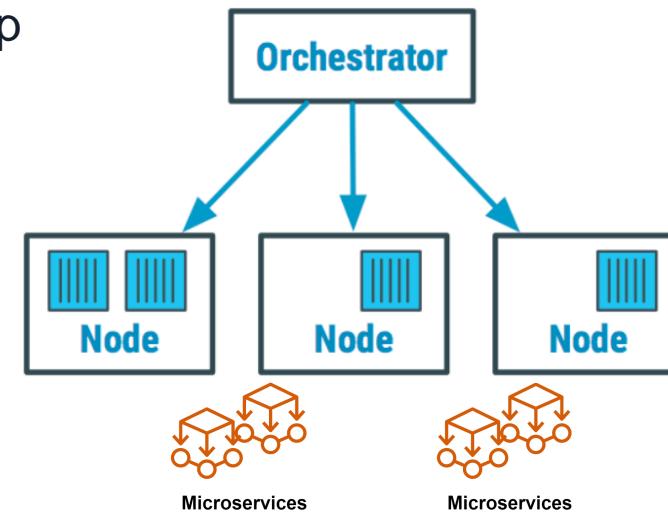
- Orchestrators monitor nodes and containers, maintaining the desired state of the application
- In case of a node failure, affected containers are rescheduled

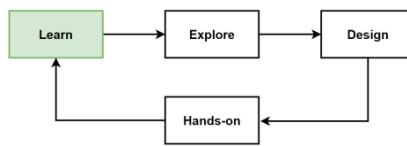
Service Discovery & Load Balancing

- Orchestrators assign unique IP addresses and DNS names for inter-container communication
- Traffic is efficiently distributed among containers, ensuring high availability

Health Checks & Self-healing

- Continuous health monitoring of containers is conducted
- Unresponsive or failed containers are automatically restarted or rescheduled
- Orchestrators provide metrics, logs, and traces for monitoring and observability tools





How Container Orchestrators Work - 2

Auto-Scaling

- Orchestrators enable application scaling by adjusting container instances based on demand
- Autoscaling features automate this process for optimal performance and resource utilization

Rolling Updates & Rollbacks

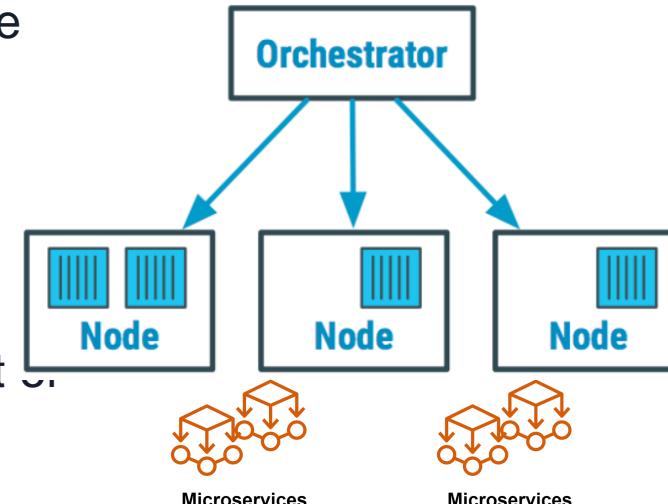
- Seamless application updates are facilitated by gradual replacement of old container instances
- In case of an update failure, orchestrators can revert to a previous version

Networking and Storage

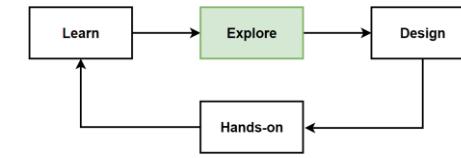
- Orchestrators manage networking and storage for containerized applications
- Provision of isolated networks for inter-container communication, and management of storage volumes

Security & Access Control

- Features like role-based access control (RBAC), network policies, and secrets management are provided for security
- Resource access control is ensured



Explore: Container Orchestrator tools: Kubernetes, Docker Swarm, Apache Mesos



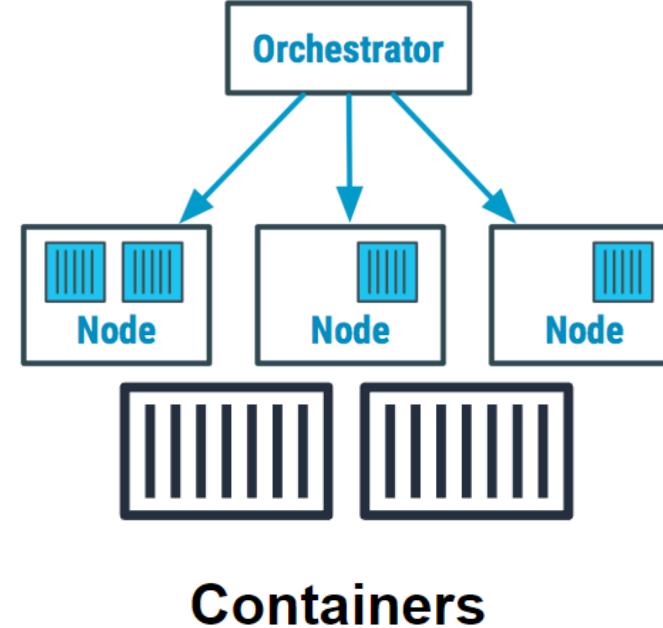
Exploring Container Orchestrator Tools

- Kubernetes
- Docker Swarm
- Apache Mesos

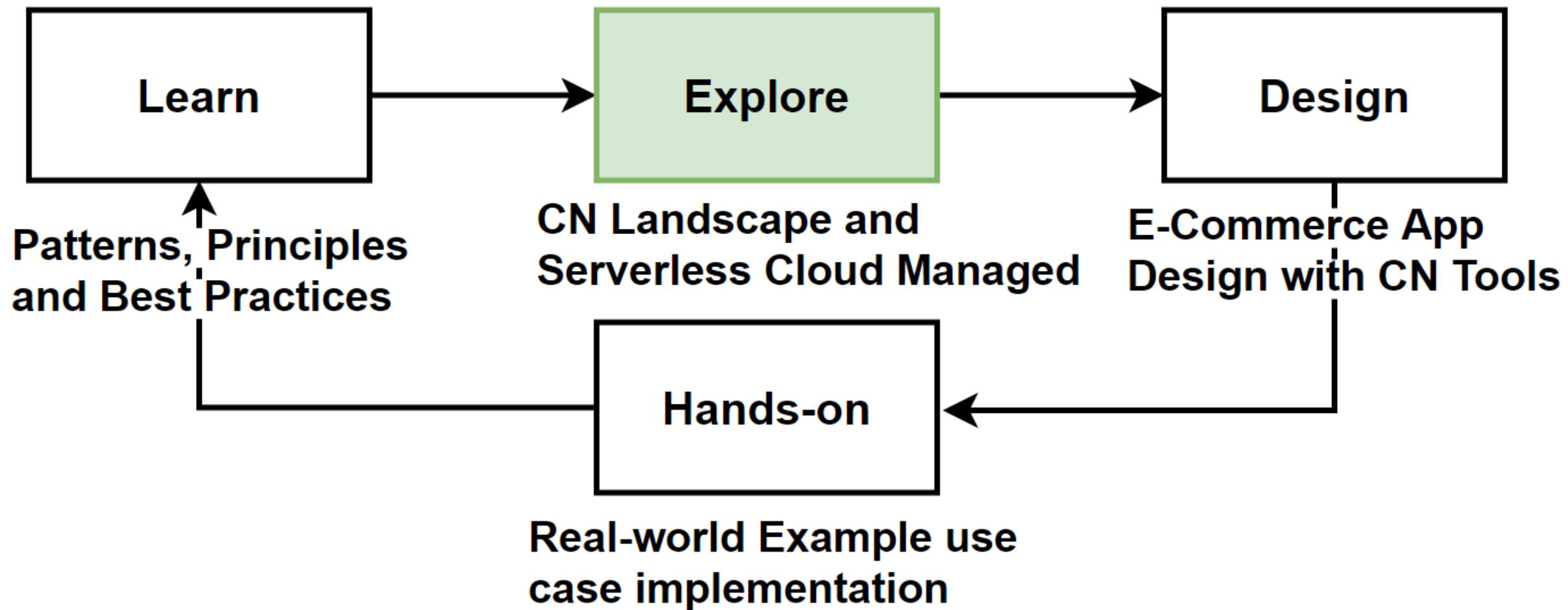
Goto -> <https://landscape.cncf.io/>



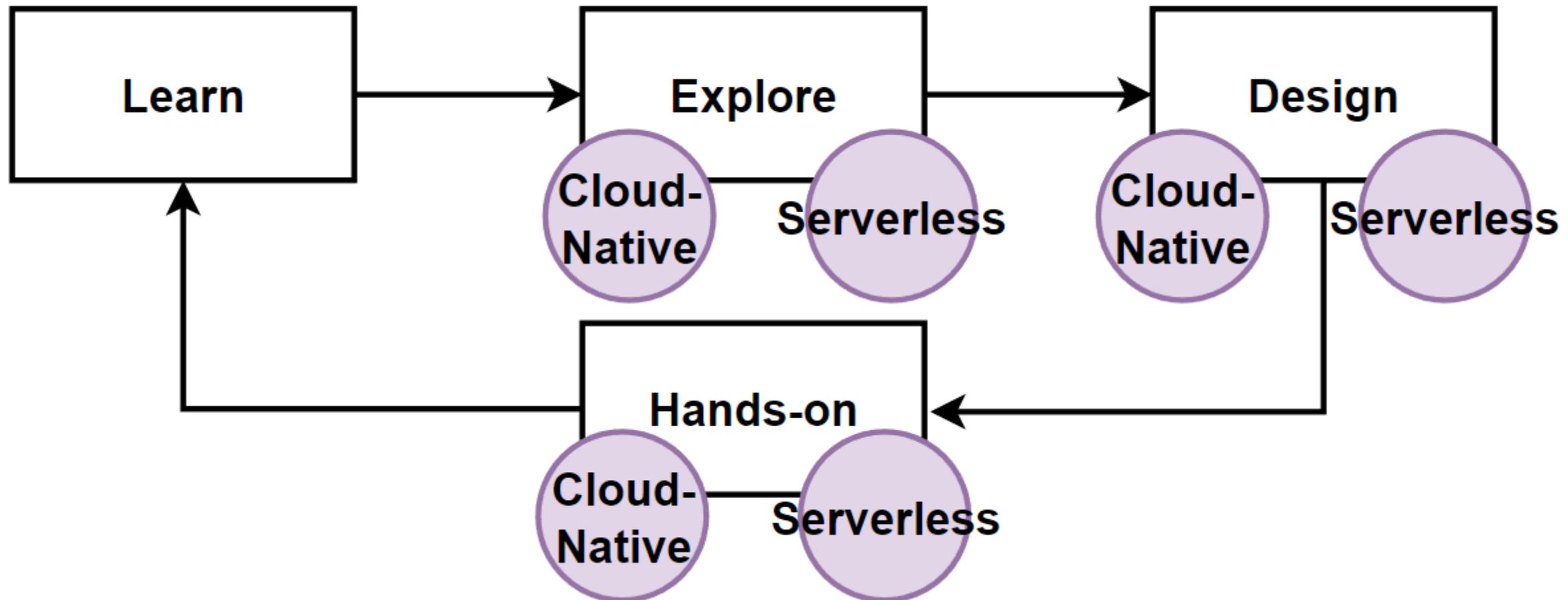
kubernetes



Way of Learning – The Course Flow



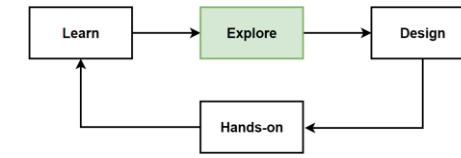
Way of Learning – Cloud-Native & Serverless Cloud Managed



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

Explore: Container Orchestrator tools: Kubernetes, Docker Swarm, Apache Mesos



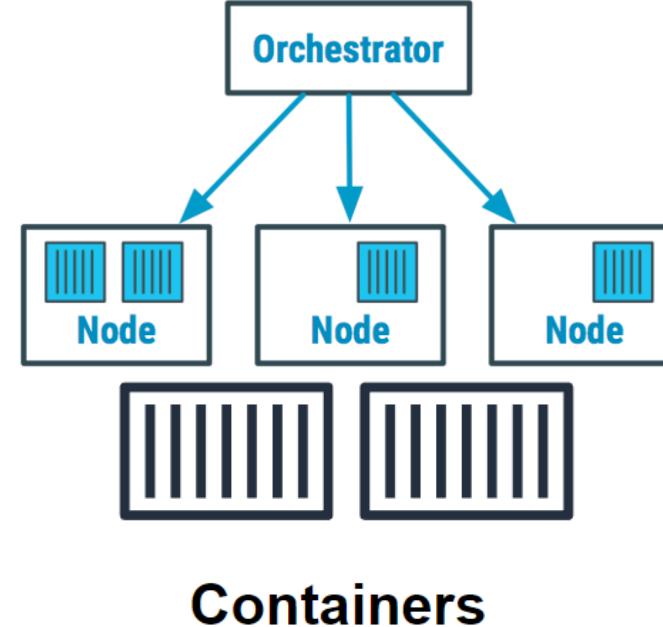
Exploring Container Orchestrator Tools

- Kubernetes
- Docker Swarm
- Apache Mesos

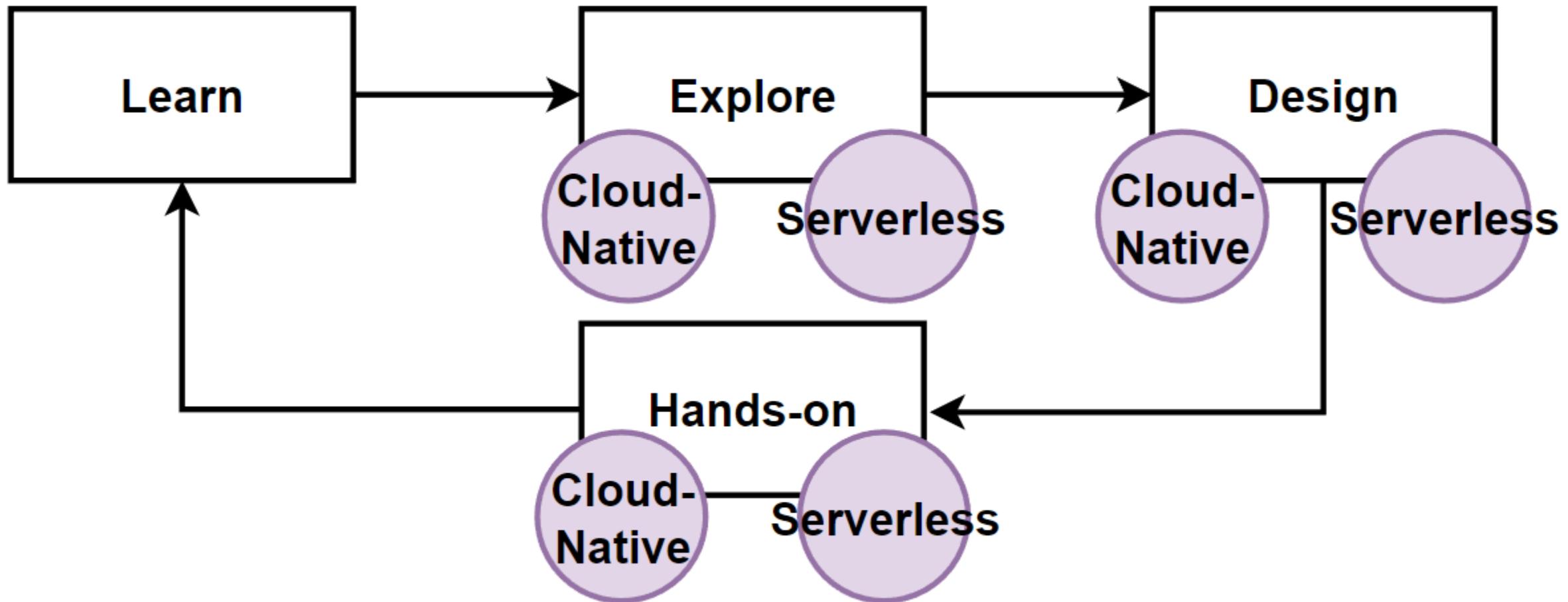
Goto -> <https://landscape.cncf.io/>



kubernetes

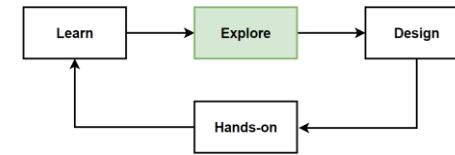


Way of Learning – Cloud-Native & Serverless Cloud Managed



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

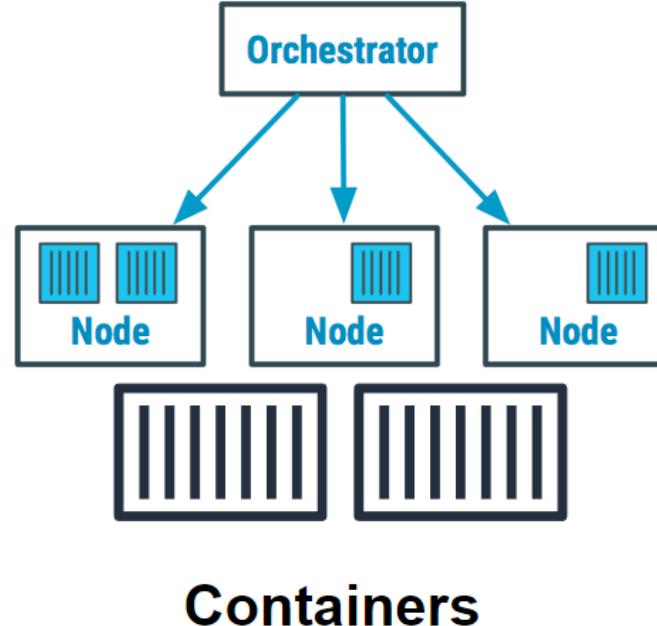


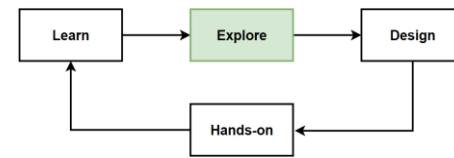
Explore: Cloud Container Orchestrator tools: EKS, GKS, AKS, Red Hat OpenShift

Exploring Cloud Container Orchestrator tools

- AWS EKS
- Google GKS
- Azure AKS

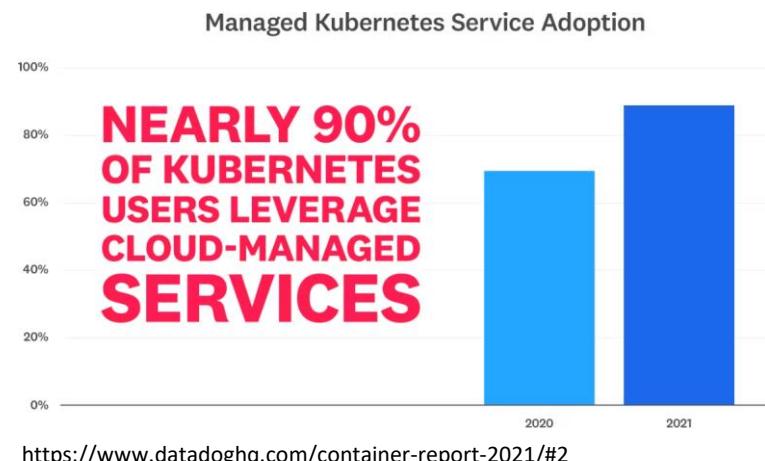
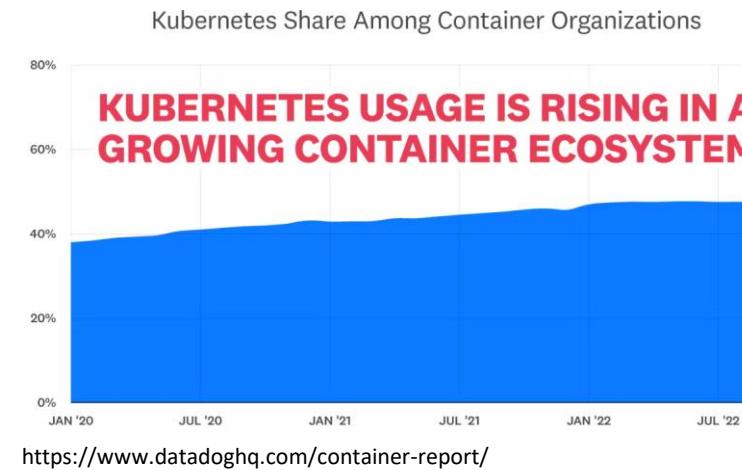
Goto -> <https://landscape.cncf.io/>



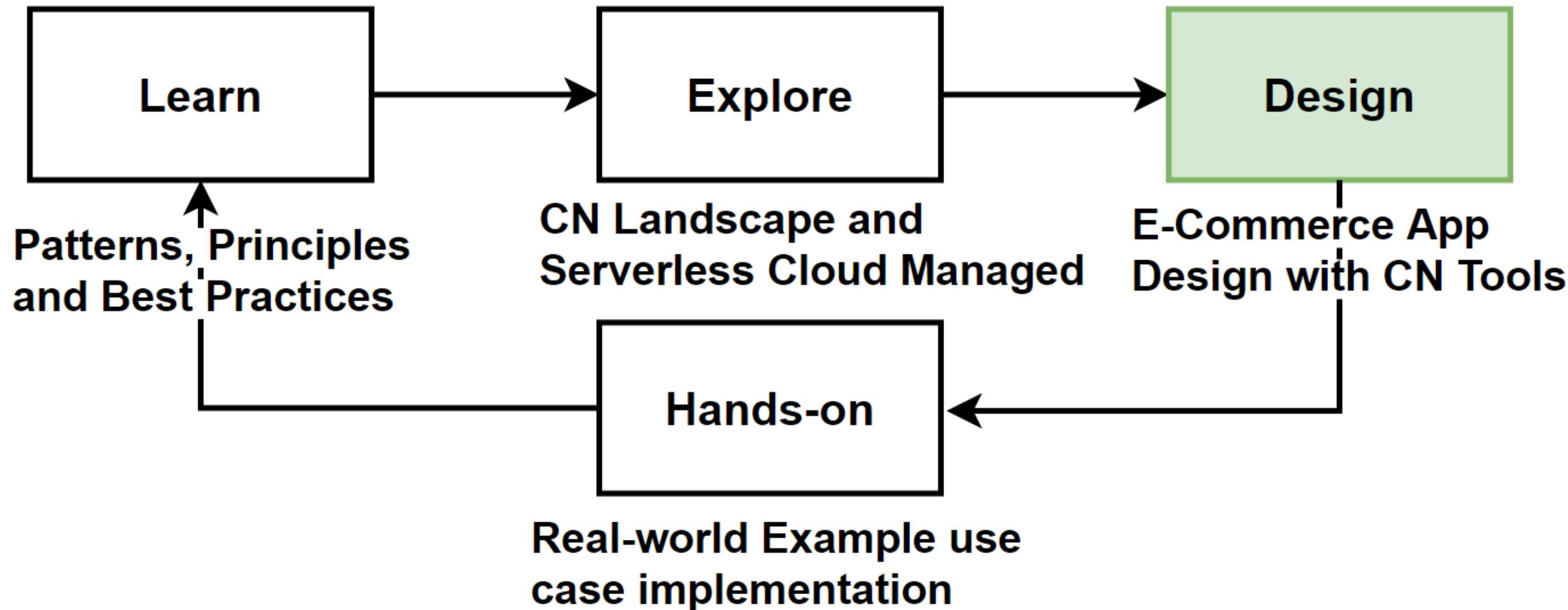


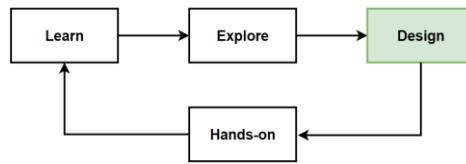
Datadog Container Reports: Kubernetes as the Defacto Standard for Containers

- **Datadog**: Essential monitoring and security platform for cloud apps that provides traces, metrics, and logs for end-to-end observability
- **Datadog releases annual Container Reports**: Kubernetes is the de facto standard for cloud-native containers
- **2020 Container Report**
Kubernetes runs in half of container environments, becoming the standard for container orchestration
- **2021 Container Report**
Almost all containers are orchestrated, with Kubernetes being used by over half of the organizations. Nearly 90% of Kubernetes users leverage cloud-managed services
- **2022 Container Report**
Container-based microservice applications are pervasive. Kubernetes continues to be the most popular container management system
- Container and Kubernetes adoption strengthens year after year
- Kubernetes is the de facto standard for the cloud-native world

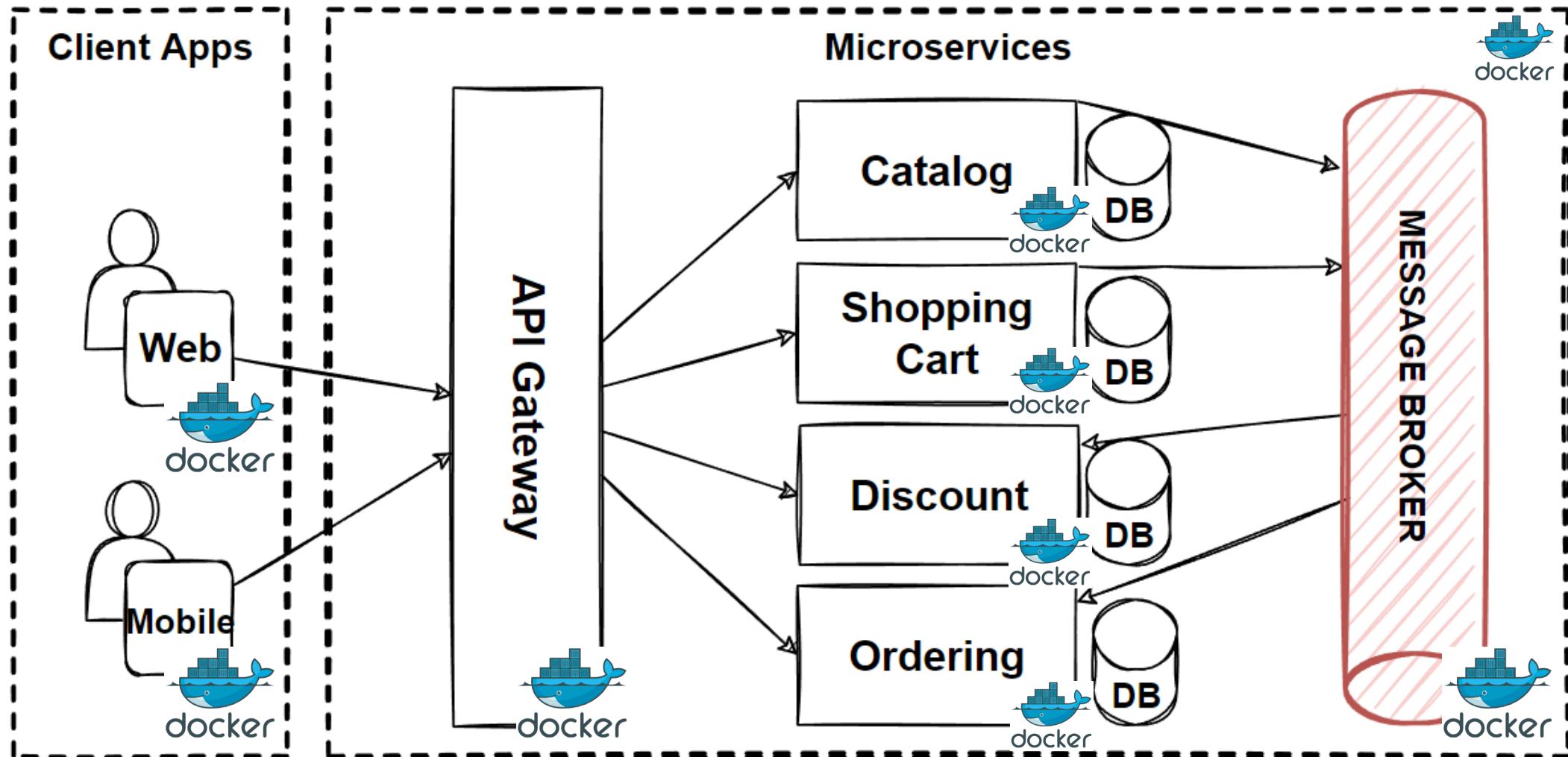


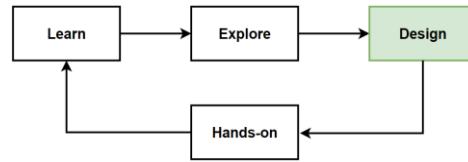
Way of Learning – The Course Flow



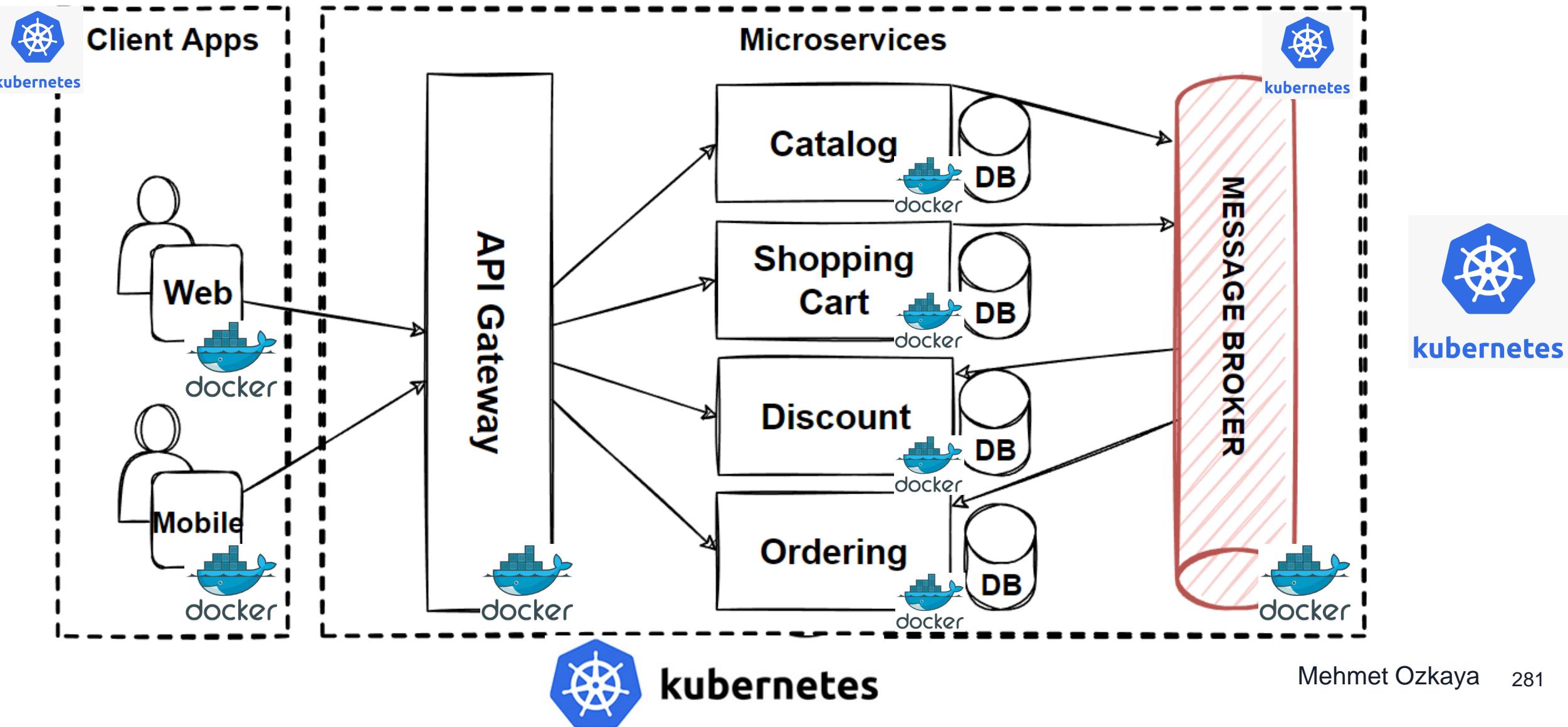


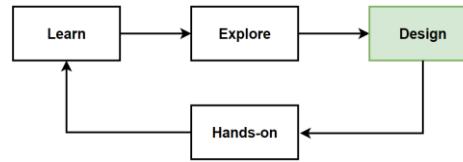
Design: Cloud-Native E-commerce w/ Containers





Design: Cloud-Native E-commerce w/ Orchestrator





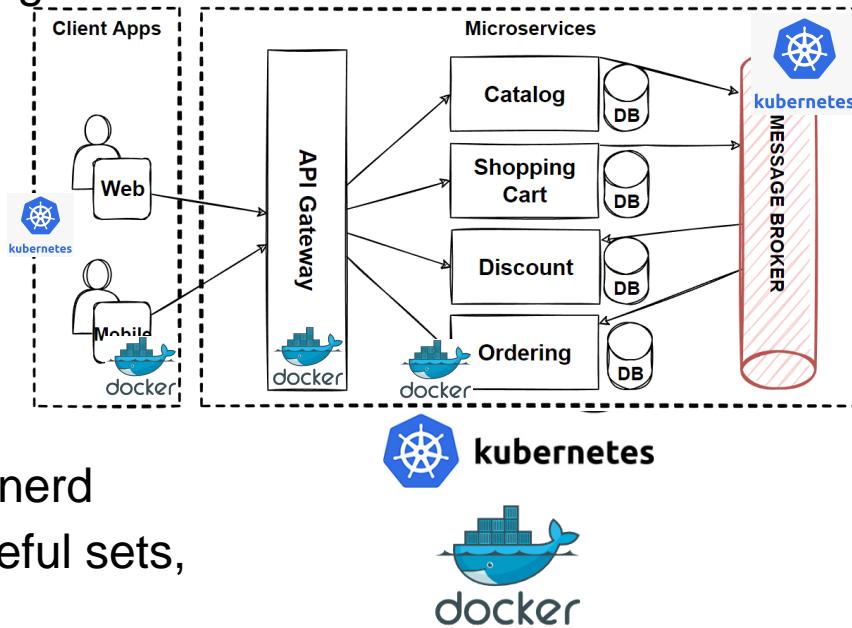
Container Orchestrator with Kubernetes

Container Orchestrator Tasks

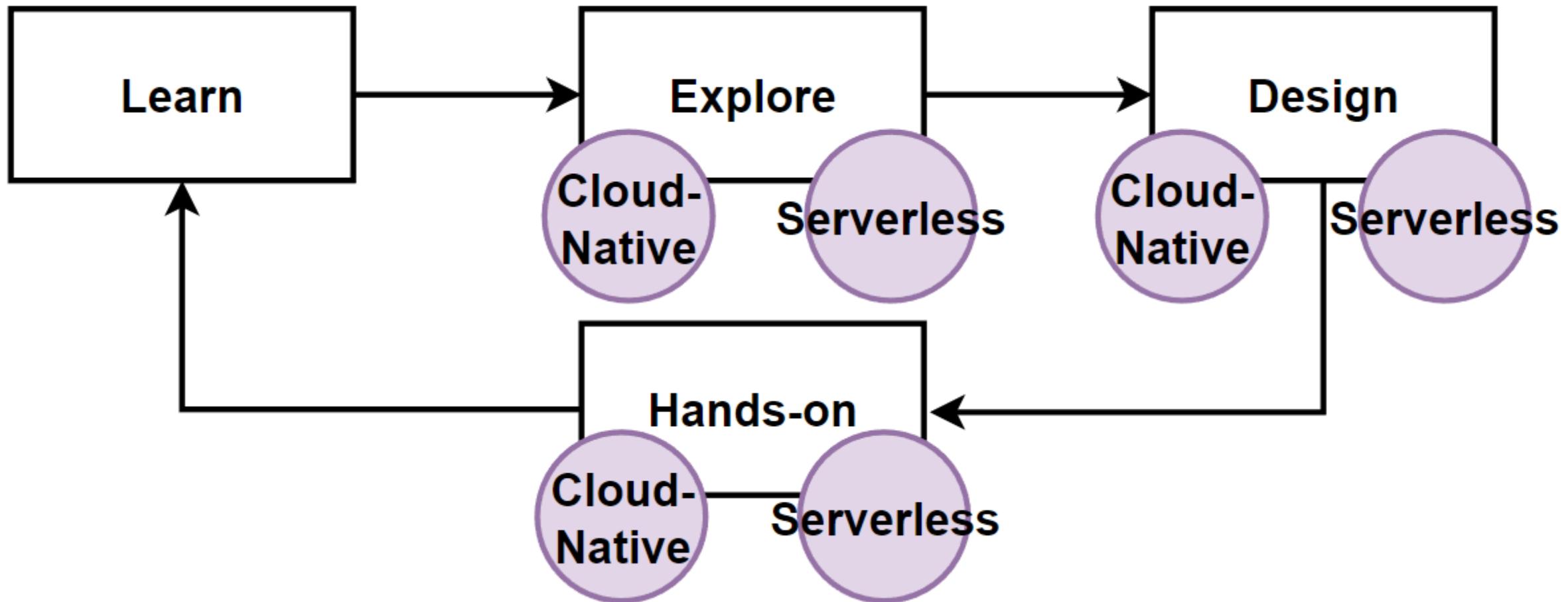
- Scheduling, Cluster management, Service discovery and load balancing, Health checks and self-healing, Scaling, Rolling updates and rollbacks, Networking and storage, Security and access control, Monitoring and observability.

Kubernetes

- Kubernetes has become the de facto standard for the cloud-native world.
- Developed by Google and maintained by the CNCF
- Highly extensible and feature-rich, with a large ecosystem of add-ons and integrations
- Supports a wide range of container runtimes, including Docker and containerd
- Provides advanced features like custom resource definitions (CRDs), stateful sets, and daemon sets
- Offers robust self-healing capabilities, with automatic rescheduling of failed containers
- Integrates well with major cloud providers, making it easy to deploy and manage containerized applications in the cloud

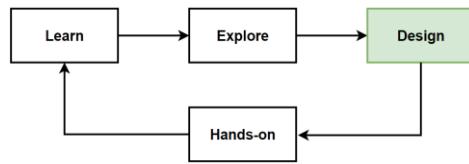


Way of Learning – Cloud-Native & Serverless Cloud Managed

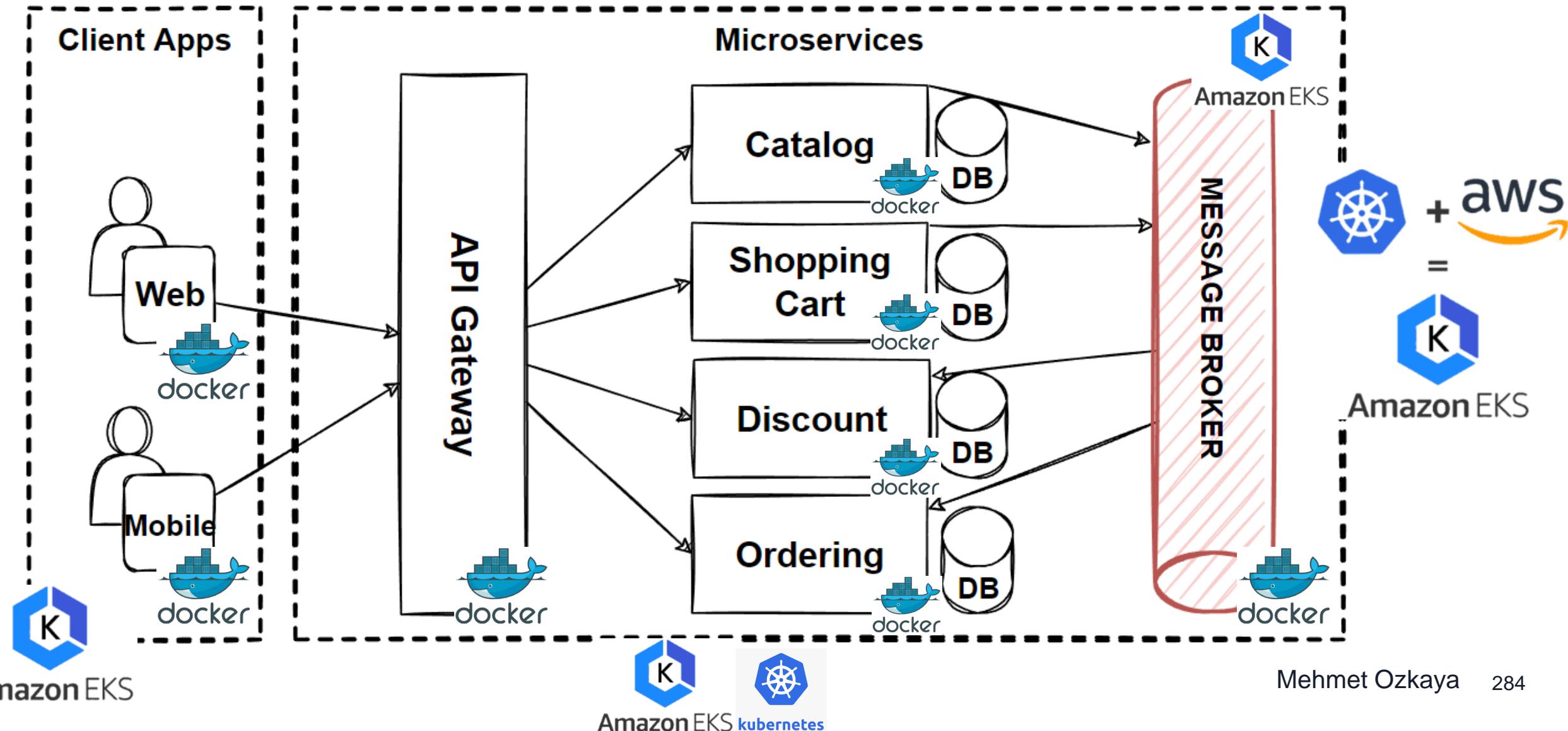


Examples of CN vs Serverless Cloud Tools:

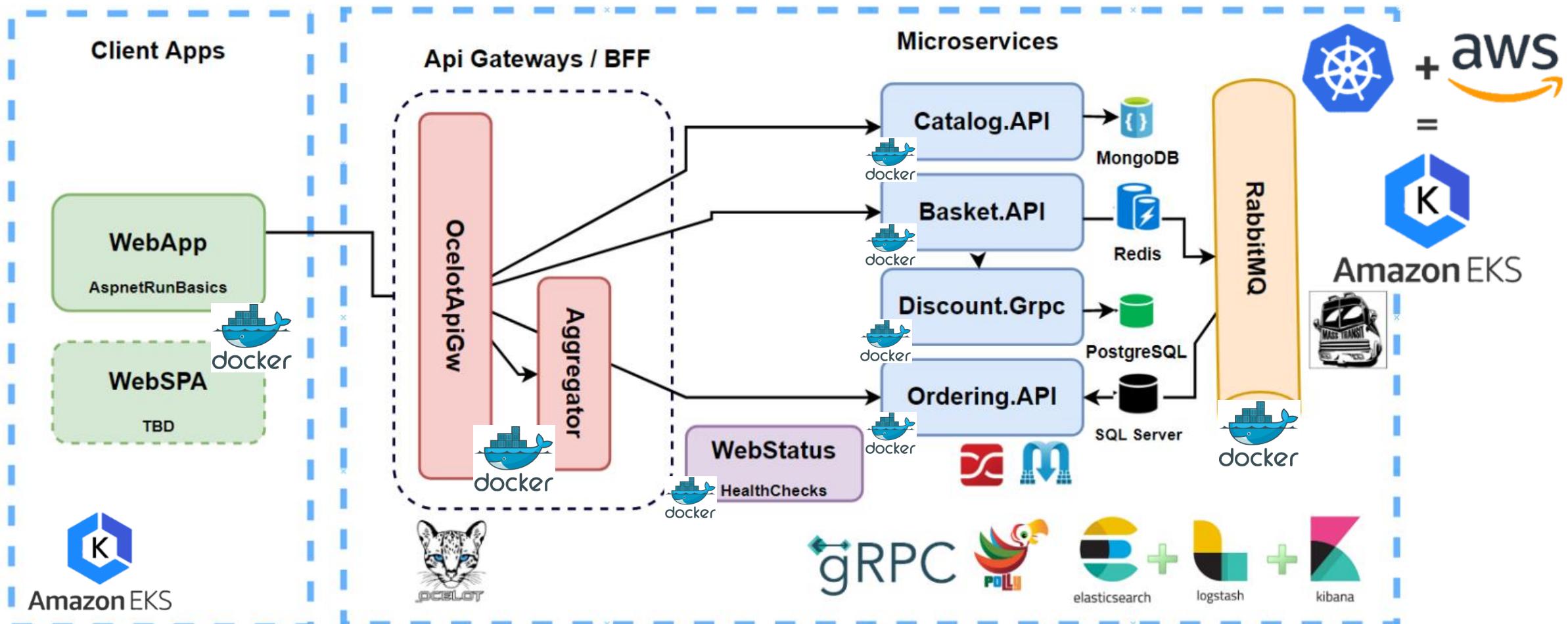
- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Design: Cloud-Native E-commerce w/ Amazon EKS



Reference Project: .Net Microservices - Cloud-Native E-commerce



DEMO: Code review of Microservices Architecture .NET Implementation

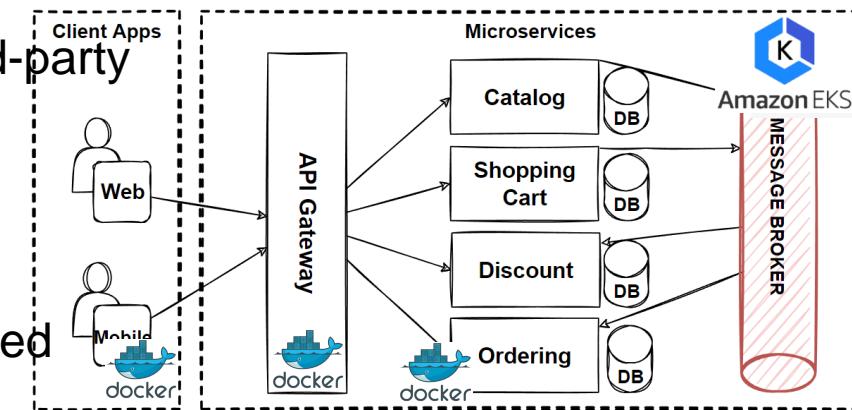
- <https://github.com/aspnetrun/run-aspnetcore-microservices>
- <https://github1s.com/aspnetrun/run-aspnetcore-microservices>

Container Orchestrator with Amazon Elastic Kubernetes Service (EKS)

- Provided by Amazon Web Services (AWS)
- Integrates well with other AWS services, such as Elastic Load Balancing, Amazon RDS, and AWS Identity and Access Management (IAM)
- Supports the Kubernetes ecosystem, including add-ons, plugins, and third-party tools

Amazon Elastic Kubernetes Service (EKS)

- Amazon EKS plays a crucial role as the container orchestrator
- It automates the deployment, scaling, and management of the containerized microservices
- handling the complexities of cluster management and ensuring high availability.
- EKS integrates with AWS services for load balancing, IAM roles, security, and monitoring, making the platform robust and secure.
- Focus on developing and deploying microservices while leaving the underlying infrastructure and orchestration complexities to AWS.



Deep Dive into Kubernetes - Defacto Standard for Containers

What Is Kubernetes? Why use microservices deployments ?

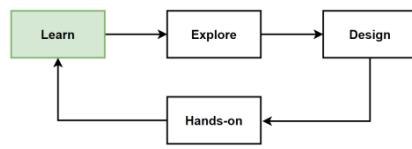
Kubernetes Use Cases and Advantages

Kubernetes Architecture and Components

Managed Kubernetes

Helm Charts with Kubernetes for Microservices Deployments

Mehmet Ozkaya

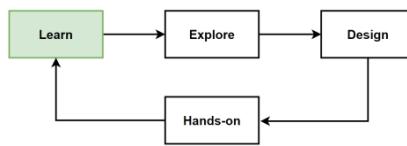


CNCF 2022 Annual Survey: The year cloud native became the new normal

- CNCF also publish annual surveys to track adoption of cloud-native products.
- Since 2015, the **Cloud Native Computing Foundation** has used its **unique position** in the cloud native community to survey the landscape
- **Understand the dynamics** and better serve users of open source, cloud native technologies.
- [CNCF 2022 Annual Survey](#)

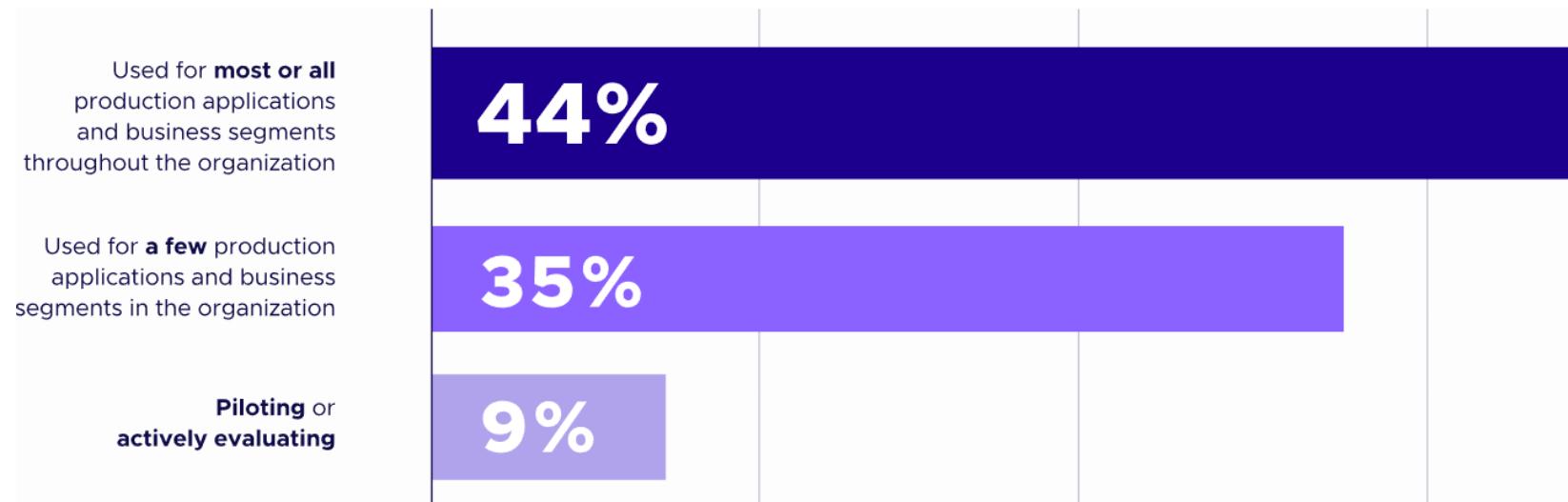
The year cloud native became the new normal

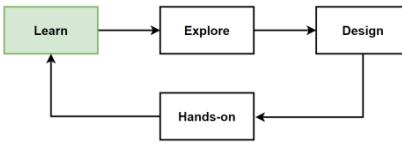




CNCF 2022 Key Findings: Containers are the new normal

- **Cloud-native technologies** empower organizations to build and run scalable applications
- **Containers, service meshes, microservices, immutable infrastructure, and declarative APIs.**
- Enable **loosely coupled** systems that are **resilient, manageable, and observable**.
- **44%** of respondents are **already using containers** for nearly all applications
- **Nearly half of all organizations** using containers **run Kubernetes** to deploy and manage containers.
- Growing adoption of containers, organizations were to **adopt a multi-cloud approach** as they grow in size.





CNCF 2022 Key Findings: Adoption of Kubernetes

- Advanced in their adoption and use of Kubernetes.
- More likely to maintain Kubernetes environments with more than 10 clusters.
- %64 using in productions

Does your organization use Kubernetes?



END USERS

64%

Using in production

25%

Piloting / evaluating



NON-END USERS

49%

Using in production

20%

Piloting / evaluating

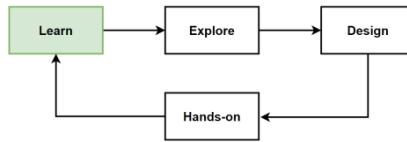
If your organization uses Kubernetes, how many production clusters do you have?

Don't Know / Not Sure 1-5 Clusters 6-10 Clusters 10+ Clusters

END USERS



Mehmet Ozkaya 290



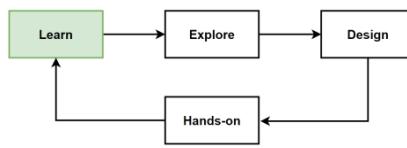
Kubernetes is Emerging as the 'Operating System'

- The **rise of modern, cloud-native computing** is intimately tied with containers and Kubernetes.
- Significant **majority of enterprises** globally are **leveraging Kubernetes** for running **critical business** applications.
- **Expanding Kubernetes technology** ecosystem introduces advanced features like **security, microservices communication, observability, scaling, and optimal resource utilization**.
- Kubernetes survey shows how organizations actually **use Kubernetes in production**.

Quote: Anita Schreiner - Dynatrace VP Delivery

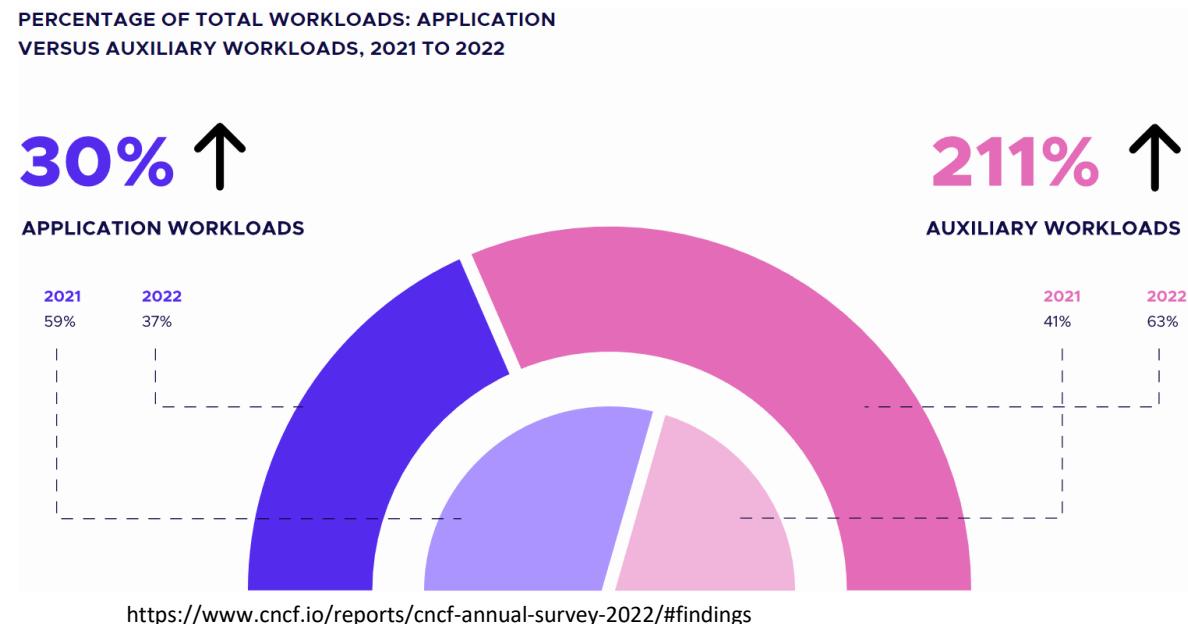
- *At Dynatrace, we use Kubernetes for any new software project, whether it's build pipelines or our SaaS offerings. We see the same trend with our customers. Kubernetes effectively has emerged as the operating system for the cloud.*
- **Kubernetes is emerging as the “operating system” of the cloud**
- The ideal orchestration platform, Kubernetes, is now **pivotal for running cloud-native microservice applications**.

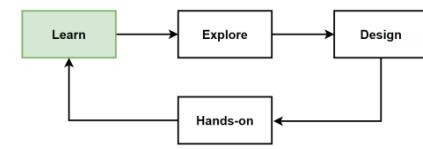




Application Workloads in Kubernetes Cluster

- In 2021, in Kubernetes cluster, **application workloads** accounted for **most of the pods (59%)**.
- **Non-application workloads**; system and auxiliary workloads, played a **relatively smaller part**.
- In 2022, as Kubernetes **adoption has grown**, auxiliary workloads now **outnumber application workloads (63% vs. 37%)**.
- Implement more **advanced Kubernetes technologies**: security controls, service meshes, messaging systems, and observability tools.
- Using Kubernetes for **build pipelines** and **scheduled utility workloads**...
- **Kubernetes is emerging as the “operating system” of the cloud**.

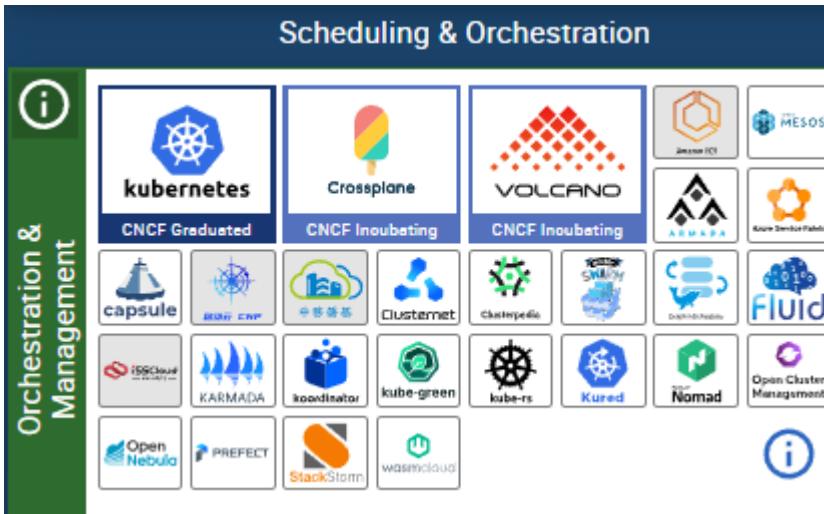




Cloud-Native Landscape and Trial Map

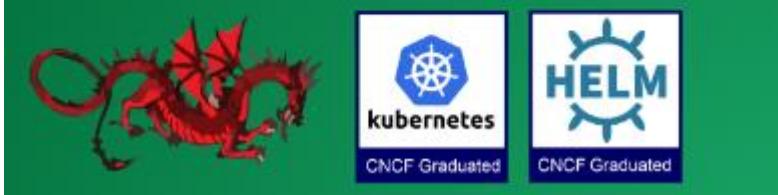
Where Kubernetes as a Container Orchestrator

- Cloud-Native Landscape
<https://landscape.cncf.io/>
 - Cloud-Native Trial Map
<https://github.com/cncf/trailmap>



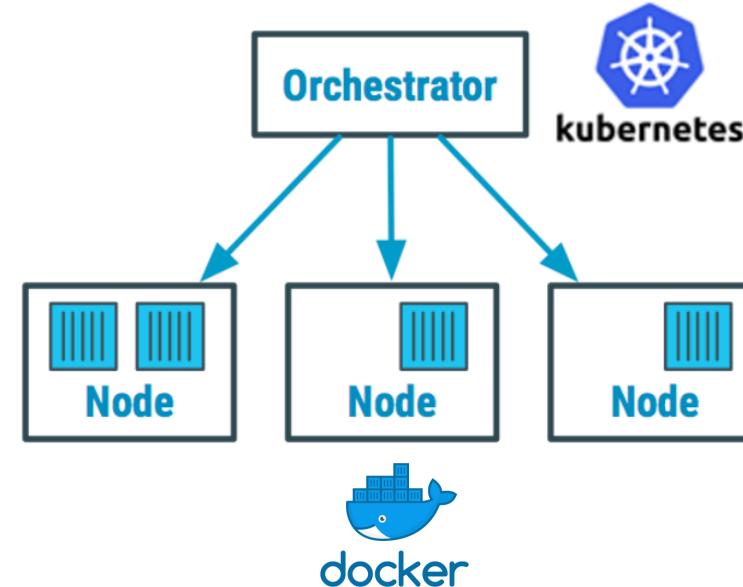
3. ORCHESTRATION & APPLICATION DEFINITION

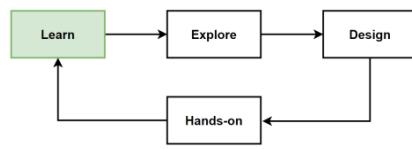
- Kubernetes is the market-leading orchestration solution
 - You should select a Certified Kubernetes Distribution, Hosted Platform, or Installer: cncf.io/ck
 - Helm Charts help you define, install, and upgrade even the most complex Kubernetes application



What is Kubernetes ?

- **Kubernetes** (also known as **k8s** or "kube") is an open source **container orchestration platform**.
- **Automates** many of the manual processes involved in **deploying, managing, and scaling containerized applications**.
- Developed by **Google** and is now maintained by the **Cloud Native Computing Foundation (CNCF)**.
- **Kubernetes** is a **portable, extensible, open-source platform** for managing containerized workloads and services, both **declarative configuration** and **automation**.
- It has a large, **rapidly growing ecosystem**. Kubernetes services, support, and tools are widely available.
- **Create cluster groups** of hosts running Linux containers, Kubernetes helps you easily and efficiently manage those clusters.
- Kubernetes is designed to provide a **platform-agnostic way to manage containerized applications at scale**.





Benefits of Kubernetes

- **Self-healing**

Kubernetes can automatically restart containers that fail, ensuring that your microservices remain available even if individual containers go down.

- **Automatic scaling**

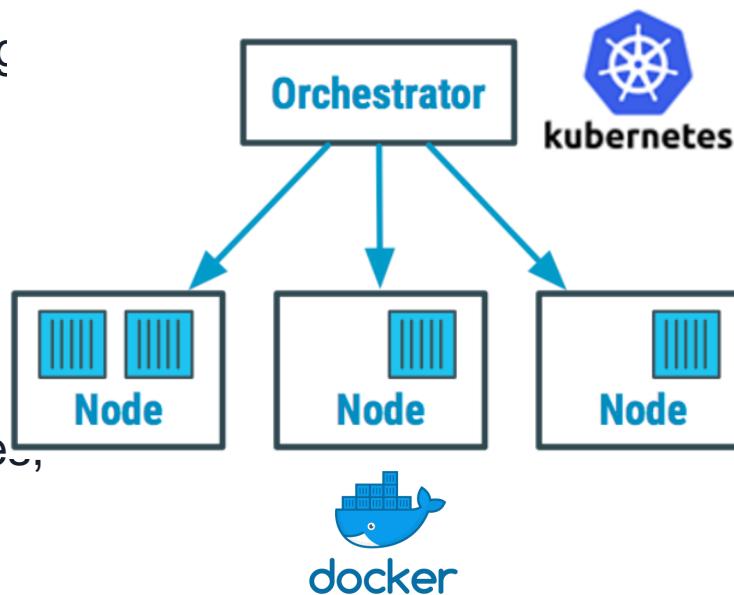
Automatically scale your microservices up or down as needed to meet changing demand, making it easier to handle sudden spikes in traffic.

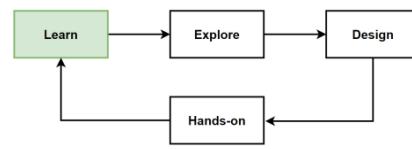
- **Load balancing**

Automatically distribute incoming traffic across multiple instances of a microservice.

- **Declarative configuration**

Allows to specify the desired state of your microservices using configuration files, easy to automate the deployment and management of your applications.





Uses Cases of Kubernetes

Microservices architecture

- Well-suited for managing apps built using a microservices architecture that simplifies the deployment, scaling, and management of independent, small services.

Hybrid and multi-cloud deployments

- Run on various cloud providers, on-premises, and hybrid environments, providing a consistent deployment experience
 - Avoid vendor lock-in and leverage the benefits of multiple cloud providers

Continuous Integration and Continuous Deployment (CI/CD):

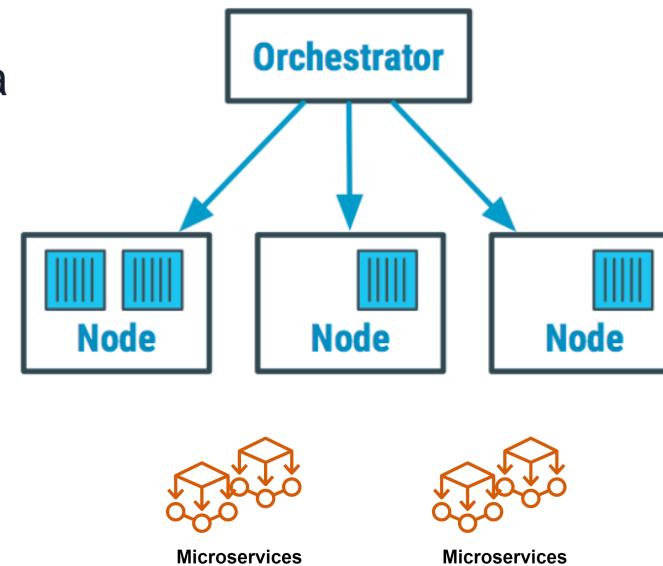
- Integrates with CI/CD pipelines for automated build, test, and deployment of app
 - Enables faster development cycles and ensures applications are always up-to-date

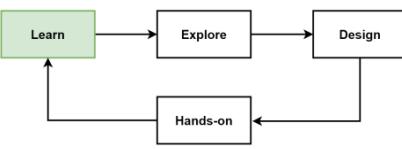
Load balancing and traffic routing

- Built-in load balancing and traffic routing capabilities, distributing traffic evenly among your services and containers

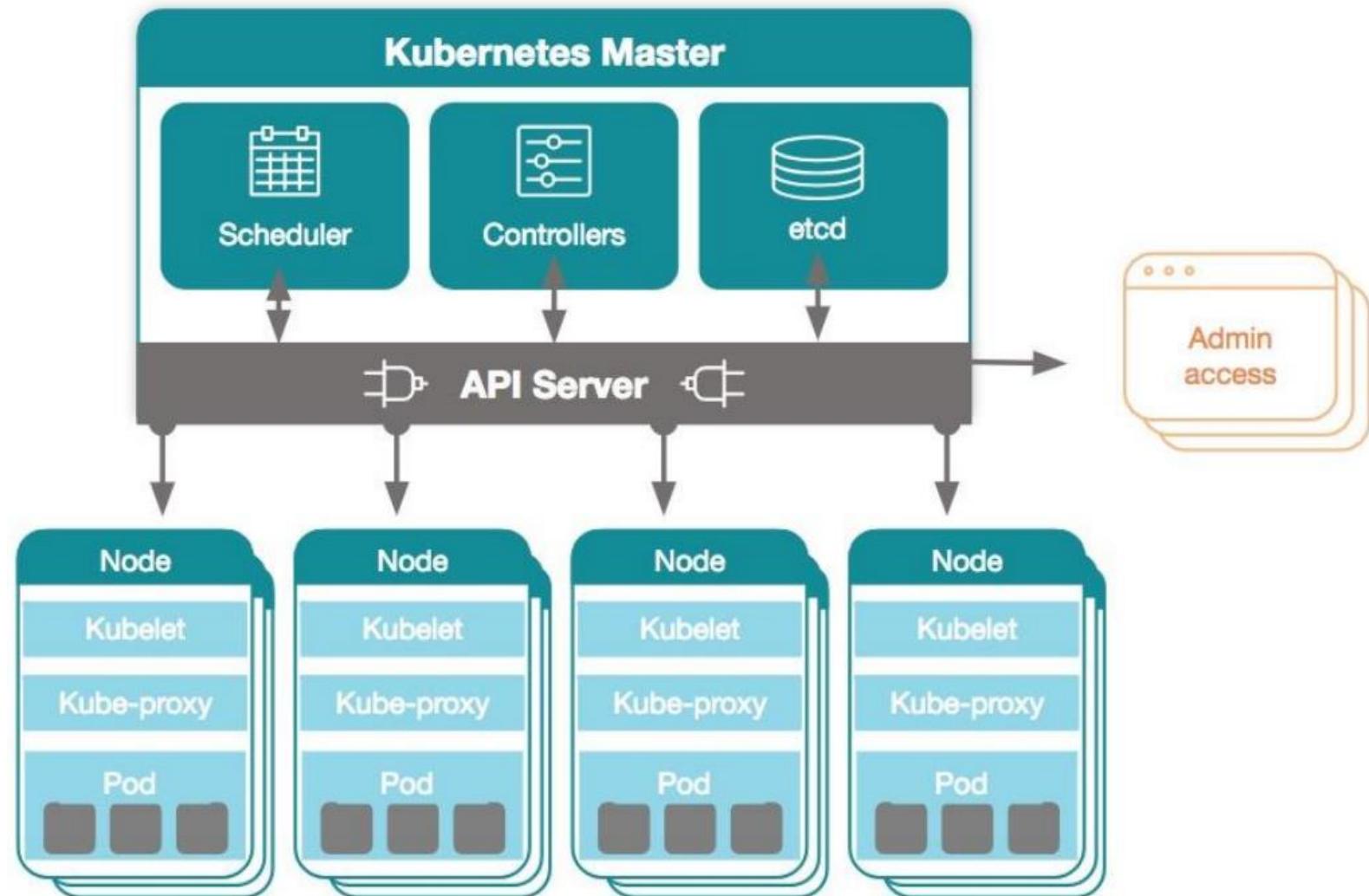
Fault tolerance and high availability with self-healing

- Automatically detects and handles container failures, ensuring applications continue to run

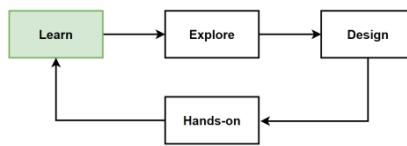




Kubernetes Architecture



<https://kubernetes.io/docs/concepts/overview/components/>



Kubernetes Components

- **Pods**

Pods are the smallest deployable units of computing, that you can create and manage in Kubernetes. Pods stores and manage our docker containers.

- **ReplicaSet**

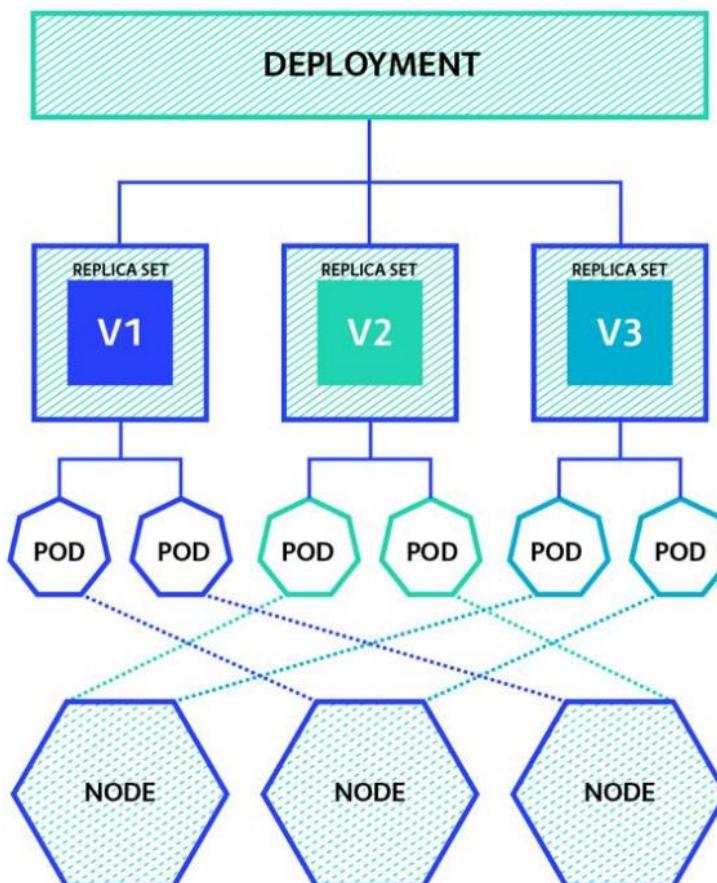
Maintain a stable set of replica Pods running at any given time. Used to guarantee the availability of a specified number of identical Pods.

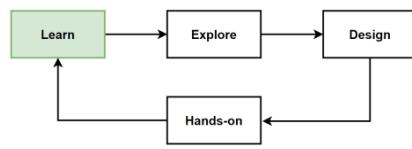
- **Deployments**

Provides declarative updates for Pods and ReplicaSets. Describe a desired state in a Deployment, and the Deployment Controller changes the actual state to the desired state at a controlled rate.

- **Deployments are an abstraction of ReplicaSets, and ReplicaSets are an abstraction of Pods.**

- **Pods should not created directly**, if needed, Deployment objects should be created.





Kubernetes Components - 2

- **Service**

Expose an application running on a set of Pods as a network service. Kubernetes gives Pods their own IP addresses and a single DNS name for a set of Pods, and can load-balance across them.

- **ConfigMaps**

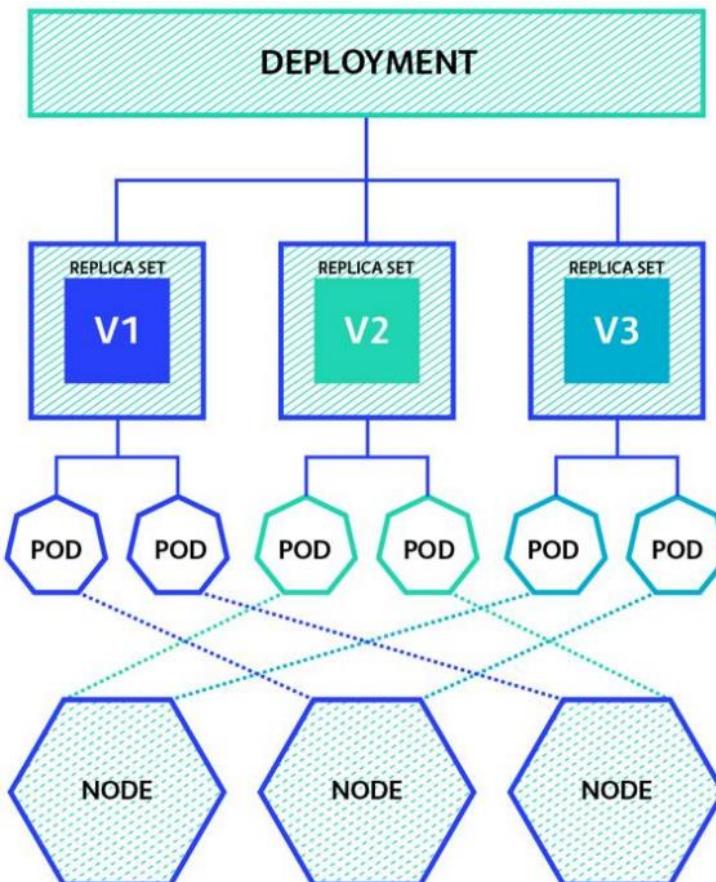
API object used to store non-confidential data in key-value pairs. Pods can consume ConfigMaps as environment variables, command-line arguments, or as configuration files in a volume.

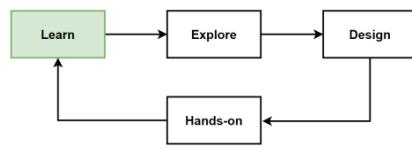
- **Secrets**

Store and manage sensitive information, such as passwords, OAuth tokens, and ssh keys.

- **Volumes**

Persistent storage location that can be mounted into a pod. Volumes can be used to store data that needs to persist across container restarts or be shared between multiple containers in a pod.





How Kubernetes Works

Container Packaging

- Kubernetes leverages containerization, packaging each microservice with its dependencies into a container for consistent behavior across different environments.

Defining Desired State

- Specify the desired state of the application using Kubernetes manifest files, detailing elements like the number of replicas, the container images, the needed network and storage resources.

Submitting Manifest to API Server

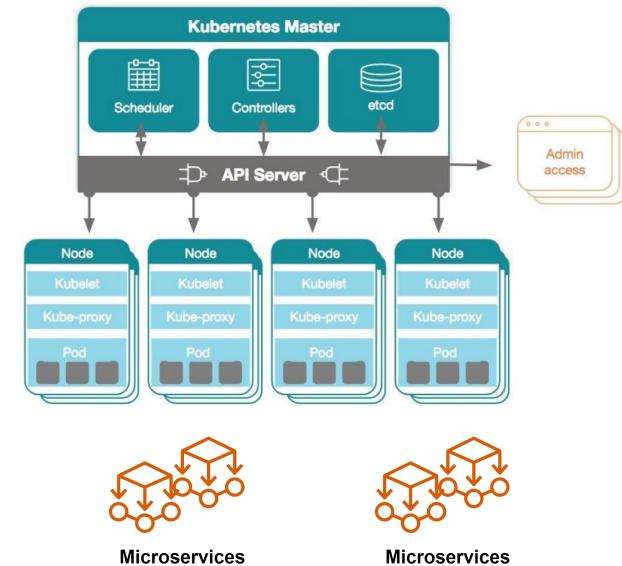
- Manifest files are validated and submitted to the Kubernetes API server, storing the configuration data in the etcd datastore.

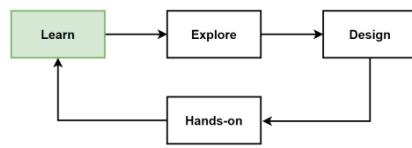
Scheduling

- The Kubernetes Scheduler assigns new or rescheduled pods to suitable worker nodes based on factors such as resource availability and node affinity.

Deployment

- The kubelet on each worker node ensures the desired containers are running, creating and starting containers using the specified container runtime.





How Kubernetes Works - 2

Networking and Services

- Manages networking between pods and services, offering each pod a unique IP address and DNS name. It facilitates communication between pods using services and ingress resources.

Monitoring

- Continuously compares the cluster's current state to the desired state, making corrective actions if necessary, like restarting or rescheduling a failing container.

Autoscaling

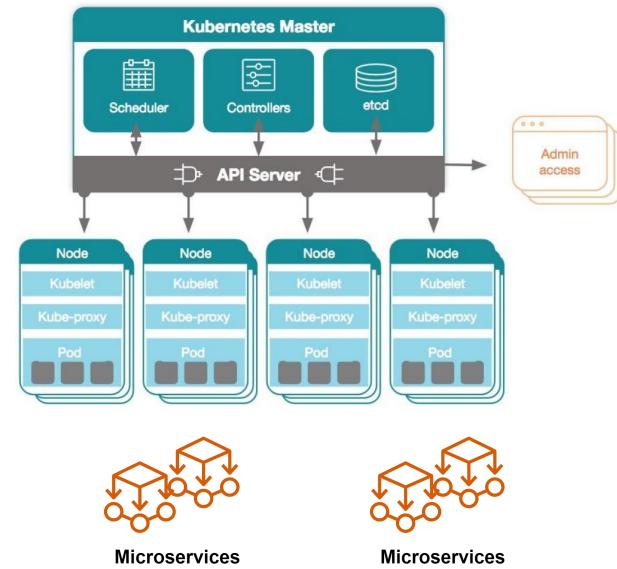
- Scale applications automatically or manually, based on rules like CPU utilization or changes to the number of desired replicas in the manifest files.

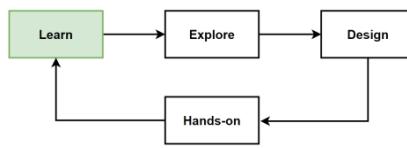
Rolling Updates and Rollbacks

- Supports seamless deployment of new app versions and can roll back to a previous version if an update fails or causes issues.

Health Checks

- Uses liveness and readiness probes to monitor container health within pods, managing traffic accordingly.





Declarative and Imperative way of Kubernetes

- **2 approaches** to manage resources and configurations: **Declarative and Imperative**.

- **Declarative Approach**

Desired state of resources is defined in configuration files, typically YAML or JSON.

- Kubernetes constantly checks the actual state against the desired state, adjusting as necessary.

- The command «**kubectl apply**» is used for this approach.

- **Example:** Using a YAML file, deployment.yaml, to define a desired state.

- To create or update the resource with the desired state, use the command:

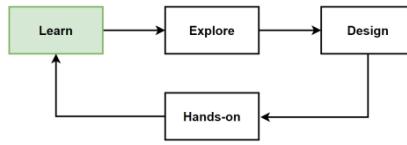
- **`kubectl apply -f deployment.yaml`**

- K8s ensure that the actual state of the resource matches the desired state defined in the deployment.yaml file.

```

apiVersion: apps/v1
kind: Deployment
metadata:
  name: my-app
spec:
  replicas: 3
  selector:
    matchLabels:
      app: my-app
  template:
    metadata:
      labels:
        app: my-app
    spec:
      containers:
      - name: my-app
        image: my-image:1.0
        ports:
        - containerPort: 8080

```



Declarative and Imperative way of Kubernetes

- **Imperative Approach**

Direct commands are issued to Kubernetes to create, update, or delete resources.

- Commands are executed using kubectl followed by an action, like create, delete, replace, or scale.

- **Example:** Creating and scaling a deployment using imperative commands:

- **Create:** `kubectl create deployment my-app --image=my-image:1.0`

- **Scale:** `kubectl scale deployment my-app --replicas=3`

- **Imperative Approach Declarative vs Imperative**

- **Declarative:** Define desired state in configuration files, Kubernetes makes necessary changes to achieve that state.

- **Imperative:** Issue direct commands to create, update, or delete resources, specifying the exact actions.

- Generally, the **declarative approach is recommended** due to its robustness and manageability.

- However, the **imperative approach can be useful for one-off operations** or for those still learning Kubernetes.

Hands-on: Deploy Microservices to Kubernetes

Getting Started with Kubernetes

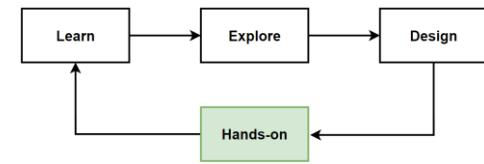
Installing Minikube for local Kubernetes Development

Installing kubectl and minikube

Troubleshooting Minikube

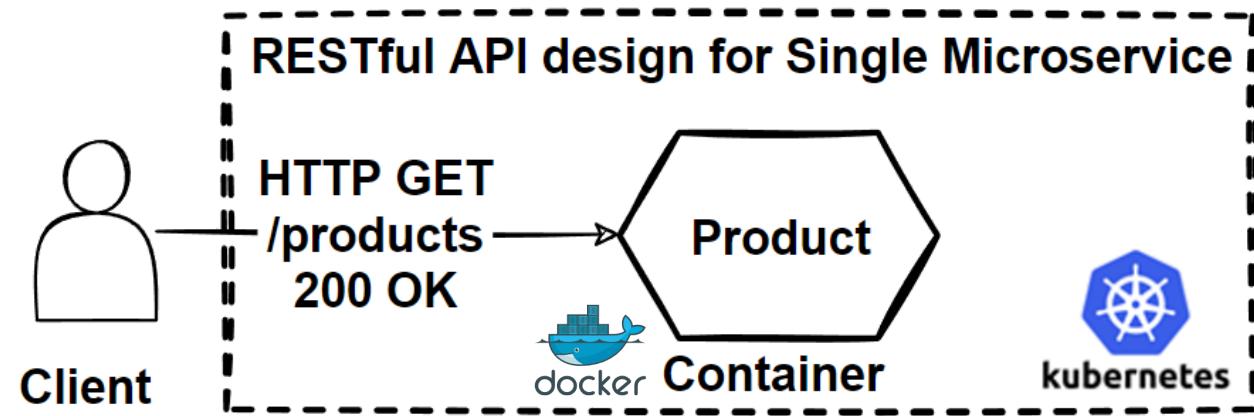
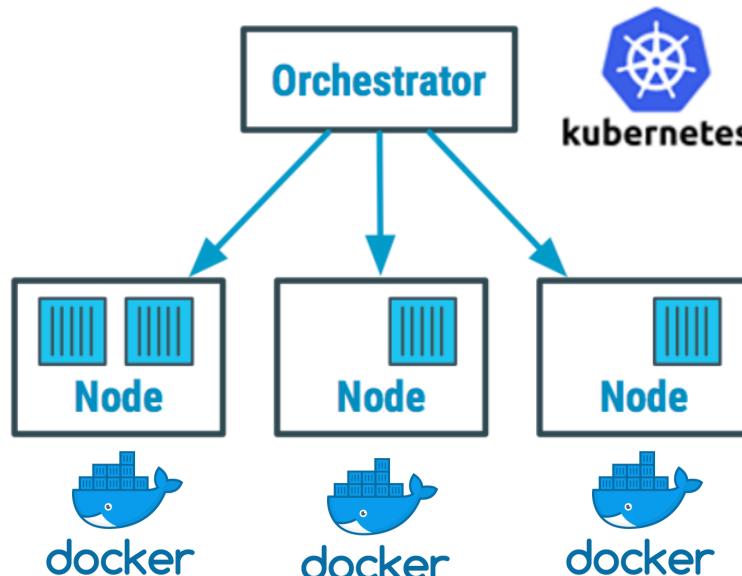
Kubernetes Pods, Services, ReplicaSets, Deployments

Mehmet Ozkaya

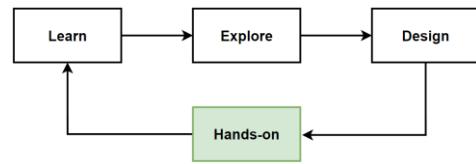


Hands-on: Deploy Microservices to Kubernetes

- Setting Up a Kubernetes Cluster
- Deploying Microservices on Kubernetes
- Docker Containers and Kubernetes Pods
- Deploying, Scaling, and Rolling Updates with Deployments
- Service Exposure and Ingress Controllers

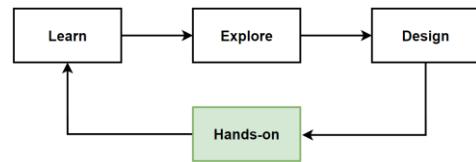


GET /api/products: Retrieves all products.
GET /api/products/{id}: Retrieves a specific product by ID.
POST /api/products: Adds a new product.
PUT /api/products/{id}: Updates an existing product by ID.
DELETE /api/products/{id}: Deletes a specific product by ID.



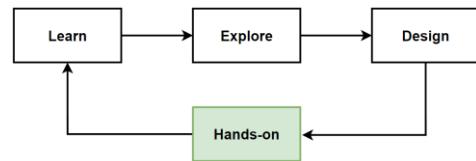
Install Minikube and kubectl

- **Install Minikube**
<https://minikube.sigs.k8s.io/docs/start/>
- **Install kubectl**
<https://kubernetes.io/docs/tasks/tools/install-kubectl/>
- **Test to ensure the version of kubectl**
kubectl version --client



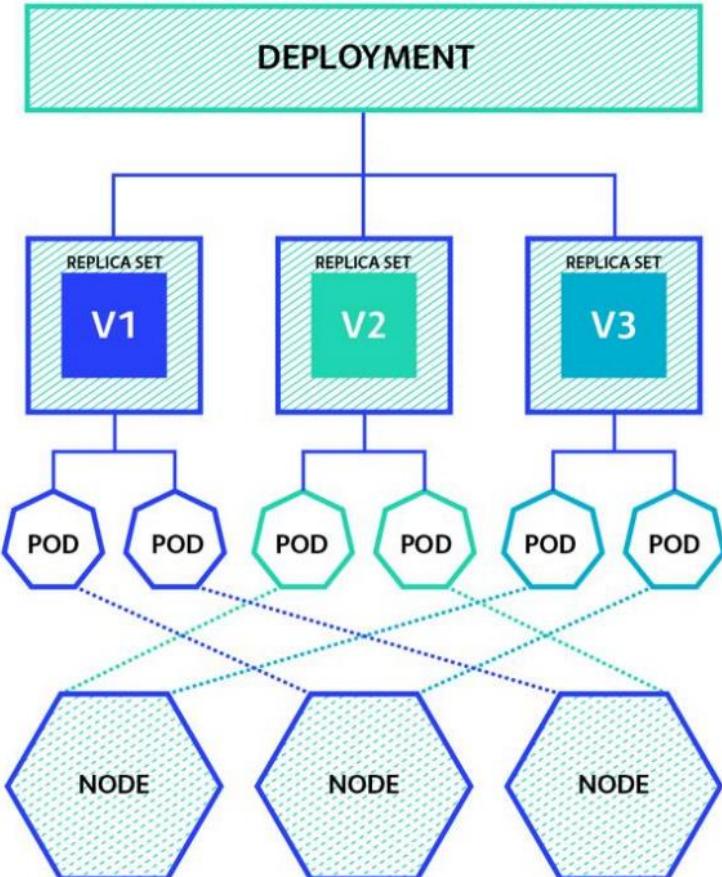
Kubernetes Configuration Best Practices for Containers

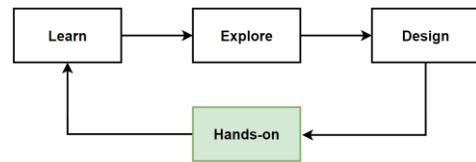
- By default, many applications bind to **localhost (127.0.0.1)** which makes them only accessible from within the container.
- To be **accessible from outside the container**, your **application should bind to 0.0.0.0**.
- To resolve the issue, **update your application to bind to 0.0.0.0** instead of localhost or 127.0.0.1.
- The process of changing the binding IP address will depend on your application and its configuration.
- For example, in a .NET 7 web application using Kestrel, you can update the Program.cs file to bind to 0.0.0.0.
- **WHY ?**
Any port which is listening on the default "0.0.0.0" address inside a container will be accessible from the network.
- **Configuration Best Practices of Kubernetes**
<https://kubernetes.io/docs/concepts/configuration/overview/>
- Java, NodeJS containers not required, Asp.Net need explicit configure port on Dockerfile or code.
<https://github.com/dotnet/dotnet-docker/issues/3968>



Why Should not Create Pod on Kubernetes ?

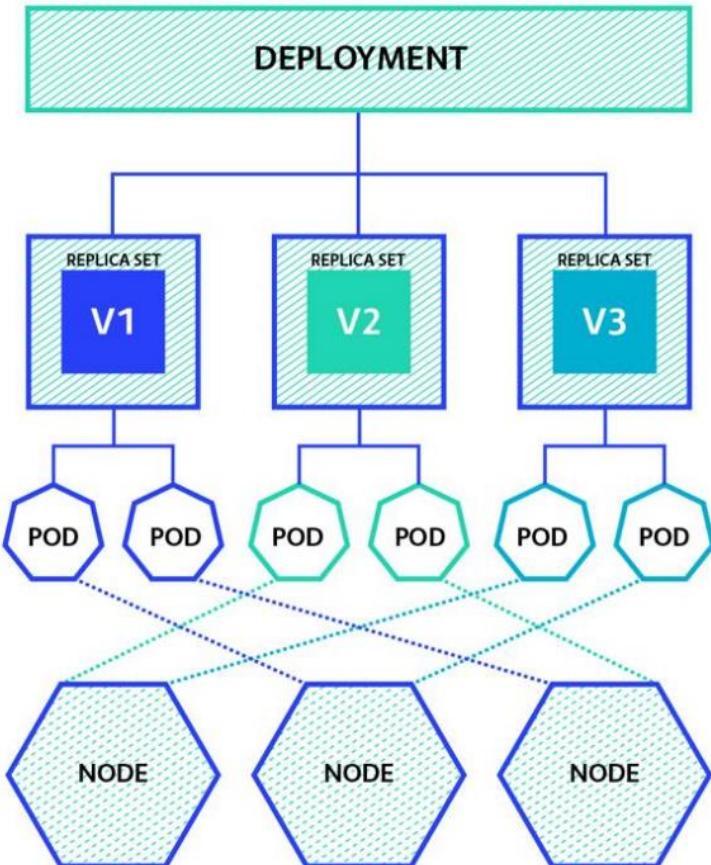
- While you can create **standalone Pods** on Kubernetes, it is **not recommended** because **Pods** are the **lowest-level abstraction** in **Kubernetes**.
- **Lack of self-healing**
If a Pod fails, is terminated, or becomes unhealthy, it will not be automatically replaced. In contrast, higher-level abstractions like Deployments automatically manage the desired number of replicas and replace any failed Pods.
- **No scaling support**
Need to manually create and manage multiple Pod YAML files to scale your application. Deployments make scaling easy by allowing you to simply update the desired number of replicas.
- Should make the Pods Resilient with Deployments.

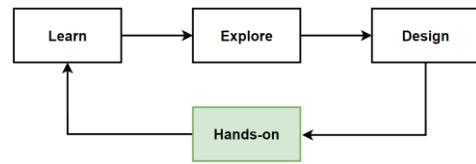




Make the Pods Resilient with Deployments

- Creating a Deployment YAML file is generally recommended over creating a standalone Pod YAML file for the following reasons:
- **Replica management**
Deployments automatically manage multiple replicas of your app. If a Pod fails or is terminated, the Deployment will automatically create a new Pod to maintain the desired number of replicas.
- **Rolling updates**
Deployments provide built-in support for rolling updates, enabling you to seamlessly update apps with minimal downtime. Deployments ensure that only a certain number of Pods are taken down and replaced with the updated version
- **Scaling**
Deployments make it easy to scale apps by simply updating the desired number of replicas. Standalone Pods would require you to manually create and manage multiple Pod YAML files.
- **Self-healing**
Deployments monitor the health of the Pods and automatically replace any failed or unhealthy Pods to ensure the application stays up and running.





4 Types of Kubernetes Service

- **ClusterIP**

Default Service type. It exposes the Service on an internal IP within the cluster, making it reachable only from within the cluster.

- **NodePort**

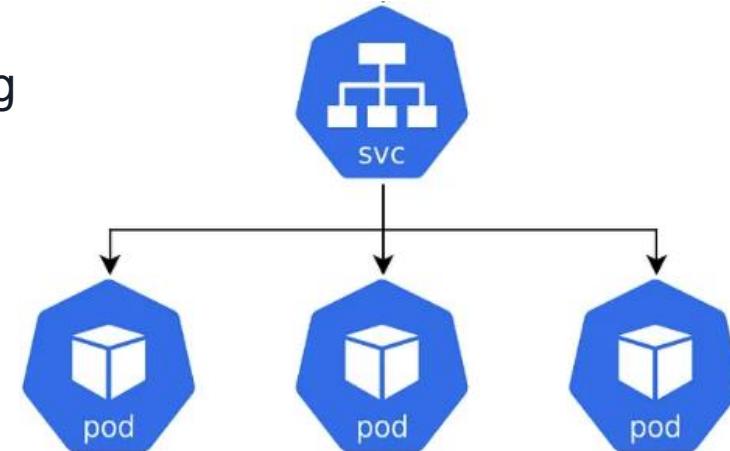
Service exposes the Service on the same port of each selected Node in the cluster using NAT. It makes the Service reachable from outside the cluster using <NodeIP>:<NodePort>.

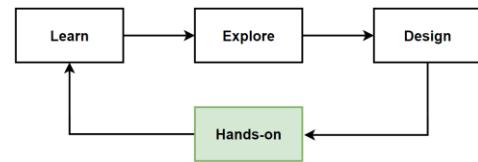
- **LoadBalancer**

Provisions an external load balancer and assigns a fixed, external IP to the Service. It routes external traffic to the backend Pods.

- **ExternalName**

Service maps a Service to a DNS name, rather than to a selector like the other types. It is used to redirect a Service to an external one, outside of the cluster.





Kubernetes Service Types - 2

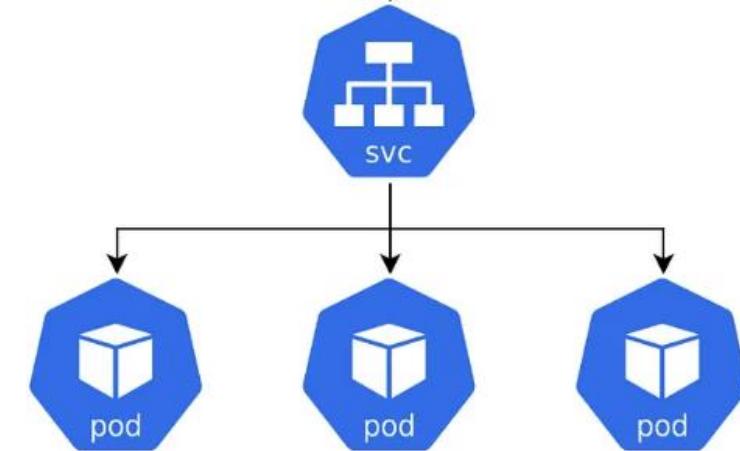
- **Default service type is ClusterIP** which provide to internal Cluster IP and provide to communicate with pods into Kubernetes.

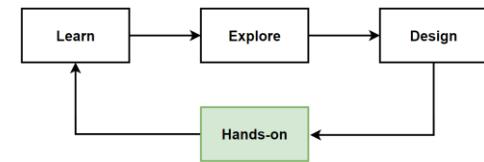
- **NodePort**

NodePort provide to generate reachable **NodeIP** and **NodePort** that we can invoke from outside of the K8s.

- ports:
- - protocol: TCP
- port: 80
- targetPort: 80
- nodePort: 30080
- type: NodePort

- **Expose the Service on the same port (30080) of each selected Node in the cluster and routes external traffic to the backend Pods with the label app: my-app on port 80.**
- Your **application should be configured to listen on 0.0.0.0** and the correct port. Listening on localhost or 127.0.0.1 inside the container will not allow external connections.

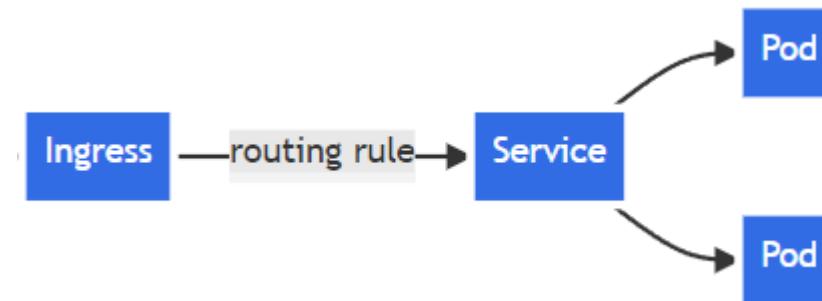


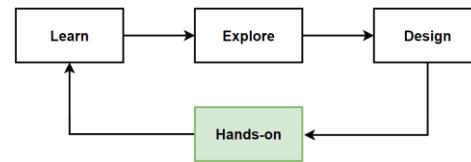


Create Ingress for External Access of Microservice

- When exposing a web application to the internet, you should use an **Ingress**.
- **Kubernetes documentation of Ingress**

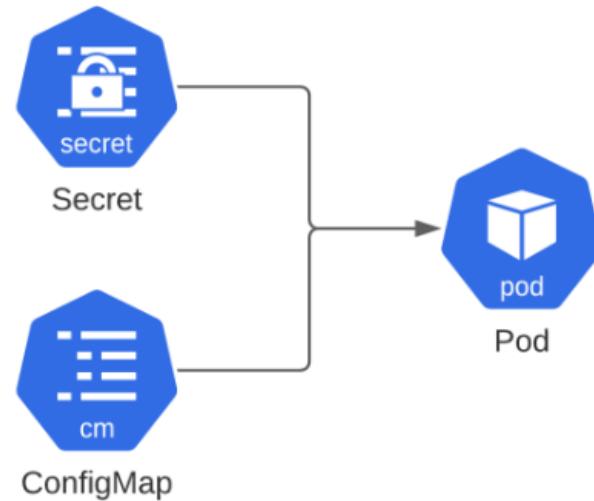
<https://kubernetes.io/docs/concepts/services-networking/ingress/#what-is-ingress>

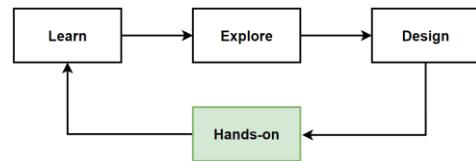




Create ConfigMaps and Secrets for Microservice

- Create **ConfigMaps** and **Secrets** to store configuration data and sensitive information for our container deployment in Kubernetes.
- **Decouple configurations with ConfigMaps and Secerts**
ConfigMap is a Kubernetes object that decouples configuration data from pod definitions. Kubernetes secrets are similar, but designed to decouple sensitive information.
- Provides **better organization** and security of your Kubernetes environment.
- Enables you to **share the same configuration** and **secrets** across multiple pods and deployments
- **Best practice** that can help to improve the scalability, security, and maintainability of your cluster.
- **Create the ConfigMap**
Can be used in one of two ways; as environment variables or volumes. We'll use a ConfigMap to create environment variables inside the pod; LOG_LEVEL.





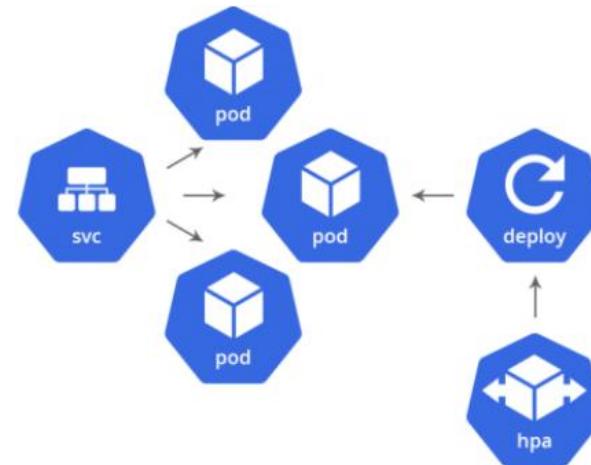
Scale Container Instance in Kubernetes

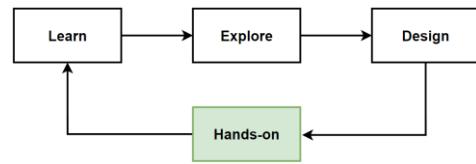
Scaling in Kubernetes

- Kubernetes allows scaling of Pods and Nodes, providing hardware and software scalability.
- Improves hardware utilization by scheduling Pods on Nodes with available resources.
- Utilizes virtualization, cloud hosting, or physical machines to expand the number of Nodes.
- Scaling is driven by resource utilization and can be triggered by other sources with Kubernetes Event-driven Autoscaling (KEDA).

Scaling Pods

- Pods can be scaled to handle varying workloads.
- Define the number of replicas for a Pod in the Deployment configuration.
- Manual scaling:
 - Update the deployment configuration file with the desired number of replicas.
 - spec:
 - replicas: 5
- Use the kubectl scale command to update the deployment with the desired number of Pods.
 - `kubectl scale --replicas=5 deployment/product-deploy`





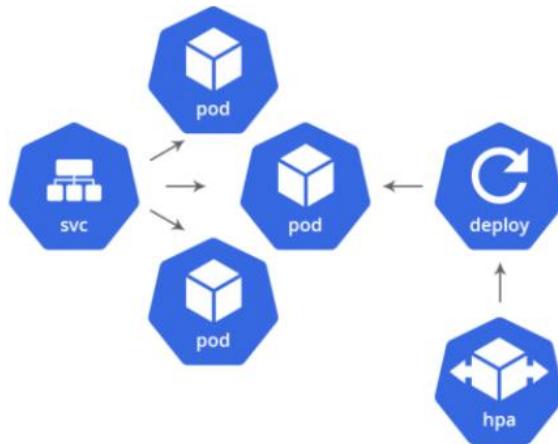
Auto-Scaling in Kubernetes

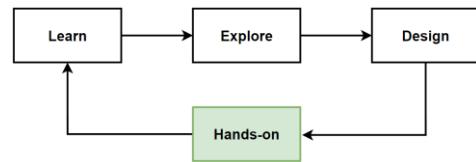
HPA - Horizontal Pod Autoscaler

- Kubernetes uses the HPA to monitor the resource demand and automatically scale the number of pods.
- HPA checks the Metrics API every 15 seconds for any required changes in replica count, and the Metrics API retrieves data from the Kubelet every 60 seconds.

KEDA - Scaling on Different Events

- CPU and memory utilization are the primary drivers for the HPA, but those might not be the best measures
- Scaling triggers and one of the more common plugins to help with that is the Kubernetes Event-driven Autoscaling (KEDA) project.
- The KEDA project makes it easy to plug in different event sources to help drive scaling.





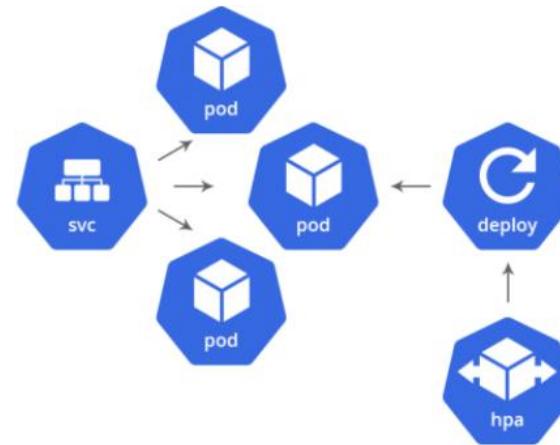
What we have done ? – Kubernetes Deployment

Container orchestrator

- Kubernetes is a portable, extensible open-source platform for managing and orchestrating containerized workloads.
- Kubernetes abstracts away complex container management tasks, and provides you with declarative configuration to orchestrate containers in different computing environments.

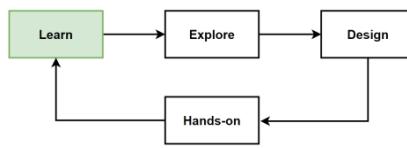
Kubernetes Deployment

- Containerized .net microservices and deploy it into a Kubernetes environment.
- Pushed the Docker images to Docker Hub, to make the images available to the Kubernetes instance to download.
- Created deployment files to declaratively describe what Kubernetes should do to each microservice.
- Learned how to scale a containerized microservice using Kubernetes.



Helm Charts - Managing Kubernetes Applications with Helm

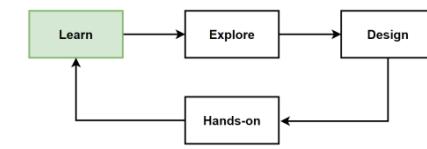
Getting Started with Helm in Kubernetes



Helm: Managing Kubernetes Applications with Helm

- What is Helm in Kubernetes ?
- Package Manager and Helm Charts
- Helm in Kubernetes Use Cases and Advantages
- Why use Helm Charts in Kubernetes for Microservices Deployments
- Best Practices of using Helm Charts with Kubernetes



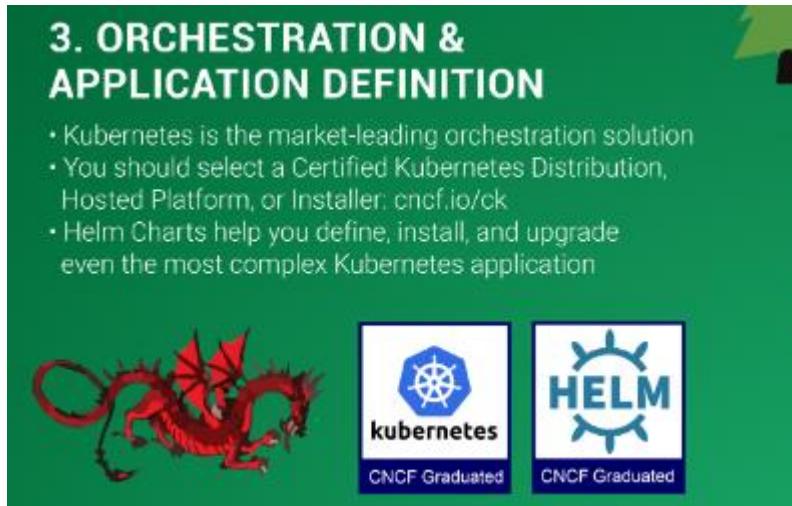
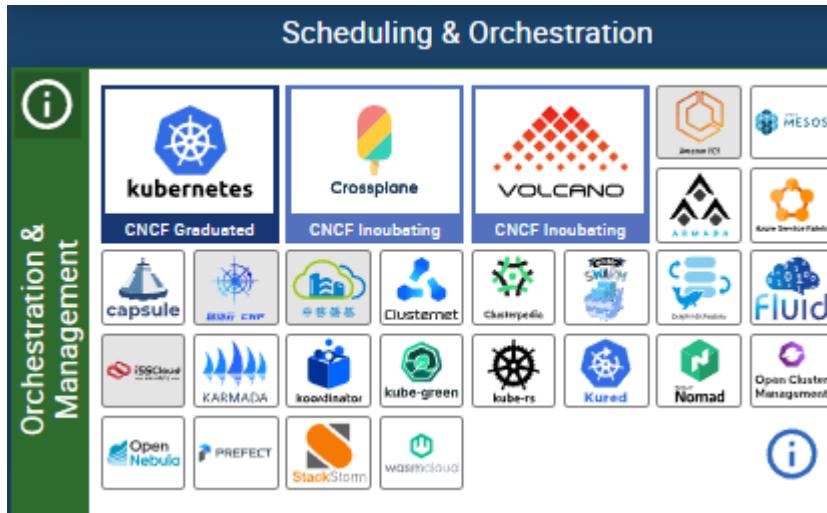


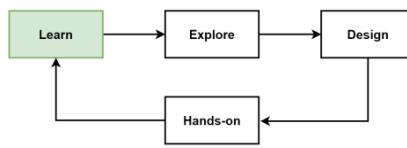
Cloud-Native Landscape and Trial Map

Where Helm Charts

- Cloud-Native Landscape
<https://landscape.cncf.io/>

- Cloud-Native Trial Map
<https://github.com/cncf/trailmap>





What is Helm and Helm Charts ?

Helm and Helm Charts

- Helm is a package manager for K8s that simplifies the deployment, scaling, and management of apps.
- Helm uses a packaging format called Helm Charts to define, version, and share K8s app
- Helm Charts are a collection of files that describe the app's K8s resources, such as deployments, services, config maps, and secrets.



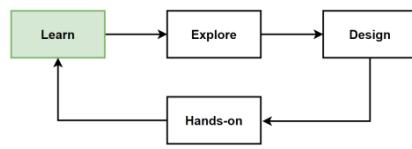
Single Microservice Requires Lots of Config yaml files

- Write lots of k8s manifest files like deployment, service, configmap, secret, ingress yaml files only for 1 single microservices.
- What if we have tens of microservices and how can we manage all these yaml files when we deploys lots of microservices ?
- How can we manage all configurations if we would like to change any configuration like replicaSet count in deployments ?

```

    <ul style="list-style-type: none;">
        <li>✓ lecture145
        <li>✓ k8s
            <ul style="list-style-type: none;">
                <li>! product-deploy.yaml
                <li>! product-pod.yaml
                <li>! product-service.yaml
                <li>! product.yaml
            </ul>
        </li>
    </ul>

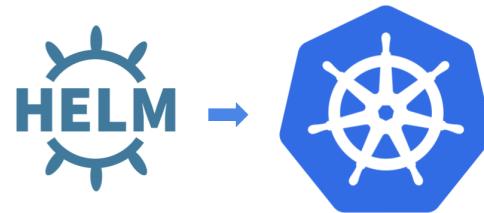
```



Solution: Using Helm Charts

Kubernetes doesn't care about our app as a whole

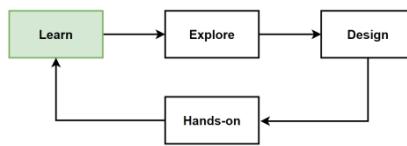
- We declared various objects and it proceeds this yaml files to make each of them exist in K8s clusters.
- K8s doesn't organize and really don't know about the deployment, service that connected each other and represent a single microservices or not.
- Helm is provide to organize these yaml files with application manner its called package manager for Kubernetes.



Helm Packages organize yaml as a group

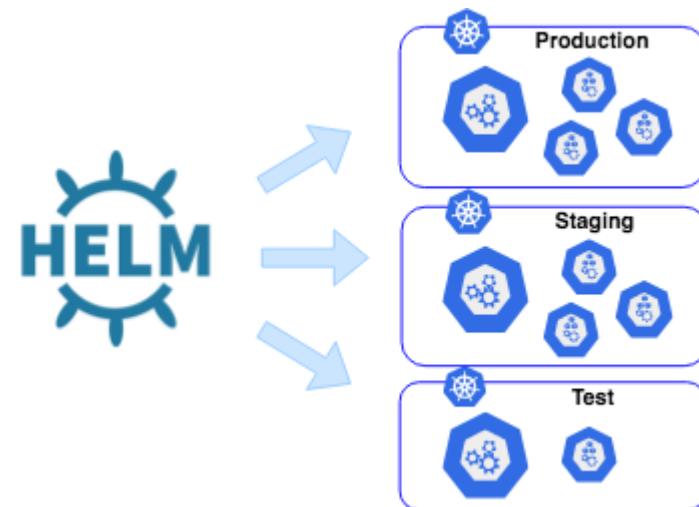
- Whenever we need to perform an action we don't tell how the objects that it should touch.
- Just tell it what package we want to act on like our wordpress app package and based on the package name it then knows what objects it should change and how.

▽ lecture145
▽ k8s
! product-deploy.yaml
! product-pod.yaml
! product-service.yaml
! product.yaml

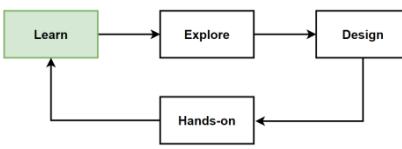


Helm Charts with Kubernetes for Microservices Deploys

- **Helm** is a **package manager** for **Kubernetes** that makes it easier to manage and deploy applications in a Kubernetes cluster.
- **Helm uses a packaging format** called **charts**, which are **collections of files** that describe the various components of an application (such as pods, services, and deployments)
- It allows you to **automate the deployment of complex applications** in a **Kubernetes cluster**.
- Instead of manually creating and managing each component of your application separately, can use a **single Helm chart** to define all of the components and their relationships.
- This makes it **much easier to manage** and **update** your **application** over time.



Benefits of Using Helm Charts in Kubernetes for Microservices Deployments



- **Easy management**

Shared and reused across multiple clusters, making it easy to deploy the same application in different environments.

- **Reusability and Templating**

Parameterized with templates, enabling reuse of the same chart for different configurations and environments, promoting consistency and reducing duplication

- **Versioning and release management**

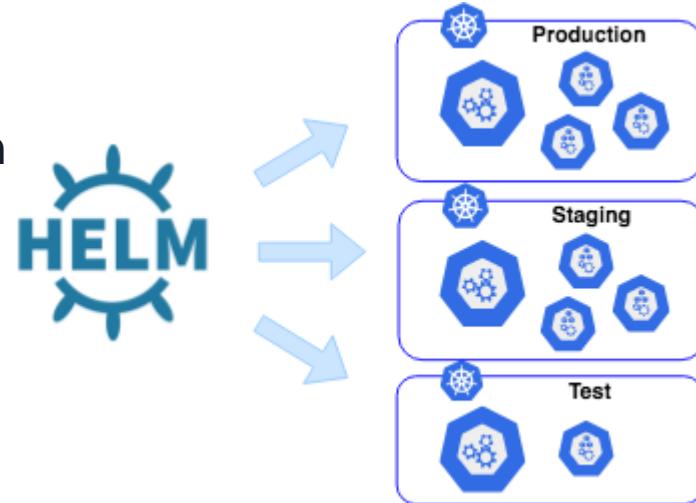
Allows versioning and easy management of application releases, including the ability to roll back to previous versions, facilitating testing and managing different versions in a Kubernetes environment.

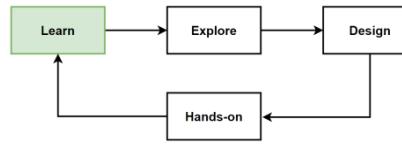
- **Centralized repository**

Stored in a centralized repository, facilitating the discovery, sharing, and deployment of microservices across an organization, ensuring collaboration and consistency.

- **Sharing and collaboration**

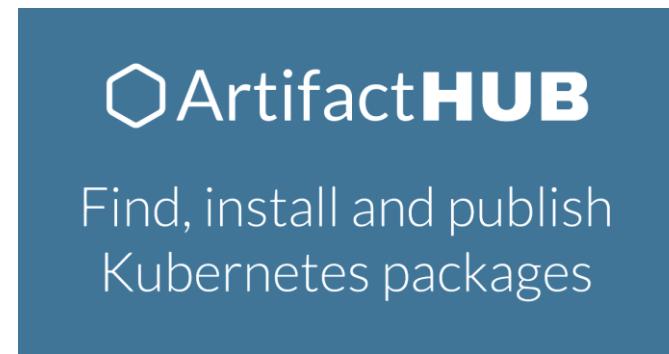
Shared through repositories, enabling teams to collaborate on development and deployment, encouraging the use of best practices and maintaining consistency across deployments.

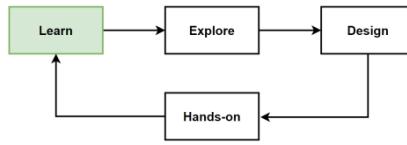




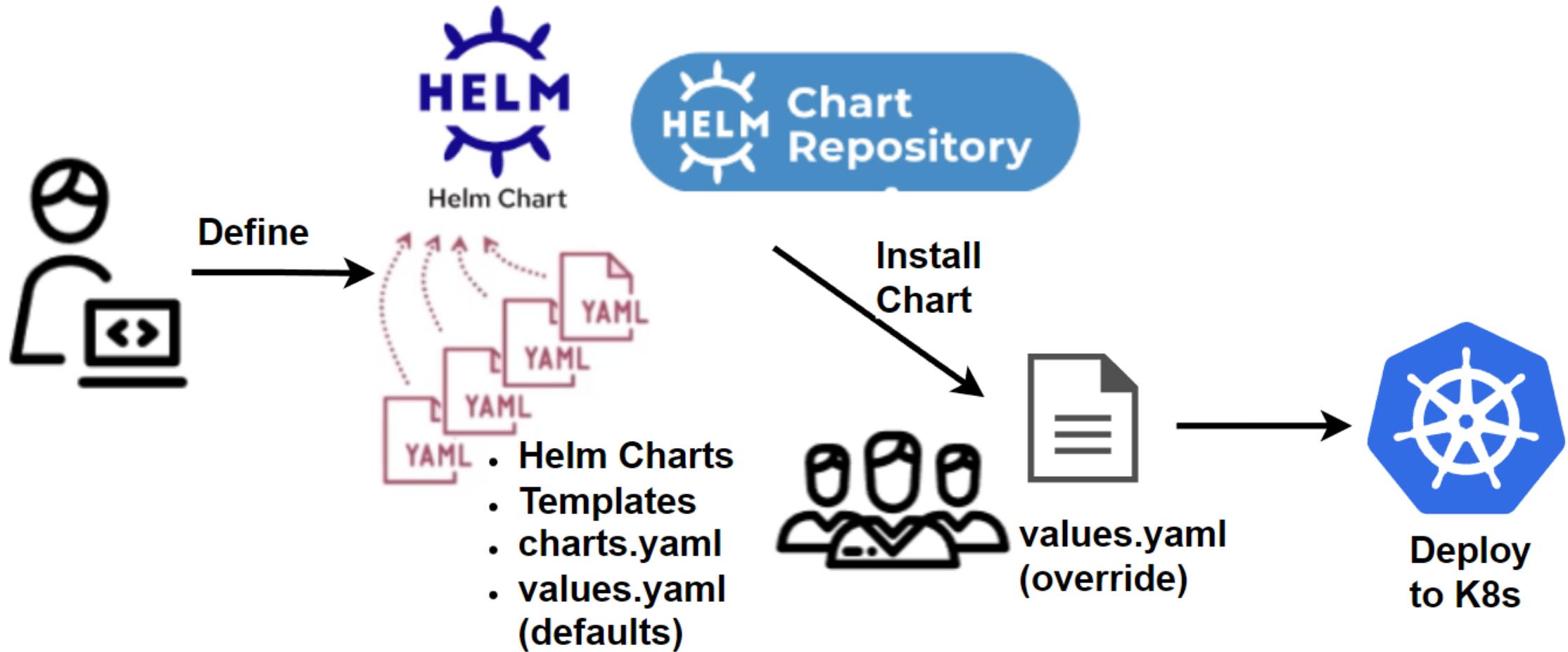
What is Artifact Hub for Helm Charts ?

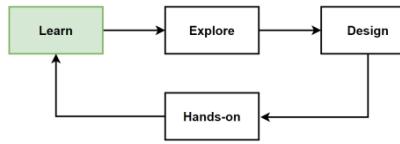
- **Artifact Hub** is a **centralized platform** where you can discover, share, and deploy Kubernetes-native applications and components.
- It serves as a **registry** for **Helm charts**, **Operators**, and other **Kubernetes resources**.
- **Artifact Hub** is maintained by the Cloud Native Computing Foundation (CNCF) and serves as an **official platform** for the Kubernetes community to find and distribute artifacts.
- Users can search for **Kubernetes-native artifacts** such as **Helm charts**, **Operators**, **Custom Resource Definitions (CRDs)**, and Falco rules.
- Artifact Hub **provides information** about the **artifacts**, their maintainers, and the associated documentation.
- <https://artifacthub.io/>
- search prometheus





How Helm Charts works ?





Understanding Helm Chart Structure

- **Chart.yaml**

Main metadata file that contains information about the chart, such as its name, version, and description.

- **values.yaml**

Contains the default configuration values for the chart. Users can override these values when installing or upgrading the chart.

- **charts/**

Optional directory contains any dependent Helm charts. If your chart relies on other charts, they should be placed in this folder.

- **templates/**

Contains the Kubernetes manifest templates. Helm uses the Go template engine to render these templates with the values from the values.yaml file and user-provided values.

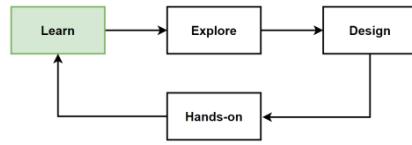
- **deployment.yaml, service.yaml, ingress.yaml**

Examples of Kubernetes manifest templates for a typical application. You can add more templates as needed.

```

WEBAPP
  charts
    templates
      tests
      _helpers.tpl
      deployment.yaml
      ingress.yaml
      NOTES.txt
      service.yaml
      .helmignore
      Chart.yaml
      values.yaml

```



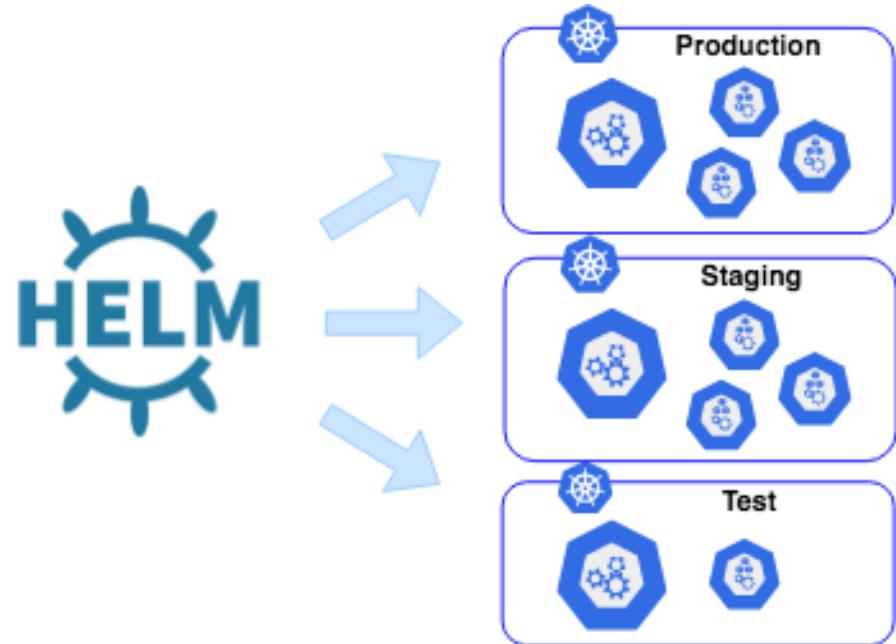
Install Helm

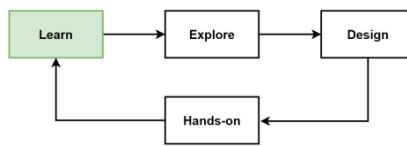
- **Install Helm**

Download and install the Helm CLI

- <https://helm.sh/docs/intro/install/>
- <https://github.com/helm/helm/releases>

- Download and un-zip
- Create C:/tools folder
- Add path into ENV variables





How to use Helm ?

- **Create or obtain a Helm Chart**

Create your own Helm Chart for your application or use an existing chart from a public or private chart repository.

- **Customize the Chart**

Customize the chart by editing its values.yaml file to provide specific configuration values for your deployment.

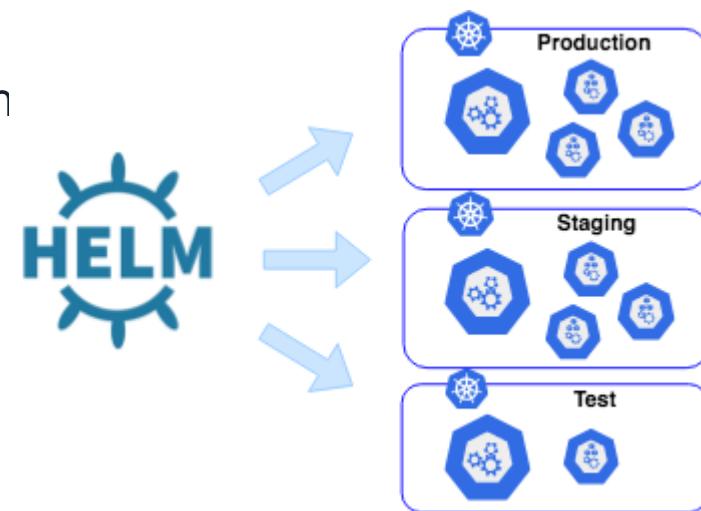
- **Deploy the Chart**

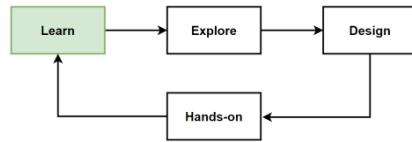
Use the helm install command to deploy the chart to your Kubernetes cluster:

- `helm install <release-name> <chart-path> --values <values-file-path>`

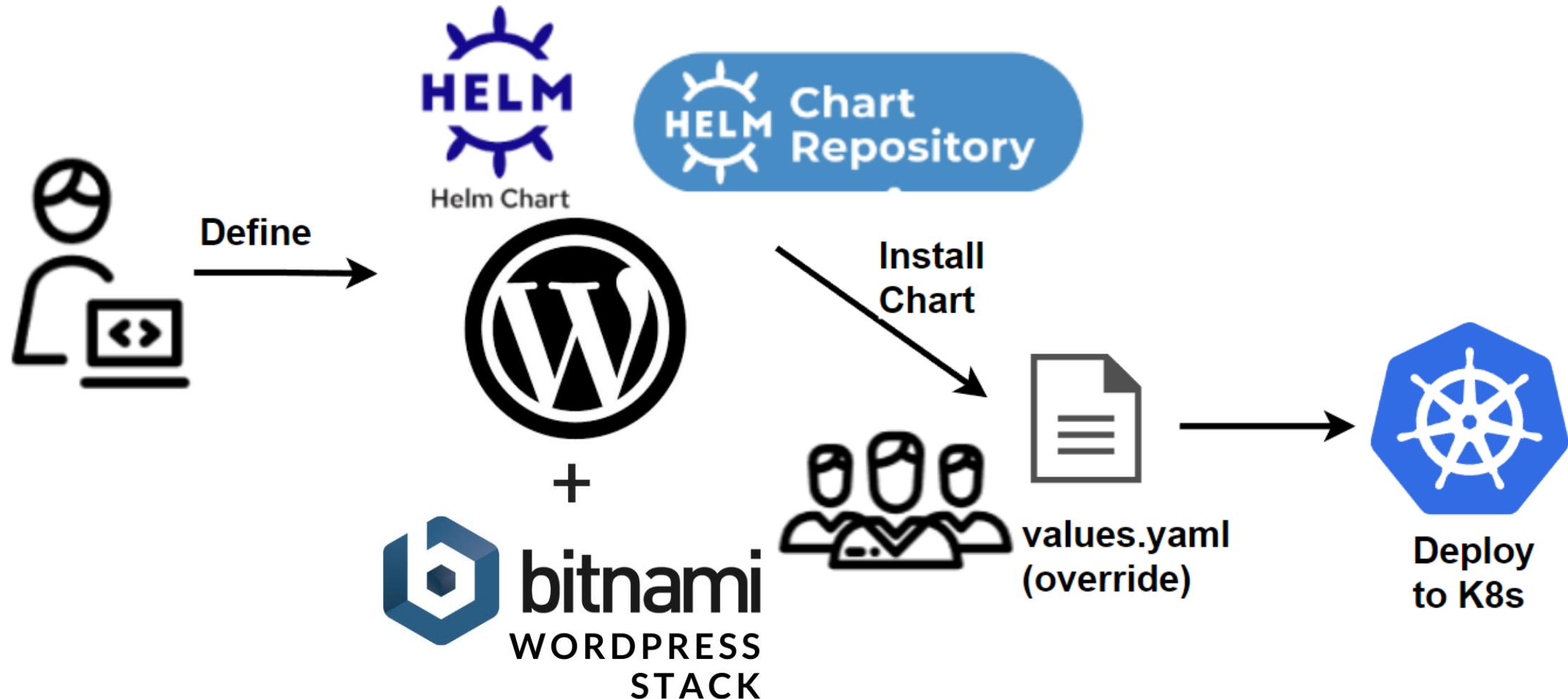
- **Manage your application**

Helm provides commands to manage your application, such as upgrading, rolling back, or uninstalling the application.

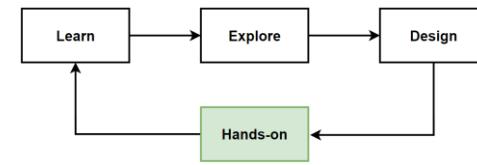




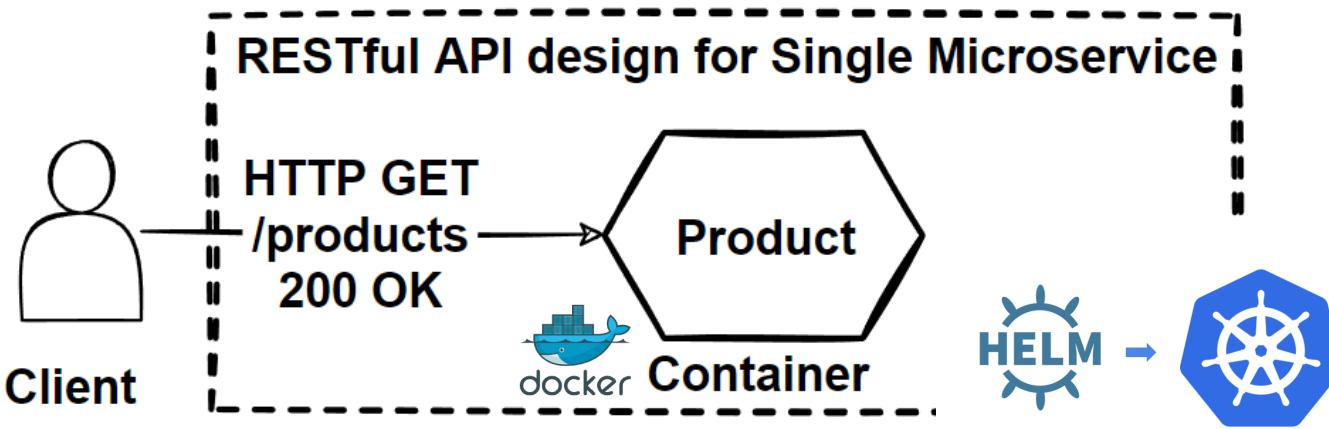
Install Wordpress Helm Chart from ArtifactHub



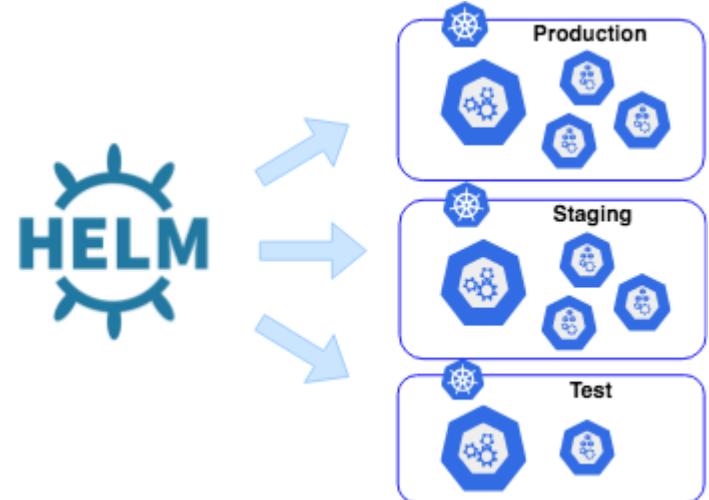
Hands-on: Deploy Microservices to Kubernetes with Helm Charts

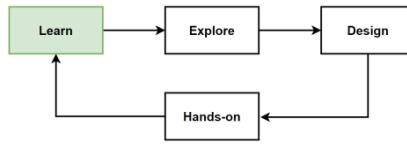


- Setting Up a Kubernetes Cluster
- Create helm chart for microservices
- Deploying Microservices on Kubernetes with helm
- Deploying, Scaling, and Rolling Updates with Deployments
- Service Exposure and Ingress Controllers

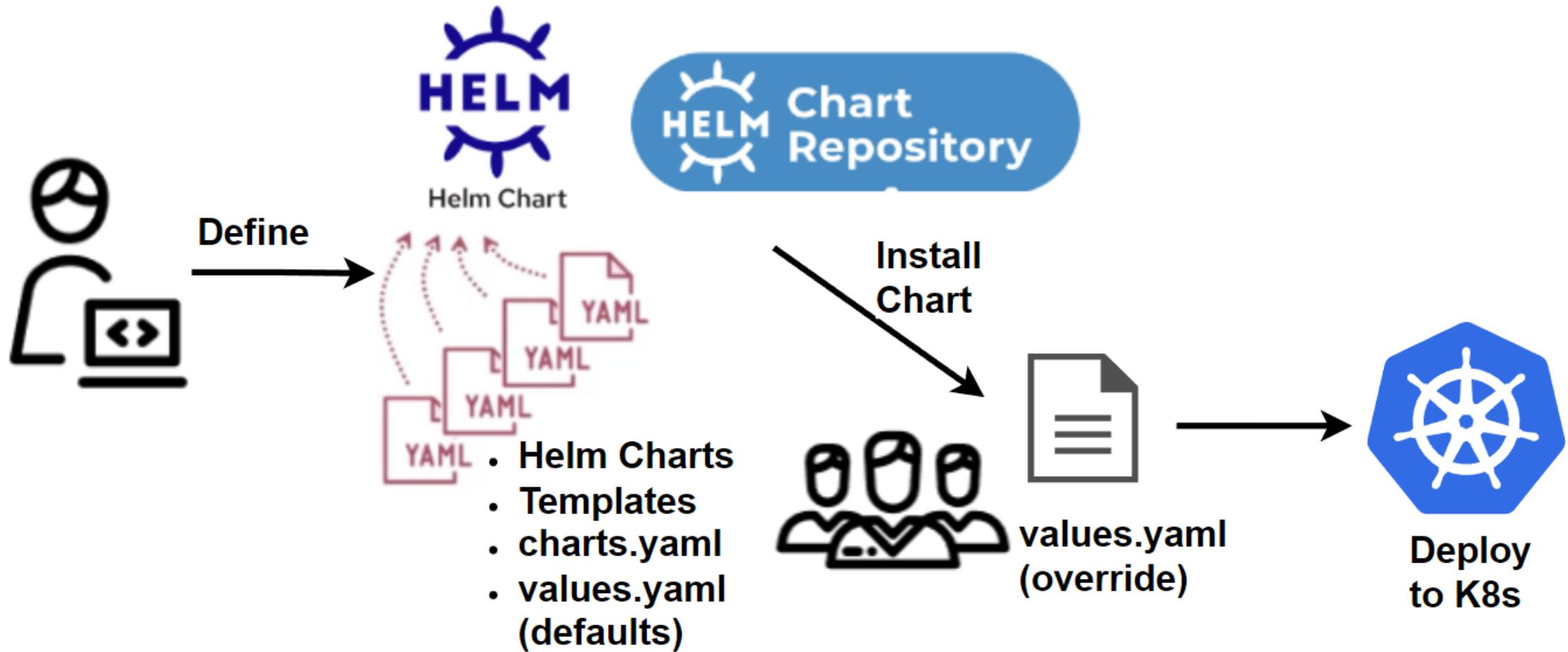


GET /api/products: Retrieves all products.
GET /api/products/{id}: Retrieves a specific product by ID.
POST /api/products: Adds a new product.
PUT /api/products/{id}: Updates an existing product by ID.
DELETE /api/products/{id}: Deletes a specific product by ID.





How Helm Charts works ?



Cloud-Native Pillar4: Communications

What are Cloud-Native Communications ?

How microservices communicate in Cloud-Native environments ?

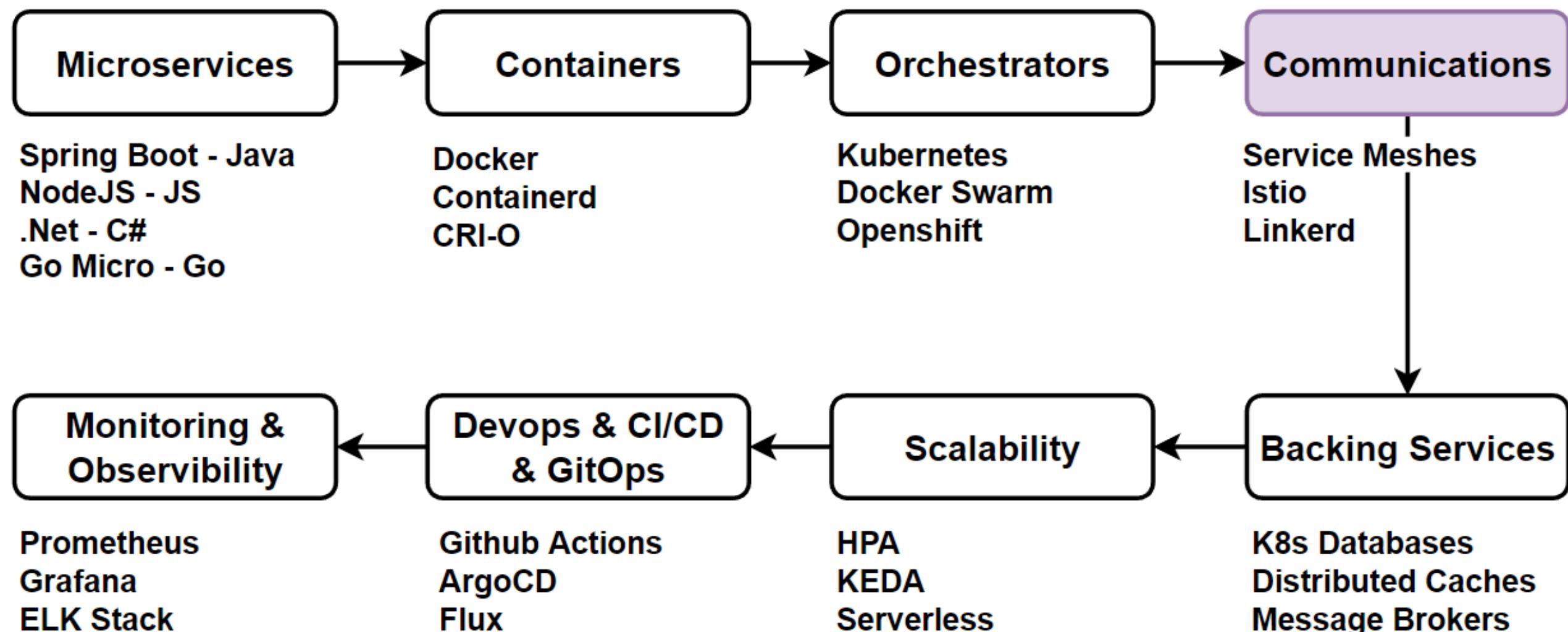
What are patterns & best practices of communications in CN environments ?

API Gateways, Sidecar and Service Mesh Pattern

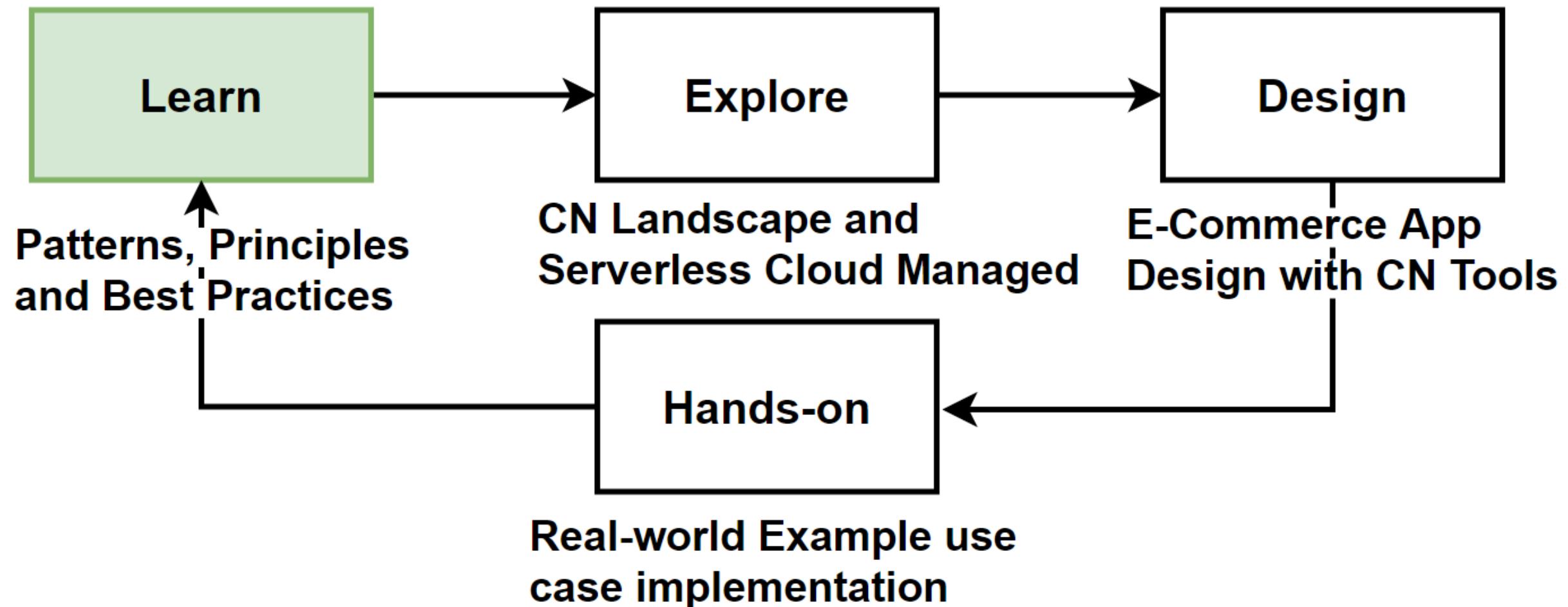
Design & Implement our E-Commerce application with Communication Tools

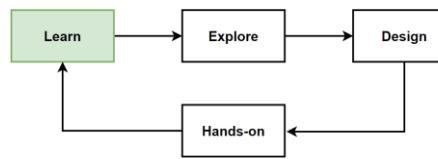
Mehmet Ozkaya

Cloud-Native Pillars Map – The Course Section Map



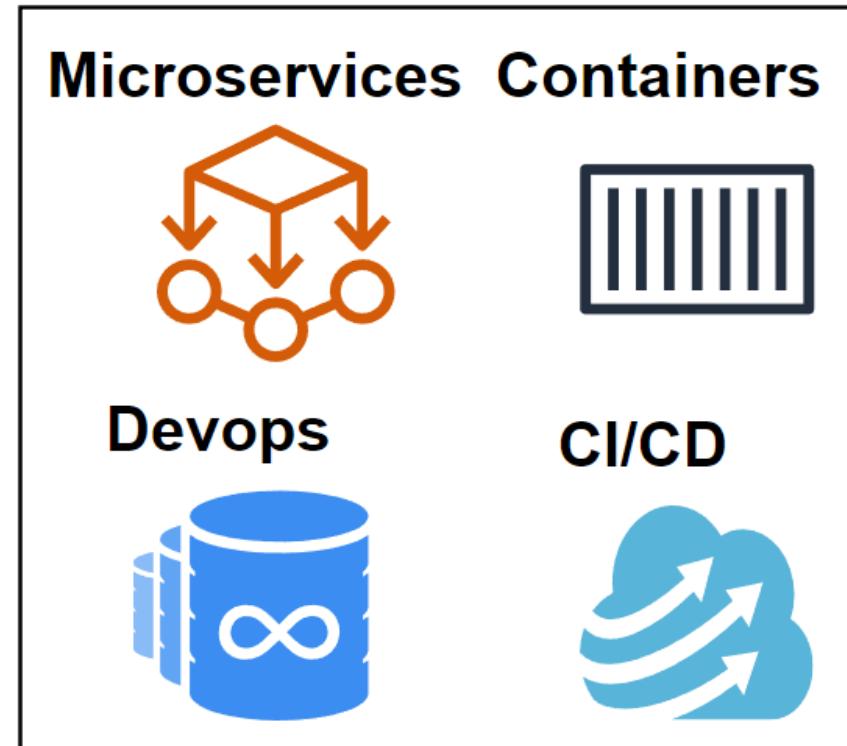
Way of Learning – The Course Flow

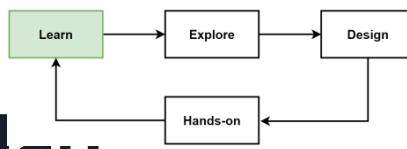




Learn: The 4. Pillar – Cloud-Native Communications

- Cloud-Native Communication
 - Microservices Communication Types: Synchronous or Asynchronous Communication
 - Microservices Communication Styles: Request-Driven or Event-Driven Architecture
- Communication Challenges in Cloud-Native Systems
- Front-End and Back-End Communication
- Back-End Microservices Communication
 - RESTful APIs for Microservices
 - gRPC Usage in Microservices Communication
- Communication Patterns and Best Practices
 - API Gateway Pattern
 - Sidecar Pattern
 - Service Mesh Infrastructure
- Explore Cloud-Native Tools
 - Envoy
 - Istio
- Related cloud-native patterns, principles and best practices.





Where «Communications» in 12-Factor App and CN Trial Map

Twelve-Factor App

- Codebase
- Dependencies
- Config
- Backing Services
- Build, release and run
- Processes
- Port Binding
- Concurrency
- Disposability
- Development/Prod Parity
- Logs
- Admin Processes



CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape <https://github.com/cncf/landscape> has a growing number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator <https://www.cncf.io/training>

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider <http://cncf.io/ksp>

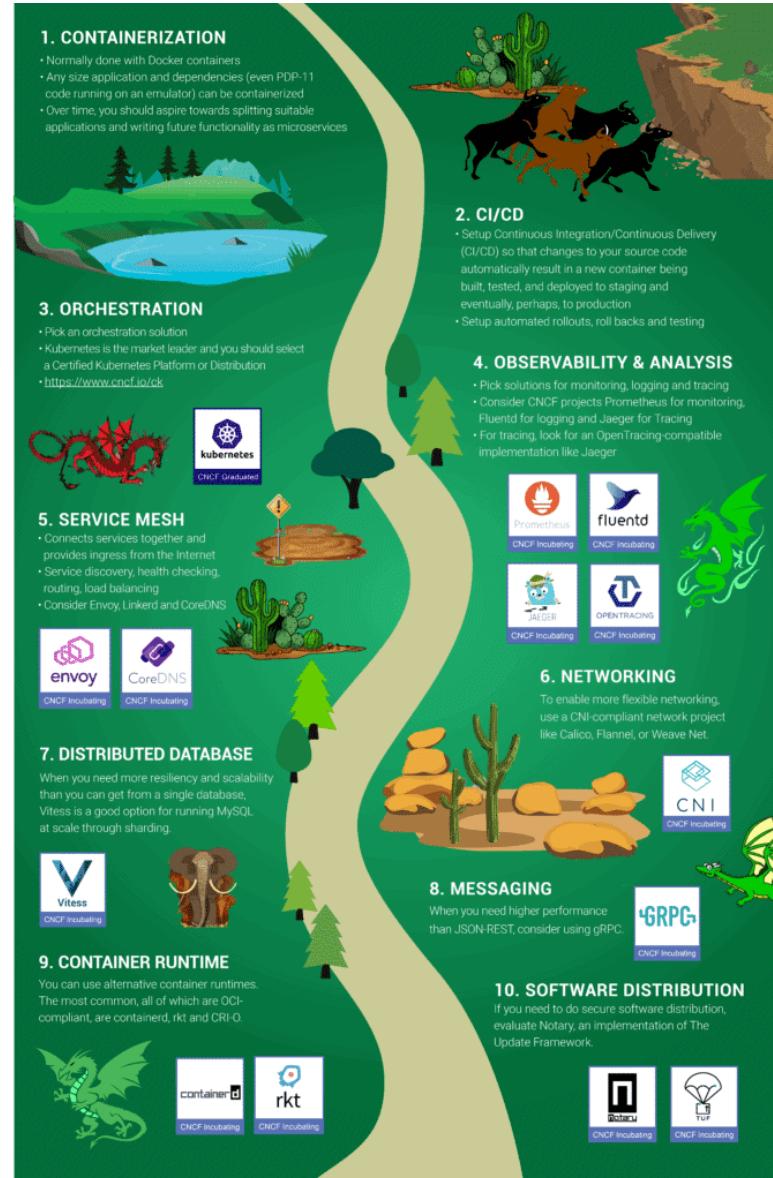
C. Join CNCF's End User Community

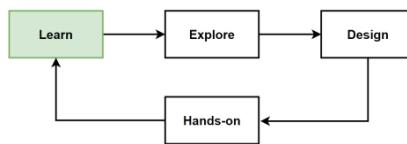
For companies that don't offer cloud native services externally <http://cncf.io/enduser>

WHAT IS CLOUD NATIVE?

- **Operability:** Expose control of application/system lifecycle.
- **Observability:** Provide meaningful signals for observing state, health, and performance.
- **Elasticity:** Grow and shrink to fit in available resources and to meet fluctuating demand.
- **Resilience:** Fast automatic recovery from failures.
- **Agility:** Fast deployment, iteration, and reconfiguration.

www.cncf.io
info@cncf.io





12-Factor App – Cloud-Native Communications

Processes

- Emphasizes that applications should be stateless and share nothing. Encourages apps to use external services for sharing state, which often involves communication between services.

Port binding

- Enables apps to be deployed and scaled independently. By binding to a specific port, an app can expose its functionality as a service, allowing other services to communicate with it without being tightly coupled.

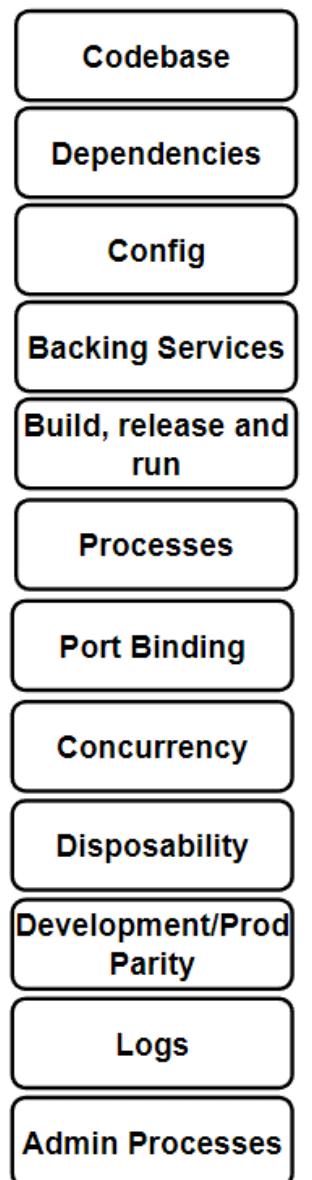
Concurrency

- Promotes scaling applications by running multiple processes concurrently. Communication and coordination between these processes and services become critical.

Disposability

- Fast startup and graceful shutdown for app processes. Services may need to communicate with other components during startup or shutdown to coordinate their actions or update their status.

Twelve-Factor App



Cloud-native Trial Map – Communications

Service Mesh

- Dedicated infrastructure layer handles service-to-service communication in a cloud-native env. It provides features like load balancing, service discovery, observability, and security.

Service Discovery

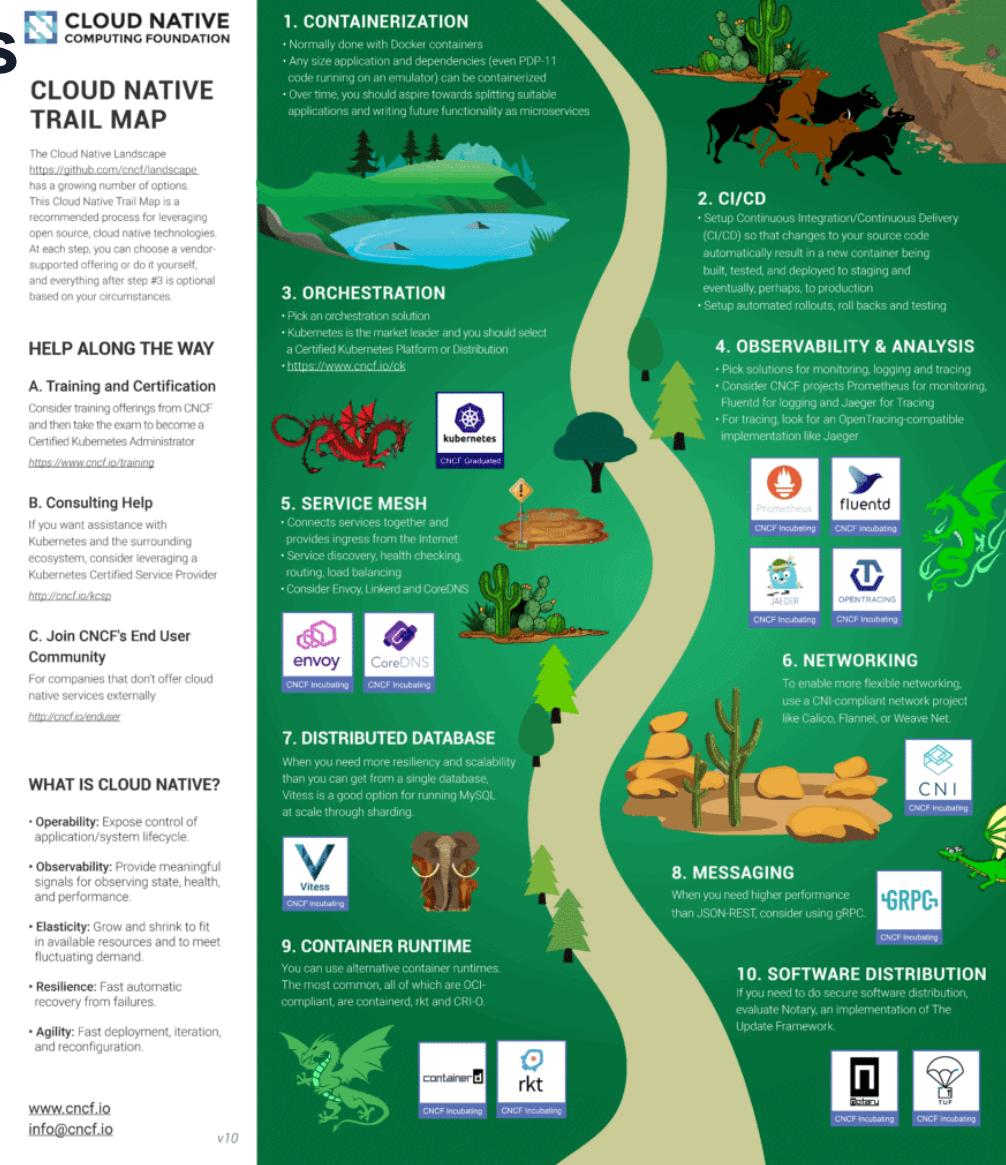
- Services need to dynamically discover and communicate with each other. Help to locate and connect to other services within the system.

API Gateway

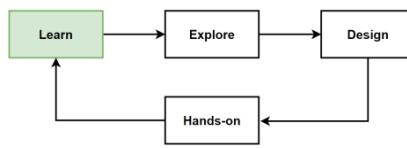
- Entry point for external requests to the microservices. It routes the requests to appropriate services, handles load balancing, and provides additional functionalities like auth and rate limiting.

Messaging Systems

- Async communication between microservices by using message queues that decouples services, allows to evolve independently and ensures resilience against temporary service failures.



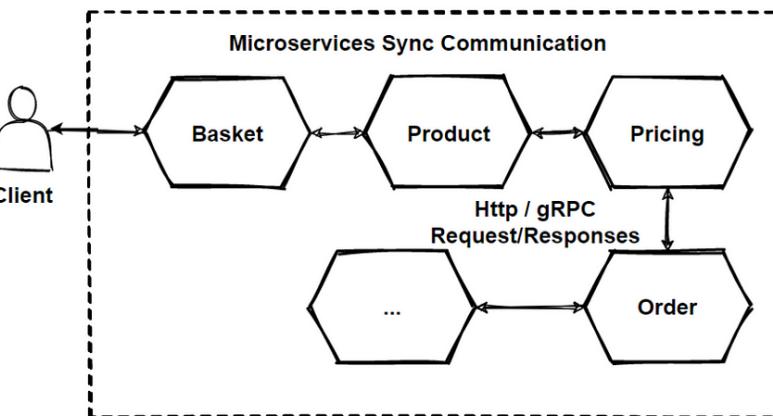
<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>



Communications in Cloud-Native Architectures

- When developing a cloud-native application, communication becomes an **crucial role** in architecture.

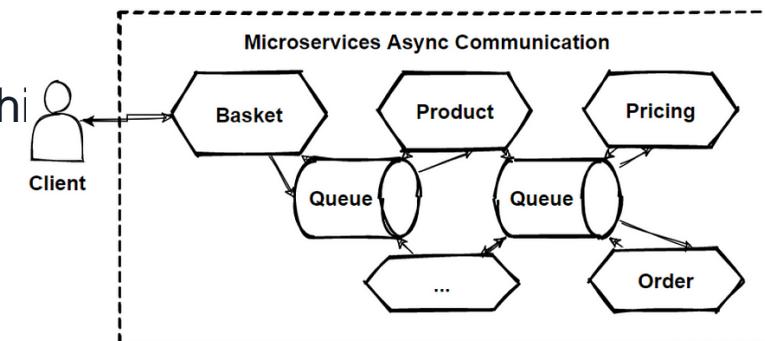
- How front-end client applications communicate with back-end microservices ?
- Should allow direct communication? Or abstract the back-end services with api gateways ?
- How back-end microservices communicate with each other ?
- Should allow direct HTTP calls that lead to coupling and impact performance and agility? Or might we consider decoupled messaging with queue and topic technologies?

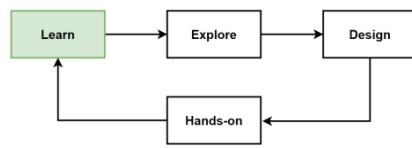


Monolithic vs. Cloud-Native Communication

In monolithic applications, communication is simple, as code modules execute together in the same process on a server.

- Cloud-native systems **adopt a microservice-based architecture**, with independent microservices running in separate processes and containers within a cluster.
- This shift brings numerous advantages but also **introduces complexity and challenges in communication**.





Communication Challenges in Cloud-Native Systems

Replacing local in-process calls with network calls adds several complexities

- Network congestion, latency, and transient faults become concerns.
- Resiliency and retrying failed requests are essential.
- Idempotent calls are needed to maintain consistent state.
- Authentication and authorization must be implemented.

Front-End and Back-End Communication

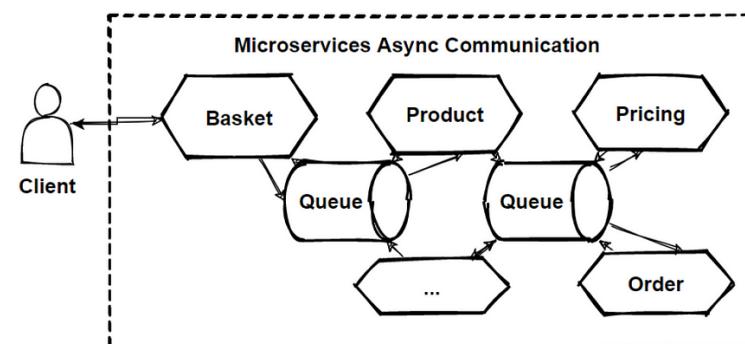
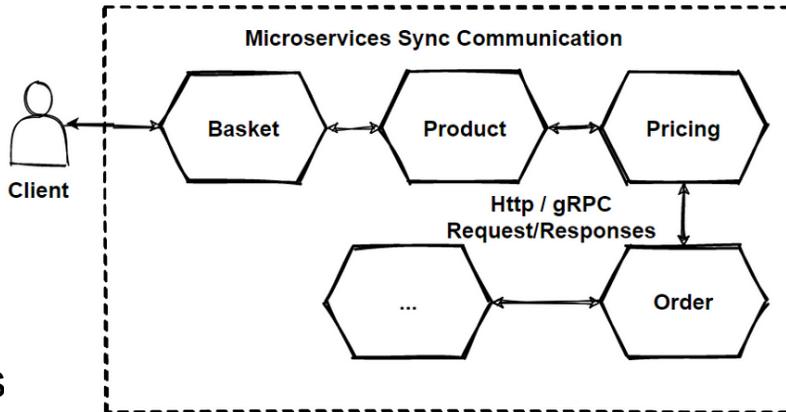
- Front-end applications typically communicate with back-end microservices using APIs and messaging technologies.
- AWS and Azure cloud provides various backing services to support cloud-native communication like Serverless API Gateways.

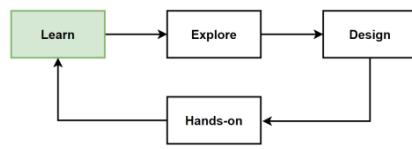
Back-End Microservices Communication

- Back-end microservices communicate with each other using patterns and technologies: RESTful APIs, gRPC, and service mesh.

Communication Patterns and Best Practices

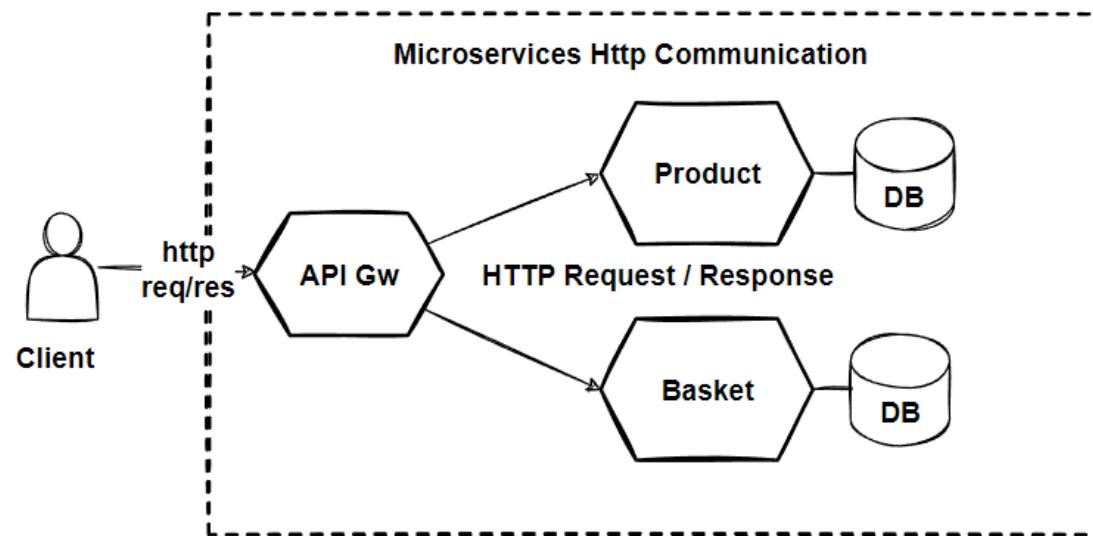
- Request/response, publish/subscribe, and event-driven architectures
- Best practices include using API gateways, implementing circuit breakers, and leveraging service meshes.

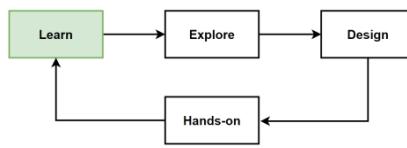




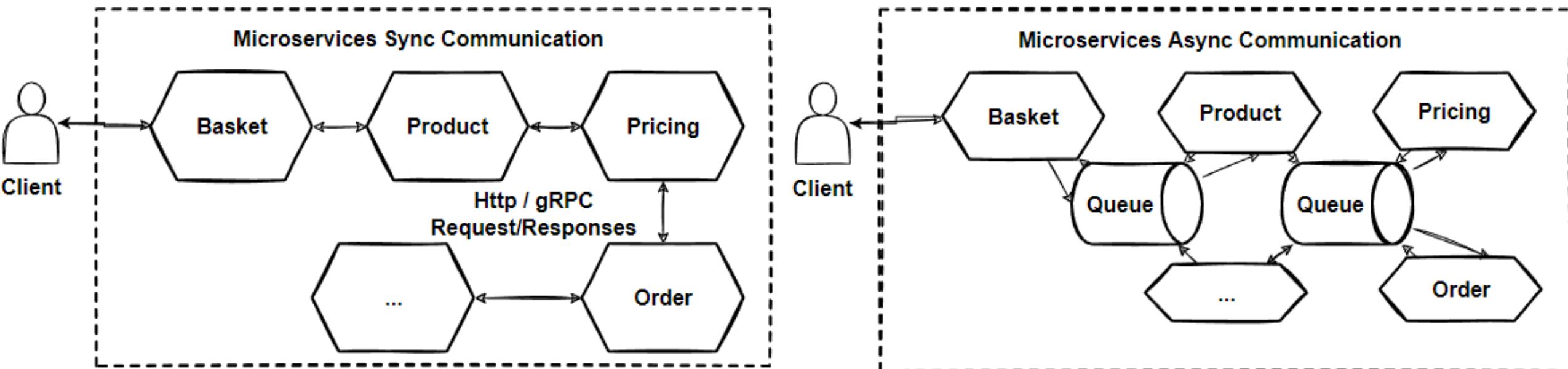
Microservices Communications

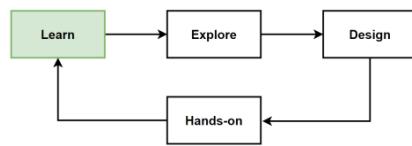
- **Isolate the business** into microservices as much as possible.
- Use **asynchronous communication** between the internal microservices as much as possible.
- Create **well-defined APIs** for **inter-service** communications.
- Monolithic inter-process method calls becomes **well-defined APIs** into Microservices.
- Group some operations and **expose aggregated APIs** that cover several calls from multiple sources.
- **Smart endpoints and dumb pipes:** microservices loosely coupling and expose endpoints with RESTful APIs in order to provide end-to-end use cases.
- Microservices communication types:
 - **Synchronous communications:** Request/Response
 - **Asynchronous communications:** Message broker event buses





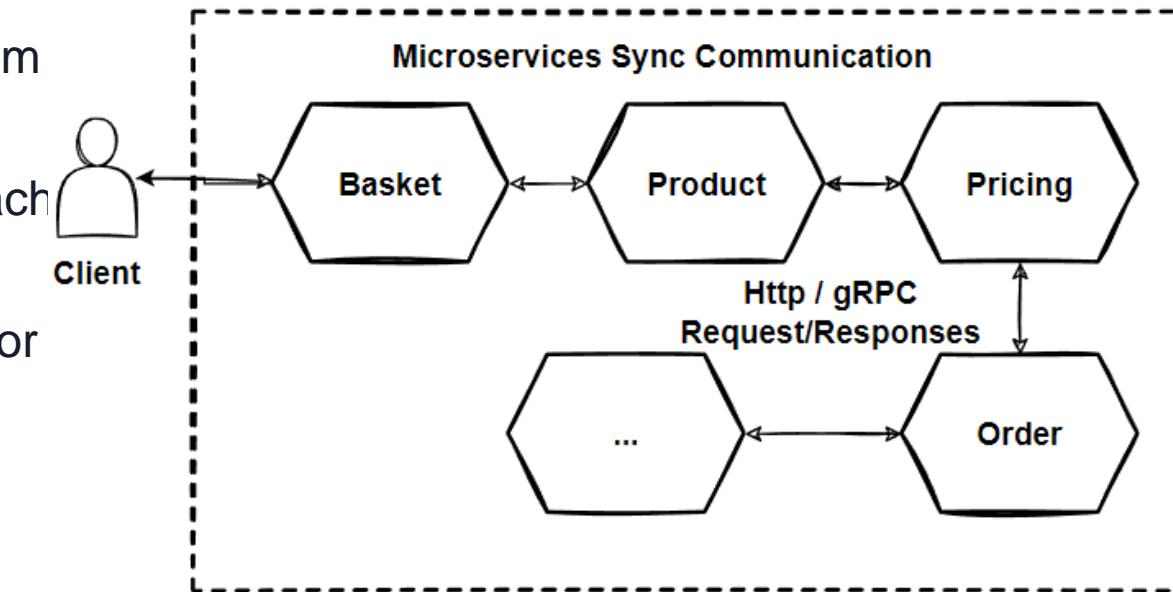
Microservices Communication Types - Sync or Async

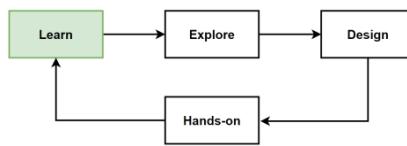




Microservices Synchronous Communication

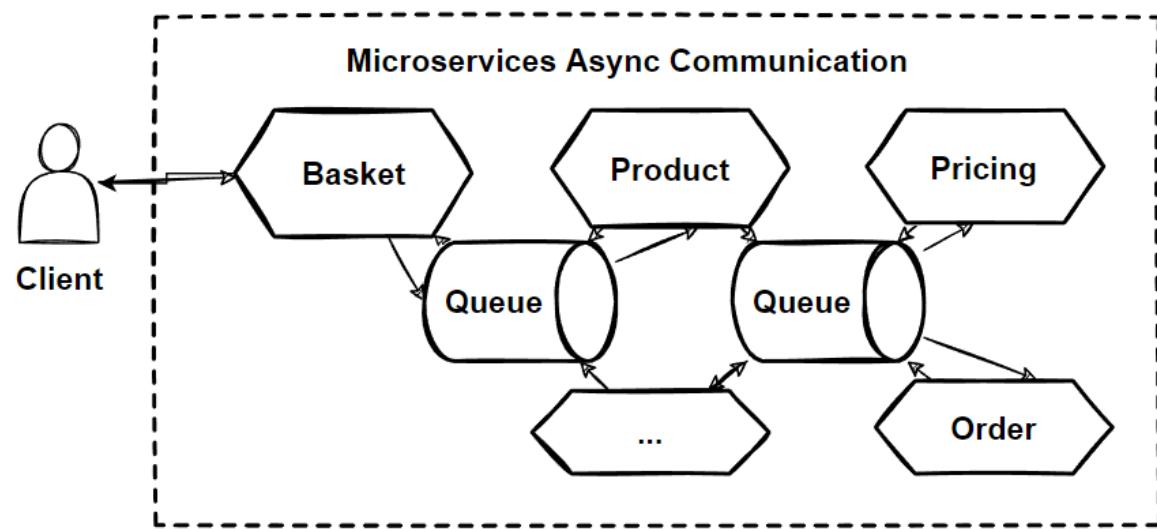
- **Synchronous communication** is using **HTTP** or **gRPC** protocol for returning synchronous response.
- The client **sends a request** and **waits for a response** from the service.
- The client code **block their thread**, until the response reach from the server.
- The synchronous communication protocols can be **HTTP** or **HTTPS**.
- The client sends a request with using **http protocols** and waits for a response from the service.
- The client **call the server** and **block** client their operations.
- The client code will **continue** its task when it **receives** the HTTP server **response**.

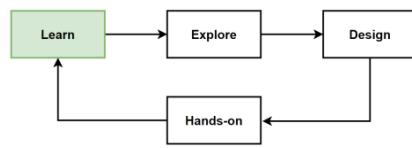




Microservices Asynchronous Communication

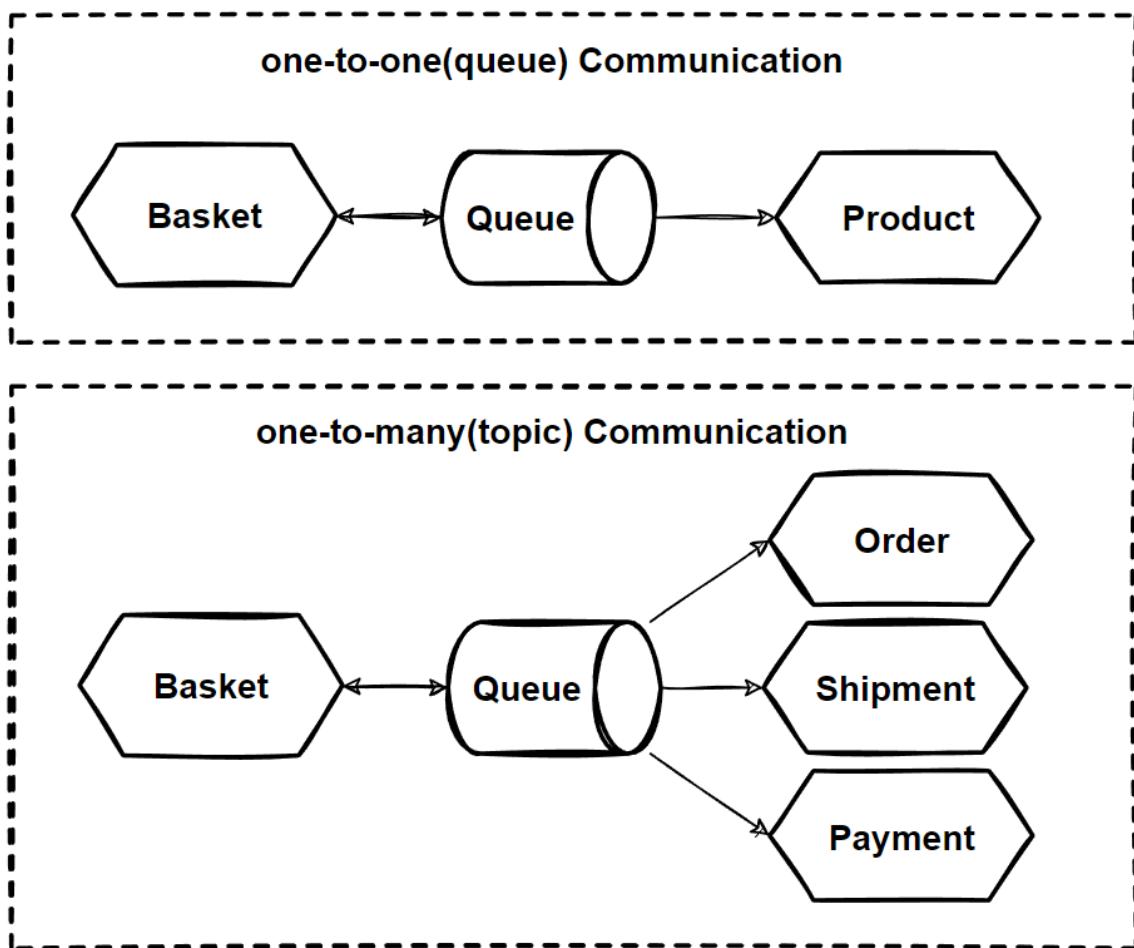
- The client sends a request but it **doesn't wait for a response** from the service.
- The client **should not have blocked** a thread while waiting for a response.
- AMQP** (Advanced Message Queuing Protocol)
- Using AMQP protocols, the client **sends the message** with using message broker systems like **Kafka** and **RabbitMQ** queue.
- The message **producer does not wait** for a **response**.
- Message consume from the **subscriber** systems in **async** way, and no one waiting for response **suddenly**.
- Asynchronous** communication also divided by 2:
 - one-to-one(queue)**
 - one-to-many (topic)**

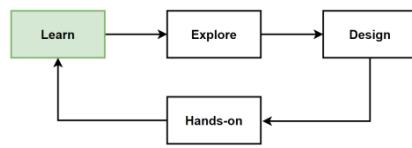




Microservices Asynchronous Communication-2

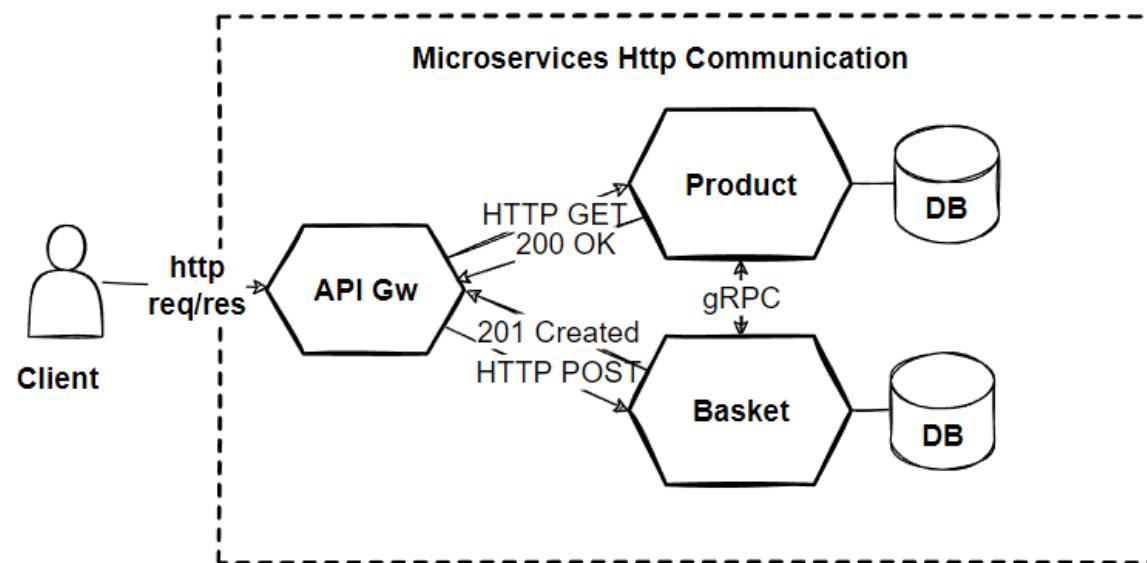
- **one-to-one(queue) implementation**
one-to-one(queue) implementation there is a single producer and single receiver.
- Command Patterns offers to receive one queue object and after that execute the command with incoming message.
This process restarts with receiving new command queue item.
- **one-to-many (topic) implementation**
In one-to-many (topic) implementation has Multiple receivers. Each request can be processed by zero to multiple receivers.
- Event-bus or message broker system is **publishing events** between multiple microservices and communication provide with subscribing these events in an async way.
- **Publish/subscribe** mechanism used in Event-driven microservices architecture.

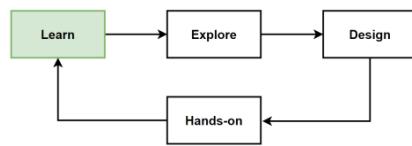




Microservices Communication Styles

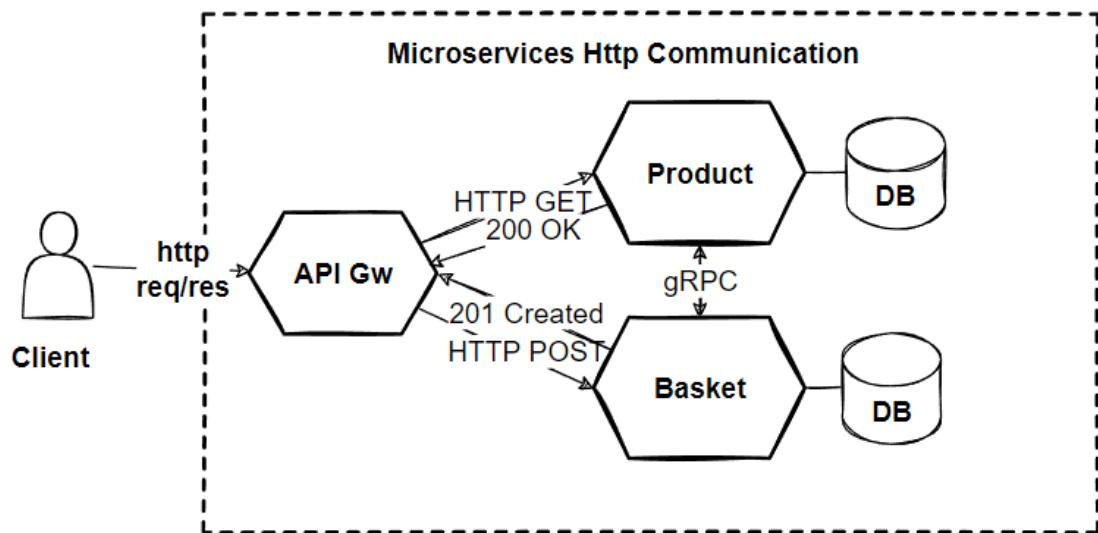
- **Request/response communication** with HTTP and REST Protocol (extends gRPC and GraphQL)
- **Push and real-time communication** based on HTTP, WebSocket Protocol
- **Pull communication** based on HTTP and AMQP (short polling - long polling)
- **Event-Driven communication** with Publish/Subscribe Model

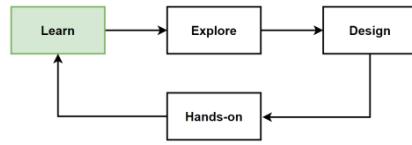




Request/response communication

- When using **synchronous** request/response-based communication mechanism, use **HTTP** and **REST** protocols that are the most common protocols.
- **Expose APIs** from our microservices using the HTTP and REST protocols.
- **REST HTTP calls** using HTTP verbs like GET, POST, and PUT.
- If our communication held between internal microservices, use **gRPC protocol** communication mechanisms to provide high performance and low latency.
- **Use GraphQL** instead of the REST APIs when performing Request/response communication.
- With GraphQL, we can define the **structure of the data** required and get 1 response in 1 request with more flexibility and efficiency way to get whole data.





Push/pull communication

- **Push and real-time communication based on HTTP, WebSocket Protocol**

Use case about real-time and one-to-many communication like chat application, use Push Model with HTTP and WebSocket Protocols.

- Build real-time two-way communication applications, such as chat apps and streaming dashboards like the score of a sports game, with WebSocket APIs.

- The client and the server can both send messages to each other at any time. Backend servers can easily push data to connected users and devices.

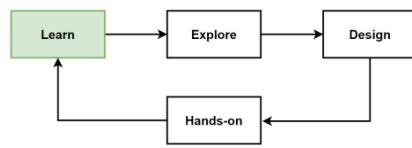
- **Pull communication based on HTTP and AMQP (short polling - long polling)**

Also called "Polling" and it's basically the same as refreshing your mail inbox every 5 minutes to check for new mail. It is a call and ask model.

- This model is become a waste of bandwidth if there are no new messages and responses comes from the server.

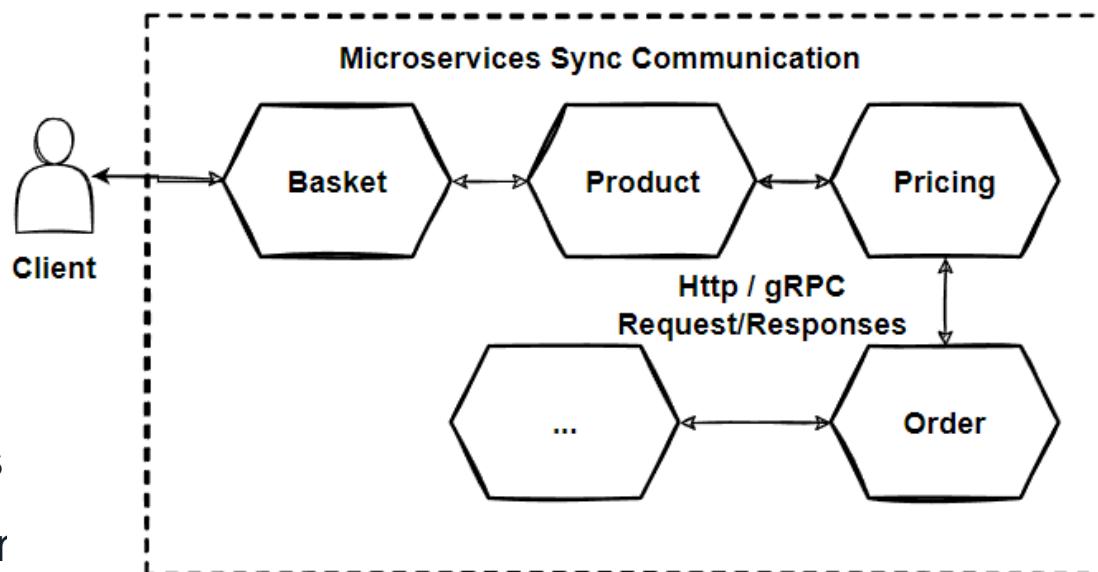
- Opening and closing connections is expensive. And we can say that this model doesn't scale well.

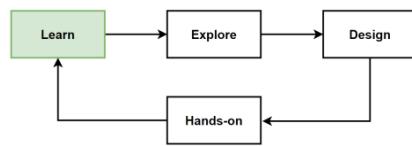
- Typically have limits like on Twitter on how often they allow you to call their API.



Microservices Synchronous Communications and Best Practices

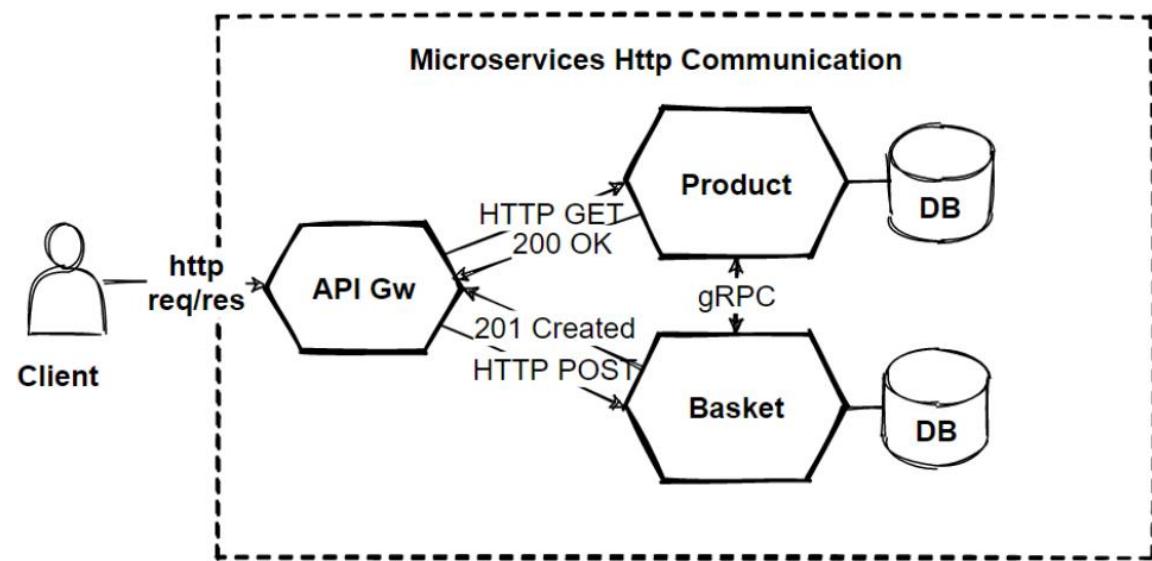
- The client **sends a request** with using http protocols and **waits for a response** from the service.
- The synchronous communication protocols can be **HTTP** or **HTTPS**.
- **Request/response communication** with HTTP and REST Protocol (extends gRPC and GraphQL)
- **REST HTTP APIs** when exposing from microservices
- **gRPC APIs** when communicate internal microservices
- **GraphQL APIs** when structured flexible data in microservices
- **WebSocket APIs** when real-time bi-directional communication
- How can we design and exposing APIs with HTTP protocols for our microservices ?

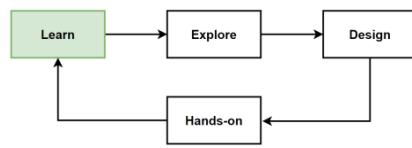




Designing HTTP based RESTful APIs for Microservices

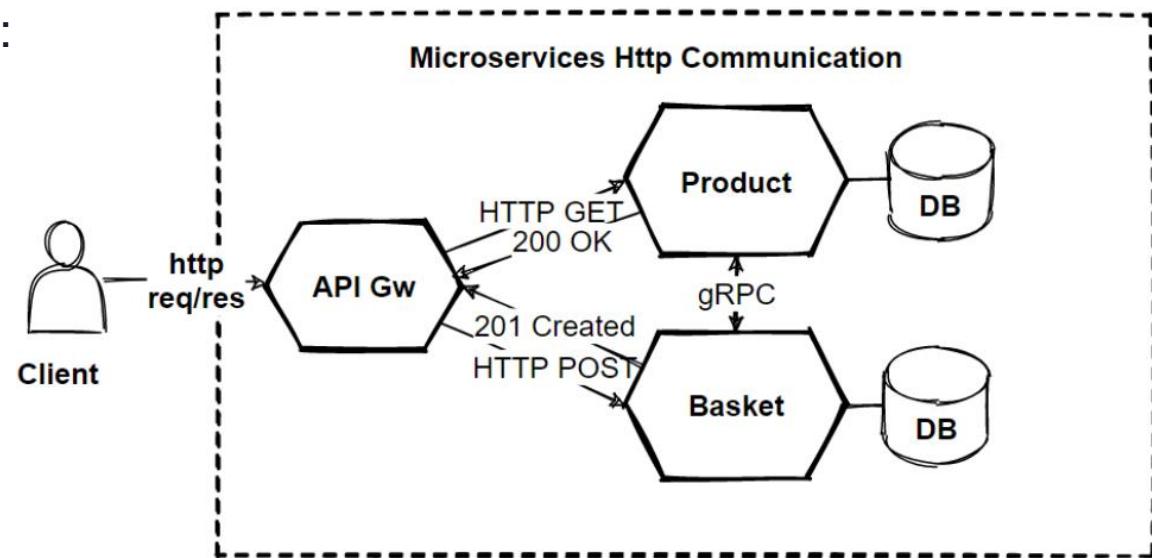
- When making **request/response communication**, we should use **REST** when designing our APIs. Its also called **Restful APIs**.
- **REST** approach is following the **HTTP protocol**, and implementing HTTP verbs like GET, POST, and PUT.
- **REST** is the most commonly used architectural communication approach when creating **APIs** for our **microservices**.
 - Java and Sprint Boot framework
 - C# with ASP.NET Core Web API
 - Python with Flask Web API framework
- How to **design** our **APIs** for **microservices** ?

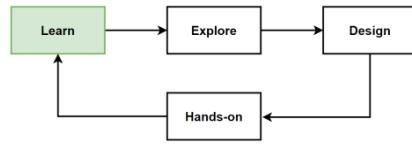




Designing HTTP based RESTful APIs for Microservice2

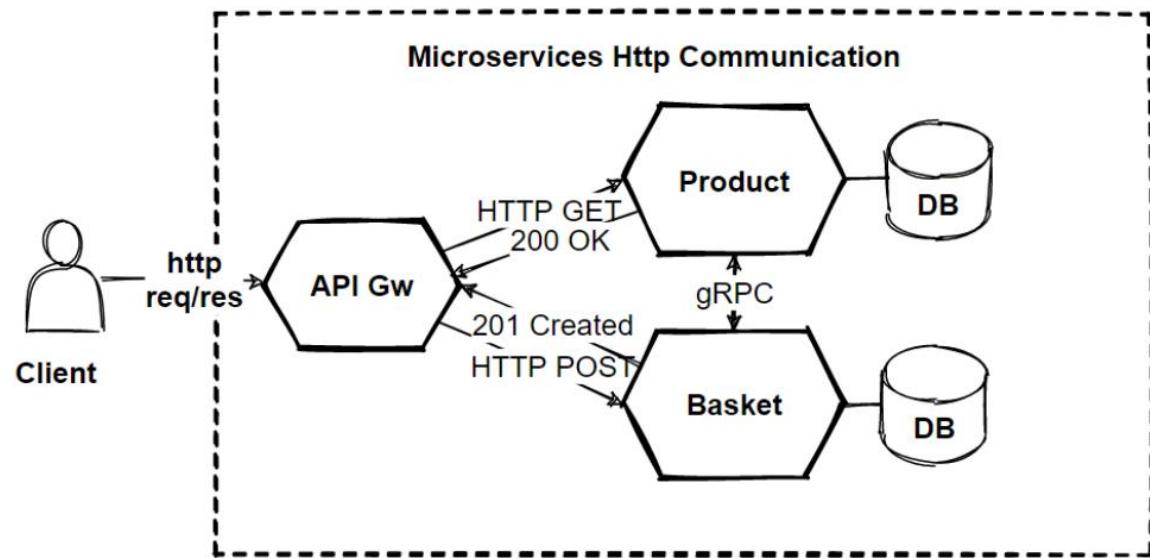
- **Well-defined API** design is very important in a microservices architecture, communication happens API calls.
- Designed APIs should be **efficient** and **not to be chatty communications**. APIs must have well-defined documented and versioning.
- There are 2 type APIs sync communication in microservices:
 - Public APIs: API calls from the client apps.
 - Backend APIs: inter-service communication between backend services.
- **Public APIs**
Use RESTful APIs over HTTP protocol. RESTful APIs use JSON payloads for request-response, that easy to check payloads and easy agreement with clients.
- **Backend APIs**
Inter-service communication can result in a lot of network traffic. serialization speed and payload size become more important. Using gRPC is mandatory for increase network performance.

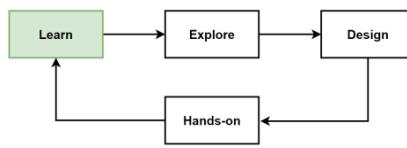




Comparison with REST and gRPC

- **REST** is using HTTP protocol, and request-response structured JSON objects.
- API interfaces design based on HTTP verbs like GET-PUT-POST and DELETE.
- **gRPC** is basically Remote Procedure Call, that basically invoke external system method over the binary network protocols.
- Payloads are **not readable** but its **faster** than **REST APIs**.
- 2 Main Approaches for Public and Backend APIs:
 - RESTful API Design over HTTP using JSON
 - gRPC binary protocol API Design

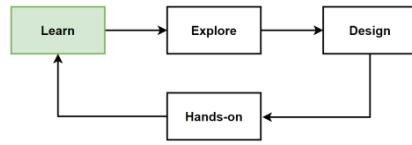




gRPC: High Performance Remote Procedure Calls

- gRPC is an open source **remote procedure call** (RPC) system developed at Google.
- gRPC is a framework to **efficiently connect services** and build distributed systems.
- It is **focused on high performance** and uses the **HTTP/2 protocol** to transport binary messages.
- It relies on the **Protocol Buffers** language to define service contracts.
- **Protocol Buffers (Protobuf)**, allow to define the interface to be used in service to service communication regardless of the programming language.
- It generates **cross-platform client and server bindings** for many languages.
- Most common **usage** scenarios include connecting services in **microservices style architecture**.

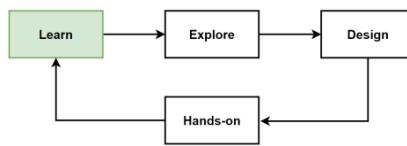




gRPC: High Performance Remote Procedure Calls

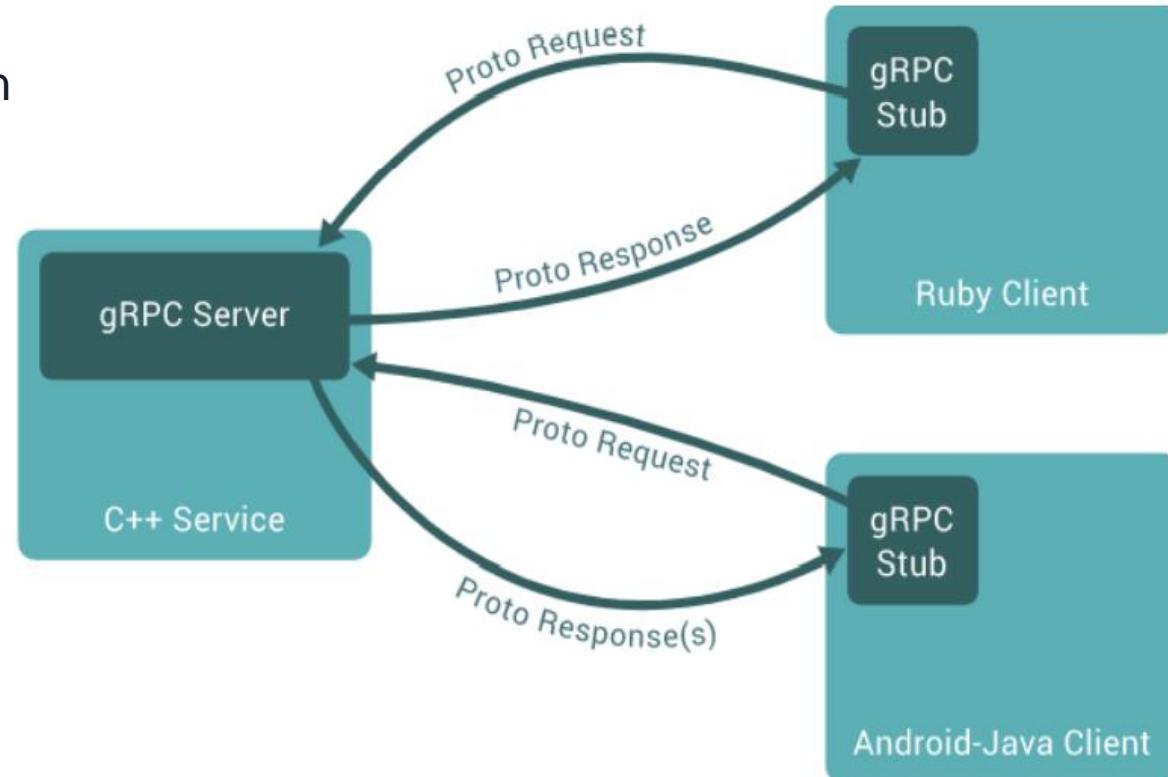
- **gRPC framework** allows developers to create services that can communicate with each other **efficiently** and **independently** with their **preferred programming language**.
- Once you **define a contract** with **Protobuf**, this contract used by each service to automatically **generate the code** that sets up the communication infrastructure.
- This feature simplifies the creation of service interaction and together with **high performance**, makes **gRPC** the **ideal framework** for creating **microservices**.

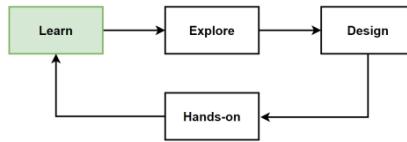




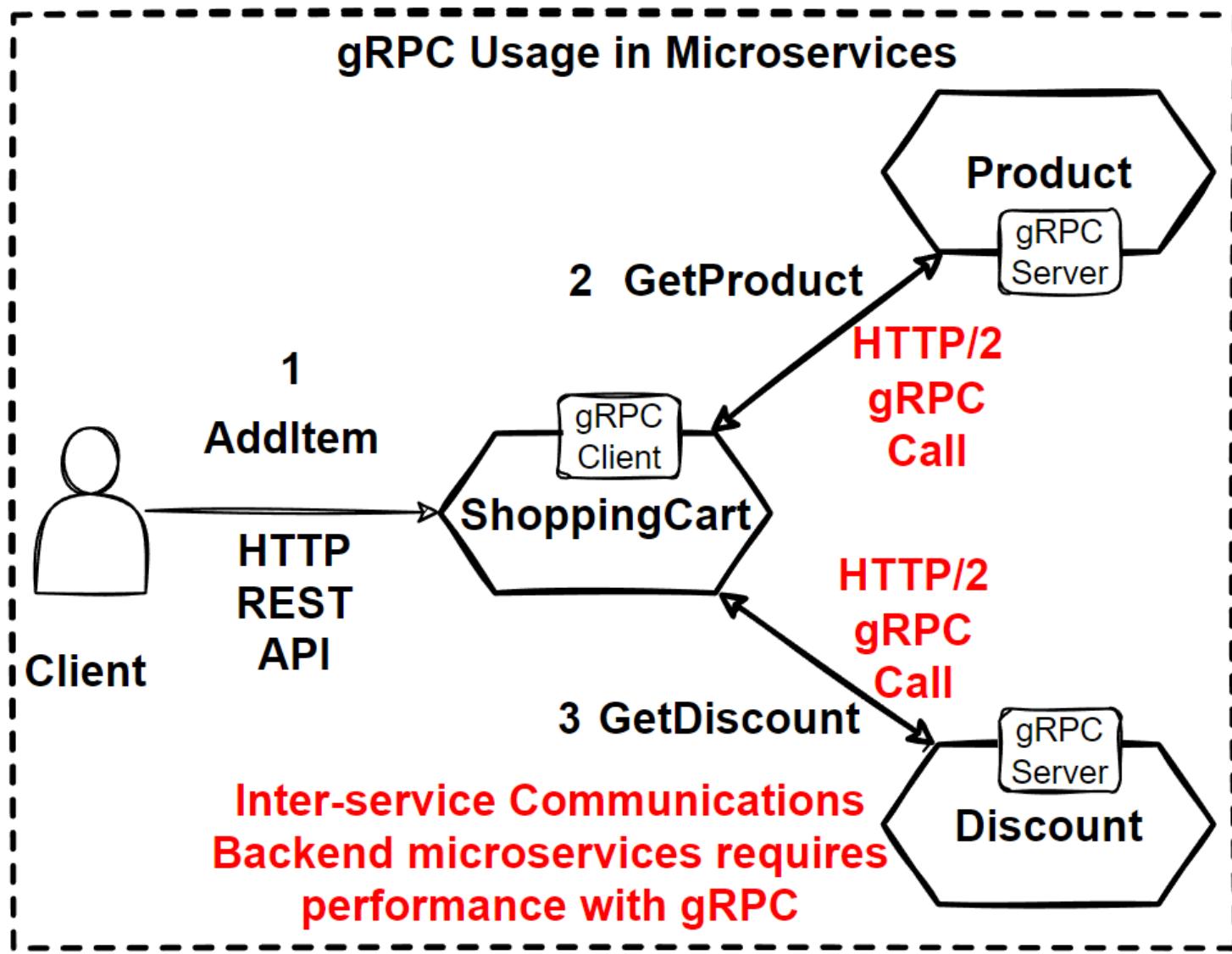
How gRPC works ?

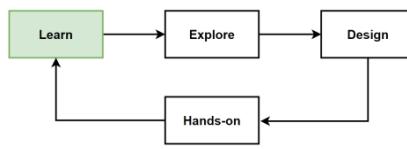
- RPC is a form of **Client-Server Communication** that uses a **function call** rather than a usual HTTP call.
- In **GRPC**, client application can **directly call a method** on a server application on a **different machine** like a local object.
- **gRPC** is based on the idea of **defining a service** and **methods**, and these can be called remotely with their parameters and return types.
- On the **server side**, the server implements this interface and runs a gRPC server to handle client calls.
- On the **client side**, the client has a stub that provides the same methods as the server.
- **gRPC clients** and **servers** can work and talk to each other in a different of environments.





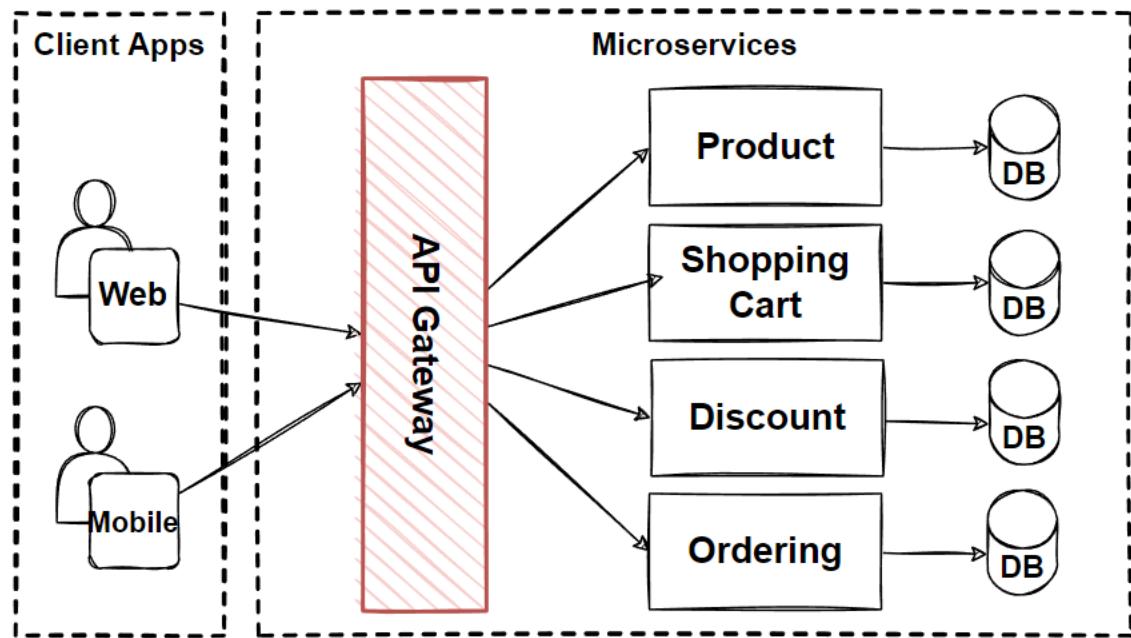
gRPC Usage in Microservices Communication

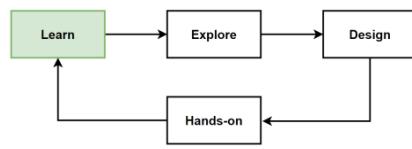




API Gateway Pattern

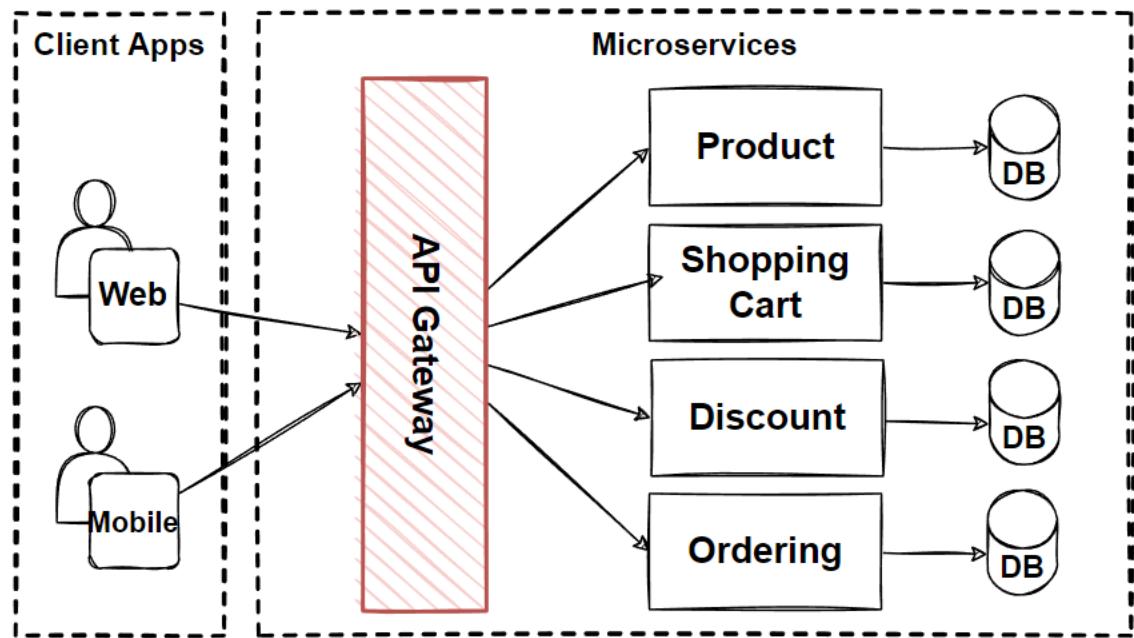
- API Gateway is a **single point of entry** to the client applications, sits between the **client** and **multiple backend**.
- API Gateways **manage routing** to internal microservices and able to **aggregate several microservice** request in 1 response and handle **cross-cutting concerns**.
- Recommended when design **complex large** microservices-based applications with **multiple client** applications.
- Similar to the **facade pattern** from object-oriented design, but it is a distributed system **reverse proxy or gateway routing** for using in synchronous communication.
- The pattern provides a **reverse proxy to redirect or route requests** to your internal microservices endpoints.
- API Gateway provides a **single endpoint** for the client applications, and it **maps the requests** to internal microservices.

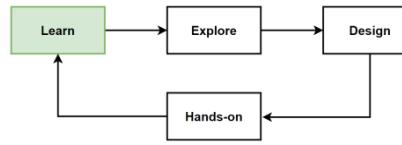




API Gateway Pattern - Summarized

- API gateway **locate between the client apps and the internal microservices.**
- Working as a **reverse proxy** and **routing requests** from clients to backend services and provide **cross-cutting concerns** like authentication, SSL termination, and cache.
- Several client applications connect to single API Gateway possible to **single-point-of-failure risk**.
- If these client applications increase, or adding more logic to **business complexity** in API Gateway, it would be **anti-pattern**.
- Best practices is **splitting the API Gateway** in multiple services or multiple smaller API Gateways: **BFF-Backend-for-Frontend Pattern**.
- Should **segregated based on business boundaries** of the client applications.





Main Features of API Gateway Pattern

- **Reverse Proxy and Gateway Routing**

Reverse proxy to redirect requests to the endpoints of the internal microservices. Using Layer 7 routing for HTTP requests for redirections. Decouple client applications from the internal microservices. Separating responsibilities on network layer and abstracting internal operations.

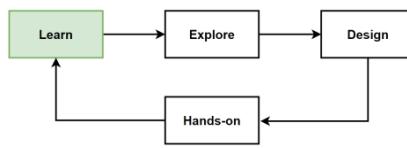
- **Requests Aggregation and Gateway Aggregation**

Aggregate multiple internal microservices into a single client request. Client application sends a single request to the API Gateway and it dispatches several requests to the internal microservices and then aggregates the results and sends back to the client application in 1 single response. Reduce chattiness communication.

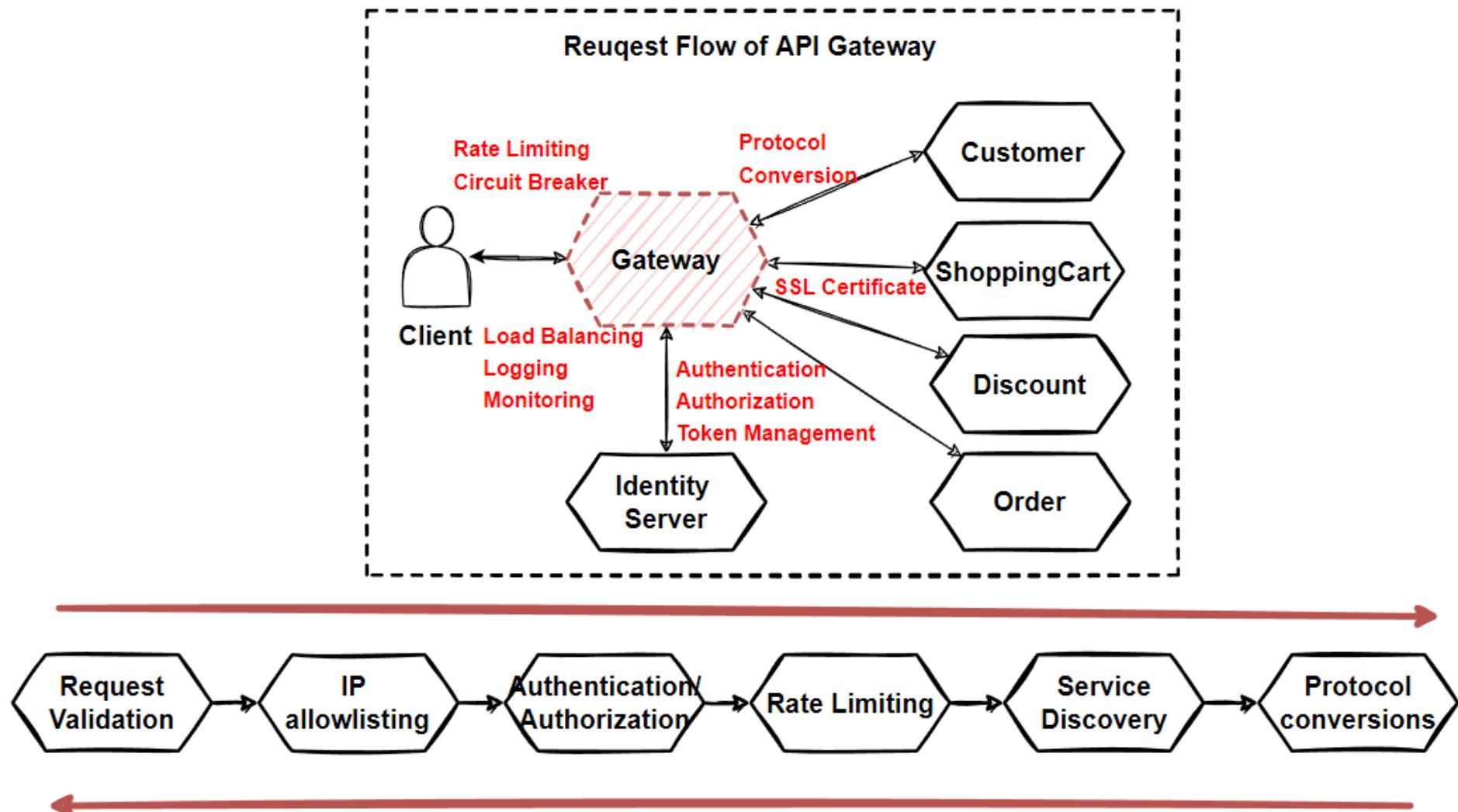
- **Cross-cutting Concerns and Gateway Offloading**

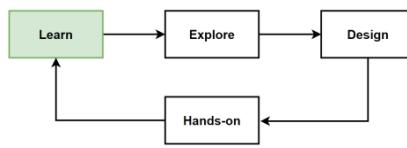
Best practice to implement cross-cutting functionality on the API Gateways. Cross-cutting functionalities can be; Authentication and authorization, Service discovery, Response caching, Retry policies, Circuit Breaker, Rate limiting and throttling, Load balancing, Logging, tracing, IP allowlisting.

Routing	Authentication
Request Aggregation	Authorization
Service Discovery with Consul & Eureka	Throttling
Load Balancing	Logging, Tracing
Correlation Pass-Through	Headers/Query String Transformation
Quality of Service	Custom Middleware



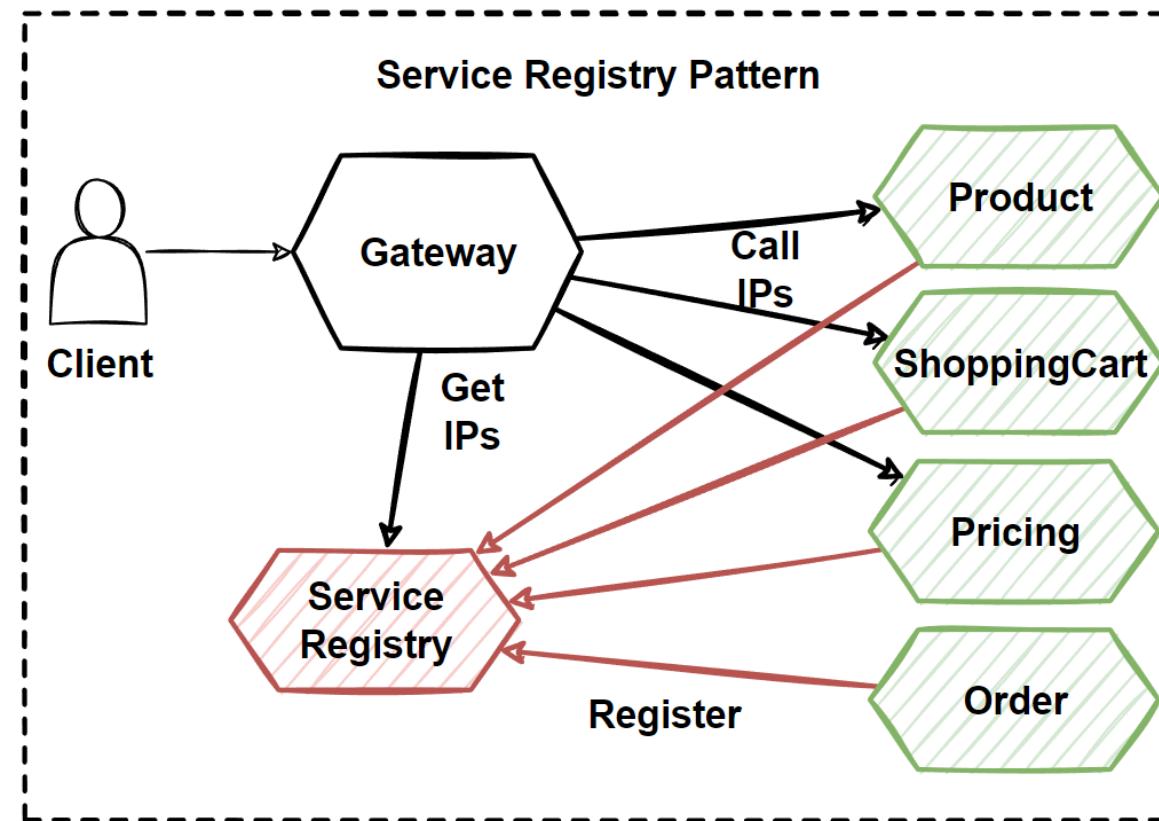
Request Flow of API Gateway Pattern

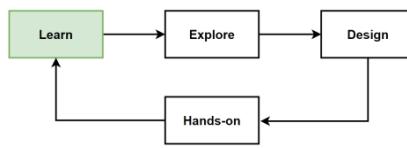




Service Registry/Discovery Pattern

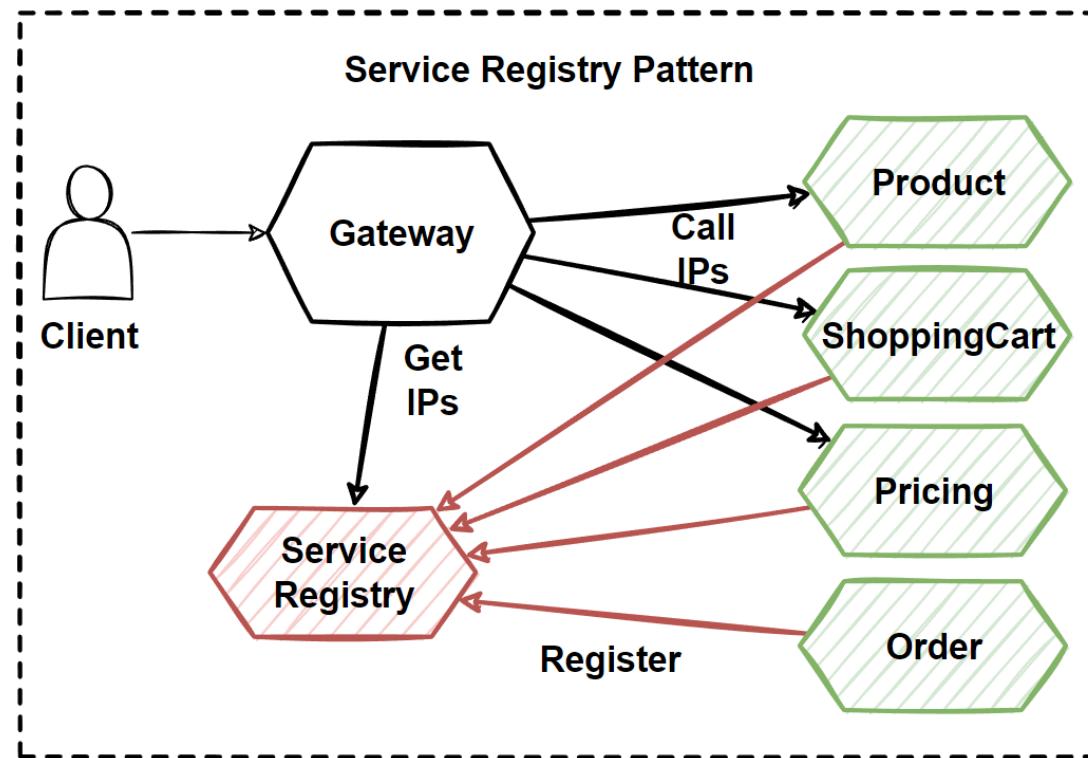
- Microservice **Discovery Patterns** and **Service Registry** provide to register and discover microservices in the cluster.
- **Why We Need Service Discovery Pattern ?**
When deploying and scaling microservices, more microservices and instances could be added to the system to provide scalability to the distributed application.
- **Service locations** can **change frequently**, more dynamic configuration is needed for the microservice architecture.
- **IP Addresses** and **network locations** are **dynamically** assigned and often change due to auto-scaling features.
- **Service Registry** and **Discovery pattern** allows to find the network locations of microservices without injecting or coupling services.

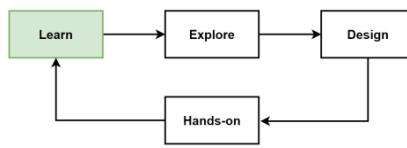




What is Service Discovery Pattern?

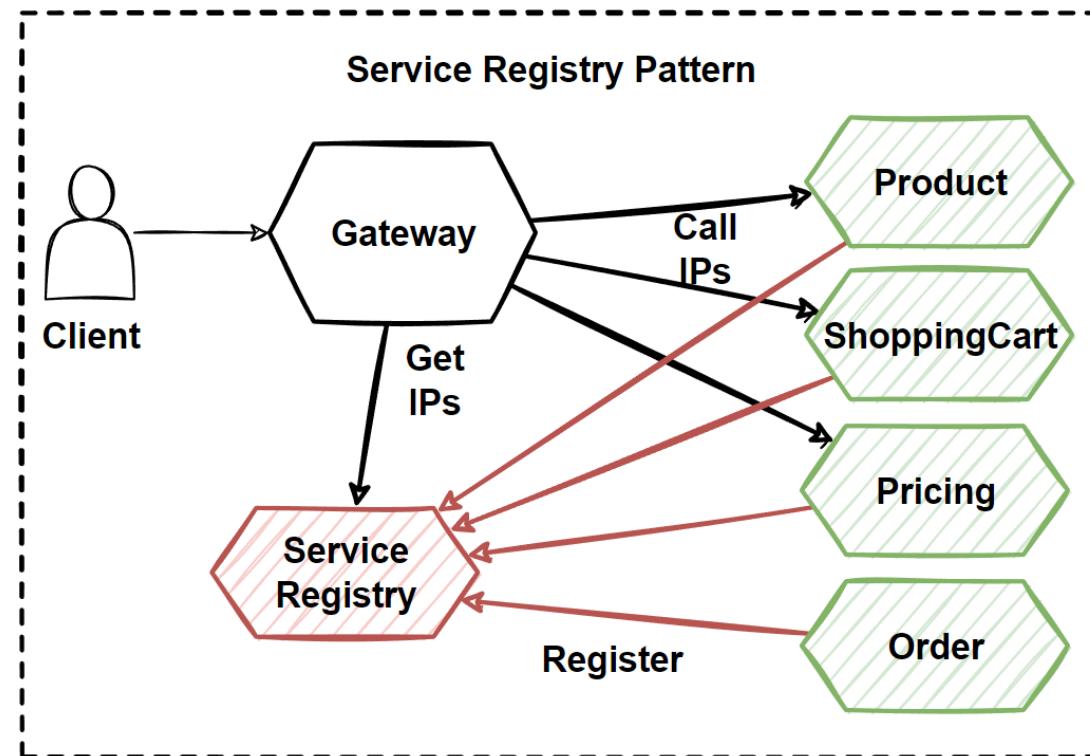
- To find the **network locations** of microservices, the **service discovery pattern** is used in microservices applications.
- It will provide to **register and discover microservices** in the cluster.
- API Gateways** for **routing the traffic** with client and internal microservices. How API Gateways access the internal backend microservices ?
- Service discovery pattern** uses a **centralized server** for «service registry» to maintain a central view of microservices network locations.
- Services **update their locations** in the **service registry** at fixed intervals. Clients can connect to the service registry and fetch the locations of microservices.
- There are **2 main service discovery patterns**:
 - Client-side service discovery
 - Server-side service discovery

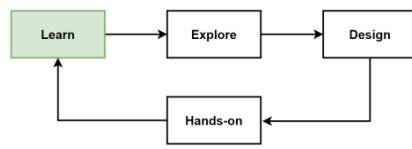




How Does Service Discovery Work?

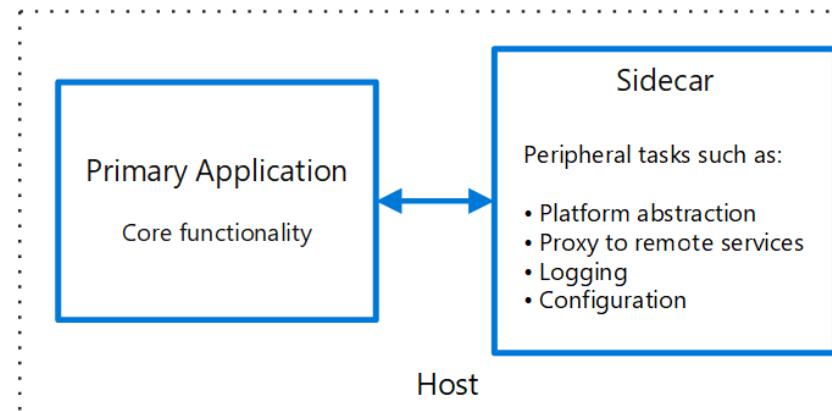
- **3 participants:** Service Registry, Client, Microservices
- **Client service finds the location of other service instances by querying a Service Registry.**
- Internal backend services **register to Service Registry** service before. The Service Registry is a **database for microservice instances**.
- The service registry **keeps records of the network locations** of the microservices.
- When the client service need to access internal services, then it will **query from Service Registry** and access them.
- Clients can **find the locations** of microservices using the **service registry** and directly call them.
- **Netflix** provides a service discovery pattern called «**Netflix Eureka**»
- **Kubernetes** container orchestrator **automatically handle** to service discovery operations.

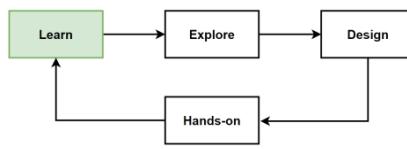




Sidecar Pattern

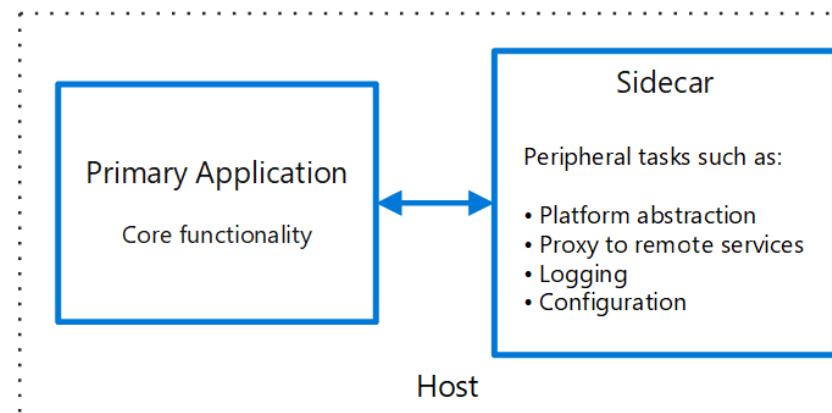
- **Sidecar pattern** is a design pattern in which a **helper container** is run alongside the **main container** in a pod.
- **The Sidecar container performs cross-cutting tasks** that are related to the main container.
- **Common use case for the sidecar pattern** is to add **logging** or **monitoring** functionality to a container.
- **A sidecar container** that runs a **logging agent** that sends application logs to a **central logging server**, and also **provide to monitoring** the **health** of the **main container**.
- **These cross-cutting tasks** can be implemented as **separate services** with **sidecar pattern** such as **monitoring**, **logging**, **configuration**, and **networking** services.
- The sidecar container and the main container **share the same network namespace**, that allows the sidecar container to **access** and **manipulate** the **data produced** by the **main container** as needed.

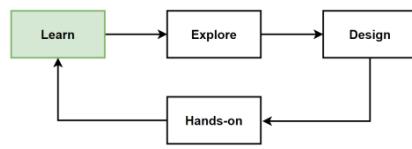




Benefits of Sidecar Pattern

- Allows to **add functionality** to a **container** without modifying the container itself.
- Useful if you want to **add logging or monitoring functionality** to an existing container image that you do not control.
- Allows you to **decouple the main container from the sidecar functionality**, easier to update or replace the sidecar without affecting the main container.
- Allows to run **multiple containers in a pod** that share the same network namespace, making it **easier to communicate** between them.
- Allows to **add common functionality** to multiple microservices **without having to modify the microservices** themselves.





Drawbacks of Sidecar Pattern

- **Increased complexity**

Adding an additional layer of complexity to your deployment, more difficult to understand and troubleshoot issues that arise.

- **Increased resource usage**

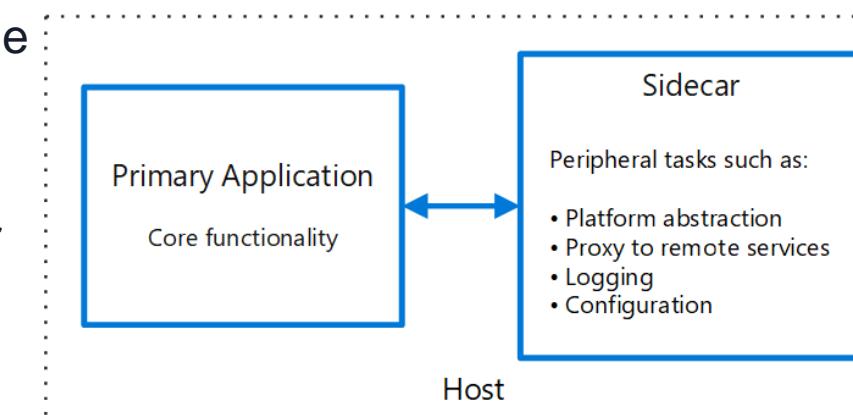
Running an additional container in a pod will increase the resource usage of the pod.

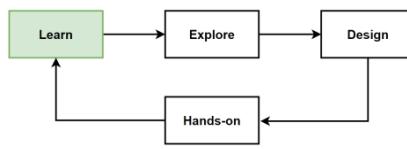
- **Decreased performance**

Pod can potentially decrease the performance of the pod, as the sidecar container will be competing for resources with the main container.

- **Limited flexibility**

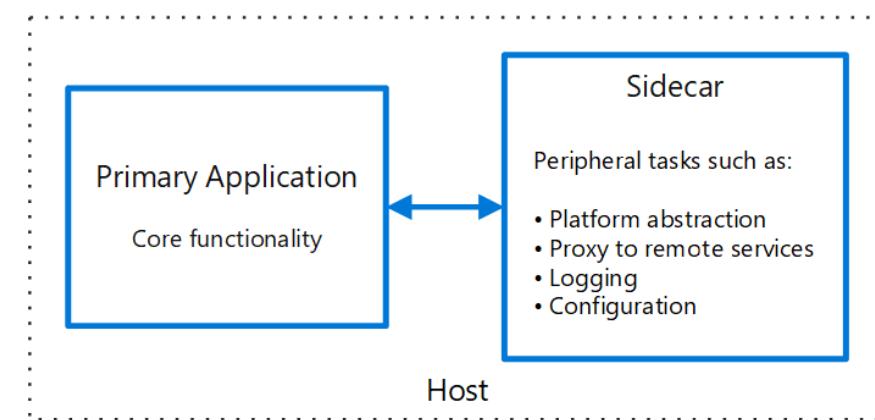
Can be inflexible in some cases, as it requires that the main container and the sidecar container run in the same pod.

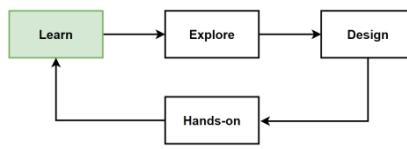




When to use Sidecar Pattern

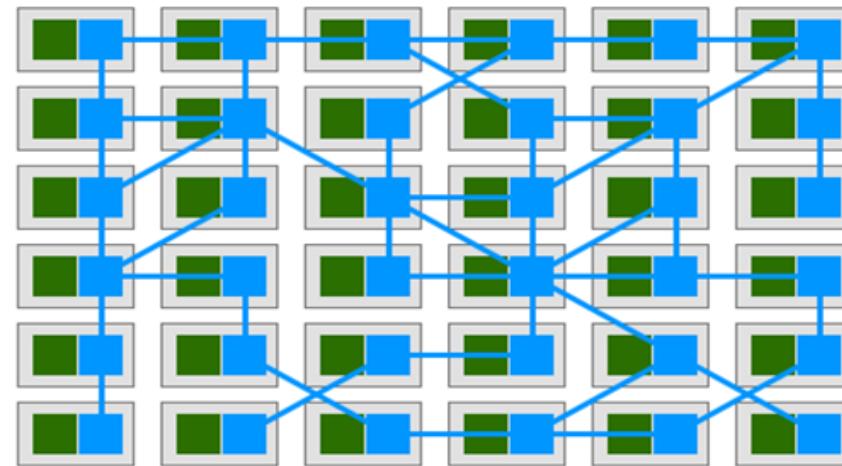
- When you want to **add functionality** to an **existing container** image
- When you want to **decouple the main container** from the additional functionality
- When you want to **run multiple containers in a pod** that need to communicate with each other
- When you want to **add common functionality** to multiple microservices

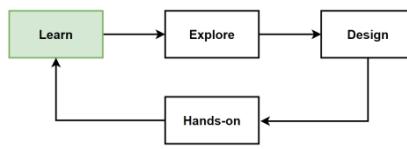




Service Mesh Pattern

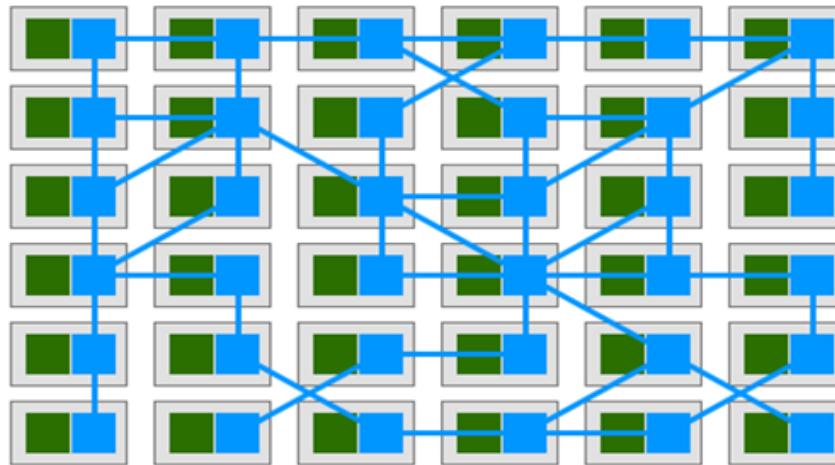
- **Service mesh pattern** is managing the **communication** between **microservices** in a distributed system.
- Designed to provide a **uniform way to route traffic** between microservices, handle **load balancing**, and **monitor the healths**.
- Consists of a **set of proxy servers (sidecars)** that are deployed alongside the microservices. Proxy servers **handle the communication** between the microservices.
- **Responsible** for tasks such as **routing requests, load balancing, and monitoring the health** of the system.
- **Abstract away the complexities** of managing communication between microservices.
- **Instead of** having to **manage** these details at the **application level**, use the **service mesh** to handle them **automatically**.
- **Popular Service mesh** implementations, including **Istio** and **Linkerd**, set up and manage a service mesh in a **Kubernetes cluster**.

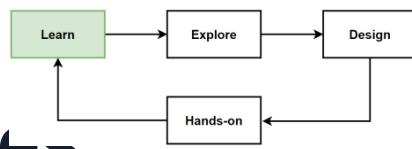




When to use Service Mesh Pattern

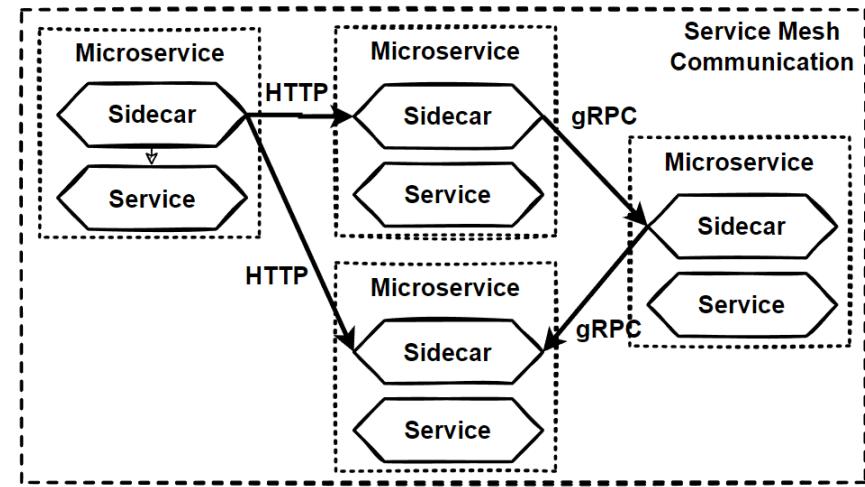
- When you want to **abstract away the complexities of managing communication** between microservices.
- When you want to **centralize the management** of communication between microservices.
- When you want to **add features** such as **load balancing, traffic management, and monitoring** to your microservices.
- **Using a service mesh**, can add these features to your microservices **without having to modify the microservices themselves**.
- **Service mesh pattern** is a useful tool for **managing the communication** between microservices in a distributed system.
- **Build and maintain complex microservices-based systems** by abstracting away the complexities of managing communication between the microservices.

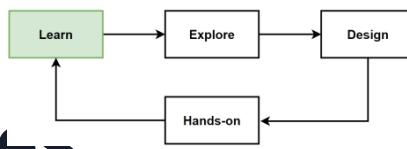




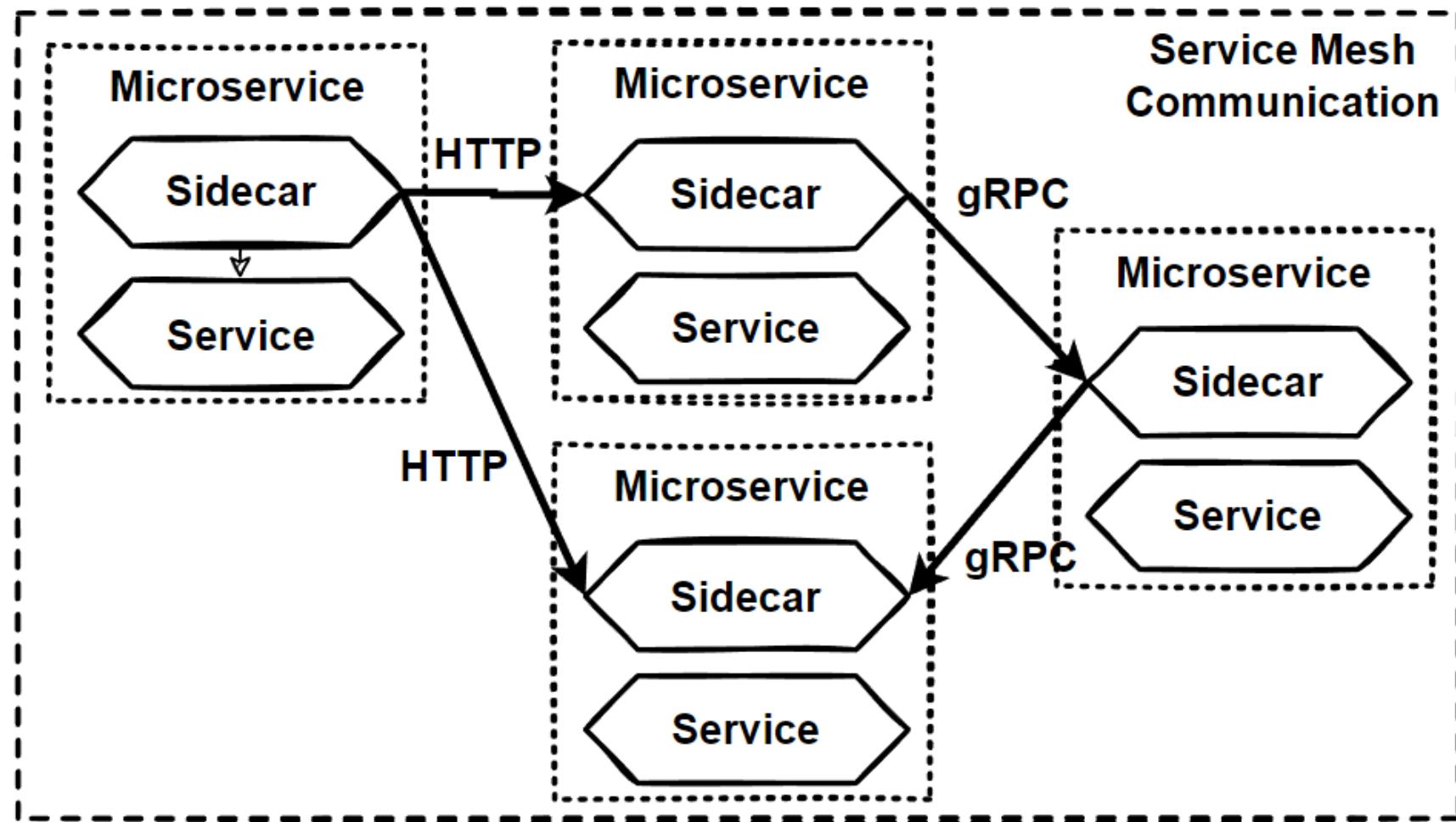
Service Mesh Communication in Cloud-Native Microservices

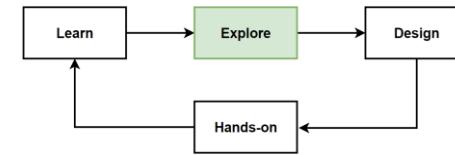
- **Minimizing inter-service communication** is ideal.
- Microservice communication: **synchronous HTTP** and **gRPC** communication and **asynchronous messaging**.
- **Service mesh is a dedicated infrastructure layer** that manages **communication between microservices**.
- Secure the interactions between services **without adding complexity** to the individual microservices themselves.
- **Built-in capabilities** for **service-to-service communication, resiliency, and handling cross-cutting concerns**.
- **Communication** concerns is **moved out of the microservices** and into the service mesh layer.
- This **abstraction** allows **communication** to be handled **independently** from your **microservices**.
- Support for **service discovery and load balancing**. Handles **traffic management, communication, and networking** concerns.





Service Mesh Communication in Cloud-Native Microservices



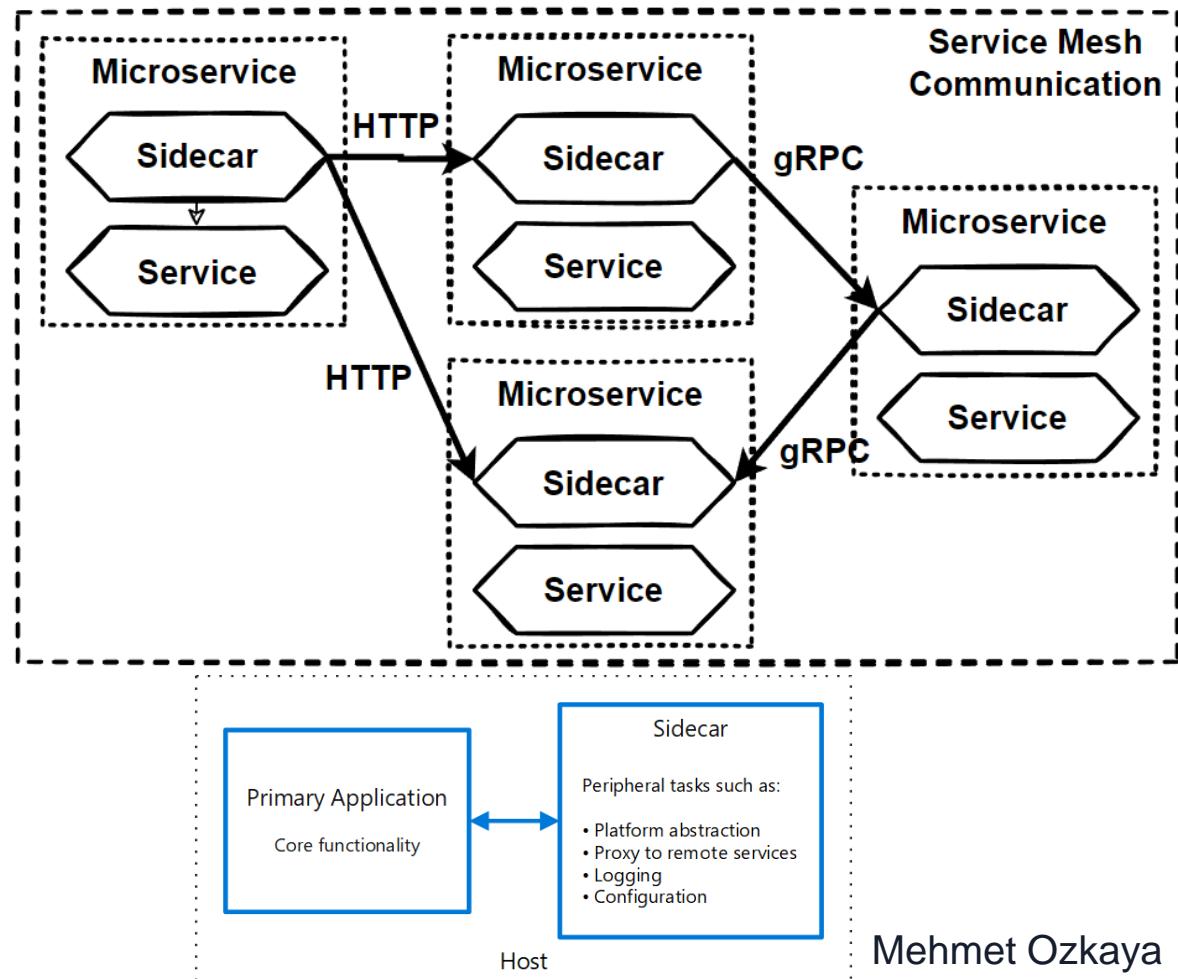
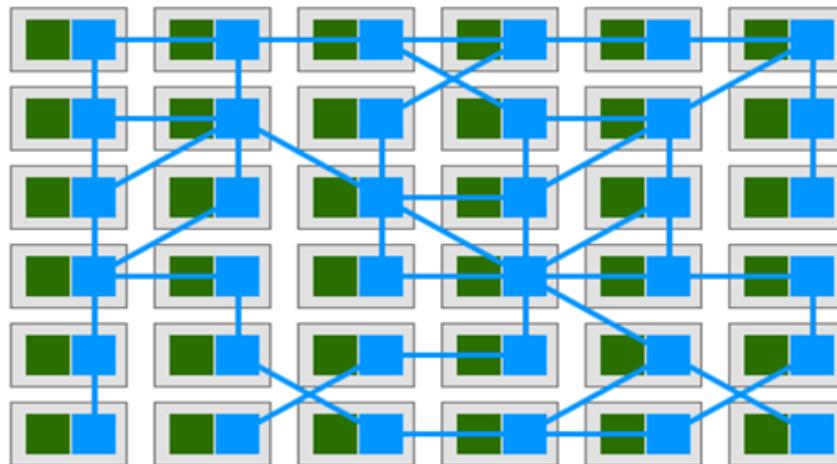


Explore: Communication tools: Service Proxy, API Gateway and Service Meshes

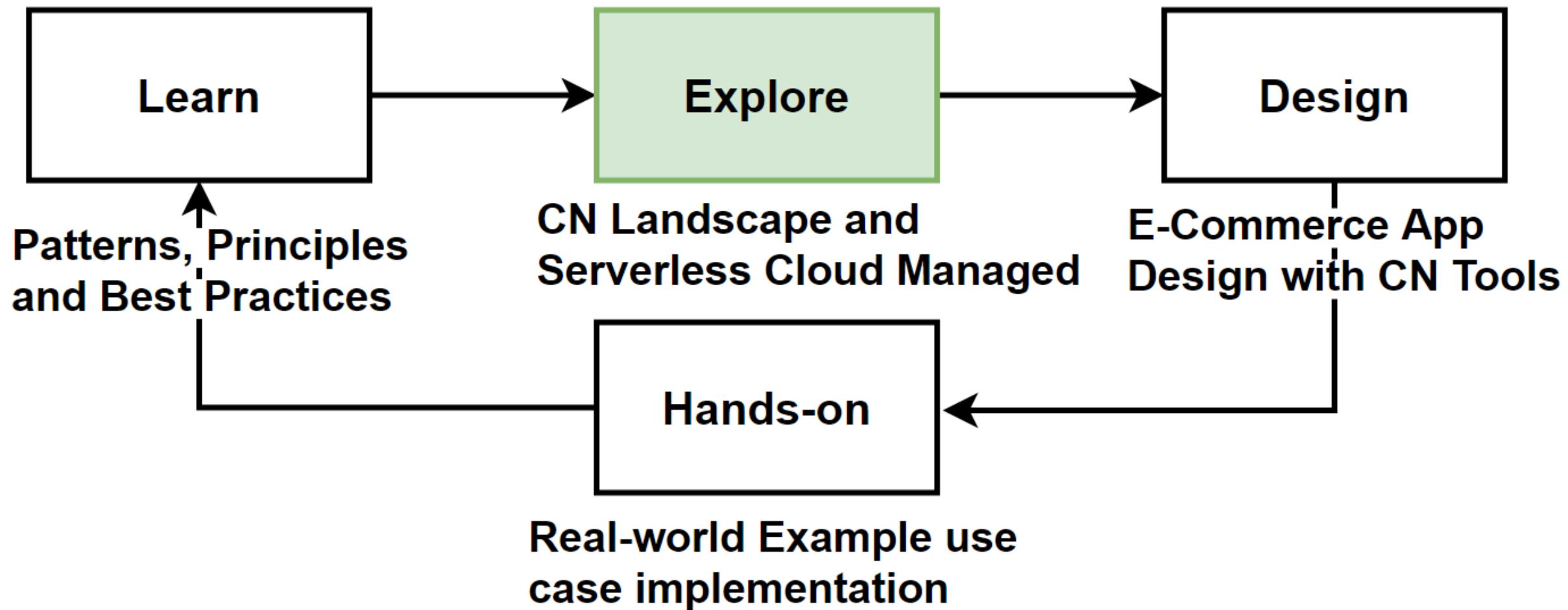
- Communication tools: Service Proxy (envoy), API Gateway(kong) and Service Meshes (istio, linkerd)

Exploring Communication Tools

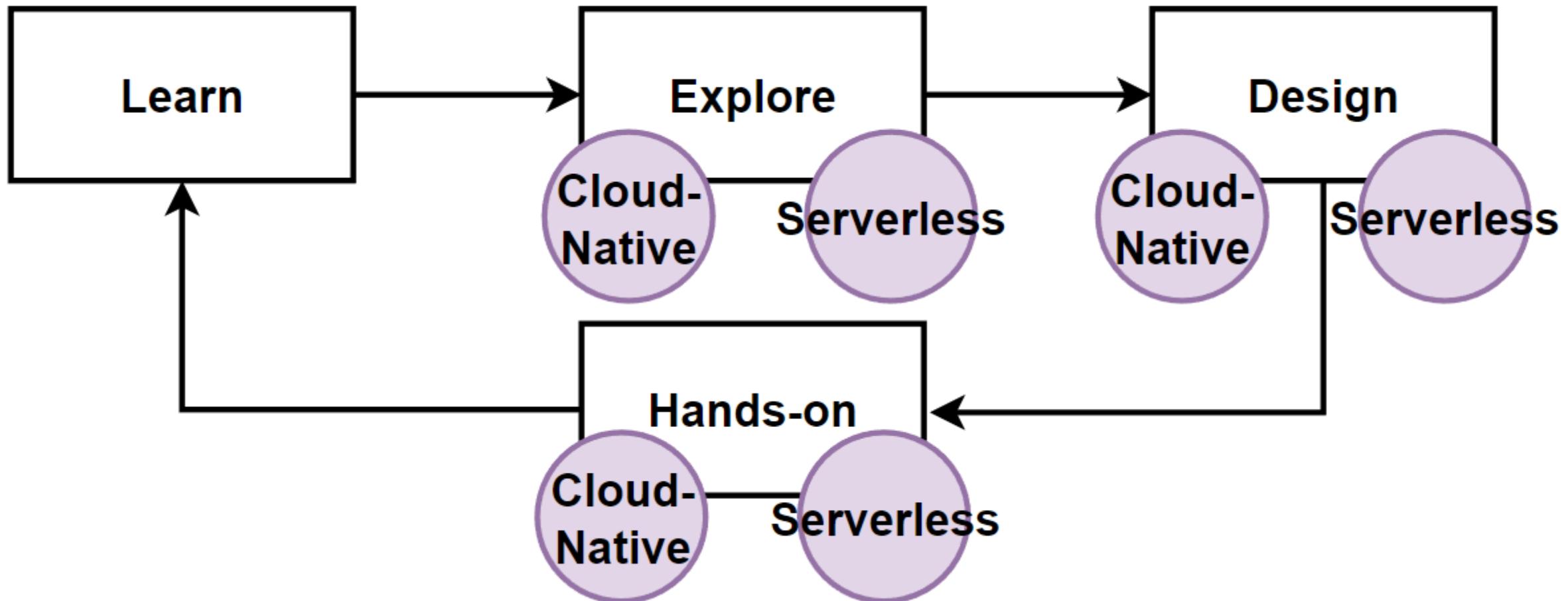
- Service Proxy
- API Gateway
- Service Meshes



Way of Learning – The Course Flow

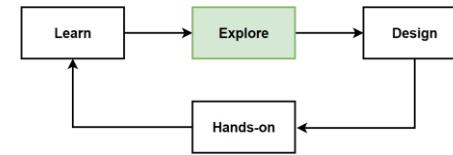


Way of Learning – Cloud-Native & Serverless Cloud Managed



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Explore: Communication tools: Service Proxy, API Gateway and Service Meshes

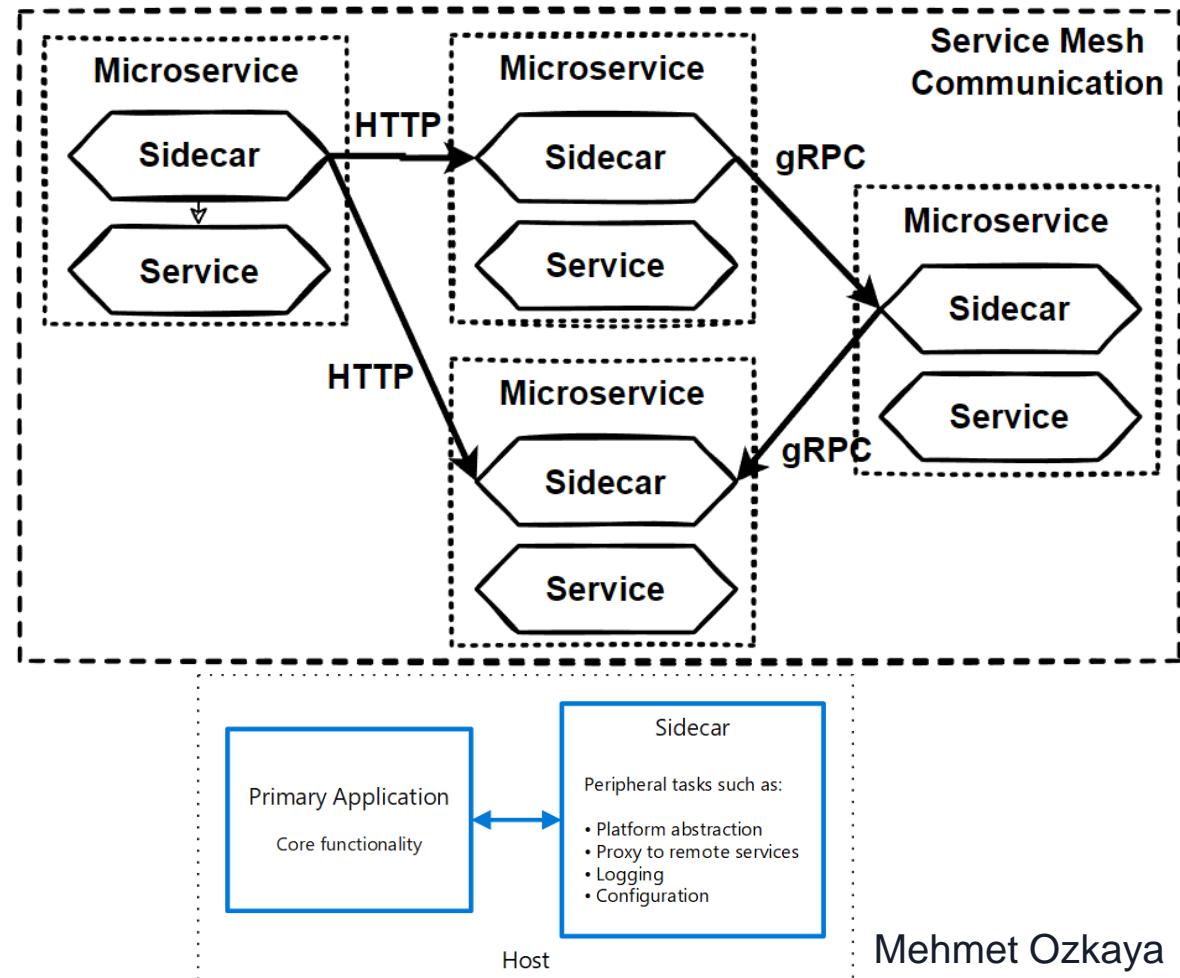
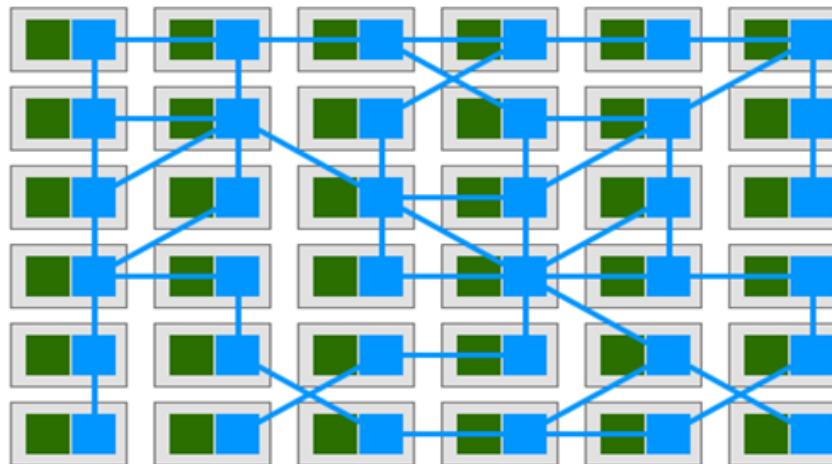
- Communication tools: Service Proxy (envoy, nginx, haproxy), API Gateway(Kong) and Service Meshes (istio, linkerd)

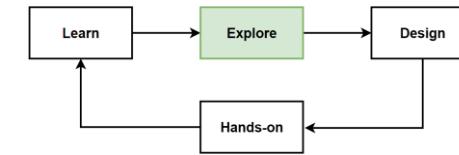
Exploring Communication Tools

- Service Proxy (envoy, nginx, haproxy)
- API Gateway (Kong, krakend, kubeGateway)
- Service Meshes (istio, linkerd, kuma)

Cloud-Native Landscape

- <https://landscape.cncf.io/>





Explore: Cloud Serverless Communication tools: AWS, Azure Service Proxy - API Gateway - Service Meshes

- AWS and Azure offer managed serverless tools and services that can work with or provide similar functionality to the CNCF projects.

AWS Service Proxy

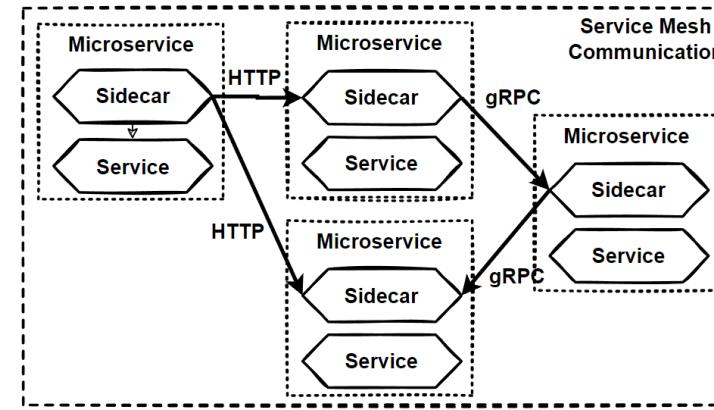
- AWS App Mesh is a service mesh that uses Envoy as a proxy, providing application-level networking for microservices on AWS.

AWS API Gateway

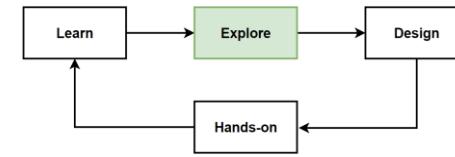
- AWS API Gateway is a fully managed service for creating, publishing, and managing APIs. It handles features like authentication, rate limiting, and caching.

AWS Service Meshes

- AWS App Mesh can also be considered a managed service mesh, as it provides observability, traffic control, and security for microservices running on AWS.



Explore: Azure Service Proxy - API Gateway – Service Meshes



Azure Service Proxy

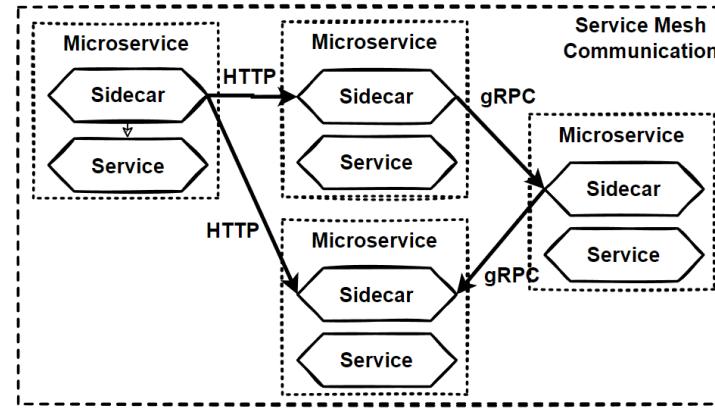
- Azure API Management can act as a service proxy, providing API gateway functionality with load balancing, rate limiting, and caching.

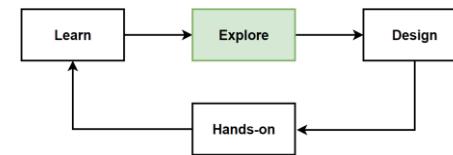
Azure API Gateway

- Azure API Management is a fully managed service for creating, publishing, and managing APIs. It provides features like authentication, rate limiting, caching, and monitoring.

Azure Service Meshes

- Azure Service Fabric Mesh is a fully managed service mesh for building and deploying microservices.
- It provides features like service discovery, load balancing, and communication security. Additionally, AKS supports the integration of Istio, Linkerd, and other service meshes.



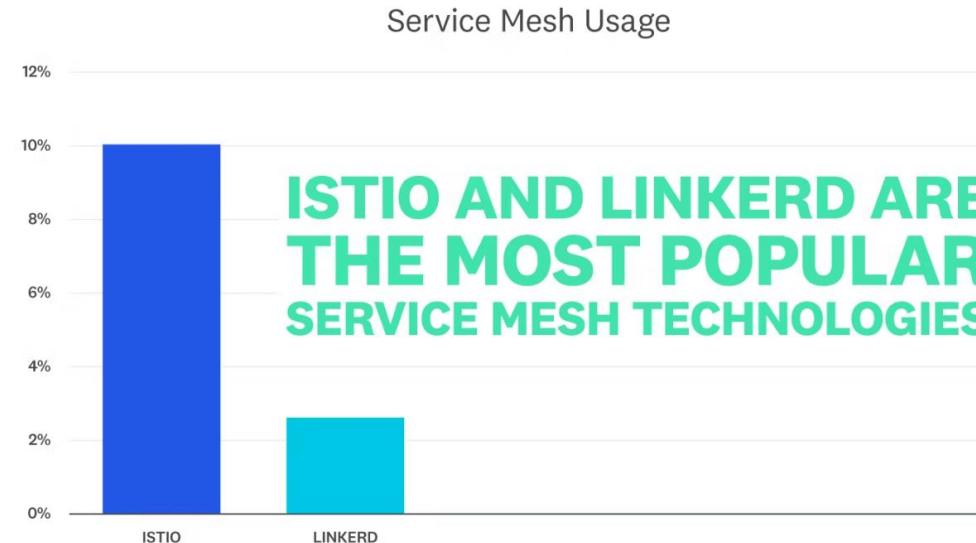


Service Meshes are still early and Istio dominates usage

- Datadog Report: Istio and Linkerd are the most popular service mesh technologies

- **Service meshes are still early and Istio dominates usage**

Service meshes provide service discovery, load balancing, timeouts, and retries, and allow administrators to manage the cluster's security and monitor its performance.

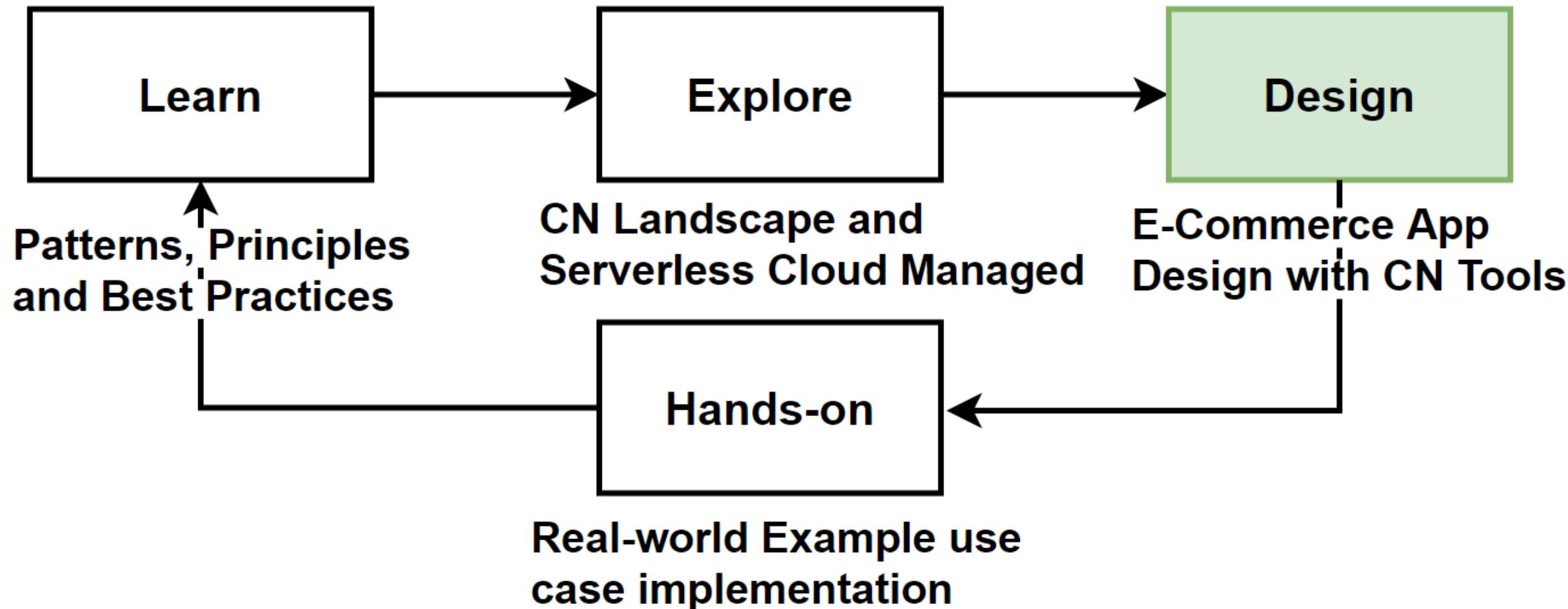


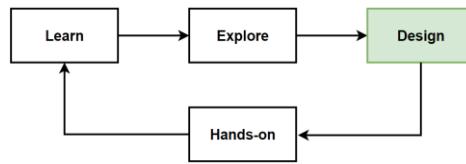
- **Louis Ryan, Co-creator of Istio**

«Service meshes have proven the value of delivering consistent security, observability & control for traffic in the enterprise. Istio has clearly established itself as the leading mesh solution and I'm proud of the work the community has done to get to this point. The recently completed donation of Istio to the CNCF will grow and strengthen our community to build on this success.»

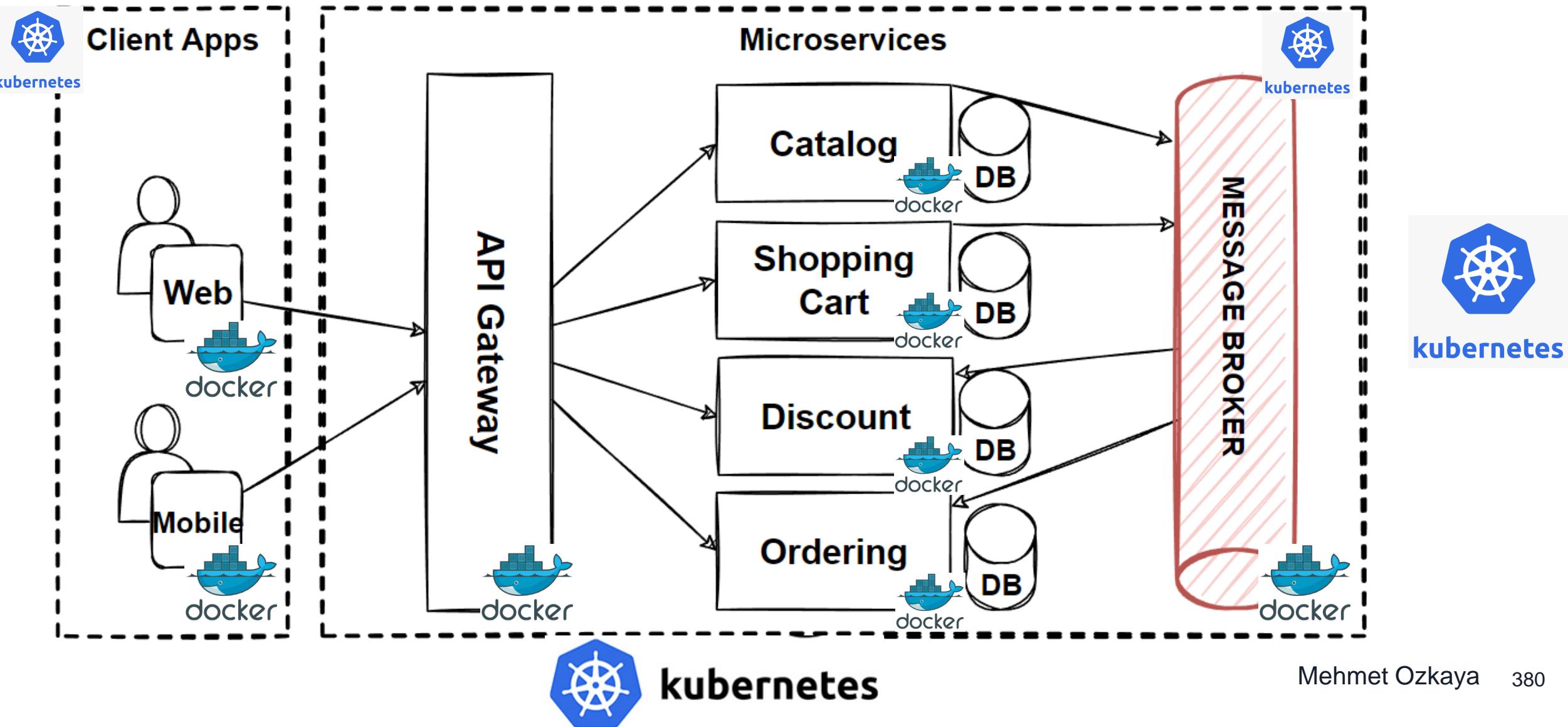
<https://www.datadoghq.com/container-report/#5>

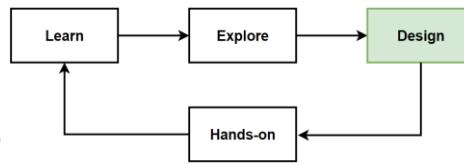
Way of Learning – The Course Flow



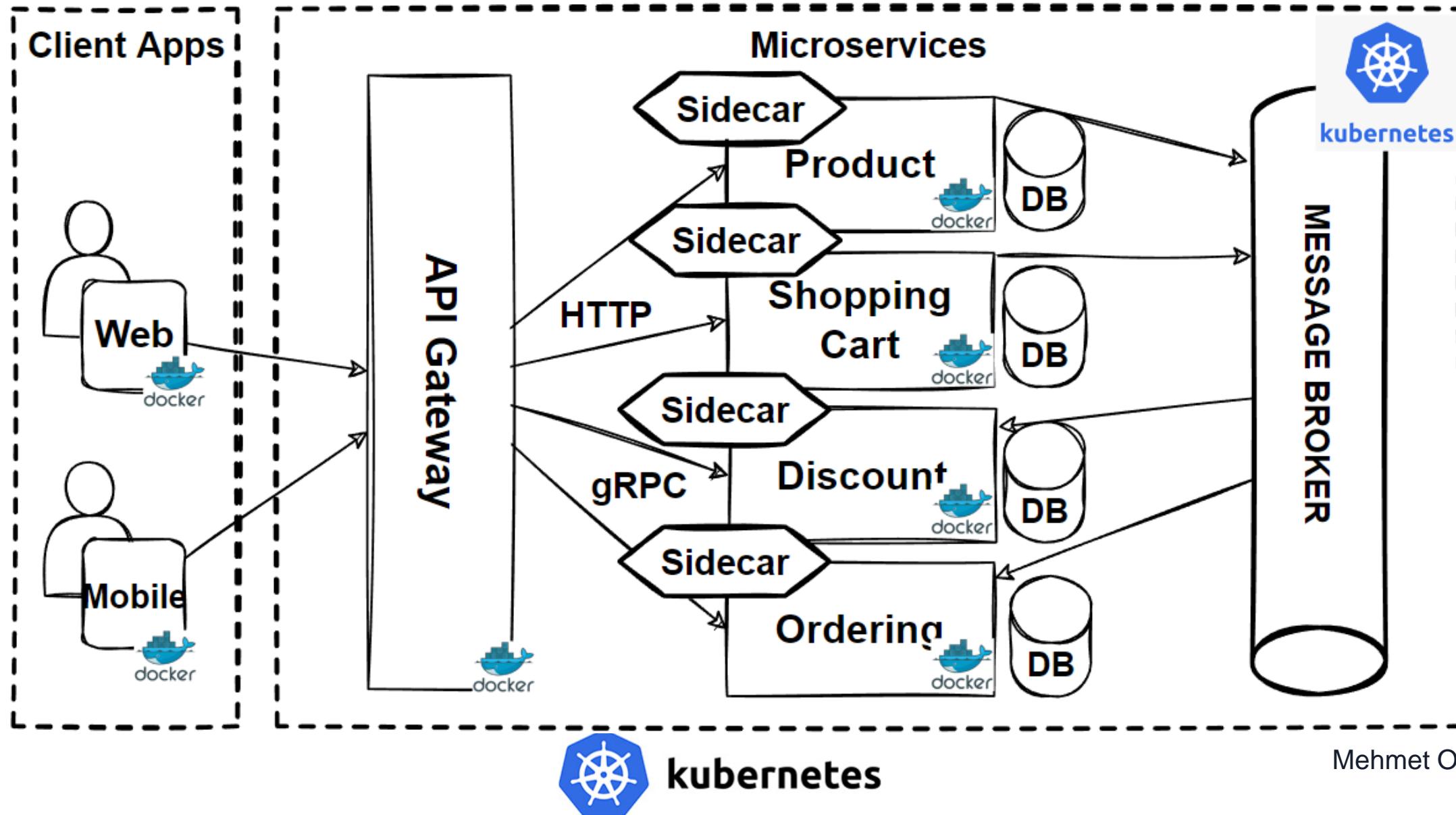


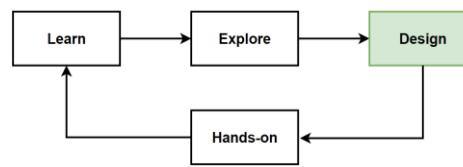
Design: Cloud-Native E-commerce w/ Orchestrator





Design: Cloud-Native E-commerce w/ Service Mesh Istio





Microservice > Container > Orchestrator > Service Mesh

Microservices Architecture

- App should be split into small, independent, and loosely coupled services, each responsible for a specific functionality.
- Allow to develop, deploy, and scale these services independently.

Containerization

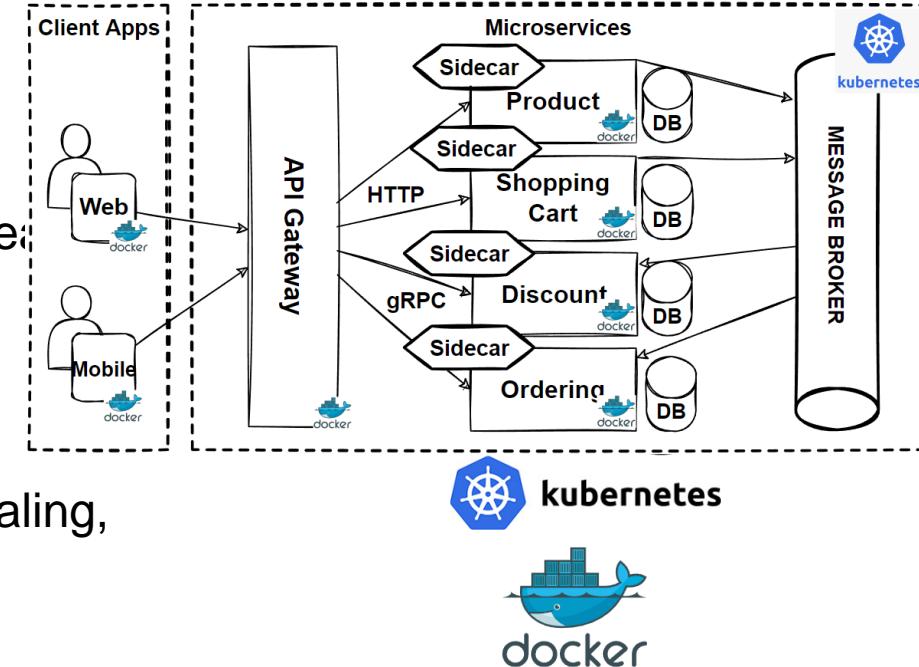
- Package your microservices into lightweight containers, which can be easily deployed and managed in a cloud-native environment.

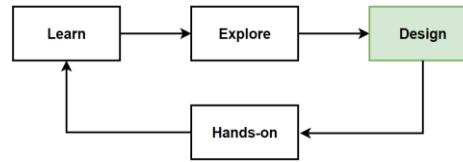
Container Orchestration

- Kubernetes to manage the deployment, scaling, and operation of your containerized microservices. Kubernetes provides features like self-healing, load balancing, and rolling updates.

Istio Service Mesh

- Implement Istio as a service mesh to provide a uniform way to connect, secure, control, and observe microservices. Istio with Envoy proxies, handle traffic management, security, and observability without modifying app code.





Traffic Man > Security > Observability > Ingress

Traffic Management

- Use Istio's traffic management capabilities: load balancing, request routing, and fault injection, to ensure services are highly available and resilient.

Security

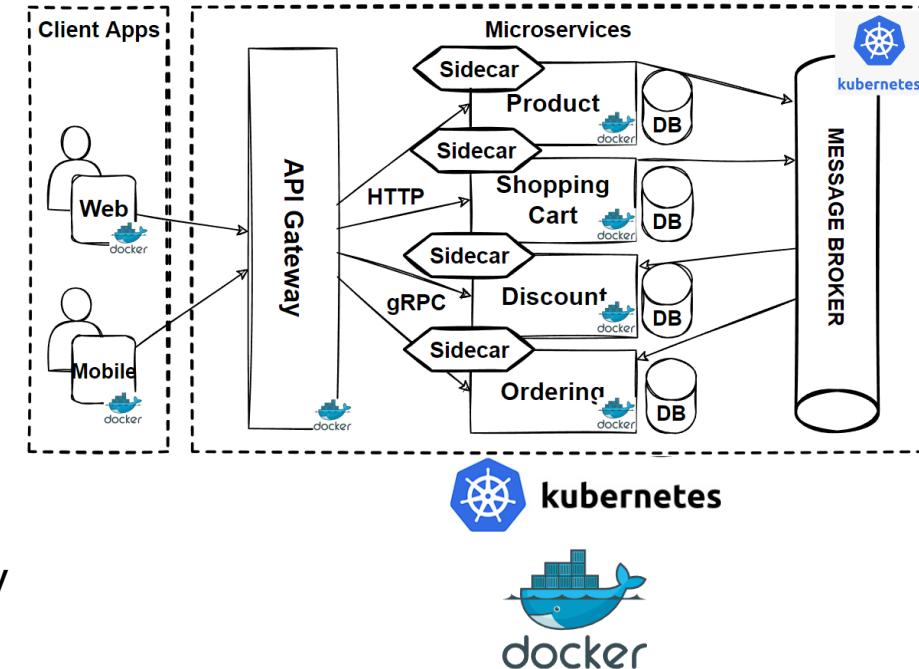
- Leverage Istio's built-in security features, including mutual TLS, authorization policies, and authentication, to secure communication between microservices.

Observability

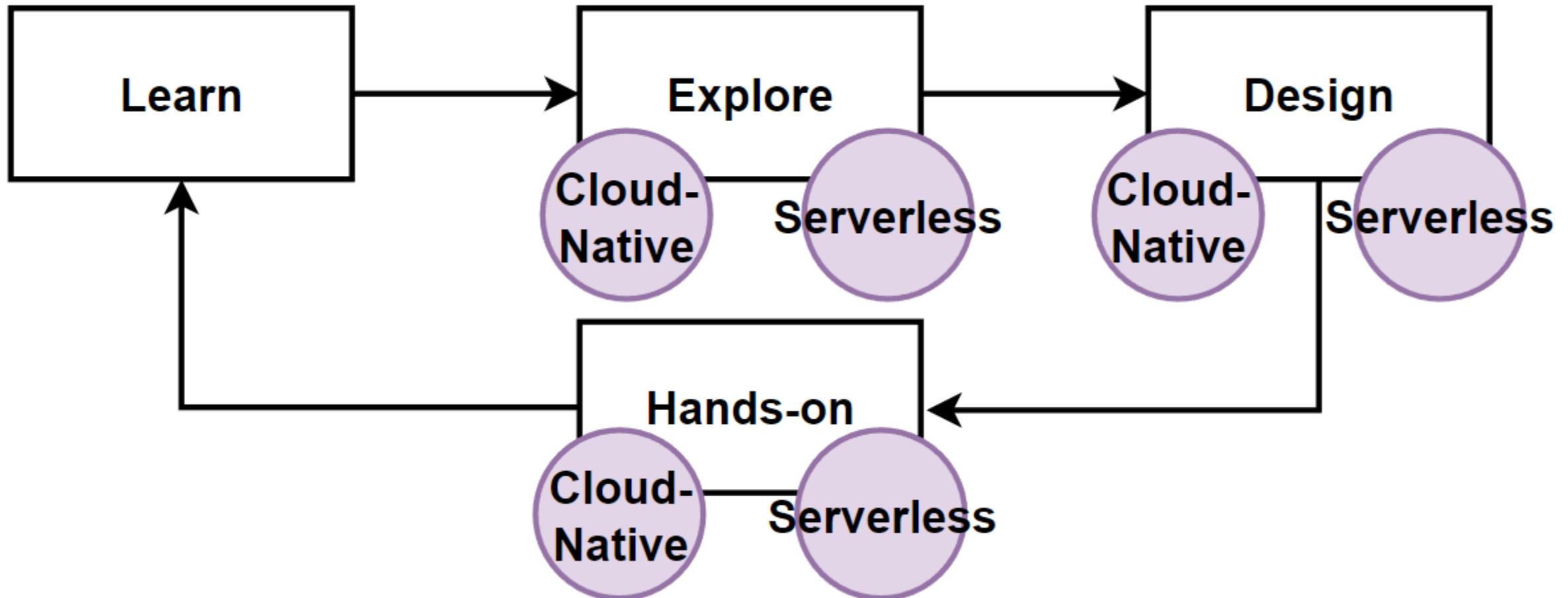
- Utilize Istio's observability features, like distributed tracing, monitoring, and logging, to gain insights into application's performance and identify issues.

Ingress Gateway

- Use an Istio ingress gateway to manage incoming traffic to application, enabling to define routing rules and expose services to the outside world.



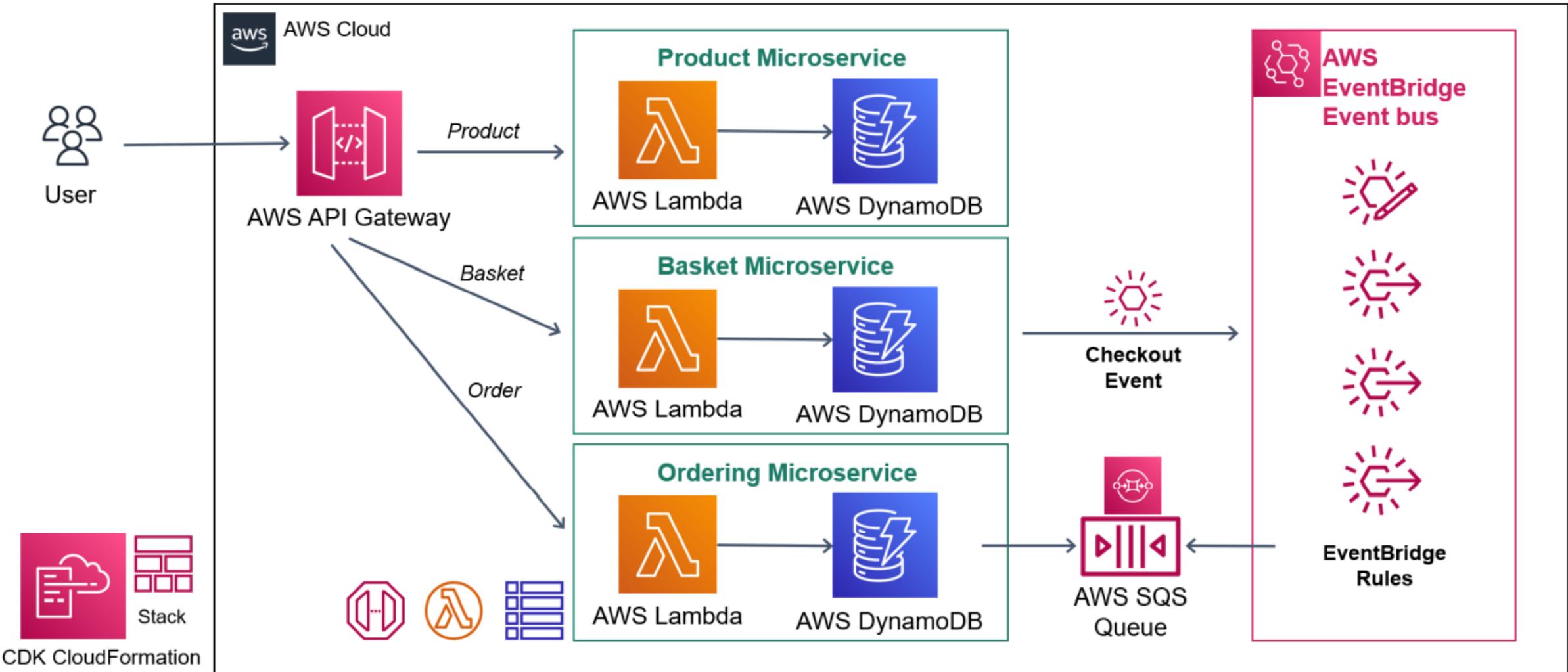
Way of Learning – Cloud-Native & Serverless Cloud Managed



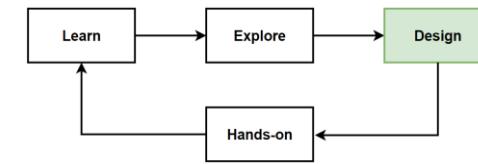
Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

Design with Cloud Managed Services - AWS API Gateway, Service Mesh



- <https://github.com/awsrun/aws-microservices>
- <https://github1s.com/awsrun/aws-microservices>



Microservice > Serverless > Managed API Gateway

Microservices Architecture

- Design app as a collection of small, independent, and loosely coupled microservices, each responsible for a specific functionality.

Serverless Compute

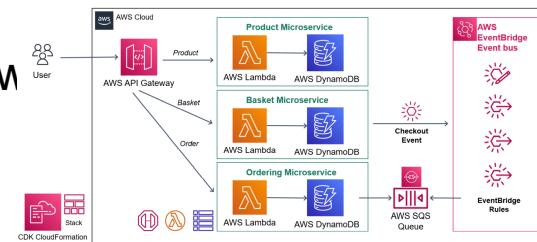
- Use serverless compute services like AWS Lambda, Azure Functions, or Google Cloud Functions to host microservices.
- Automatically manage the scaling, patching, and capacity planning of microservices, allowing focus on application logic.

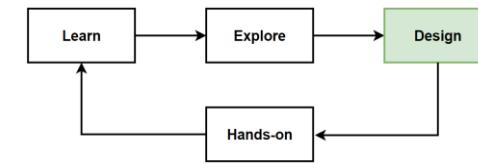
Managed API Gateway

- Utilize managed API Gateway services: Amazon API Gateway, Azure API Management or Google Cloud API Gateway to expose microservices.
- Provide features like request routing, authentication, caching, throttling, and monitoring.

Managed Service Mesh

- Use managed service meshes like AWS App Mesh, Google Cloud Traffic Director, or Azure Service Mesh.
- Integrate with serverless compute services and provide features like traffic routing, load balancing, and end-to-end encryption.





Event-driven > Data Storage > Security > Observability

Event-driven Architecture

- Design microservices to be event-driven and use managed services like Amazon EventBridge, Azure Event Grid, or Google Cloud Pub/Sub for asynchronous communication between services.

Data Storage

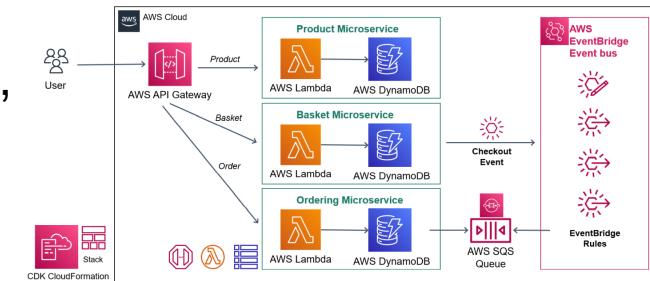
- Use managed serverless storage services: Amazon DynamoDB, Azure Cosmos DB, Google Cloud Firestore to store application data.
- Provide low-latency, scalable, and fully managed data storage options.

Security

- Leverage managed Identity and Access Management (IAM) services provided by the cloud providers to control access to microservices and ensure secure communication.

Observability

- Use managed monitoring, logging, and tracing services: Amazon CloudWatch, Azure Monitor, or Google Cloud Operations Suite to gain insights into application's performance and troubleshoot issues.



Hands-on: Deploy Microservices to Kubernetes with Service Mesh Istio and Envoy

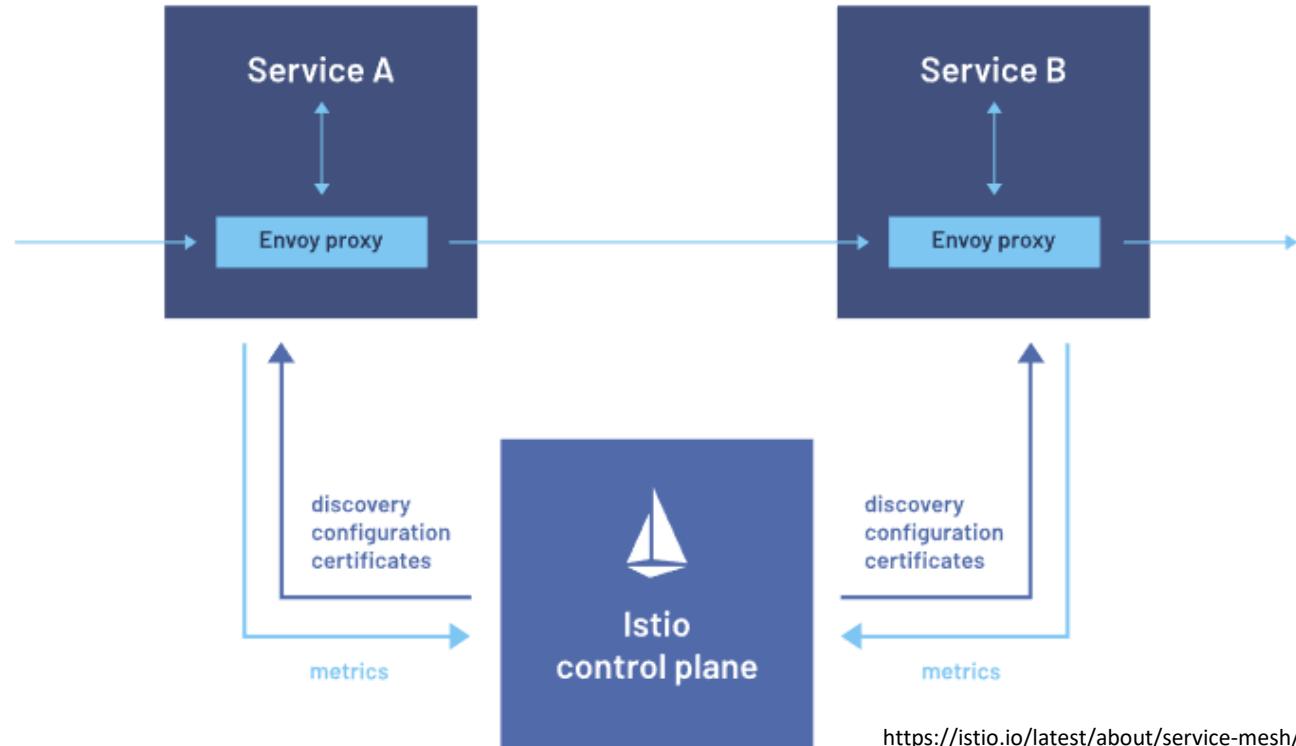
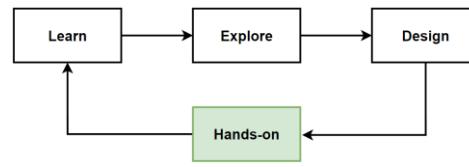
What is Istio Service Mesh

How Istio Service Mesh works

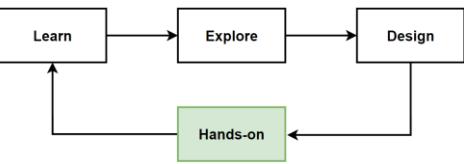
Concepts and Architecture of Istio Service Mesh works

Hands-on: Deploy Microservices to Kubernetes with Service Mesh Istio and Envoy – Task List

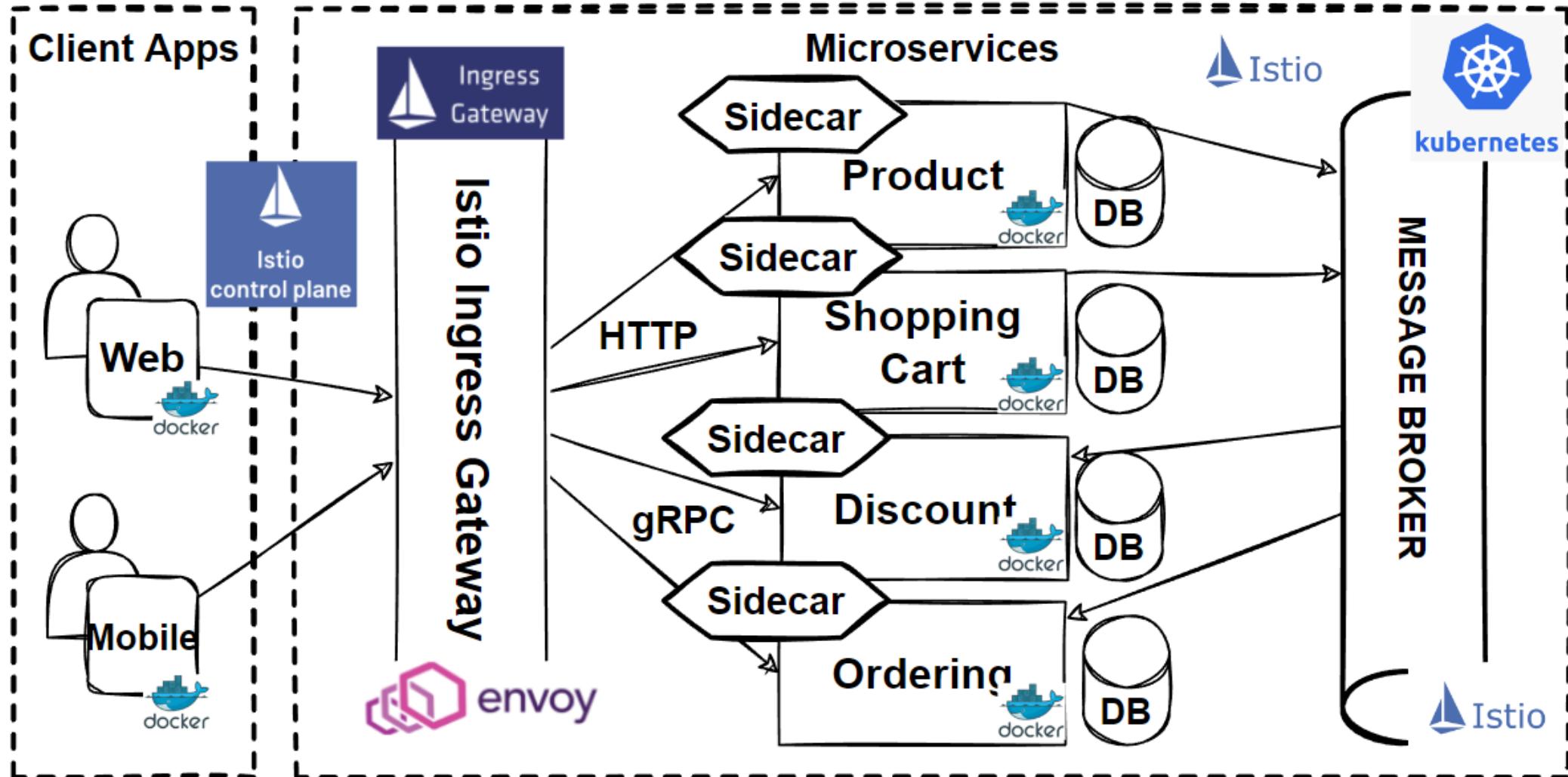
- Getting Started with Istio and Envoy with Minikube
- Download and install Istio Service Mesh onto Minikube K8s cluster
- Deploy the microservices application on Kubernetes with Istio Service Mesh
- Open the application to outside traffic with Istio Ingress Gateway
- Deploy the Kiali dashboard, along with Prometheus, Grafana, and Jaeger

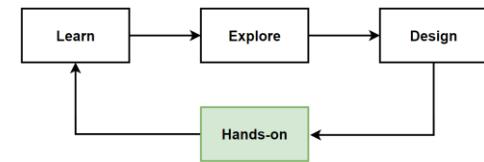


<https://istio.io/latest/about/service-mesh/>



Hands-on: Deploy Microservices to Kubernetes with Service Mesh Istio and Envoy

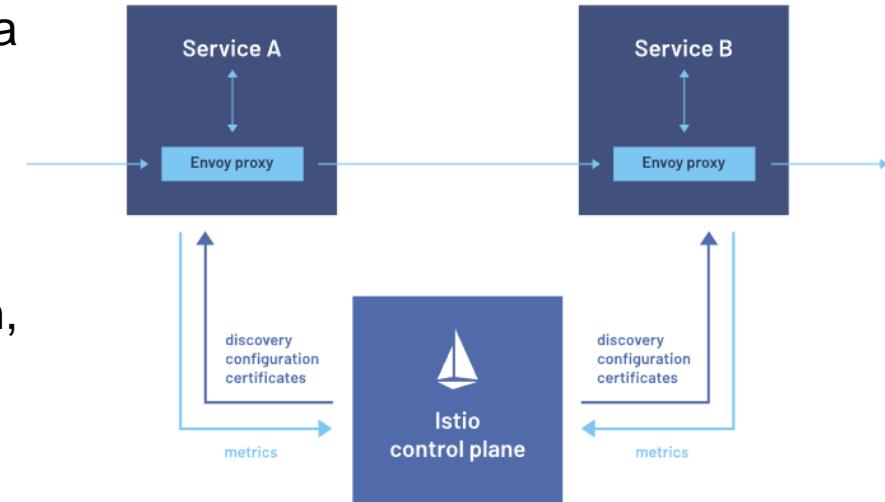




What is Istio Service Mesh

Istio Service Mesh

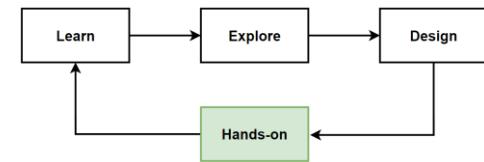
- Istio is an open-source service mesh platform that provides a way to connect, secure, control, and observe services within a microservices-based application.
- Simplify the deployment and management of microservices by providing a uniform way to handle networking, security, traffic management, and observability across services.



Istio Service Mesh Features

- Secure service-to-service communication in a cluster with TLS encryption, identity-based authentication and authorization
- Automatic load balancing for HTTP, gRPC, WebSocket, and TCP traffic
- Fine-grained control of traffic behavior with rich routing rules, retries, failovers, and fault injection
- A pluggable policy layer and configuration API supporting access controls, rate limits and quotas
- Automatic metrics, logs, and traces for all traffic within a cluster, including cluster ingress and egress

<https://istio.io/latest/about/service-mesh/>



Concepts of Istio Service Mesh

Service Mesh

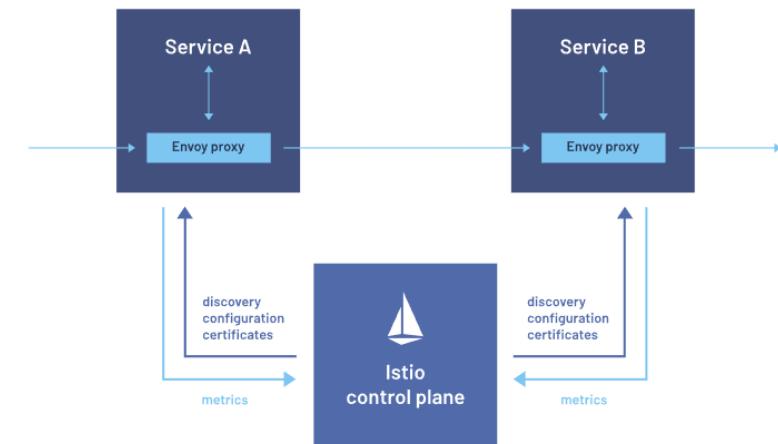
- Dedicated infrastructure layer that sits between the services in a distributed application.
- Consists of a set of interconnected proxies (called sidecars) deployed alongside each service instance. These sidecars intercept and manage network communication between services.

Sidecar Proxies

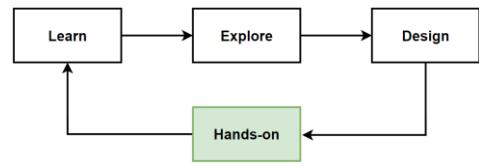
- Additional container called a sidecar proxy (e.g., Envoy Proxy) deployed alongside it. The sidecar proxies handle all the inbound and outbound network traffic for the service, providing features such as load balancing, traffic routing, TLS termination, and telemetry collection.

Data Plane and Control Plane

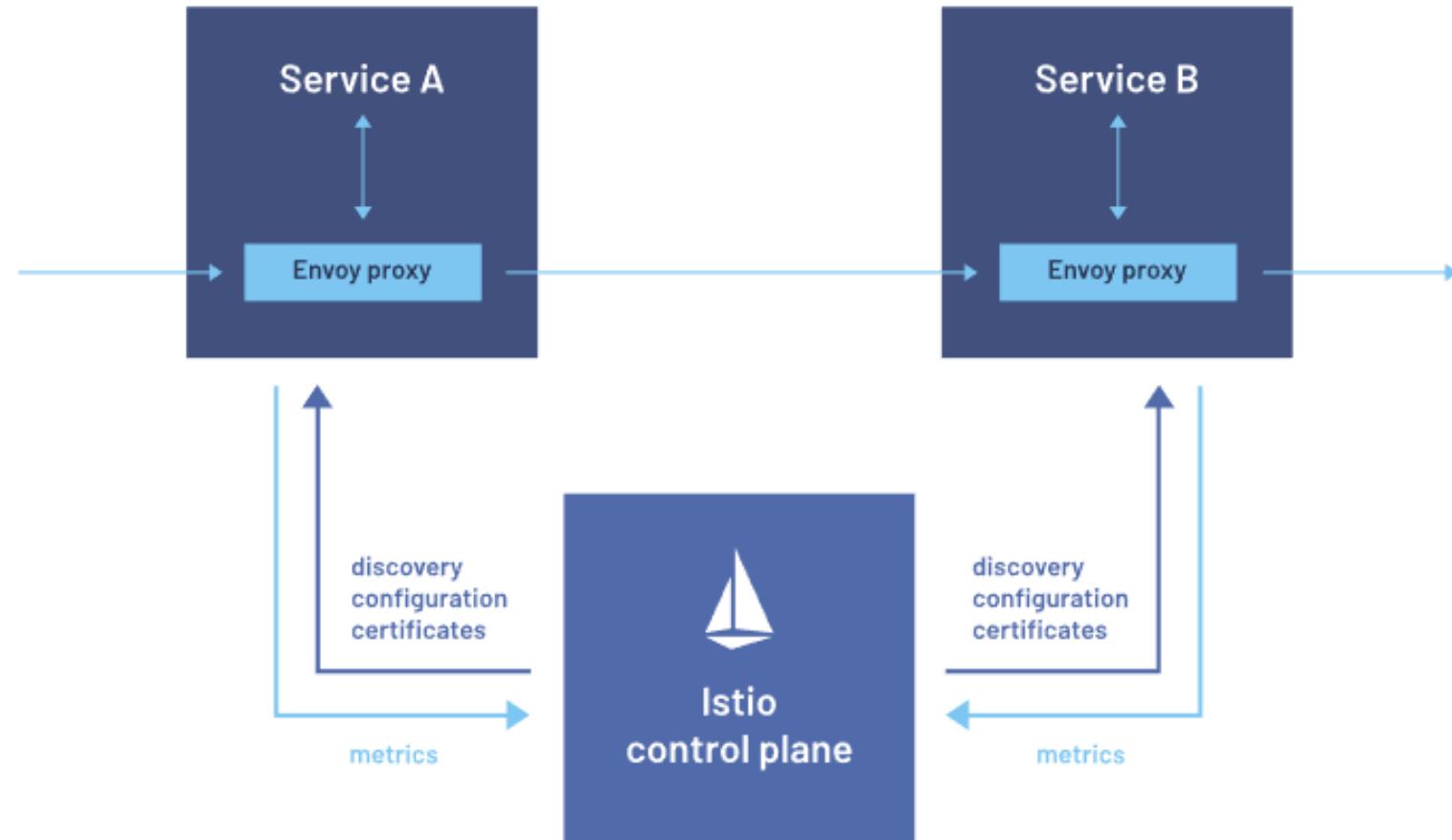
- The data plane includes the sidecar proxies and handles the actual network traffic. The control plane manages and configures the proxies to enforce the desired behavior and policies.



<https://istio.io/latest/about/service-mesh/>



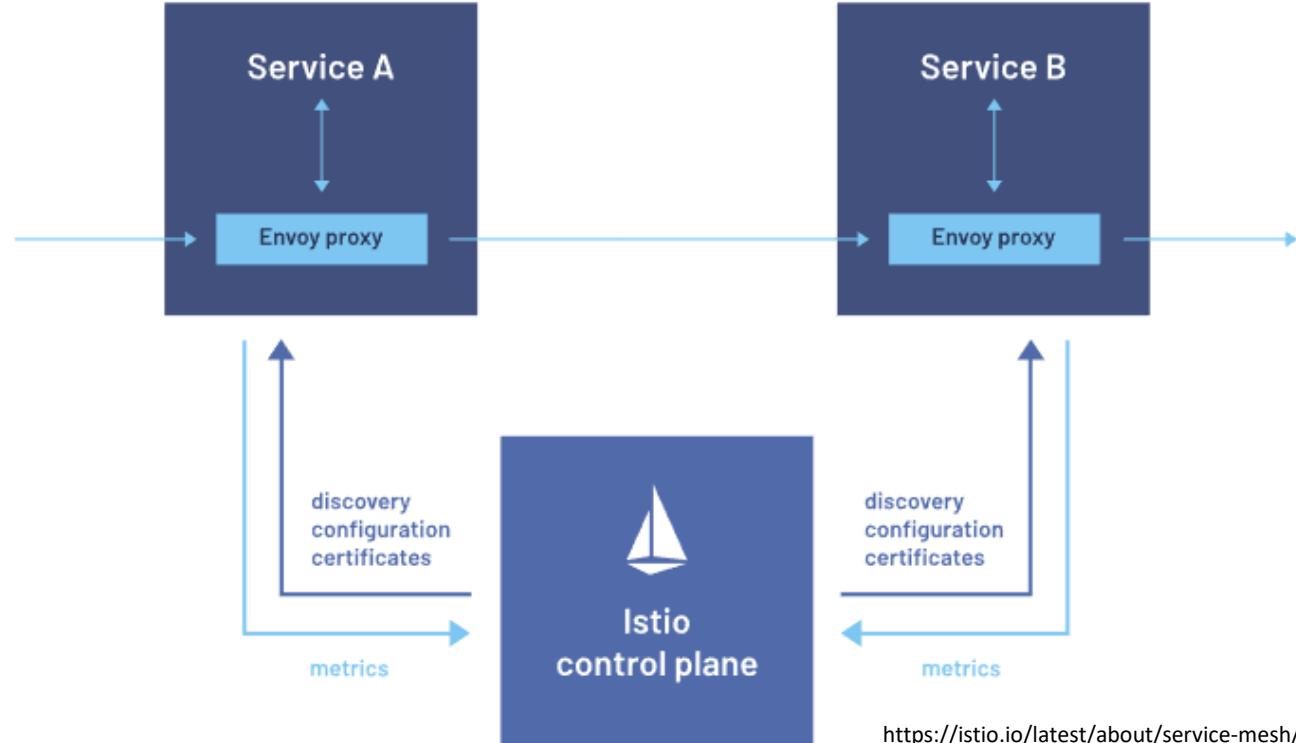
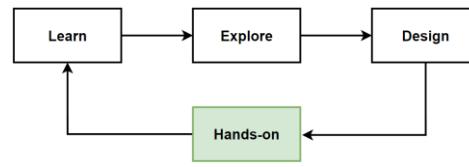
How Istio Service Mesh Works ?

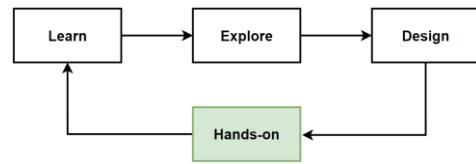


<https://istio.io/latest/about/service-mesh/>

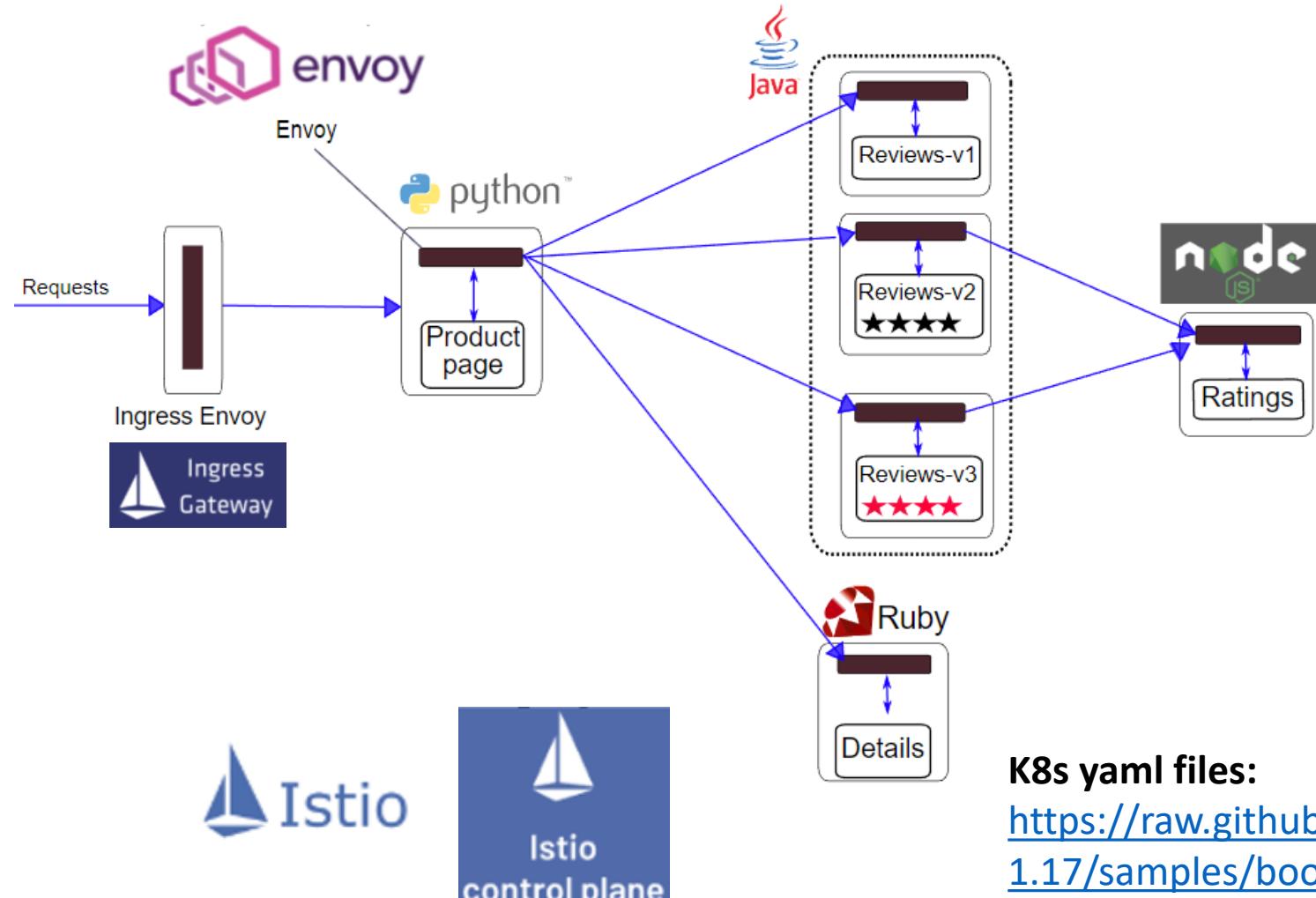
Hands-on: Deploy Microservices to Kubernetes with Service Mesh Istio and Envoy – Task List

- Getting Started with Istio and Envoy with Minikube
- Download and install Istio Service Mesh onto Minikube K8s cluster
- Deploy the microservices application on Kubernetes with Istio Service Mesh
- Open the application to outside traffic with Istio Ingress Gateway
- **Deploy the Kiali dashboard, along with Prometheus, Grafana, and Jaeger**





E2E Architecture of Microservices deploy to Kubernetes with Service Mesh Istio and Envoy



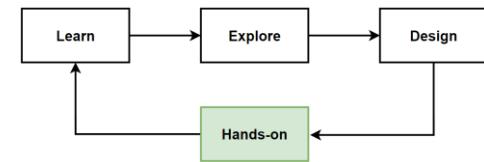
kubernetes



<https://istio.io/latest/docs/examples/bookinfo/>

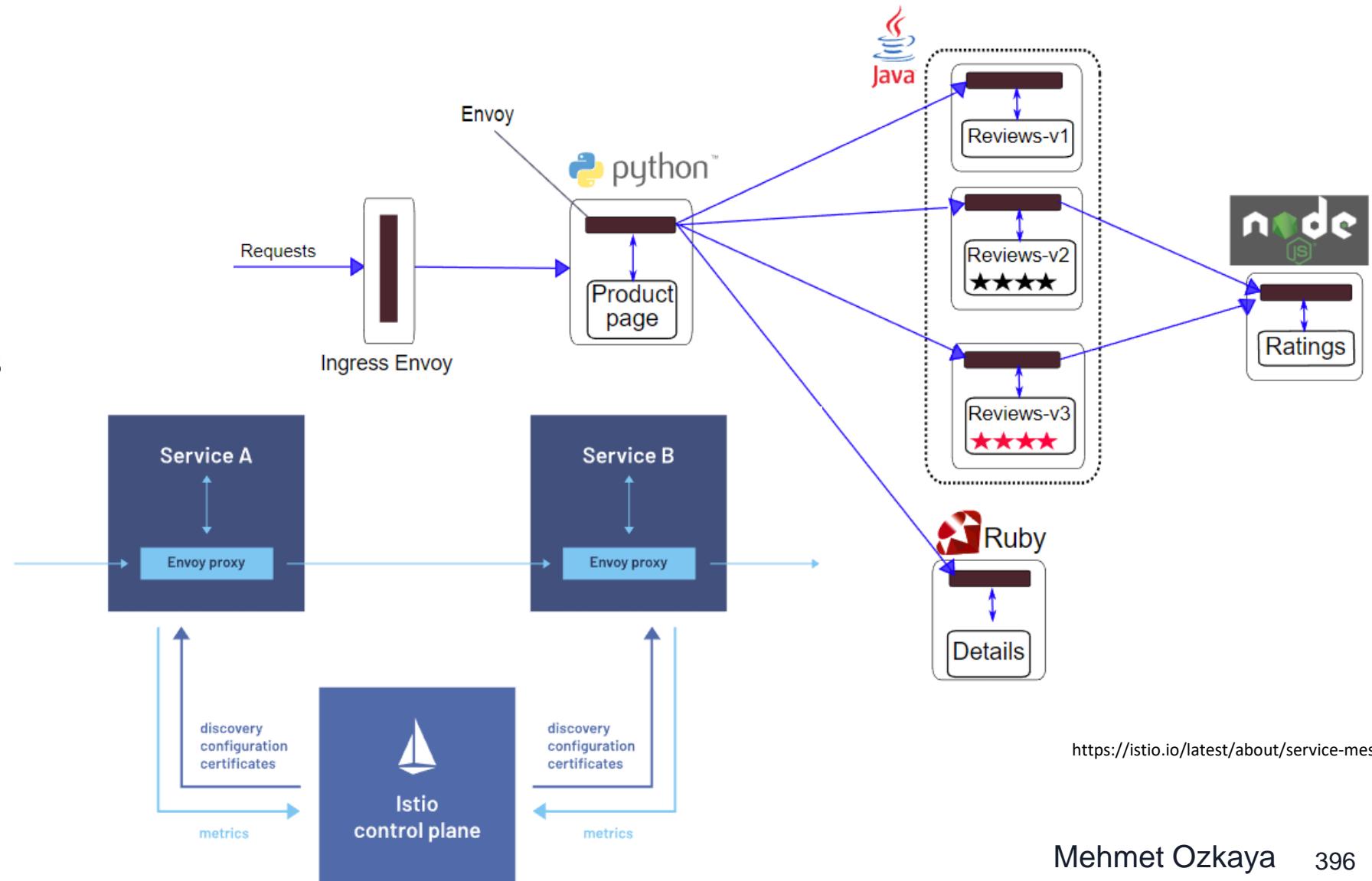
K8s yaml files:

<https://raw.githubusercontent.com/istio/istio/release-1.17/samples/bookinfo/platform/kube/bookinfo.yaml>



What's Next ?

- Request routing
- Fault injection
- Traffic shifting
- Querying metrics
- Visualizing metrics
- Accessing external services
- Visualizing your mesh



Cloud-Native Pillar5: Backing Services - Data Management, Caching, Message Brokers

What are Cloud-Native Backing Services ?

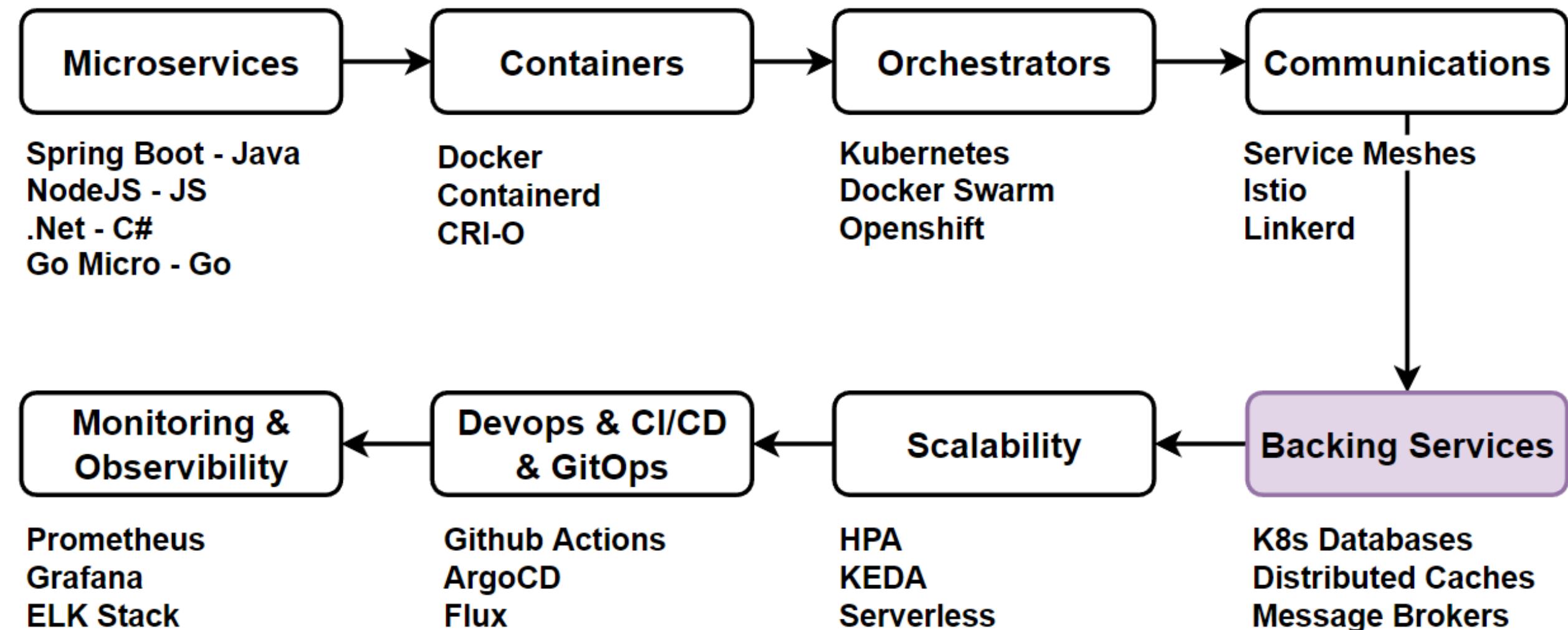
Which Backing Services for Cloud-Native Microservices ?

How microservices use Backing Services in Cloud-Native environments ?

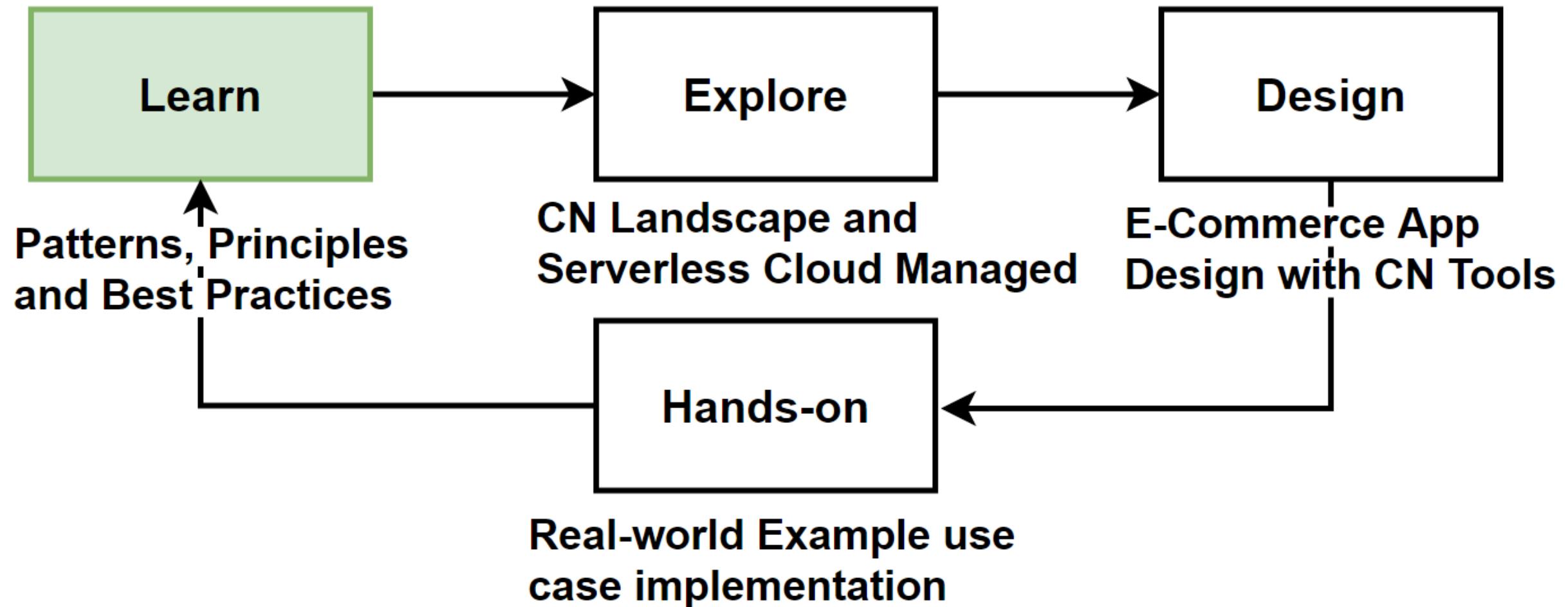
What are patterns & best practices of Backing Services in Cloud-native apps ?

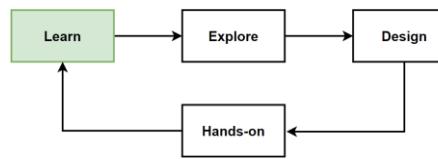
Implement Hands-on Development of Backing Services in Cloud-native microservices

Cloud-Native Pillars Map – The Course Section Map



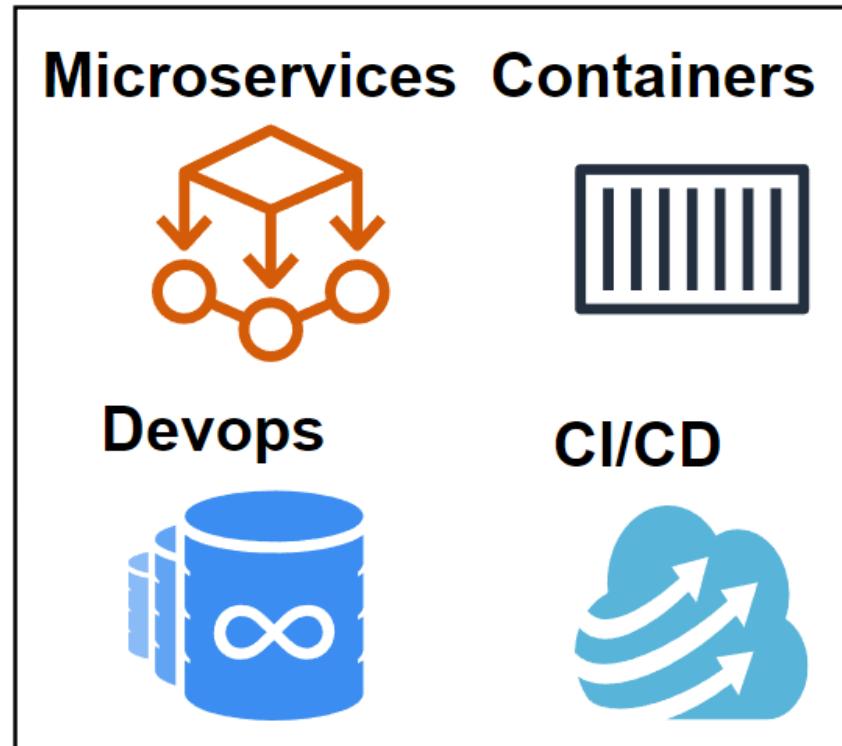
Way of Learning – The Course Flow

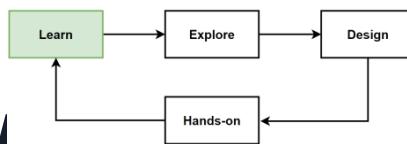




Learn: The 5. Pillar – Cloud-Native Backing Services

- Cloud-Native Backing Services:
 - Databases (e.g. MySQL, PostgreSQL, Cloud Spanner)
 - File storage (e.g. NFS, FTP, Cloud Filestore)
 - Message Brokers, Enterprise Message Queues
 - Streaming Services
 - Distributed Caches
 - Monitoring and Analytic tools
 - Security Identity Services
 - Logging services (e.g. syslog endpoints, Cloud Logging)
- Which Backing Services for Cloud-Native Microservices ?
- How microservices use Backing Services in Cloud-Native environments ?
- Best practices of using Backing Services in Cloud-native environments
- Backing Services Patterns in CN microservices
- Benefits and Challenges of Backing Services
- Explore - Backing Services for Databases, Caching, Message Brokers.
 - K8s and Serverless Databases
 - K8s and Serverless Caching
 - Message Brokers Kafka, EventBridge





Where «Backing Services» in 12-Factor App and CN Trial Map

Twelve-Factor App

- Codebase
- Dependencies
- Config
- Backing Services**
- Build, release and run
- Processes
- Port Binding
- Concurrency
- Disposability
- Development/Prod Parity
- Logs
- Admin Processes



CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape <https://github.com/cncf/landscape> has a growing number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator <https://www.cncf.io/training>

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider <http://cncf.io/ksp>

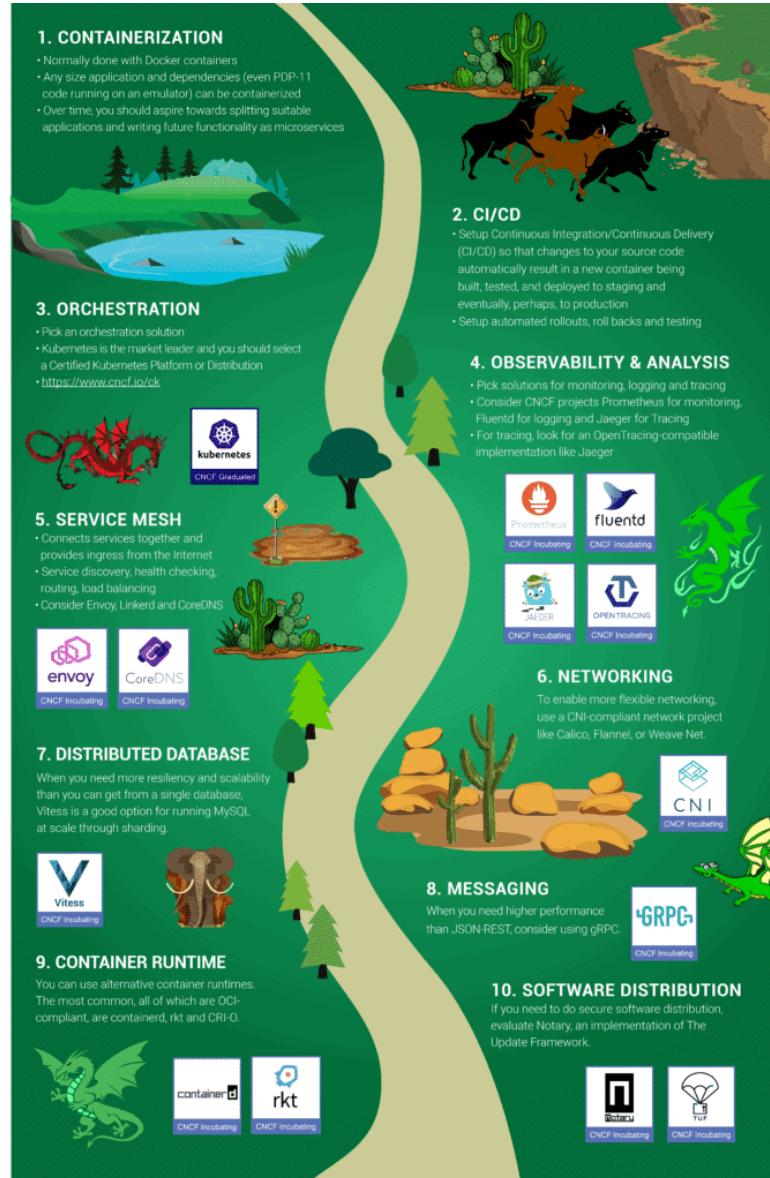
C. Join CNCF's End User Community

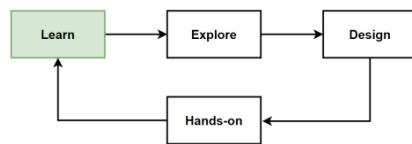
For companies that don't offer cloud native services externally <http://cncf.io/enduser>

WHAT IS CLOUD NATIVE?

- **Operability:** Expose control of application/system lifecycle.
- **Observability:** Provide meaningful signals for observing state, health, and performance.
- **Elasticity:** Grow and shrink to fit in available resources and to meet fluctuating demand.
- **Resilience:** Fast automatic recovery from failures.
- **Agility:** Fast deployment, iteration, and reconfiguration.

www.cncf.io
info@cncf.io





12-Factor App – Cloud-Native Backing Services

Factor IV: Treat backing services as attached resources

- Treat it as an attached resource that can be easily swapped or replaced without modifying the application code.
- Enhances flexibility and maintainability, allowing developers to switch between different backing service instances or providers without making code changes.

Use uniform access

- All backing services are accessed via a uniform mechanism, like a URL or connection string that helps in decoupling the application code from the backing service implementation, allowing for easy replacement or updates.

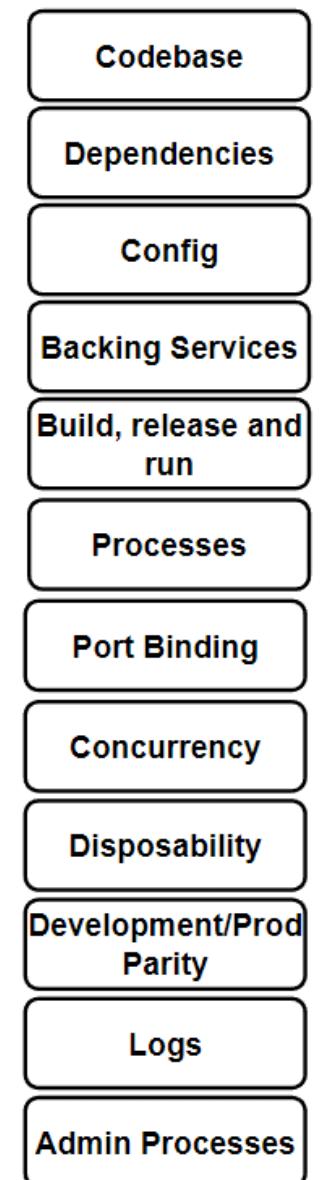
Externalize configuration

- Store connection details for backing services, like credentials, URLs, and ports, separately from the application code. Instead, use environment variables or configuration files to store this information.

Loosely couple with backing services

- Design app to be loosely coupled with its backing services so that the app should be resilient to backing service failures and able to recover gracefully.

Twelve-Factor App



Cloud-native Trial Map – Backing Services

Distributed Databases

- Managed relational databases like Amazon RDS and Google Cloud SQL, NoSQL databases like Amazon DynamoDB and Google Cloud Firestore, and cloud-native databases like CockroachDB and Vitess.

Messaging and Eventing Systems

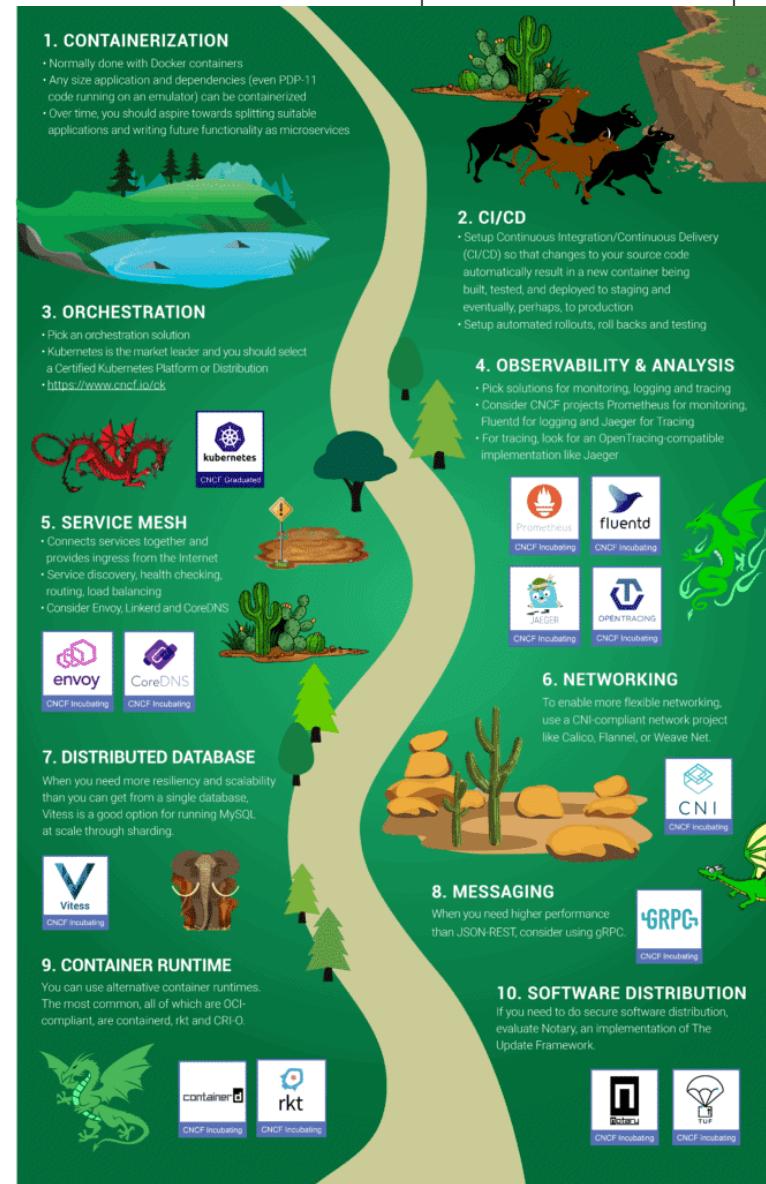
- Apache Kafka, NATS, and RabbitMQ are popular for building event-driven architectures in cloud-native applications.
- Amazon SQS/SNS, Google Cloud Pub/Sub, and Azure Service Bus.

Caching

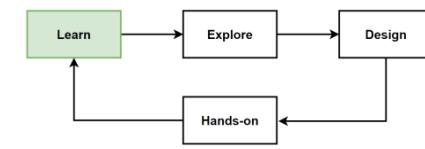
- Redis and Memcached are often used as backing services providing caching for cloud-native applications.
- These can be run on Kubernetes, or used via managed services like Amazon ElastiCache and Google Cloud Memorystore.

Logging, Monitoring, and Tracing

- Backing services are needed to capture and analyze logs, metrics, and traces from your applications.
- Prometheus for monitoring, Fluentd or Logstash for logging, and Jaeger for tracing



<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>



Kubernetes Growth Areas are Open Source Databases, CI/CD technologies and Security

- [Dynatrace Kubernetes Report: Strongest Kubernetes growth areas](#)

▪ Security

55% growth rate, Kubernetes security tools are becoming more important for organizations. These tools help ensure the security and compliance of backing services, such as databases, message brokers, and caches

▪ Databases and Caches

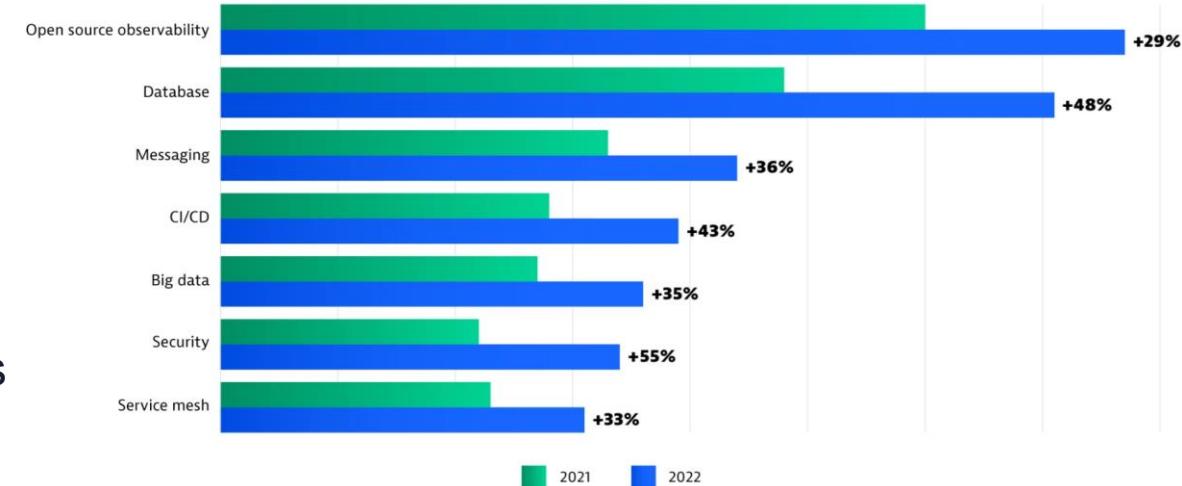
48% increase in the use of databases and caches within Kubernetes shows that organizations are focusing on stateful applications and the importance of persistence.

▪ CI/CD Technologies

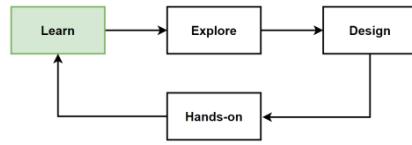
43% growth of CI/CD technologies in Kubernetes clusters highlights the trend of organizations leveraging Kubernetes for their software build, test, and deployment pipelines.

▪ Messaging

36% growth of Messaging technologies in Kubernetes, provide asynchronous communications within microservices architectures and high-throughput distributed systems.

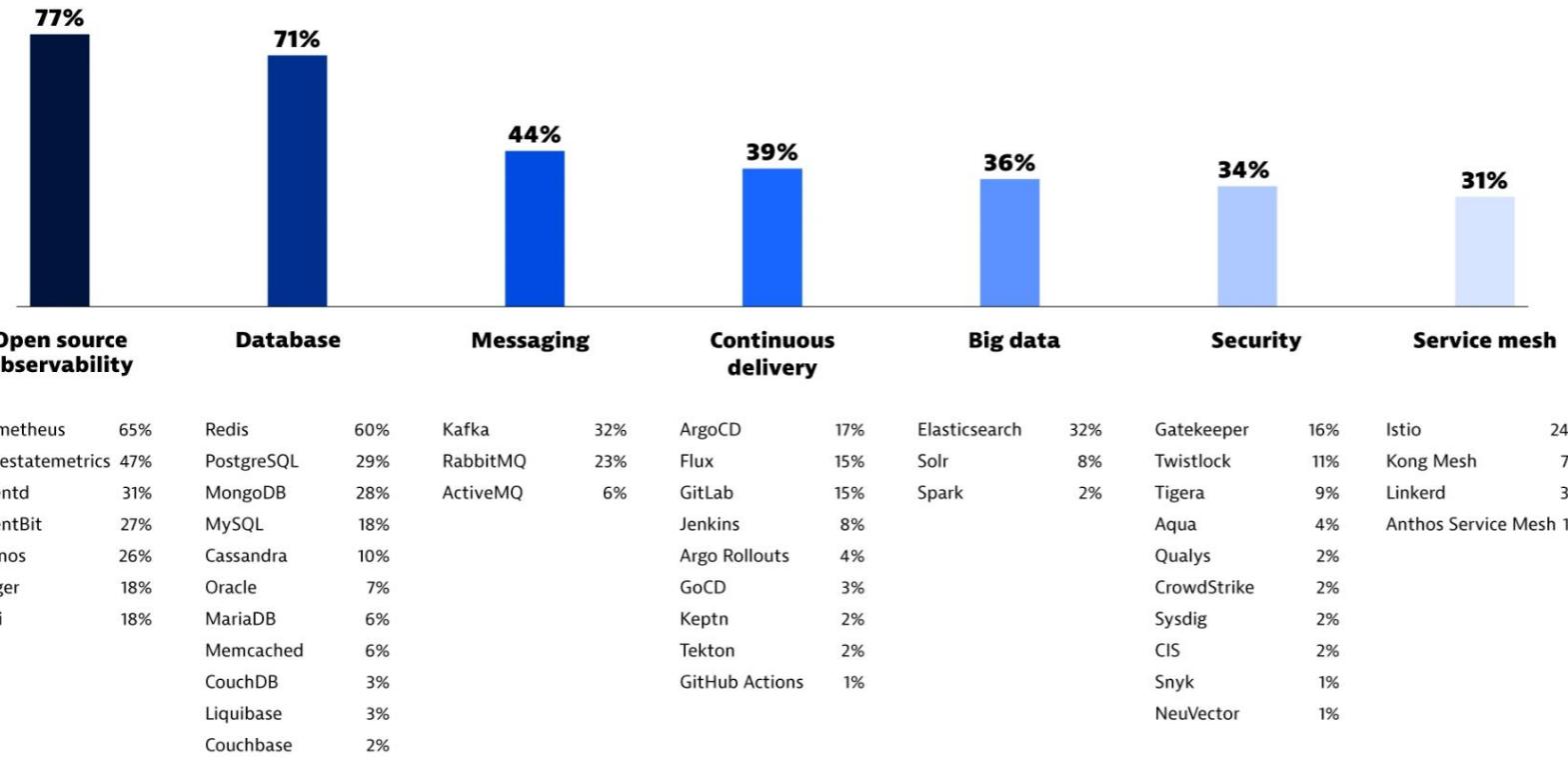


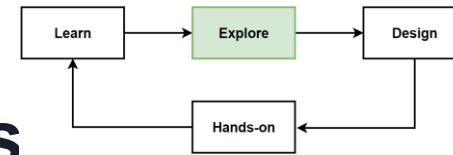
<https://www.dynatrace.com/news/blog/kubernetes-in-the-wild-2023/>



Open source software drives a vibrant Kubernetes

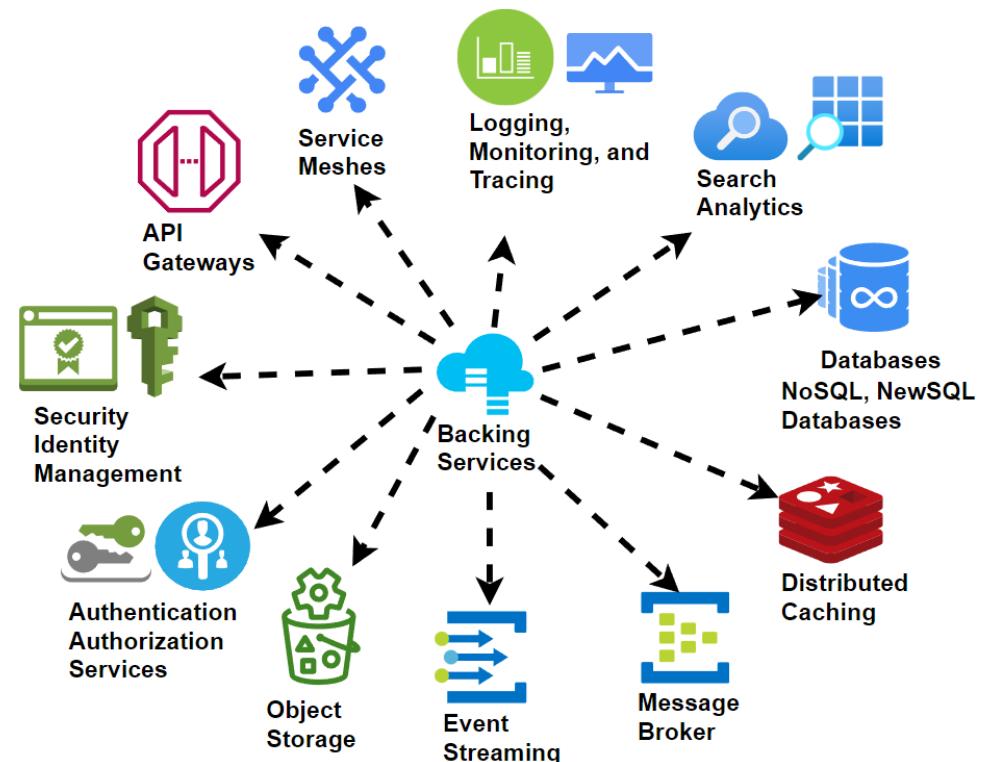
- Open-source projects in the **Kubernetes** across various categories, including **observability**, **databases**, **messaging**, **CI/CD**, **big data**, **security**, and **service meshes**.
- Open-source solutions: Prometheus, Redis, RabbitMQ, Kafka, ArgoCD, Flux, GitLab, Jenkins, Elasticsearch, Gatekeeper, and Istio, are widely used as backing services or tools to enhance the capabilities of cloud-native applications.

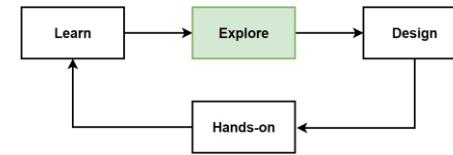




What is Backing Services for Cloud-Native Microservices

- **External components** that microservices depend on for their operation.
- Provide support for various functionalities, such as **data storage, messaging, caching, and authentication**.
- **Decoupled from the microservices** themselves, promoting flexibility, scalability, and easier maintenance.
- Backing services are **external resources** that a cloud-native microservices application **depends on** to function.
- Typically **managed by the cloud provider**, allowing developers to **focus on building and deploying** their microservices **without worrying about the operational overhead** of managing the underlying infrastructure.





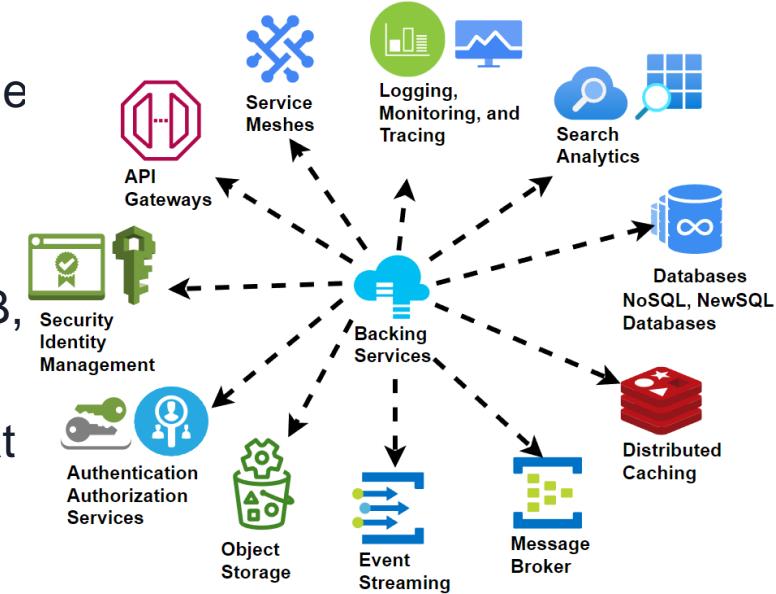
Backing Services for Cloud-Native Microservices

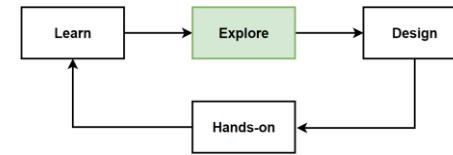
Databases

- Popular databases used in cloud-native apps include relational databases (PostgreSQL, MySQL), NoSQL databases (MongoDB, Cassandra), and in-memory databases (Redis).
- Managed cloud databases like Amazon RDS, Azure SQL Database, and Google Cloud SQL provide scalable, resilient, and fully managed data storage options.

NoSQL and NewSQL Databases

- Managed cloud NoSQL databases like Amazon DynamoDB, Azure Cosmos DB, or Google Cloud Firestore are suitable choices.
- NewSQL Databases are modern relational database management systems that aim to combine the best features of traditional SQL databases and NoSQL databases.
- Examples of NewSQL databases include Google Spanner, CockroachDB, VoltDB, and TiDB.





Backing Services for Cloud-Native Microservices2

Distributed Caches

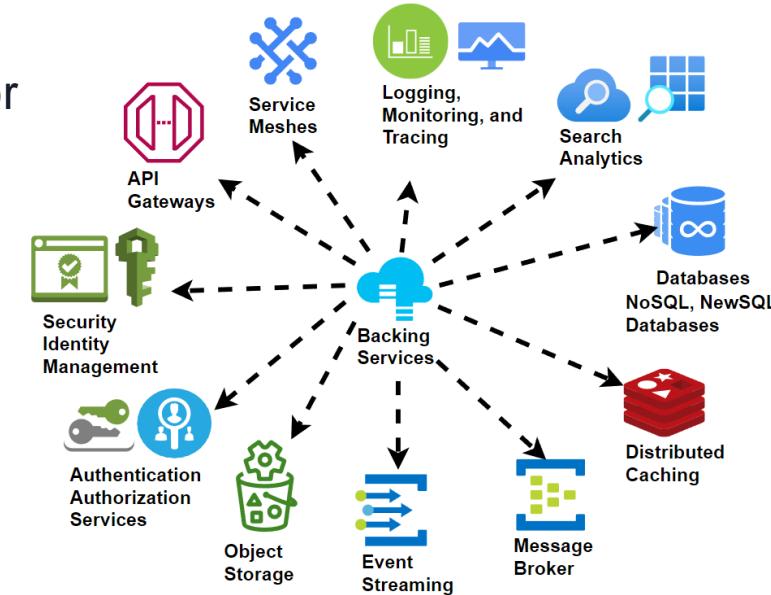
- Store frequently accessed data to improve microservice performance and reduce resource-intensive operations like database queries. Redis and Memcached are popular caching systems.
- Managed caching services like Amazon ElastiCache, Azure Cache for Redis, or Google Cloud Memorystore provide in-memory data storage for faster data access and reduced latency, improving the performance of microservices.

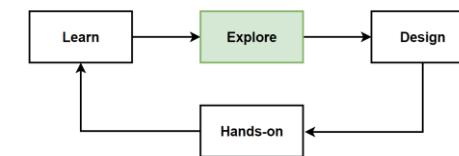
Message Brokers

- Enable asynchronous communication between microservices, allowing to exchange messages or events. This improves scalability and decouples microservices.
- Examples of message brokers include RabbitMQ and Apache Kafka.

Messaging and Event Streaming

- Managed cloud messaging and event streaming services like Amazon SQS, Amazon SNS, Amazon Kinesis, Azure Service Bus, Azure Event Hubs, or Google Cloud Pub/Sub enable asynchronous communication between microservices.





Backing Services for Cloud-Native Microservices3

Object Storage

- Managed object storage services: Amazon S3, Azure Blob Storage, or Google Cloud Storage provide highly scalable, durable, and cost-effective storage options for storing and retrieving unstructured data like images, videos.

Authentication and Authorization Services

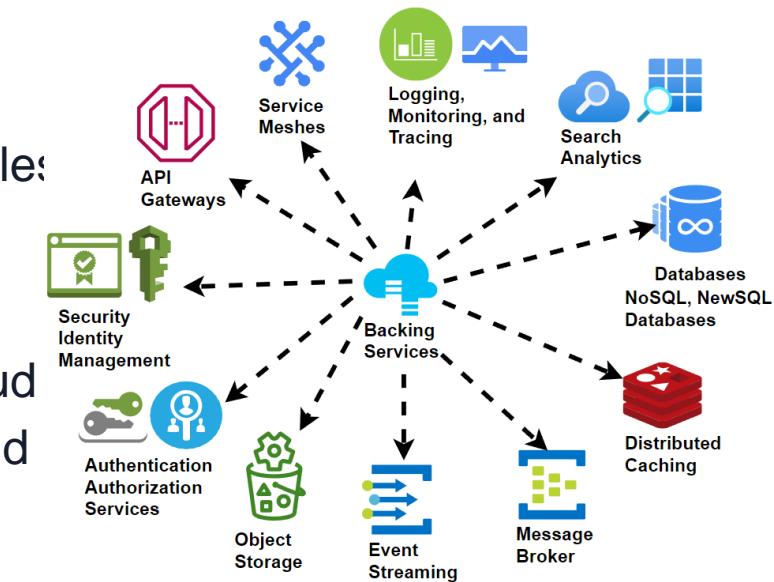
- User authentication and access control management for microservices. Examples include OAuth 2.0, OpenID Connect, and LDAP.

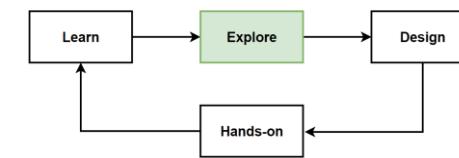
Security and Identity Management

- Managed Identity and Access Management (IAM) services provided by the cloud providers help control access to your microservices, secure communication, and manage authentication and authorization.

API Gateways

- Load balancing, authentication, rate limiting, and request routing. Examples include Kong, Ambassador, and Amazon API Gateway.
- Managed API Gateway services like Amazon API Gateway, Azure API Management, or Google Cloud API Gateway provide request routing, authentication, caching, throttling, and monitoring for your microservices apis.





Backing Services for Cloud-Native Microservices4

Service Meshes

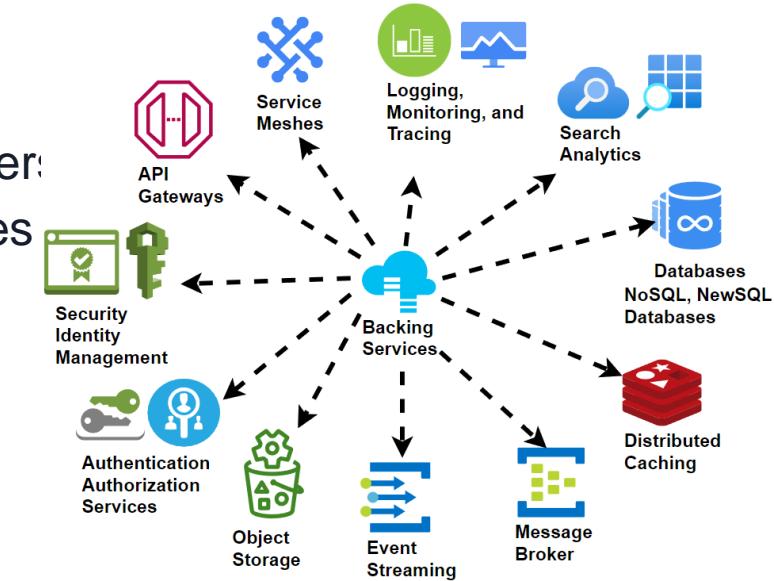
- Infrastructure layer for managing and controlling communication between microservices, handling tasks like load balancing, service discovery, and observability. Examples of service meshes include Istio, Linkerd, and Kuma.

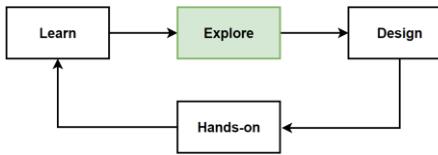
Logging, Monitoring, and Tracing

- Collect, store, and analyze logs and metrics for microservices, helping developers and operators gain insights into performance and troubleshoot issues. Examples include Elasticsearch, Logstash, Kibana (ELK stack), Prometheus, and Grafana.
- Managed services like Amazon CloudWatch, Azure Monitor, or Google Cloud

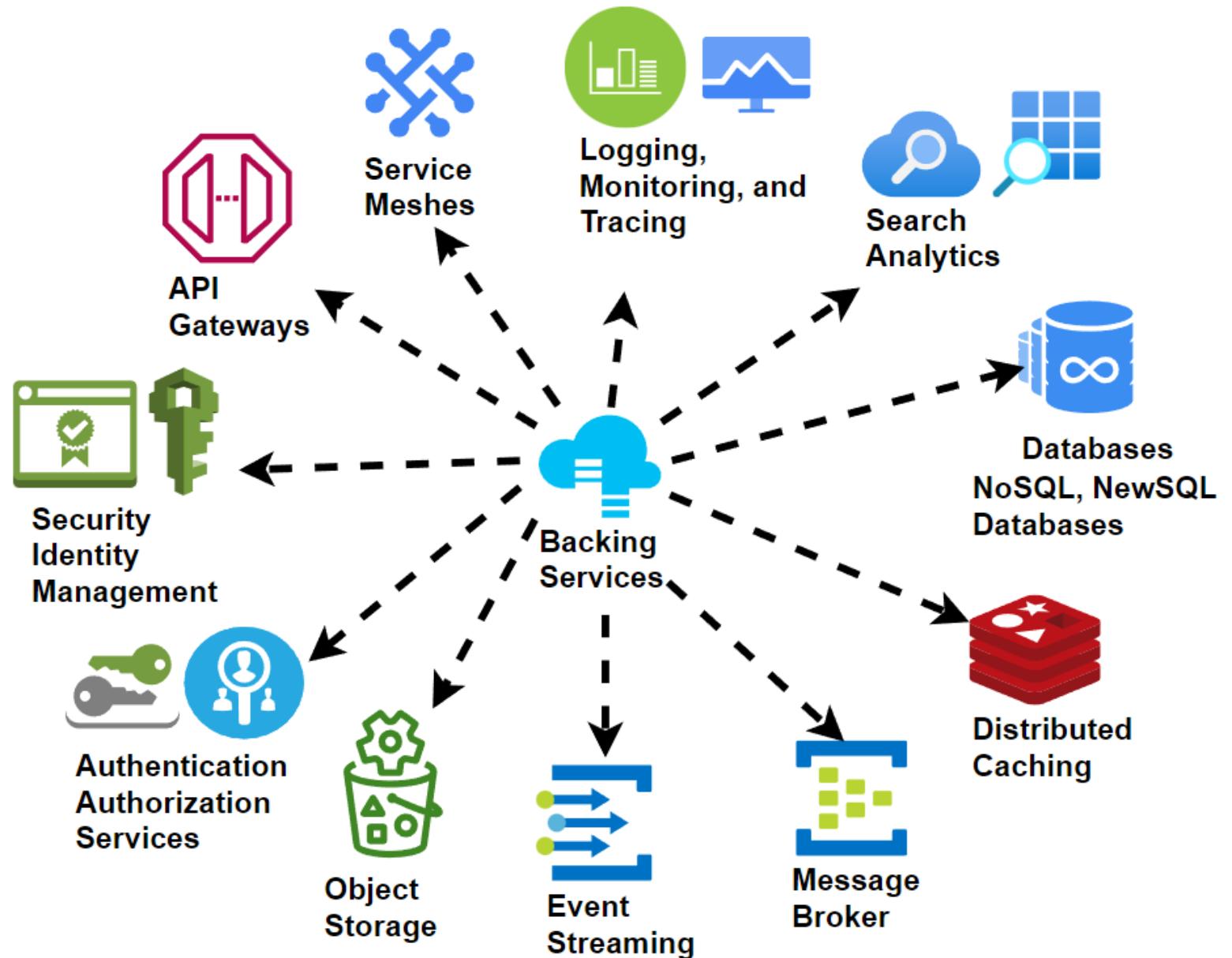
Search and Analytics

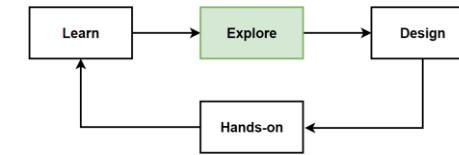
- Process, analyze, and search large datasets. Elasticsearch is a popular choice for this purpose.
- Managed search services like Amazon Elasticsearch, Azure Cognitive Search, or Google Cloud Search provide indexing and searching capabilities for microservices.





Backing Services for Cloud-Native Microservices





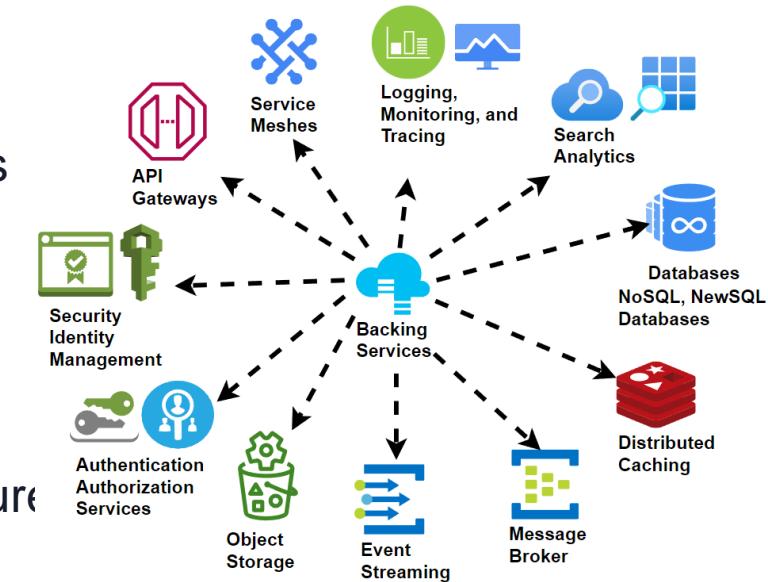
Which Backing Services we focus on ?

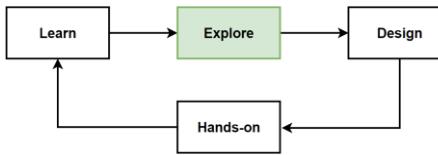
Cloud-Native Backing Services – Course Cover

- Data Management (K8s and Serverless Databases)
- Caching (K8s and Serverless Caching)
- Message Brokers (Kafka, EventBridge)

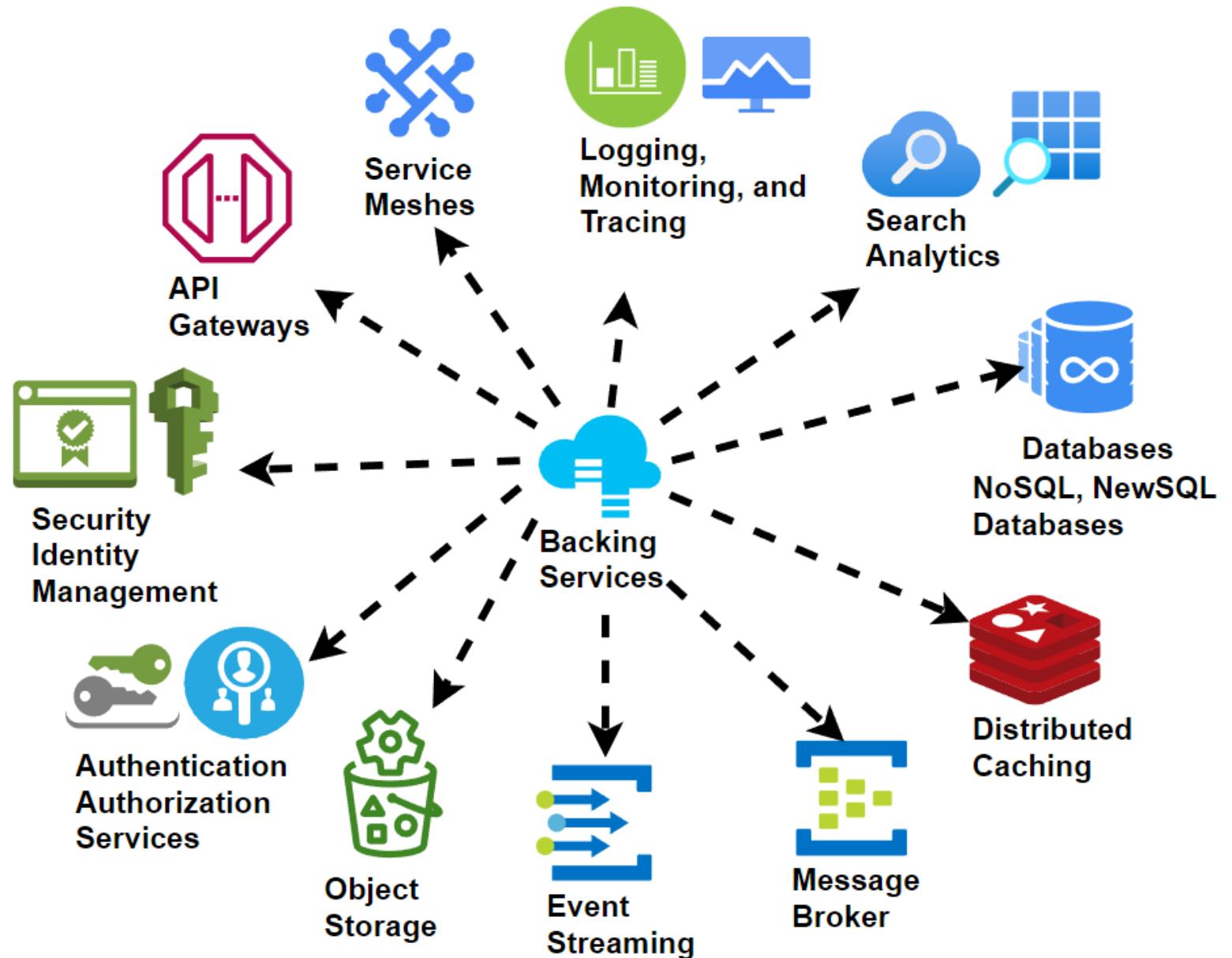
Explore Open source CN Tools & Managed Cloud Tools

- Examine open source CNCF Cloud-native tools and Managed Cloud serverless tools.
- **NoSQL Databases** - MongoDB, Cassandra
- **Cloud Managed NoSQL DB** - Amazon DynamoDB, Azure Cosmos DB
- **Message Brokers** - Kafka, RabbitMQ
- **Cloud Message Brokers** - Amazon SNS, EventBridge, Azure Service Bus, Azure Event Hubs, Google Cloud Pub/Sub





Backing Services for Cloud-Native Microservices



Cloud-Native Pillar5: Backing Services - Data Management (K8s and Serverless Databases)

Relational, NoSQL and NewSQL Databases

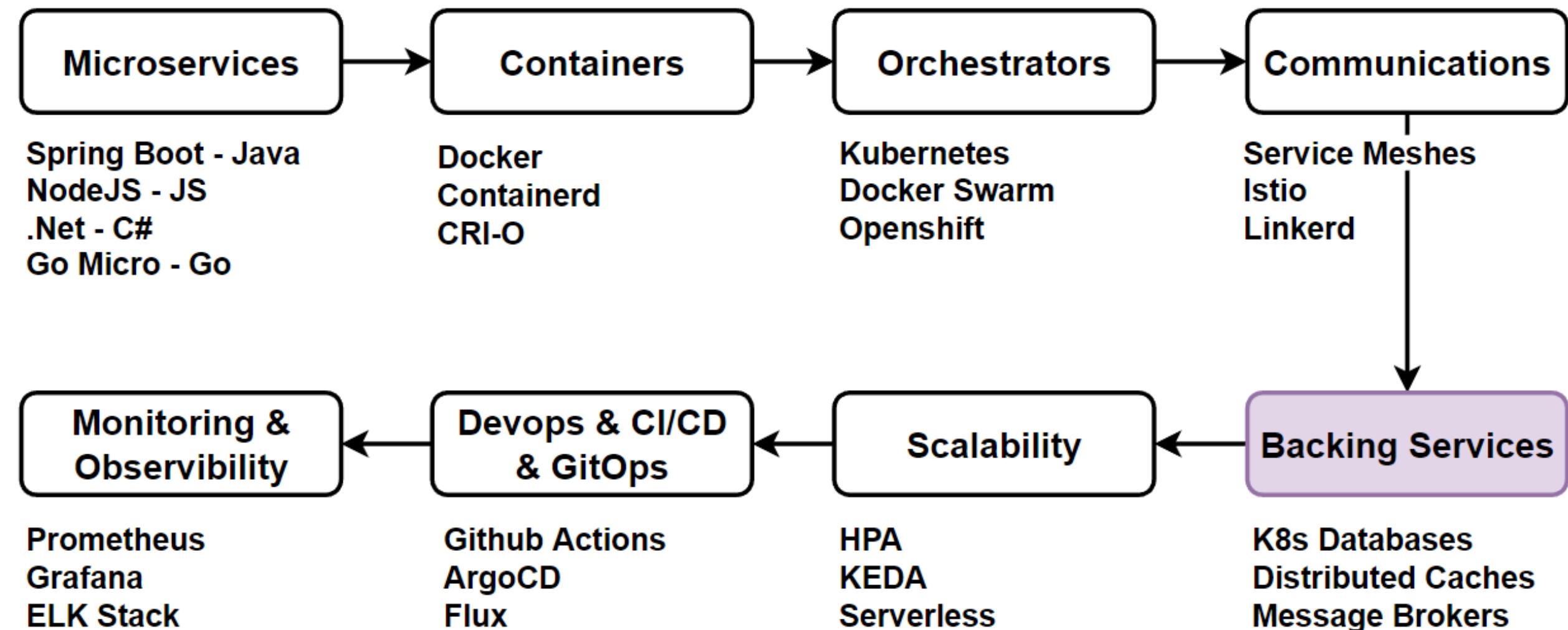
When to Use Relational / NoSQL / NewSQL Databases ?

Which Databases should select for Backing Services for Cloud-Native Microservices ?

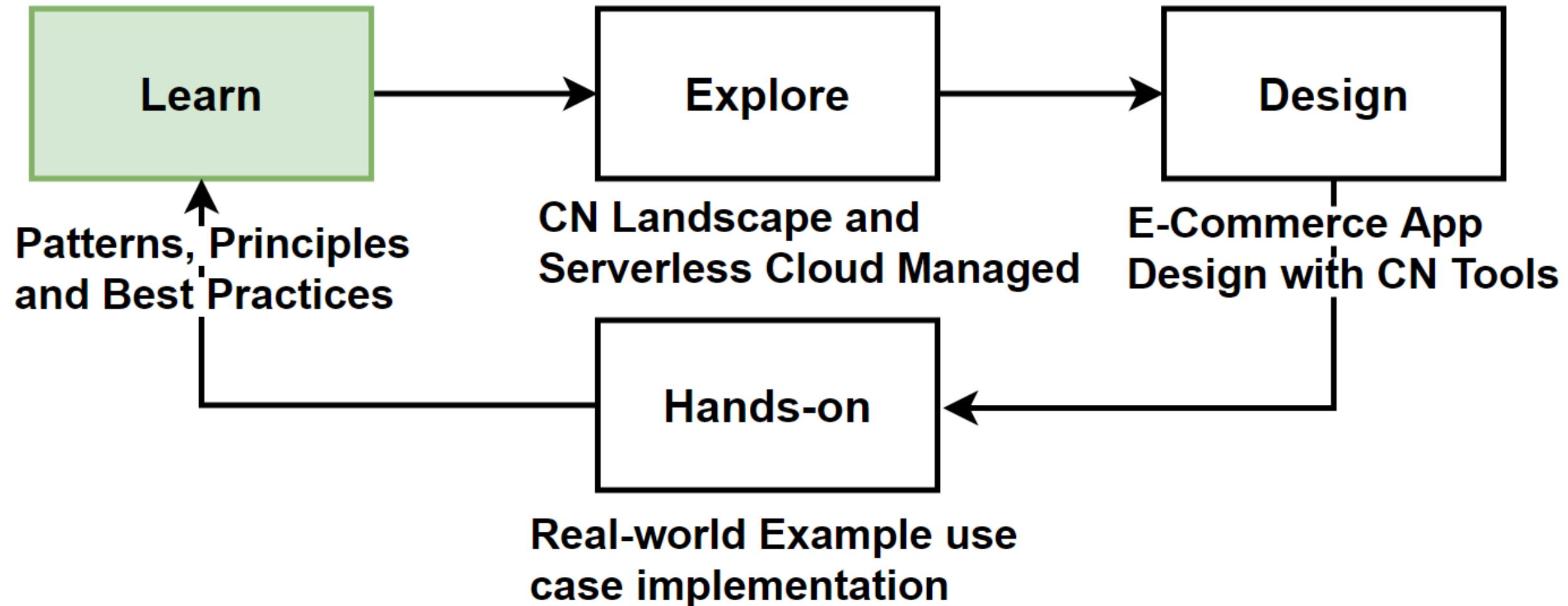
How microservices use Databases in Cloud-Native environments ?

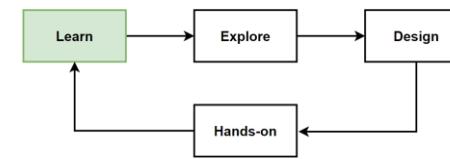
What are patterns & best practices of using Databases in Cloud-native environments?

Cloud-Native Pillars Map – The Course Section Map



Way of Learning – The Course Flow

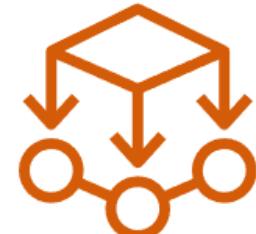




Backing Services - Data Management

- Cloud-Native Backing Services – Data Management :
 - Databases (e.g. MySQL, PostgreSQL, Cloud Spanner)
- Relational, NoSQL and NewSQL Databases
- Explore - Backing Services for Databases.
 - K8s and Serverless Databases
- Hands-on - Backing Services for Databases.
 - K8s and Serverless Databases

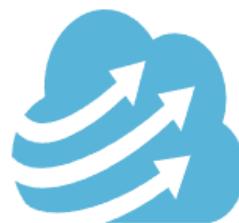
Microservices Containers



Devops

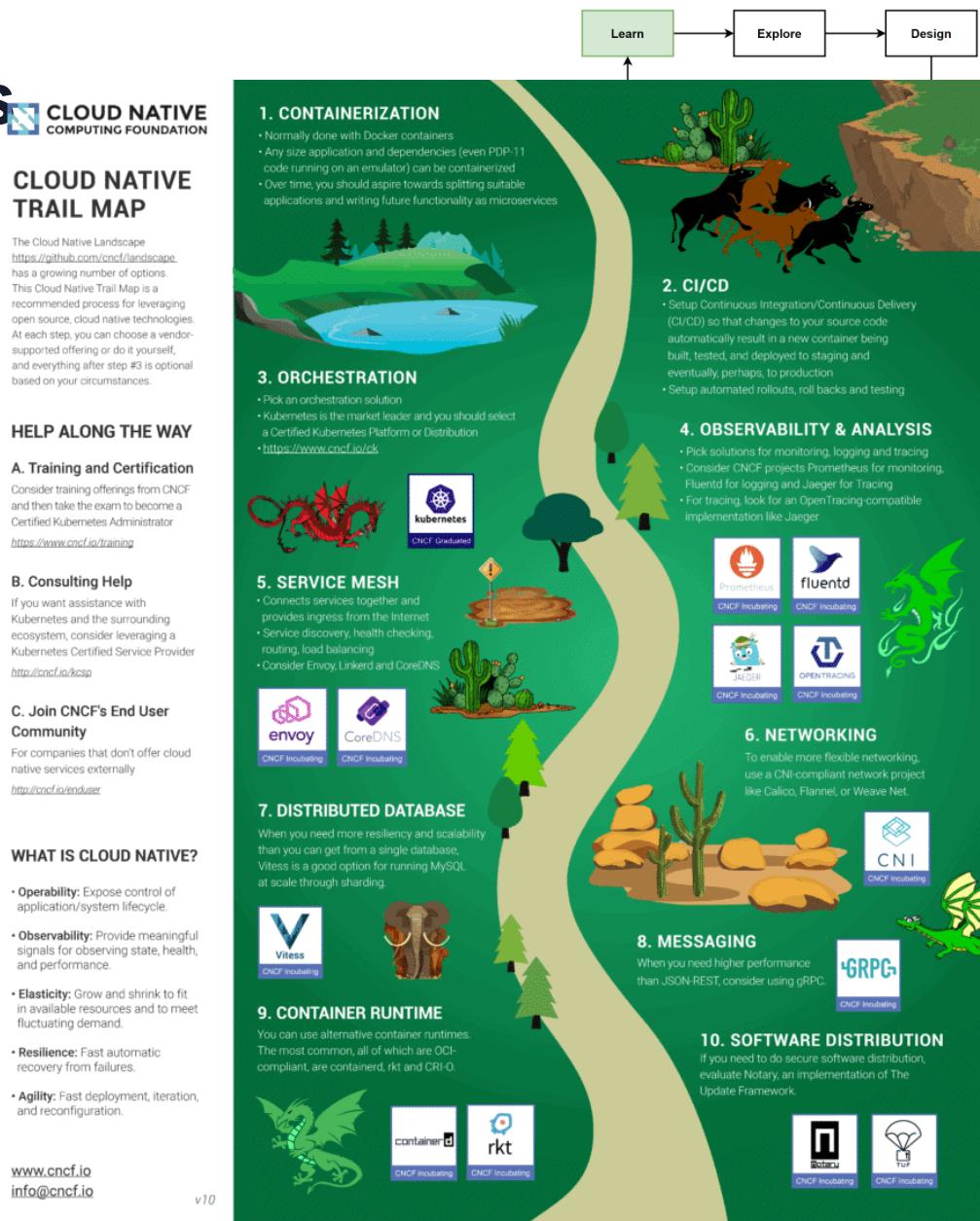


CI/CD

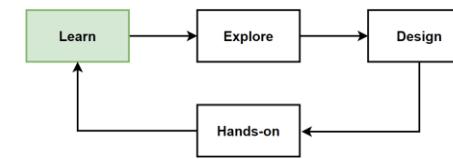


Cloud-native Trial Map – Backing Services Databases

- Use of **distributed** or **decentralized** systems for managing data storage and retrieval in a cloud-native environment.
- **Traditionally**, databases were **designed as centralized solutions**, where all data was stored and managed in a single location.
- However, as the **scale** and **complexity** of apps **increased**, the need for more scalable and resilient data management systems arose.
- **Distributed databases** are more **easy to deploy** in **Kubernetes** stateful sets, increasing to store stateful sets into K8s clusters.
- **Data is partitioned or replicated** across **multiple nodes** or servers, forming a distributed network.
- **Each node** can **store** and **process** a portion of the data, allowing for increased performance, fault tolerance, and scalability.
- **Distributed databases** provide Scalability, Fault tolerance, Performance, Elasticity, Consistency and Data locality.



<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>

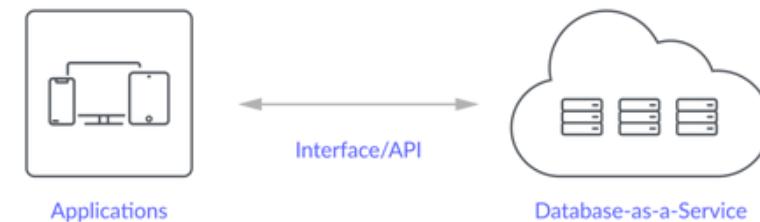


Database as a Service - DBaaS for Cloud-Native Apps

Database as a Service (DBaaS) provides access to a fully managed database instance without the need for in-house database management, hardware provisioning, or software installation.

Simplified database management

- Handle database setup, configuration, backups, updates, and maintenance, eliminating the need for in-house database administration.
- Developers to concentrate on implementing microservice functionality.

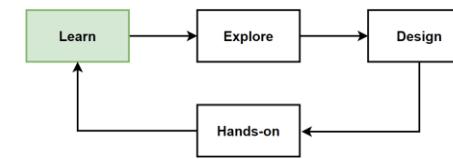


Scalability

- Horizontal and vertical scaling to accommodate increased load or growing data storage requirements.
- Particularly important in cloud-native microservices, which are designed to scale independently.

High availability and fault tolerance

- Built-in redundancy, automatic failover, and data replication across multiple availability zones or regions, ensuring high availability and fault tolerance for your microservices' data.



Database as a Service - DBaaS for Cloud-Native Apps2

Multi-cloud and hybrid deployments

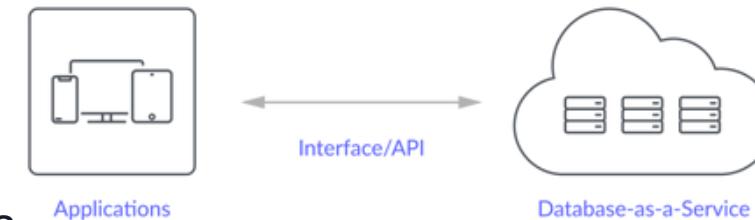
- Deployed across multiple cloud providers or in a hybrid cloud environment, providing flexibility and reducing vendor lock-in for cloud-native microservices.

Managed security

- Managed security features such as encryption, access control, network isolation, and regular security updates, ensuring that your microservices' data is secure.

Cost-effectiveness

- With a pay-as-you-go pricing model, DBaaS allows you to pay only for the resources you use, helping to optimize costs for your cloud-native microservices.
- Popular DBaaS providers and services include Amazon RDS, Google Cloud SQL, Azure SQL Database, and MongoDB Atlas.



How DBaaS use as a Backing Services into Cloud-Native Microservices ?

Database as a Service (DBaaS) is a type of backing service that provides managed database instances for cloud-native microservices.

When a microservice needs to store, retrieve or manipulate data, it can rely on a DBaaS solution as a backing service to handle the database operations.

Decoupling and abstraction

- Microservice can be decoupled from the underlying database infrastructure and focus on its core business logic.

Consistent and standardized access

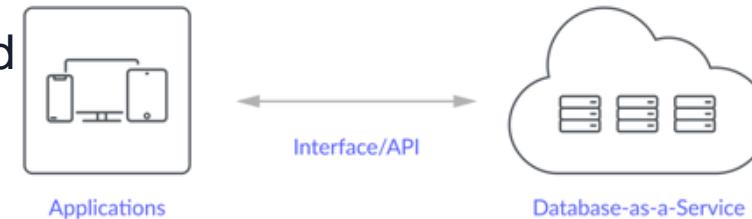
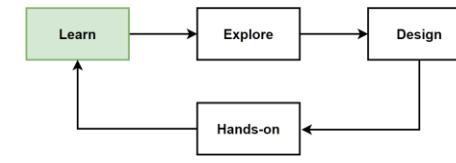
- Microservices can access the database through standardized APIs and protocols, ensuring consistent communication and interaction with the data.

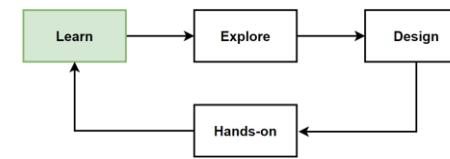
Easier integration and deployment

- Seamless integration with cloud-native orchestration tools, such as Kubernetes and Helm, easier to deploy and manage microservices depend on databases.

Flexibility and interoperability

- Developers can easily switch between different database types or cloud providers, allowing for greater flexibility and reduced vendor lock-in.

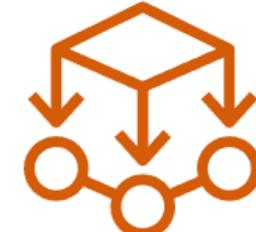




Backing Services - Data Management

- Cloud-Native Backing Services – Data Management :
 - Databases (e.g. MySQL, PostgreSQL, Cloud Spanner)
- **Relational, NoSQL and NewSQL Databases**
- Explore - Backing Services for Databases.
 - K8s and Serverless Databases
- Hands-on - Backing Services for Databases.
 - K8s and Serverless Databases

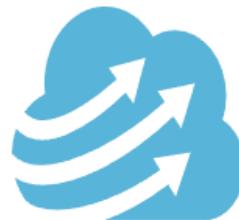
Microservices Containers

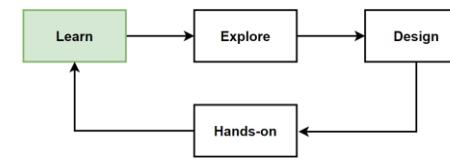


Devops



CI/CD

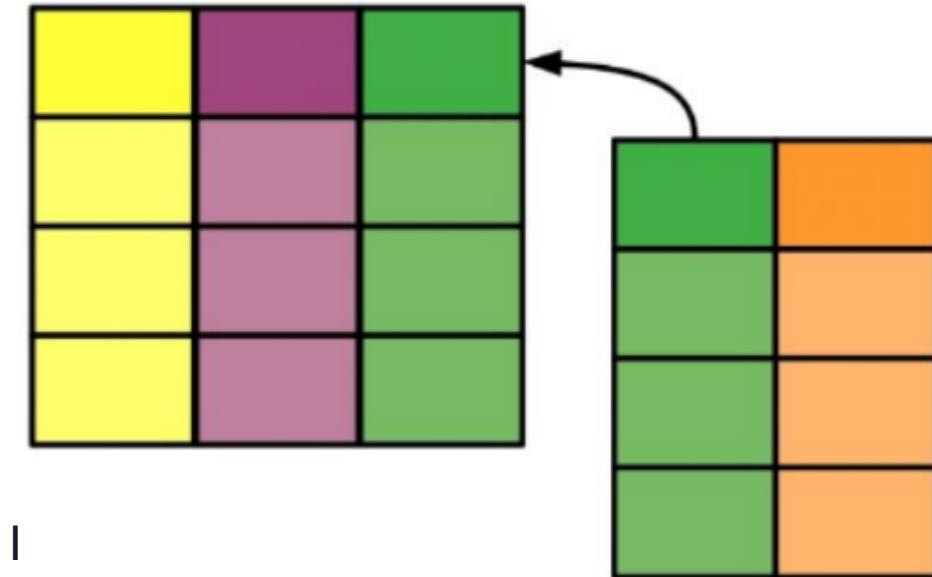




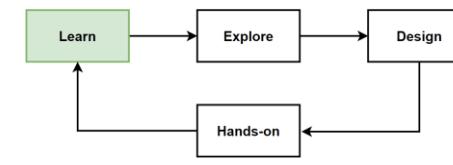
Relational Databases - RDBMS

- **Relational databases** provides storing data into related data tables.
- Relational database tables have a **fixed schema**, use SQL to manage data and support transactions with **ACID principles**.
- A table uses **columns** and **rows** for storing the actual data. Each table will have a column that must have **unique values**—known as the **primary key**.
- When one **table's primary key** is used in **another table**, this column in the second table is known as the **foreign key**.
- On microservices, we should **not consider** using **single big relational database**.
- The main advantages of Relational database is **ACID compliance**. If one change fails, the **whole transaction will fail**.
- Polyglot persistence in microservices, still **relational databases** is **good option** some certain type of problems.
- **Example of relational databases** are Oracle, MS SQL Server, MySQL, PostgreSQL.

Relational

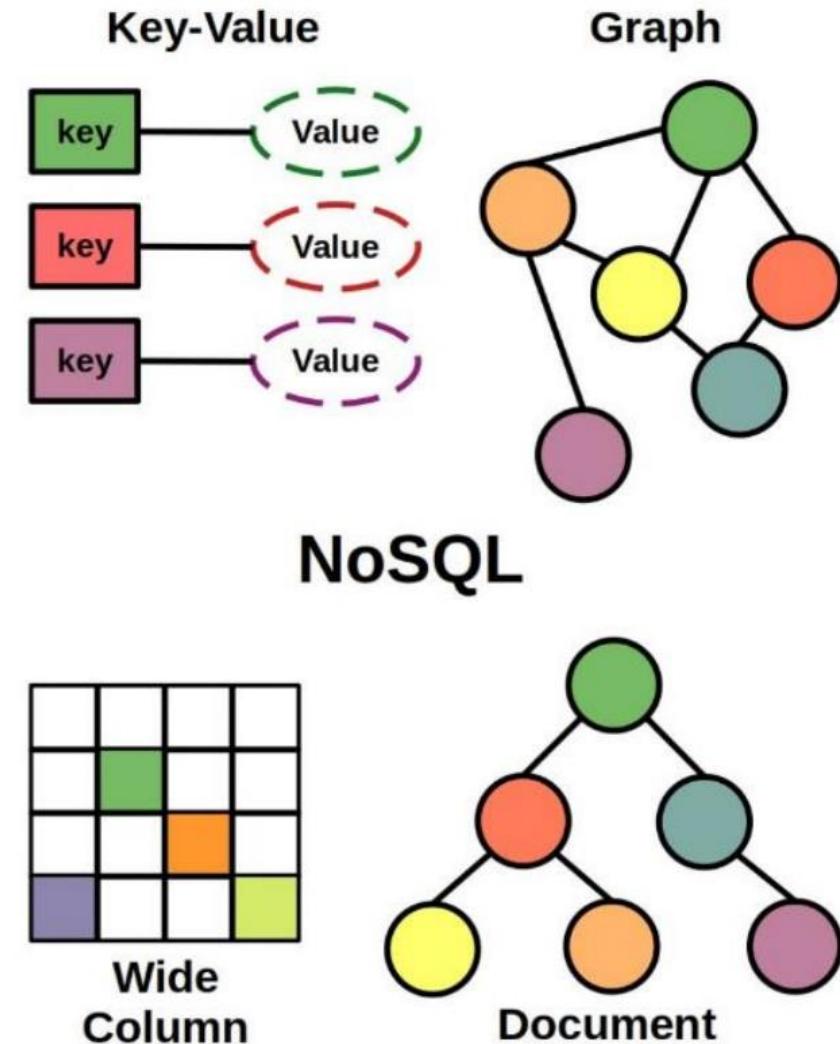


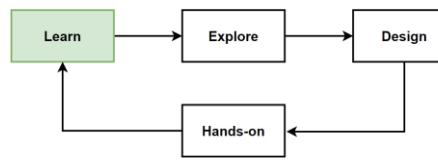
SQL



No-SQL Databases (Non-Relational Databases)

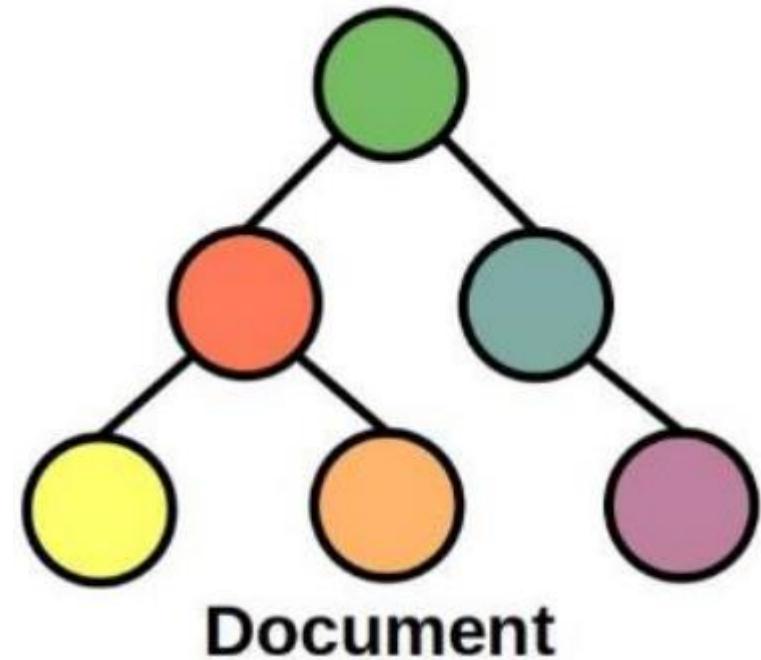
- **No-SQL databases** has different types of stored data and data models: **Document**, **Key-value**, **Graph-based**, **Column-based** databases.
- **Ease-of-use, scalability, resilience, and availability** characteristics.
- NoSQL databases stores **unstructured data**, and this gives huge **performance advantage**.
- **NoSQL stored unstructured data** in key-value pairs or JSON documents.
- No-SQL databases **don't provide ACID** guarantees.
- **Drawback is transaction management.**

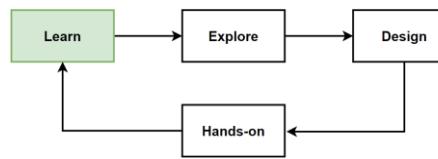




No-SQL Document Databases

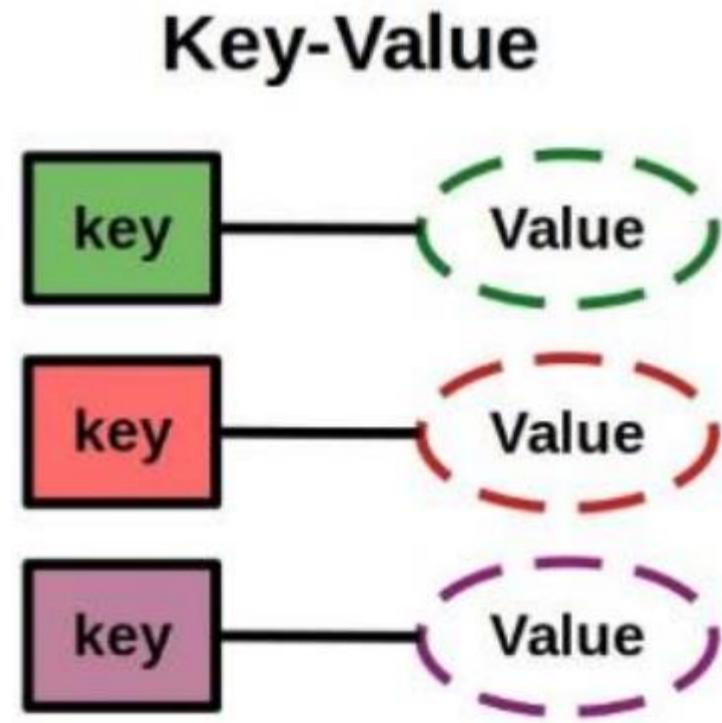
- **Document databases** stores and query data in JSON-based documents.
- **Data and metadata are stored hierarchically.**
- **Objects are mapping to the application code.**
- **Don't have to run JOINs** or decompose data across tables.
- **Scalability**, document databases can **distributed very well**.
- Best choice for **content management** and **storing catalogs**.
- I.e. **products** data can store in **document database** for **e-commerce applications**.
- **Example Document Databases:** MongoDB and Cloudant.

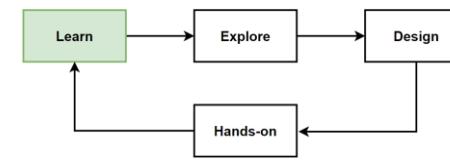




No-SQL Key-Value Databases

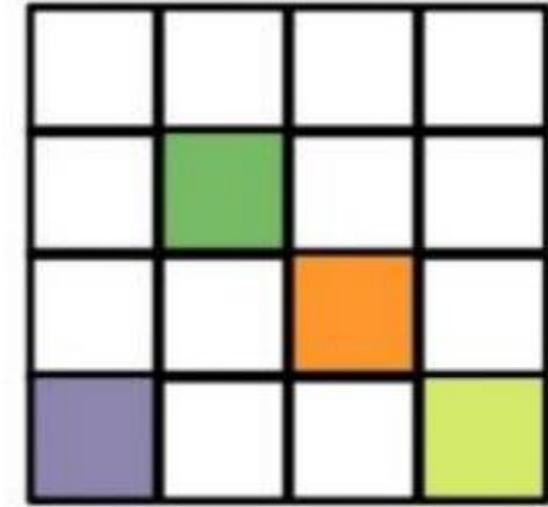
- Data is stored as a collection of **key-value pairs** in Key-value **NoSQL database**.
- Data is represented as a **group of key-value** in the database.
- **Best choice** for **session-oriented** applications.
- I.e. storing **customer basket data** into **key-value database**.
- **Example Key-Value Databases:** Redis, Amazon DynamoDB, Azure CosmosDB, Oracle NoSQL Database.



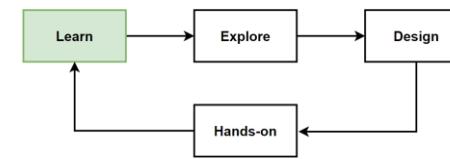


No-SQL Column-Based Databases

- **Column-based databases** also known Wide-Column Databases.
- **Data is stored in columns**, by this way, it can access necessary data **more faster** than if we compare to storing data in rows.
- If you **select mostly same columns** in your databases, its good to use this databases.
- It **doesn't scanning the unnecessary information** in a whole row.
- **Column-based databases can scale by columns independently.**
- **Columns could be different database servers.**
- I.e. building a **Data warehouse**, **Big Data processing**.
- **Apache Cassandra, Apache HBase or Amazon DynamoDB, Azure CosmosDB.**



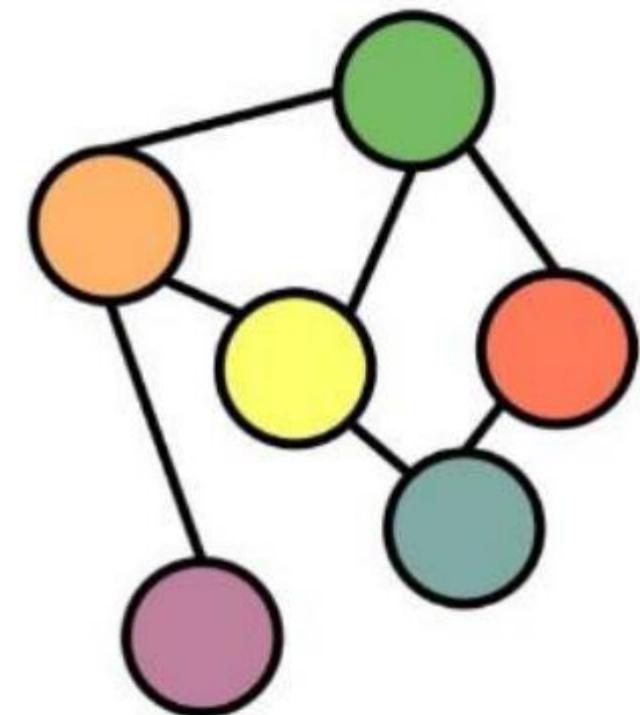
**Wide
Column**

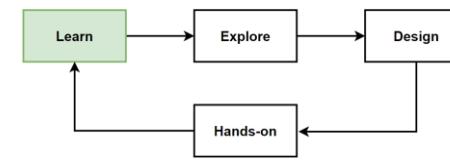


No-SQL Graph-Based Databases

- **Graph-based databases** stores data in a **graph structure** into **node**, **edge**, and **data properties**.
- **Data entities** are **connected in nodes**.
- The main benefit of a **graph-based databases** is to **store and navigate graph relationships**.
- I.e. **fraud detection**, **social networks**, and **recommendation engines**.
- **Example of Graph-based databases** are OrientDB, Neo4j, and Amazon Neptune.

Graph





When to Use Relational Databases ?

- **ACID compliance, Data Consistency**

Relational database is ACID compliant. ACID - Atomicity, Consistency, Isolation, and Durability (ACID) guarantees the reliability of database transactions.

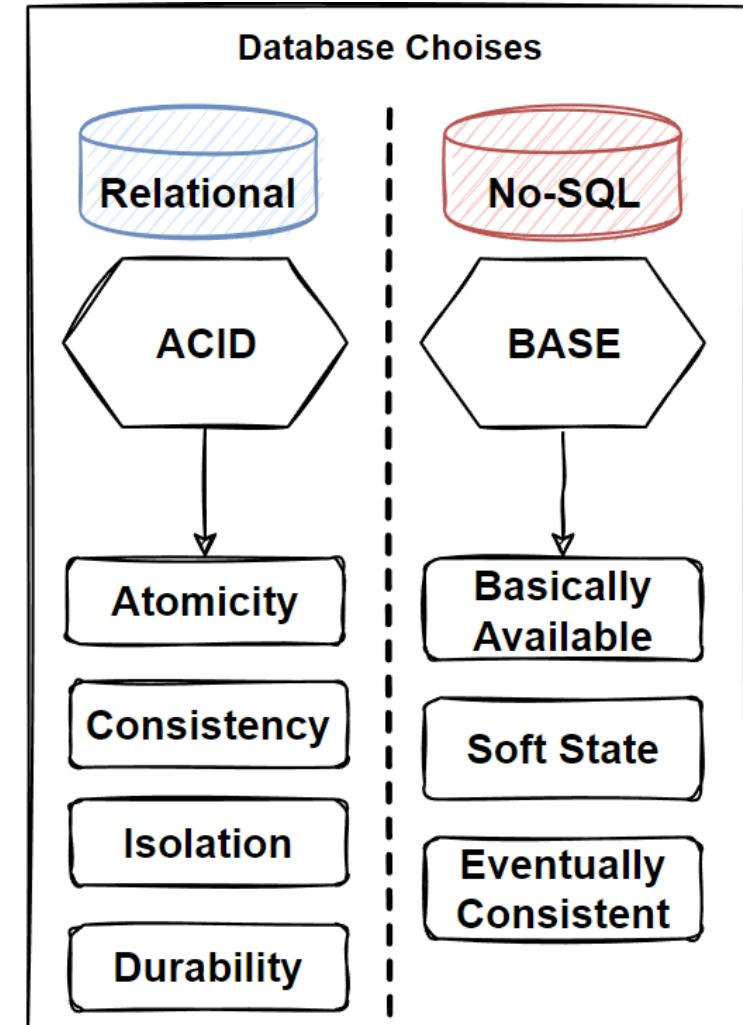
- If one change fails on database, database should remain in the previous state that was before the transaction.

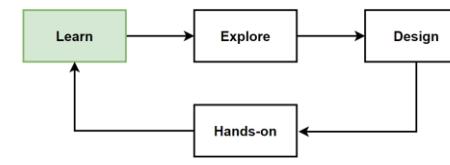
- Data Consistency is fully provided, if you need strong consistency, good to choose Relational Databases, supports complex transactions.

- **Predictable Data, Low Workload Volume**

If application data is predictable, as per structure, size, and frequency of access, within thousands of transactions per second, relational databases are still the best choice.

- Normalization also reduces the size of the data on disk by limiting duplicate data and anomalies.





When to Use Relational Databases ? - 2

- **Read Requirements, Complex Join Queries**

Relational Database has a fixed schema. When relationships between tables are important and data is highly structured and requires referential integrity.

- Can work with complex queries, table joins and reports on normalized data models.

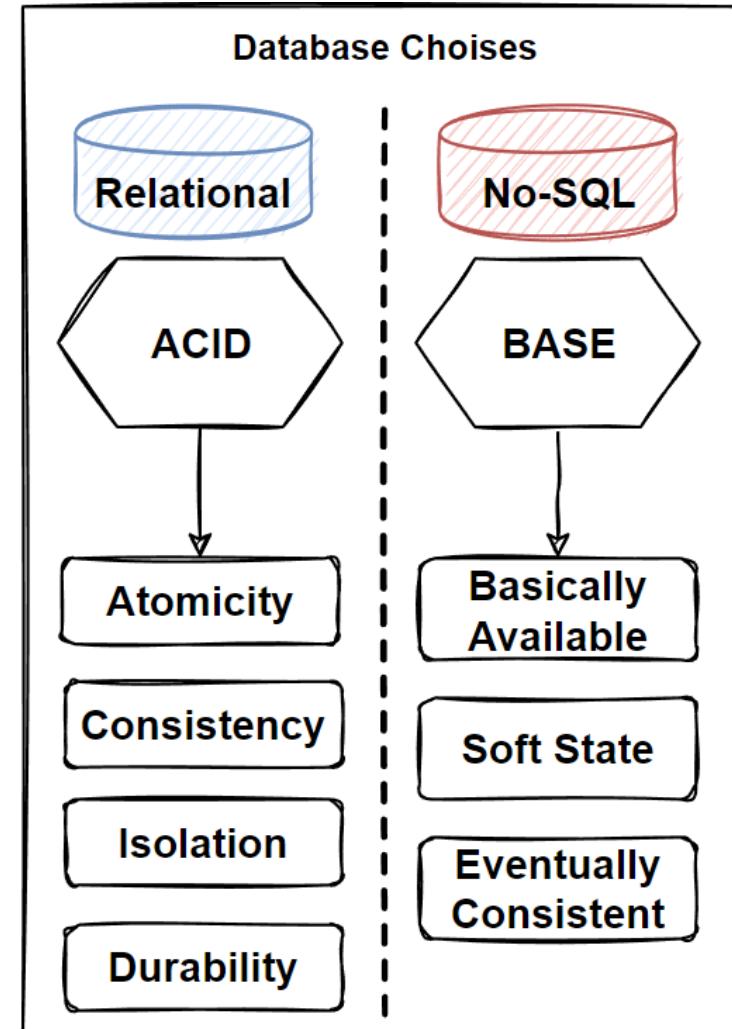
- Relational Database supports a powerful SQL query language.

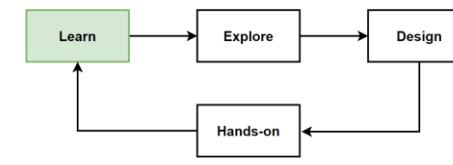
- **Deployments, Centralized Structure**

Relational Databases will be deployed to large and one or few locations. Relational database has centralized structure.

- Relational databases have a single point of failure with failover.

- Relation database is deployed in vertical fashion.





When to Use No-SQL Databases ?

- **Flexible Schema, Dynamic Data**

NoSQL Database has no fixed schema. Allows to add or remove attributes into their model with dynamically. When your data is dynamic and frequently changes.

- Use case of implement an IoT platform that stores data from different kinds of sensors with frequently changed the attributes of your data.

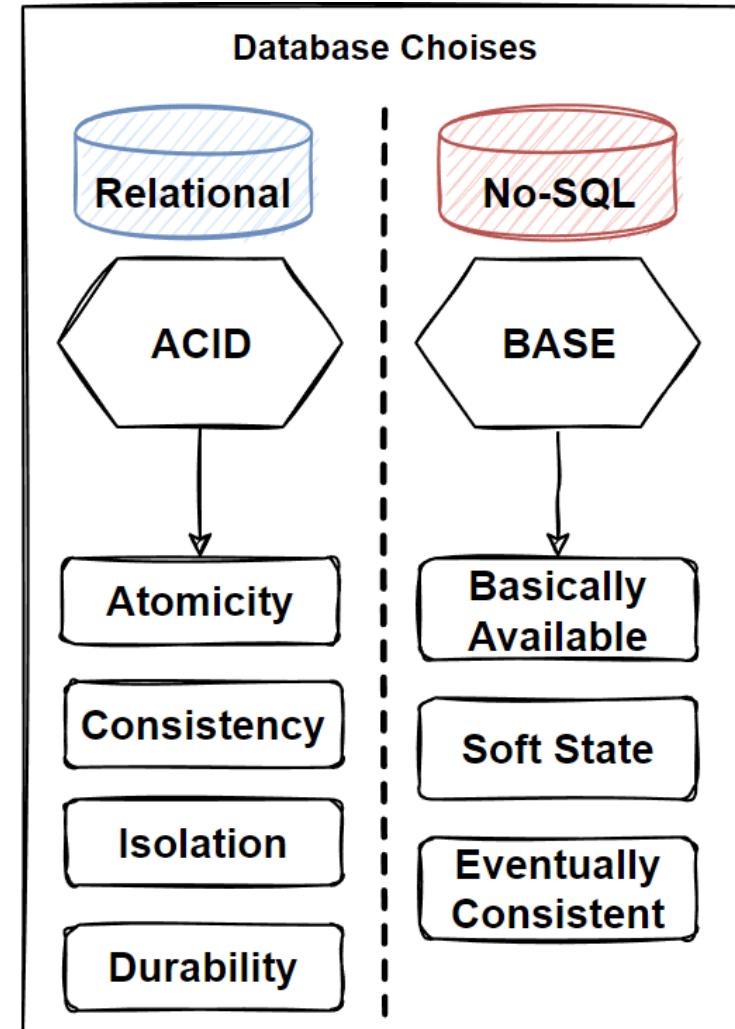
- **Un-predictable Data, High Workload Volume**

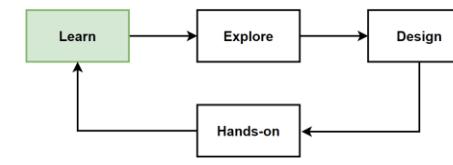
When you have high volume workloads and needs to horizontal scale with low latency. NoSQL databases have been designed for the cloud that naturally good for horizontal scaling.

- NoSQL Databases prioritize partition tolerance that designed for handling large amount of data or data coming in high velocity.

- **Frequently Change Data and Read Requirements**

When data is dynamic and frequently changes and Relationships are de-normalized data models and Data retrieve operations are simple and performs without table joins.





When to Use No-SQL Databases ? - 2

- Data Consistency, BASE Model - Basically Available, Soft State, Eventually Consistent**

NoSQL Database is only eventually consistent and don't support transactions, focus on high volume data and horizontal scaling.

- Write Performance Requirements**

NoSQL Database compromise consistency to achieve fast write performance, offers fast write operations with Eventual consistency.

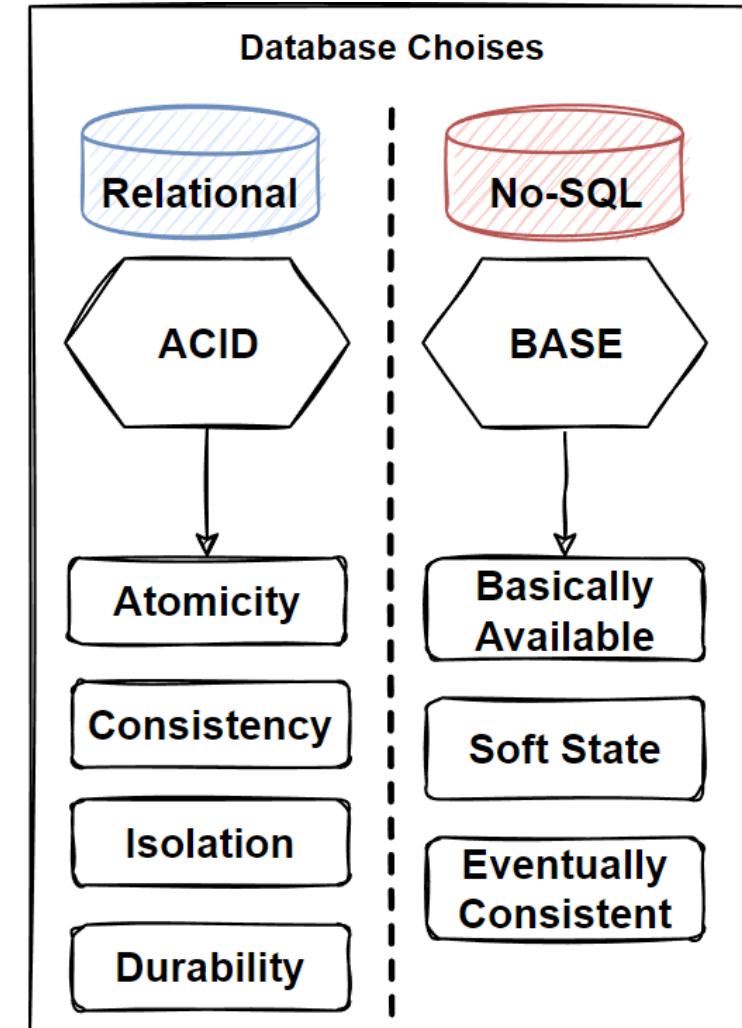
- Not Good for Complex Join Queries**

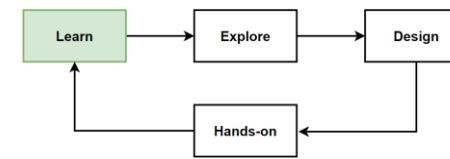
NoSQL Databases perform best when data is stored in the same format not require relation and join operations.

- Deployments, De-centralized Structure**

NoSQL scales horizontally so data is replicated across different geographical zones and provides better control over consistency, availability, and performance.

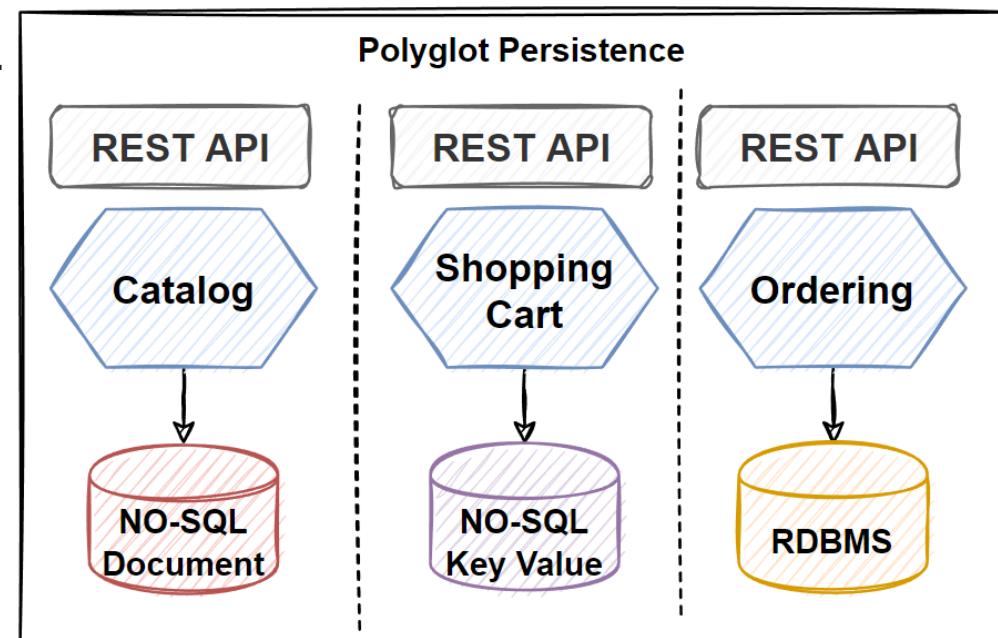
- NoSQL databases have no single point of failure, has decentralized structure, gives both read and write scalability, deployed horizontally.

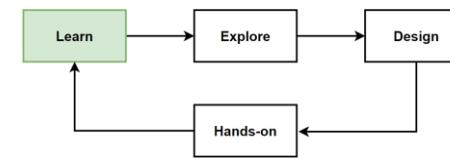




Best Practices When Choosing Data Store

- Use Right Tool for Right Job, Use the best data store for your data.
- **Don't use Relational Database Everywhere**
If using Relational Database for all microservices, would probably going wrong way, consider other data stores to data requirements.
- **Don't use Single Data Store Technology, Differentiate**
Choose Alternatives to relational databases; Key/value stores, Document, Search engine, Time series, Column family, Graph databases.
- **Focus The Data Type That Need to Store**
Consider the type of data that you have.
 - Store JSON documents in No-SQL Document database.
 - Put transactional data into a Relational SQL database.
 - Use a time series data base for telemetry databases.
 - Choose Blob Data Storage for blob datas.
 - Put application logs into Elastic Search Databases.





Best Practices When Choosing Data Store - 2

- Trade-offs between Availability and Consistency**

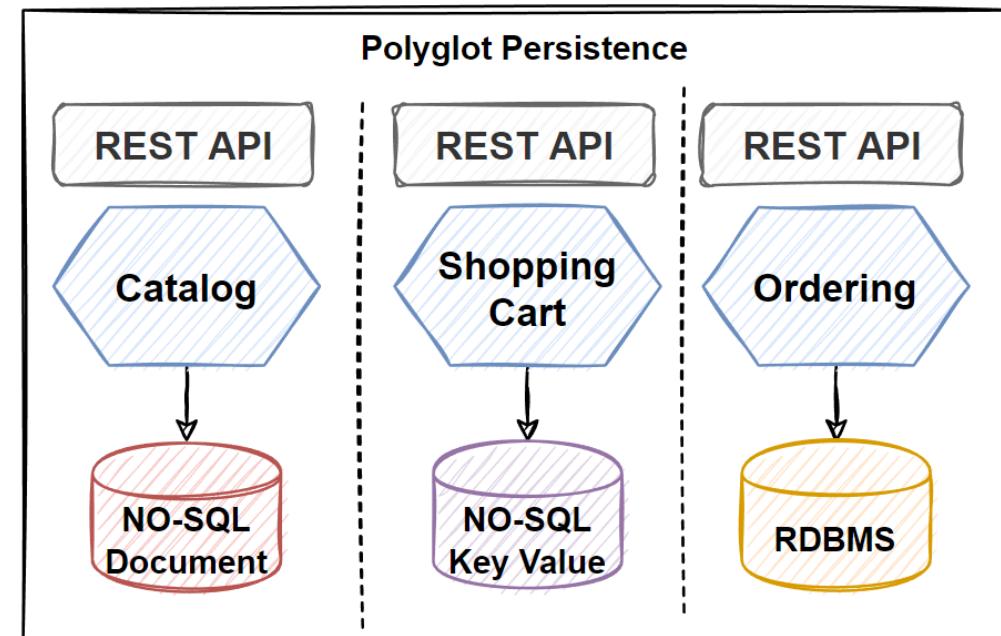
Understand the Trade-offs between Availability and Consistency.
Should prefer High Availability over strong consistency as soon as possible to scale horizontally. (the CAP theorem)

- Transactional Boundaries Between Microservices**

Consider business Transactional Boundaries Between Microservices, that data need to consistent across those microservices. Prepare for compensating transactions in case of fail.

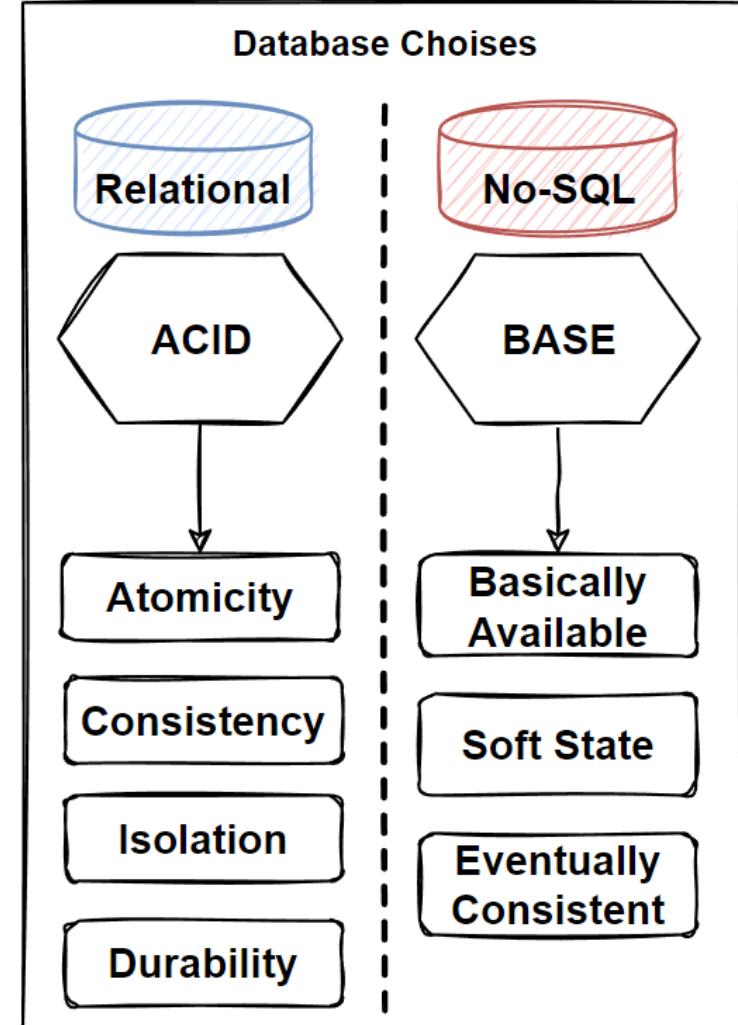
- Competence of Development Teams**

Consider your team competences about database technologies.
Skill set of your development team should cover optimize queries and tune for performance improvements.



How to Choose a Database for Microservices ? (Question Set)

- **Data Consistency Level**
Do we need Strict-Strong consistency or Eventual consistency ?
- Do we need ACID compliance ? Should follow Eventual consistency in microservices to gain high scalability and availability.
- **Fixed or Flexible Schema Choise, Predictable or Dynamic Data**
Are we work with fixed or flexible schema that need to change frequently, dynamically changed data ?
- Are we have Predictable Data or Dynamic Data ?
- **High or Low Data Volume, Predictable or Un-predictable Data**
Are we work with High Volume Data or Low Volume Data ?
- Can we have predictable data that we store our microservices database ?
- NoSQL Databases prioritize partition tolerance that handling large amount of data or data coming in high velocity.



How to Choose a Database for Microservices ? (Question Set) - 2

- **Read Requirements, Relational or non-Relational Data, Complex Join Queries**

Our data is highly structured and requires referential integrity or not required for relationships that is dynamic and frequently changes ?

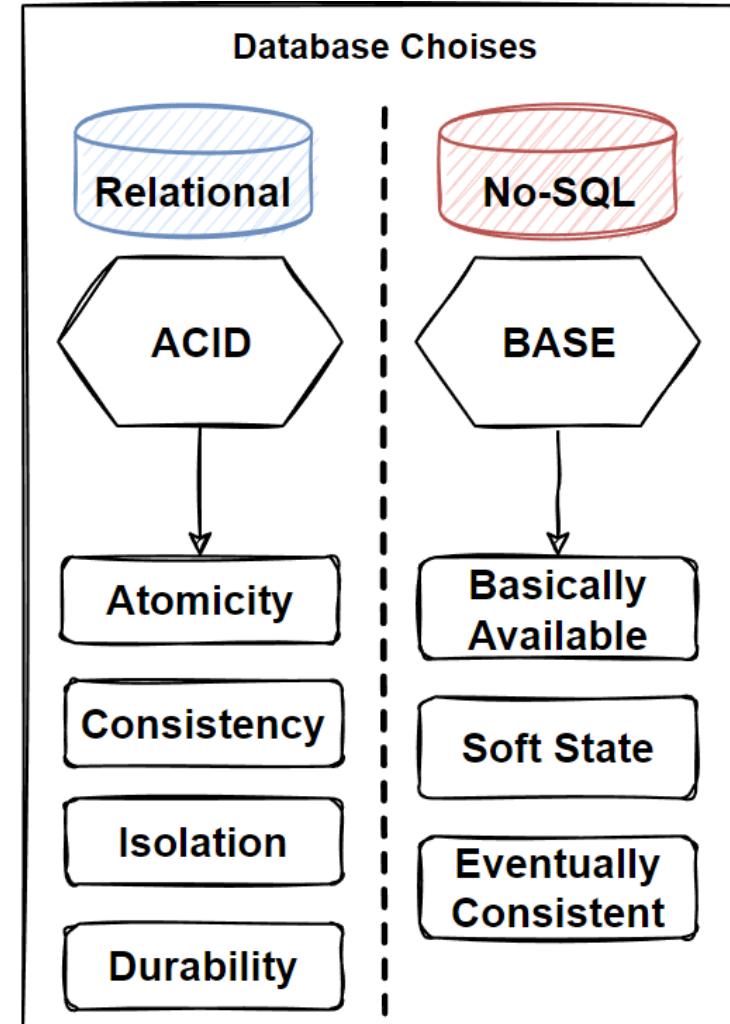
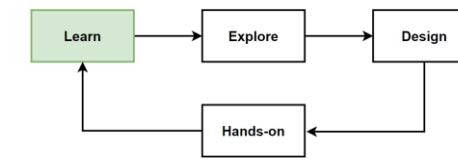
- Should it work with complex queries, table joins and run SQL queries on normalized data models or Retrieve data operations are simple and performs without table joins ?

- **Deployments, Centralized or De-centralized Structure**

Do we deployed to large and one or few locations with centralized structure ? or Do we need to deploy and replicate data across different geographical zones ?

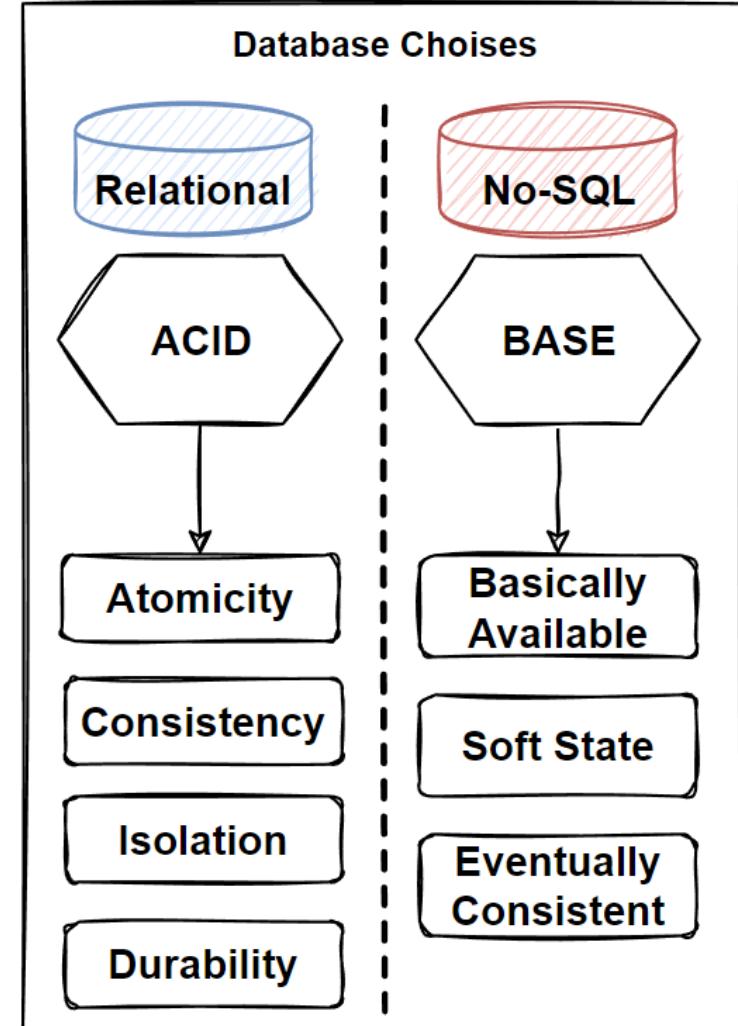
- **High Performance Requirements**

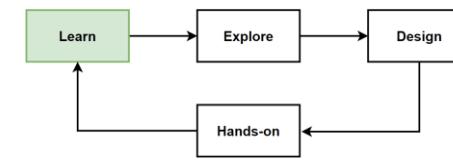
Do we need to achieve fast read-write performance ?



How to Choose a Database for Microservices ? (Question Set) - 3

- **High Scalability Requirements**
Do we need High Scalability Requirements both vertical and horizontally scaling ?
- To accomodate millions of request should sacrifice strong consistency.
- **High Availability and Low Latency Requirements**
Do we need High Availability and Low Latency Requirements that need to separate data across different geographical zones ?
- **Can we provide ALL OF THESE FEATURES at the same time ?**
Is it possible to provide High Scalability, High Availability and Low Latency with High Performance and able to run Complex Join Queries providing with ACID principles strong data consistency ?
- **Yes.**
- **Which ?**
- **NewSQL Databases**

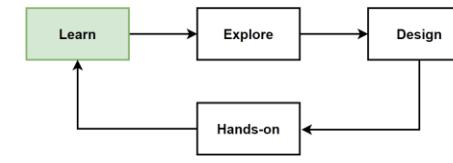




NewSQL Databases

- **NewSQL databases aim to combine the best features of traditional SQL databases and NoSQL databases.**
- Provide the **ACID (Atomicity, Consistency, Isolation, Durability) guarantees** and **support for SQL queries** like traditional RDBMS, while also offering the **scalability, performance, and flexibility** of **NoSQL databases**.
- NewSQL databases emerged as a **response to the limitations of traditional RDBMS** when dealing with the growing demands of big data, distributed systems, and high-velocity transactions.
- The goal of NewSQL databases is to **bridge the gap between the two database paradigms** by offering a solution that addresses the shortcomings of both.
- **Represent a hybrid approach** that attempts to combine the benefits of NoSQL with the **transactional consistency** required found in Relational databases.
- Due to NewSQL databases very **convenient to deploy K8s with Horizontally scalable Distributed Cloud-native databases**, we also call these databases as a K8s databases.
- Examples of NewSQL databases include **Google Spanner, CockroachDB, TiDB, and VoltDB**.





Key Characteristics of NewSQL databases

- **ACID Compliance**

NewSQL databases ensure data integrity and consistency through ACID transactions, similar to traditional SQL databases.

- **Scalability**

Horizontal scalability, allowing them to handle large amounts of data and high levels of concurrency, similar to NoSQL databases.

- **High Performance and Reduced Latency**

In-memory processing, multi-version concurrency control (MVCC), and distributed architectures, NewSQL databases deliver high performance for both read-heavy and write-heavy workloads.

- **SQL Support**

Use SQL as the query language, making it easier for developers and data professionals familiar with SQL to work with these databases.

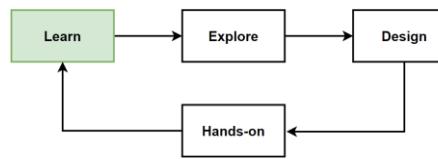
- **Schema Flexibility**

Schema flexibility, making it easier to adapt to changing application requirements.

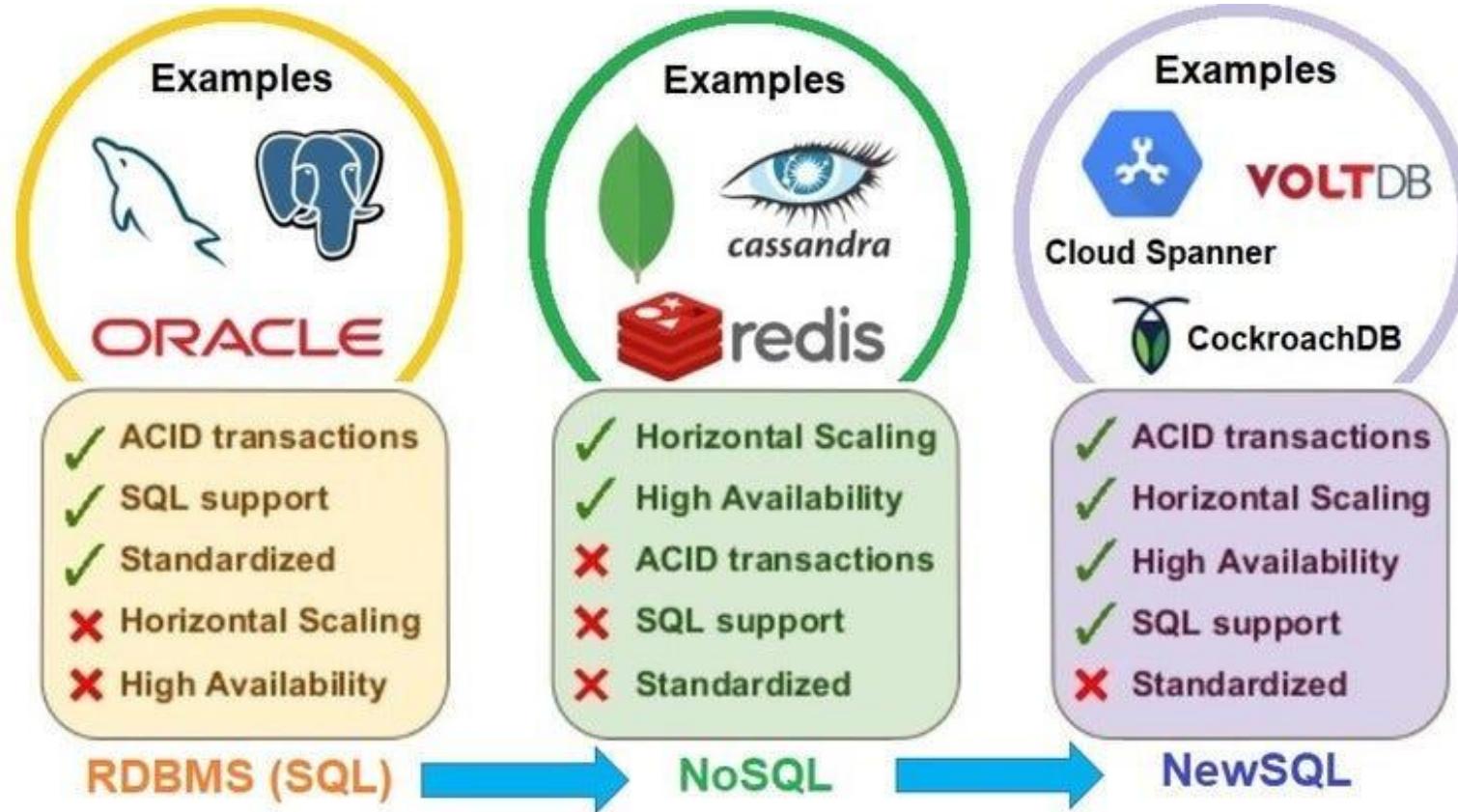
- **Fault Tolerance**

Built-in support for replication and recovery, making them resilient to failures.



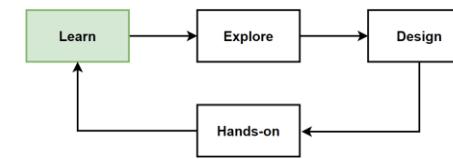


Comparison of Relational, NoSQL and NewSQL DBs



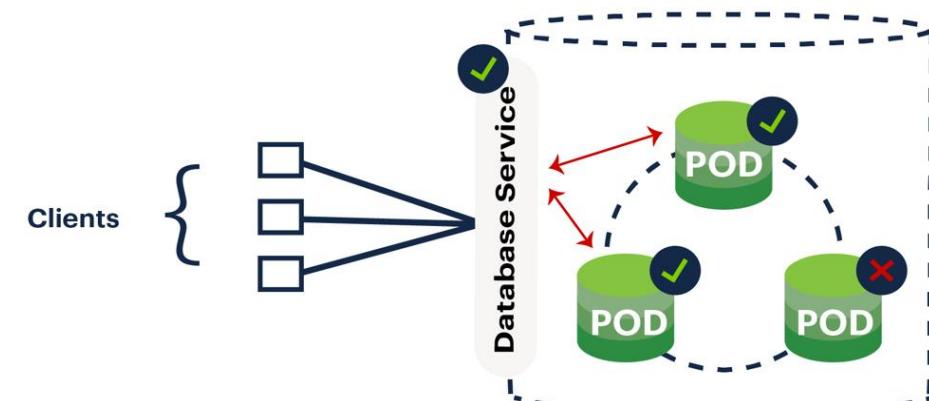
Excellent fit for
Kubernetes
Deployments

<https://medium.com/rabiprasadpadhy/google-spanner-a-newsql-journey-or-beginning-of-the-end-of-the-nosql-era-3785be8e5c38>



The Rise of the Kubernetes Native Database

- Cloud-native application architectures are **transforming with Kubernetes** the way developers deliver services to their customers.
- The **stateful nature of databases** presents a **challenge** to running them on **Kubernetes**.
- Traditional relational databases and NoSQL databases weren't designed to take full advantage of the dynamic, **distributed environments**.
- **Kubernetes-Native Database** are designed from the ground up to operate in the cloud-native ecosystem and take full advantage of **Kubernetes' capabilities**.
- **Distributed SQL databases**, such as CockroachDB, represent a **new generation of Kubernetes-native databases**.
- Unlike traditional relational databases and NoSQL databases, which were not built with Kubernetes in mind, **Distributed SQL databases** are **designed to work out-of-the-box** with **Kubernetes**.



<https://www.cockroachlabs.com/blog/dash-kubernetes-native-database/>

Key Characteristics of Kubernetes-native Databases

- **Distributed Architecture**

Designed for the distributed nature of cloud environments that allows them to scale seamlessly across multiple Kubernetes nodes.

- **Stateful Applications**

Handle the persistent state of applications, a key challenge in Kubernetes environments with maintaining high availability and data consistency.

- **Highly Compatible with Kubernetes**

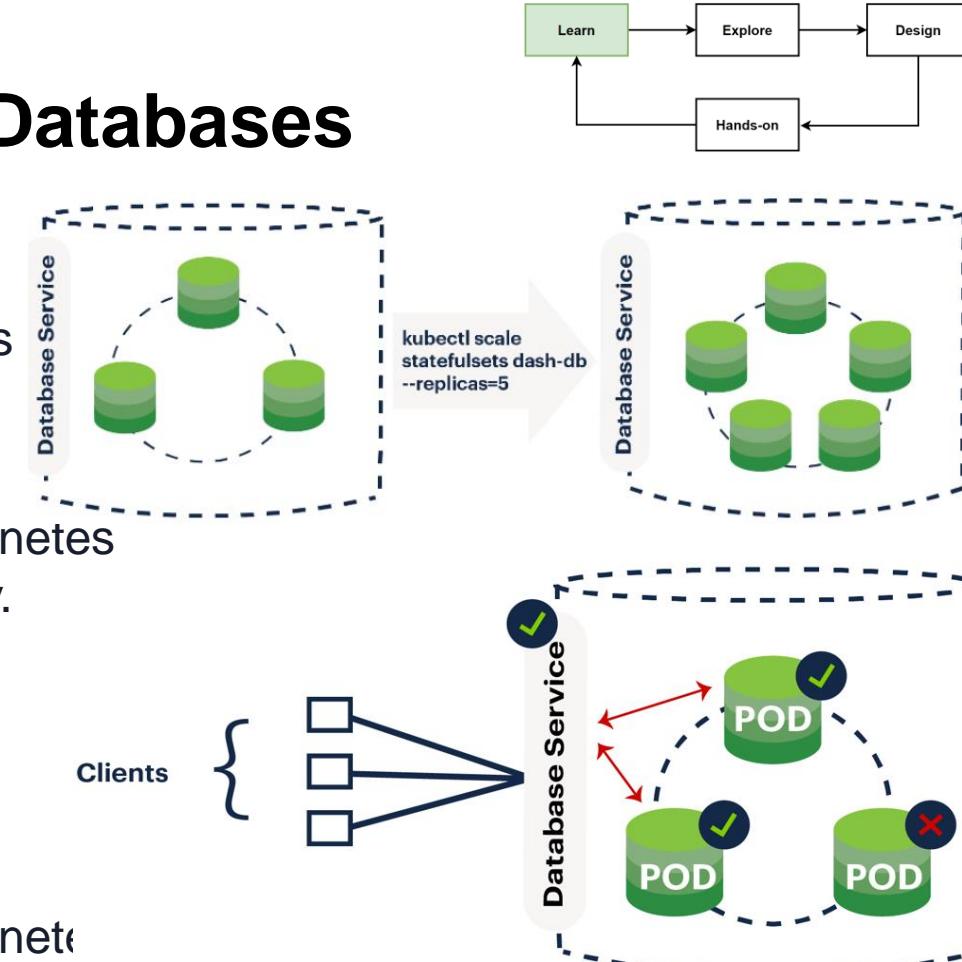
Align with Kubernetes' architectural primitives, reducing the need for complex operators to manage the databases.

- **Full Utilization of Kubernetes**

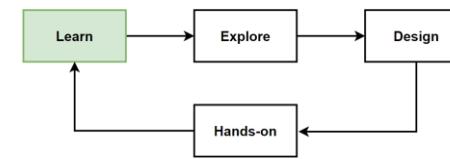
Allow organizations to fully harness the power and benefits of Kubernetes across their entire application stack.

- Adhere to the **DASH framework**, taking full advantage of the distributed, highly available, and resilient nature of cloud-native applications.

- What is DASH framework ?

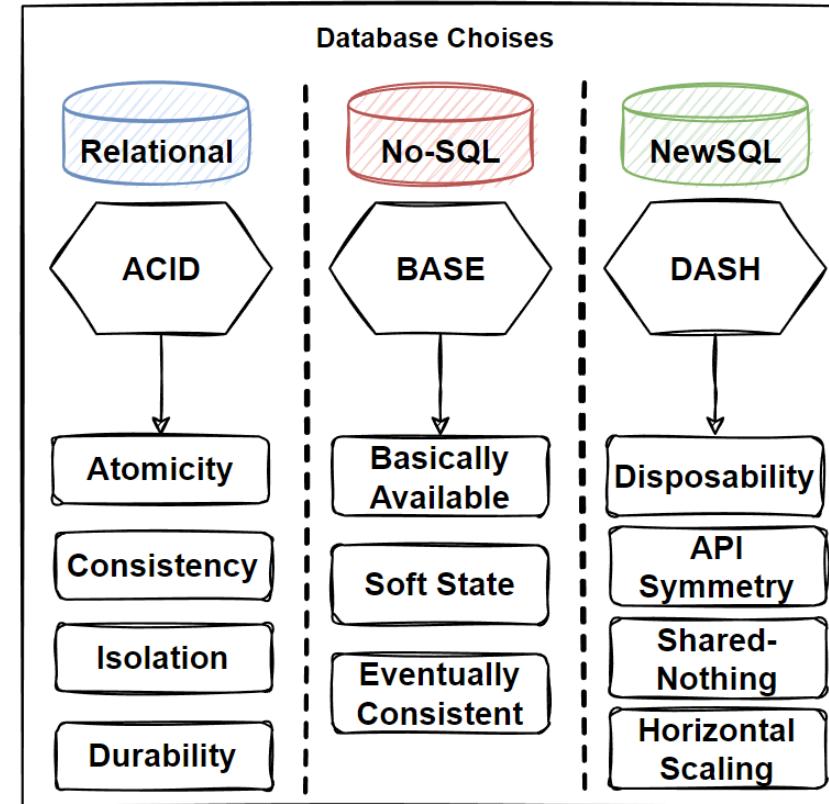


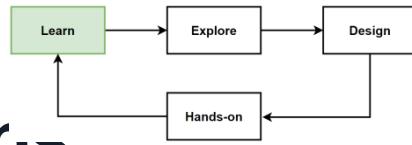
<https://www.cockroachlabs.com/blog/dash-kubernetes-native-database/>



DASH Framework for Kubernetes-Native Databases

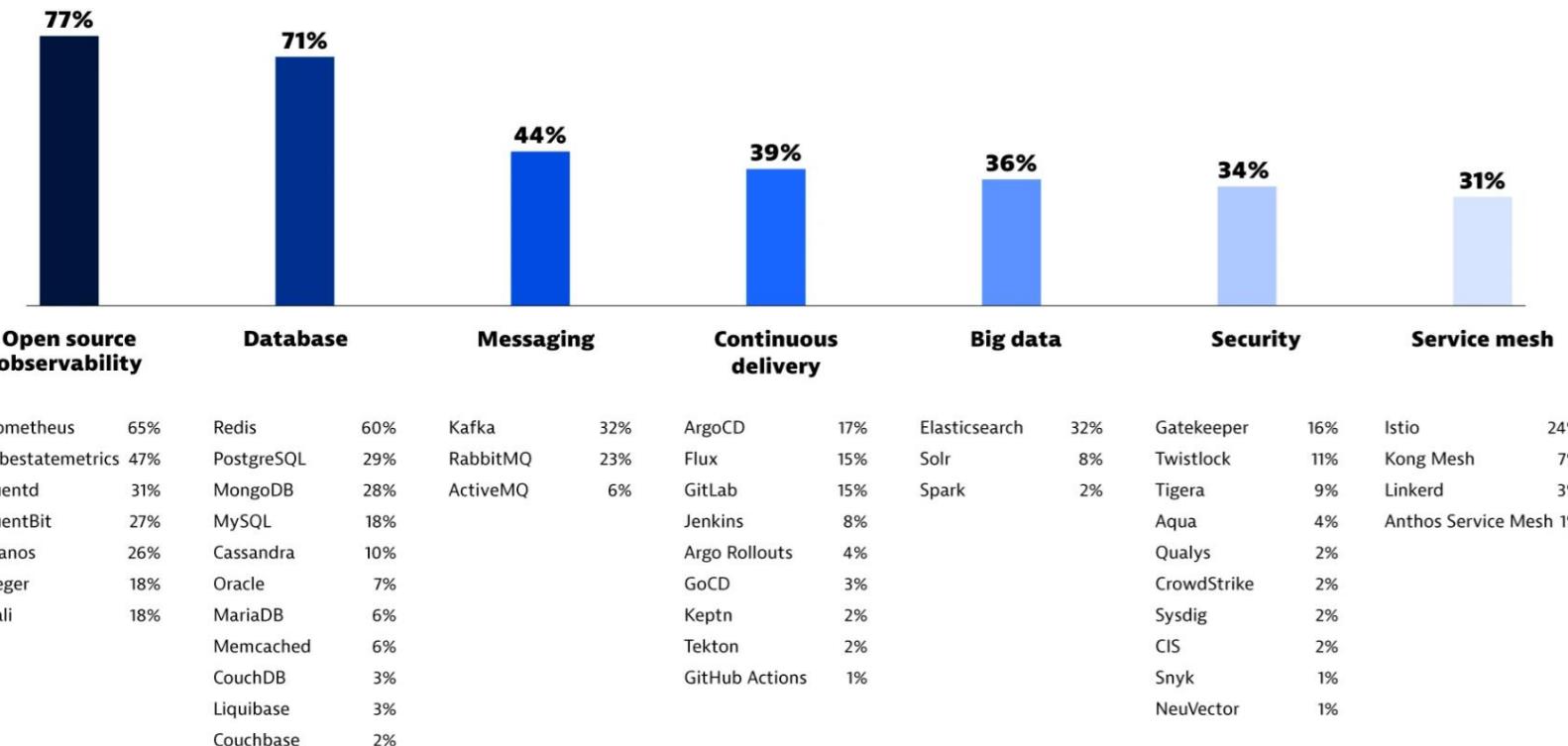
- **DASH Framework for Kubernetes-Native Databases:** Disposability, API Symmetry, Shared Nothing, and Horizontal Scaling.
- **Disposability**
Ability of a database to handle failures, or processes stopping, starting, or crashing with little-to-no notice. The Kubernetes Scheduler plays an important role in managing disruption incidents.
- **API Symmetry**
Every server in the network provides the same answer to the same question. Kubernetes-native databases act as a single logical database with the consistency guarantees of a single-machine system.
- **Shared-Nothing**
True Kubernetes-native database can operate without any centralized coordinator or single point of failure. A database should be able to operate without any centralized coordinator or single point of failure.
- **Horizontal Scaling**
Kubernetes Controllers and Schedulers combine to make horizontal scaling an easy, with K8s declarative process. Simplify utilization of resources and deliver effortless scale.

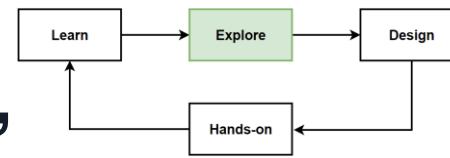




Most Usage Databases in Kubernetes for Cloud-Native Apps

- [Dynatrace Kubernetes Report: Strongest Kubernetes growth areas](#)
- Open-source projects in the Kubernetes across various categories, including observability, databases, messaging, CI/CD, big data, security, and service meshes.
- Open-source solutions: Prometheus, Redis, RabbitMQ, Kafka, ArgoCD, Flux, GitLab, Jenkins, Elasticsearch, Gatekeeper, and Istio, are widely used as backing services or tools to enhance the capabilities of cloud-native applications.

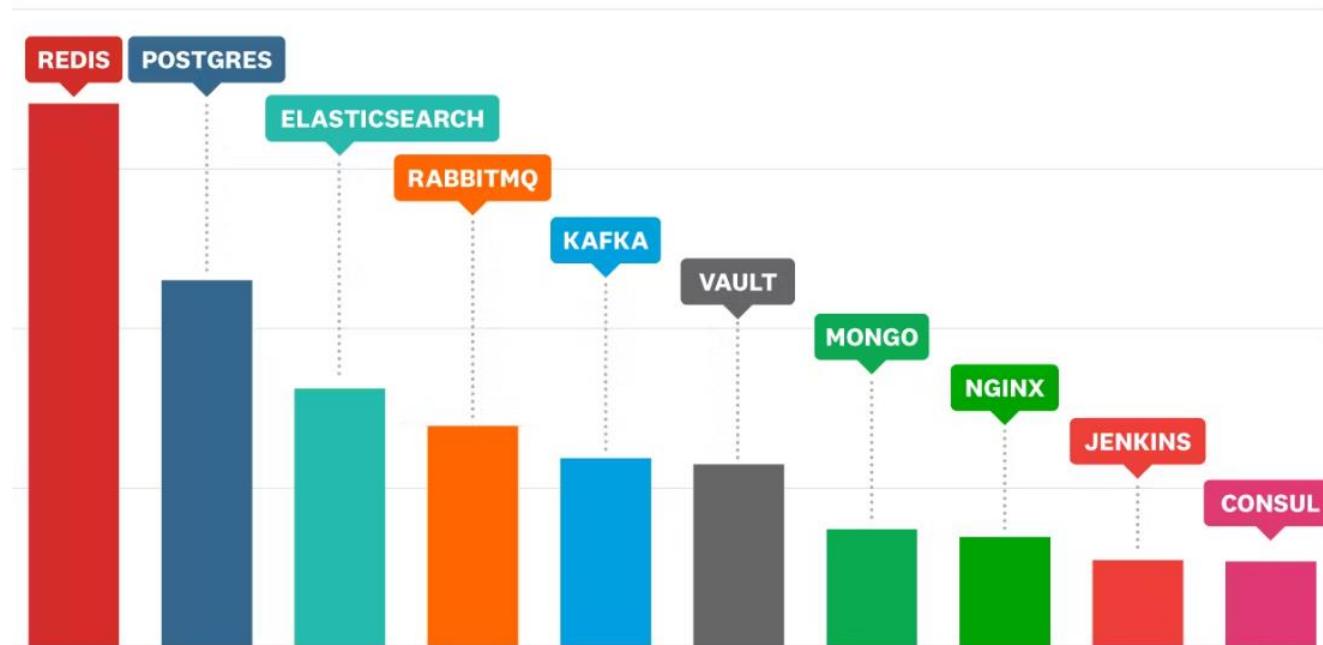




Kubernetes StatefulSets: Redis, Postgres, Elasticsearch, RabbitMQ, and Kafka

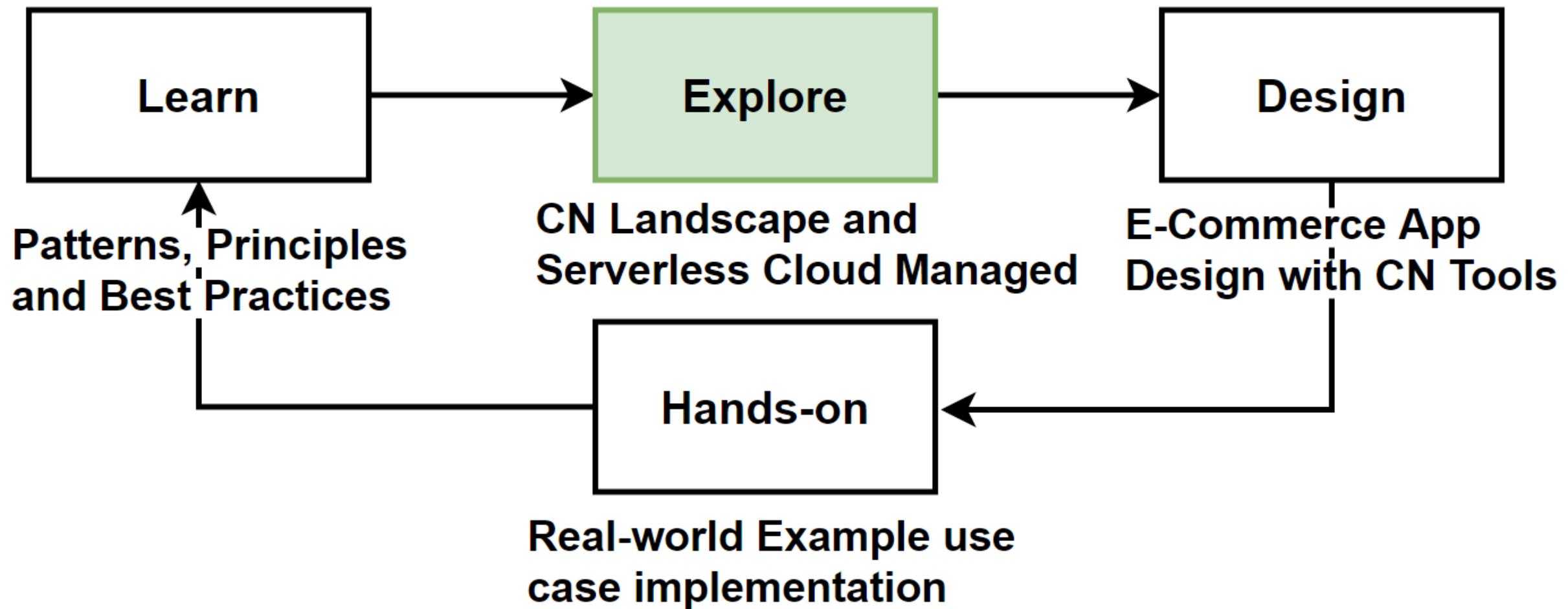
- [Datadog Report: The Most Popular Container Images: Redis, Postgres, ElasticSearch, Kafka, MongoDB](#)
- **Kubernetes StatefulSets:** Redis, Postgres, Elasticsearch, RabbitMQ, and Kafka were the most commonly deployed images.

Top Container Images Running in Kubernetes StatefulSets

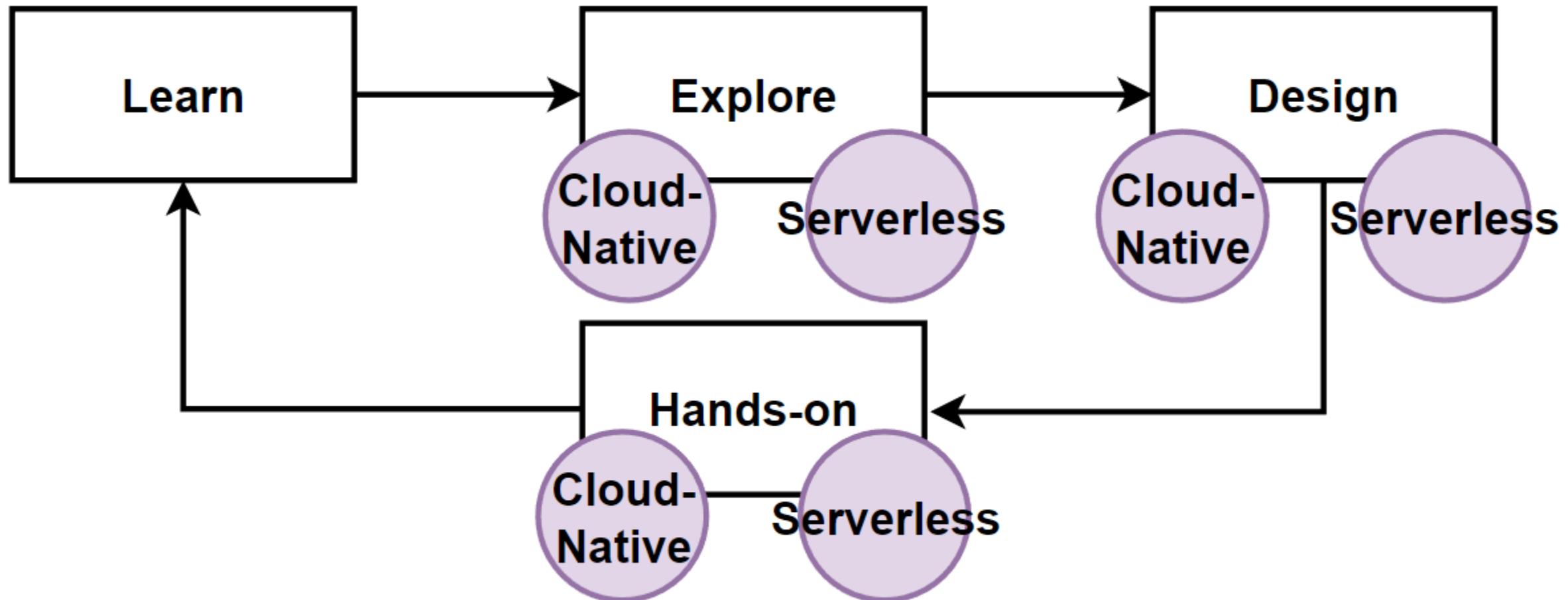


<https://www.datadoghq.com/container-report/#9>

Explore: Cloud Managed and Serverless Microservices Frameworks

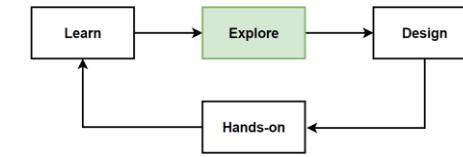


Way of Learning – Cloud-Native & Serverless Cloud Managed Tool



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Explore: Cloud-Native Databases: Relational, NoSQL NewSQL Databases

Relational Databases

- PostgreSQL
- MySQL
- Oracle
- SQL Server

NoSQL Databases

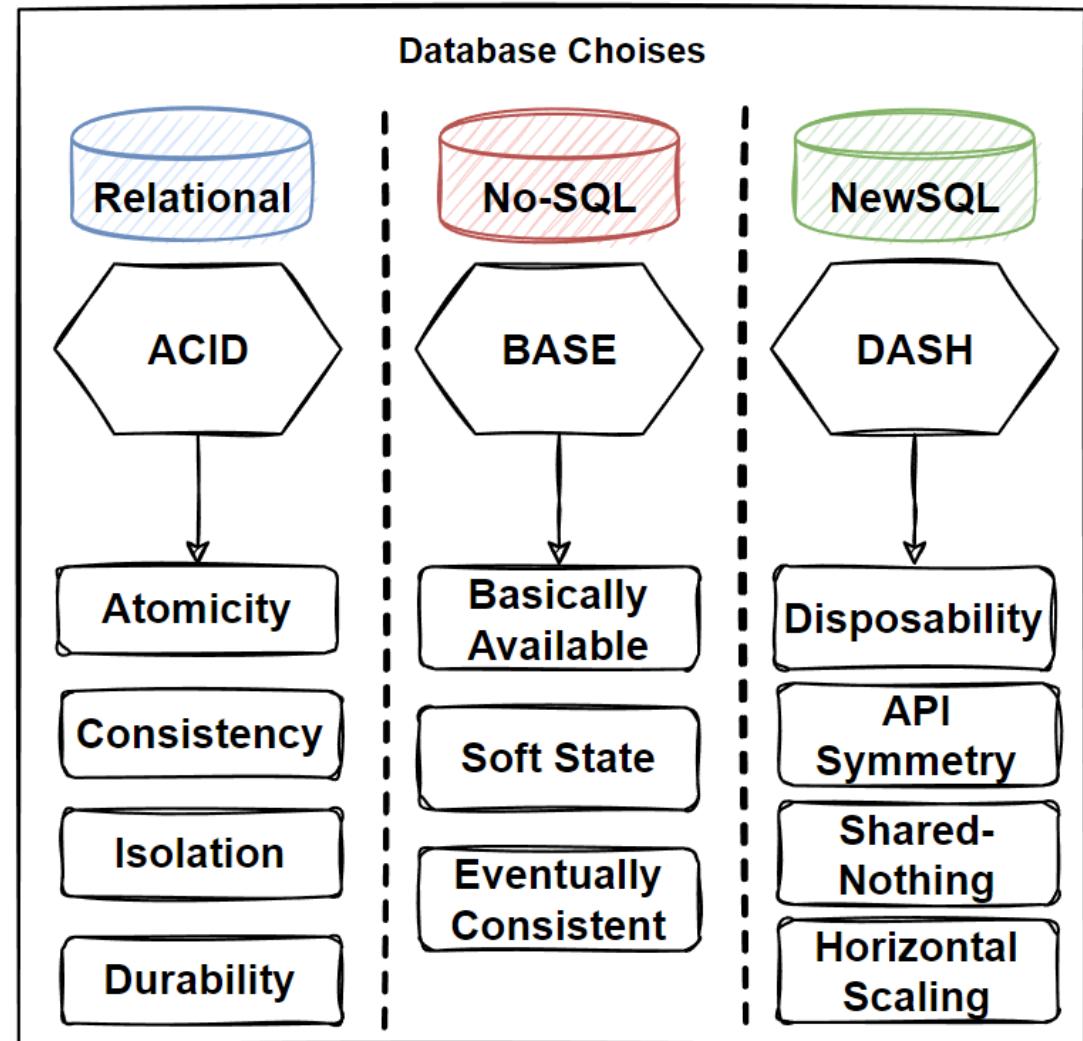
- MongoDB
- Redis
- Cassandra

NewSQL Databases

- Vitess
- TiDB - TiKV
- CockroachDB
- YugabyteDB

Cloud Serverless Databases

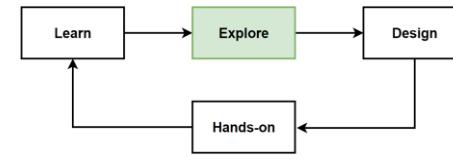
- Amazon DynamoDB
- Azure CosmosDB
- Google Cloud Spanner



Goto -> <https://landscape.cncf.io/>

Mehmet Ozkaya

448



Explore: Relational Databases

Relational Databases

- PostgreSQL
- MySQL
- Oracle
- SQL Server
- Goto
- <https://landscape.cncf.io/>

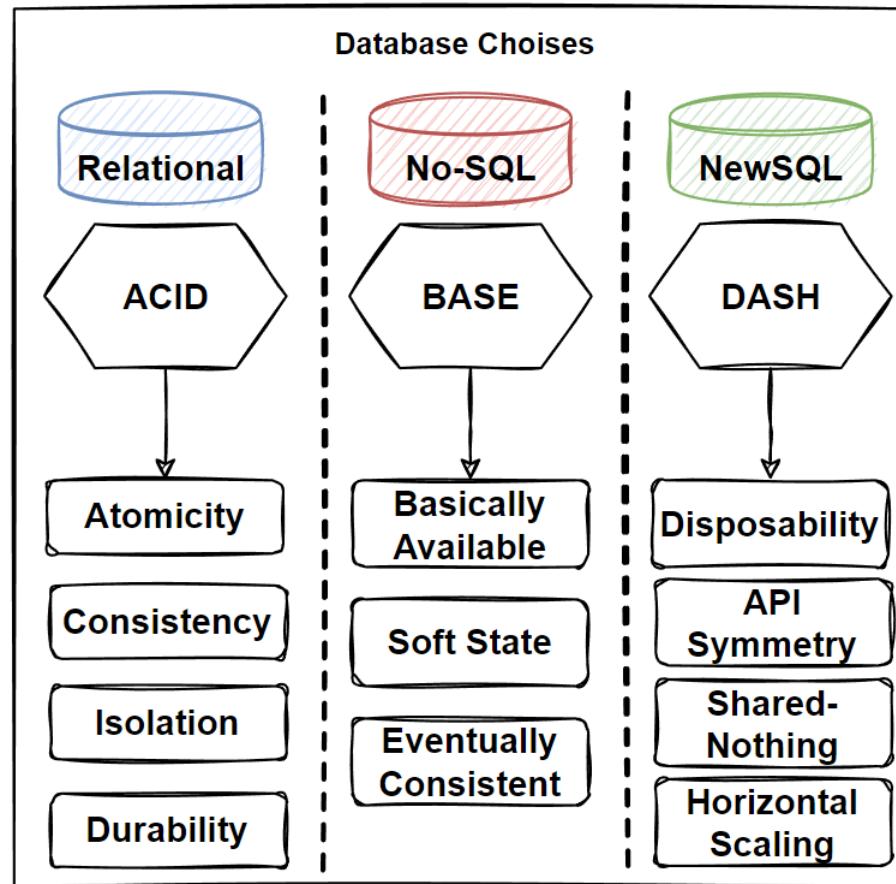
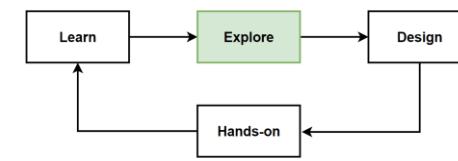


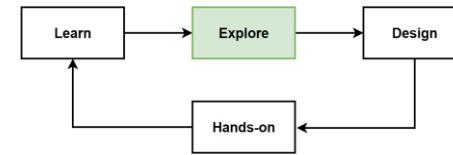
PostgreSQL

Explore: NoSQL Databases

NoSQL Databases

- MongoDB
- Redis
- Cassandra
- Goto
- <https://landscape.cnfc.io/>





Explore: Cloud-Native NewSQL Kubernetes Databases: Horizontally scalable Distributed Databases

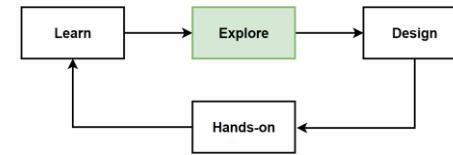
- Designed to **work seamlessly with cloud-native architectures** and scale out by adding more nodes to the system, rather than scaling up by increasing resources in a single node.
- Built to provide **high availability, fault tolerance, and consistency** across multiple nodes, making them an **excellent fit for microservices** and other cloud-native applications.

NewSQL Kubernetes Databases

- Vitess
- TiDB - TiKV
- CockroachDB
- YugabyteDB

- Goto
- <https://landscape.cncf.io/>



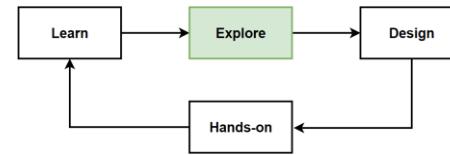


Explore: YugabyteDB: The Scalable Cloud Native DB

- **YugabyteDB** is a high-performance, cloud-native, distributed SQL database that aims to support all PostgreSQL features.
- Best suited for **cloud-native real-time** and **business-critical applications** that need absolute data correctness and required to **scalability**, **high tolerance** to failures, or **globally-distributed** deployments.
- Goto -> <https://www.yugabyte.com/>



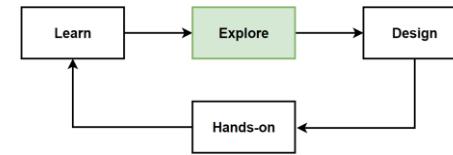
yugabyte**DB**



Explore: Vitess: Scalable. Reliable. MySQL-compatible. Cloud-native. Database.

- **Vitess** is a database clustering system for horizontal scaling of MySQL through generalized sharding.
- **Vitess** has been a core component of YouTube's database infrastructure since 2011, and has grown to encompass tens of thousands of MySQL nodes.
- **Vitess** is a Cloud Native Computing Foundation graduated project.
- Goto -> <https://vitess.io/>





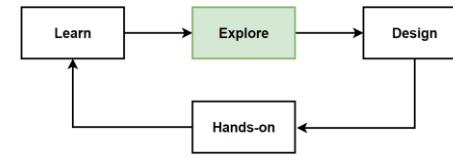
Explore: CockroachDB: A distributed SQL database built for Kubernetes

- CockroachDB is a **distributed SQL database** built on a **transactional** and **strongly-consistent key-value store**.
- **Scales horizontally**; survives disk, machine, rack, and even datacenter failures with minimal latency disruption and no manual intervention; **supports strongly-consistent ACID transactions**; and provides a **familiar SQL API** for structuring, manipulating, and querying data.
- CockroachDB is a **distributed SQL database** built for **Kubernetes**
- CockroachDB is architected and **built from the ground up** to deliver on the **core distributed principles** of **atomicity, scale** and **survival** so you can manage your database in **Kubernetes**, not along the side of it.
- **Using StatefulSets**, CockroachDB is a **natural for deployment within a Kubernetes cluster**. Simply attach storage and CockroachDB **handles distribution** of data across nodes and will survive any failure.
- Goto -> <https://www.cockroachlabs.com/>



CockroachDB





Explore: Cloud Serverless Databases

- **Managed serverless databases** are **cloud-based database** services that automatically handle the operational aspects of database management, like **provisioning, scaling, backup, and maintenance**.
- Allow developers to **focus on building apps** without worrying about infrastructure management. These databases can **scale seamlessly** with the needs of app, and you only pay for the resources you consume.

Amazon Cloud Databases

- [Amazon DynamoDB](#)
- [Amazon Aurora Serverless DB](#)



Google
Cloud
Spanner



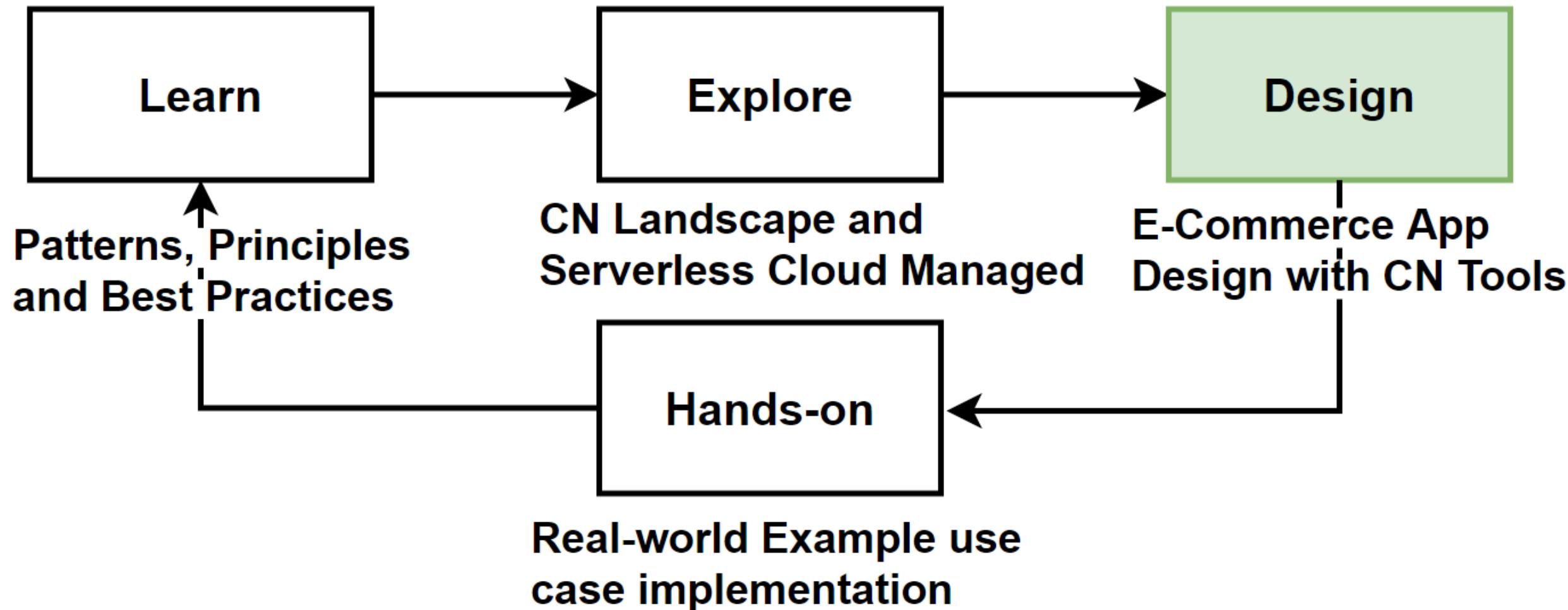
Azure Cloud Databases

- [Azure CosmosDB](#)
- [Azure SQL Database](#)

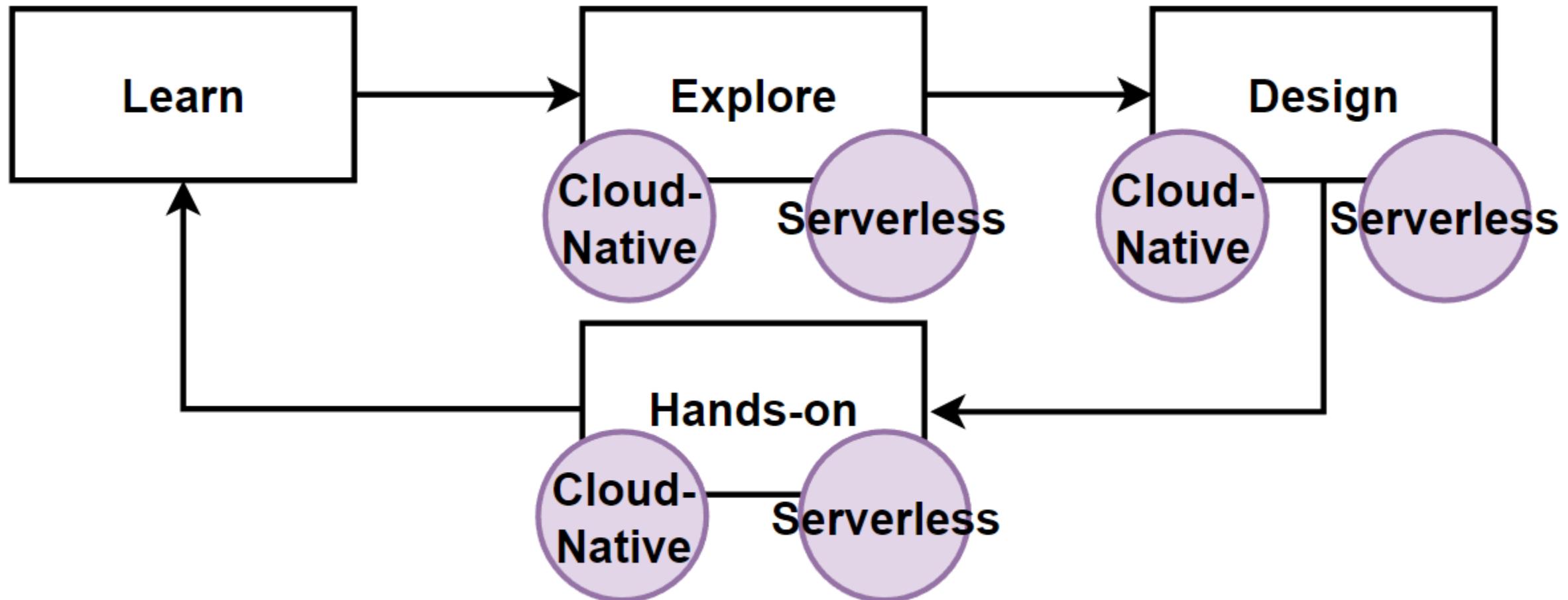
Google Cloud Databases

- [Google Cloud Spanner](#)
- [Google Cloud Firestore](#)

Way of Learning – The Course Flow



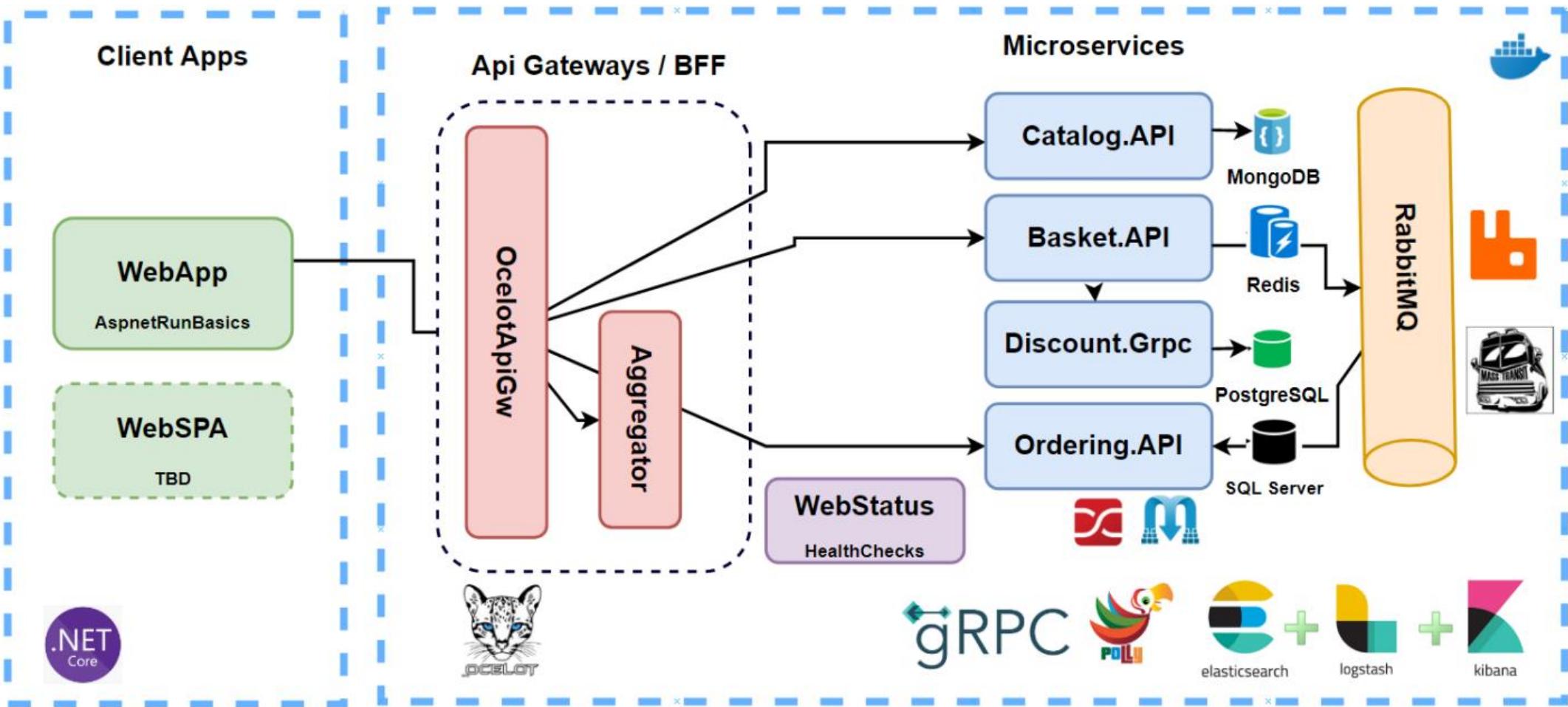
Way of Learning – Cloud-Native & Serverless Cloud Managed



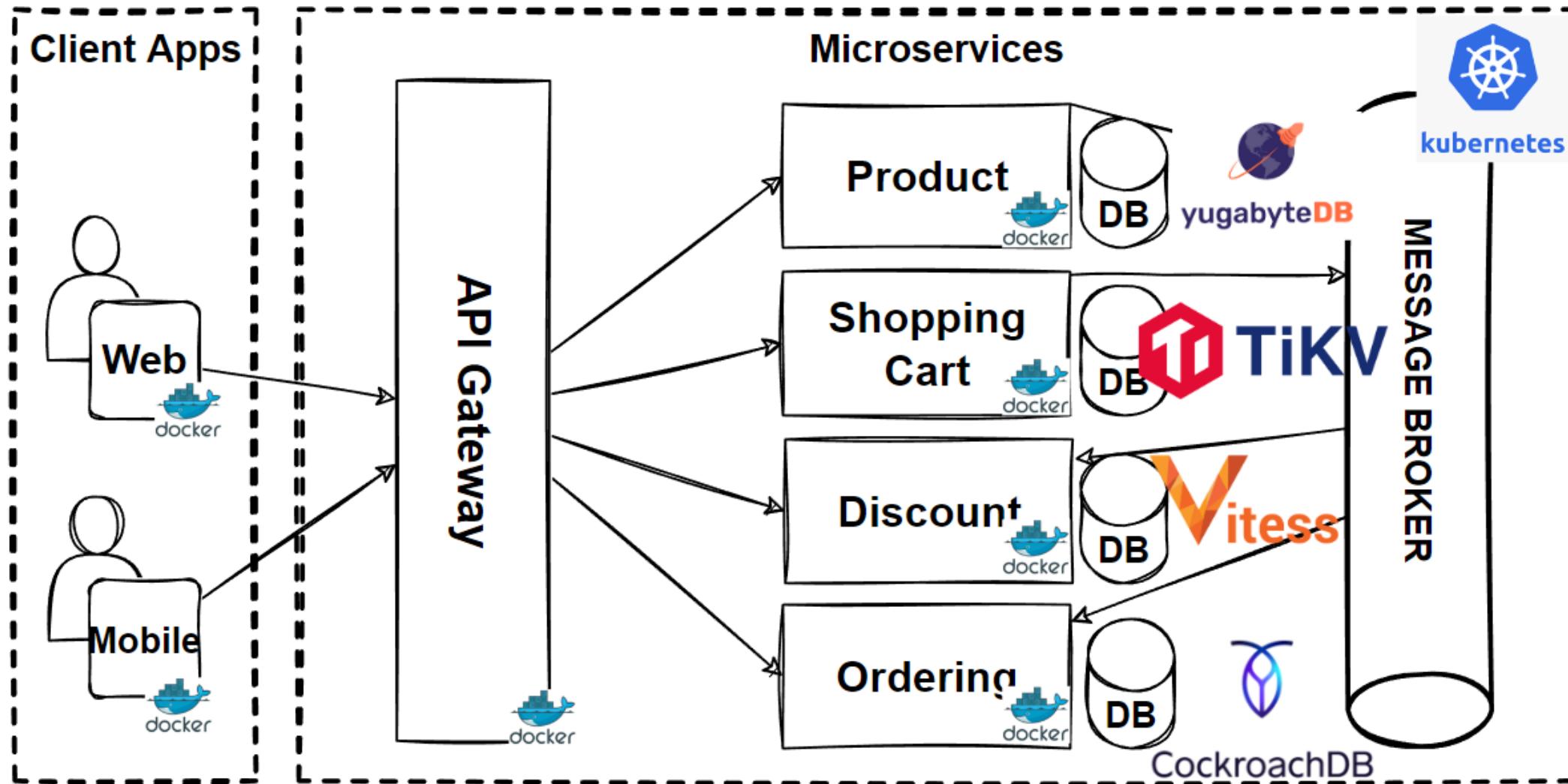
Examples of CN vs Serverless Cloud Tools:

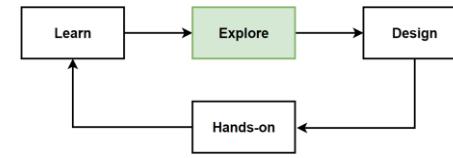
- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

Design: Microservices with Relational and NoSQL Databases



Design: Microservices with NewSQL Kubernetes Databases





Explore: Cloud Serverless Databases

- **Managed serverless databases** are **cloud-based database** services that automatically handle the operational aspects of database management, like **provisioning, scaling, backup, and maintenance**.
- Allow developers to **focus on building apps** without worrying about infrastructure management. These databases can **scale seamlessly** with the needs of app, and you only pay for the resources you consume.

Amazon Cloud Databases

- [Amazon DynamoDB](#)
- [Amazon Aurora Serverless DB](#)



Google
Cloud
Spanner



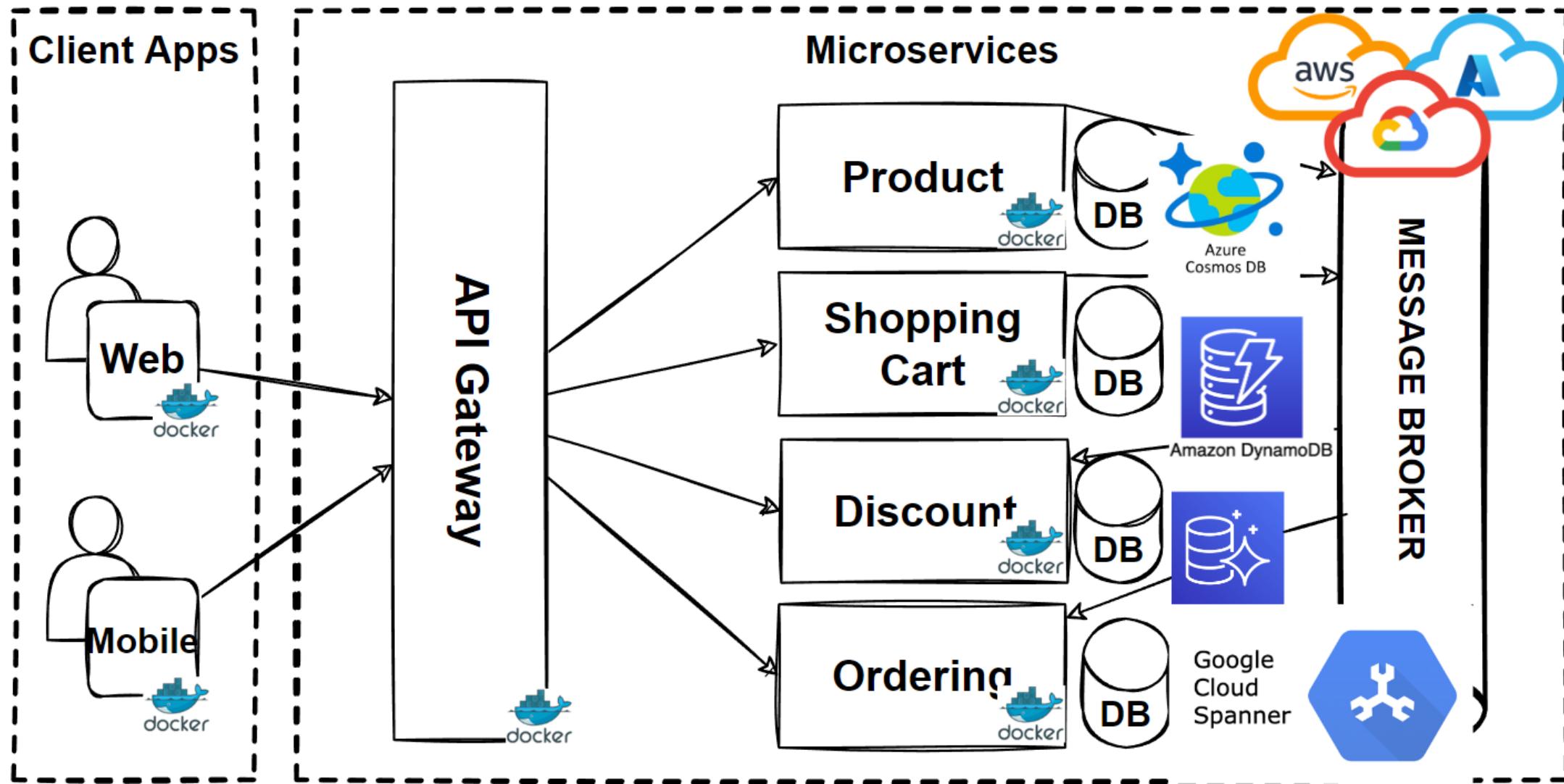
Azure Cloud Databases

- [Azure CosmosDB](#)
- [Azure SQL Database](#)

Google Cloud Databases

- [Google Cloud Spanner](#)
- [Google Cloud Firestore](#)

Design: Microservices with Cloud Serverless Databases



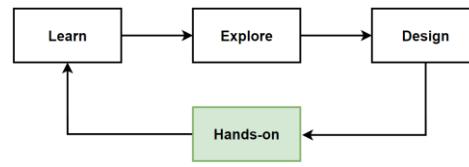
Hands-on: Deploy Cloud-Native CockroachDB Database on a Kubernetes Cluster with Minikube

Leveraging Horizontally Scalable Distributed Cloud-Native Databases

Kubernetes-Native Databases

Using StatefulSets, CockroachDB is a natural for deployment within a Kubernetes cluster

Handles distribution of data across nodes and will survive any failure



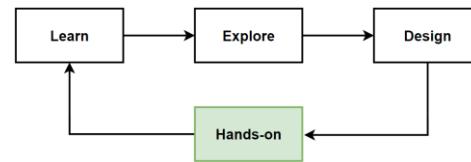
CockroachDB: A distributed SQL database built for Kubernetes

- **CockroachDB** is a distributed SQL database built on a transactional and strongly-consistent key-value store.
- **Scales horizontally**; survives disk, machine, rack, and even datacenter failures with minimal latency disruption and no manual intervention; **supports strongly-consistent ACID transactions**; and provides a **familiar SQL API** for structuring, manipulating, and querying data.
- **Survive Pod Failures**
Replicates data & automates placement across pods so you can survive failures & avoid downtime.
- **Scale with ease**
Each instance of CockroachDB is the same, allowing you to spin up new instances and scale without manual work.
- **Deploy with Kubernetes Operator**
The CockroachDB operator simplifies basic configuration and enables you to patch & roll upgrades in production.



CockroachDB





CockroachDB: Advanced Features

- Don't federate clusters, distribute data**

Combines the familiarity of relational data with limitless cloud scale, allowing you to deploy a single logical database across multiple regions and still guarantee transactional consistency

- Multi-region, Multi-cluster**

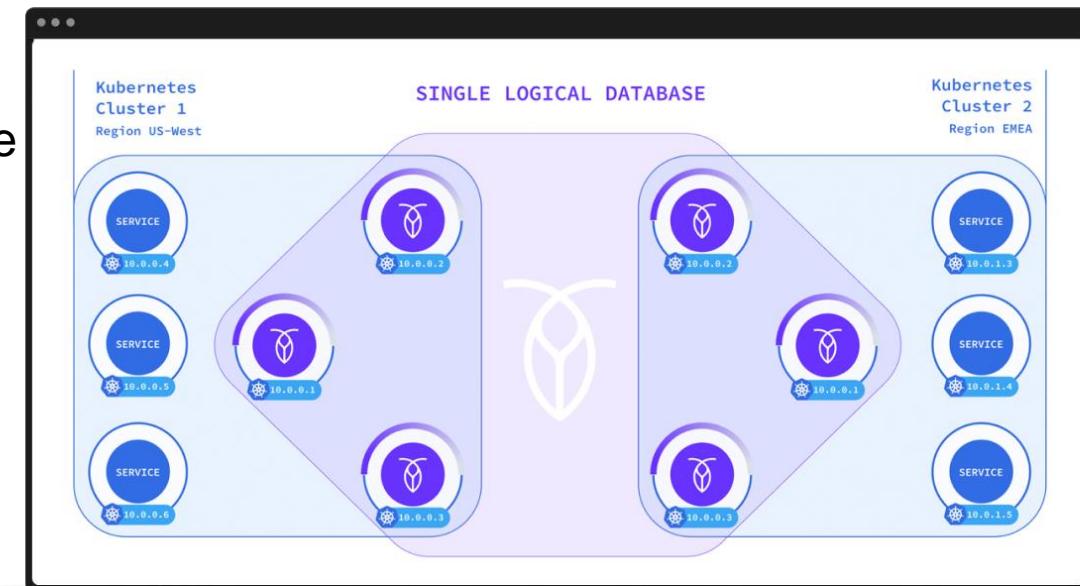
Deployed across multiple Kubernetes clusters and function like a single logical database, eliminating the need to federate clusters.

- Multi-master, Guaranteed transactions**

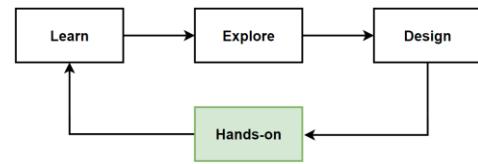
Break free from manual sharding and complex workarounds. CockroachDB dramatically simplifies scaling applications. Expand with simple, familiar SQL DML.

- No downtime upgrades, patches & schema changes**

Roll software updates, patches, and schema modifications to each node, one at a time without service disruption.



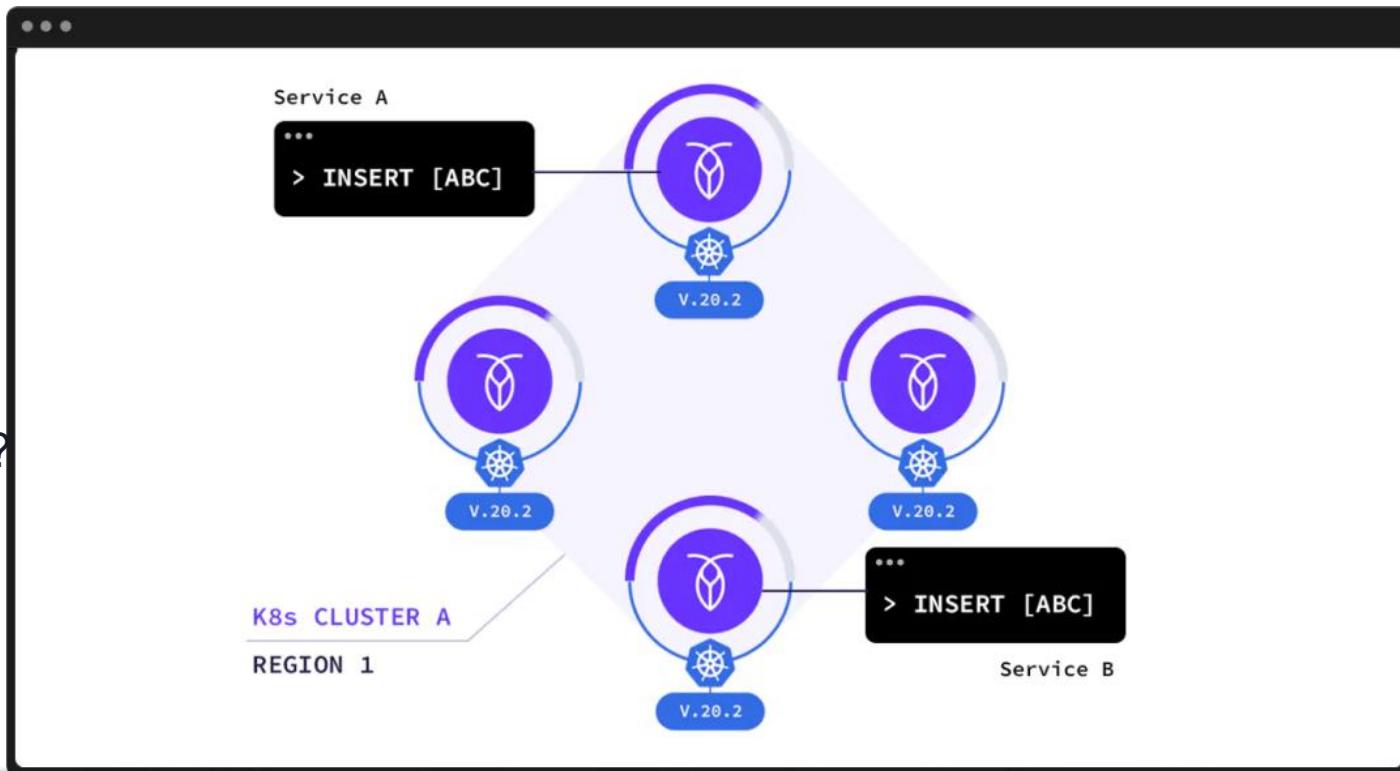
<https://www.cockroachlabs.com/product/kubernetes/>



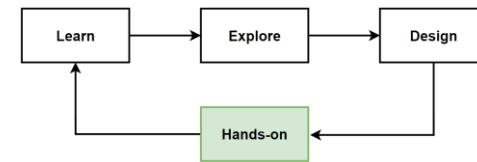
Hands-on: Deploy CockroachDB in a Single Kubernetes Cluster with Minikube – Task List

- Step 1. Start Kubernetes - minikube start
- Step 2. Start CockroachDB - Deploy with Kubernetes Operator
- Step 3. Use the built-in SQL client
- Step 4. Access the DB Console
- Step 5. Simulate node failure and node scales
- Step 6. Stop the cluster

- What is K8s Operator and Why use Operators ?
- CockroachDB Kubernetes Operator



<https://www.cockroachlabs.com/product/kubernetes/>

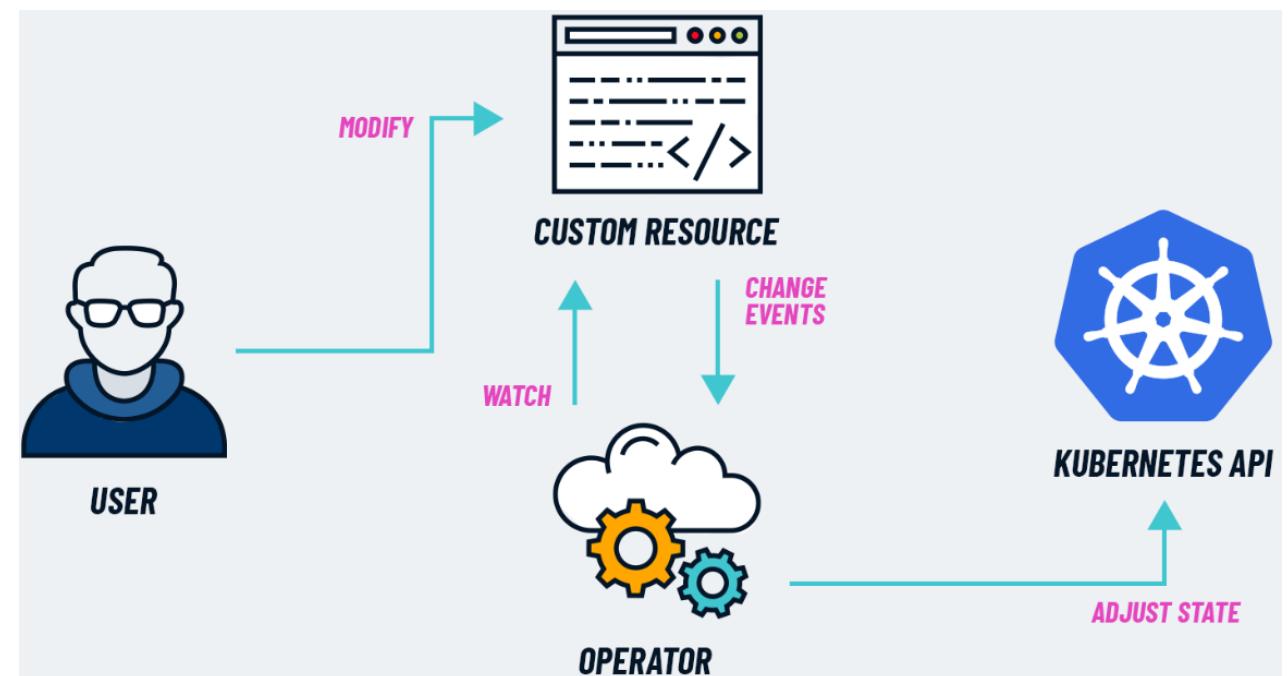


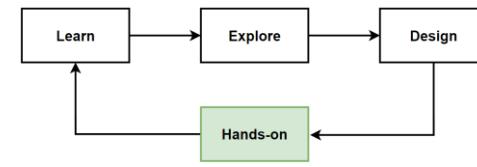
What is Kubernetes Operator ?

- **Kubernetes Operator** is a method of **packaging**, **deploying**, and **managing** a **Kubernetes application**.
- **Operators** are a **design pattern** for managing **applications** and infrastructure on Kubernetes, implemented as custom controllers and **custom resources**.
- **Extension of the Kubernetes API** that **encapsulates** the **operational knowledge** of running and managing a specific application or service.

Kubernetes Operator Control Loop

1. Monitors the state of your application.
2. Compares the actual state to the desired state.
3. Updates the application if needed to achieve the desired state.





Why use Kubernetes Operators ?

Application-specific operations

- Operators know how to create a database backup, how to upgrade an application, or how to scale a service.

Automation

- Operators can automate many tasks related to the application lifecycle, such as deployment, backup, failure recovery, upgrades, and scaling.

Declarative control

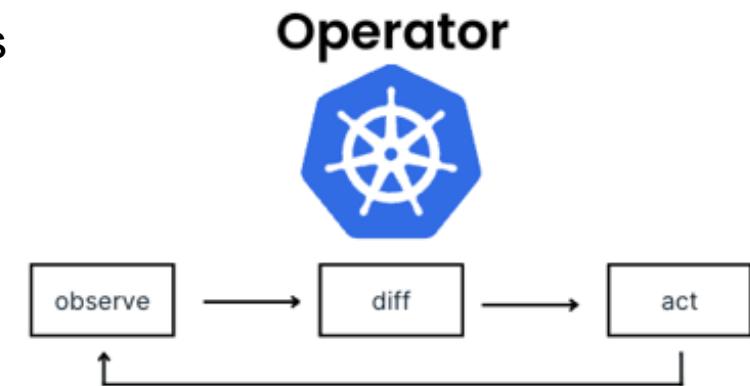
- Operators use a declarative approach, where you describe the desired state, and the Operator takes the necessary actions to achieve that state.

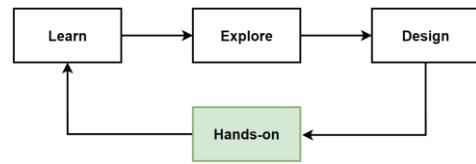
Repeatability and consistency

- Operators help to make operations repeatable and consistent across different environments.

Self-healing applications

- By using operators, you can make your applications self-healing, which means they can automatically recover from failures.





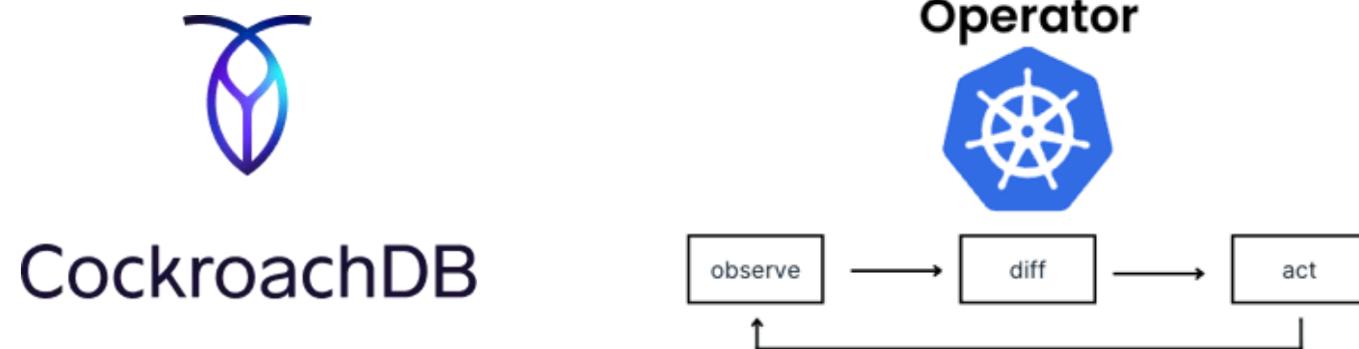
CockroachDB Kubernetes Operator

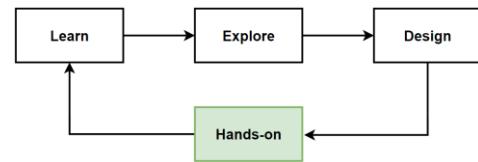
CockroachDB Kubernetes Operator

- Deploy CockroachDB clusters on Kubernetes with the easiest and most effective database available for Kubernetes.

GitHub Link of Kubernetes Operator

- <https://github.com/cockroachdb/cockroach-operator>

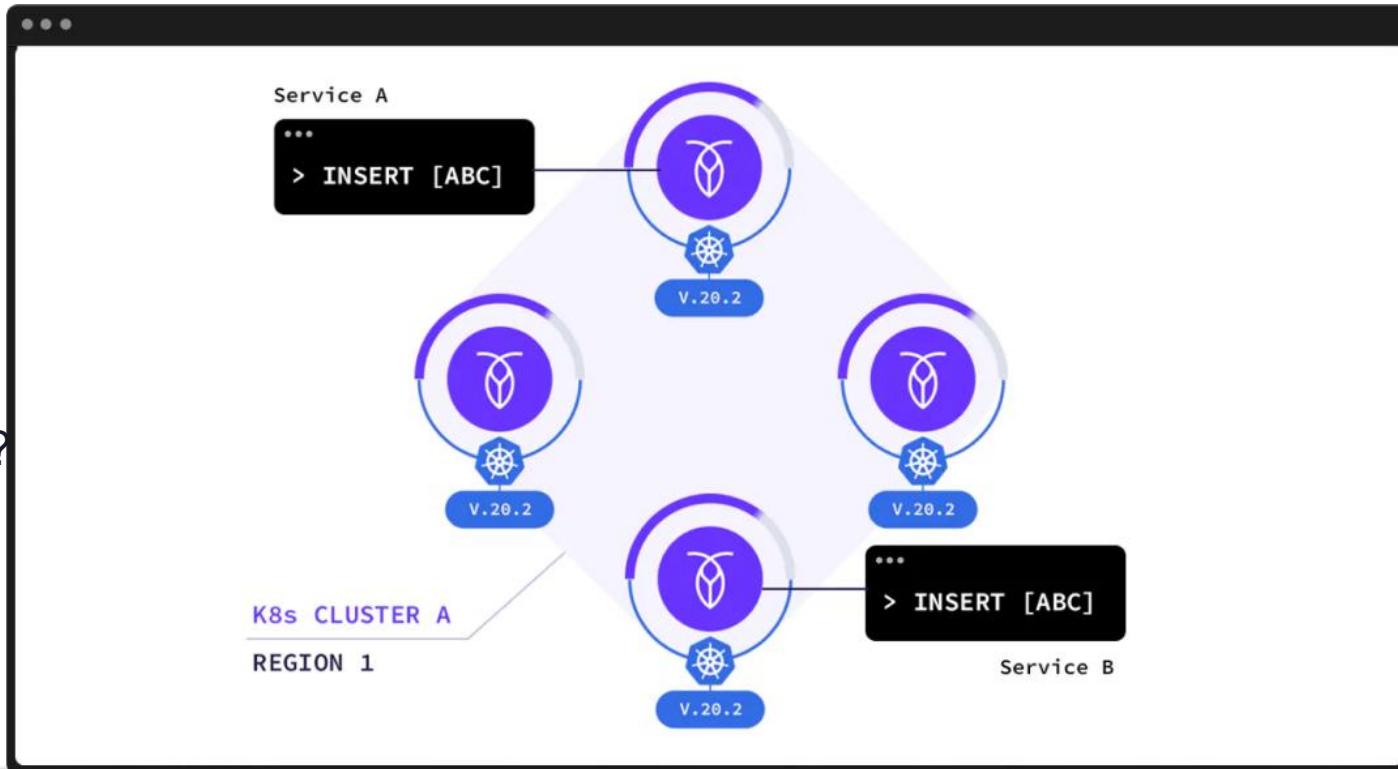




Hands-on: Deploy CockroachDB in a Single Kubernetes Cluster with Minikube – Task List

- Step 1. Start Kubernetes - minikube start
- Step 2. Start CockroachDB - Deploy with Kubernetes Operator
- Step 3. Use the built-in SQL client
- Step 4. Access the DB Console
- Step 5. Simulate node failure and node scales
- **Step 6. Stop the cluster**

- What is K8s Operator and Why use Operators ?
- CockroachDB Kubernetes Operator



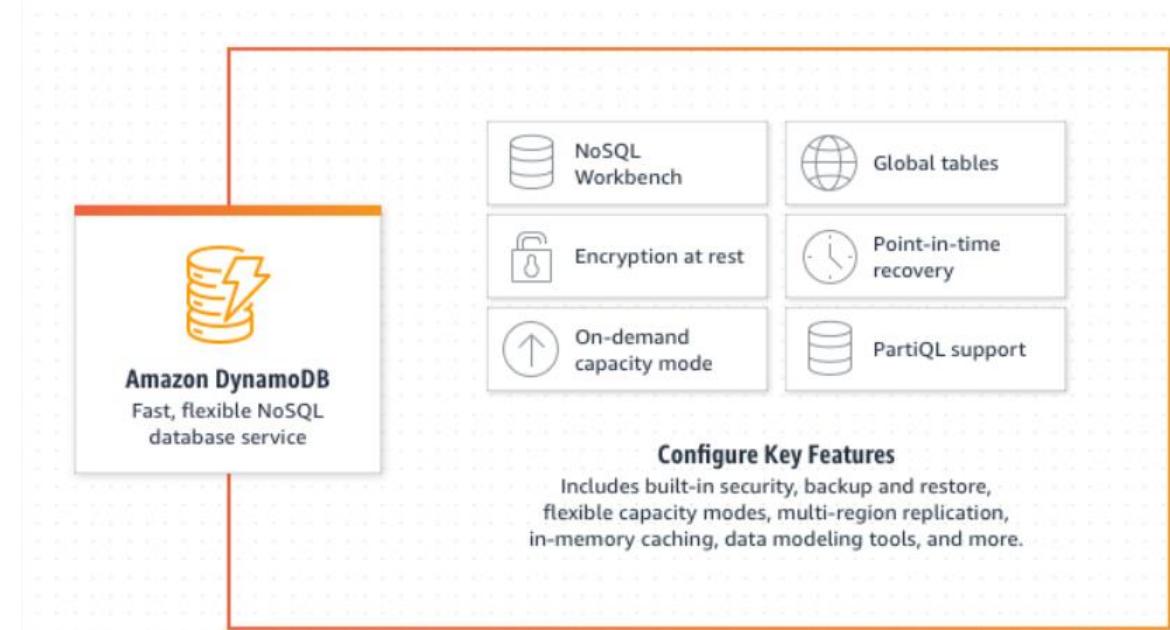
<https://www.cockroachlabs.com/product/kubernetes/>

Hands-on: Building RESTful Microservices with AWS Lambda, API Gateway and DynamoDB

Building RESTful Microservices with AWS Lambda, API Gateway and DynamoDB

What Is Amazon DynamoDB?

- **Amazon DynamoDB** is a fully managed NoSQL database service that provides **fast** and **predictable** performance with seamless scalability.
- **Serverless, key-value NoSQL database** designed to run high-performance applications at any scale.
- Create **database tables** that can store and retrieve any amount of data and serve any level of request traffic.
- **Scale up or down** the throughput of your tables without downtime or performance degradation. DynamoDB provides on-demand backup capability.
- **High Availability and Durability**
DynamoDB automatically spreads data and traffic for your tables across enough servers to meet your throughput.



<https://aws.amazon.com/dynamodb/>

AWS DynamoDB Core Concepts - Tables, Items, Attributes, Indexes

- Tables, Items, and Attributes are the core components.
- Uses primary keys and secondary indexes to uniquely identify each item in a table for greater query flexibility.

- **Tables**

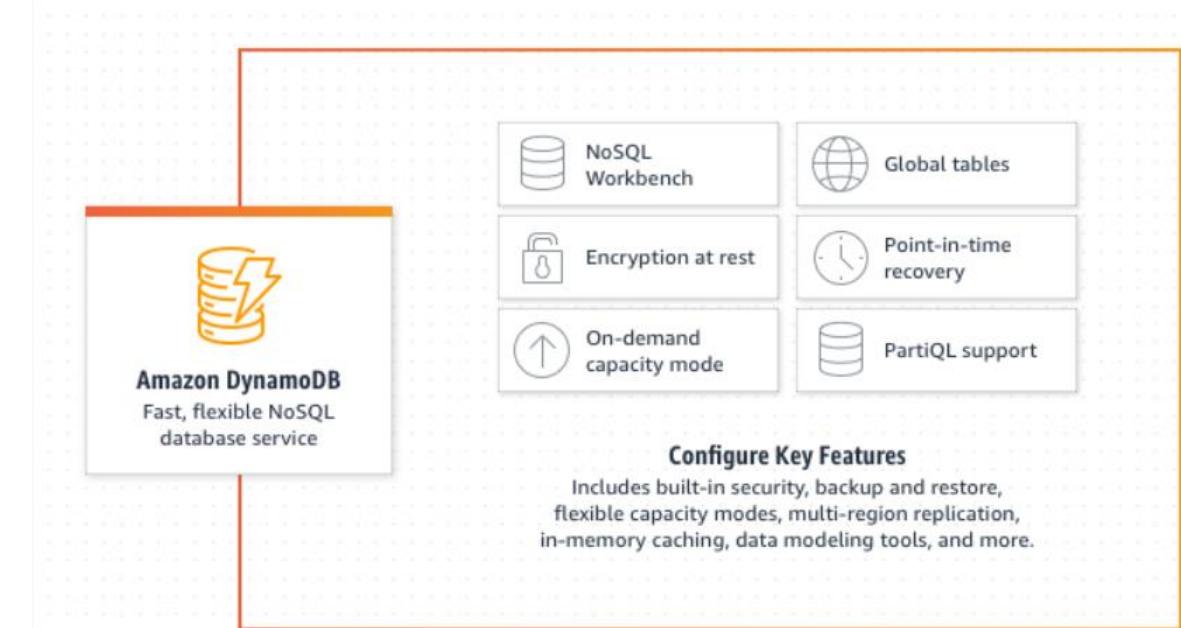
DynamoDB stores data in tables. A table is a collection of data items. For example, see the sample table People.

- **Items**

Each table contains zero or more items. An item is a set of attributes that can be uniquely identified among all of the other items.

- **Attributes**

Each item is composed of one or more attributes. An attribute is a fundamental data element.



<https://aws.amazon.com/dynamodb/>

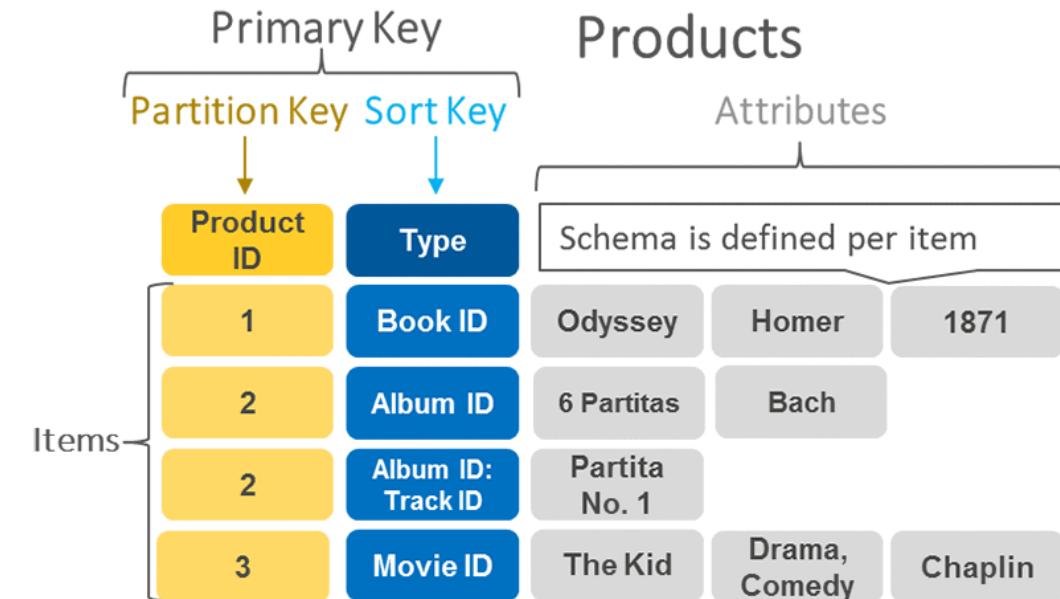
AWS DynamoDB Core Concepts - Tables, Items, Attributes, Indexes



<https://docs.aws.amazon.com/amazondynamodb/latest/developerguide/HowItWorks.CoreComponents.html>

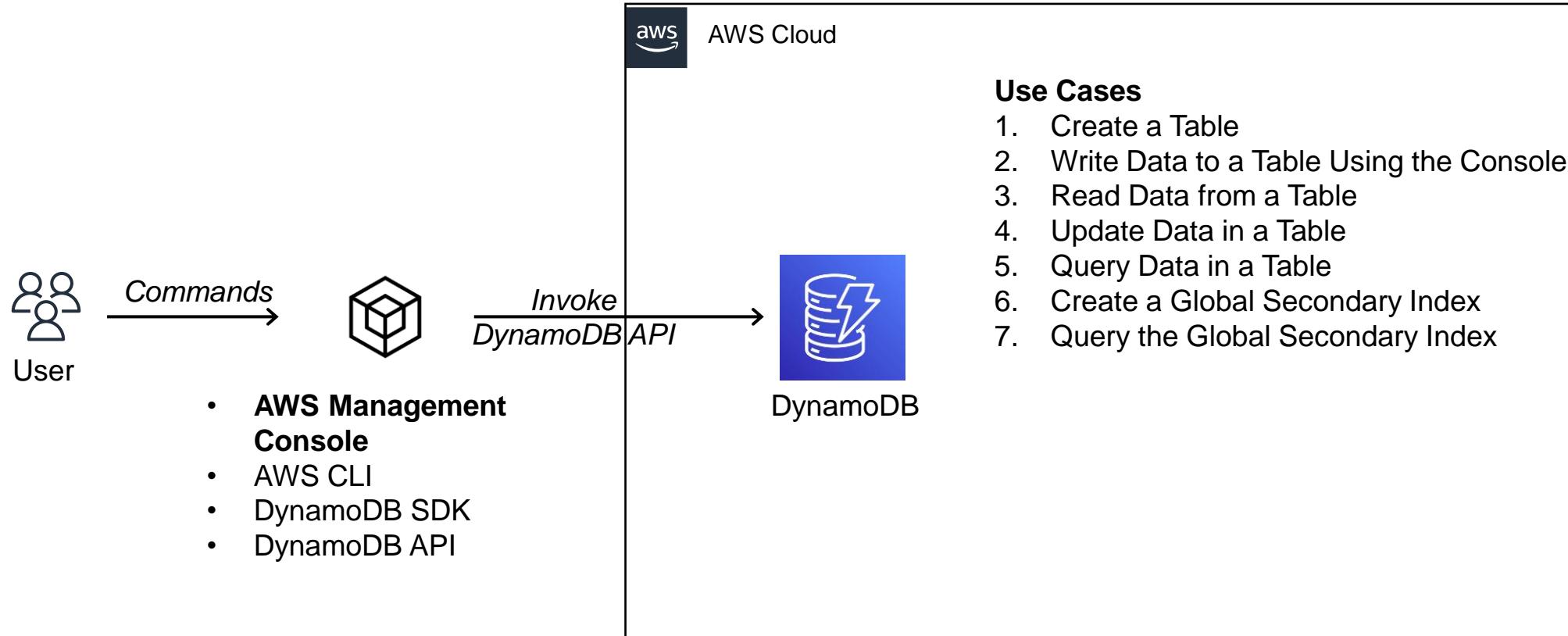
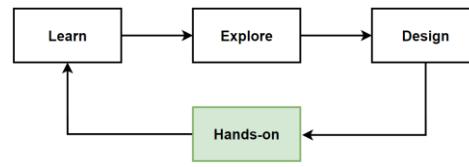
DynamoDB Primary Key, Partition Key and Sort Key

- A primary key **uniquely identifies** each item in the table, so no two items can have the same key. DynamoDB supports two different kinds of primary keys:
 - Partition key
 - Partition key and sort key
- **Partition key**
A simple primary key, composed of one attribute known as the partition key.
- **Partition key and Sort Key**
It is Referred to as a composite primary key, this type of key is composed of two attributes. The first attribute is the partition key, and the second attribute is the sort key.
- DynamoDB uses the partition key value as input to an internal hash function. A composite primary key gives you additional flexibility when **querying data**.



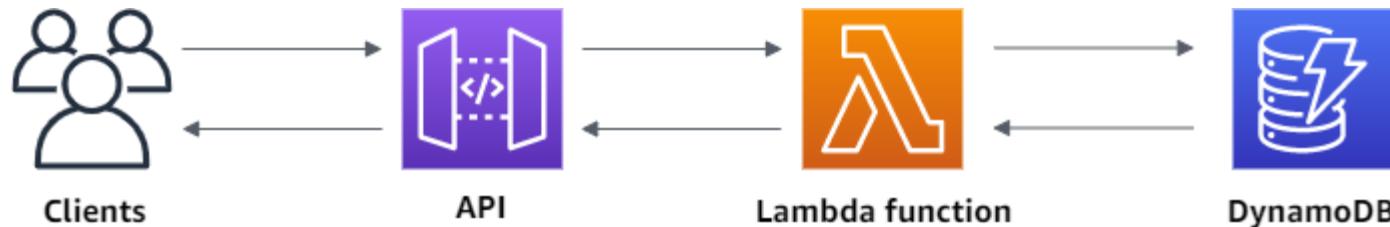
<https://aws.amazon.com/blogs/database/choosing-the-right-dynamodb-partition-key/>

Amazon DynamoDB Walkthrough with AWS Management Console



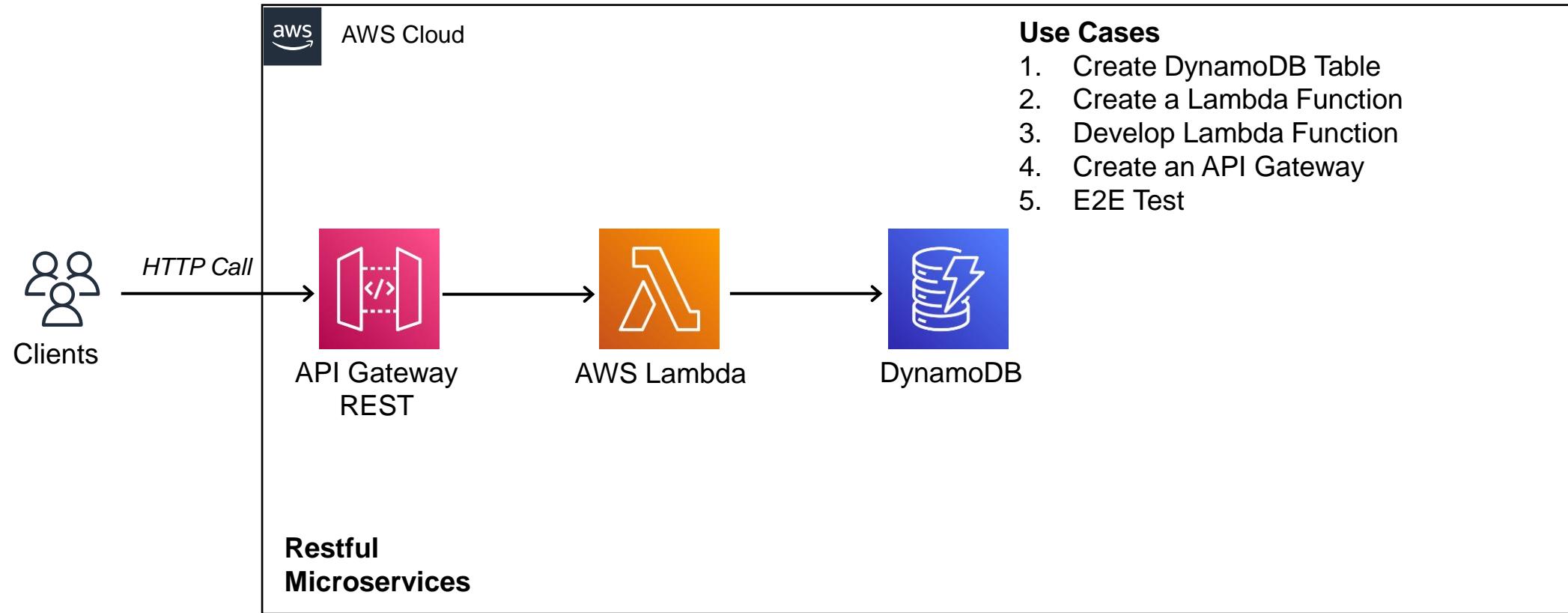
RESTful Microservices with AWS Lambda, Api Gateway and DynamoDb

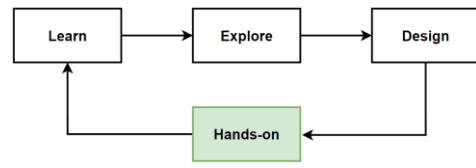
- Create a Serverless API that creates, reads, updates, and deletes items from a **DynamoDB table**.
- Create a **DynamoDB table** using the DynamoDB console.
- Create a **Lambda function** using the AWS Lambda console.
- Create an **REST API** using the API Gateway console. Lastly, we test your API.



- Clients **send request** to our **microservices** by making HTTP API calls.
- Amazon **API Gateway** hosts RESTful HTTP requests and responses to customers.
- AWS **Lambda** contains the business logic to process **incoming API calls** and leverage DynamoDB as a persistent storage.
- Amazon DynamoDB **persistently stores** **microservices** data and scales based on demand.

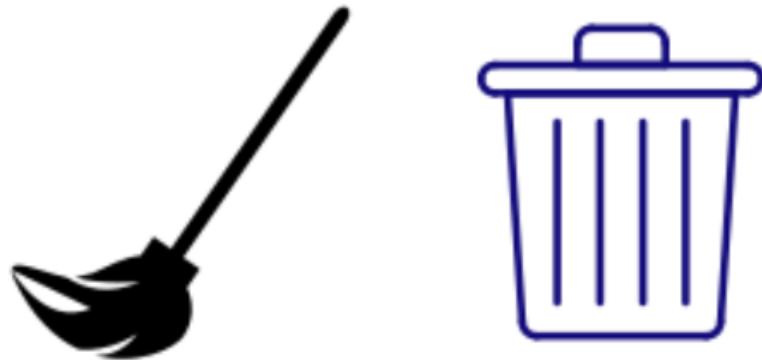
Hands-on Lab: Building RESTful Microservices with AWS Lambda, API Gateway and DynamoDB





Clean up Resources

- Delete AWS Resources that we create during the section.



Cloud-Native Pillar5: Backing Services - Caching (K8s and Serverless Caching)

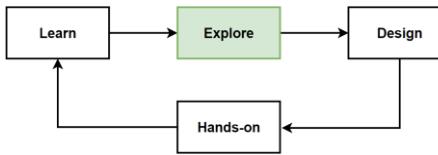
What are Cloud-Native Backing Services for Caching ?

How microservices use Caching in Cloud-Native environments ?

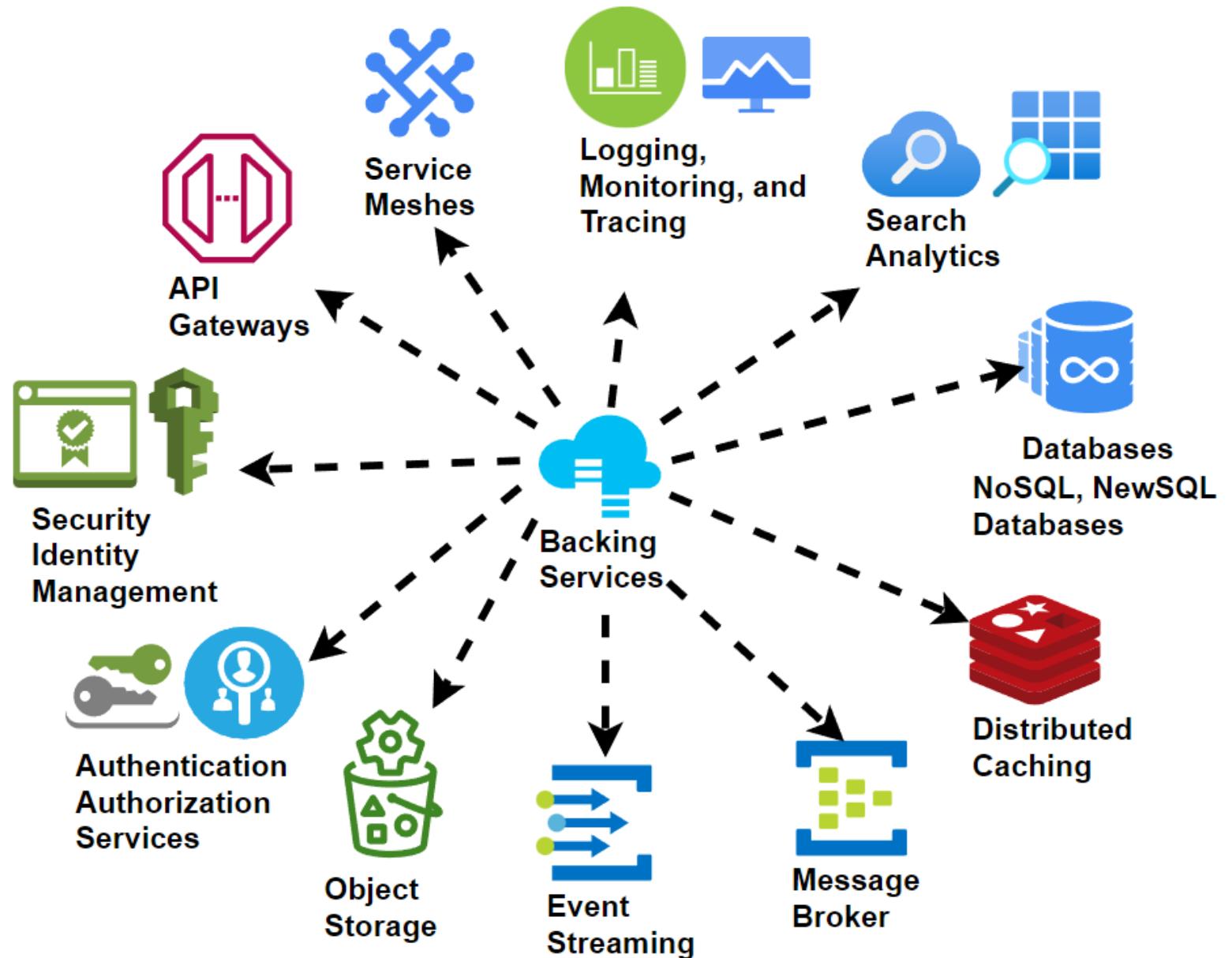
How to Choose a Caching for Microservices ?

Which Caching should select for Backing Services for Cloud-Native Microservices ?

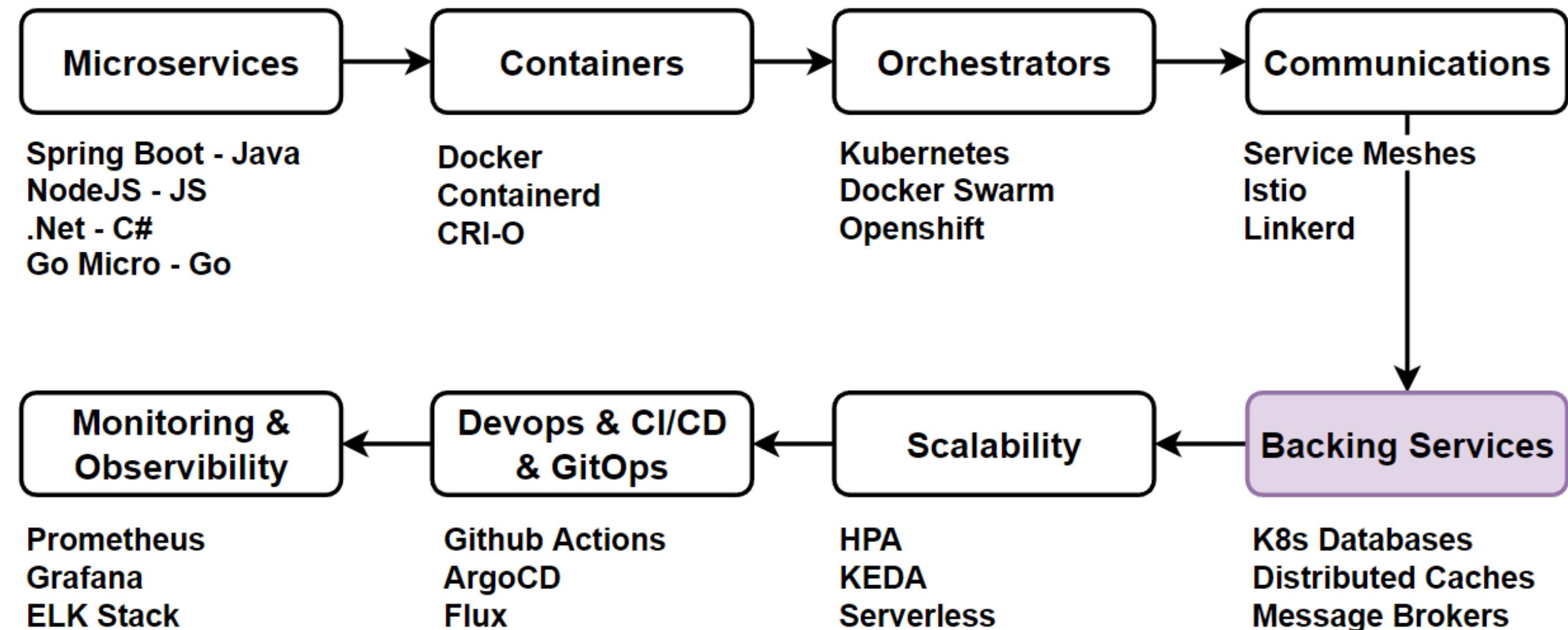
What are patterns & best practices of using Cache in Cloud-native environments ?



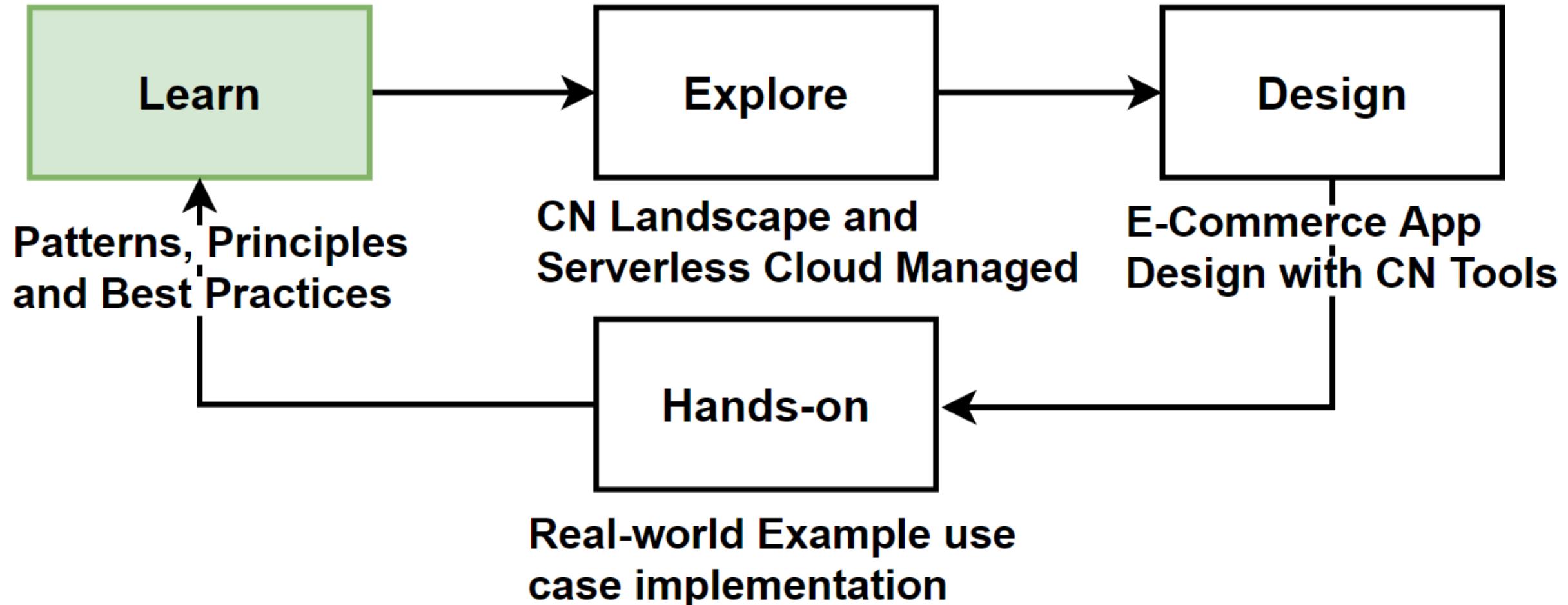
Backing Services for Cloud-Native Microservices

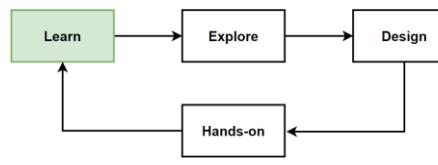


Cloud-Native Pillars Map – The Course Section Map



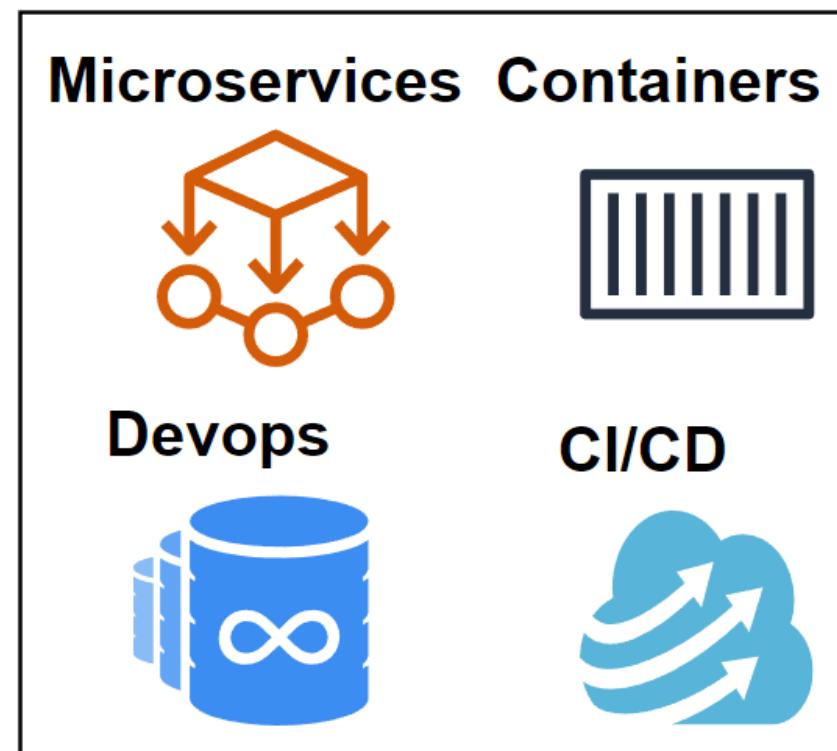
Way of Learning – The Course Flow





Backing Services - Caching

- What are Cloud-Native Backing Services for Caching ?
- Types of Caching
- Distributed Caching in Microservices
- Caching Strategies in Distributed Caching for Microservices
- How microservices use Caching in Cloud-Native environments ?
- Explore - Backing Services for Caching.
 - K8s and Serverless Caches
- Hands-on - Backing Services for Caching.
 - K8s and Serverless Caches

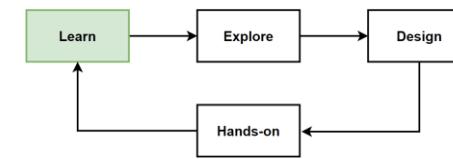


Cloud-native Trial Map – Backing Services Cache

- **Improve the performance and responsiveness** of microservices in a cloud-native environment by **storing frequently accessed data in memory, reducing latency and decreasing the load** on databases or other data stores.
- **Distributed caching** typically falls under the "**Backing Services**" pillar in the Cloud-Native Trial Map.
- **A distributed cache** is a cache that **spans multiple nodes**, so it's not confined to a single machine.
- It provides a way to **share data** that's **computationally expensive** to retrieve or generate, allowing applications to **retrieve that data more efficiently**.
- In a **microservices architecture**, a distributed cache can be used to **improve performance and reduce the load** on services and databases.
- **Significantly improves application performance** by storing **frequently accessed data** closer to the application.

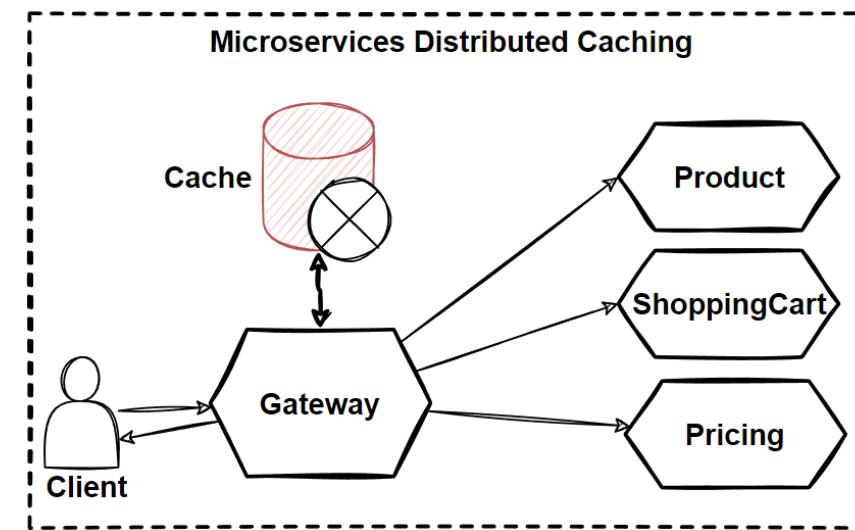


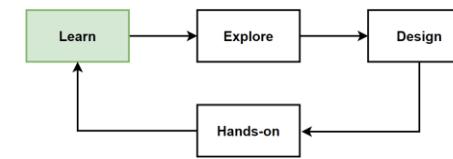
<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>



What is Caching ?

- **Caching** for improving the **performance** of a system by **storing frequently accessed data** in a **cache** that can be **quickly accessed from memory**.
- **Caching** is to **reduce the number of expensive operations**, such as **database queries or network requests**.
- **Caching** can **increase performance, scalability, and availability** for microservices with **reducing latency** with cache and makes application **faster**.
- When the **number of requests are increased**, caching provide to **handle requests with high availability**.
- If application **request** mostly comes for **reading data** that is not **changes so frequently**, then **Caching** will be so **efficient**.
- I.e. **reading product catalog** from e-commerce application. Caching also provide to **avoid re-calculation processes**.
- By **storing frequently accessed data** in a **cache**, system can **avoid the overhead of repeatedly expensive operations**.





Types of Caching

- **In-memory cache**

Stores data in the main memory of a computer. In-memory caches are typically the fastest type of cache, but the data is lost when the cache is restarted or the machine is shut down.

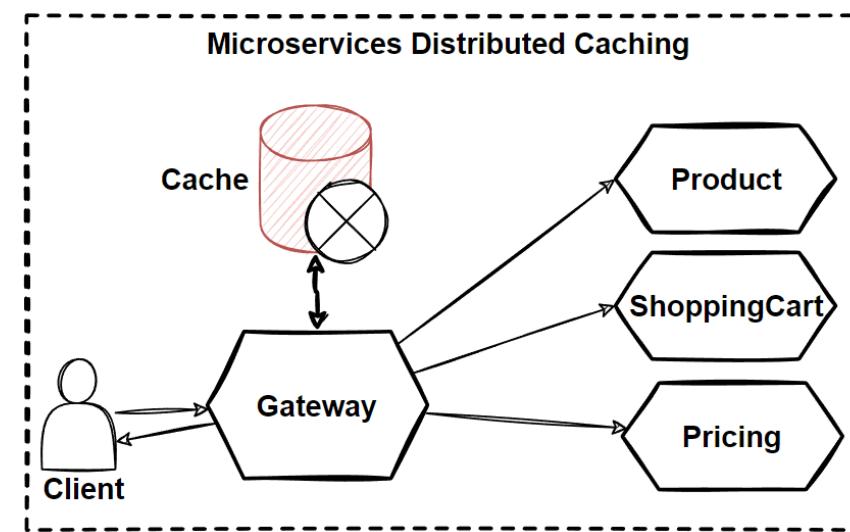
- **Disk cache**

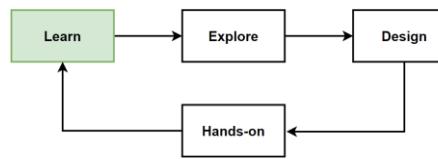
Stores data on a hard drive or solid-state drive. Disk caches are slower than in-memory caches, but they can persist data.

- **Distributed cache**

Cache is distributed across multiple machines and is typically used in distributed systems, such as microservices architectures.

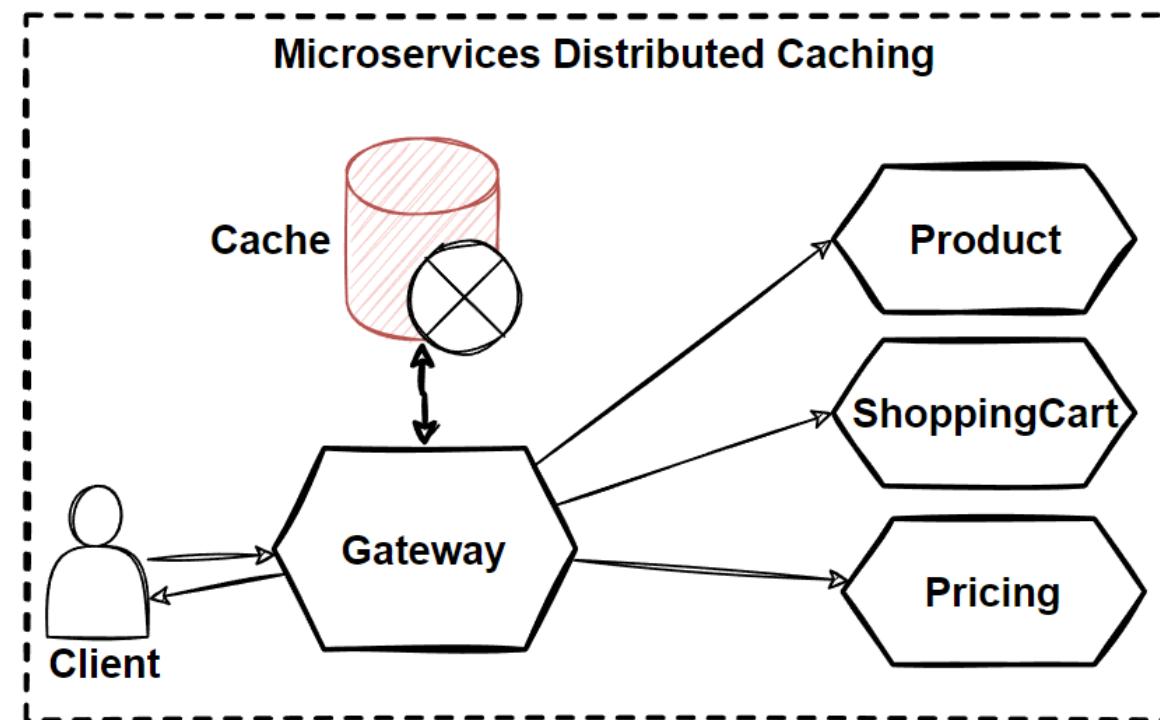
- Distributed caches can improve the performance and scalability of a system by allowing data to be stored and accessed from multiple locations.

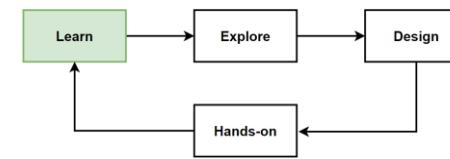




Distributed Caching in Microservices

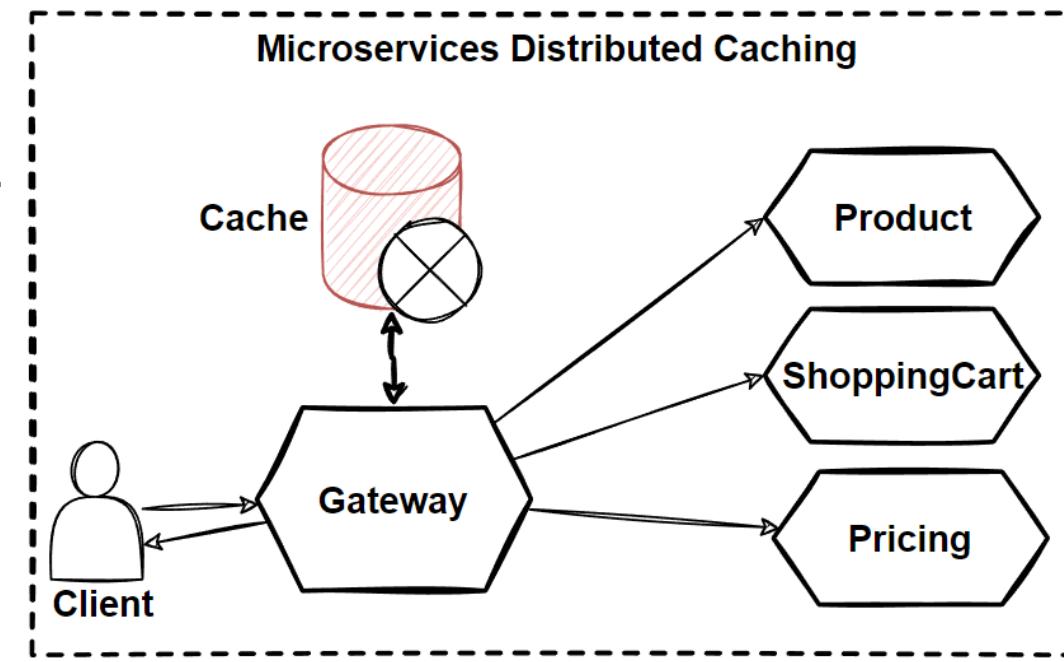
- **Distributed caching** is improving the performance by **storing frequently accessed data** in a **cache** that can be **quickly accessed** from **multiple locations**.
- **Microservices architectures** are typically implement a **distributed caching** architecture:
 - Improve the performance of individual services by storing frequently accessed data locally.
 - Reducing the need to make expensive calls to a database or other external system.
- **How can we increase the speed of the microservices ?** With using **Distributed Cache**.

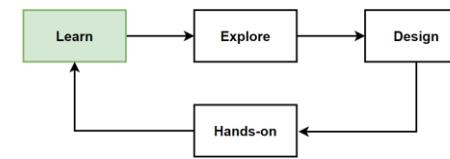




Distributed Caching in Microservices - 2

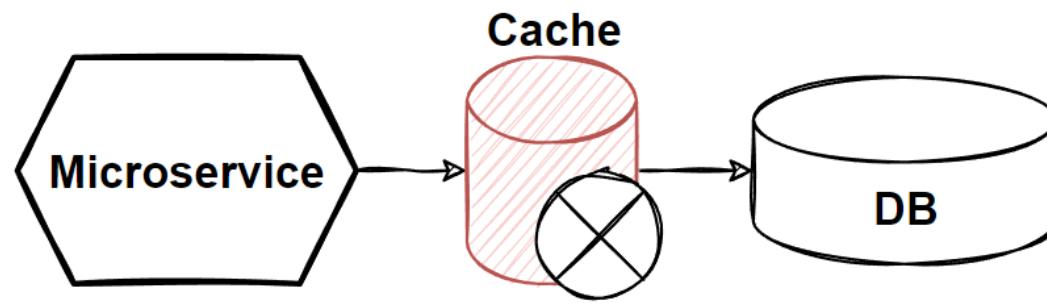
- Microservices are **responsible for a specific function** and **communicates** with other services through well-defined interfaces, **typically using APIs**.
- By **storing frequently accessed data locally in a cache**, microservices can **avoid the overhead of making repeated calls** to an external system, resulting in **faster response times**.
- Benefits to using distributed caching in a microservices:
- **Improved performance**
Services can avoid the overhead of making repeated calls to a database or other external system.
- **Resilience**
Allowing services to continue functioning even if an external system becomes unavailable.
- **Scalability**
Allowing services to handle increased traffic without the need to scale up the external system.

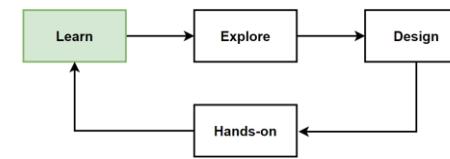




Cache Hit and Cache Miss

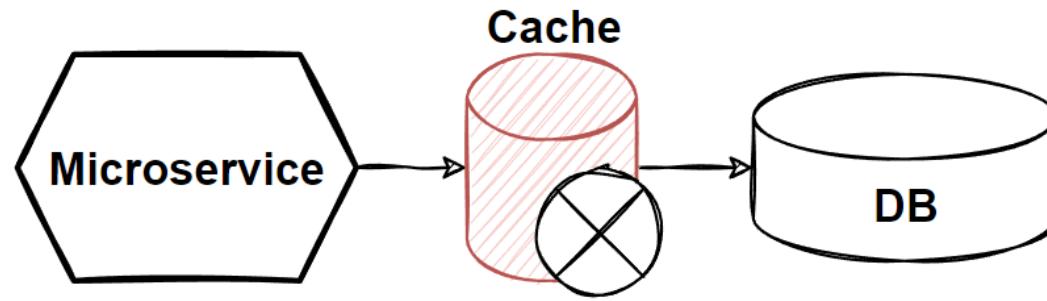
- **Cache hit** occurs when the **requested data** can be **found** in the **cache**.
- **Cache miss** occurs when the **requested data is not** in the **cache** and **must be retrieved** from a slower storage db.
- **Cache hits** are **desirable** because they can improve the performance of a system by **reducing the number of requests**.
- **Cache misses** can have a **negative impact** on performance, because they **require additional time** and resources to retrieve the requested data.
- **The cache hit rate** is a measure of **how often a cache** is able to **fulfill requests** from its own store.
- **High cache hit rate** indicates that the cache is **effective** at storing frequently accessed data.
- **Low cache hit rate** may indicate that the **cache is not large enough**.

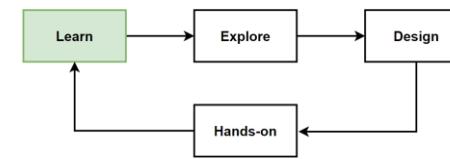




Caching Strategies in Distributed Caching

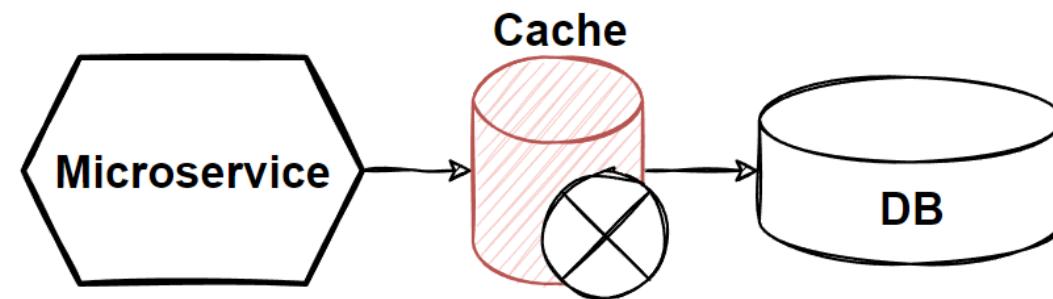
- There are **several caching strategies** that can be used in distributed microservices:
 - Cache Aside
 - Read-Through
 - Write-Through
 - Write-Back, Write-Behind
- **Cache Aside Strategy**
 Client checking the cache for data before making a request to the backend service. When microservices needs to read data from the database, it checks the cache first to determine whether the data is available.
- If the **data is available (a cache hit)**, the cached data is returned. If the **data isn't available (a cache miss)**, the database is queried for the data.
- The client will **retrieve the data from the backend service** and store it in the cache for future requests.
- Data is **lazy loaded into cache** by client application.

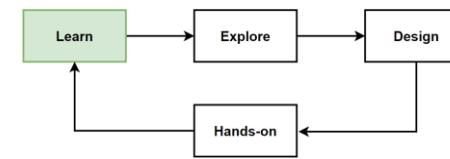




Caching Strategies in Distributed Caching

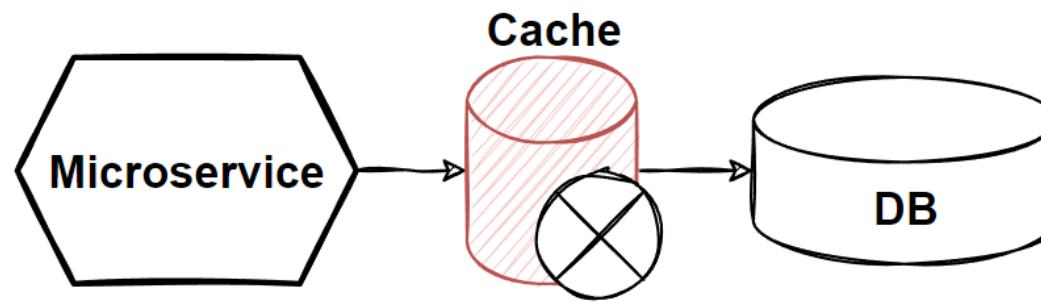
- **Read-Through Strategy**
When there is a cache miss, it loads missing data from the database, populates the cache and returns it to the application.
- When a client **requests data that is not found** in the cache, the cache will **automatically retrieve** the data from the underlying database and **store** it in the **cache** for future requests.
- **Cache-aside strategy**, when a client requests data that is **not found in the cache**, **the client is responsible** for retrieving the data from the database.
- **Read-through cache strategy**, when a client requests data that is **not found in the cache**, **the cache will automatically retrieve** the data from the database.
- **Cache always stays consistent** with the database.

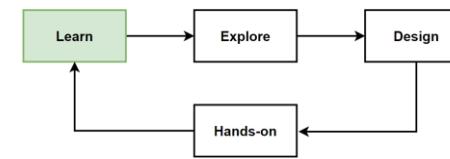




Caching Strategies in Distributed Caching

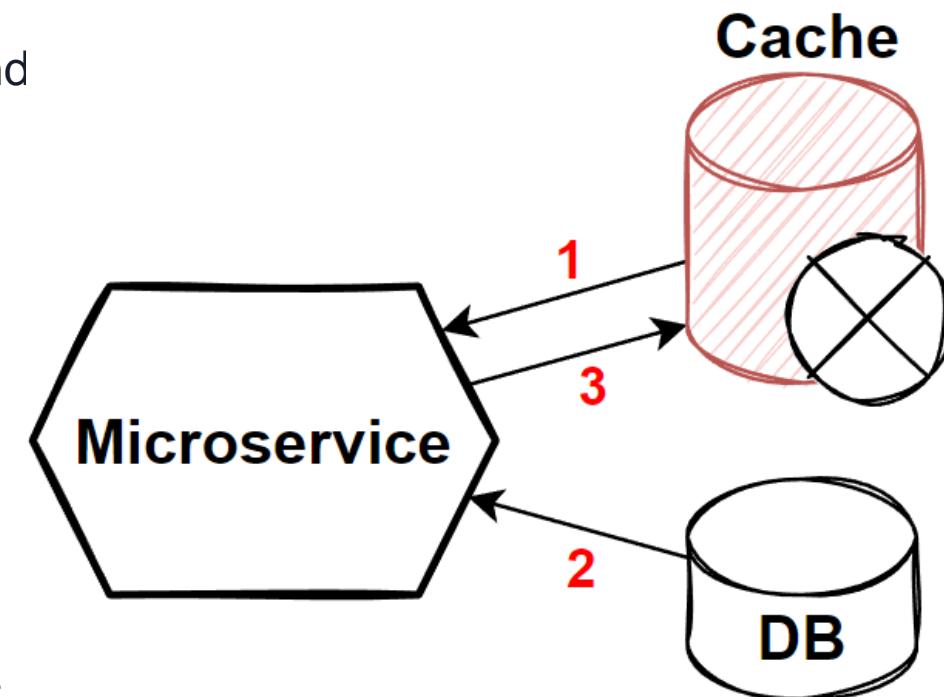
- **Write-Through Strategy**
Update the cache whenever data is written to the backend service. Cache always has the most up-to-date data, but it can also result in a higher number of write operations.
- **Instead of lazy-loading** the data in the cache after a cache miss, the cache is proactively updated immediately following the primary database update.
- Data is first written to the cache and then to the database.
- **Write-Back or Write-Behind Strategy**
Delays updating the cache until a later time. This reduces the number of write operations, but the cache may not have the most up-to-date data.
- **In Write-Through**, the data written to the cache is synchronously updated in the main database.
- **In Write-Back or Write-Behind**, the data written to the cache is asynchronously updated in the main database.

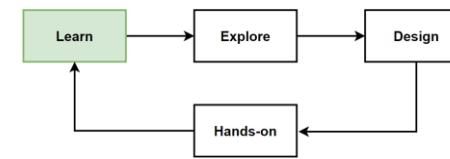




Cache-Aside Pattern for Microservices

- **(1)** When a client needs to access data, it first checks to see if the data is in the cache.
- **(2)** If the data is in the cache, the client retrieves it from the cache and returns it to the caller.
- **(3)** If the data is not in the cache, the client retrieves it from the database, stores it in the cache, and then returns it to the caller.
- Some of caching systems provide **read-through** and **write-through/write-behind** operations. In these systems, **client application retrieves data over by the cache**.
- For not supported Caches, it's the **responsibility the applications use the cache and update the cache** if there is a **cache-miss**.
- **Microservices** good example to implement **Cache-Aside pattern**, it is common to use a **distributed cache** that is **shared across multiple services**.



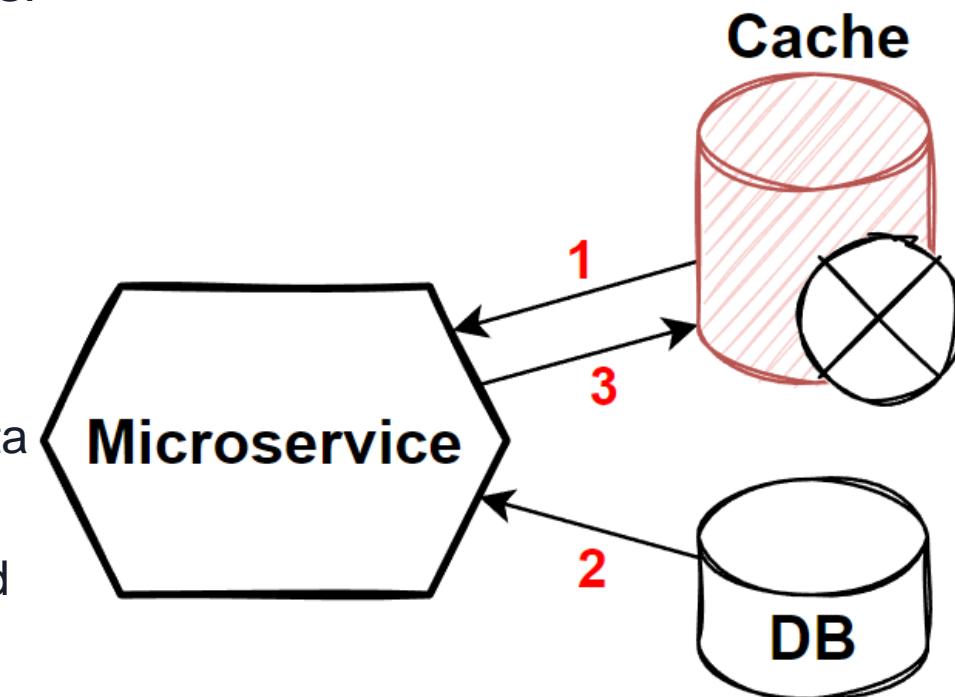


Cache-Aside Pattern for Microservices - 2

- Cache-aside pattern can **improve performance** of a **microservices** architecture, by **reducing the number of expensive database calls**.
- To use the Cache-aside pattern in a microservice, **need to implement a cache layer** in your service.
- Involve using a **Cache library** or framework, such as **Redis** or **Memcached**, or implementing a custom cache solution.

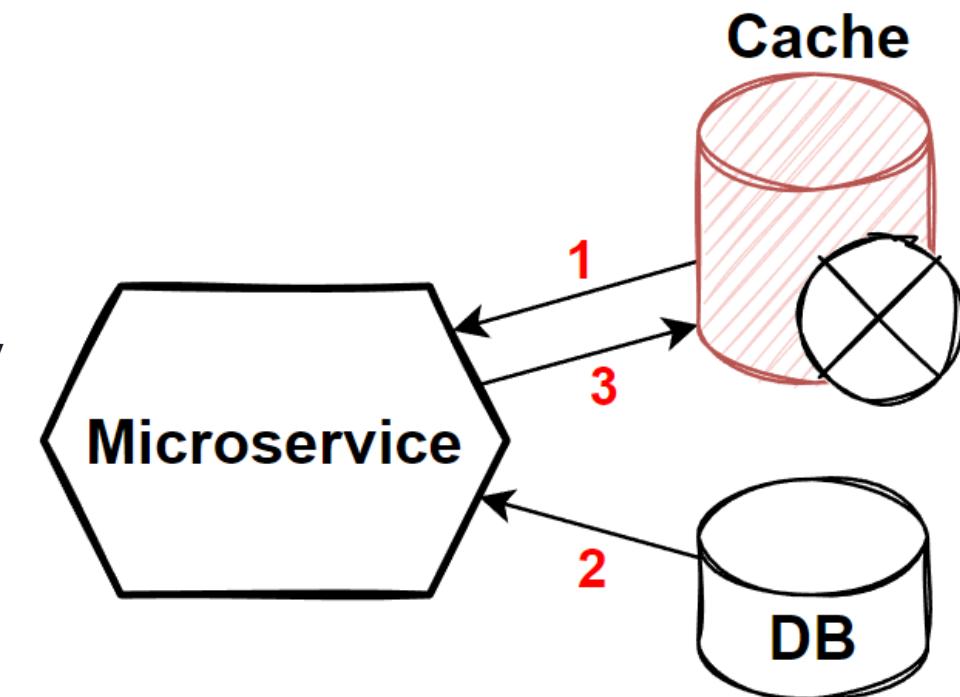
Process:

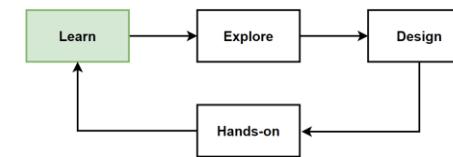
- When a service needs to access data, it first checks to see if the data is in the cache.
- If the data is in the cache, the service retrieves it from the cache and returns it to the caller.
- If the data is not in the cache, the service retrieves it from the database or other data store, stores it in the cache, and then returns it to the caller.



Drawbacks of Cache-Aside Pattern for Microservices

- Cache can introduce **additional complexity** and may not be suitable for all situations.
- The cache may **need to be invalidated** or **refreshed** when data is updated in the database or data store.
- This can **require additional coordination** between the microservices.
- The cache may introduce **additional latency** if it is located remotely from the microservices that are using it.





Best Practices of using Cache in Cloud-native Apps

- Understand Your Data**

Not all data is suitable for caching. Identify the data that is frequently accessed, relatively static, or expensive to compute or retrieve. Get the most value from your caching strategy.

- Use the right Caching Strategy**

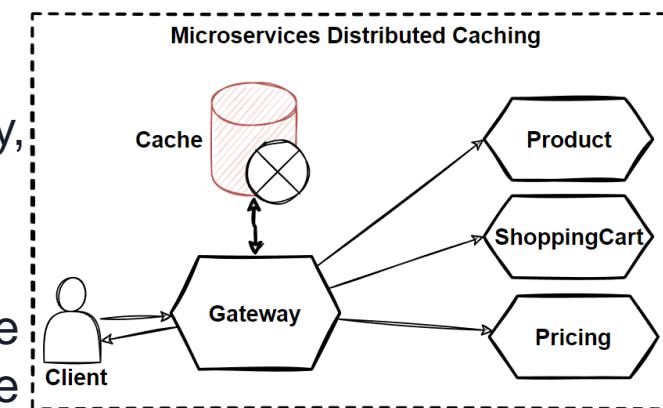
Choose the appropriate caching strategy based on your application's access patterns and consistency requirements. Mostly reads data and requires low latency, consider using the cache-aside or read-through strategy.

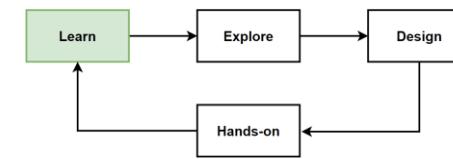
- Set appropriate Cache Expiration - Cache Invalidation**

Determine the appropriate expiration time for cache entries based on how often the data changes. Use Time-To-Live (TTL) or other cache eviction strategies to remove stale data from the cache automatically.

- Use Distributed Caching for Scalability and High Availability**

Using a distributed cache can help ensure that cached data is available to all instances, even if one node goes down. Improves the overall resilience and scalability.





Best Practices of using Cache in Cloud-native Apps - 2

- Be careful of Data Consistency**

Caching can introduce data consistency challenges. Ensure that you have a strategy to handle cache invalidation and maintain consistency between the cache and the underlying data store. Maintain strong consistency between cache and database.

- Monitor and Optimize Cache Performance**

Monitor cache hit and miss rates, latency, and other performance metrics to ensure that your caching strategy is effective. Monitor cache hit rate, miss rate, eviction rate, and memory usage. Set up alerts for abnormal metrics.

- Plan for Cache Failures**

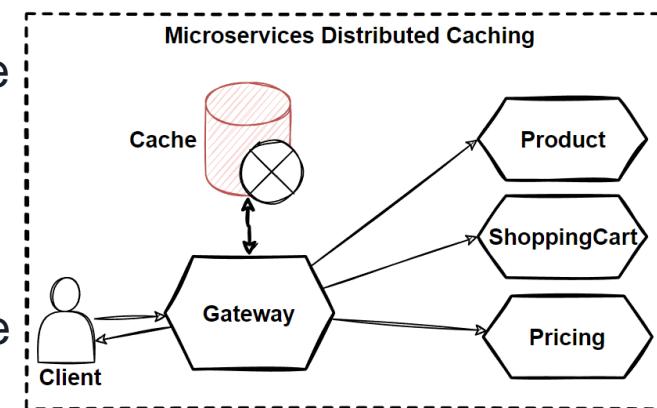
Implement fallback strategies: retrieving data directly from the data store, to ensure that your application remains functional in case of cache failures.

- Test your Caching Strategy**

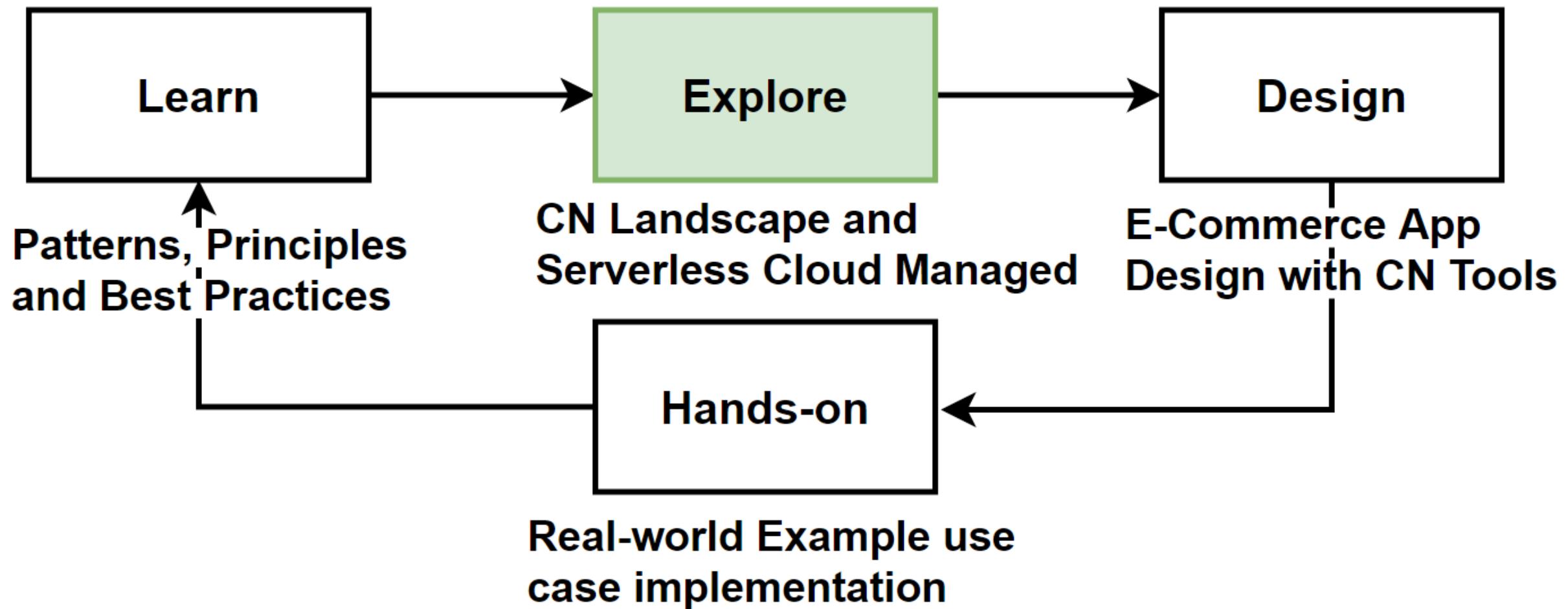
Test under various workloads and scenarios to ensure that it meets your performance and consistency requirements that help you identify potential issues and optimize your caching strategy.

- Keep it Simple**

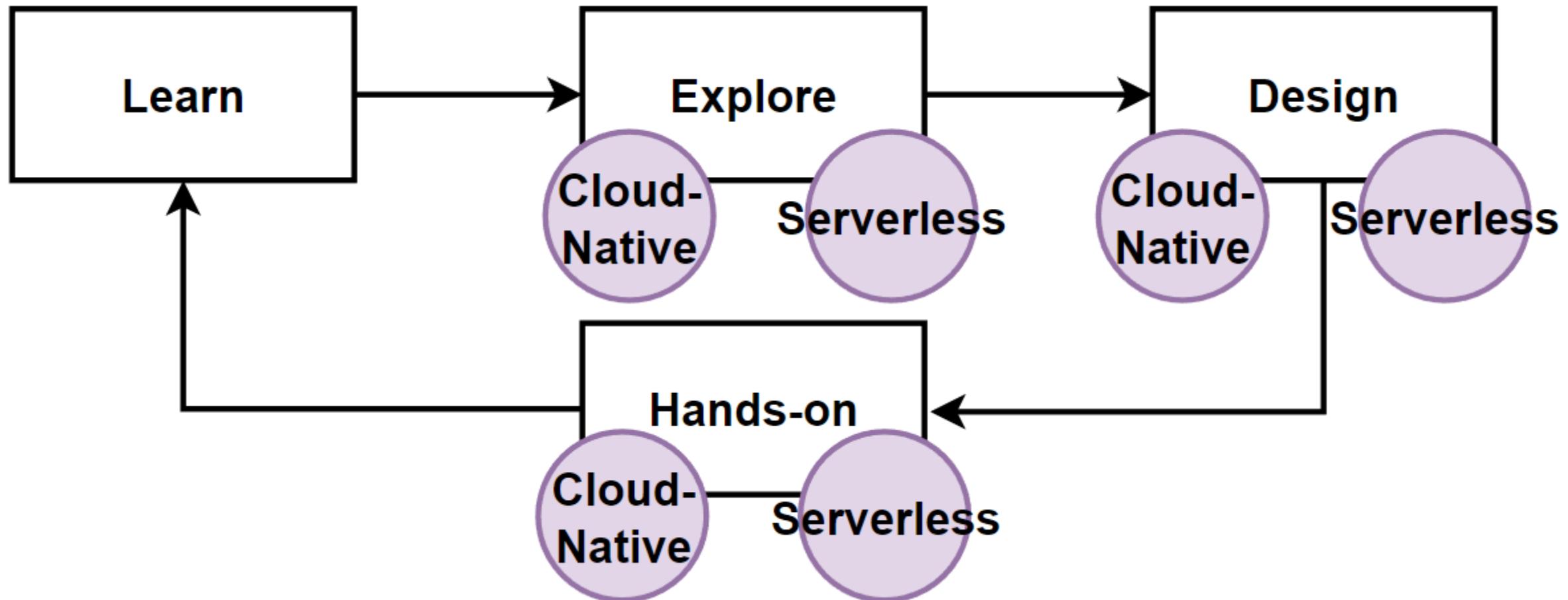
Caching can add complexity to application, so keep your caching strategy and implementation as simple as possible. Start with basic caching and iterate.



Explore: Cloud Managed and Serverless Microservices Frameworks

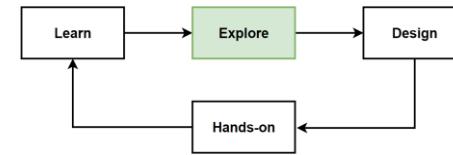


Way of Learning – Cloud-Native & Serverless Cloud Managed Tool



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Explore: Cloud-Native Distributed Caches

- **Horizontally Scalable Distributed Caches** that designed to work seamlessly with cloud-native architectures and **scale out by adding more nodes** to the system.

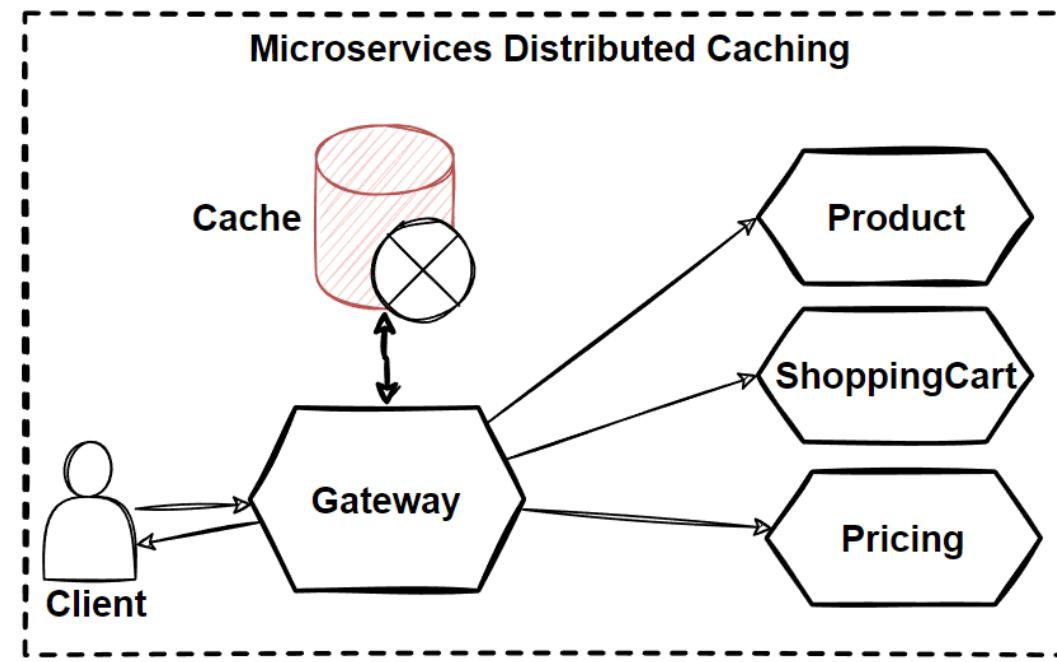
Distributed Caches

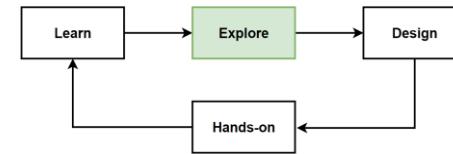
- Redis
- Memcached
- Hazelcast
- TiKV
- etcd

Cloud Serverless Caching Services

- Amazon ElastiCache
- Azure Cache for Redis
- Google Cloud Memorystore
- Upstash Redis

Goto -> <https://landscape.cncf.io/>



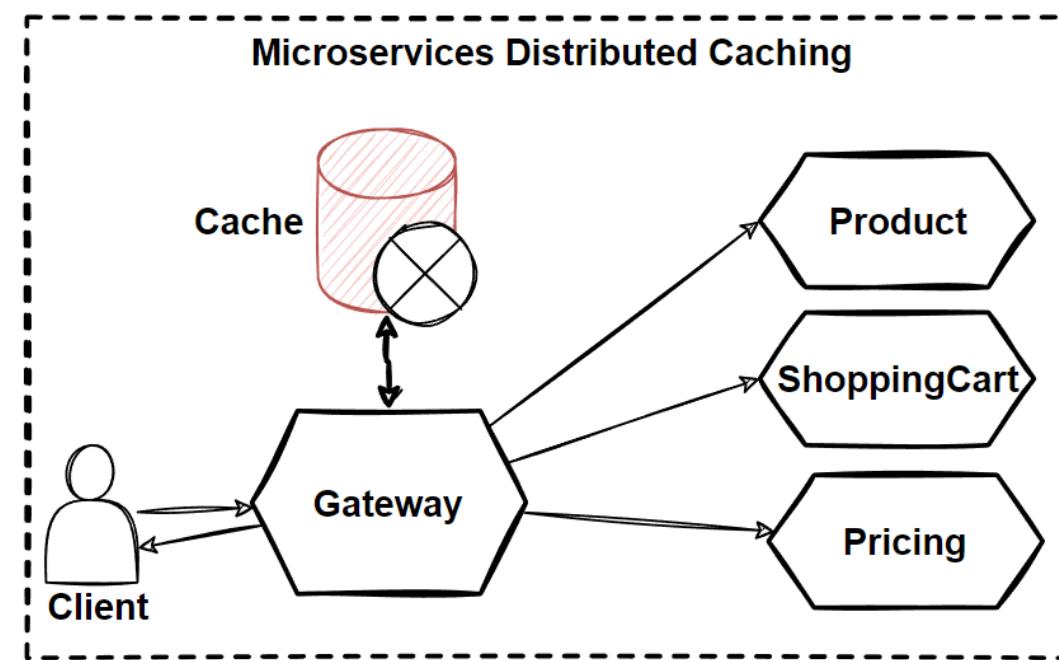


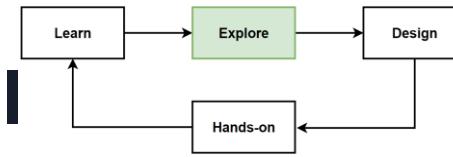
Explore: Cloud Serverless Distributed Caches

- **Managed serverless caching services** are provided by major cloud providers to **simplify** the process of **setting up, managing, and scaling** caching solutions for cloud-native apps.
- Fully managed and **automatically scale to handle the workload, reducing the operational overhead** of managing caching infrastructure.

Cloud Serverless Caching Services

- [Amazon ElastiCache](#)
- [Azure Cache for Redis](#)
- [Google Cloud Memorystore](#)
- [Upstash Redis](#)





Explore: Upstash Redis: Serverless fully managed global distributed Redis-compatible caching

- **Upstash Redis** is a serverless, fully managed, globally distributed Redis-compatible caching service.
- It provides a **low-latency, high-performance** caching solution for cloud-native applications.
- It offers features like **on-demand scaling, pay-as-you-go pricing, multi-cloud support**, and easy integration with other cloud services.

▪ **Serverless Architecture**

Designed as a serverless service, meaning it automatically scales with the workload and only charges for the resources used, reducing operational complexity and costs.



▪ **Global Distribution**

Multi-cloud and global distribution, allowing you to deploy your cache closer to your users and improve the performance of your applications.

▪ **High Availability**

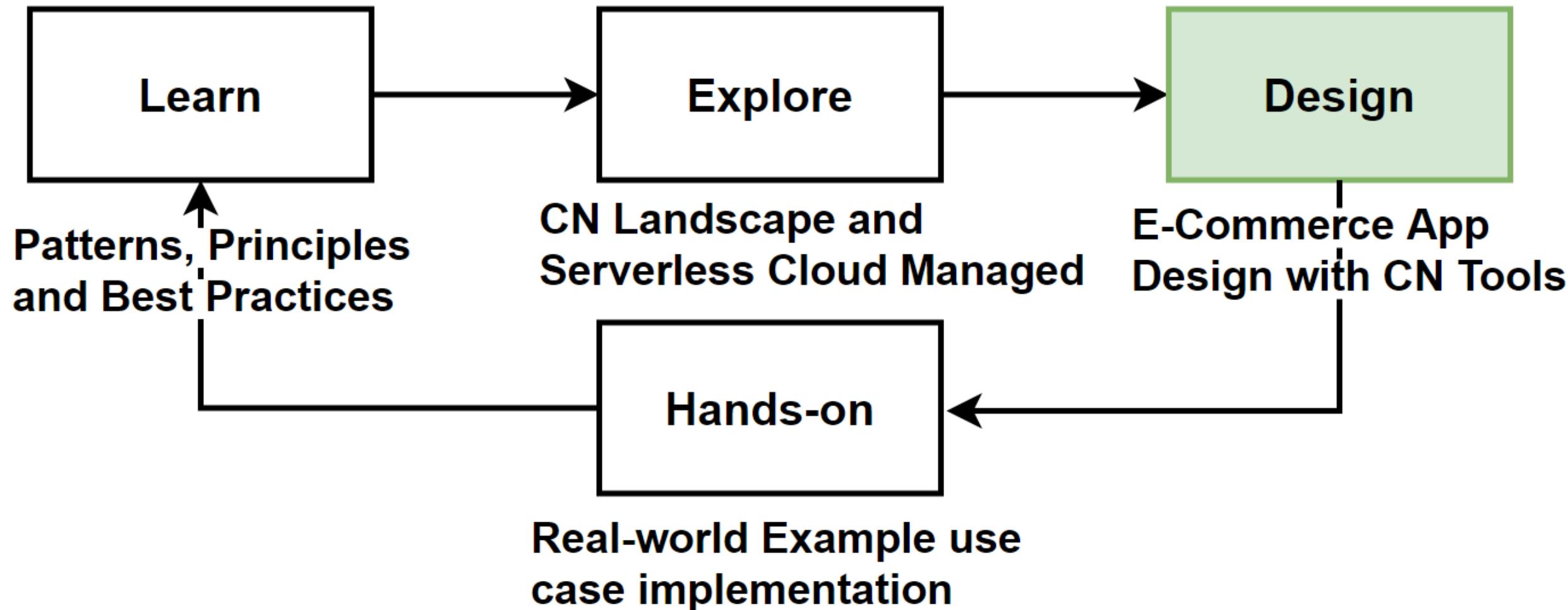
Ensures high availability through data replication and automatic failover mechanisms.

▪ **Easy Integration**

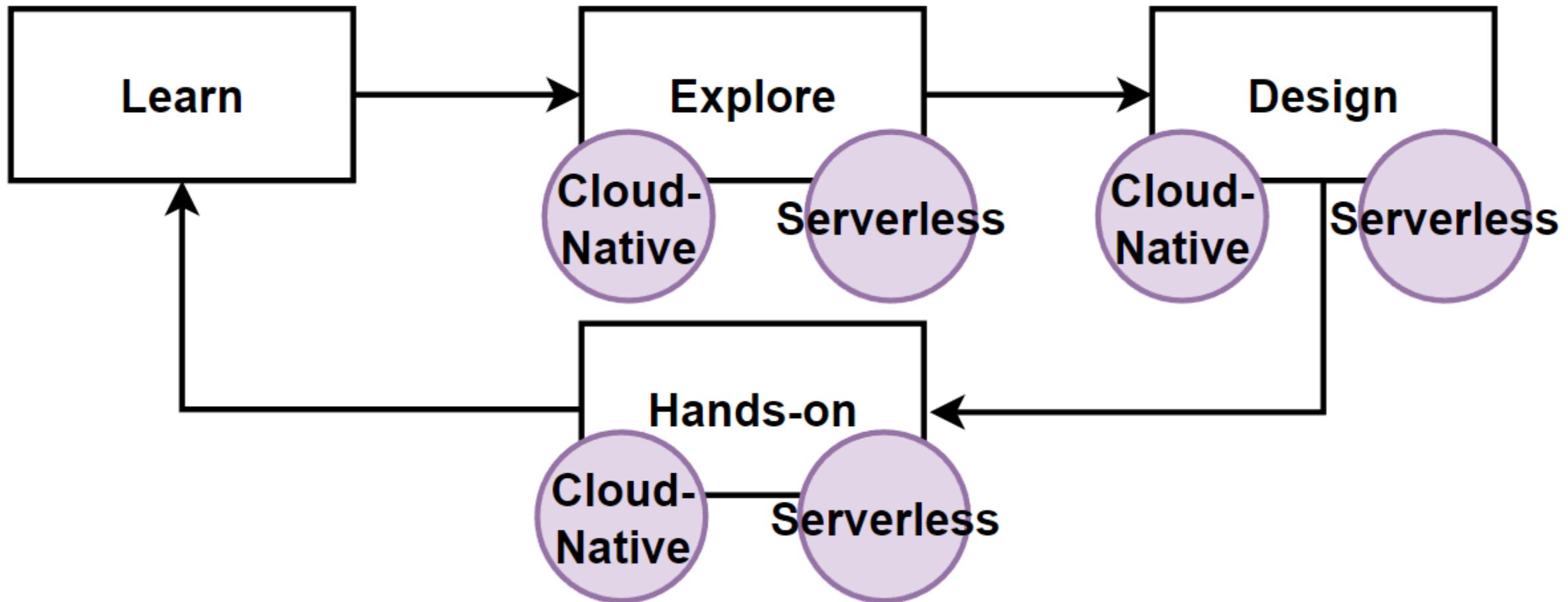
Seamless integration with popular cloud-native platforms and services, making it easy to incorporate caching into your application architecture.

- Goto -> <https://upstash.com/>

Way of Learning – The Course Flow



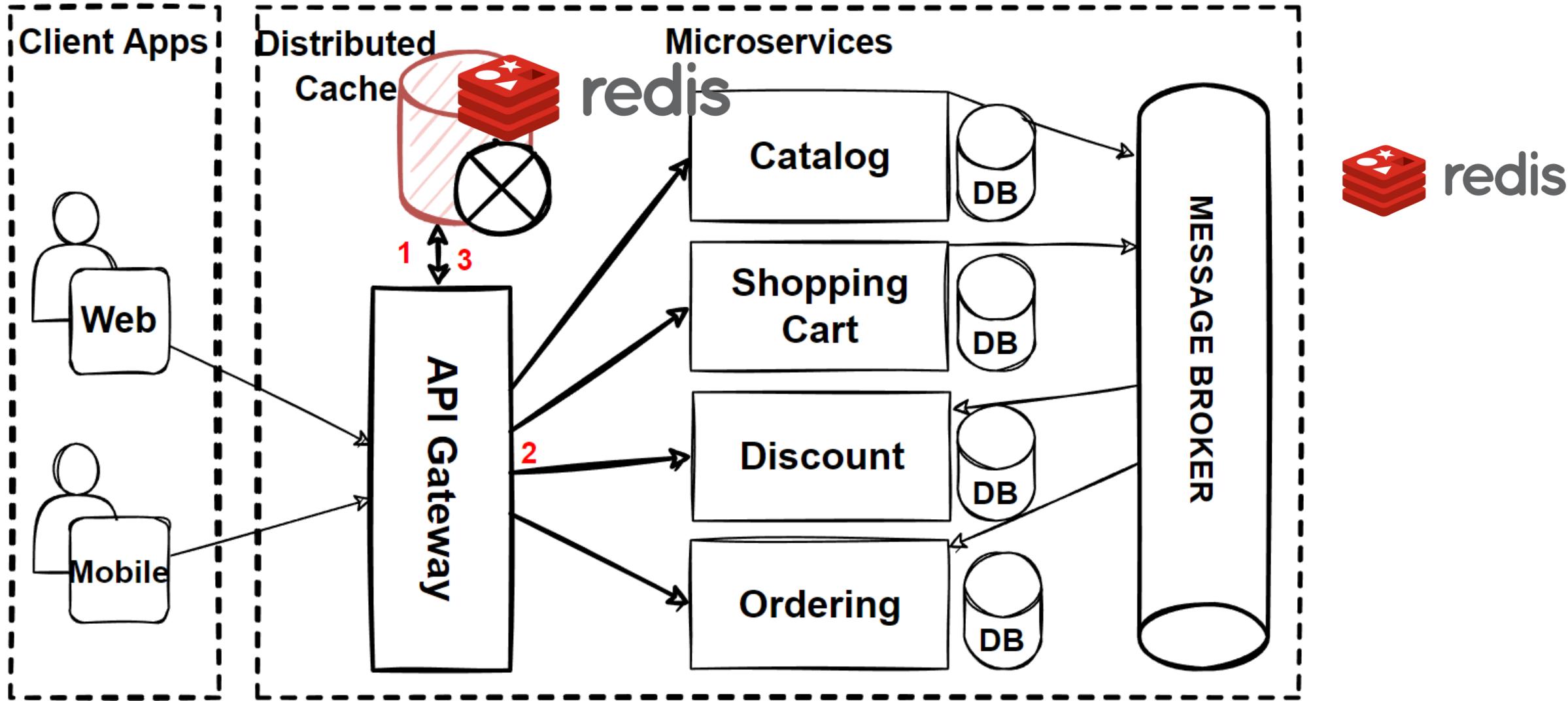
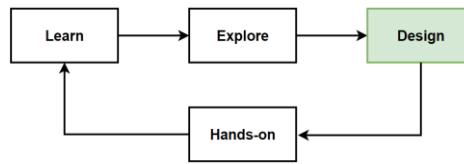
Way of Learning – Cloud-Native & Serverless Cloud Managed

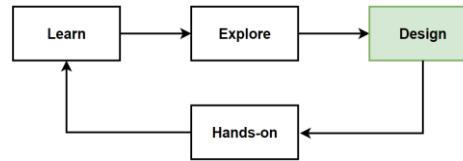


Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

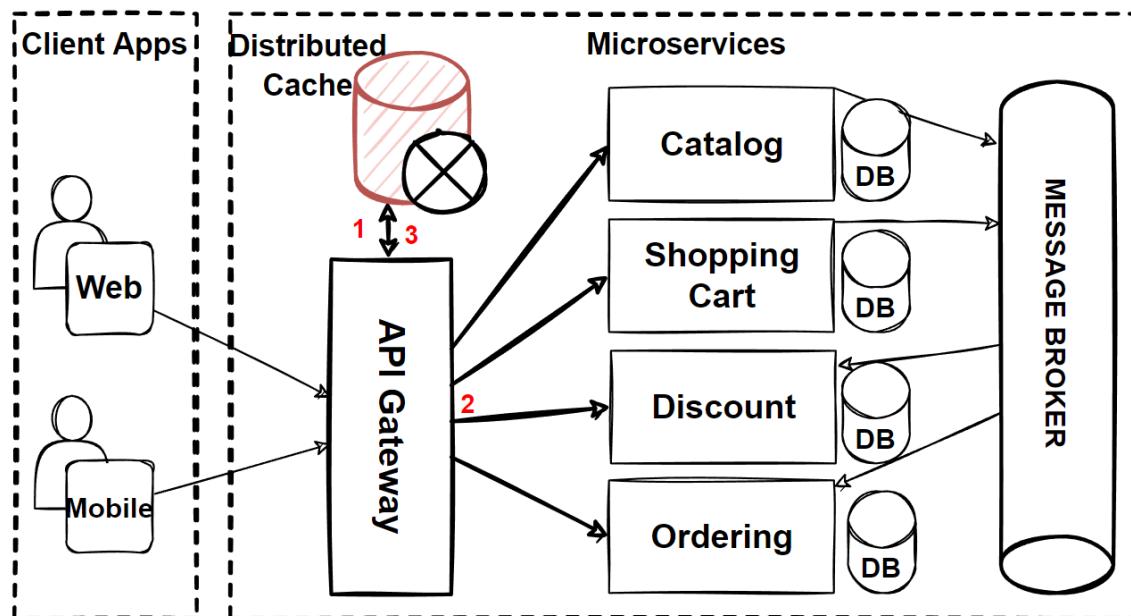
Microservices Distributed Caching with Cache-Aside Pattern

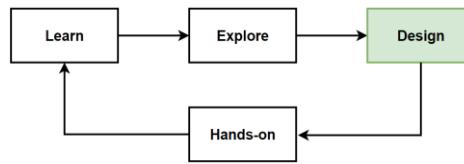




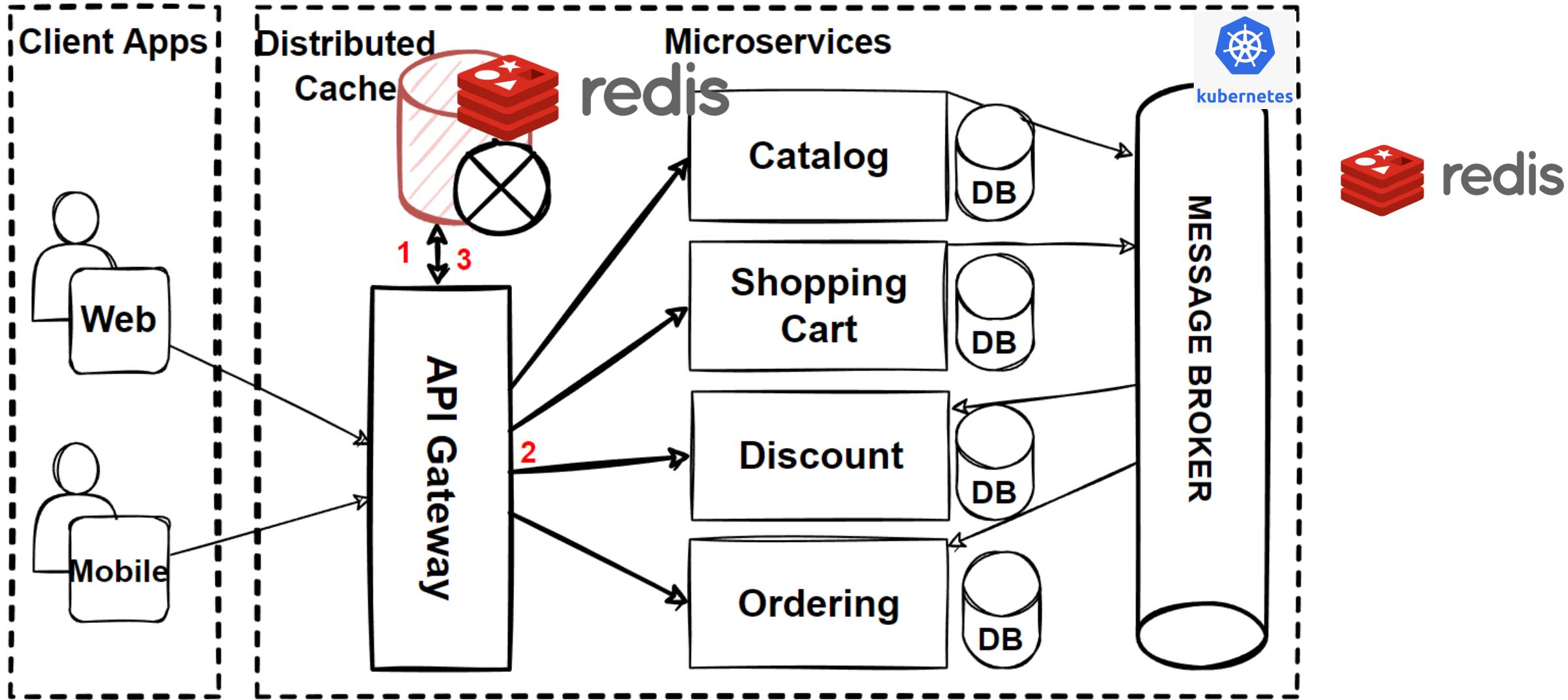
Microservices Distributed Caching apply Cache-Aside Pattern on API Gateway

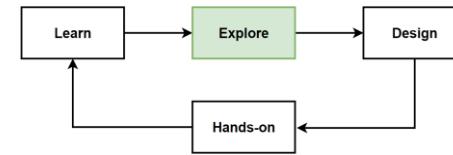
1. User makes a request to an API gateway to retrieve some data.
2. API gateway determines which microservice is responsible for handling the request and receives the request and checks to see if the data is in the cache.
3. If the data is in the cache, API gateway retrieves it from the cache and returns it to the client.
4. If the data is not in the cache, API gateway forward responsible microservice to retrieves it from the database, and then returns it to the API gateway.
5. The API gateway receives the data from the microservice and stores it in the cache and returns it to the user.





Microservices Distributed Caching with Cache-Aside Pattern - Redis



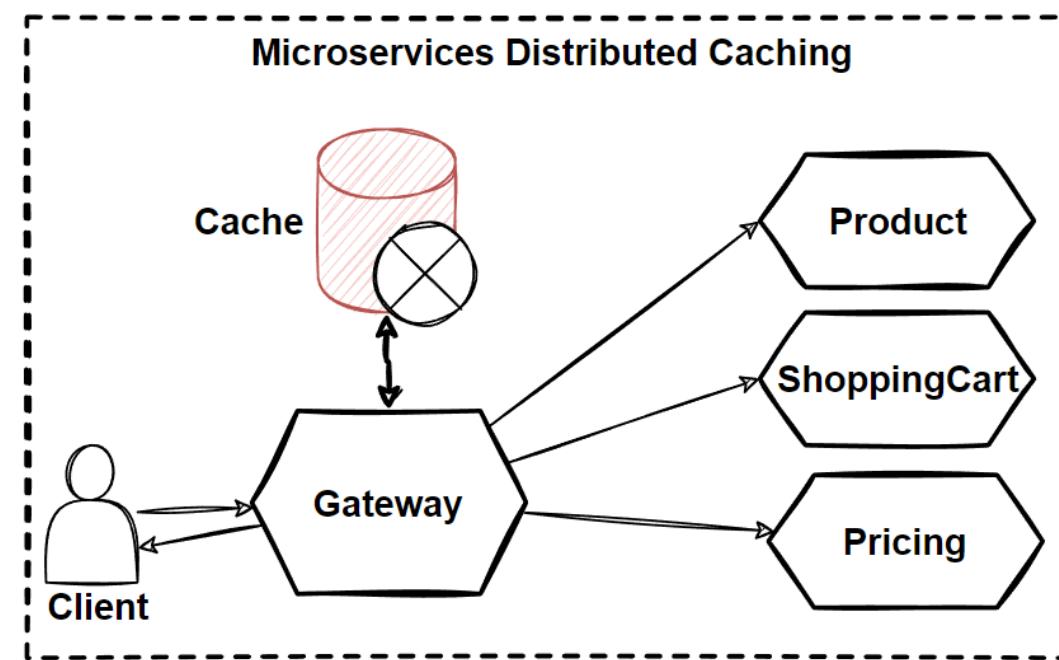


Explore: Cloud Serverless Distributed Caches

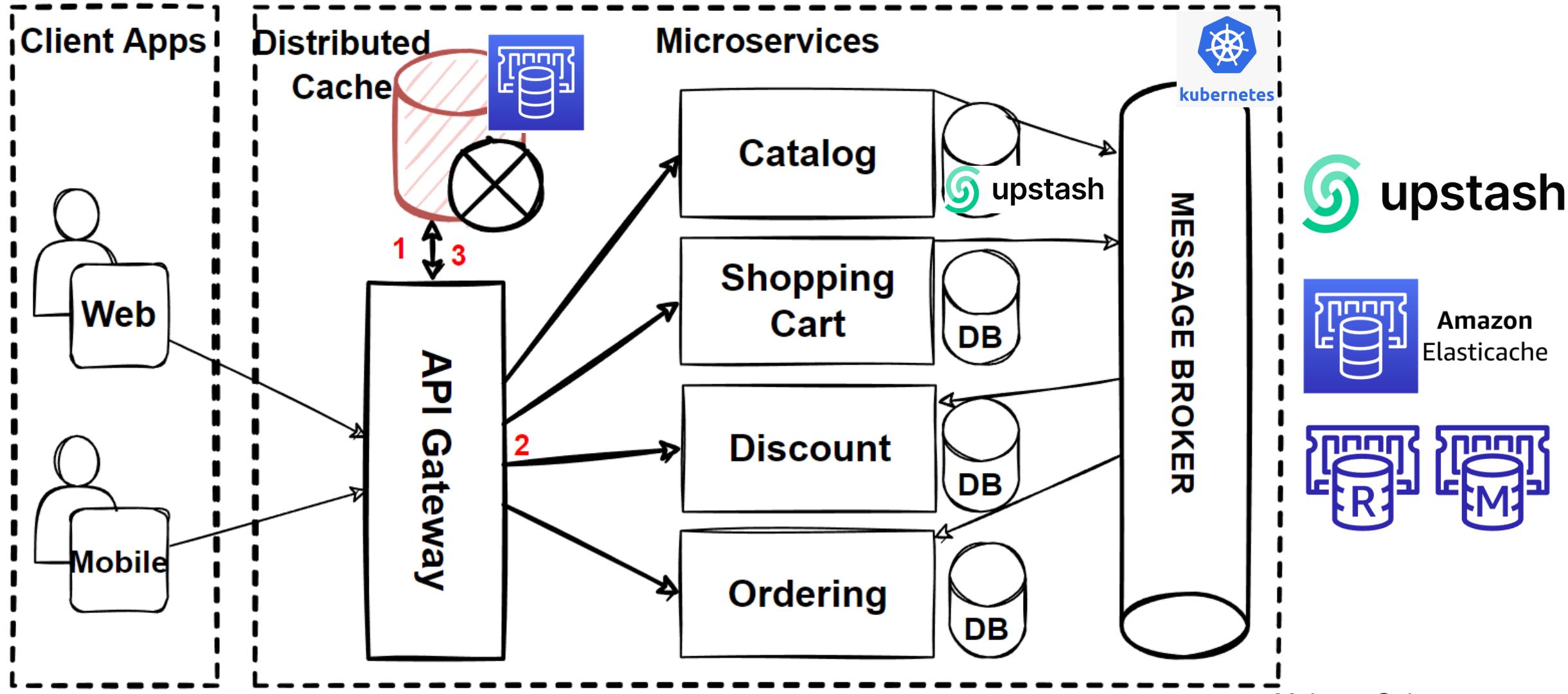
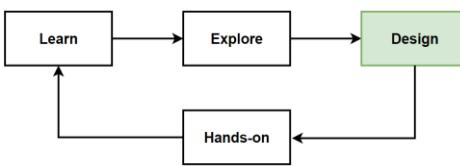
- **Managed serverless caching services** are provided by major cloud providers to **simplify** the process of **setting up, managing, and scaling** caching solutions for cloud-native apps.
- Fully managed and **automatically scale to handle the workload, reducing the operational overhead** of managing caching infrastructure.

Cloud Serverless Caching Services

- [Amazon ElastiCache](#)
- [Azure Cache for Redis](#)
- [Google Cloud Memorystore](#)
- [Upstash Redis](#)

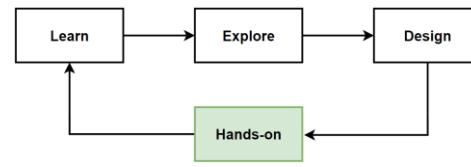


Microservices Distributed Caching with Cache-Aside Pattern – Amazon ElastiCache



Hands-on: Deploy Redis Cache on a Kubernetes Cluster with Minikube

Leveraging Horizontally Scalable Distributed Cloud-Native Caches
Using Helm Charts from Bitnami and deploy Redis into Minikube



Redis: A Distributed Cache

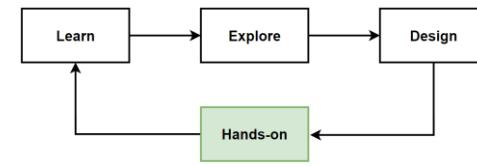
- **Redis** is a widely-used, open-source, in-memory data structure store that can be used as a cache, message broker, or database.
- Supports various data structures such as strings, hashes, lists, sets, and more.
- High performance, simplicity, and versatility, making it a popular choice for caching in cloud-native environments.
- Distributed caching layer that optimizes the performance and scalability of the system.



▪ **Redis in Kubernetes**

Redis is easily deploy and manage in Kubernetes cluster. Helm charts available that can simplify the process of deploying a Redis Cluster in a Kubernetes environment.

- Once deployed, microservices communicate with Redis to store and retrieve data from the cache.
- As the load increases, you can easily scale your Redis Cluster by adding more nodes, and Kubernetes will ensure that the new nodes are integrated into the cluster.

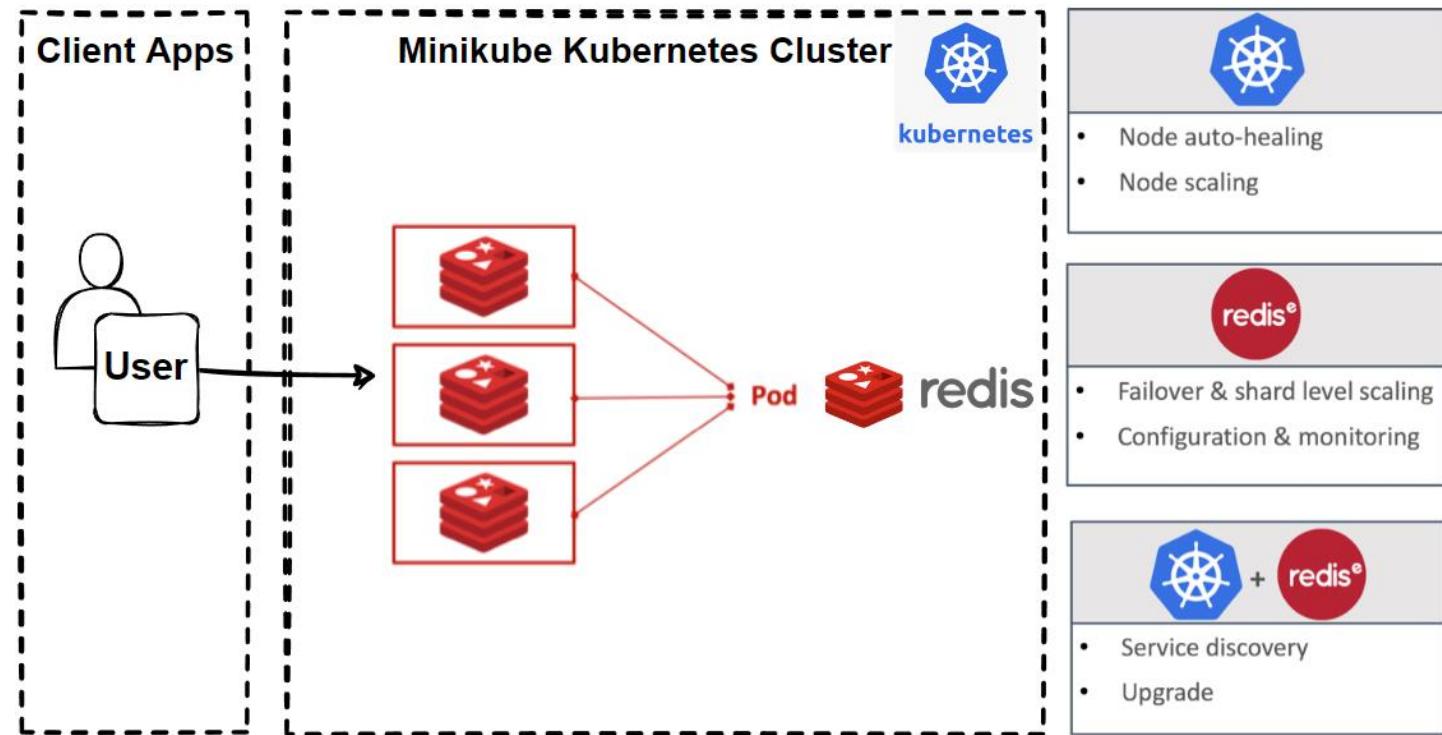


Hands-on: Deploy Redis in a Single Kubernetes Cluster with Minikube – Task List

- Step 1. Start Kubernetes - minikube start
- Step 2. Start Redis - Deploy with Bitnami Helm Charts
- Step 3. Use the built-in Redis client
- Step 4. Simulate node failure and node scales
- **Step 5. Stop the cluster**

Redis Artifact Hub:

<https://artifacthub.io/packages/helm/bitnami/redis>

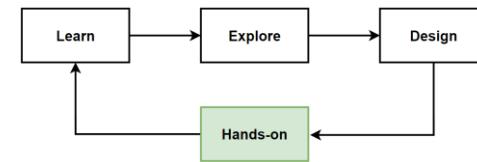


Scale Up Redis:

`helm upgrade my-redis bitnami/redis --set replica.replicaCount=4`

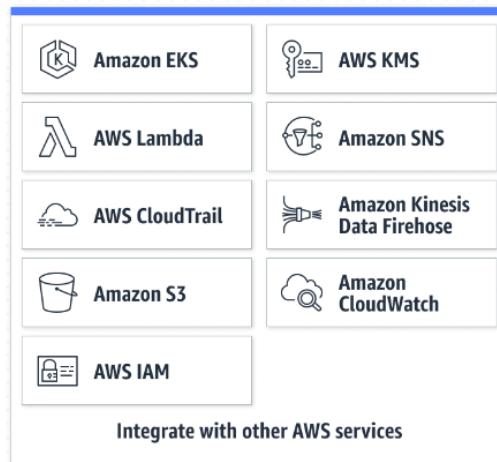
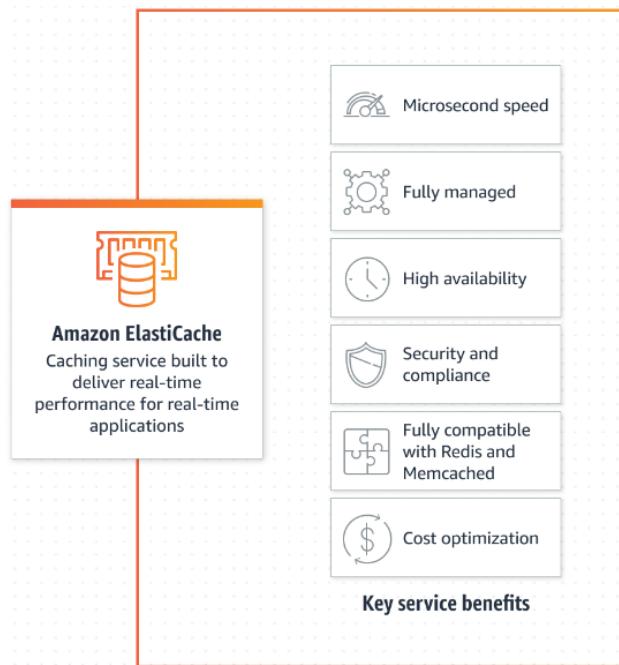
Hands-on: Create Cache Cluster with Amazon ElastiCache for Redis

Create and Manage Redis Cache Cluster on Cloud using Amazon ElastiCache

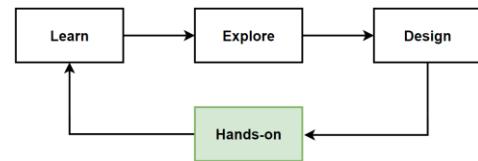


What Is Amazon ElastiCache ?

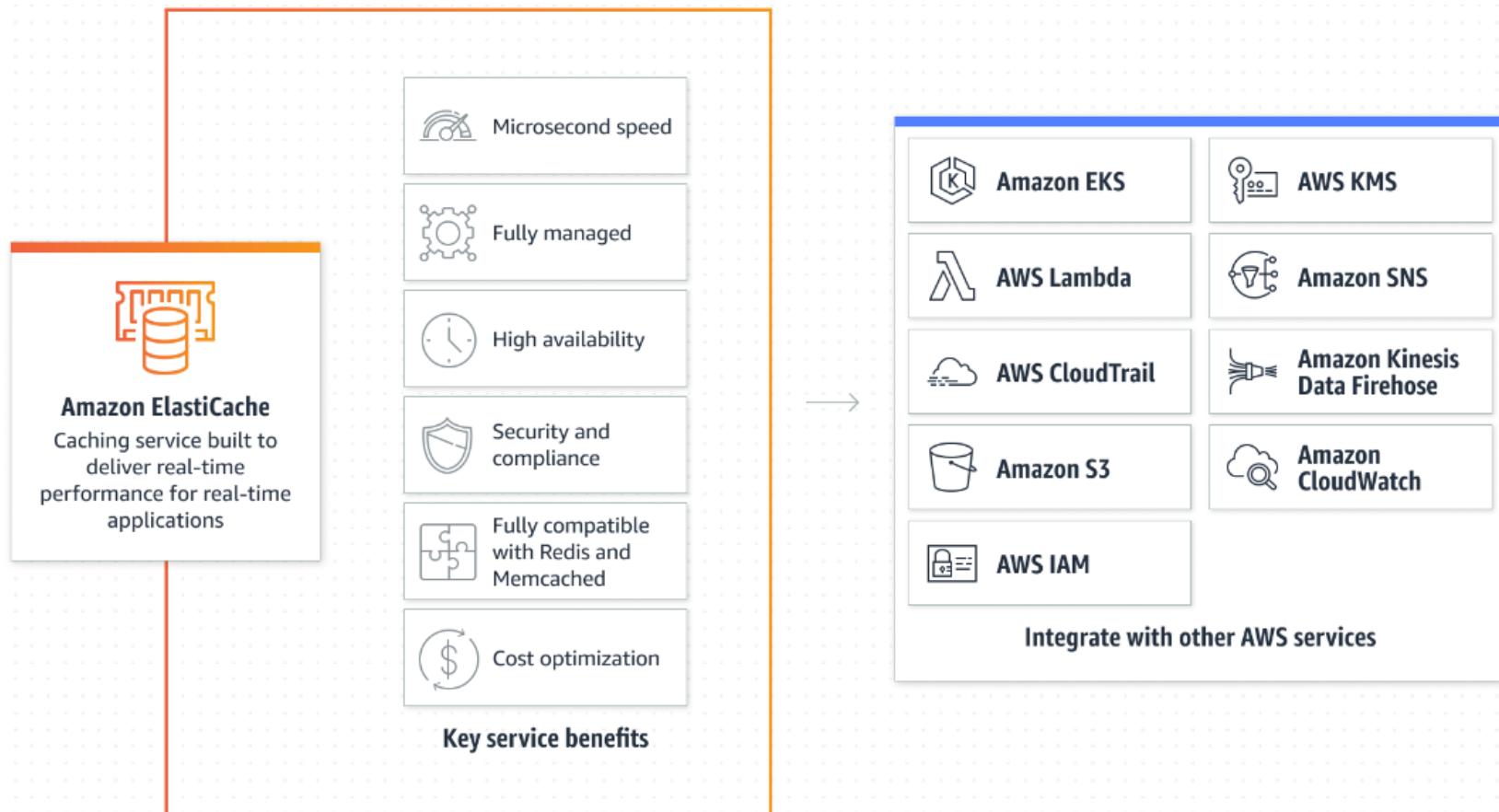
- **Amazon ElastiCache** is a fully managed in-memory data store and cache service by AWS.
- Improves the **performance** of web applications by retrieving information from managed in-memory caches.
- ElastiCache **supports two open-source** in-memory caching engines: **Redis** and **Memcached**.
- **ElastiCache for Redis** is fully compatible with open-source Redis, and includes additional enhancements for durability, availability, and scalability.
- **Redis** is commonly used as a **database, cache, message broker, and queue**.
- **ElastiCache** is a good choice for workloads that require a **sub-millisecond latency** for data access. Enhance performance of the read-heavy and compute-intensive workloads.



<https://aws.amazon.com/elasticsearch/>



Getting Started with Amazon ElastiCache for Redis



Links:

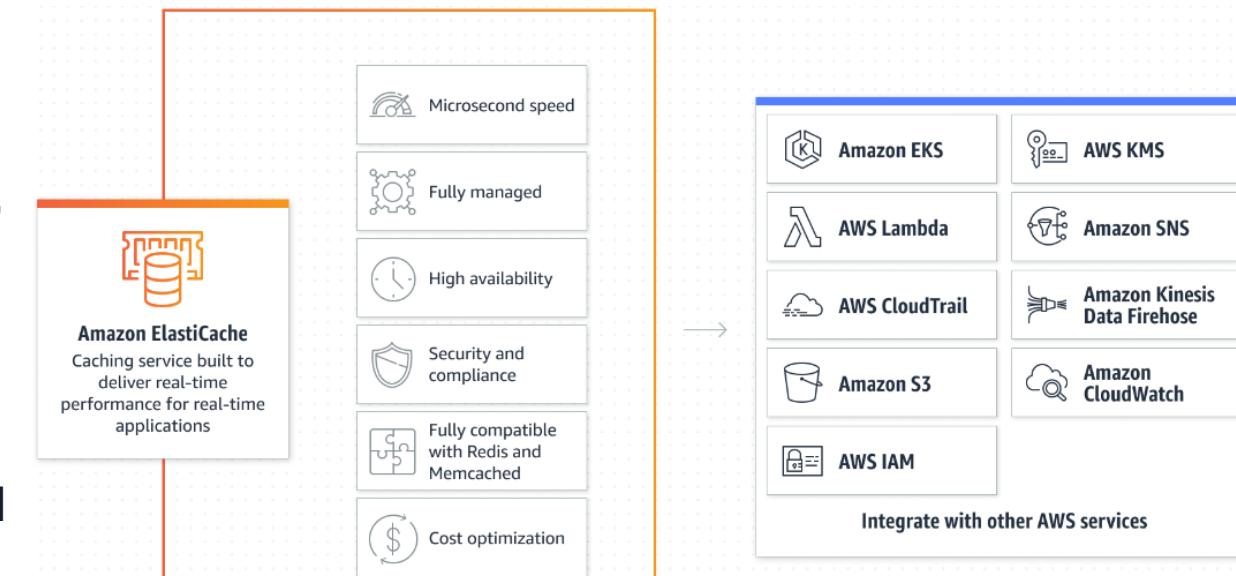
- <https://aws.amazon.com/elasticsearch/>
- <https://aws.amazon.com/elasticsearch/redis/?nc=sn&loc=2&dn=1>
- <https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/WhatIs.html>

<https://aws.amazon.com/elasticache/>

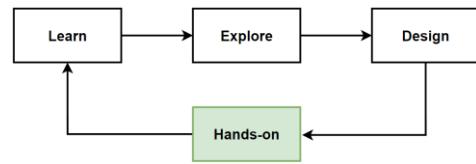
Create Redis Cache Cluster with Amazon ElastiCache for Redis - Task List

- Step 1: Create a subnet group
- Step 2: Create a cluster
- Step 3: Authorize access to the cluster
- Step 4: Connect to the cluster's node - connect to VPC, download redis-cli and run redis commands
- Step 5: Deleting a cluster

- These steps will Create VPC, Create Subnet group and IPs, create EC2 instances, Authorize and access EC2..
- These service **can incur extra charge** on your AWS account.
- **Show how to create** but not activate Redis cluster on AWS Cloud.



<https://aws.amazon.com/elasticsearch/>



After Created Redis Cluster

Shards and nodes Metrics Logs Network and security Maintenance and backups Service updates

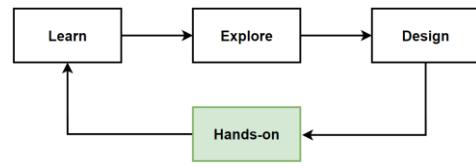
Shards and nodes (1)

Find shards and nodes

<input type="checkbox"/>	<input checked="" type="checkbox"/> Name	Type	Nodes per shard	Slots/keyspaces	Zone	Status
<input type="checkbox"/>	<input checked="" type="checkbox"/> demo-redis-0001	Shard	1	0-16383	-	<input checked="" type="checkbox"/> Available
<input type="checkbox"/>	<input type="checkbox"/> demo-redis-0001-001	Node	-	-	us-east-2b	<input checked="" type="checkbox"/> Available

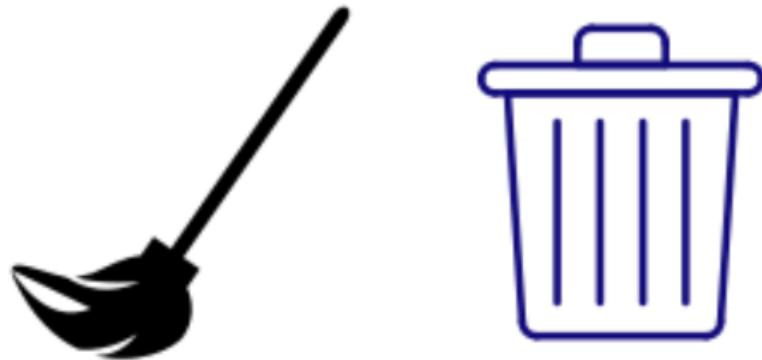
Connect to Cluster Node

<https://docs.aws.amazon.com/AmazonElastiCache/latest/red-ug/GettingStarted.ConnectToCacheNode.html>



Clean up Resources

- Delete AWS Resources that we create during the section.



Cloud-Native Pillar5: Backing Services - Message Brokers (Event-driven communication)

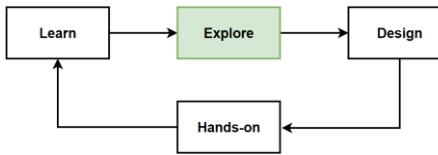
Event-driven Async Communication for K8s and Serverless Message Brokers

What are Cloud-Native Backing Services for Message Brokers ?

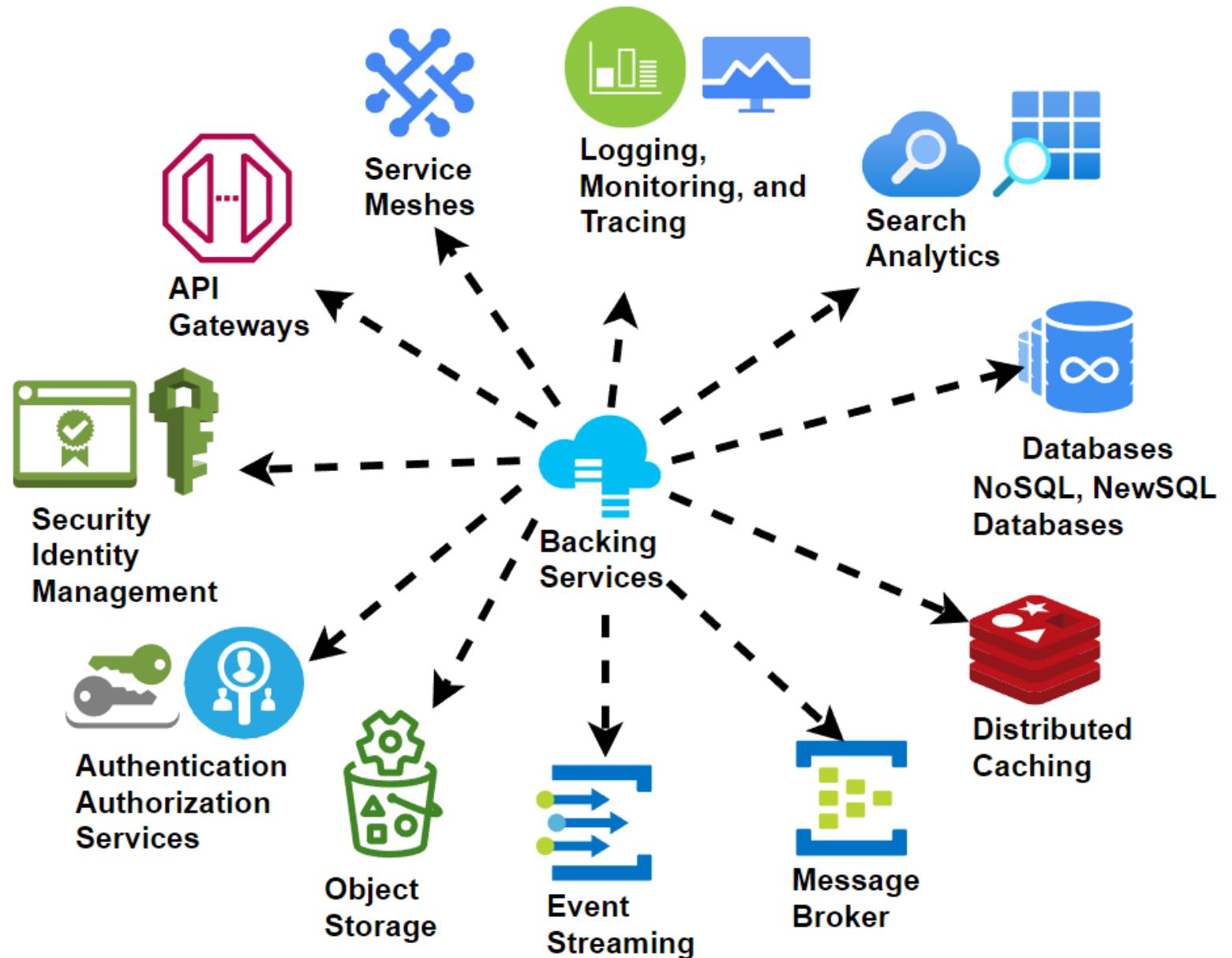
Which Message Brokers should select for Backing Services for Cloud-Native ?

How microservices use Message Brokers in Cloud-Native environments ?

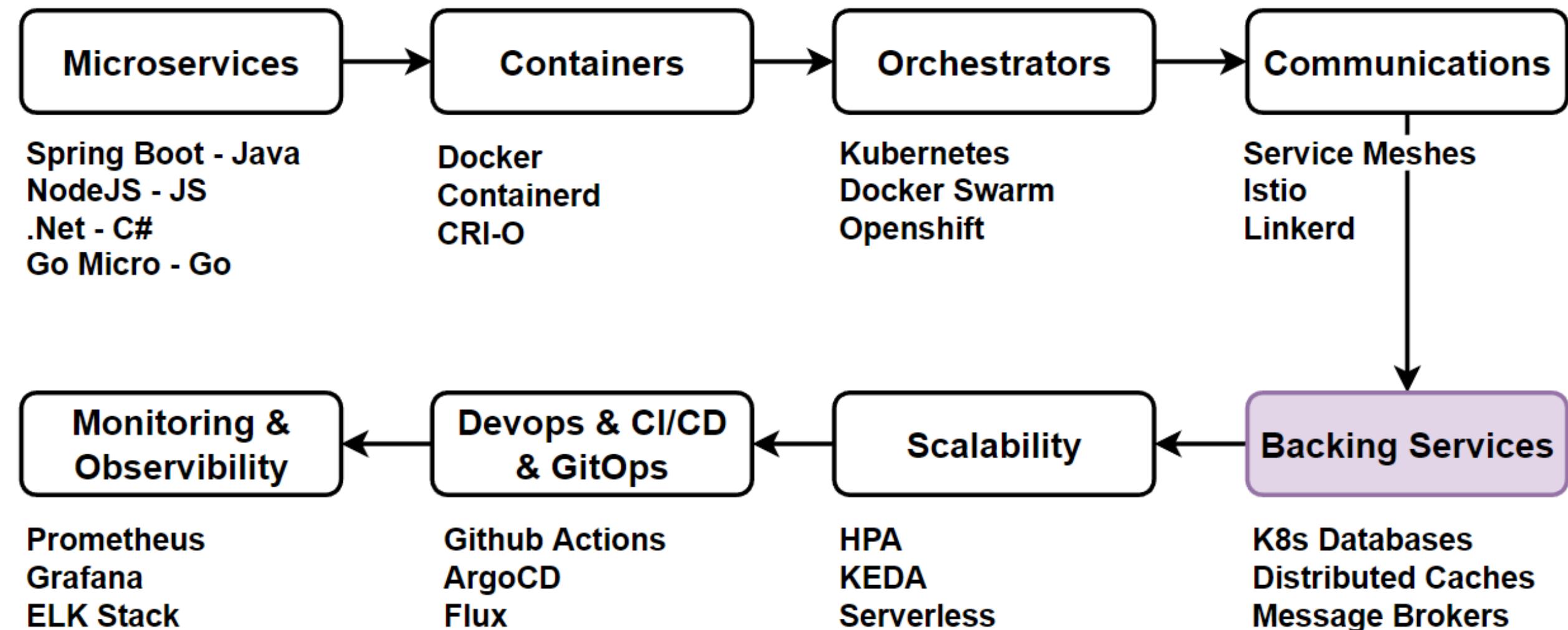
What are patterns & best practices of using Message Brokers in Cloud-native ?



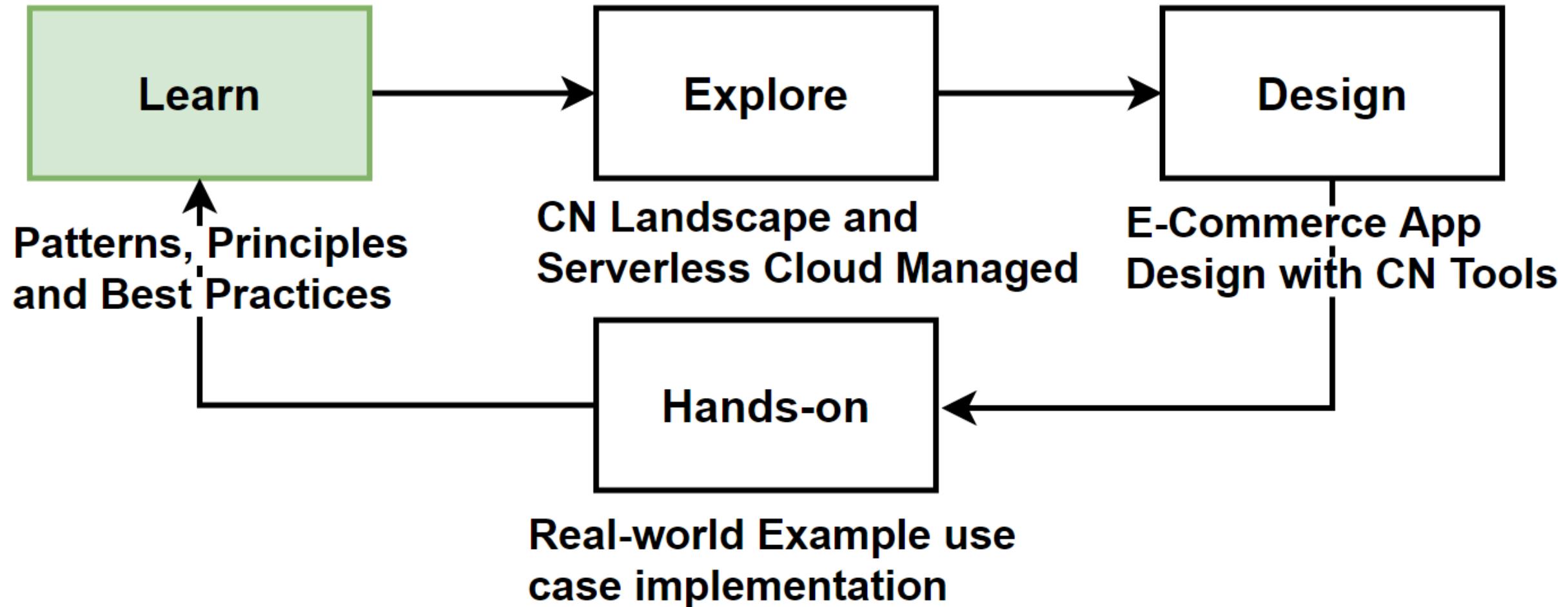
Backing Services for Cloud-Native Microservices

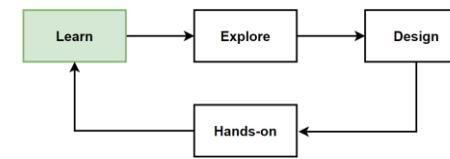


Cloud-Native Pillars Map – The Course Section Map



Way of Learning – The Course Flow

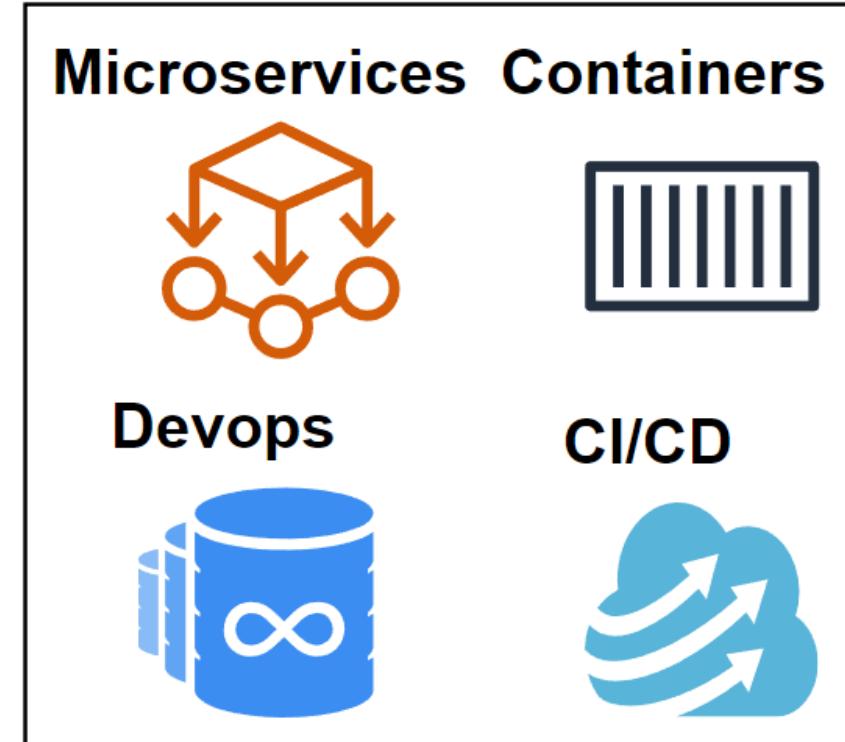


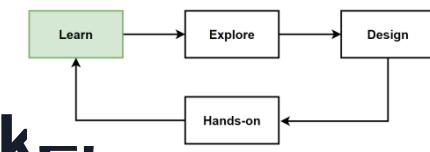


Backing Services - Message Brokers

- Cloud-Native Backing Services – Message Brokers:
- Microservices Asynchronous Message-Based Communication Patterns
- Fan-Out Publish/Subscribe Messaging Pattern
- Event-Driven Microservices Architectures

- Explore - Backing Services for Message Brokers.
 - K8s and Serverless Message Brokers
- Hands-on - Backing Services for Message Brokers.
 - K8s and Serverless Message Brokers





Cloud-native Trial Map – Backing Services – Message Broker

Streaming and **messaging** play a critical role in cloud-native architectures.

Streaming

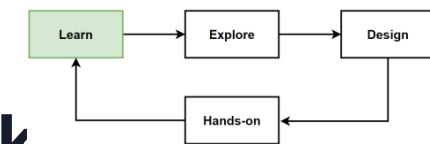
- Data streaming, where a continuous flow of data is processed and often analyzed in real time.
- Use cases like real-time analytics, anomaly detection, and live leaderboards, among others.
- Apache Kafka, Amazon Kinesis, or Google Pub/Sub to manage these streams of data.
- Handle vast amounts of data in near-real time, necessary for microservices-based apps that require high-throughput and low-latency processing.

Messaging

- Asynchronous communication between services.
- Decoupling between microservices, allowing them to interact without being directly connected or aware of each other.



<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>



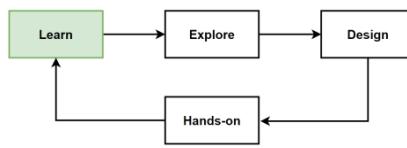
Cloud-native Trial Map – Backing Services – Message Broker

Messaging

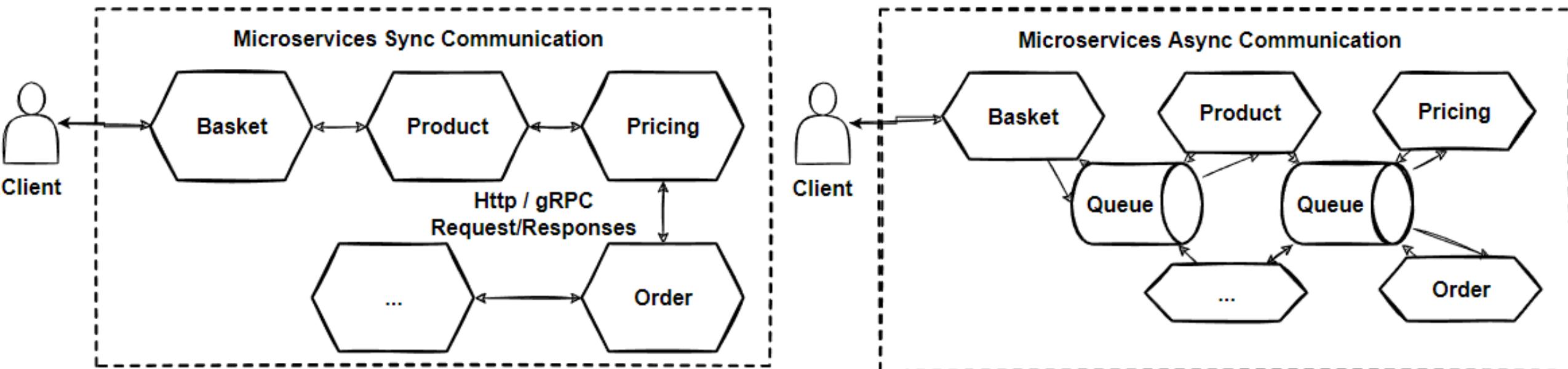
- Messaging systems are often categorized into two types: message queues and publish/subscribe (pub/sub) systems.
- **Message queues:** messages are stored on the queue until they are processed and deleted by a receiver.
- **Pub/sub systems:** messages are sent to all subscribers, or to those that have subscribed to specific topics.
- Publish messages that are then consumed by other services, without the services needing to be aware of each other's existence.
- Greatly **decouples services** and allows them to evolve independently.
- **Message brokers** such as RabbitMQ, Apache Kafka, AWS SQS/SNS, and Google Pub/Sub, provide robust messaging capabilities.
- **Use cases** such as event notification, workflow processing, and decoupling services in a microservices architecture.

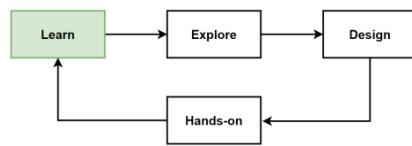


<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>



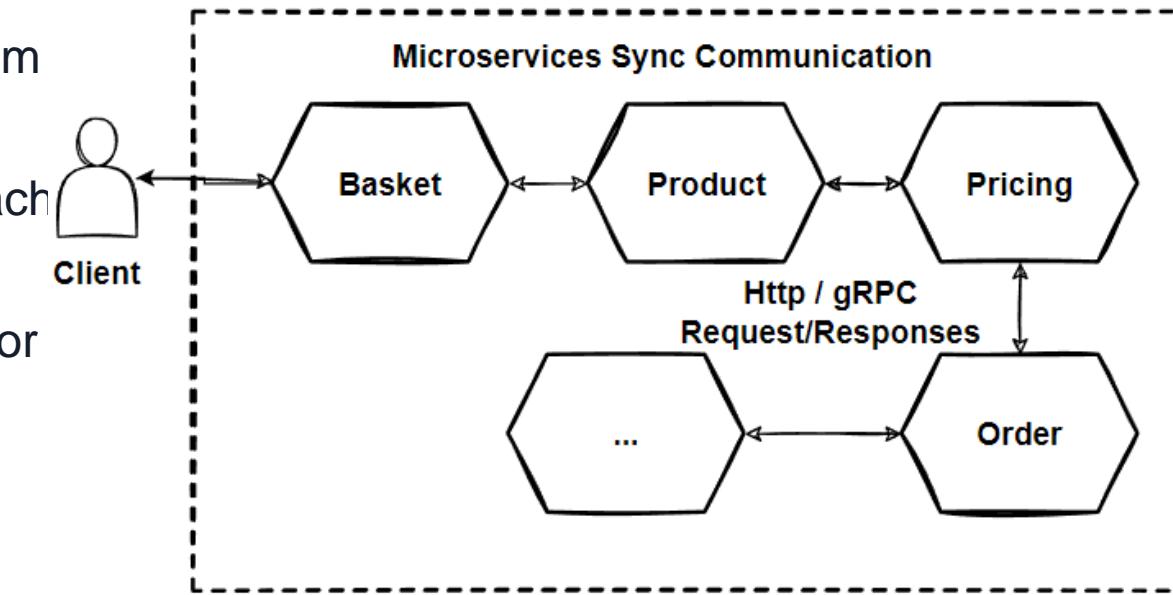
Microservices Communication Types - Sync or Async

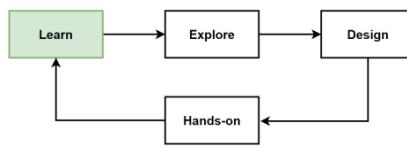




Microservices Synchronous Communication

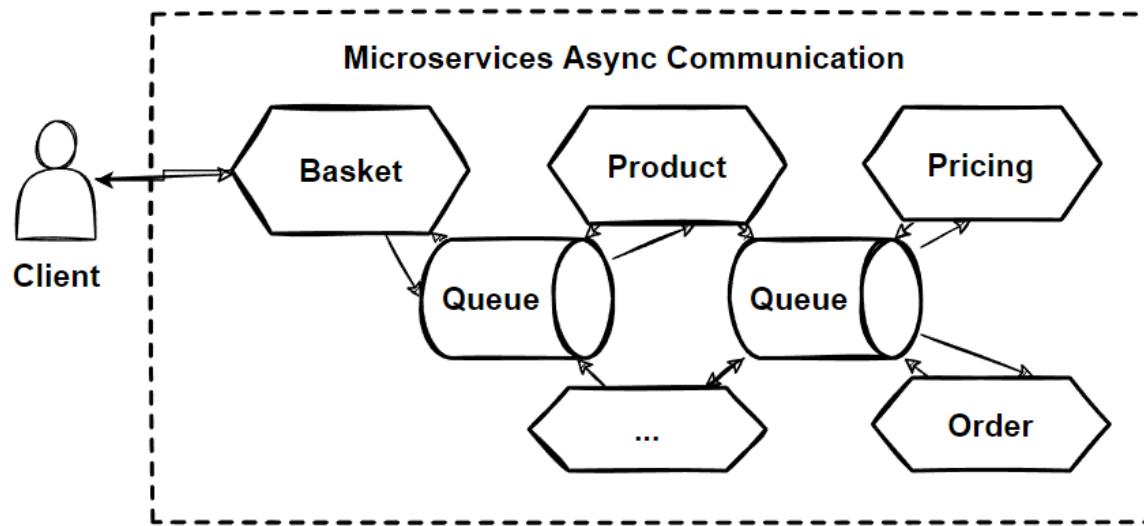
- **Synchronous communication** is using **HTTP** or **gRPC** protocol for returning synchronous response.
- The client **sends a request** and **waits for a response** from the service.
- The client code **block their thread**, until the response reach from the server.
- The synchronous communication protocols can be **HTTP** or **HTTPS**.
- The client sends a request with using **http protocols** and waits for a response from the service.
- The client **call the server** and **block** client their operations.
- The client code will **continue** its task when it **receives** the HTTP server **response**.

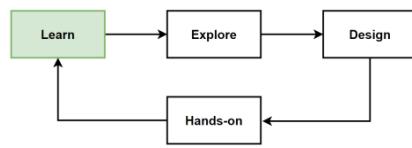




Microservices Asynchronous Communication

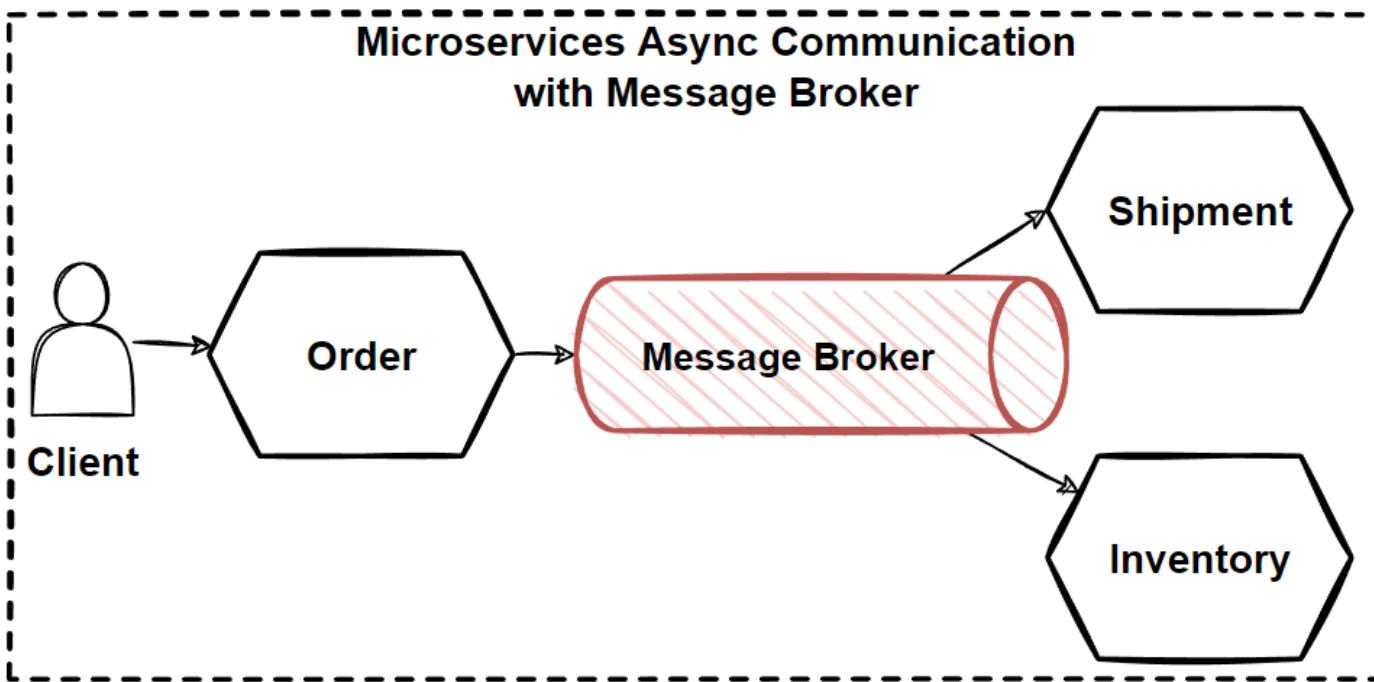
- The client sends a request but it **doesn't wait for a response** from the service. The client **should not have blocked** a thread while waiting for a response.
- AMQP** (Advanced Message Queuing Protocol)
- Using AMQP protocols, the client **sends the message** with using message broker systems like **Kafka** and **RabbitMQ** queue.
- The message **producer does not wait** for a **response**.
- Message consume from the **subscriber** systems in **async** way, and no one waiting for response **suddenly**.
- If there is **busy interactions** in communication across multiple microservices, then use **asynchronous messaging platforms**.

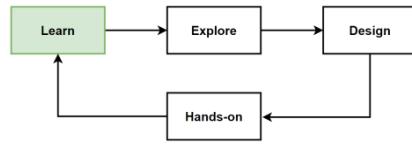




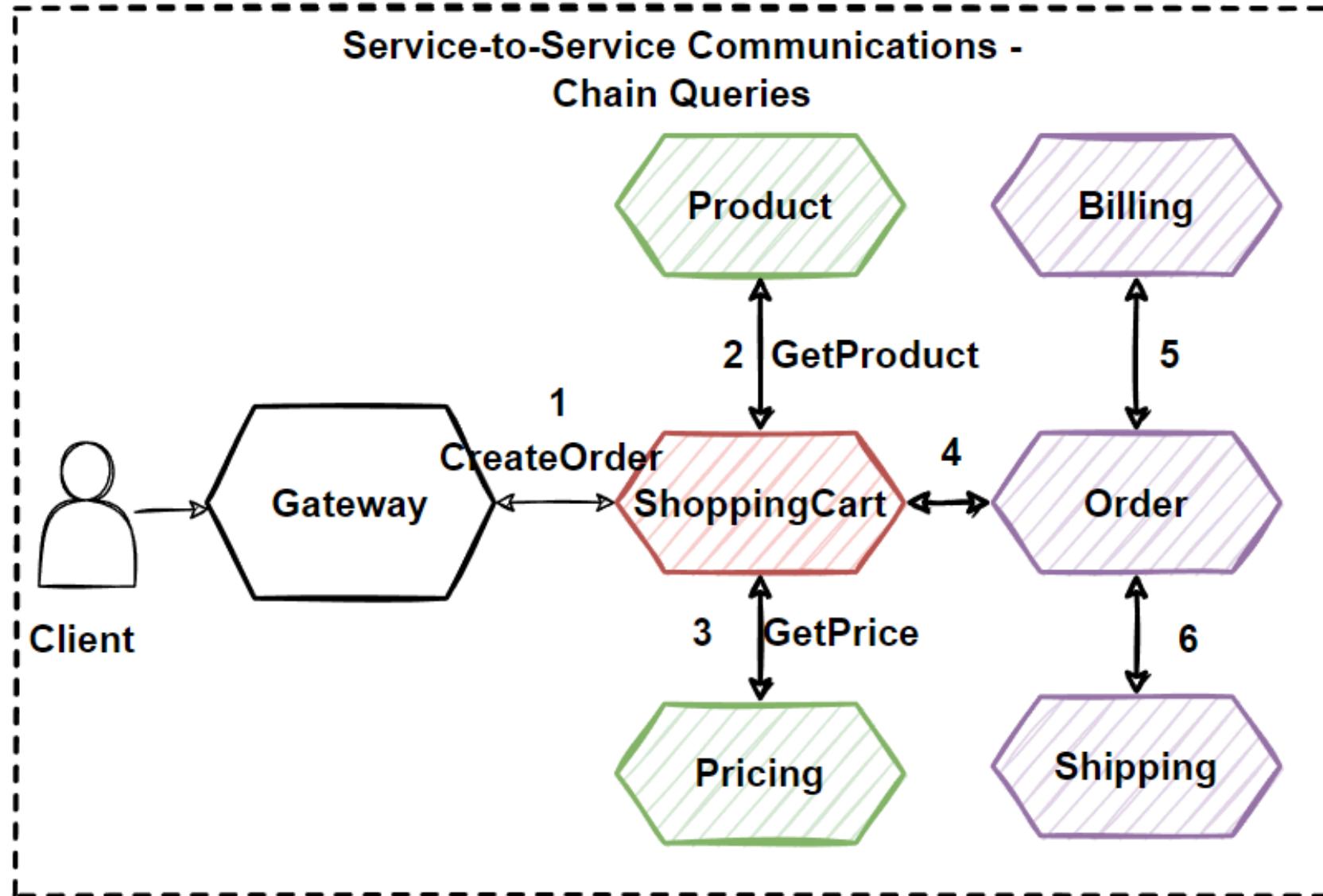
Microservices Asynchronous Communication-2

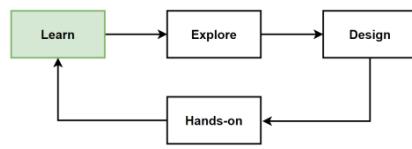
- **Message brokers** are responsible for handling the message sent by the producer service in async messaging-based communication.
- If the **consumer service** is **down** at the moment, the broker might be configured to **retry** as long as necessary for successful delivery.
- Messages can be **persisted if required** or stored only in memory.
- Message broker is **responsible for delivering** the message.
- No longer necessary for both microservices to be **up and running** for successful communication.
- Async messaging provides **loosely couple communication**.





Problem: Chain of request and highly coupled dependent microservices





Benefits of Asynchronous Communication

- New Subscriber Services**

Adding new services is very simple. We can easily subscribe to message that we want to receive. The producer doesn't need to be know about subscribers, we can remove and add subscribers without affecting producer service.

- Scalability**

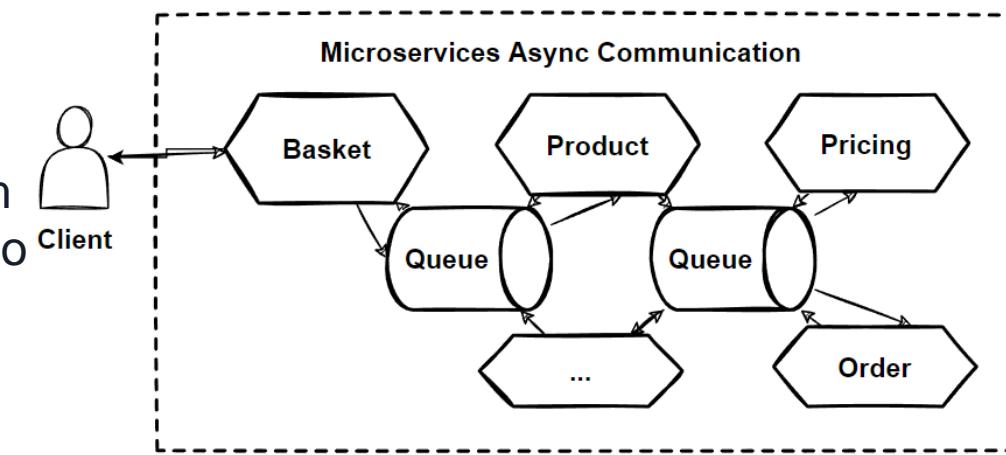
With async communications we can easier to manage scalability issues, can scale producer, consumer and message broker system independently. Scale services according to incoming messages into event bus system. Kubernetes KEDA Auto-scalers.

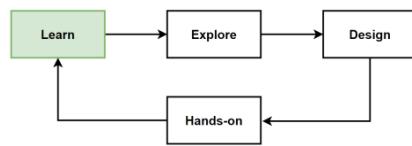
- Event-driven Microservices**

With async communication, we can provide event-driven architectures which is best way to communicate between microservices.

- Retry mechanisms**

Brokers can retry to sending message and keep trying automatically without any custom solutions.





Challenges of Asynchronous Communication

- Single Point of Failure - Message Broker**

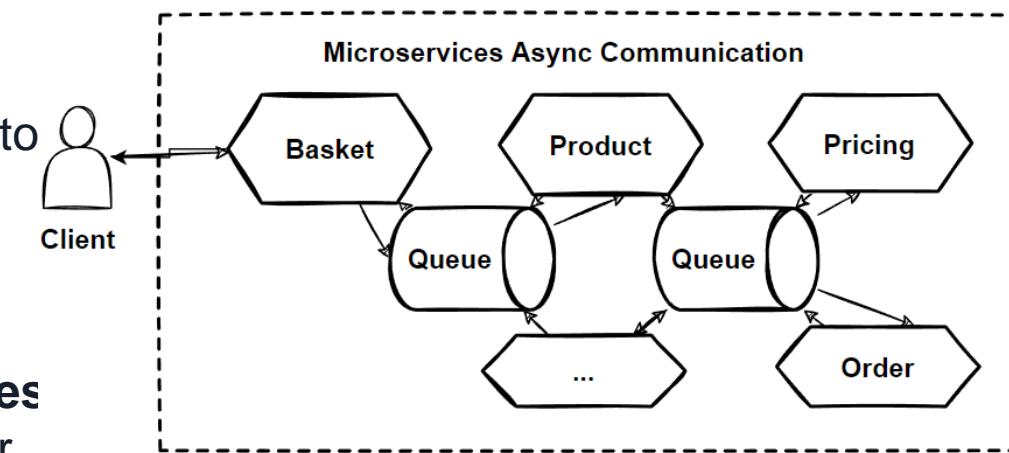
The message broker becomes a single point of failure. We should not rely of all communication with a single node of message brokers, instead we should scale it and use hybrid communication with sync and async in your cases.

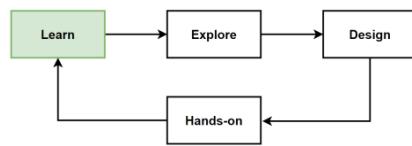
- Debugging**

difficult to debug issues with async communication, it can be hard to trace the flow of a single operation across service boundaries. debugging of the flow and the payload of events takes so many times and hard to debug at the same time.

- At-least-once delivery and Not Guarantee an order of messages**

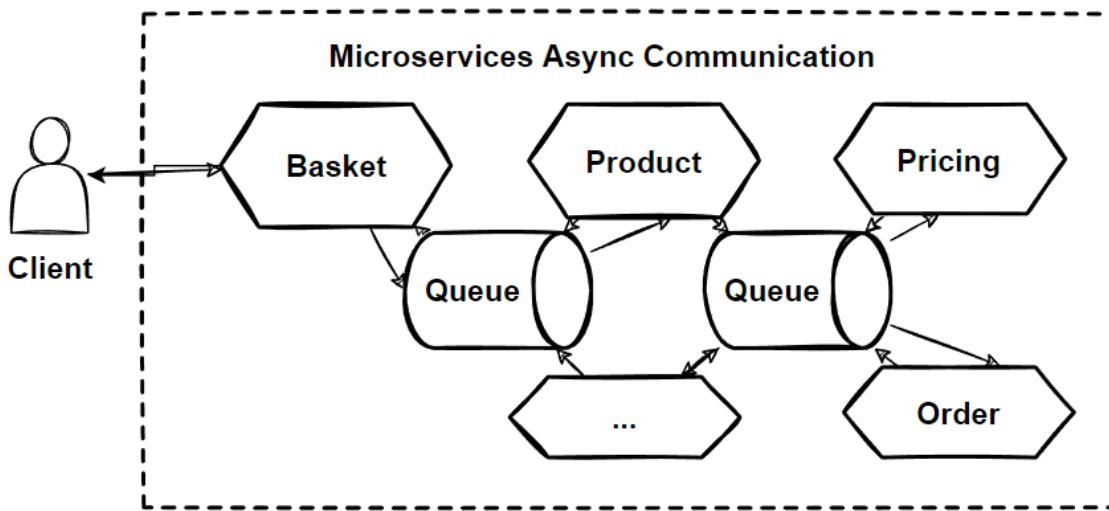
Mostly Brokers use at-least-once delivery and not Guarantee order of messages. Should embrace these message delivery mechanism with applying idempotency consumers and not designing FIFO requires cases.

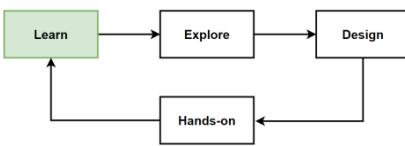




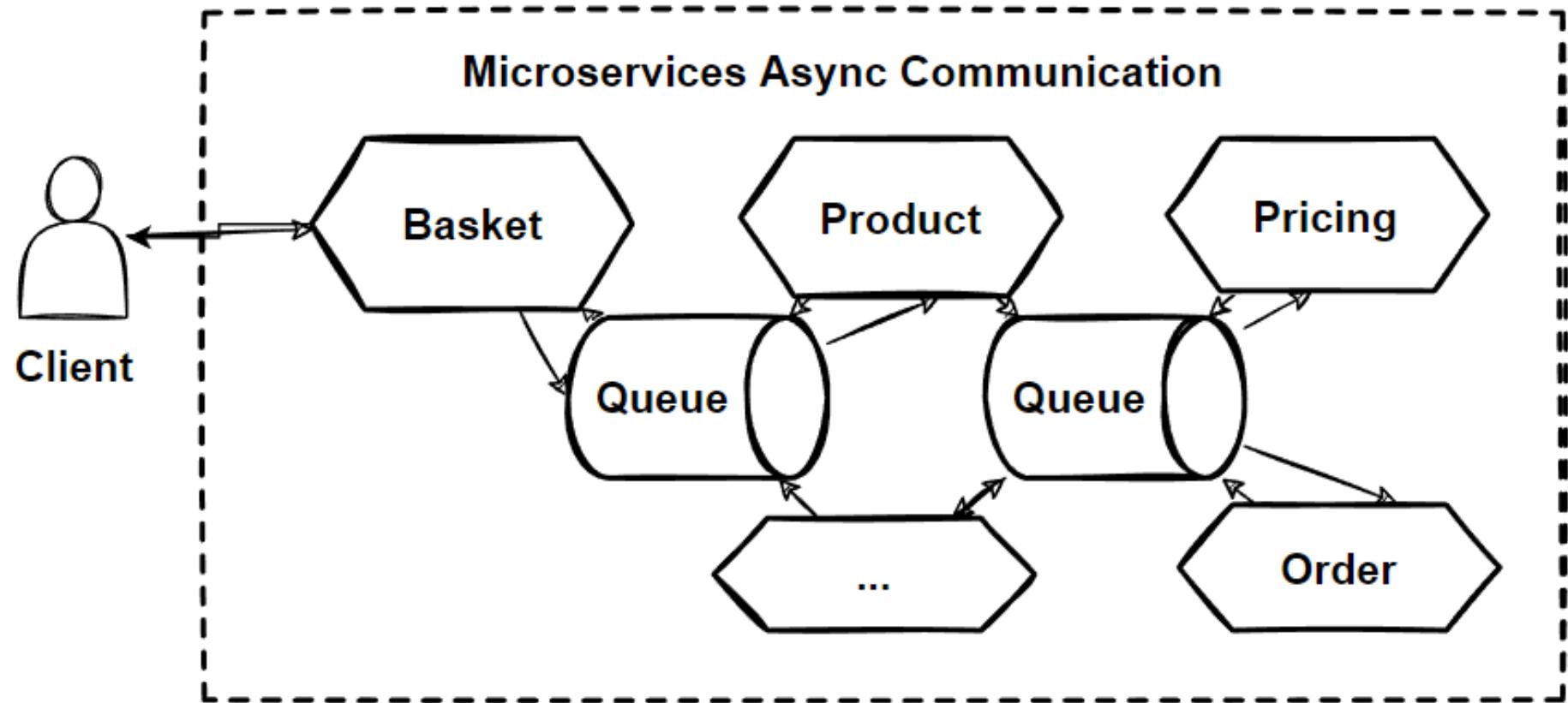
Asynchronous Message-Based Communication

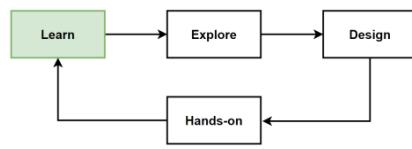
- **Use Asynchronous** message-based communication when you have **multiple microservices are required to interact each other** without any dependency.
- Asynchronous message-based communication is works with **events**.
- **Events** can place the communication between microservices: **Event-driven communication**.
- If **any changes happens** in microservices, it is **propagating changes** across **multiple microservices** as an event.
- Events consumed by **subscriber microservices**.
- **Event-driven** communication bring us **Eventual consistency**.





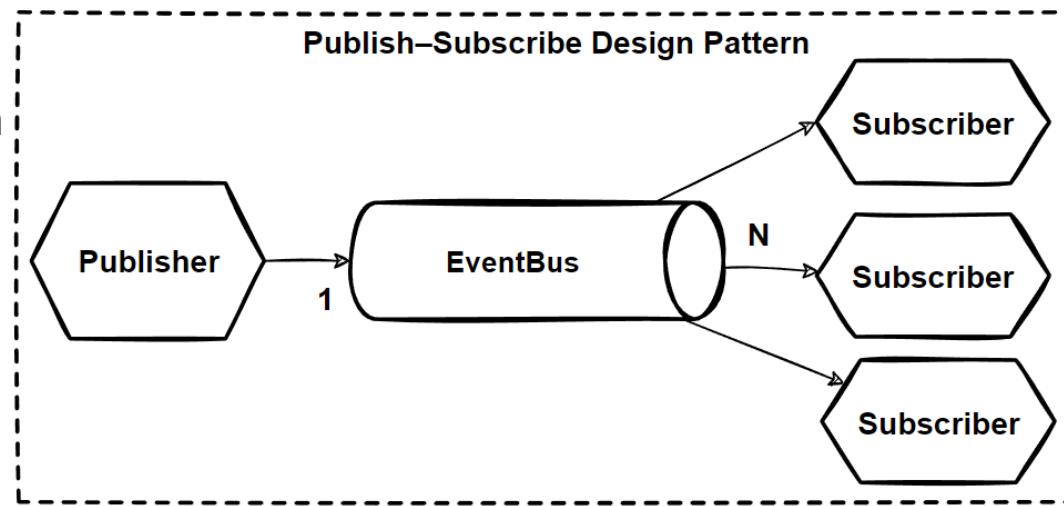
Asynchronous Message-Based Communication-2

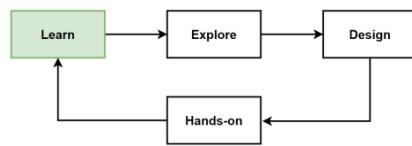




Fan-Out & Message Filtering with Publish/Subscribe Pattern

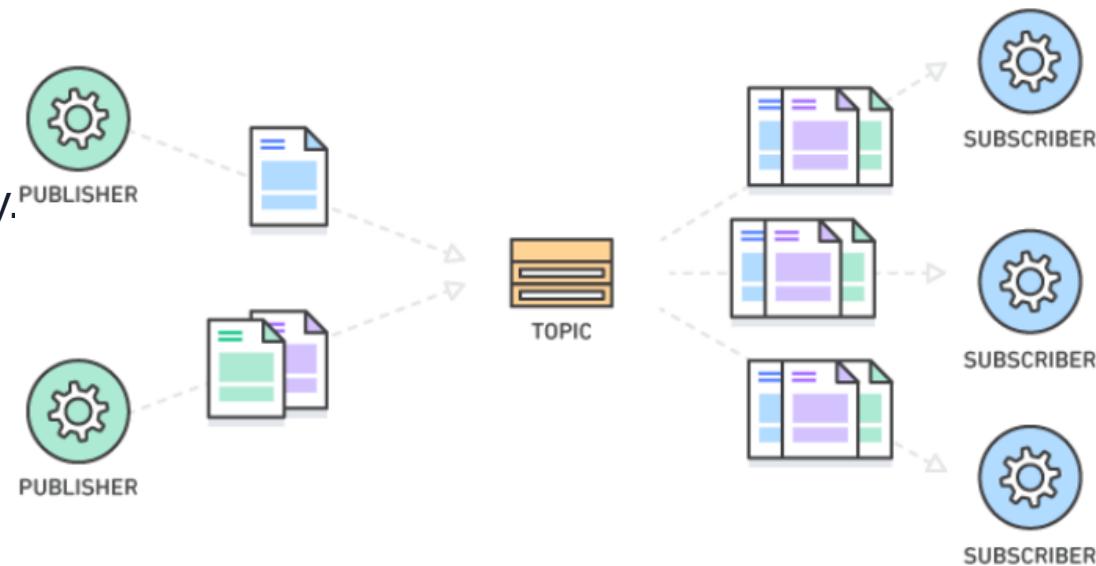
- **Fan-out** is a messaging pattern; ‘fanned out’ to multiple destination in parallel.
- Each of destinations can work and process this messages in parallel.
- **Publisher/subscriber model** to define a topic which is logical access point to enabling message communication with asynchronously.
- Publisher sends the message to the topic, message is immediately fanned out to all subscribers of this topic.
- Each service can operate and scale independently and individually that completely decoupled and asnycronously.
- The publisher and the subscribers don't need to know who is publishing / consuming this message that is broadcasting.
- Deliver the same message to multiple receivers is to use the **Fanout Publish/Subscribe Messaging Pattern**.

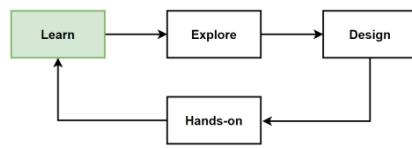




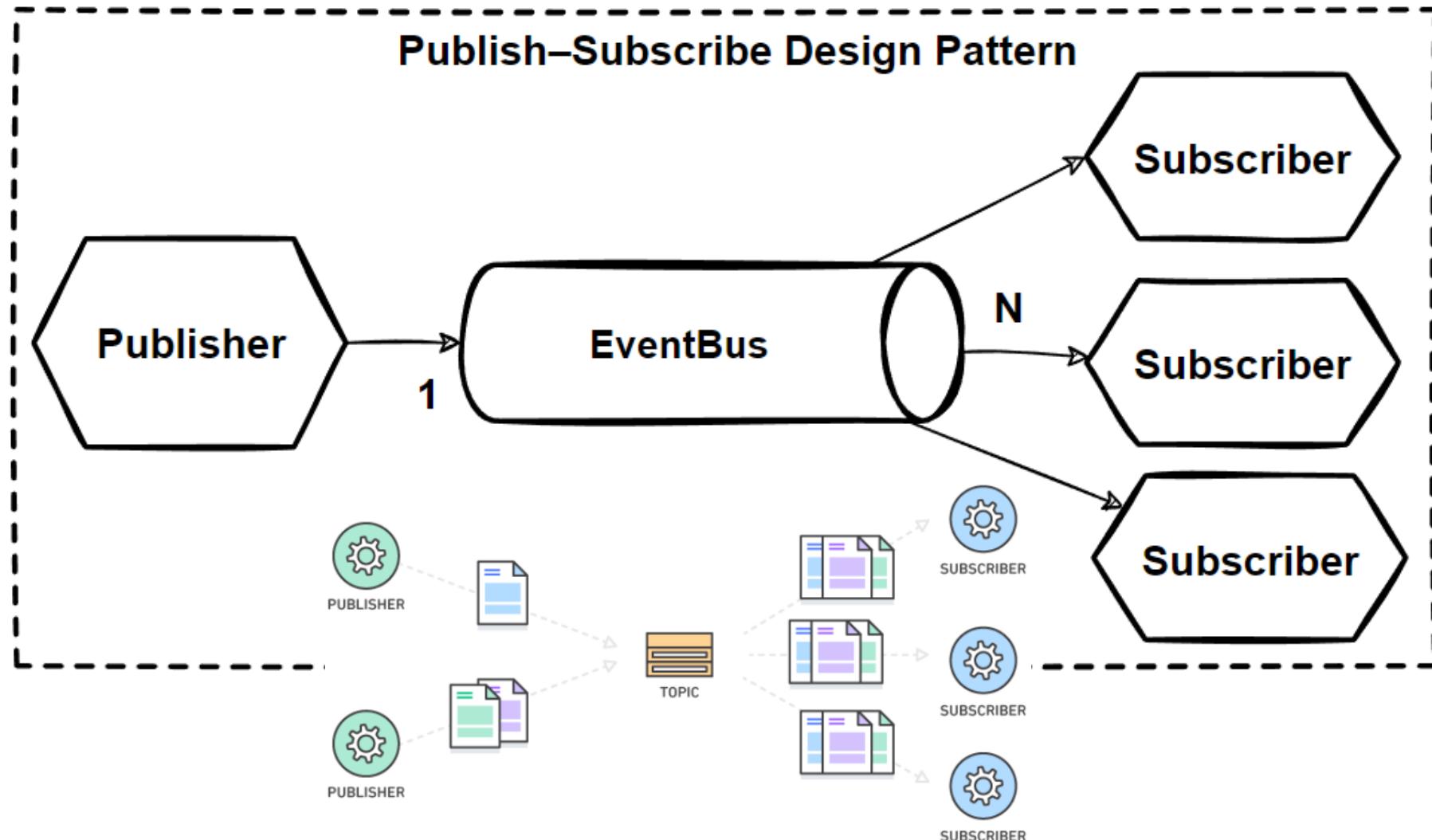
Publish/Subscribe Messaging Pattern

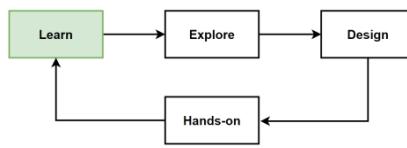
- **Publish/subscribe messaging** is a form of **asynchronous service-to-service communication**.
- Any message **published** to a **topic** is **immediately received** by all of the **subscribers** to the **topic**.
- Enable **event-driven architectures**, and **decouple applications** to increase performance, reliability and scalability.
- Applications are **decoupled** into **smaller, independent building blocks** that are easier to develop, deploy and maintain.
- Publish/Subscribe (Pub/Sub) messaging provides **instant event notifications** for these distributed applications.
- A **message topic** provides a **lightweight mechanism** to broadcast **asynchronous event notifications**.
- All components that subscribe to the topic receive every message, unless a **message filtering policy** is set by the subscriber.





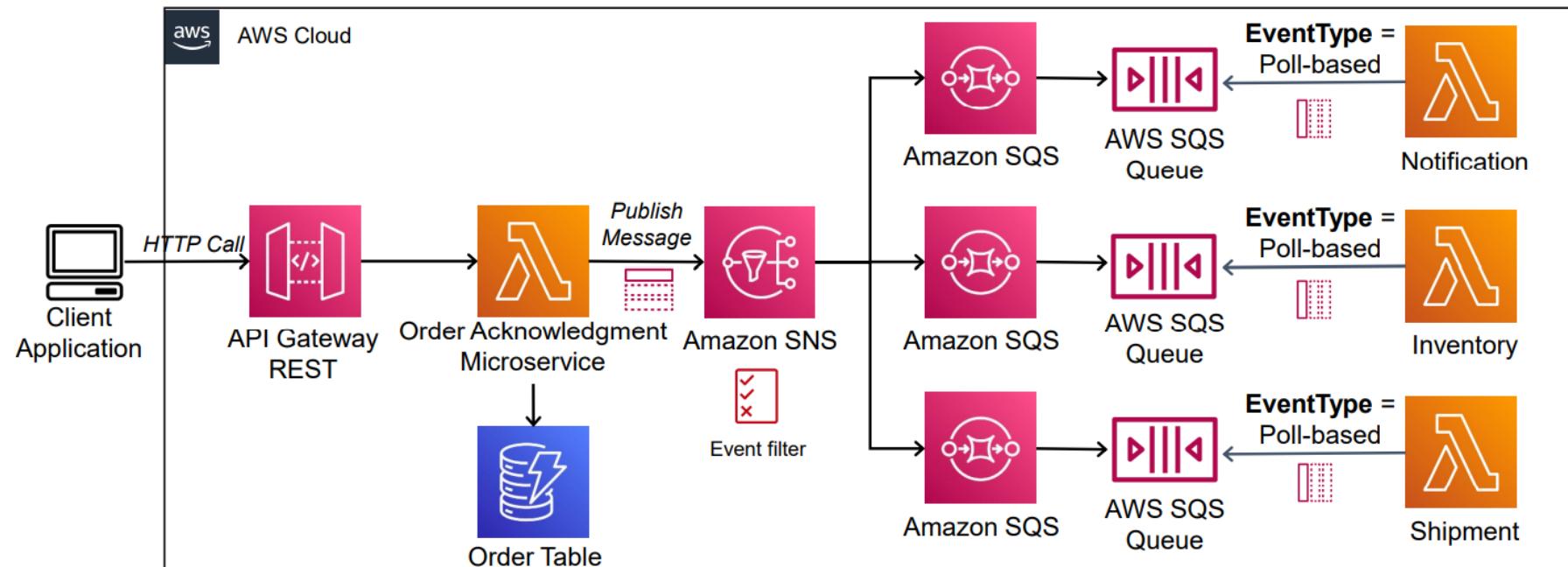
Publish/Subscribe Messaging Pattern-2

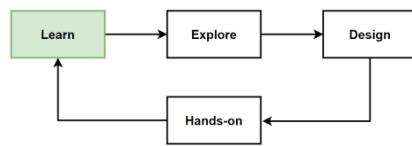




Topic-Queue Chaining & Load Balancing Pattern

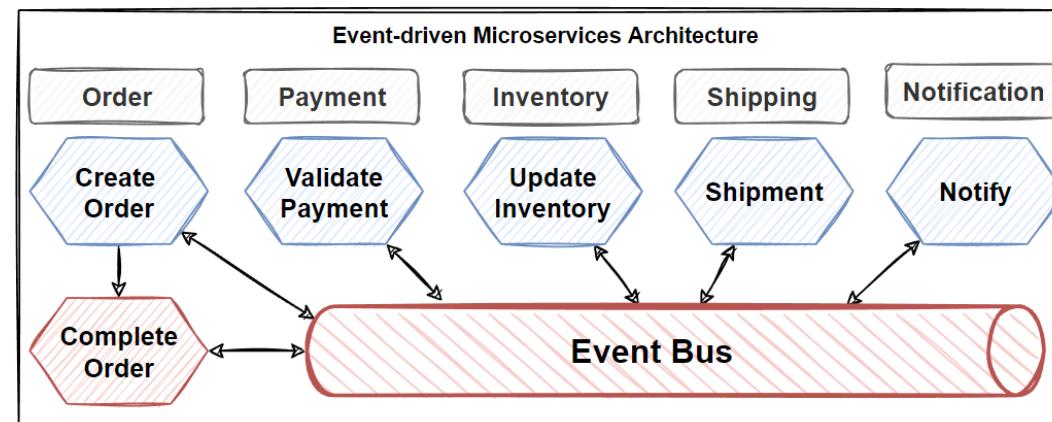
- Use a queue that acts as a **buffer** between the **service** to avoid loss data if the service to fail.
- Services can be down or getting exception or taken offline for maintenance, then events will be loses, disappeared and can't process after the **subscriber service** is up and running.
- Put **Amazon SQS** between EventBridge and Ordering microservices.
- Store this event messages into **SQS queue** with durable and persistent manner, no message will get lost. Queue can act as a **buffering load balancer**.

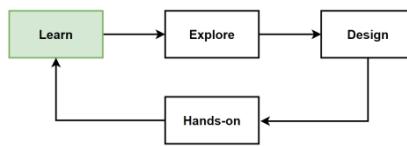




Event-Driven Microservices Architecture

- **Event-driven microservices architecture**, services communicate with each other by **publishing** and **subscribing** to **events**.
- When a service needs to communicate with another service, it **publishes an event** to a message queue or event bus. Other services can then **subscribe** to that **event** and take appropriate action when the **event is received**.
- **Asynchronous communication**
Allows services to communicate asynchronously. Service can publish an event and continue processing without waiting for a response from the other service.
- **Decoupled communication**
Decouples the publisher and subscriber, allows to evolve independently without affecting each other.





Event-Driven Microservices Architecture - 2

- Real-time processing**

Support real-time processing, as events are published and consumed as soon as they occur. Need to react to events in real-time, such as in systems that use CDC to track changes to a database.

- High volume events**

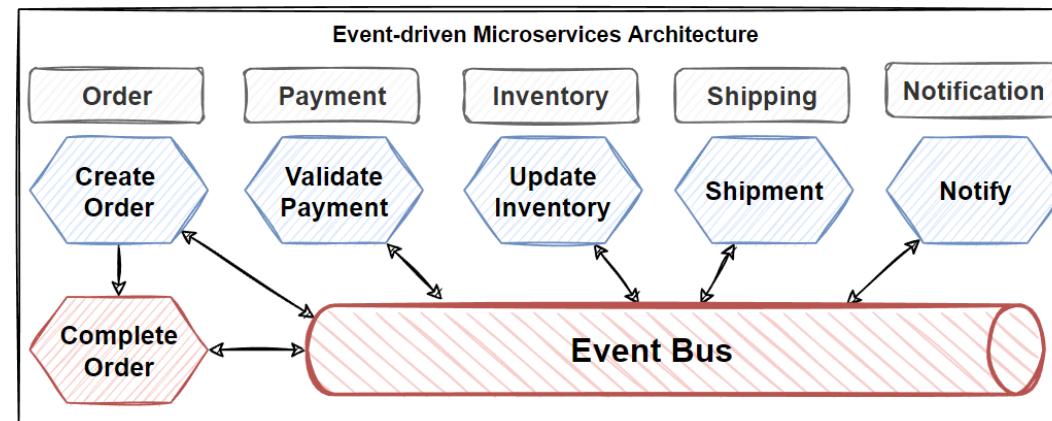
Well-suited to handling high volume events, as they can scale horizontally by adding more event consumers as needed. Can be scaled independently to handle increased load.

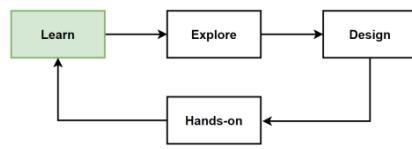
- Responsible business capability**

Each service is responsible for a specific function or business capability.

- Services communicate with each other by publishing and subscribing to events, that make it easier to build and maintain complex systems.

- Allows to work on different parts of the system in parallel without having to worry about the impact on other components.





Real-time Processing and High Volume Events in Event-Driven Microservices Architecture

- **Real-time processing**

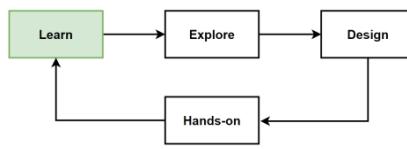
Real-time processing is achieved by using a message queue or event bus to publish and consume events as they occur.

- When an event is generated, it is published to the message queue or event bus and made available to any interested subscribers.
- Allows you to react to events in real-time, as they are published and consumed as soon as they occur.
- When need to perform real-time analytics or trigger actions based on changes to the data.

- **High Volume Events**

Using a message queue or event bus that can handle high volumes of events and distribute them to multiple consumers.

- When you have a system that generates a large number of events, use a event bus to distribute those events to multiple consumers, that can process events in parallel.
- Allows you to scale up the number of event consumers as needed to handle increased load.
- Event-driven microservices architectures to process events in real-time and scale to handle high volumes of events.



Event Hubs and Event Streaming in Event-Driven Microservices Architecture

- **Event Hubs**

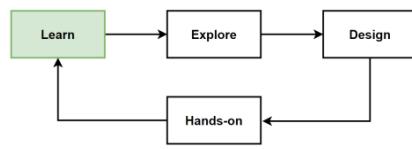
Act as a central hub for data ingestion and distribution, allowing microservices to publish and subscribe to data streams.

- Each microservice publish its data to an Event Hub. Other microservices can then subscribe to the Event Hub and consume the data as needed.
- Communicate with each other in real-time, that needs to be highly reliable and scalable.

- **Event Streaming**

Allows you to capture and process a stream of events in real-time. Publish and consume events as they occur.

- Allowing to build real-time data pipelines that can process and analyze data as it is being generated.
- One common use case for Event Hubs and event streaming in microservices architectures is real-time analytics.



Use Cases of Event Hubs and Event Streaming

- **Use Case – Customer Purchase on E-Commerce**

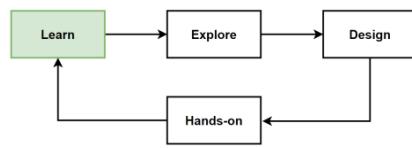
E-commerce website that generates a large number of events as customers browse and purchase products.

- Using event streaming, build a real-time analytics pipeline that processes these events as they occur and generates insights in real-time.
- This could include tracking customer behavior, identifying trends and patterns, and triggering actions based on changes to the data.

- **Use Case – Data Synchronization**

Different services may need to access the same data, and not be able to access each other's databases directly.

- Using event streaming, we can build a data synchronization pipeline that captures changes made to the data and streams them to other services in real-time. Ensures to access the most up-to-date data.
- Event Hubs and event streaming is a powerful tool for building real-time data pipelines in microservices architectures that provides flexible platform for data ingestion and distribution.

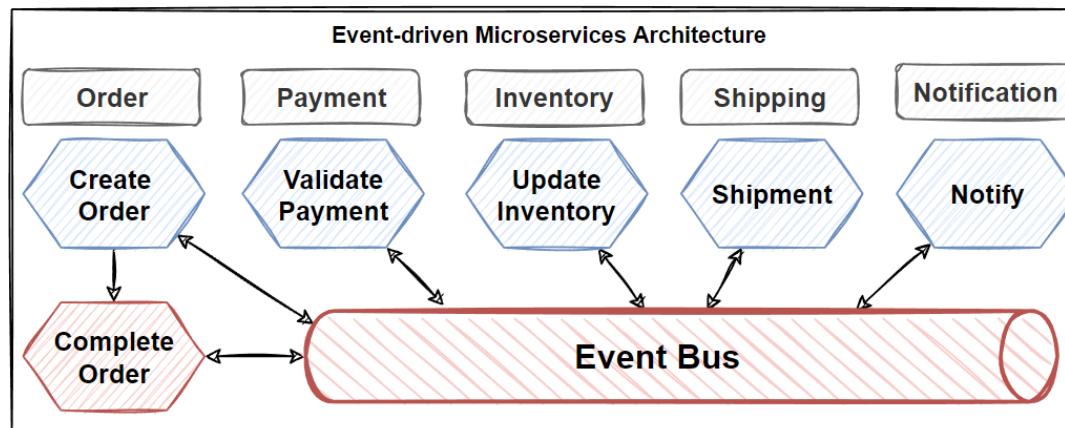


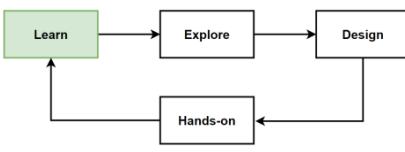
Real-world Examples of Event-Driven Microservices

- **E-commerce application** we have a few microservices like: **customer, order, payment and products**.
- **Customer create orders** with some products and, if the payment is successful, the products should be delivered to the customer.

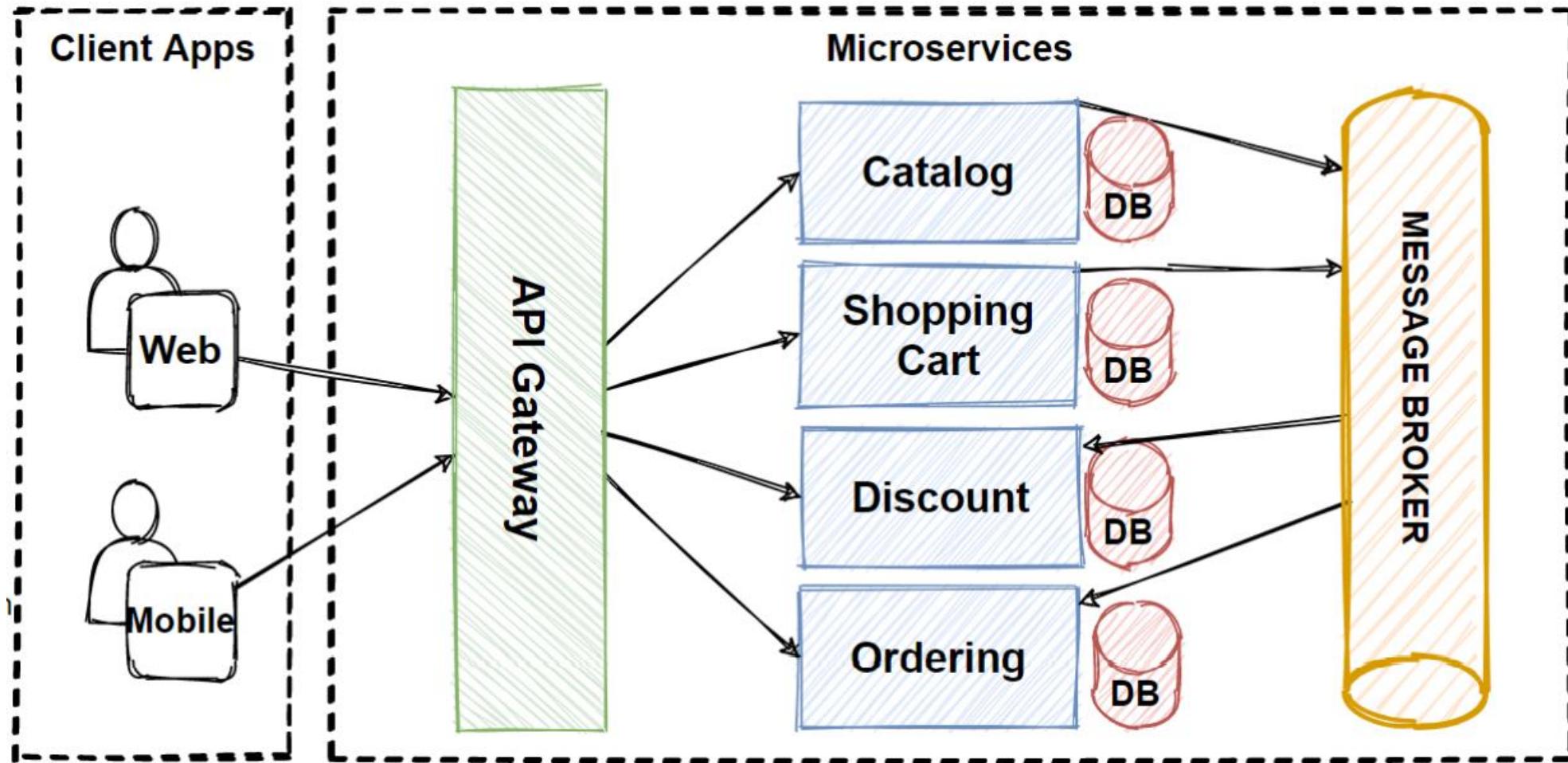
Events

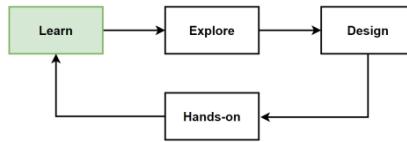
- a customer creates an order
- the customer receives a payment request
- if the payment is successful the stock is updated and the order is delivered
- if the payment is not successful, rollback the order and set order status is not completed.
- This is more **humanly readable** and, if a new business requirement appears, it is **easier to change** the flow.
- **Microservices will only care about the events**, not about the other microservices, **process only events** and **publish new event** to trigger other services.





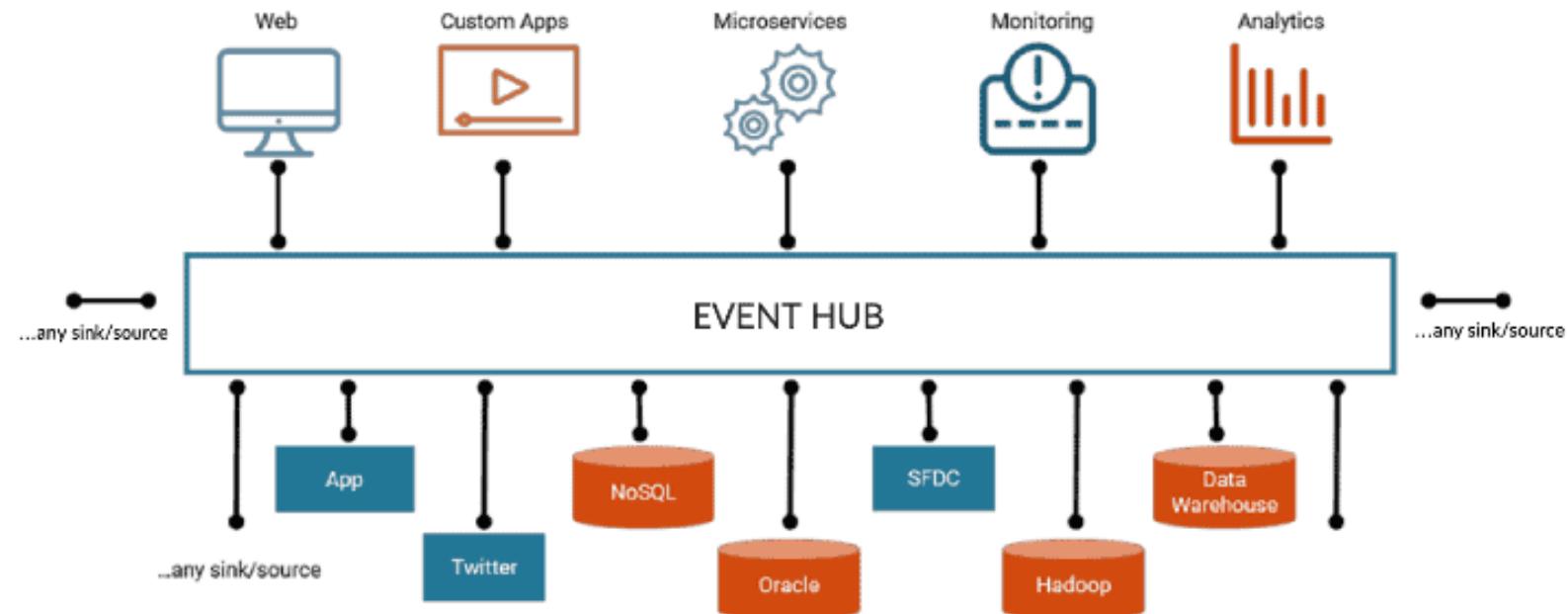
Traditional Event-Driven Microservices Architecture

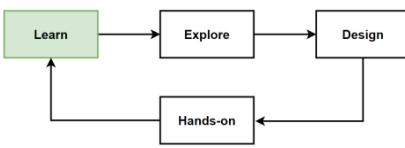




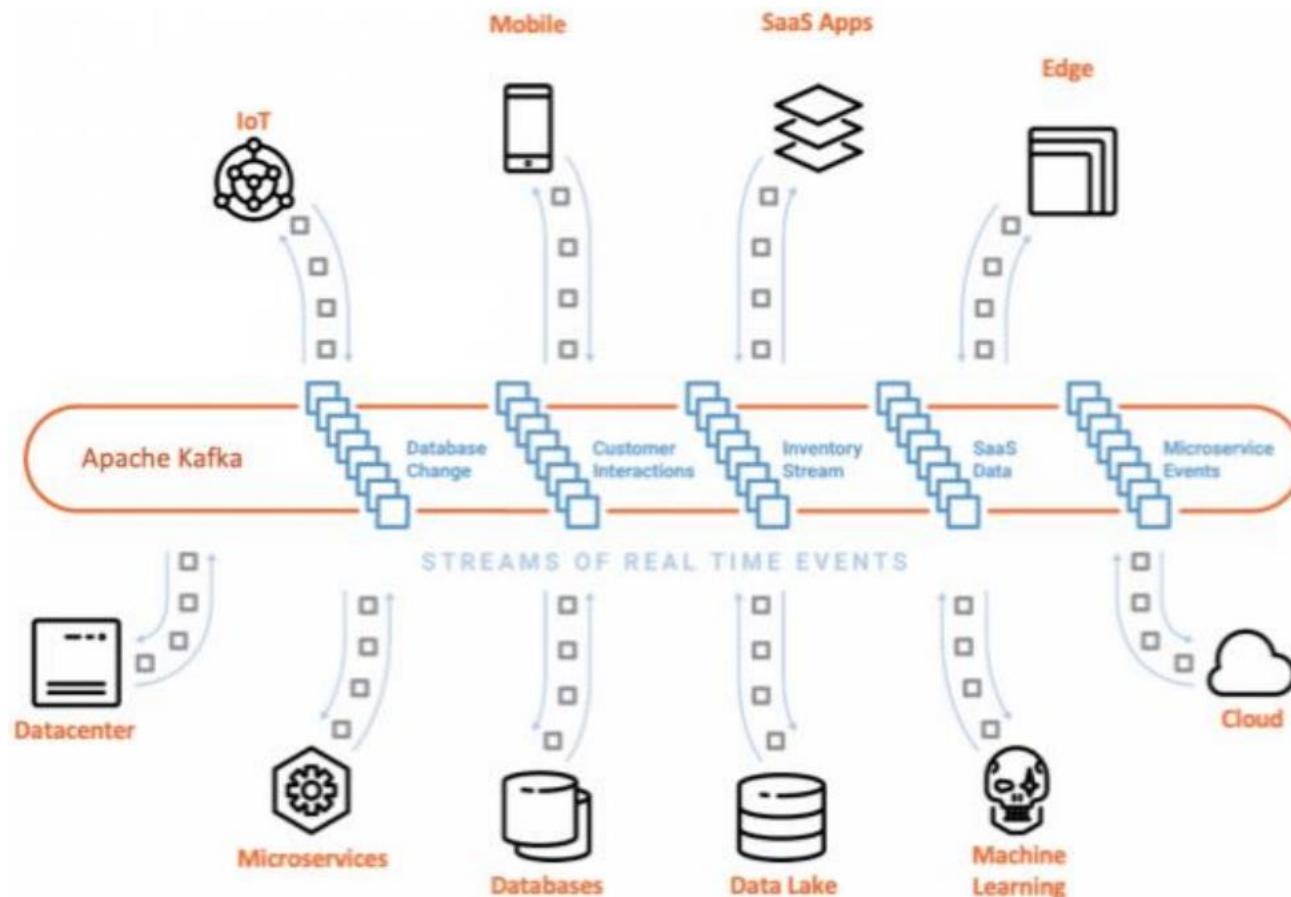
Evolved Event-Driven Microservices Architecture

- **Huge Innovations** on the Event-Driven Microservices Architectures.
- **Real-time messaging platforms, stream-processing, event hubs, real-time processing, batch processing, data intelligence.**
- **Event-Hubs** is huge event store database that can make **real-time processing**.





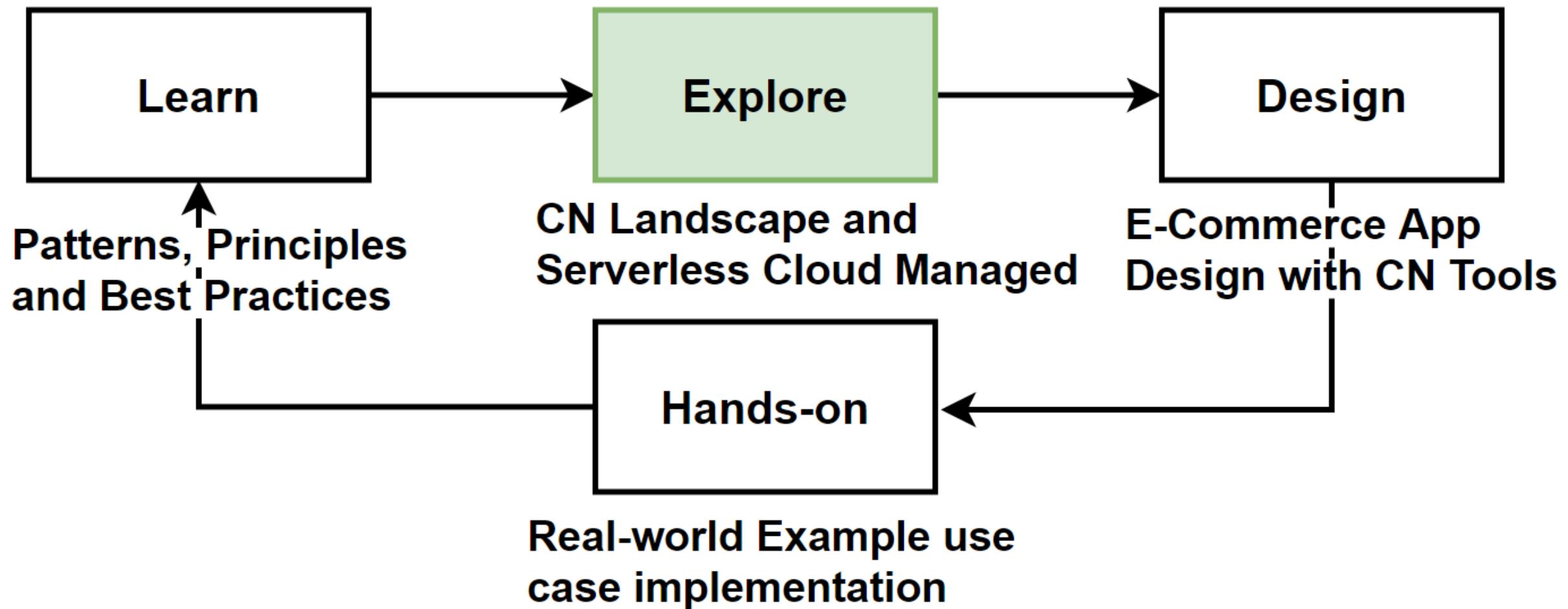
Kafka Event-Driven Microservices Architecture



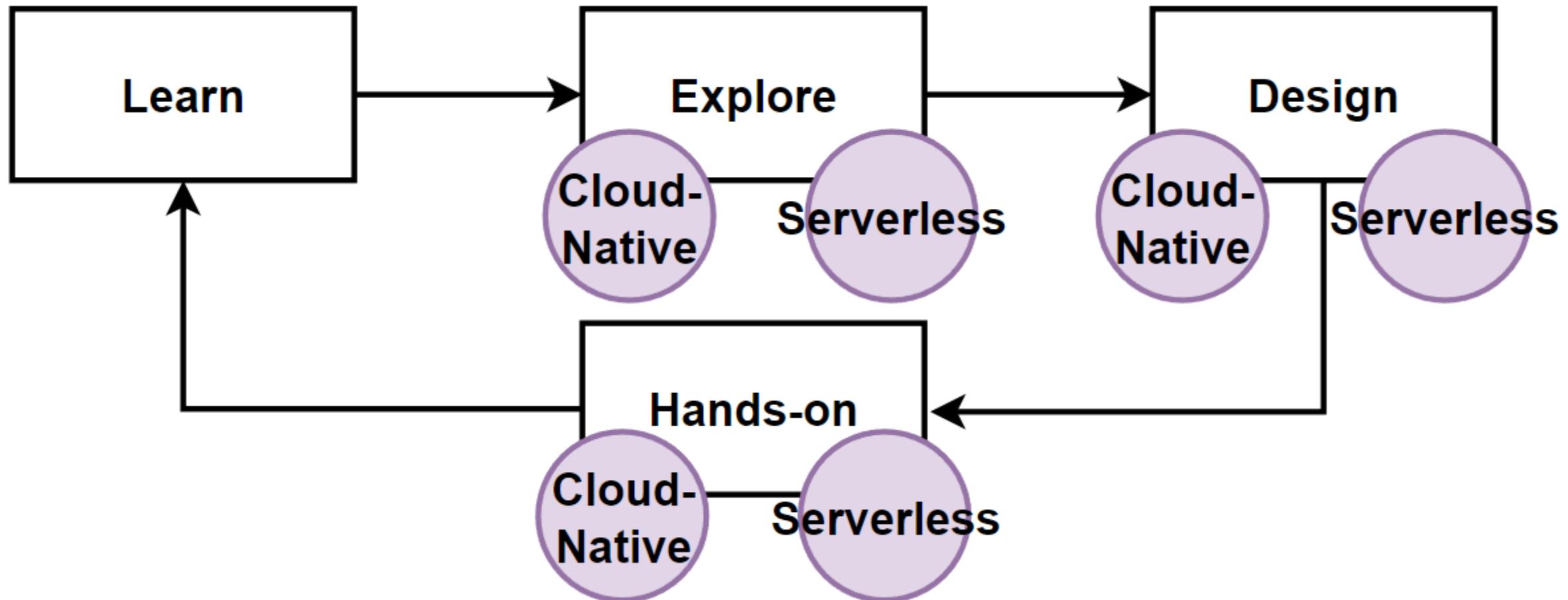
- Global-scale
- Real-time
- Persistent Storage
- Stream Processing



Explore: Cloud Managed and Serverless Microservices Frameworks

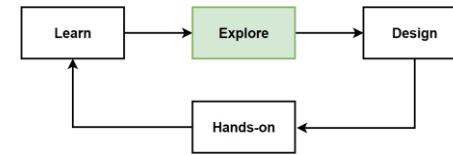


Way of Learning – Cloud-Native & Serverless Cloud Managed Tool



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Explore: Cloud-Native Distributed Message Brokers

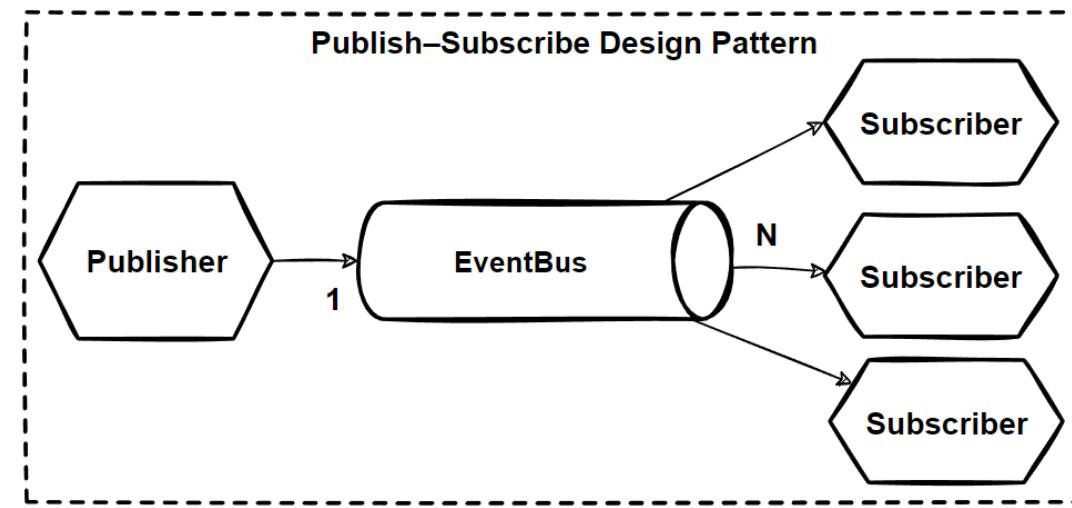
- **Horizontally Scalable Distributed Message Brokers** that designed to work seamlessly with cloud-native architectures and scale out by adding more nodes to the system.

Distributed Message Brokers

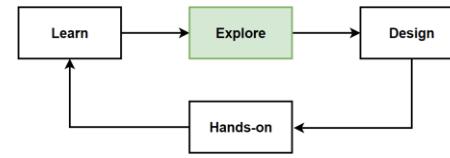
- Apache Kafka
- RabbitMQ
- Redis Pub/Sub
- Apache Rocket MQ
- Apache Spark
- Apache Storm
- cloudevents - CNCF Incubating Project

Cloud Serverless Message Brokers Services

- Amazon SNS
- Amazon EventBridge
- Amazon Kinesis
- Azure Service Bus
- Azure Event Grid
- Azure Event Hubs
- Google Cloud Events
- Upstash Kafka



Goto -> <https://landscape.cncf.io/>

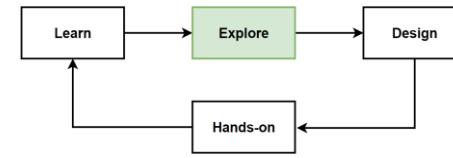


What is Apache Kafka ?

- Apache Kafka is the most popular **open-source event streaming** platforms.
- Designed for **horizontally scalable, distributed, and fault-tolerant**
- Distributed **publish-subscribe event streaming** platform
- Publishers **send** messages to **topics**, **subscriber** of a **topic** receives all the messages published to the topic.
- **Event-driven Architecture:** Kafka is used as an event router, and the microservices publish and subscribe to the events.
- **Distributed working**, and it provides a **horizontal scalable** system.
- Built on top of **ZooKeeper synchronization** service.
- **Key Components:** Topics, Partitions, Brokers, Producer, Consumer, Zookeeper



kafka



Apache Kafka Benefits

- **Reliability**

Kafka is distributed, partitioned, replicated and fault tolerance. So it is also High Availability.

- **Scalability**

Since its distributed architecture, Kafka scales easily without any down time.

- **Durability**

Kafka uses Distributed commit log. This persists to events on disk as log commit very fast. Its durable for infinitive or a given parameter days or weeks.

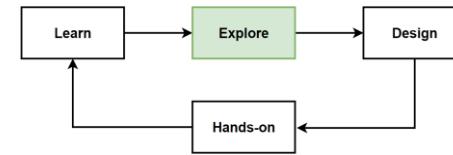
- **Performance**

Kafka has high throughput for both publishing and subscribing messages. It is distributed event streaming platform capable of handling trillions of events a day.

- Kafka has better throughput, built-in partitioning, replication, and fault-tolerance architecture.



kafka



Apache Kafka Use Cases

▪ Messaging

Message brokers are used to decouple services from each other. In event-driven architecture, Kafka is used as an event router, and microservices publish and subscribe to the events.

▪ Metrics

Kafka is often used for operational monitoring data.

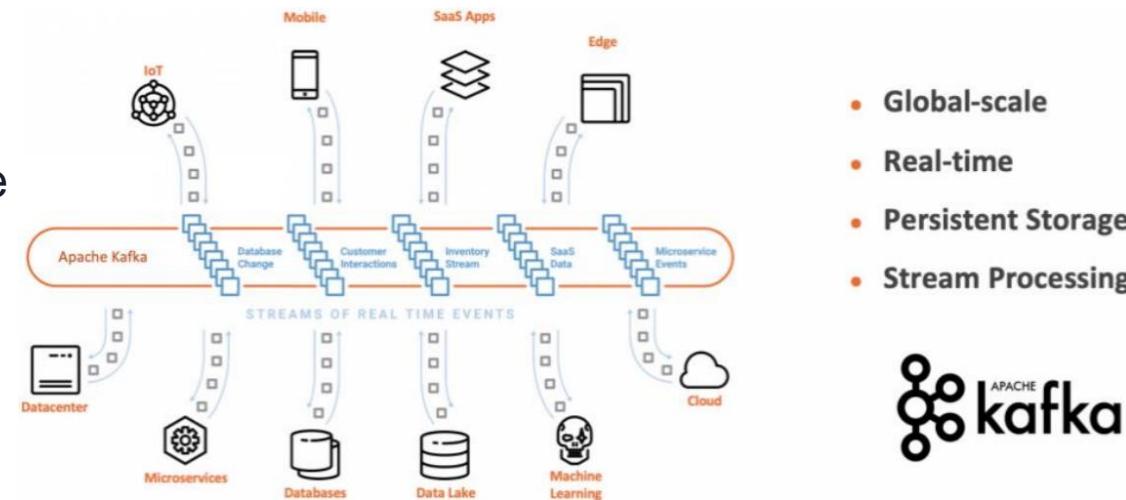
Aggregating statistics from distributed applications to produce centralized feeds of operational data.

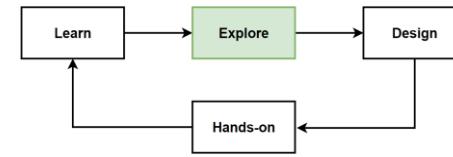
▪ Log Aggregation

Log aggregation solution that collect logs from multiple services. Log aggregation collects physical log files servers and puts them in a central place for processing.

▪ Stream Processing

Kafka process data in processing pipelines consisting of multiple stages. Storm and Spark Streaming read data from a topic and processes it.





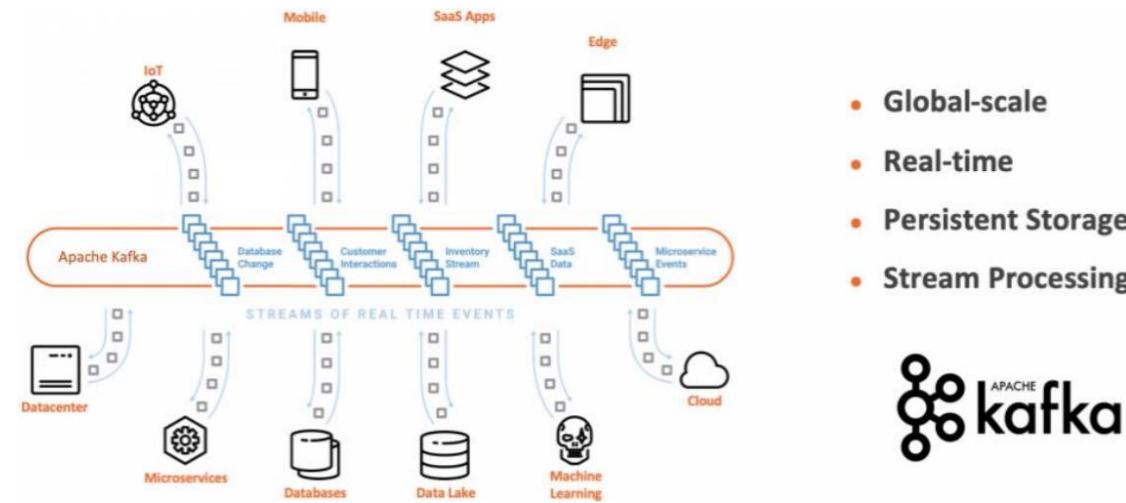
Apache Kafka Use Cases

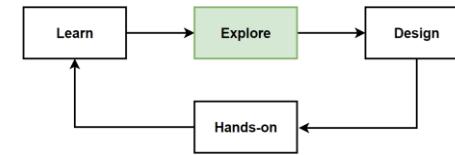
- **Website Activity Tracking**

Kafka follows the user activity tracking pipeline as a set of real-time publish-subscribe feeds.

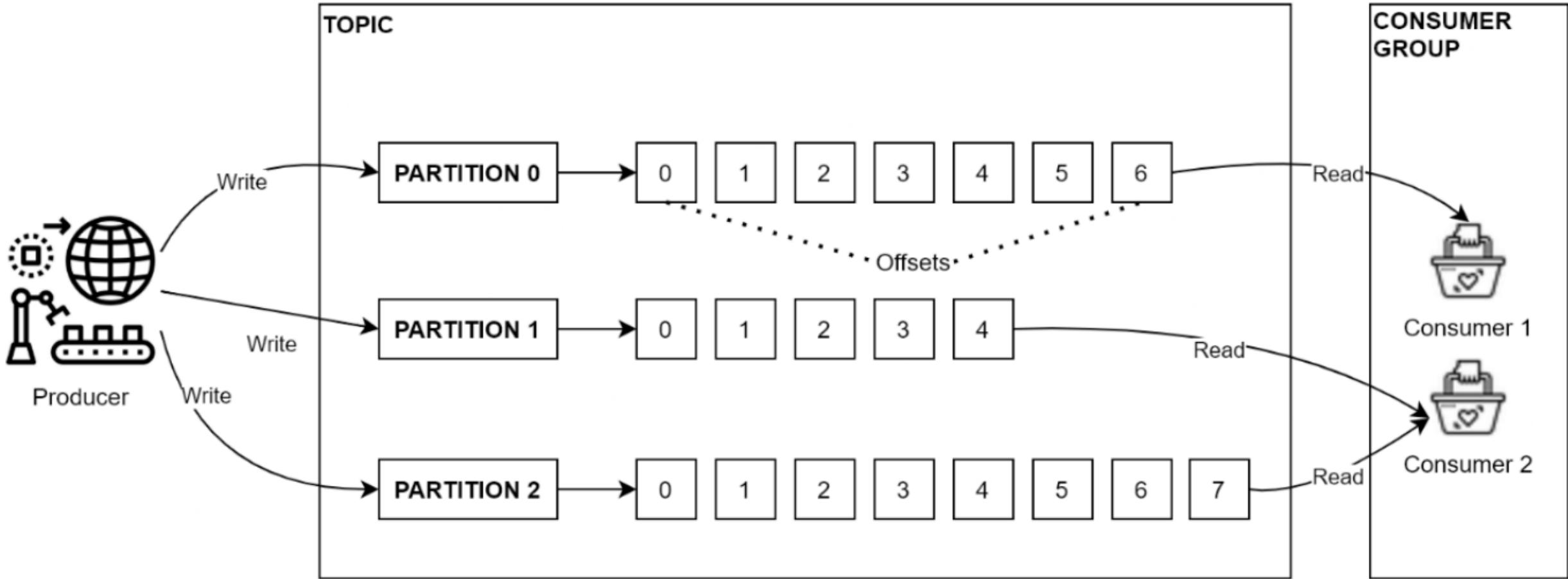
- **Event Sourcing**

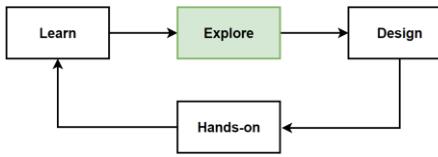
Kafka is using with the event-driven architecture, so in that cases also can be using for event storing and event sourcing.





Kafka Components: Topic, Partitions, Offset and Replication Factor





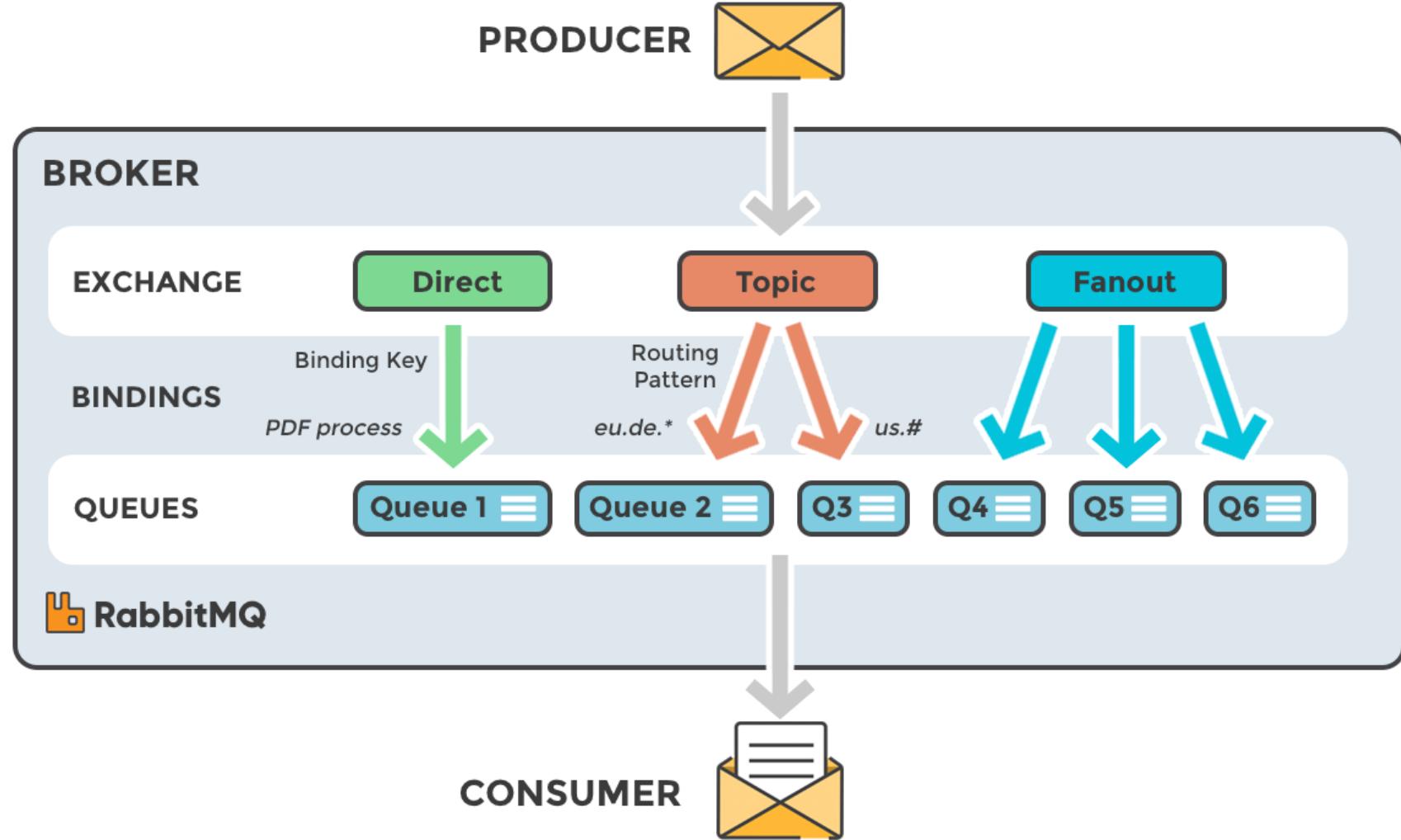
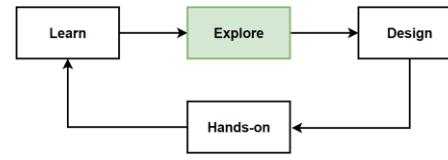
What is RabbitMQ ?

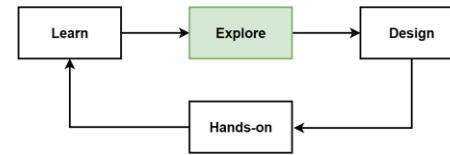
- **RabbitMQ** is a **message broker software** that implements the **Advanced Message Queuing Protocol (AMQP)**.
- It allows applications to communicate with each other by **sending and receiving messages** through **queues**.
- **RabbitMQ** is a **message queuing system** that transmit a message received from any source to another source.
- Similar ones can be listed as **Apache Kafka, Msmq, Microsoft Azure Service Bus, Kestrel, ActiveMQ**.
- All **transactions** can be **listed in a queue** until the source to be transmitted gets up.
- It allows to **send and receive messages asynchronously**.
- **RabbitMQ's** support for **multiple operating systems** and open source code is one of the most preferred reasons.
- Main Components of RabbitMQ: **Producer, Queue, Consumer, Message, Exchange, Binding and FIFO**.



RabbitMQ

RabbitMQ Components: Producer, Queue, Consumer, Message, Exchange, Binding





RabbitMQ Queue Properties

- **Queue Name**

The name of the queue we have defined.

- **Durable**

Determines the lifetime of the queue. If we want persistence, we have to set it true.

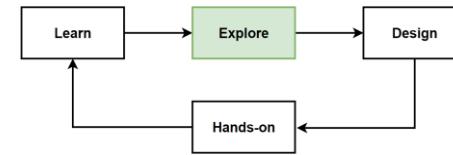
- **Exclusive**

Contains information whether the queue will be used with other connections.

- **AutoDelete**

Contains information about deletion of the queue with the data sent to the queue passes to the consumer side.



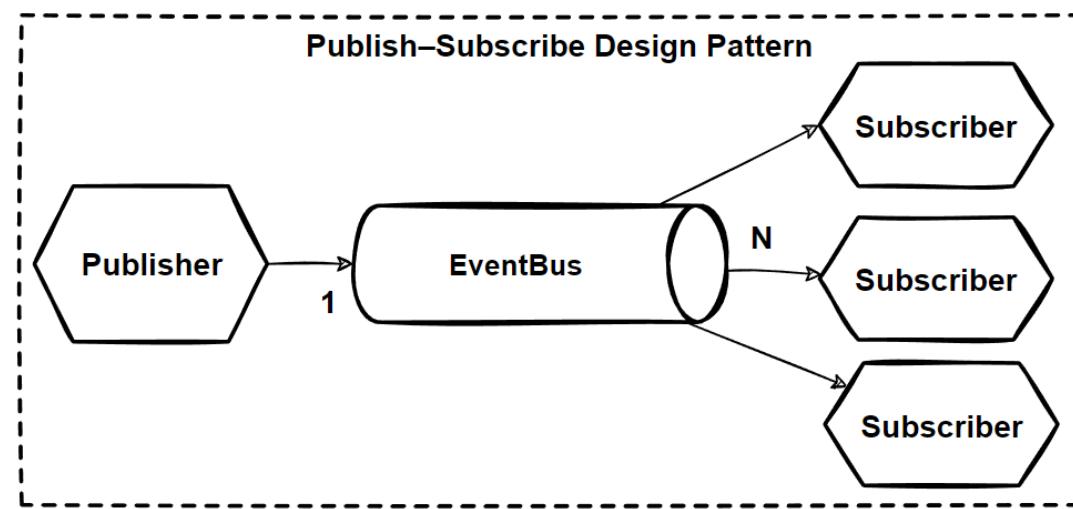


Explore: Cloud Serverless Message Brokers

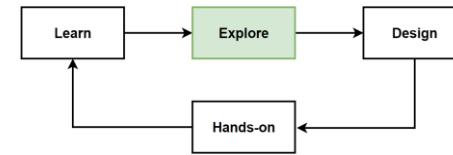
- **Managed serverless message brokers** and event hub services provided by major cloud providers **simplify the process** of setting up and maintaining messaging systems for pub/sub messaging in cloud-native applications.
- Fully managed and **automatically scale to handle the workload**, reducing the operational overhead of managing caching infrastructure.

Cloud Serverless Message Brokers

- [Amazon SNS](#)
- [Amazon EventBridge](#)
- [Amazon Kinesis](#)
- [Azure Service Bus](#)
- [Azure Event Grid](#)
- [Azure Event Hubs](#)
- [Google Cloud Events](#)
- [Google Cloud Pub/Sub](#)
- [Upstash Kafka](#)
- [memphis.dev](#)



Goto -> <https://landscape.cncf.io/>

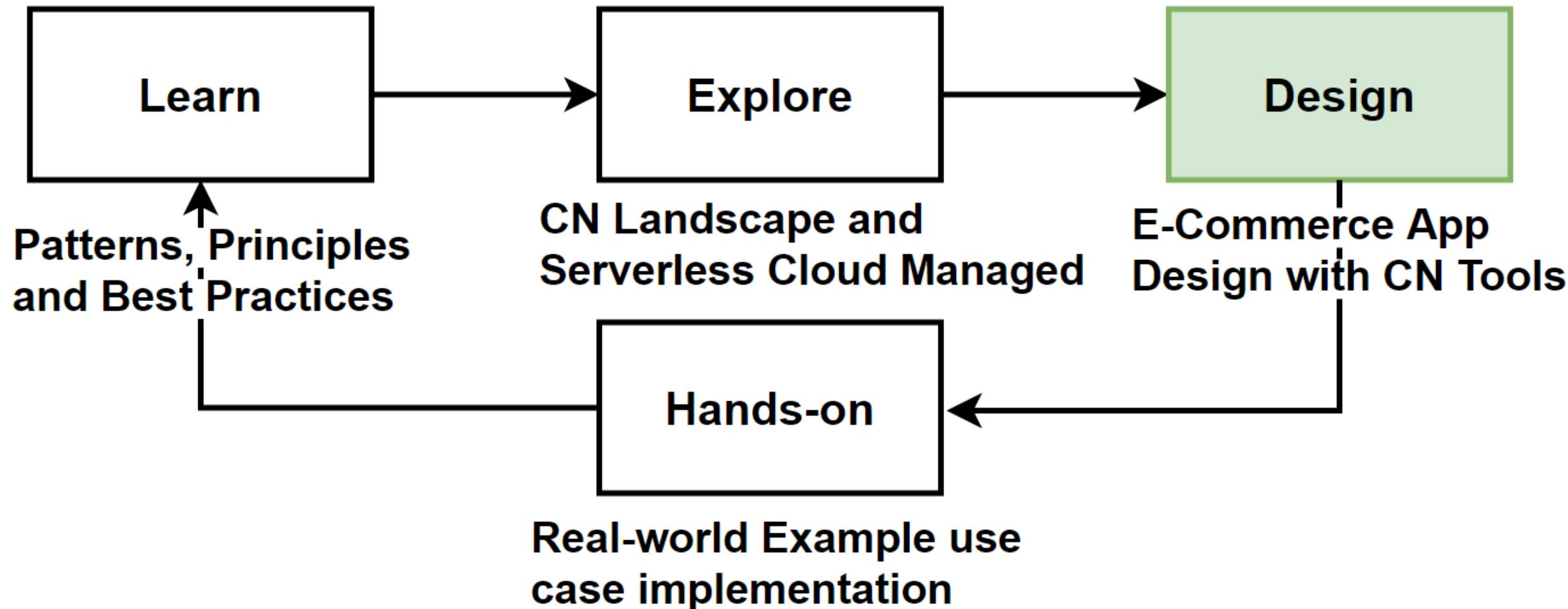


Explore: Upstash Kafka: Serverless Message Broker

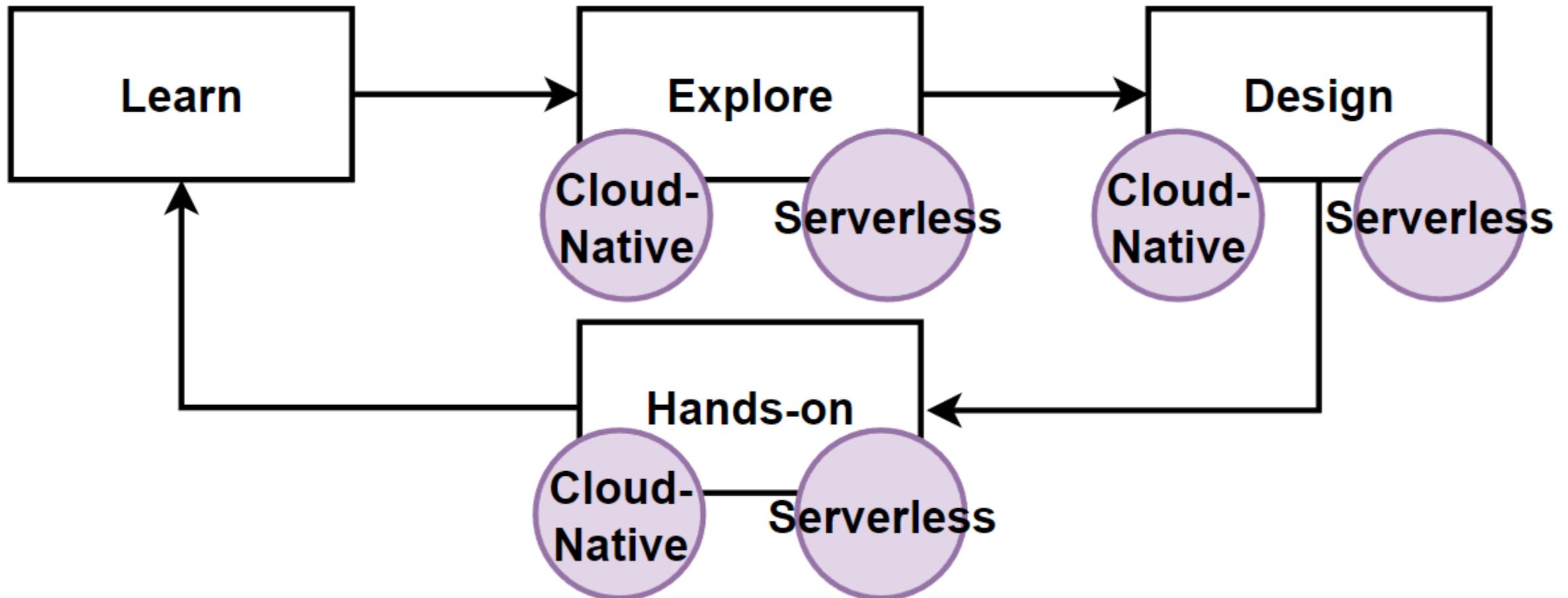
- **Upstash Redis** is a serverless, fully managed, globally distributed Kafka-compatible message broker service.
- It provides a low-latency, high-performance messaging and streaming solution for cloud-native applications.
- It offers features like on-demand scaling, pay-as-you-go pricing, multi-cloud support, and easy integration with other cloud services.
- **Serverless Architecture**
Designed as a serverless service, meaning it automatically scales with the workload and only charges for the resources used, reducing operational complexity and costs.
- **Global Distribution**
Multi-cloud and global distribution, allowing you to deploy your cache closer to your users and improve the performance of your applications.
- **High Availability**
Ensures high availability through data replication and automatic failover mechanisms.
- **Easy Integration**
Seamless integration with popular cloud-native platforms and services, making it easy to incorporate caching into your application architecture.



Way of Learning – The Course Flow



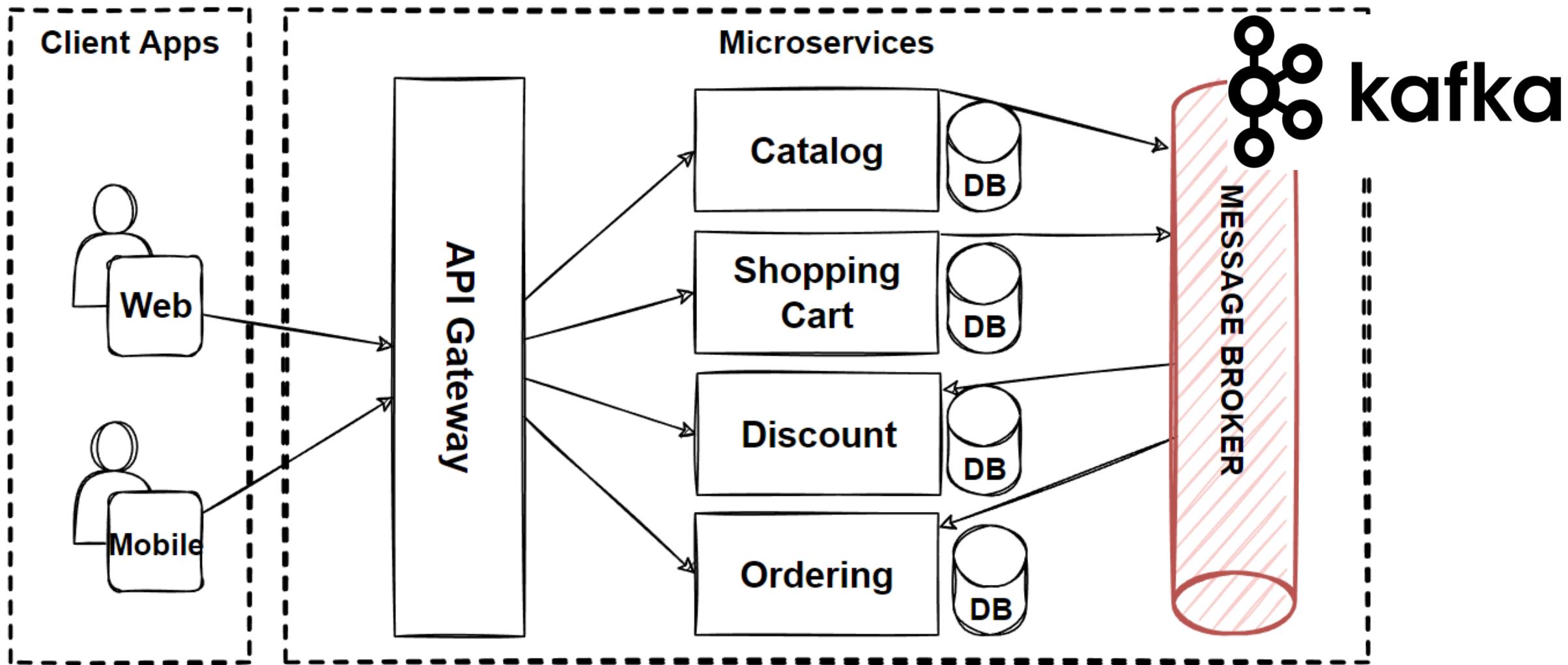
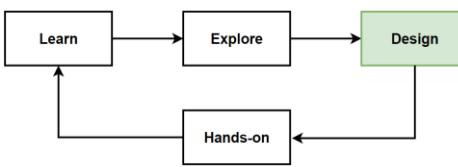
Way of Learning – Cloud-Native & Serverless Cloud Managed

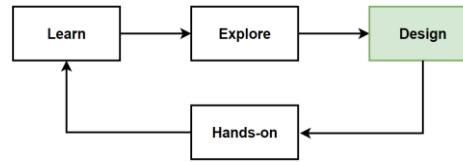


Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

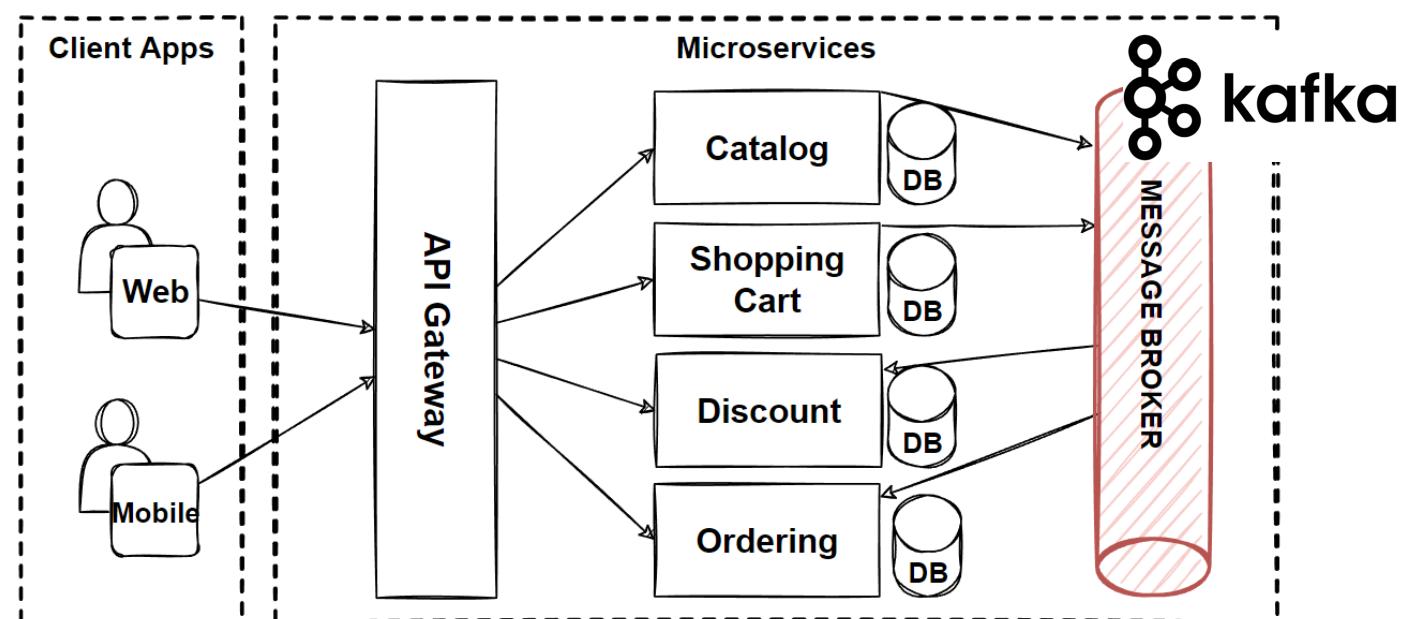
Microservices Distributed Message Brokers with Fan-Out Publish/Subscribe Messaging Pattern



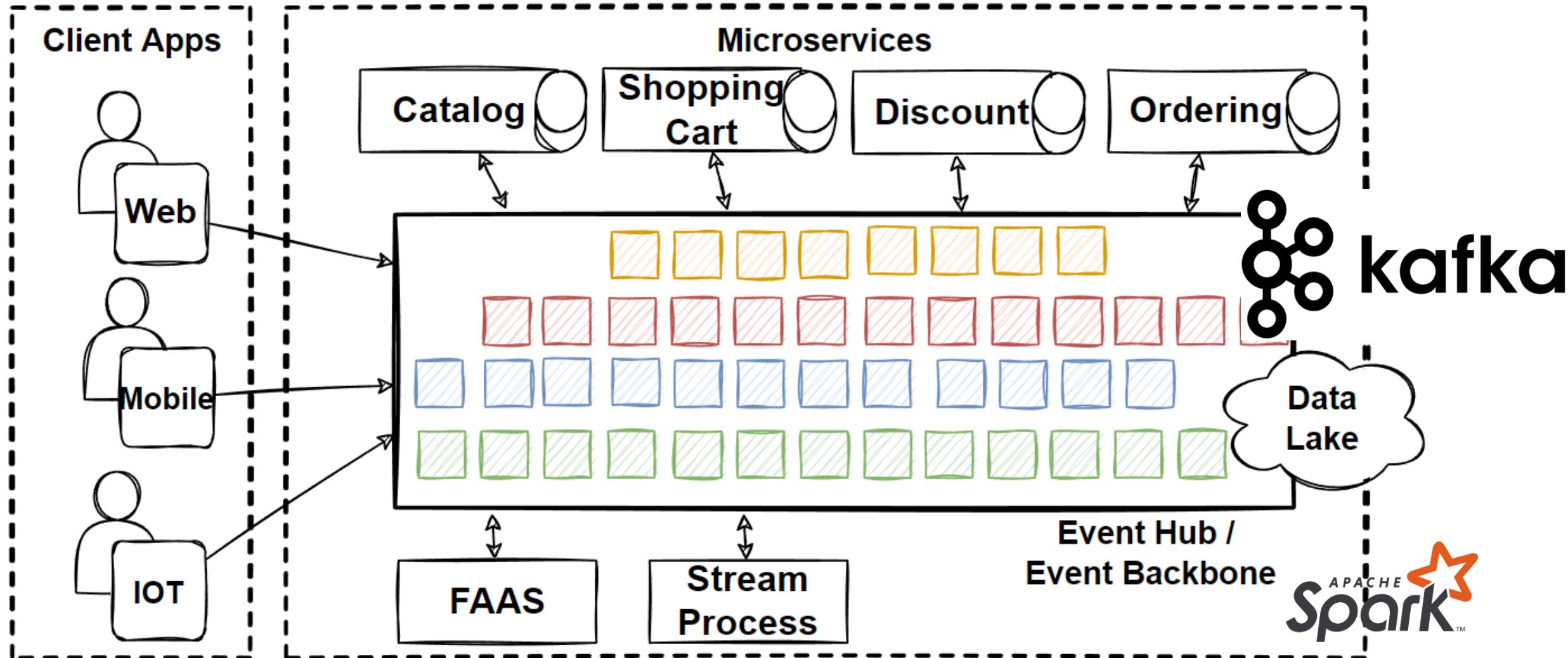
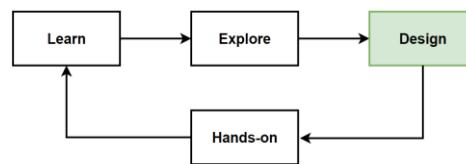


Checkout Shopping Cart Use Case

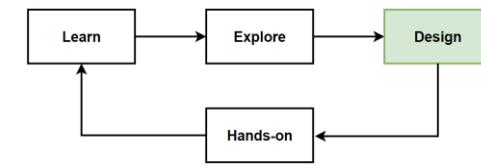
1. A customer **adds products** to their **shopping cart** using the Cart-Service.
2. The **Cart-Service** publishes a **CheckoutEvent** to a Kafka topic. Event includes: customer ID, shopping cart ID, and the list of products in the cart.
3. The **Order-Service** is **subscribed** to this **Kafka topic** and consumes the **CheckoutEvent**. It creates a new order and reserves the products included in the shopping cart.
4. If successful, it publishes an **OrderCreatedEvent** to another Kafka topic and starts to Order fullfilment process.
5. In **Order fullfilment process**, Invetory, Shipment and Payment services consume this event and perform fullfilment actions.



Event-Driven E-commerce Microservices Architecture with Event Hubs



Event-Driven E-commerce Microservices Architecture with Event Hubs

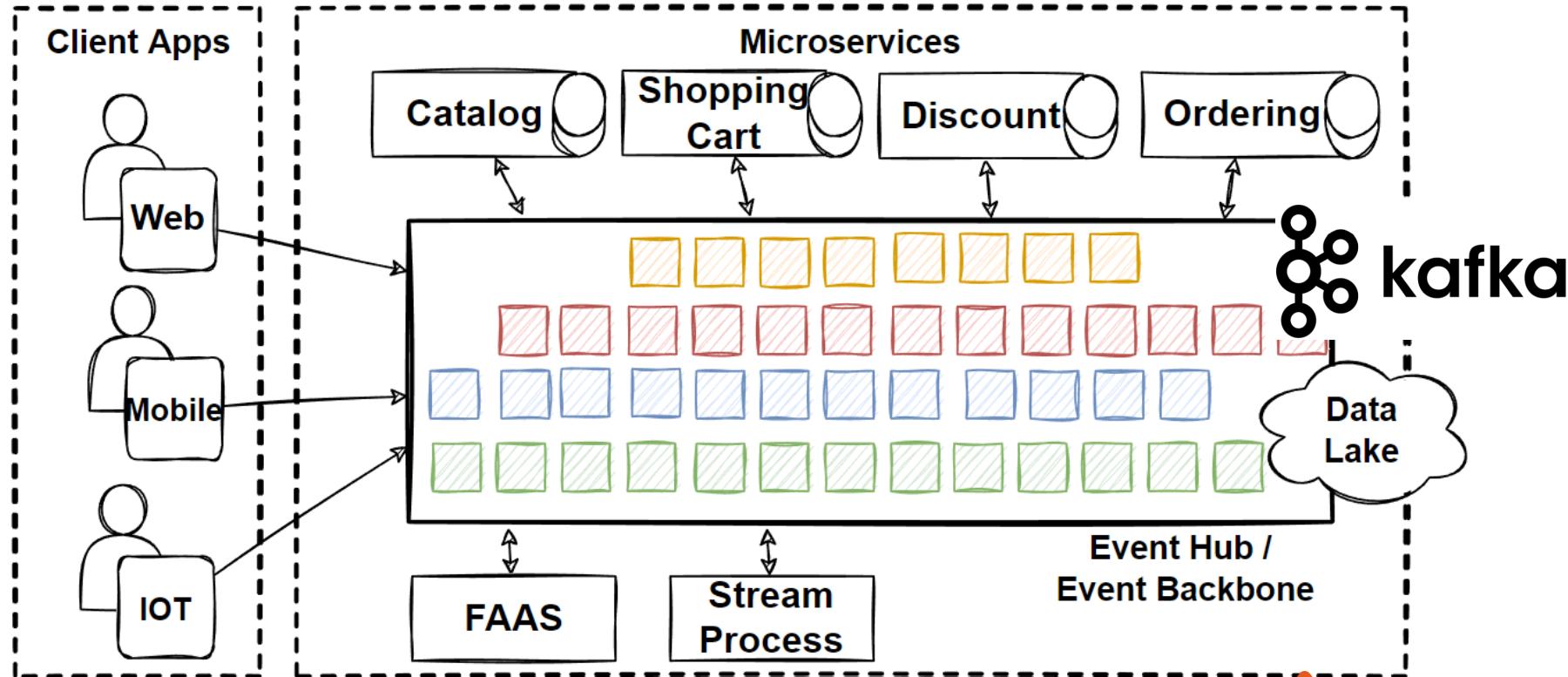


Benefits

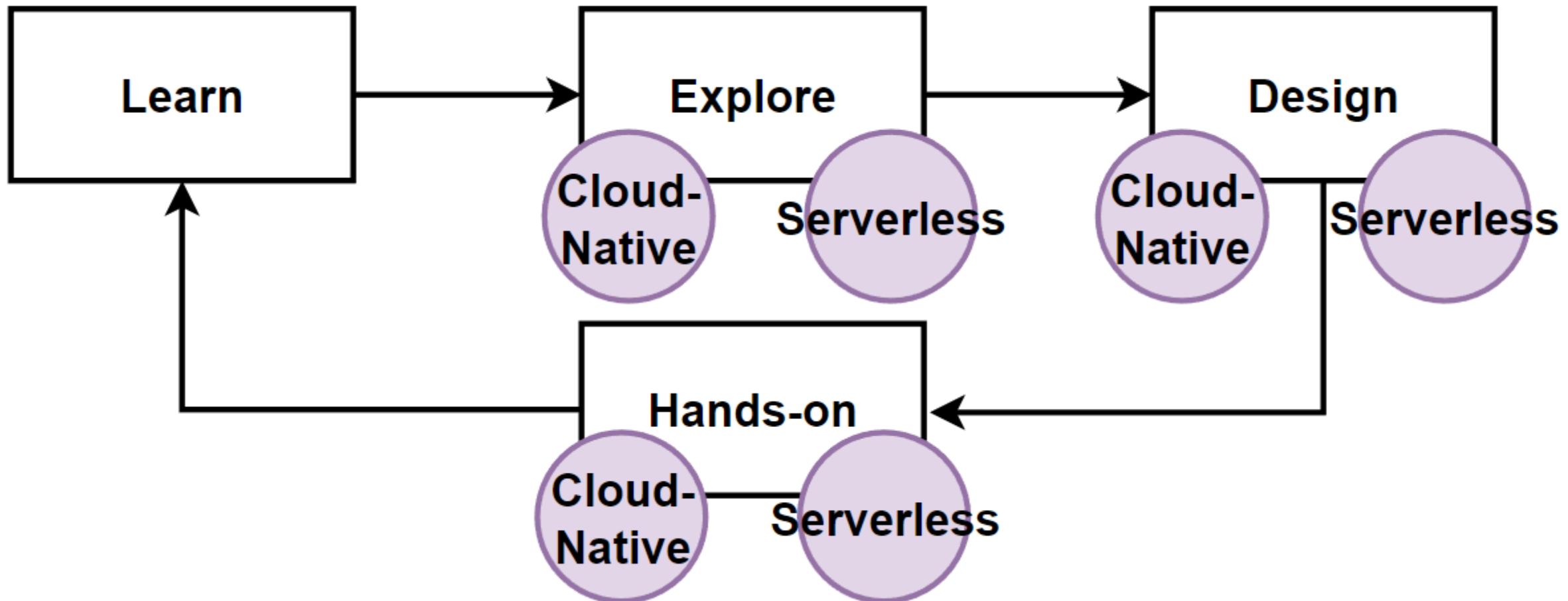
- Event Streaming
- Real-time Processing
- High Volume Events
- Decoupling
- Increased Scalability
- Resilience

Drawbacks

- Increased Complexity
- Debugging
- Latency
- Distributed Transactions

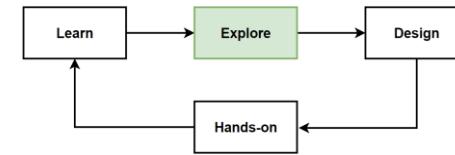


Way of Learning – Cloud-Native & Serverless Cloud Managed



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

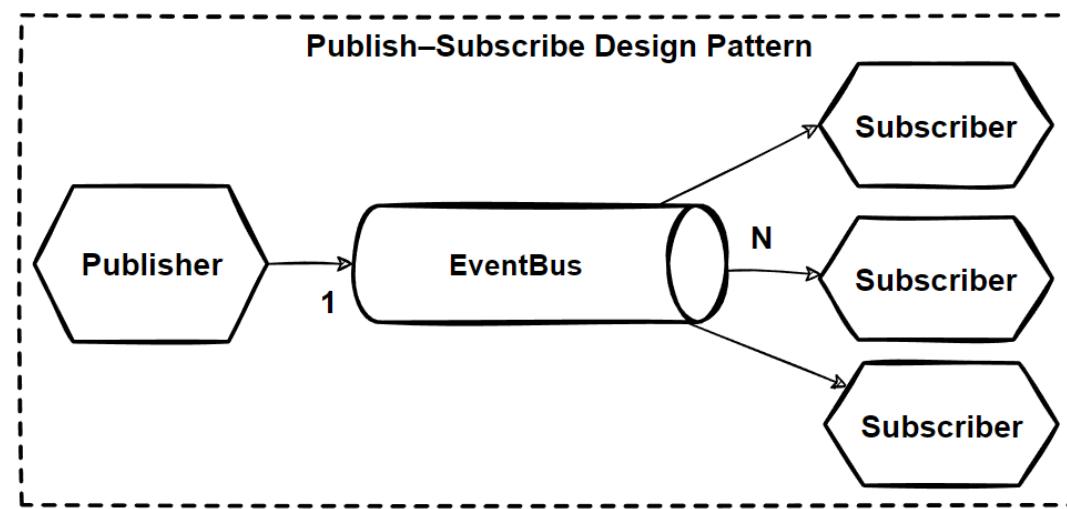


Explore: Cloud Serverless Message Brokers

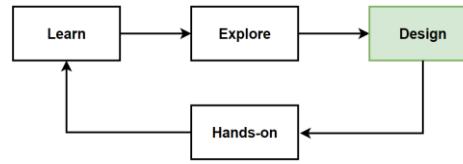
- **Managed serverless message brokers** and event hub services provided by major cloud providers **simplify the process** of setting up and maintaining messaging systems for pub/sub messaging in cloud-native applications.
- Fully managed and **automatically scale to handle the workload**, reducing the operational overhead of managing caching infrastructure.

Cloud Serverless Message Brokers

- [Amazon SNS](#)
- [Amazon EventBridge](#)
- [Amazon Kinesis](#)
- [Azure Service Bus](#)
- [Azure Event Grid](#)
- [Azure Event Hubs](#)
- [Google Cloud Events](#)
- [Google Cloud Pub/Sub](#)
- [Upstash Kafka](#)
- [memphis.dev](#)



Goto -> <https://landscape.cncf.io/>



Amazon EventBridge and Amazon SNS

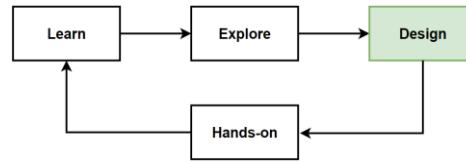
- At AWS, offer Amazon EventBridge to build event buses and Amazon Simple Notification Service (SNS) to build event topics.
- **Amazon EventBridge** is recommended when you want to build an application that reacts to events from SaaS applications, AWS services, microservices or custom applications.
- EventBridge uses a predefined schema for events and allows you to create rules that are applied across the entire event body to filter before pushing to consumers.
- **Amazon SNS** is recommended when you want to build an application that reacts to high throughput and low latency events published by other applications, microservices, or AWS services.
- Or for applications that need very high fanout (thousands or millions of endpoints). SNS topics are agnostic to the event schema coming through.
- Both of these services can decouple microservices and implement publish/subscribe pattern.



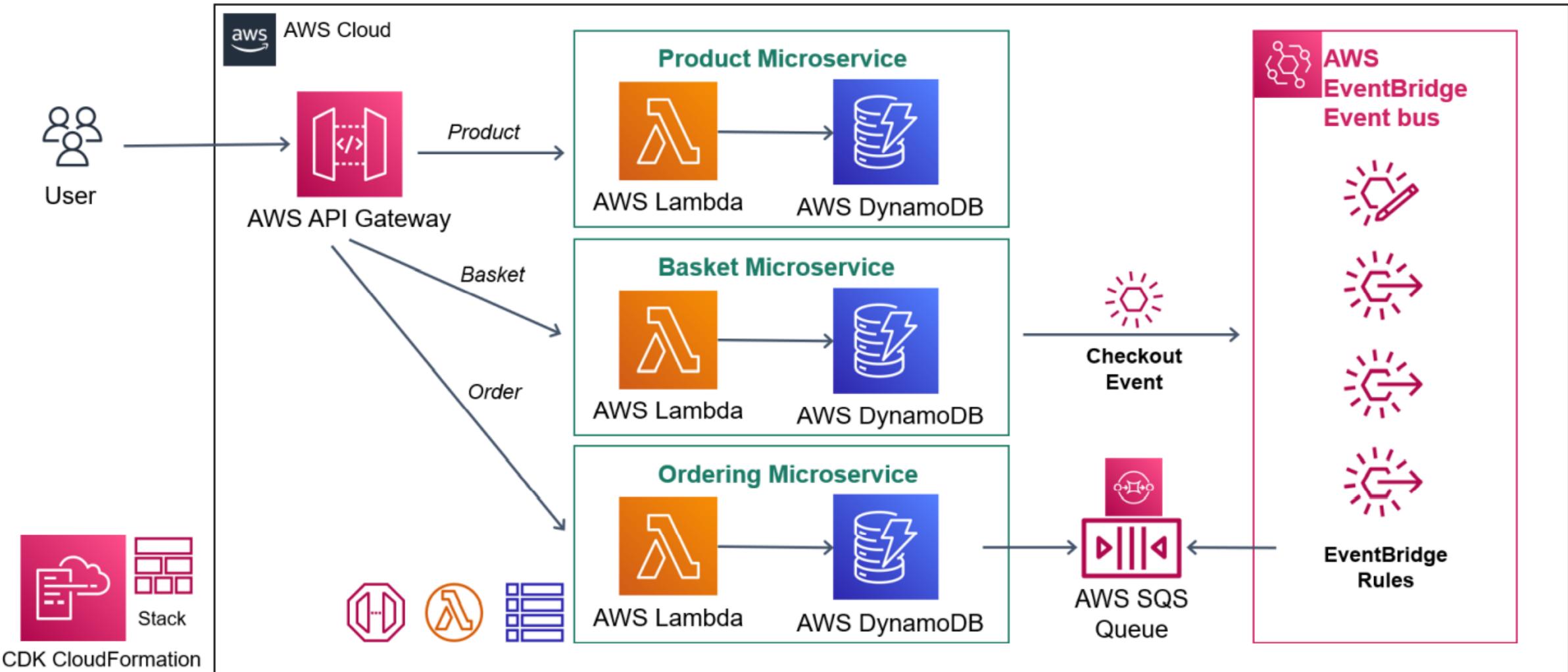
Amazon EventBridge



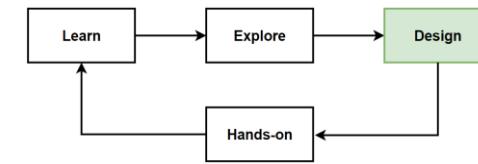
Amazon SNS



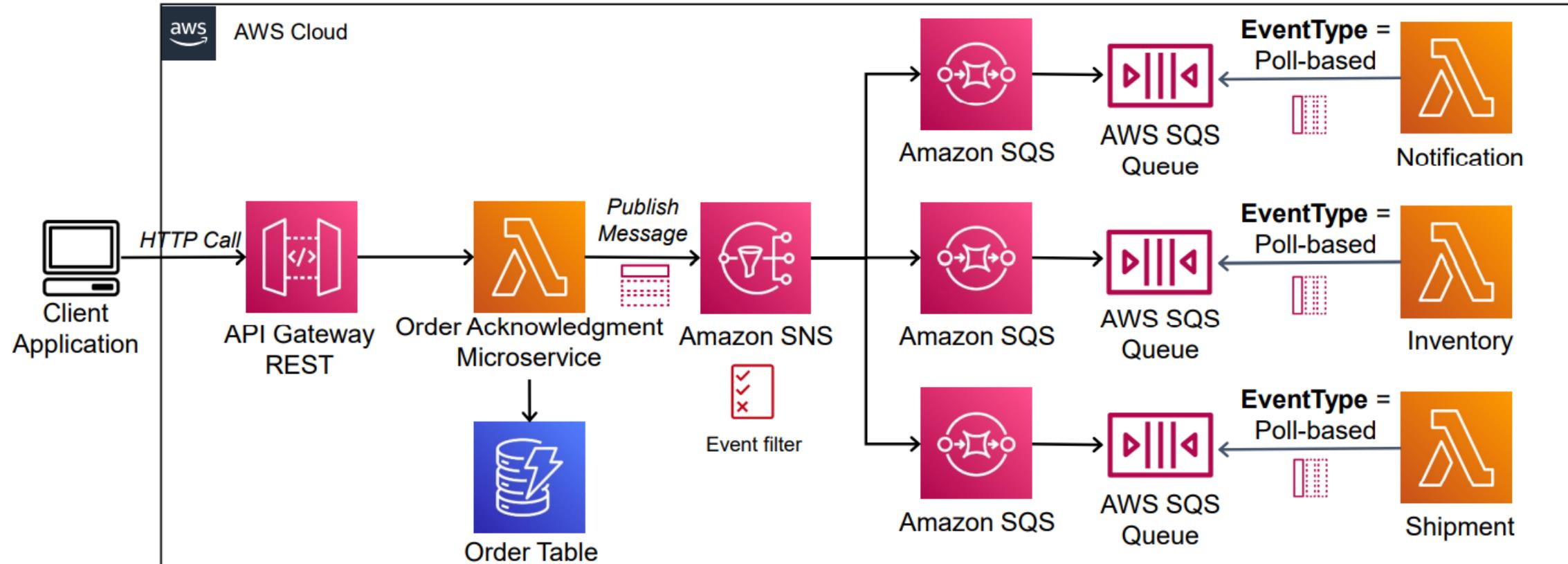
Design with Cloud Event Bus – Amazon EventBridge

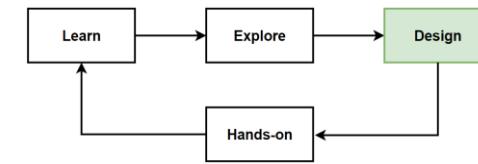


- <https://github.com/awsrun/aws-microservices>
- <https://github1s.com/awsrun/aws-microservices>



Design with Cloud Event Bus – Amazon SNS





Other Cloud Serverless Message Brokers

Azure Service Bus

- Use Azure Service Bus for reliable messaging between services.

Azure Event Grid

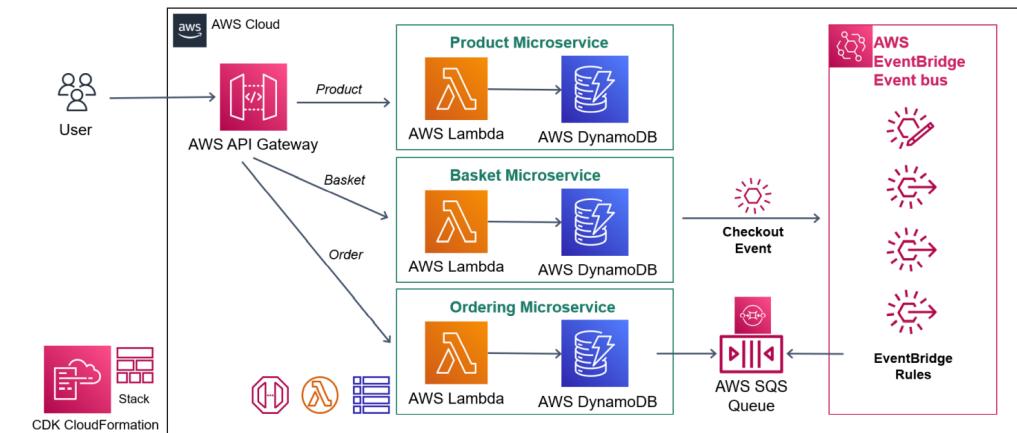
- Use Azure Event Grid for event-driven architectures.

Google Cloud Pub/Sub

- Use Google Cloud Pub/Sub for scalable and reliable messaging between services.

Upstash Kafka

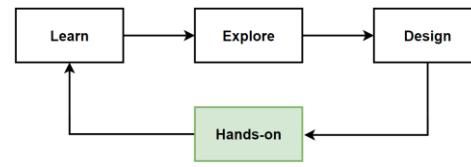
- Use Upstash Kafka for high-throughput event streaming.



BURDASİN burdasin !!

Hands-on: Deploy Kafka Message Broker on a Kubernetes Cluster with Minikube

Leveraging Horizontally Scalable Distributed Cloud-Native Message Broker
Using Helm Charts from Bitnami and deploy Kafka into Minikube

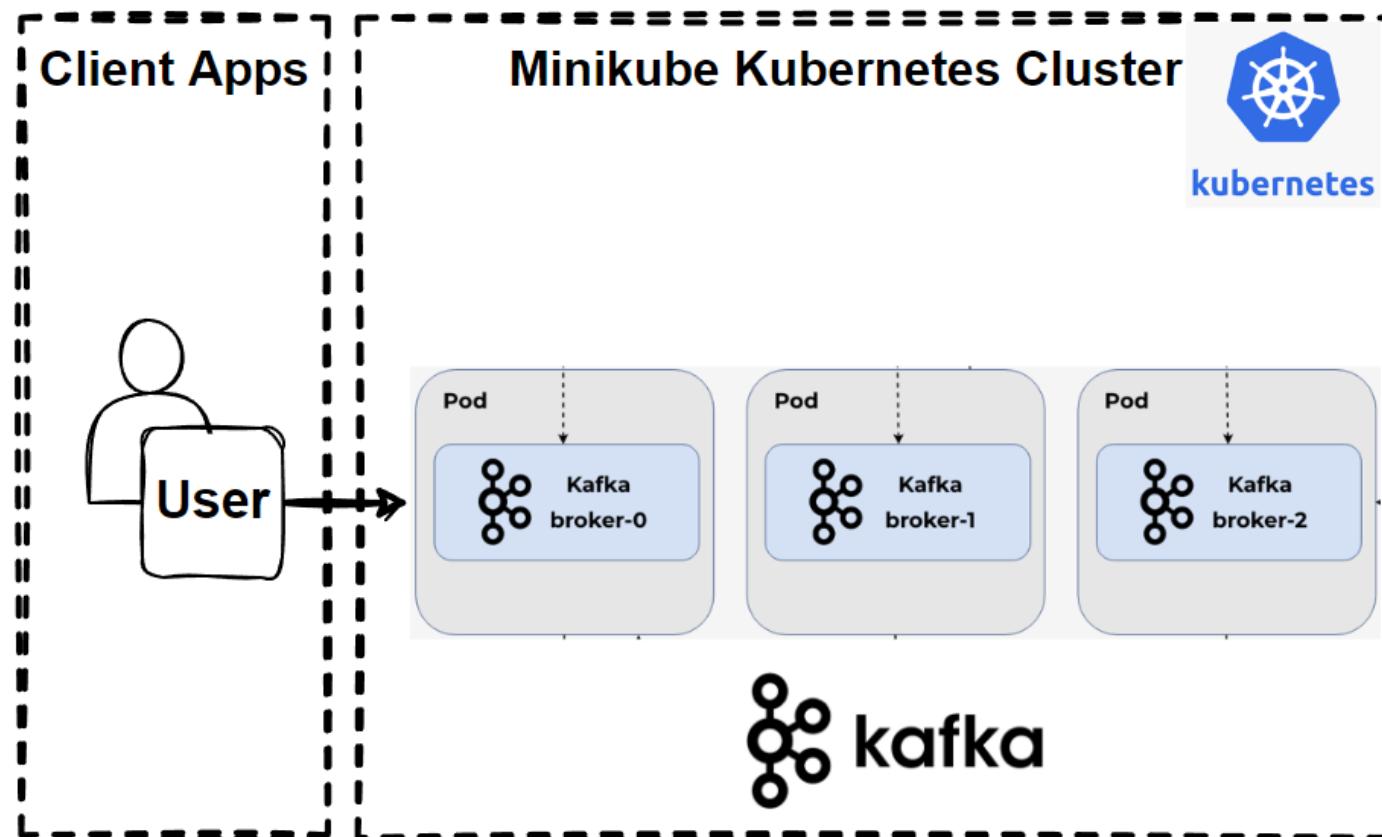


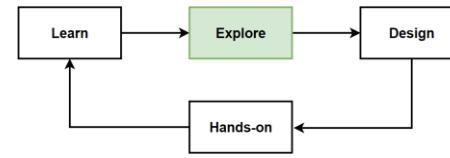
Hands-on: Deploy Kafka in a Single Kubernetes Cluster with Minikube – Task List

- Step 1. Start Kubernetes - minikube start
- Step 2. Start Kafka - Deploy with Bitnami Helm Charts
- Step 3. Use the built-in Kafka client - Publish and Subscribe Topic
- Step 4. Simulate node failure and node scales
- Step 5. Stop the cluster

Kafka Artifact Hub:

<https://artifacthub.io/packages/helm/bitnami/kafka>



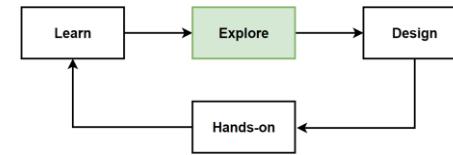


What is Apache Kafka ?

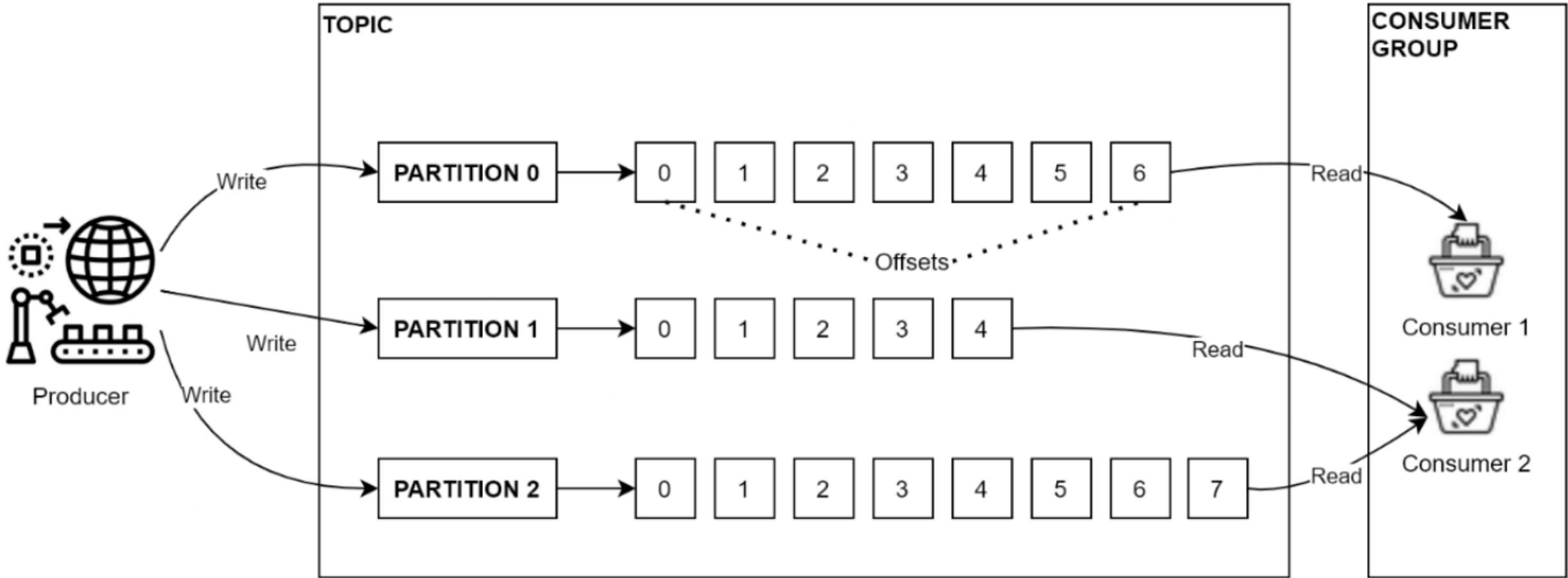
- Apache Kafka is the most popular **open-source event streaming** platforms.
- Designed for **horizontally scalable, distributed, and fault-tolerant**
- Distributed **publish-subscribe event streaming** platform
- Publishers **send** messages to **topics**, **subscriber** of a **topic** receives all the messages published to the topic.
- **Event-driven Architecture:** Kafka is used as an event router, and the microservices publish and subscribe to the events.
- **Distributed working**, and it provides a **horizontal scalable** system.
- Built on top of **ZooKeeper synchronization** service.
- **Key Components:** Topics, Partitions, Brokers, Producer, Consumer, Zookeeper



kafka

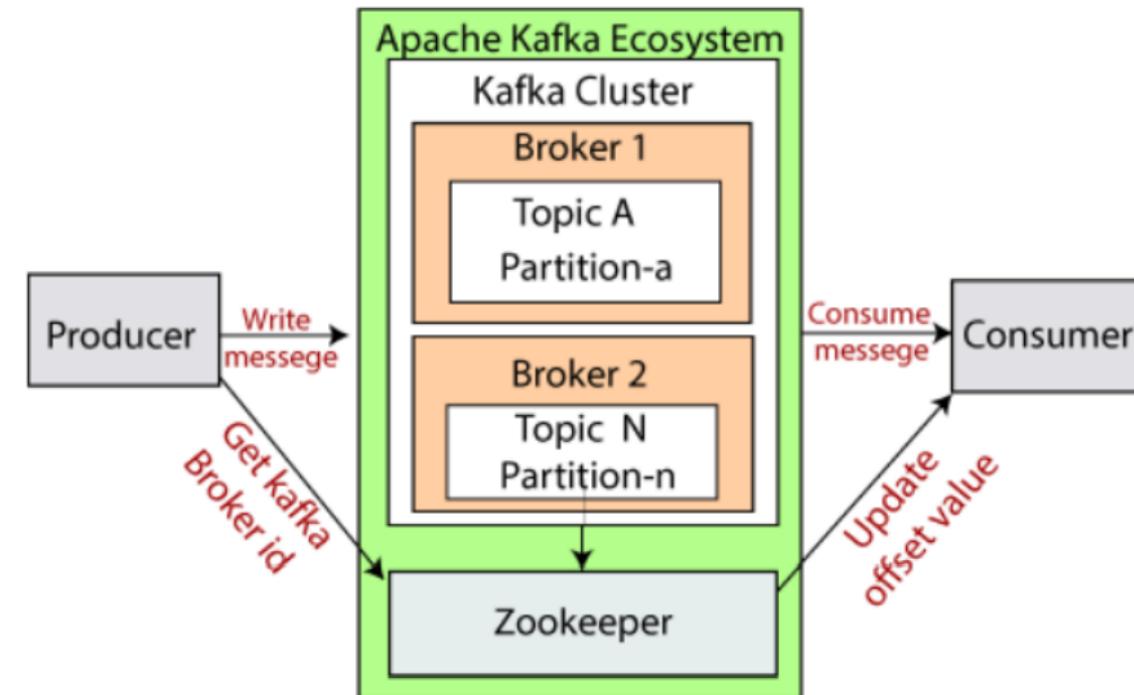
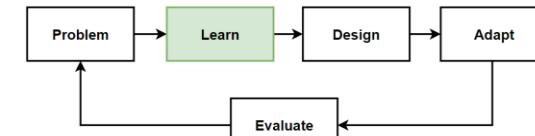


Kafka Components: Topic, Partitions, Offset and Replication Factor

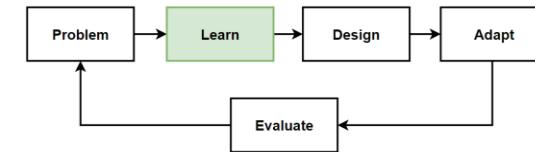


Apache Kafka Cluster Architecture: Kafka Brokers - Kafka Cluster

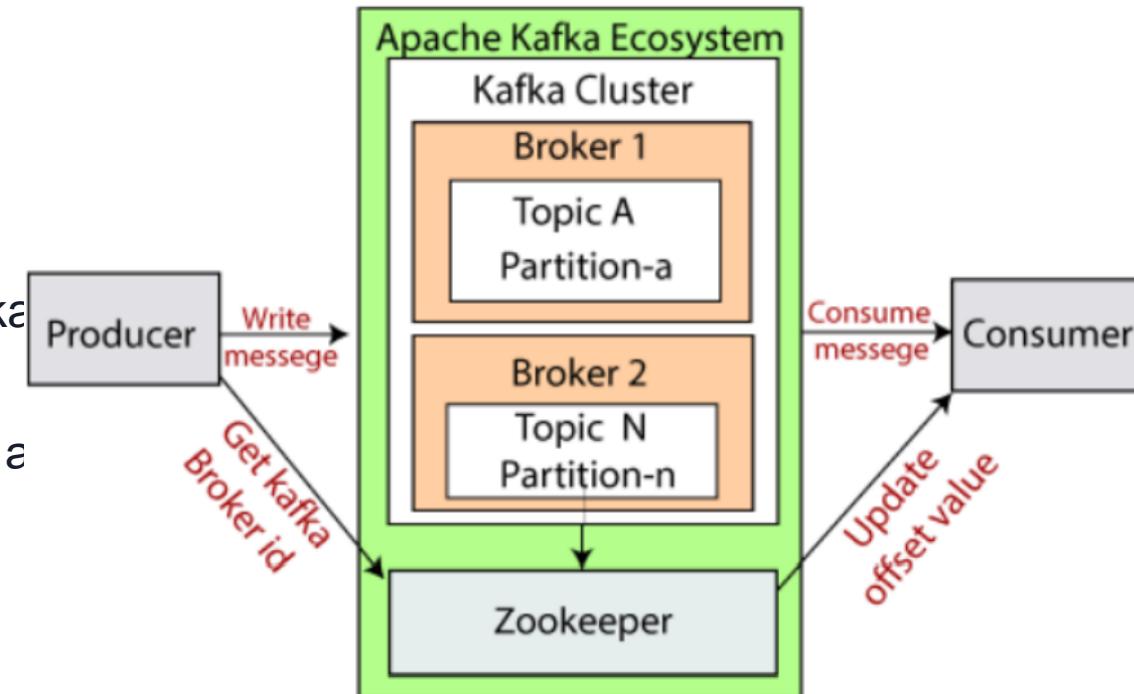
- Kafka can be **set up across multiple servers**, which are called **Kafka Brokers**.
- Mostly prefer to run **multiple Kafka broker instance** at the same time into our **Kafka cluster**.
- With multiple brokers, we can get the benefit of **data replication, fault tolerance, and high availability** of your Kafka cluster.
- **Kafka cluster** consists of multiple brokers to **maintain load balance**.
- Kafka brokers are **stateless**, so they use **ZooKeeper** for maintaining their cluster state.
- **Kafka broker instance** can handle hundreds of **thousands of reads and writes per second**.
- Kafka broker **leader election** can be done by **ZooKeeper**.



Apache Kafka Cluster Architecture: Zookeeper



- How we are managing and coordinating Kafka brokers ?
- Kafka Cluster and including Kafka brokers need to manage.
- **Management job** is handled by the **master node** in the distributed systems.
- This master node **manage** and **maintainances** the **other worker(nodes)** in order to work correctly.
- In Apache Kafka, **there isn't any master** in the Apache Kafka Cluster.
- Apache Kafka Cluster isn't a master-worker architecture; it's a **master-less architecture**.
- **ZooKeeper** is used for **managing** and **coordinating Kafka broker**. The zookeeper manages and maintains the brokers in the cluster.
- Finds which brokers have been **crashed** or which broker recently **added** to the cluster and **manage** their **lifecycle**.



Apache Kafka Core APIs: Producer, Consumer, Streams and Connect API

- **Producers API**

Producers push data to brokers. Publish a stream of data to Kafka topics. When the new broker is started, all the producers search and sends a message to that new broker.

- **Consumer API**

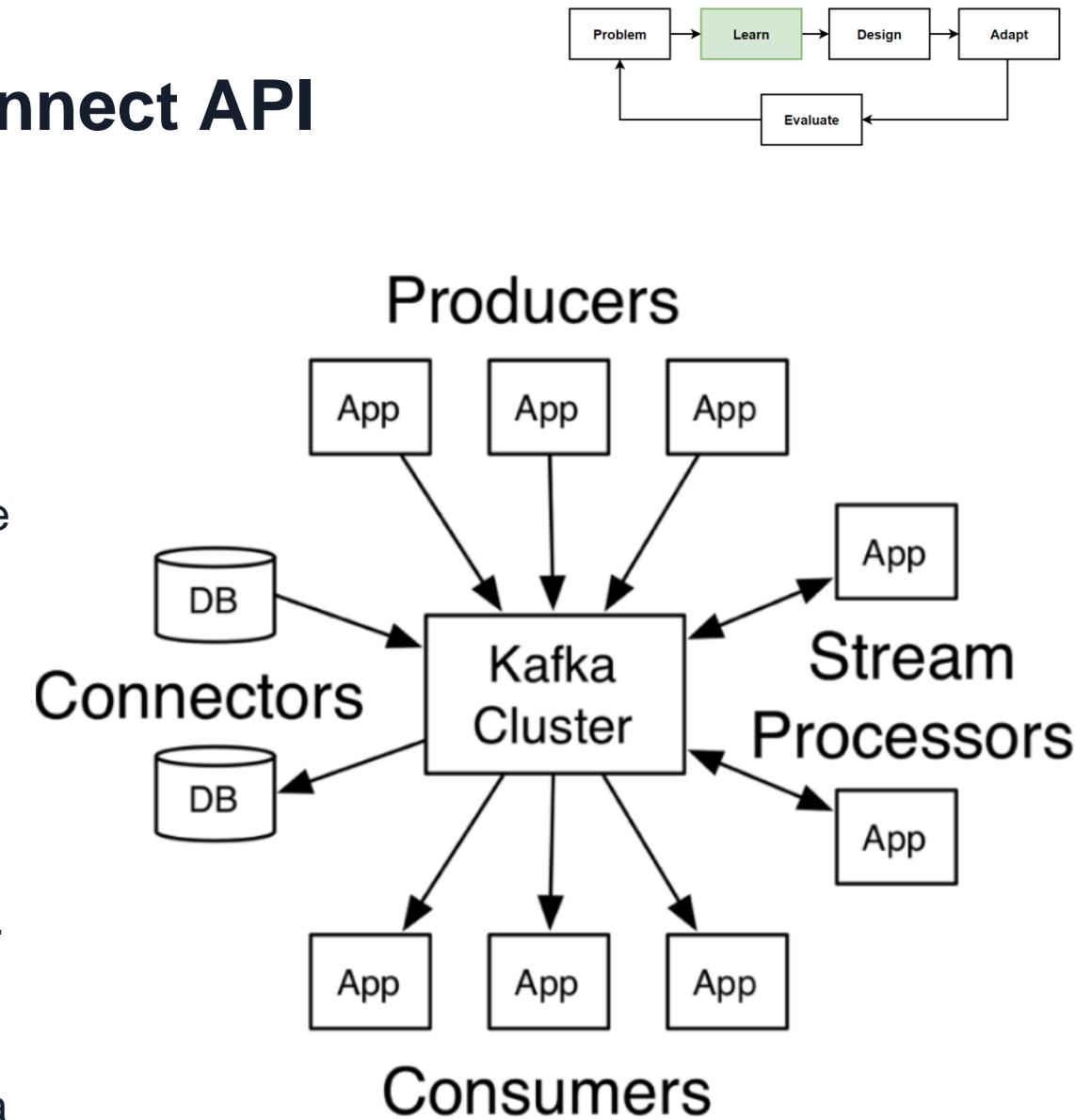
Consumer API provide to subscribe to topics and process the streams of data. Manage messages have been consumed from Kafka by using partition offset. Consumer offset values are managed by ZooKeeper.

- **Streams API**

Streams API that is useful for transforming data from one topic to another. Allows applications to act as a stream processor. Transforming the input streams to output streams.

- **Connect API**

Provide to implement connectors that pulling data from external systems into Kafka. Use stream data between Kafka and other external systems. Collect metrics from servers into Kafka topics.



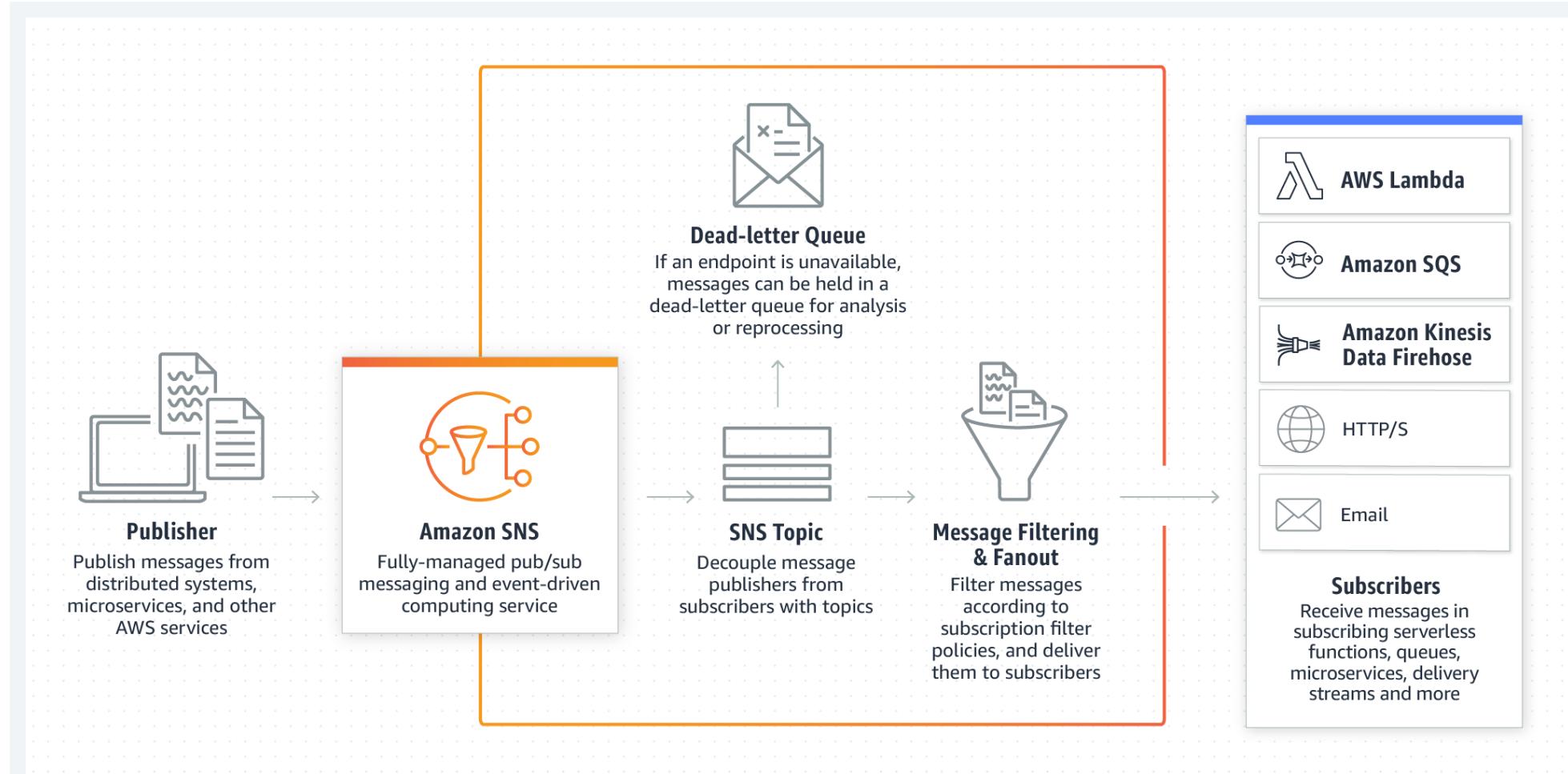
Hands-on: Amazon SNS Notifications Topic Subscribe From AWS Lambda

Developing Hands-on Lab: Amazon SNS Notifications Subscribe From AWS Lambda

Amazon SNS: Fully Managed Pub/Sub Messaging



Application
Integration



<https://aws.amazon.com/sns/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=deschttps%3A%2F%2Fdocs.aws.amazon.com%2Fsns%2Flatest%2Fdg%2Fwelcome.html>

Amazon SNS: Fully Managed Pub/Sub Messaging



- **Application integration**

The Fanout scenario is when a message published to an SNS topic is replicated and pushed to multiple endpoints.

- **Application alerts**

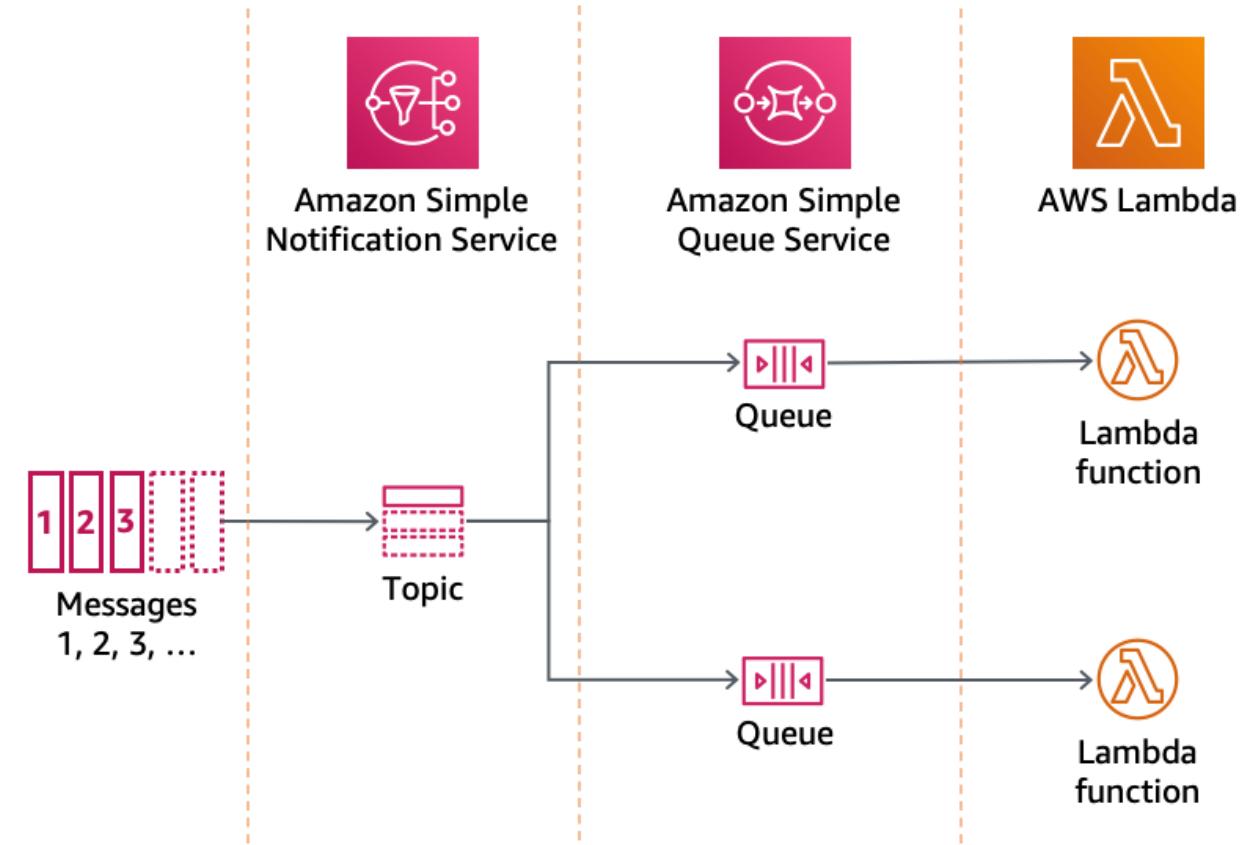
Amazon SNS can send notifications to specified users via SMS and email.

- **User notifications**

Amazon SNS can send push email messages and text messages to individuals or groups.

- **Mobile push notifications**

Mobile push notifications enable you to send messages directly to mobile apps.



Amazon SNS Event Sources and Destinations



Application
Integration

Amazon SNS Event Sources

Application integration services

- EventBridge
- Step Functions

Compute services

- Lambda
- EC2

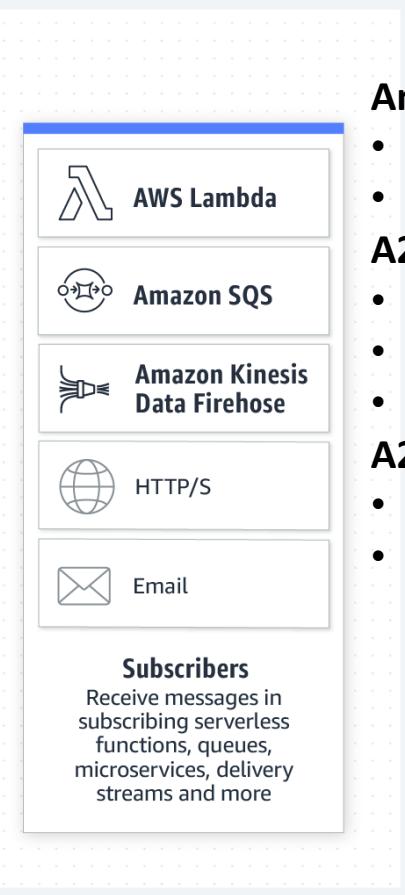
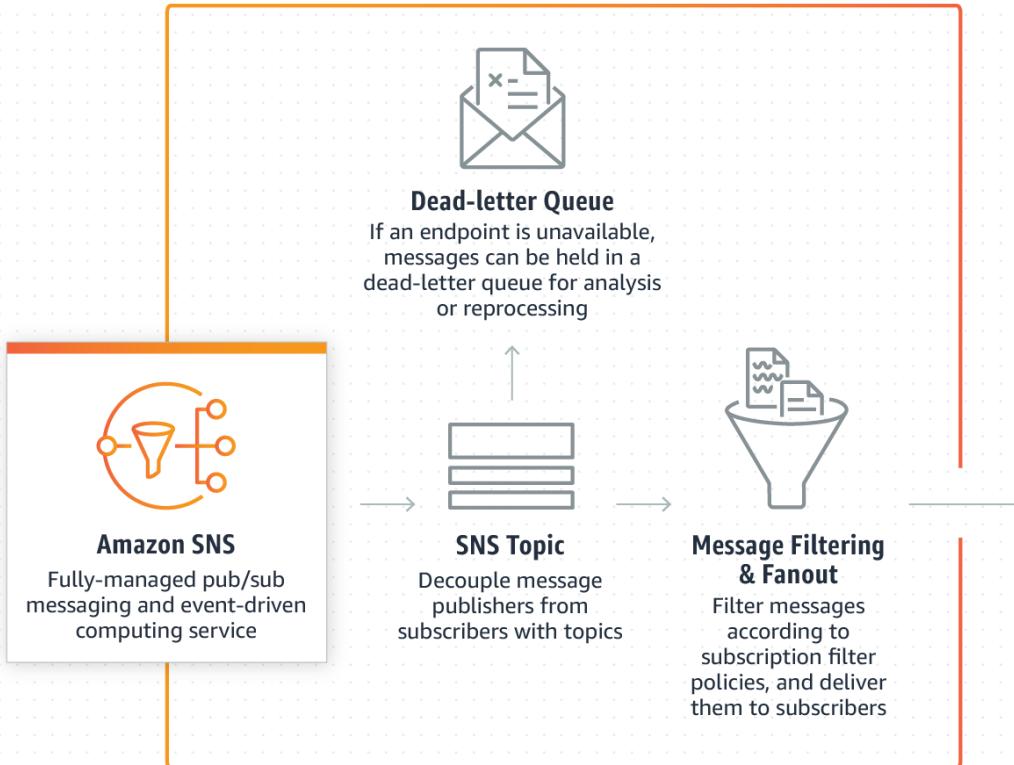
Database services

- DynamoDB
- Elatic Cache



Publisher

Publish messages from distributed systems, microservices, and other AWS services



<https://aws.amazon.com/sns/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=deschttps%3A%2F%2Fdocs.aws.amazon.com%2Fsns%2Flatest%2Fdg%2Fwelcome.html>

Amazon SNS Actions



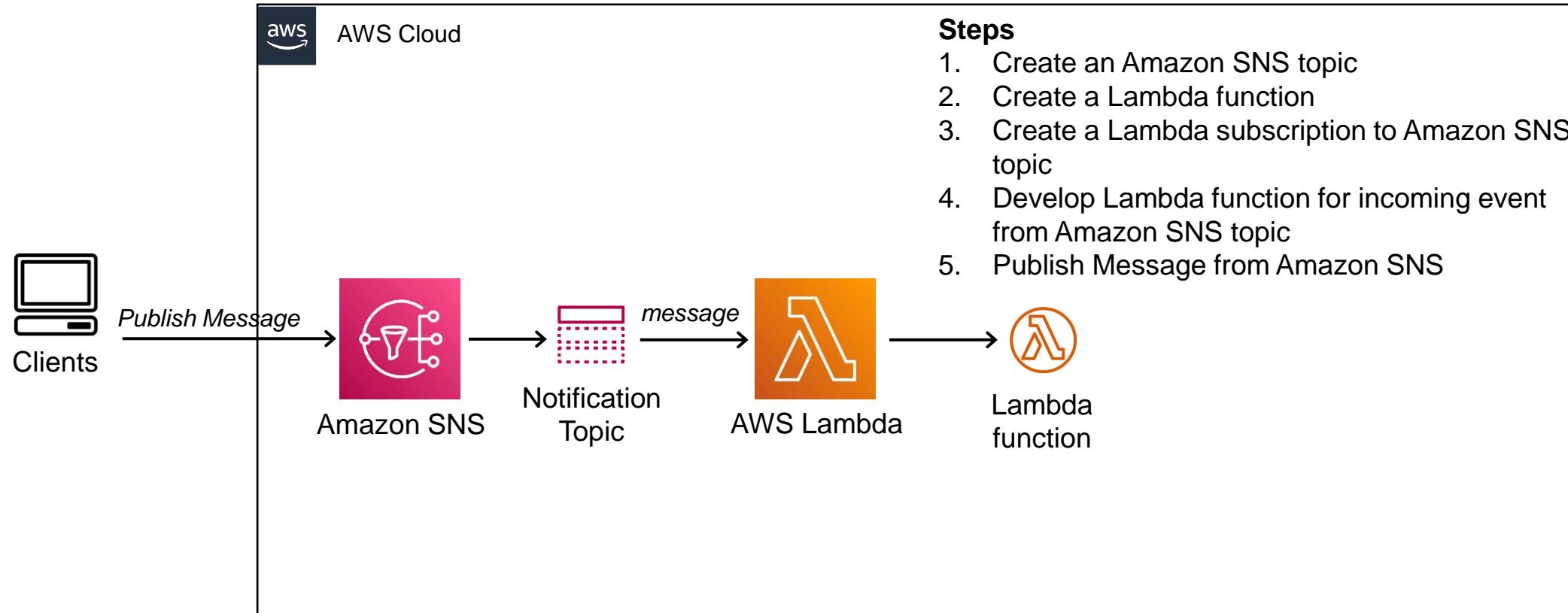
Amazon SNS Actions

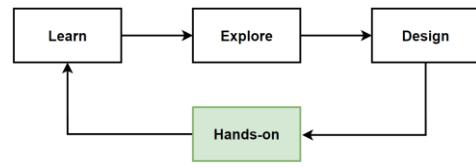
- Create a topic
- Delete a subscription
- Delete a topic
- List topics
- Publish an SMS text message
- Publish to a topic
- Set a dead-letter queue for a subscription
- Set a filter policy
- Set the default settings for sending SMS messages
- Set topic attributes
- Subscribe a Lambda function to a topic
- Subscribe a mobile application to a topic



<https://aws.amazon.com/sns/?whats-new-cards.sort-by=item.additionalFields.postDateTime&whats-new-cards.sort-order=deschttps%3A%2F%2Fdocs.aws.amazon.com%2Fsns%2Flatest%2Fdg%2Fwelcome.html>

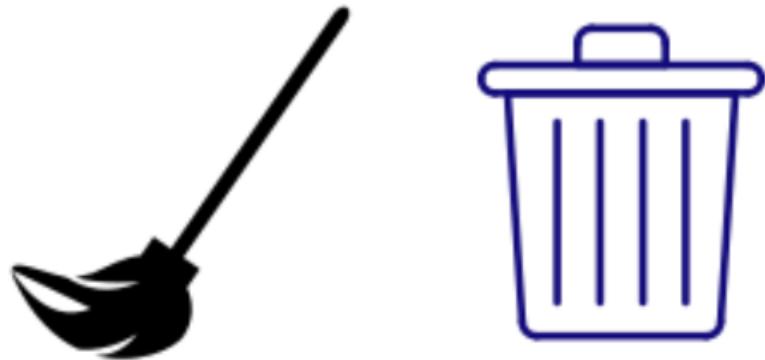
Hands-on Lab: Amazon SNS Notifications Subscribe From AWS Lambda





Clean up Resources

- Delete AWS Resources that we create during the section.



Cloud-Native Pillar6: Scalability: Kubernetes Horizontal Pod Autoscaler (HPA) and KEDA

Kubernetes Horizontal Pod Autoscaler (HPA) and KEDA (Kubernetes Event-Driven Autoscaling)

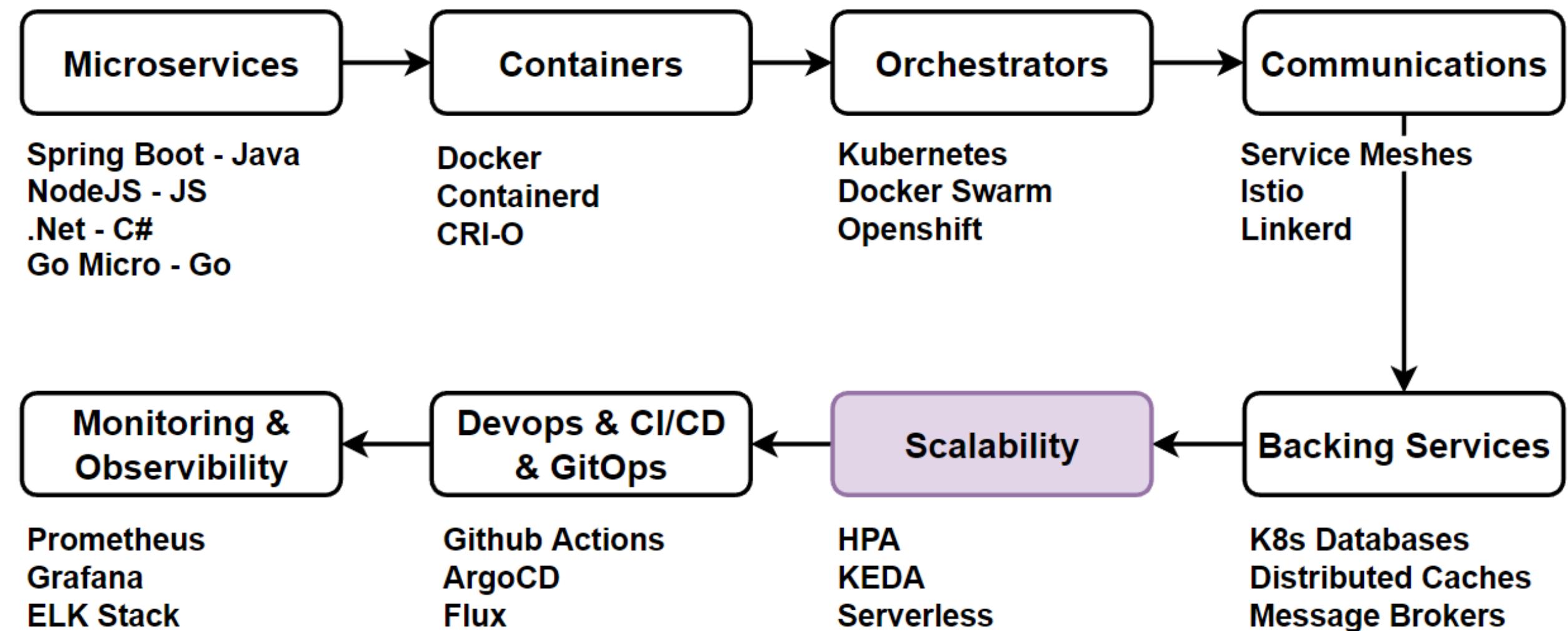
What are Scalability ? - Vertical and Horizontal Scalability

How microservices scale in Cloud-Native environments ?

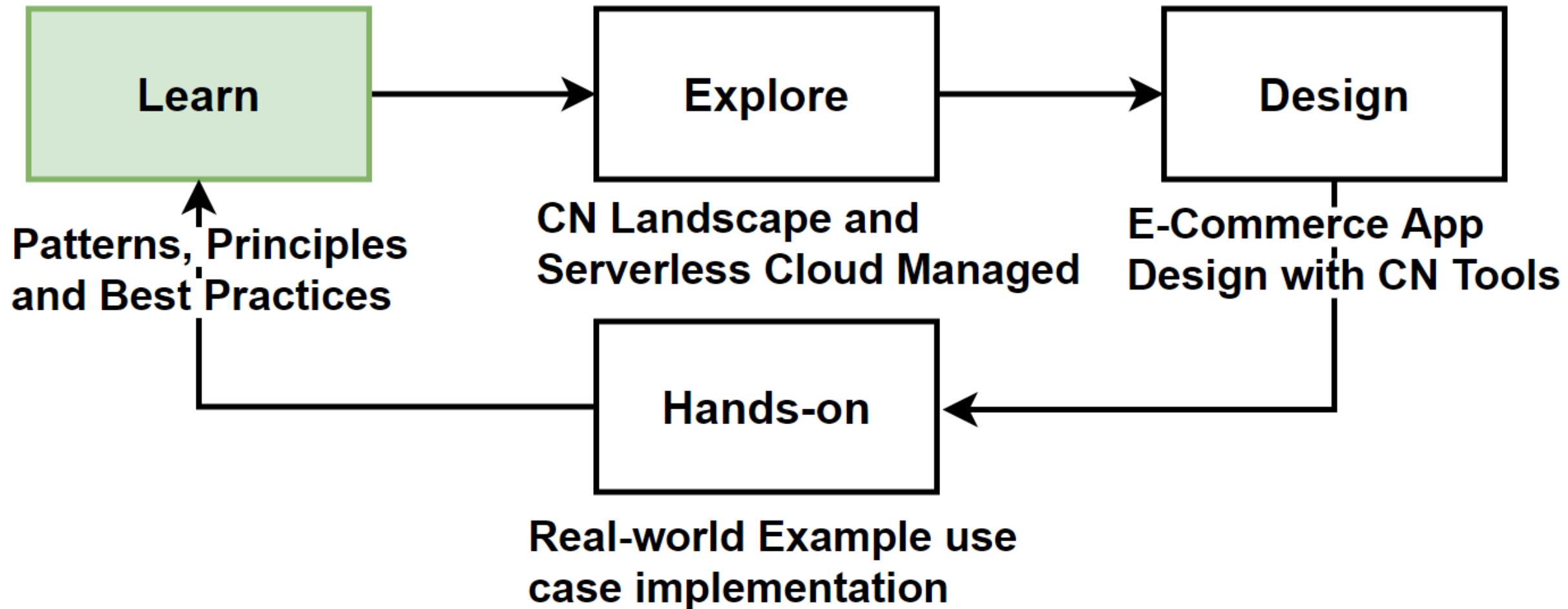
What are patterns & best practices in Cloud-native microservices ?

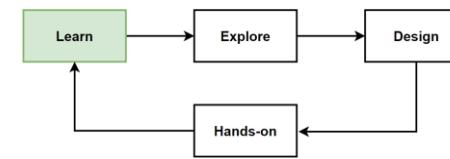
Implement Hands-on Development of Scalability in CN Kubernetes cluster

Cloud-Native Pillars Map – The Course Section Map



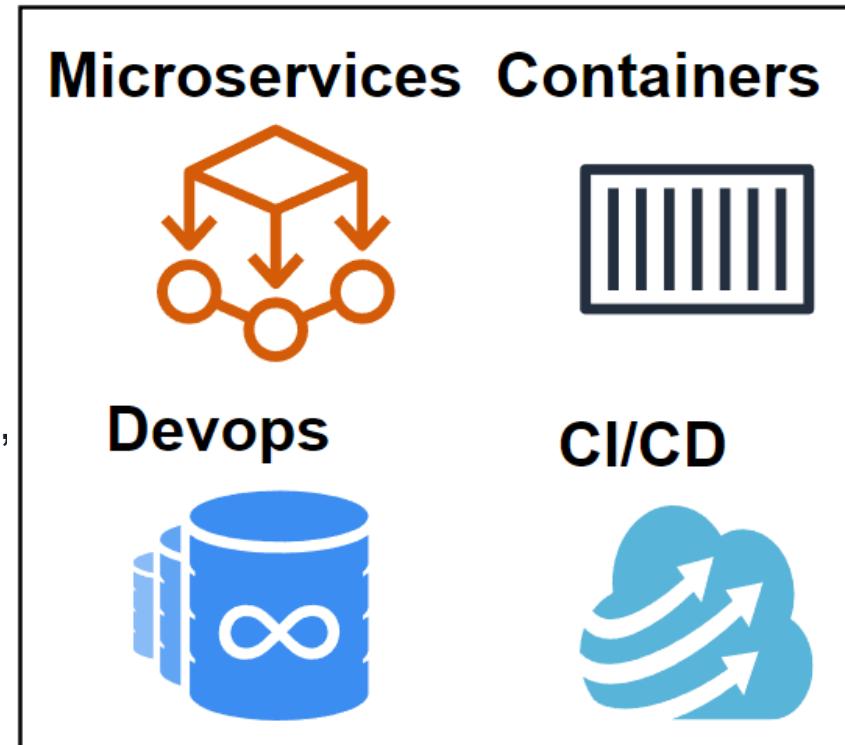
Way of Learning – The Course Flow

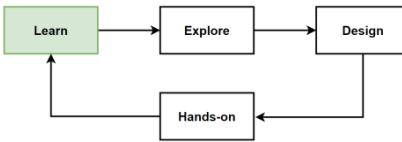




Learn: The 6. Pillar – Cloud-Native Scalability

- What are Scalability ? Why need to Scale ?
- Vertical-Horizontal Scaling, VPA, HPA, Cluster Autoscaler
- Best Practices of Scaling Cloud-native Applications in Kubernetes
- KEDA Event-driven Autoscaling Cloud-native Applications in Kubernetes
- Cloud Serverless Scalability: AWS Fargate, Azure Container Apps, Google CloudRun, Knative, Openshift Serverless
- Hands-on: Cloud-Native Scalability(Vertical-Horizontal Scaling, VPA, HPA, Cluster Autoscaler, KEDA) on a Kubernetes Cluster with Minikube





Where «Scalability» in 12-Factor App and CN Trial Map

Twelve-Factor App

- Codebase
- Dependencies
- Config
- Backing Services
- Build, release and run
- Processes
- Port Binding
- Concurrency
- Disposability
- Development/Prod Parity
- Logs
- Admin Processes



CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape <https://github.com/cncf/landscape> has a growing number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator <https://www.cncf.io/training>

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider <http://cncf.io/ksp>

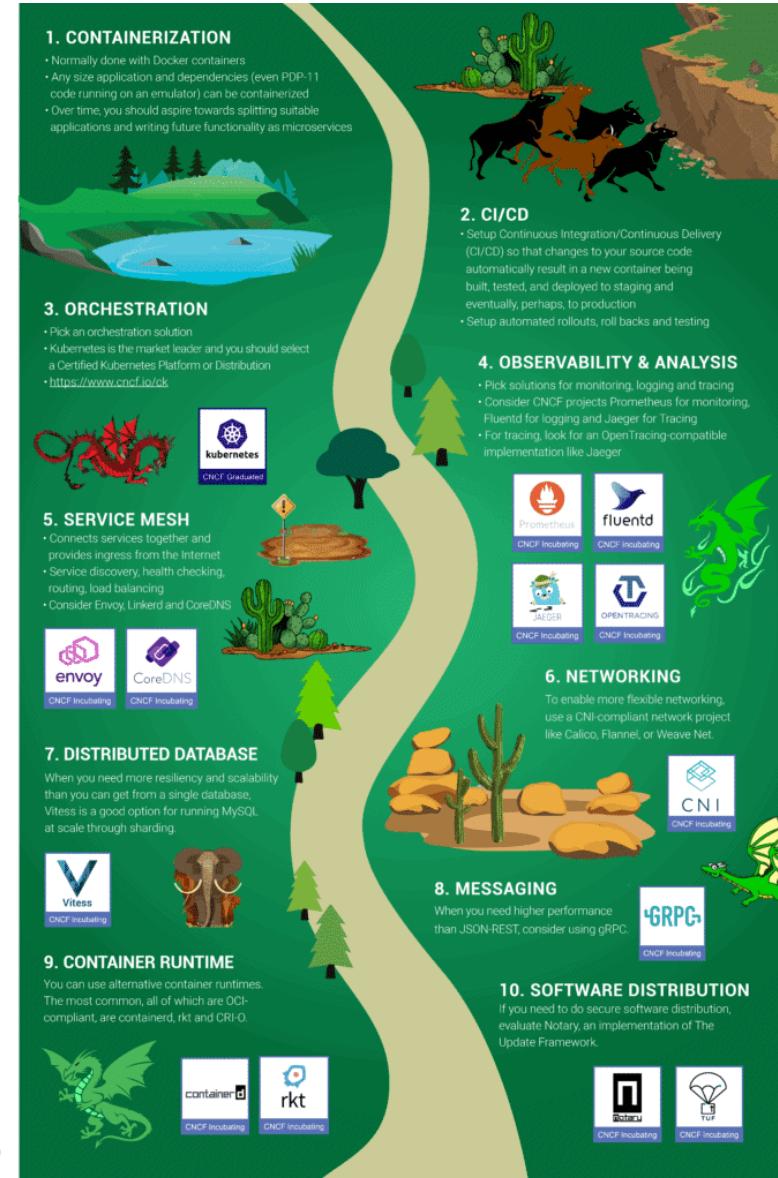
C. Join CNCF's End User Community

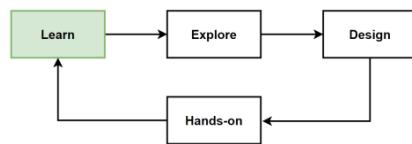
For companies that don't offer cloud native services externally <http://cncf.io/enduser>

WHAT IS CLOUD NATIVE?

- **Operability:** Expose control of application/system lifecycle.
- **Observability:** Provide meaningful signals for observing state, health, and performance.
- **Elasticity:** Grow and shrink to fit in available resources and to meet fluctuating demand.
- **Resilience:** Fast automatic recovery from failures.
- **Agility:** Fast deployment, iteration, and reconfiguration.

www.cncf.io
info@cncf.io





12-Factor App – Cloud-Native Scalability

VI. Processes

- Execute the app as one or more stateless processes.
- Stateless applications can be easily scaled horizontally.
- Rely on horizontal scaling to add or remove instances of an app based on demand.

VIII. Concurrency

- Scale out via the process model. Using independent processes to handle different tasks, allowing for better resource utilization and easier scaling.
- Design to scale out rather than up, it supports the use of autoscaling solutions like HPA and KEDA.

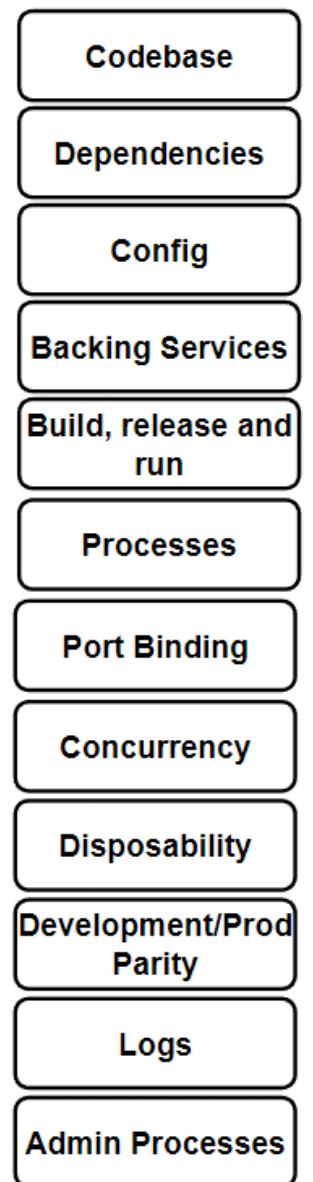
IX. Disposability

- Maximize robustness with fast startup and graceful shutdown.
- App can start quickly and shut down gracefully enables autoscalers like HPA and KEDA to rapidly scale your app up or down in response to changing demand.

X. Dev/prod parity

- Keep development, staging, and production environments as similar as possible.
- Simplifies the process of deploying autoscaling configurations and monitoring the behavior of autoscaled applications in production.

Twelve-Factor App



Cloud-native Trial Map – Scalability

Containerization

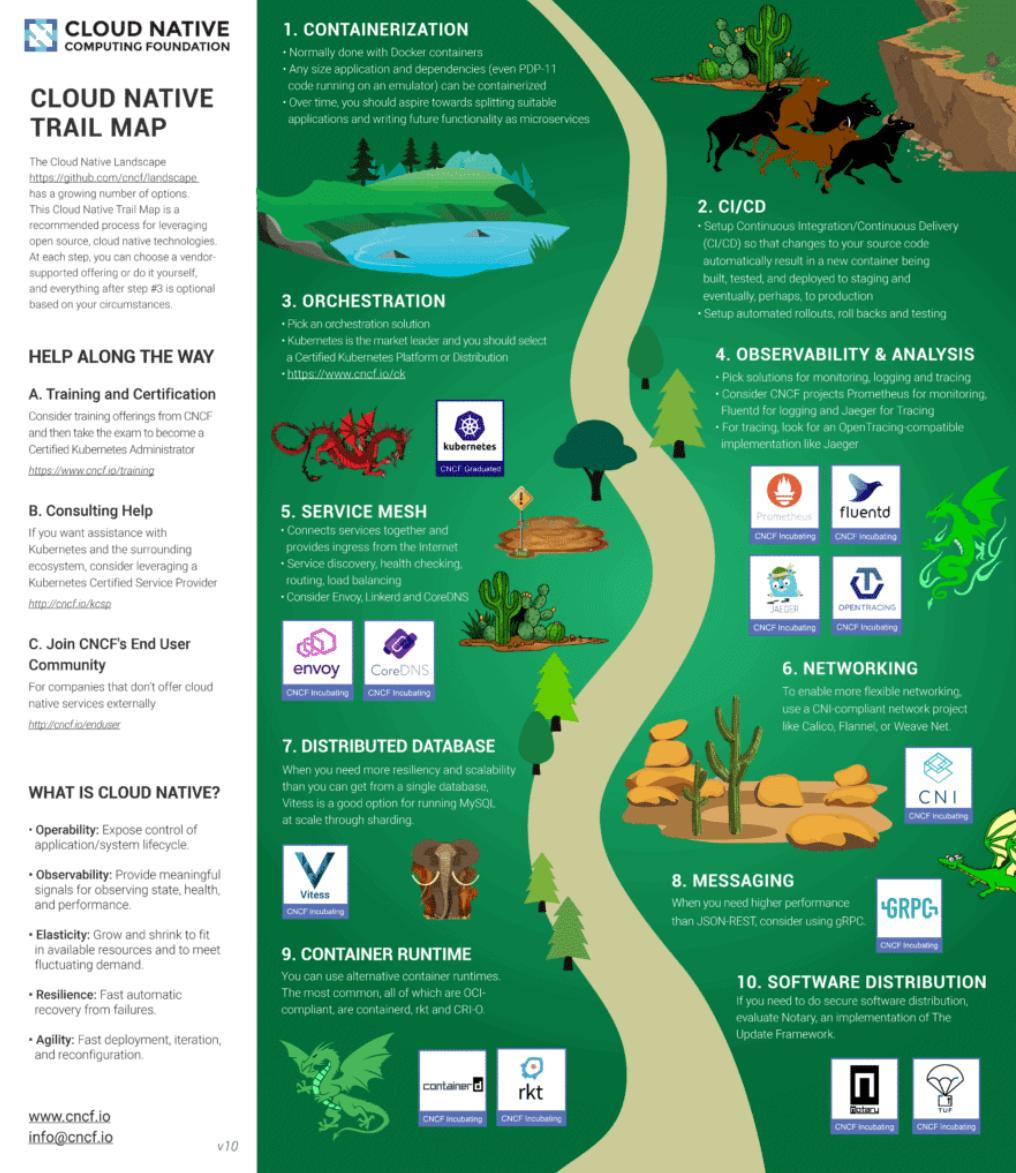
- Package applications enables better resource utilization and makes it easier to scale applications horizontally.
- Docker and containerd facilitate the containerization process, essential for autoscaling solutions like HPA and KEDA.

Orchestration & Application Definition

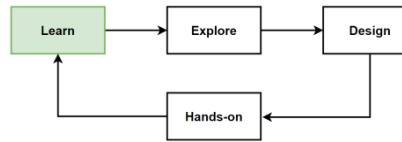
- Kubernetes provides built-in support for HPA, while KEDA is a CNCF sandbox project that extends Kubernetes for event-driven autoscaling.
- Both HPA and KEDA rely on Kubernetes for managing the lifecycle of your apps and scaling them based on various metrics.

Observability & Analysis

- Monitoring, logging and tracing help teams understand the decisions about when and how to scale.
- Tools like Prometheus, Grafana, Fluentd, and Jaeger provide visibility into application behavior, which can help you fine-tune autoscaling configurations and respond to issues more effectively.



<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>



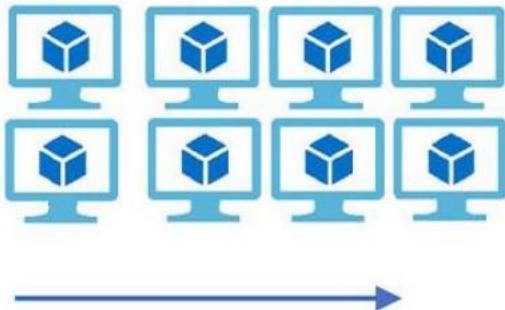
Scalability - Vertical Scaling - Horizontal Scaling

- **Scalability** is the **number of requests** an application can handle
- Measured by the number of requests and it can effectively support **simultaneously**.
- If no longer handle any more simultaneous requests, it has reached its **scalability limit**.
- To prevent downtime, and reduce latency, you must scale
- **Horizontal scaling** and **vertical scaling** both involve adding resources to your computing infrastructure
- **Horizontal scaling** by adding more machines
- **Vertical scaling** by adding more power

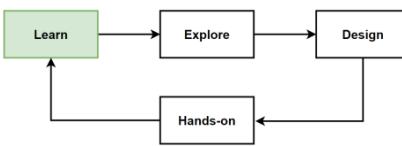
Vertical Scaling
(Increase size of instance (RAM , CPU etc.))



Horizontal Scaling
(Add more instances)



<https://www.webairy.com/horizontal-and-vertical-scaling/>

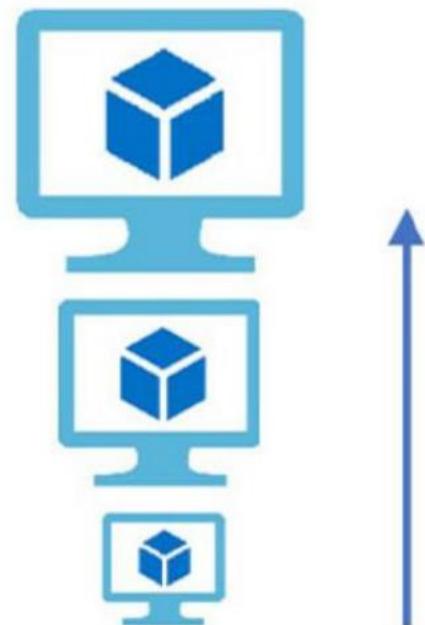


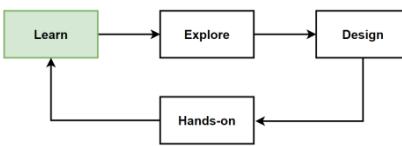
Vertical Scaling - Scale up

- **Vertical scaling** is basically makes the **nodes stronger**.
- Make the server stronger with **adding more hardware**. Adding more resources to a **single node**.
- Make **optimization** the hardware that will allow you to **handle more requests**.
- Vertical scaling keeps your existing infrastructure but **adding more computing power**.
- Your existing **code doesn't need to change**.
- Adding additional **CPU, RAM, and DISK** to cope with an increasing workload.
- By scaling up, you increase the capacity of a single machine. And it has limits. That is named **Scalability Limits**.
- Because even the hardware has **maximum capacity limitations**.
- For handling millions of request, we **need horizontal scaling** or scaling out.

Vertical Scaling

(Increase size of instance (RAM , CPU etc.))



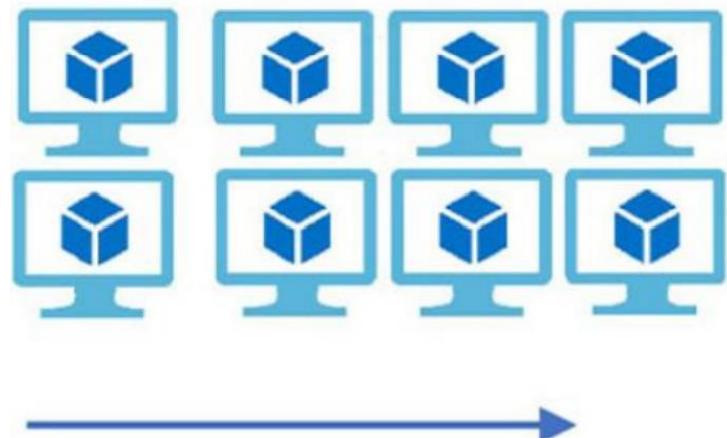


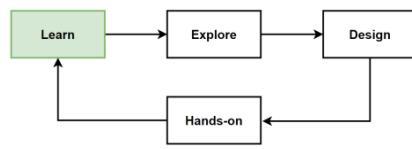
Horizontal Scaling - Scale out

- **Horizontal scaling is splitting the load between different servers.**
- Simply **adds more instances** of machines without changing to existing specifications.
- **Share the processing power and load balancing** across multiple machines.
- Horizontal scaling means adding more machines to the resource pool.
- **Scaling horizontally** gives you scalability but also **reliability**.
- Preferred way to scale in **distributed architectures**.
- When splitting into multiple servers, we need to consider if you have a **state or not**.
- if we have a state like database servers, than we need to manage more considerations like **CAP theorem**.

Horizontal Scaling

(Add more instances)





Scaling Cloud-native Applications in Kubernetes

- Vertical Scaling, Horizontal Scaling, Vertical Pod Autoscaler (VPA), Horizontal Pod Autoscaler (HPA), Cluster Autoscaler, Kubernetes Event-Driven Autoscaling (KEDA)

- **Vertical Scaling**

Increase the resources (CPU and memory) allocated to your app. This can be done by updating the resources field in your container specifications in your deployment configuration.

- **Horizontal Scaling**

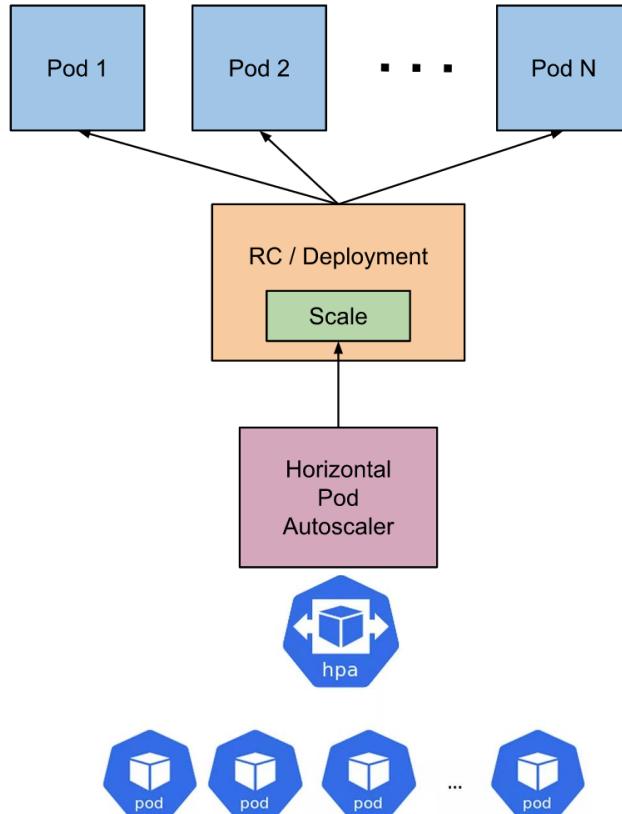
Increase the number of instances (pods) of your app to handle more load. To manually scale the number of replicas, you can use the kubectl scale command or update the replicas field in your deployment configuration.

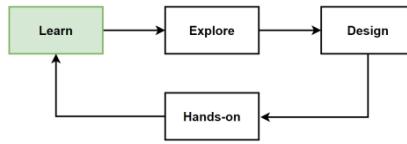
- **Vertical Pod Autoscaler (VPA)**

VPA adjusts the resources (CPU and memory) allocated to a pod based on its usage. Improve resource utilization and prevent overprovisioning or underprovisioning of resources.

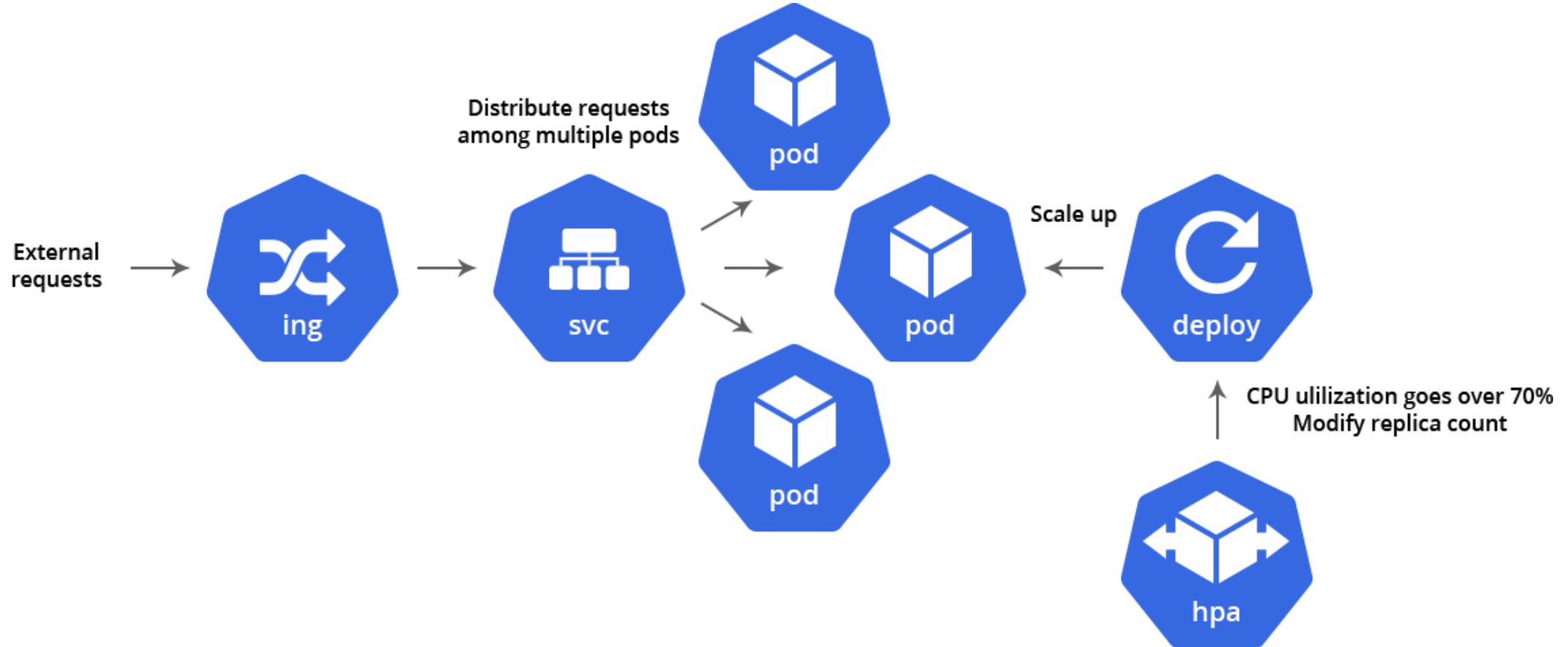
- **Horizontal Pod Autoscaler (HPA)**

HPA automatically scales the number of pod replicas based on predefined resource utilization (CPU, memory) or custom metrics. To use HPA, create a HorizontalPodAutoscaler resource that targets your deployment and specifies the desired CPU utilization or custom metric thresholds.

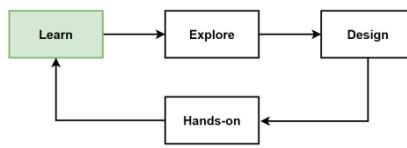




Horizontal Pod Autoscaler (HPA) Scaling in Kubernetes



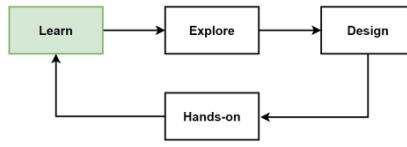
<https://www.virtuozzo.com/company/blog/scaling-kubernetes/>



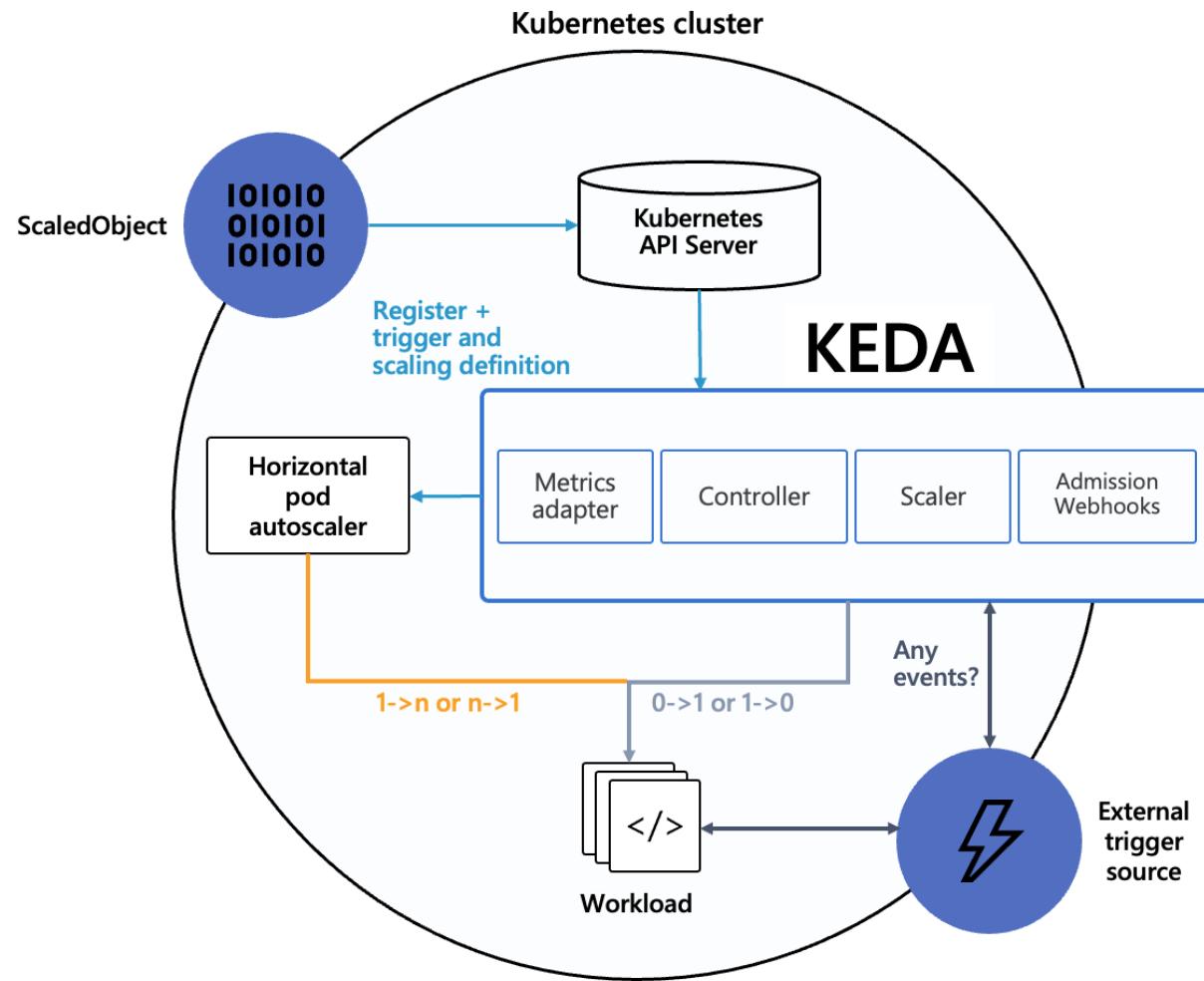
Scaling Cloud-native Applications in Kubernetes - KEDA

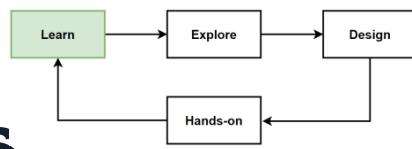
- **Cluster Autoscaler**
Adjusts the size of the Kubernetes cluster based on the current resource utilization, adding or removing nodes as needed. It ensures that there are enough resources for your app to scale out while minimizing the number of underutilized nodes.
- **Kubernetes Event-Driven Autoscaling (KEDA)**
KEDA is an extension to Kubernetes that provides event-driven autoscaling for applications. It can scale applications based on external events like the length of a message queue or the number of incoming HTTP requests.
- **KEDA** is an open-source project that **enables event-driven autoscaling** for Kubernetes workloads.
- It allows you to scale your applications **based on various event sources** and custom metrics, such as message queue length or HTTP requests per second.





Kubernetes Event-Driven Autoscaling (KEDA) Scaling in Kubernetes





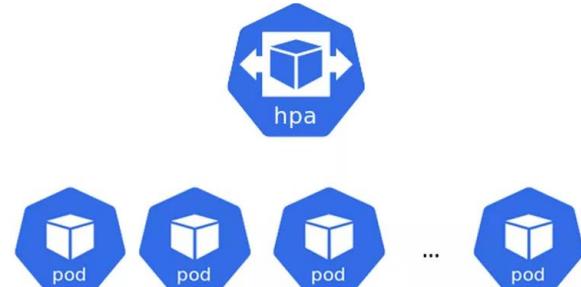
Best Practices of Scaling Cloud-native Apps in Kubernetes

Design for Statelessness

- Stateless apps are easier to scale horizontally, as they don't need to maintain state between instances. Design your apps to be stateless whenever possible.

Embrace Horizontal Scaling

- Focus on horizontal scaling (adding more instances) rather than vertical scaling.
- Horizontal scaling is more flexible and typically more cost-effective in the long run.



Use Autoscaling

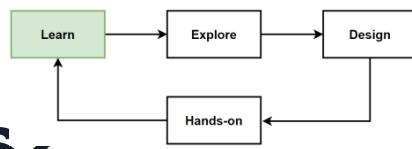
- Implement Horizontal Pod Autoscaler (HPA) and Cluster Autoscaler to adjust the number of instances and cluster size based on resource utilization and demand.

Implement Load Balancing

- Distribute incoming traffic evenly across your app instances using Kubernetes services, such as LoadBalancer or Ingress that ensure no single instance is overloaded.

Optimize Container Images

- Use small and efficient base images, remove unnecessary files, and optimize application code for better resource utilization and faster startup times.



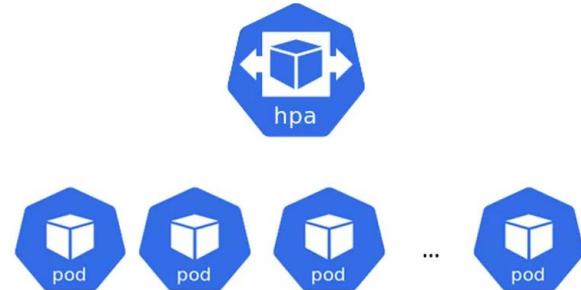
Best Practices of Scaling Cloud-native Apps in Kubernetes

Leverage Caching

- Use caching mechanisms to store frequently accessed data, reducing the need for expensive database queries or API calls.

Implement Asynchronous Communication

- Use async communication: message queues or event-driven architectures to offload processing to background tasks, reducing the load on your application instances.

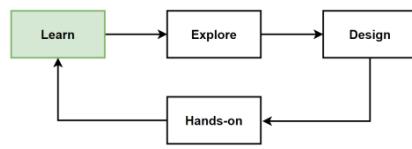


Decouple Components

- Break down your application into smaller, independent components (microservices) that can be scaled and deployed independently.

Monitor and Observe

- Continuously monitor app performance, resource utilization, and errors using tools like Prometheus, Grafana, and Jaeger.
- This helps you make informed decisions about when and how to scale your applications.

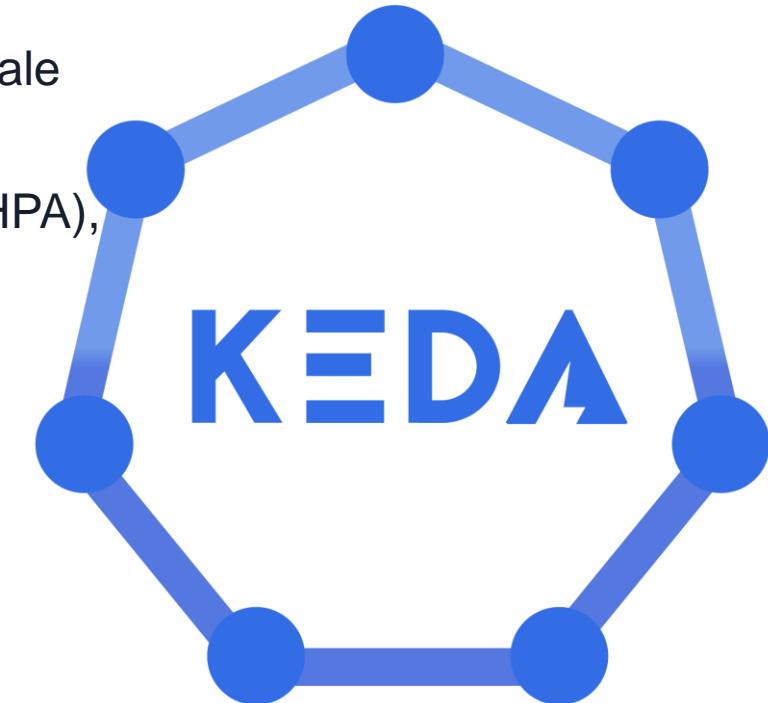


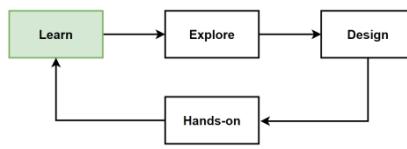
KEDA Event-driven Autoscaling Cloud-native Applications in Kubernetes

- **Kubernetes Event-Driven Autoscaling (KEDA)**

KEDA is an open-source project that provides event-driven autoscaling for container workloads in Kubernetes.

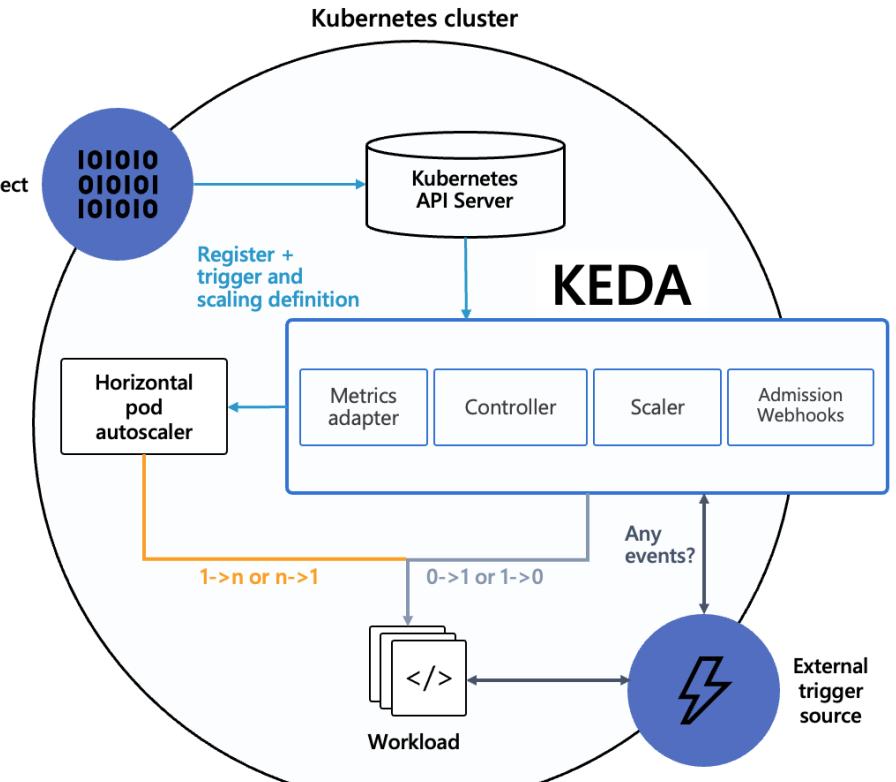
- It **enables** the deployment of **serverless containers** that can automatically scale based on **events** and **external metrics**.
- KEDA acts as an **extension** to the Kubernetes **Horizontal Pod Autoscaler (HPA)**, which usually scales based on CPU and memory utilization metrics.



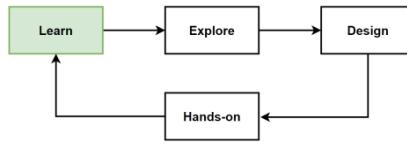


KEDA Architecture Components

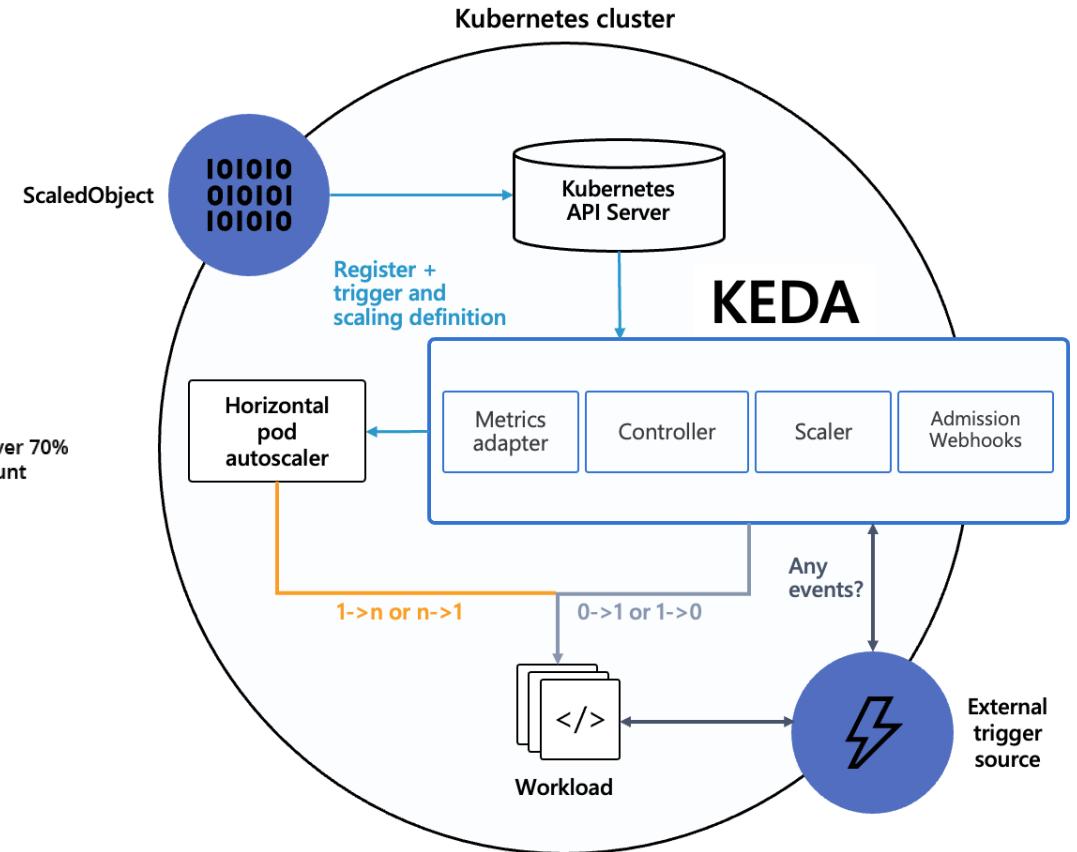
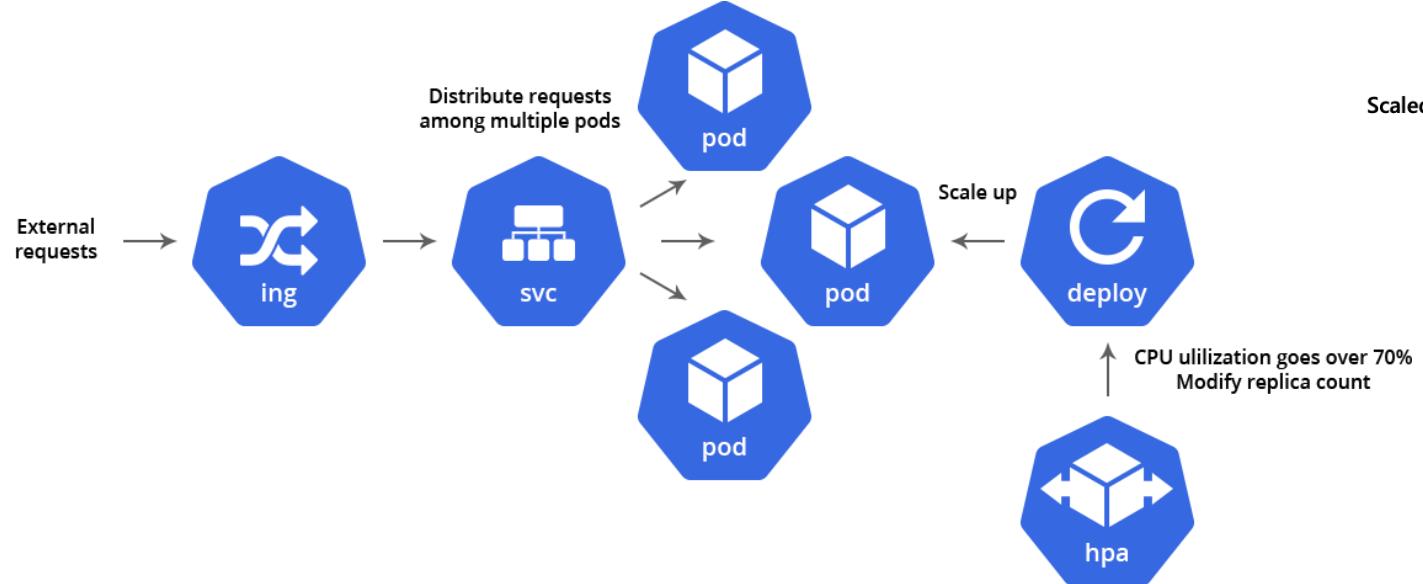
- KEDA provides **2 main components**: KEDA Operator and Metrics Server
- **KEDA Operator**
KEDA operator allows end-users to scale workloads in/out from 0 to N instances with support for Kubernetes Deployments, Jobs, StatefulSets or any custom resource that defines /scale subresource.
- **Metrics Server**
Metrics server exposes external metrics to Horizontal Pod Autoscaler (HPA) in Kubernetes for autoscaling purposes such as messages in a Kafka topic, or number of events in an Azure event hub.
- KEDA must be the only installed metric adapter.
- KEDA provides wide range of rich catalog of 50+ KEDA scalers.
- **Goto Official Website - Kubernetes Event-driven Autoscaling**
<https://keda.sh/>



<https://keda.sh/docs/2.10/concepts/>



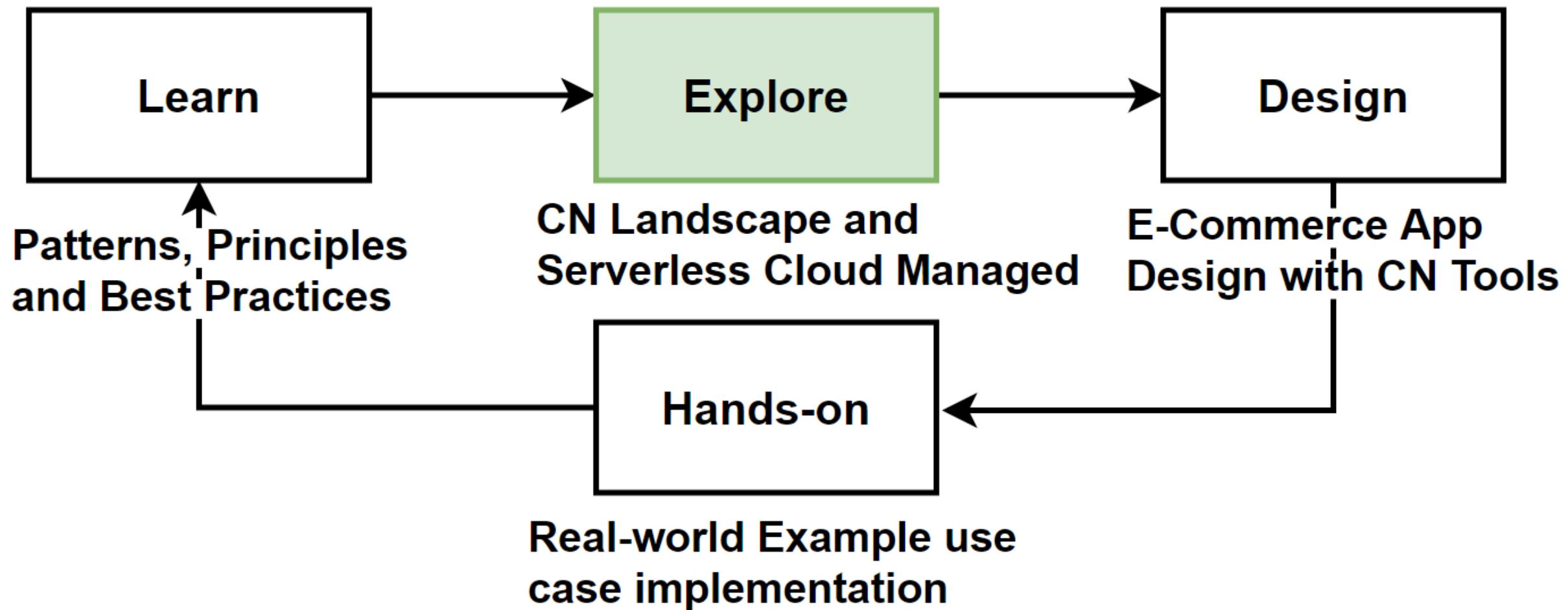
HPA and KEDA Scaling in Kubernetes



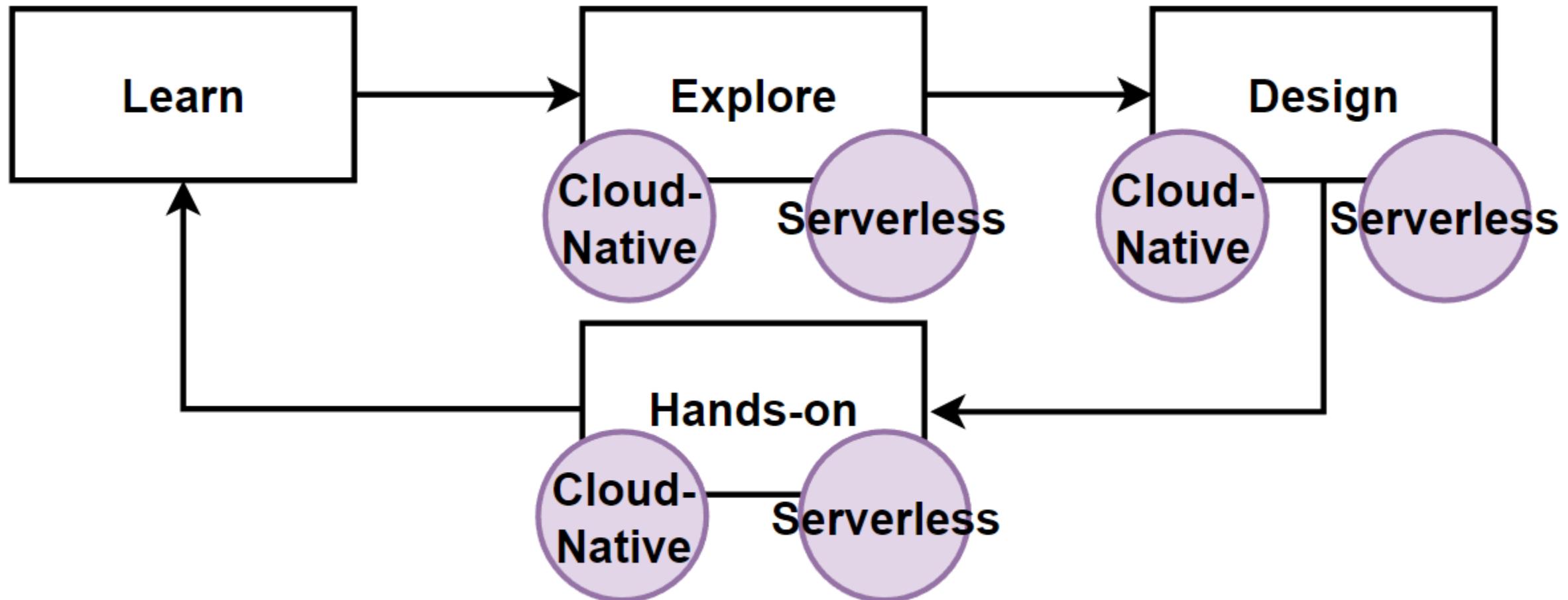
<https://www.virtuozzo.com/company/blog/scaling-kubernetes/>

<https://keda.sh/docs/2.10/concepts/>

Explore: Cloud Managed and Serverless Microservices Frameworks

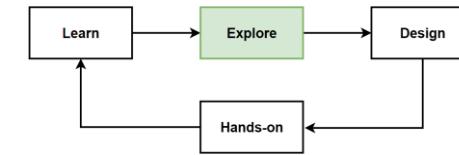


Way of Learning – Cloud-Native & Serverless Cloud Managed Tool



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Explore: Cloud-Native Scaling in Kubernetes

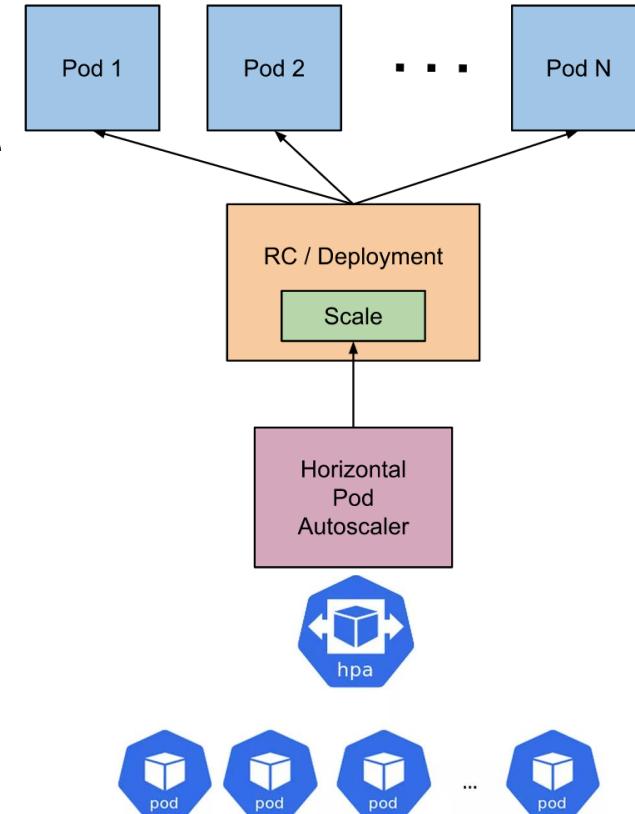
- The most commonly used orchestrators are Kubernetes, **scale with using Kubernetes components**.
- Vertical Scaling, Horizontal Scaling, Vertical Pod Autoscaler (VPA), Horizontal Pod Autoscaler (HPA), Cluster Autoscaler, Kubernetes Event-Driven Autoscaling (KEDA)

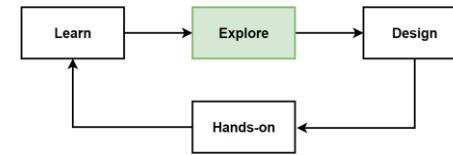
Cloud-Native Serverless Platforms for Kubernetes Auto-scaling

- Knative
- OpenFaas
- Kubeless
- Virtual Kubelet
- OpenFunction

Goto ->

<https://landscape.cncf.io/>





Explore: Cloud Serverless Scalability for Kubernetes

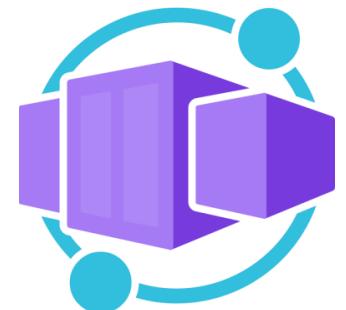
- **Serverless Kubernetes** solutions allow you to build and deploy applications without worrying about the underlying infrastructure.
- **Automatically scale based on demand** and only charge you for the resources you consume.

Cloud Serverless Kubernetes that Auto-scale on demand

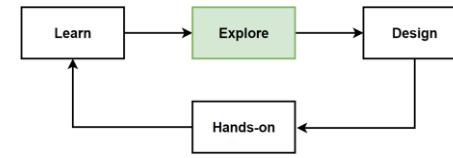
- [AWS Fargate for Amazon EKS](#)
- [Azure Container Apps](#)
- [Google Cloud Run on Google Kubernetes Engine \(GKE\)](#)
- [OpenShift Serverless](#)



AWS Fargate



Goto ->
<https://landscape.cncf.io/>



Cloud Serverless Kubernetes

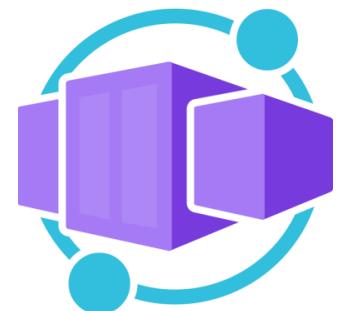
- **AWS Fargate for Amazon EKS**

AWS Fargate is a serverless compute engine for containers. Run Kubernetes workloads without managing the underlying infrastructure. Fargate automatically scales your applications based on demand and takes care of resource provisioning, patching, and maintenance.



- **Azure Container Apps**

ACA is a fully managed serverless platform for running containerized applications. It is built on top of Kubernetes and supports auto-scaling based on demand. Deploy and scale apps without worrying about the underlying infrastructure.



- Dynamically scale based on the following characteristics: HTTP traffic, Event-driven processing, CPU or memory load, Any KEDA-supported scaler

- **Google Cloud Run on Google Kubernetes Engine (GKE)**

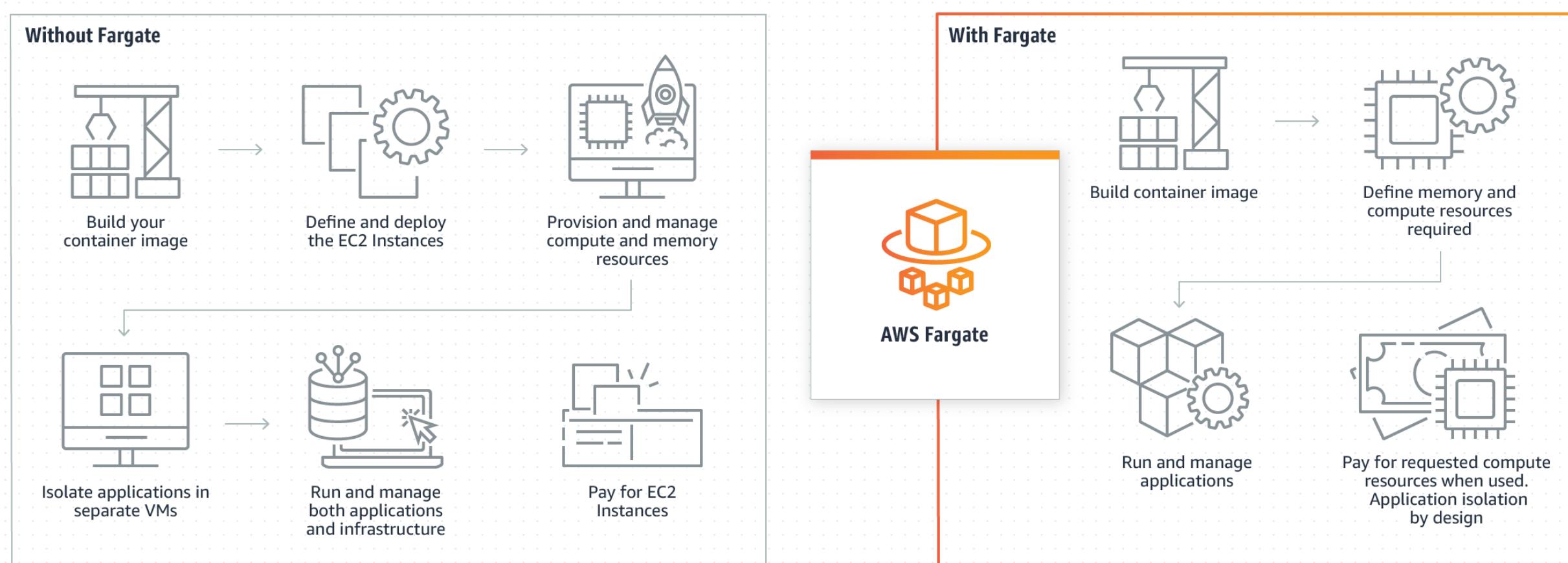
Cloud Run is a managed compute platform on Google Cloud that enables you to run containerized applications in a serverless environment. With Cloud Run on GKE, you can deploy serverless workloads on your existing GKE clusters.



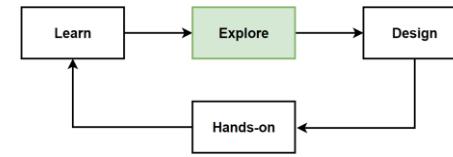
- Cloud Run automatically scales your apps based on demand, don't need to worry about managing the underlying infrastructure.

Cloud Run

AWS Fargate for Amazon EKS



<https://aws.amazon.com/fargate/>



Cloud Serverless Kubernetes

- **Knative on Kubernetes**

Knative is an open-source project: a set of components for building serverless apps on Kubernetes. Knative allows to deploy and scale containerized apps automatically on demand.

- **OpenShift Serverless**

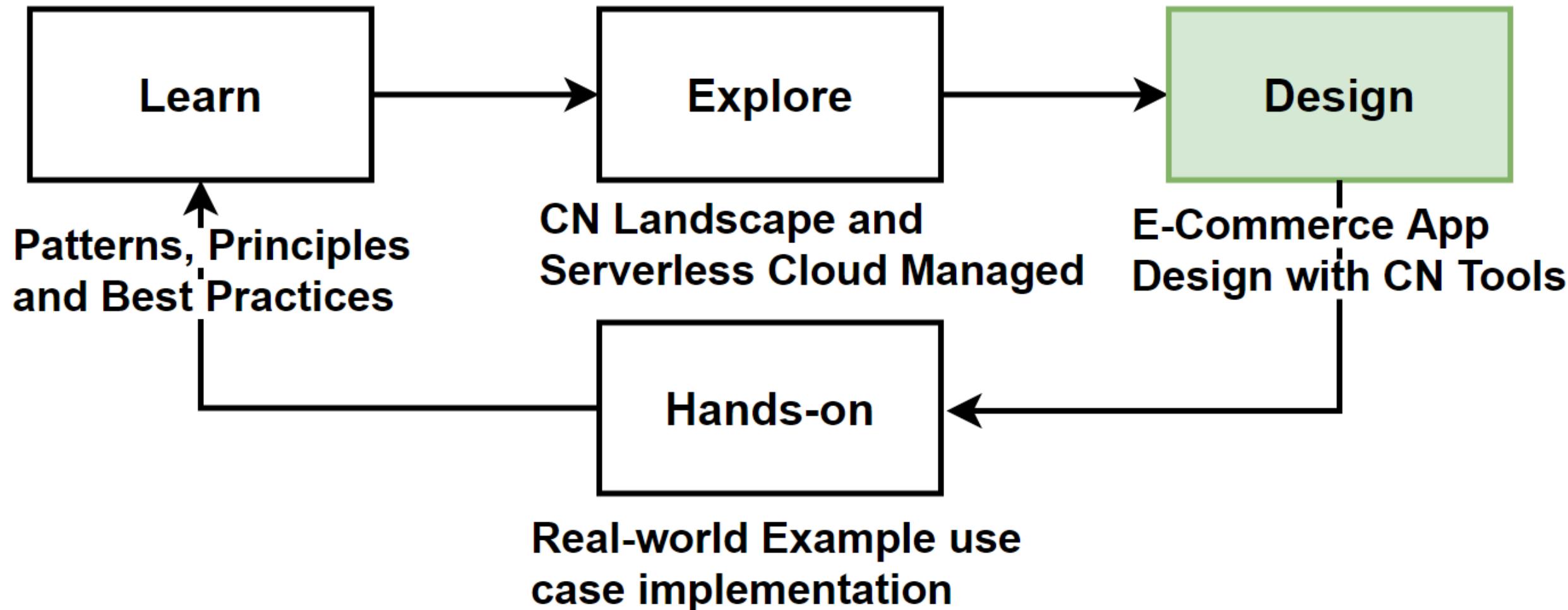
Red Hat OpenShift Serverless is a serverless platform built on top of Kubernetes and Knative. It enables you to deploy, scale, and manage serverless workloads on OpenShift clusters.

- OpenShift Serverless automatically scales your applications based on demand, allowing you to focus on your application code rather than infrastructure management.

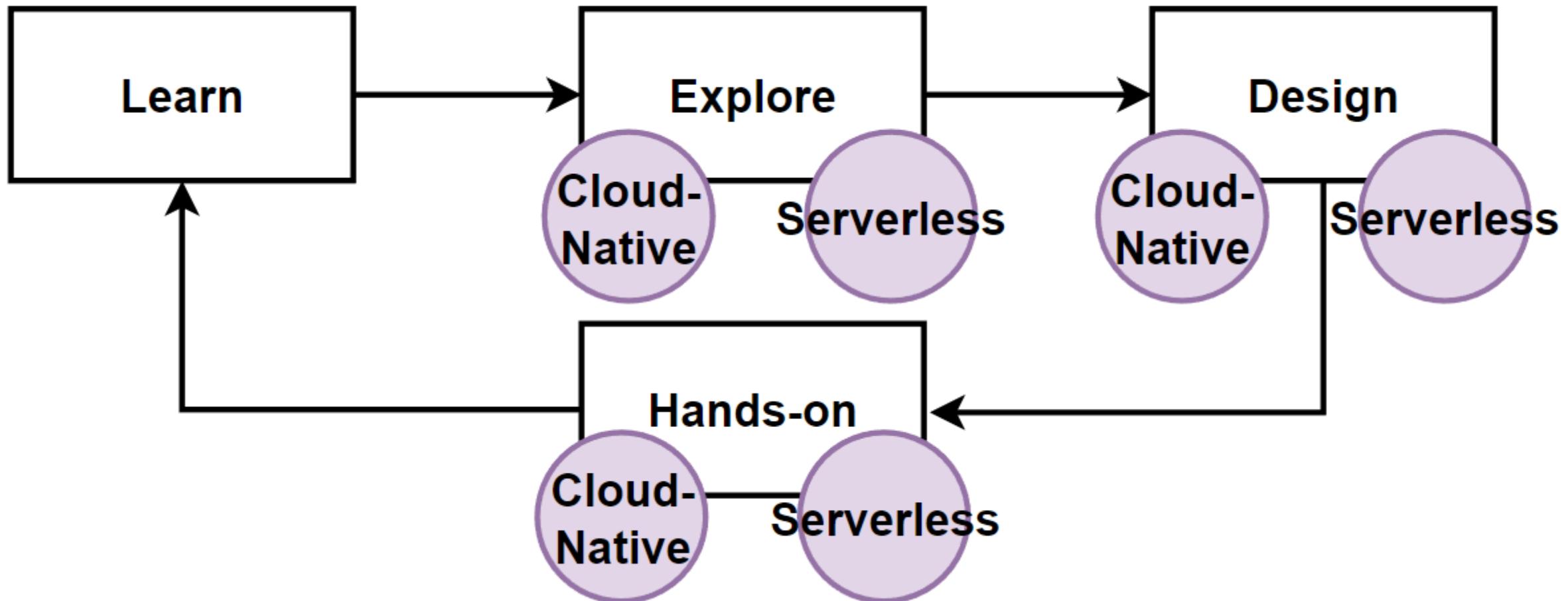


OPENSIFT

Way of Learning – The Course Flow

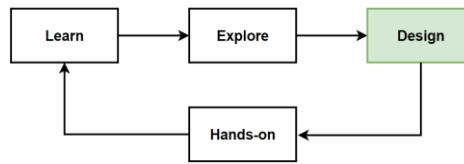


Way of Learning – Cloud-Native & Serverless Cloud Managed

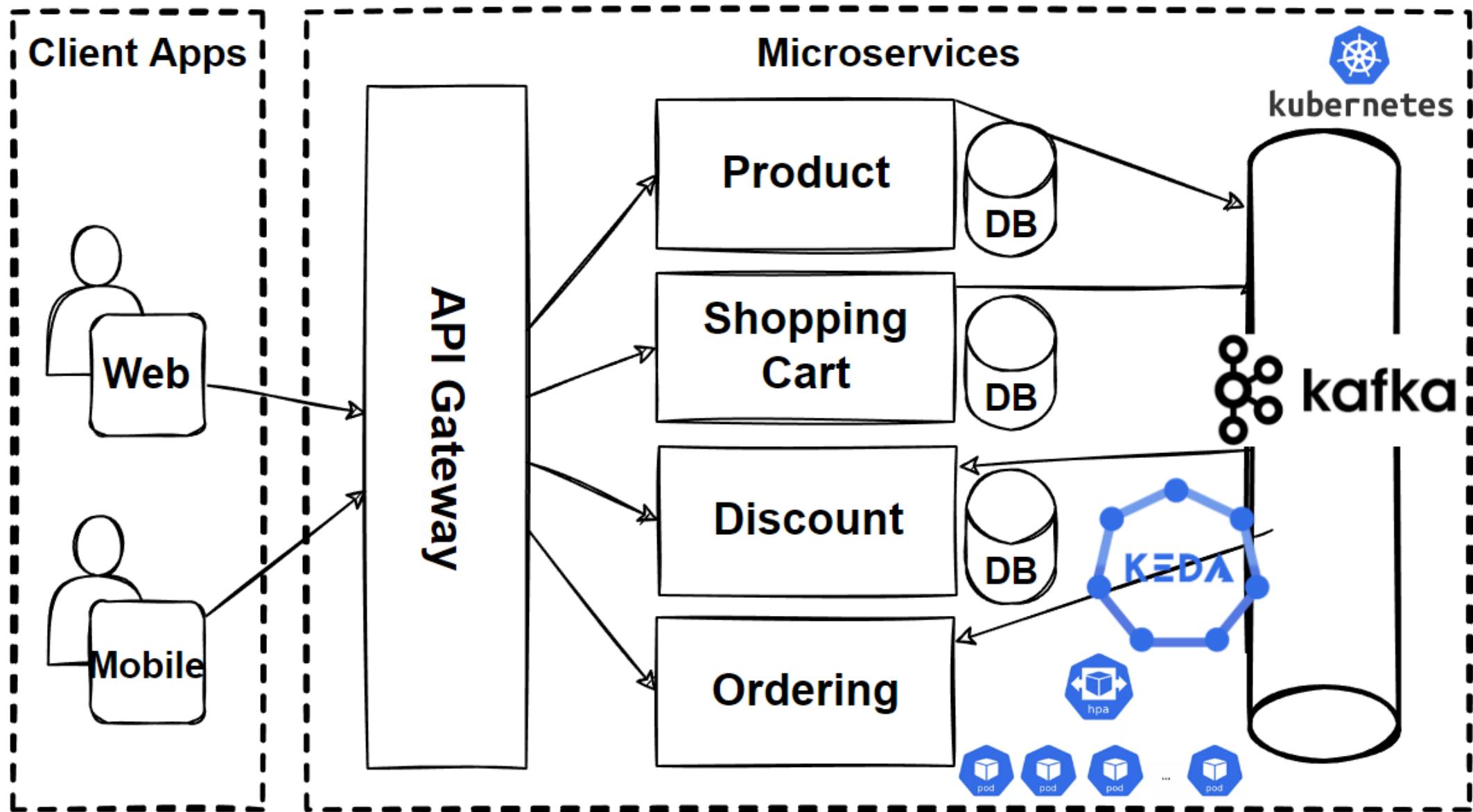


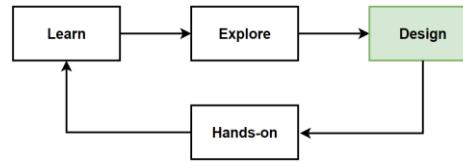
Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



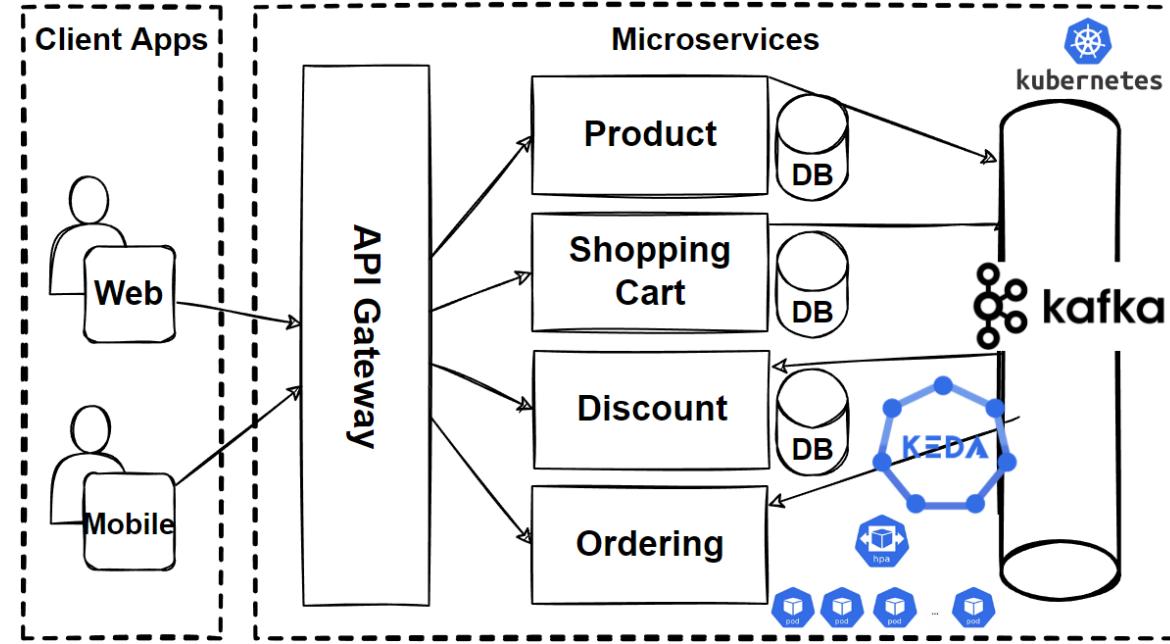
Microservice Cloud-Native Scalability w/ KEDA and HPA

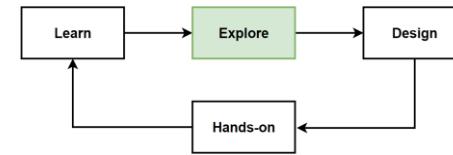




Shopping Cart – Kafka – Ordering – KEDA Workflow

1. Users interact with the shopping cart and eventually proceed to checkout.
2. The Shopping Cart microservice publishes the checkout event to the Kafka topic.
3. KEDA monitors this topic, and if there is a sudden increase in events, it starts scaling out the Ordering microservice by increasing the number of pods.
4. Multiple instances of the Ordering microservice consume the messages from the Kafka topic in parallel, processing the orders.
5. As the load decreases and the events in the Kafka topic are processed, KEDA scales in the Ordering microservice.





Explore: Cloud Serverless Scalability for Kubernetes

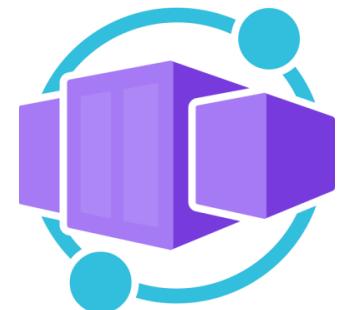
- **Serverless Kubernetes** solutions allow you to build and deploy applications without worrying about the underlying infrastructure.
- **Automatically scale based on demand** and only charge you for the resources you consume.

Cloud Serverless Kubernetes that Auto-scale on demand

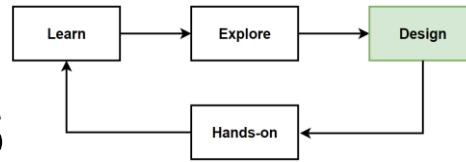
- [AWS Fargate for Amazon EKS](#)
- [Azure Container Apps](#)
- [Google Cloud Run on Google Kubernetes Engine \(GKE\)](#)
- [OpenShift Serverless](#)



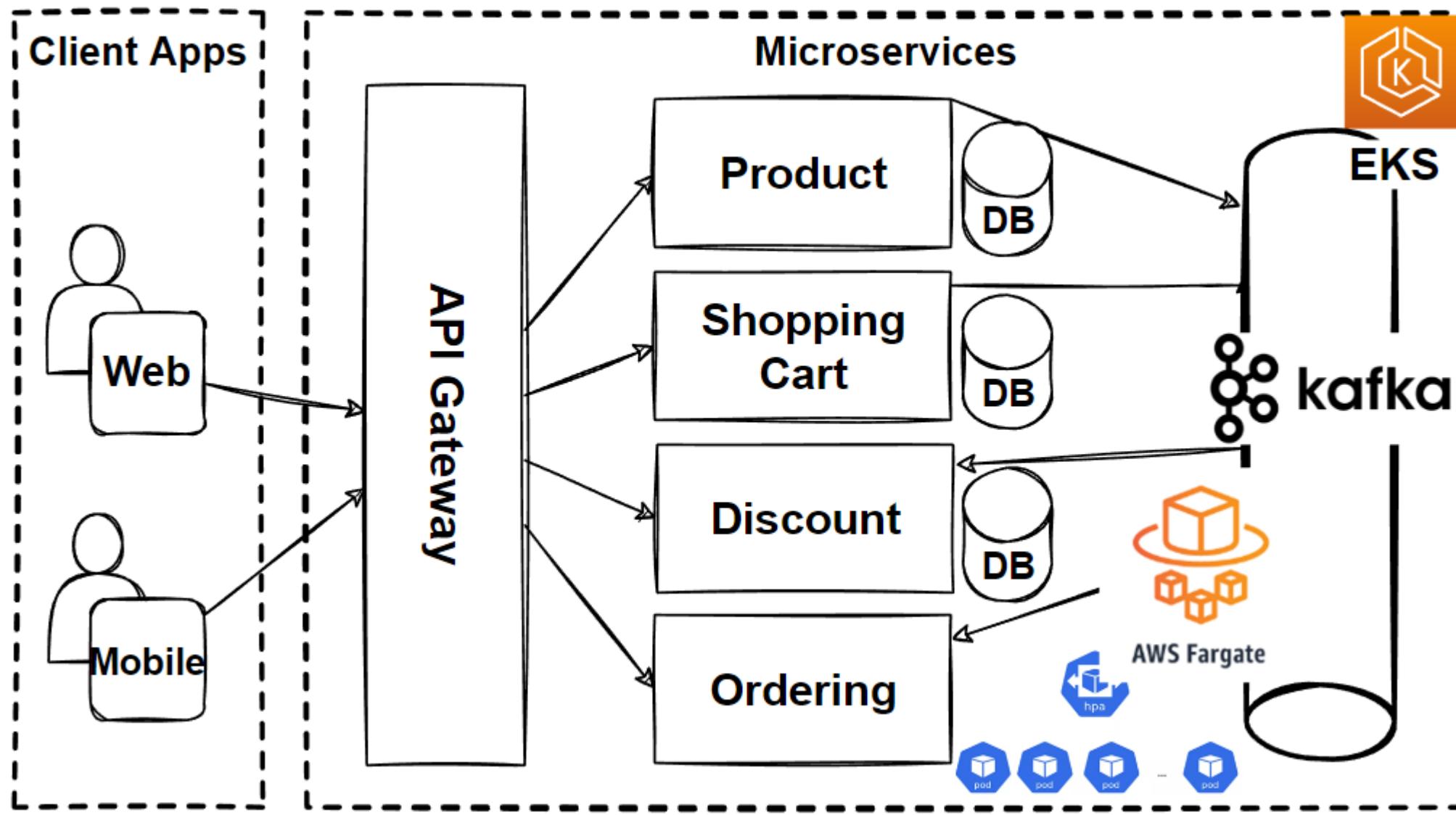
AWS Fargate

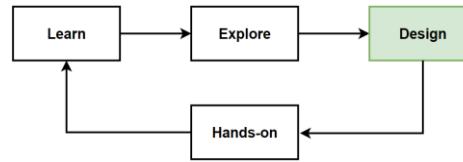


Goto ->
<https://landscape.cncf.io/>



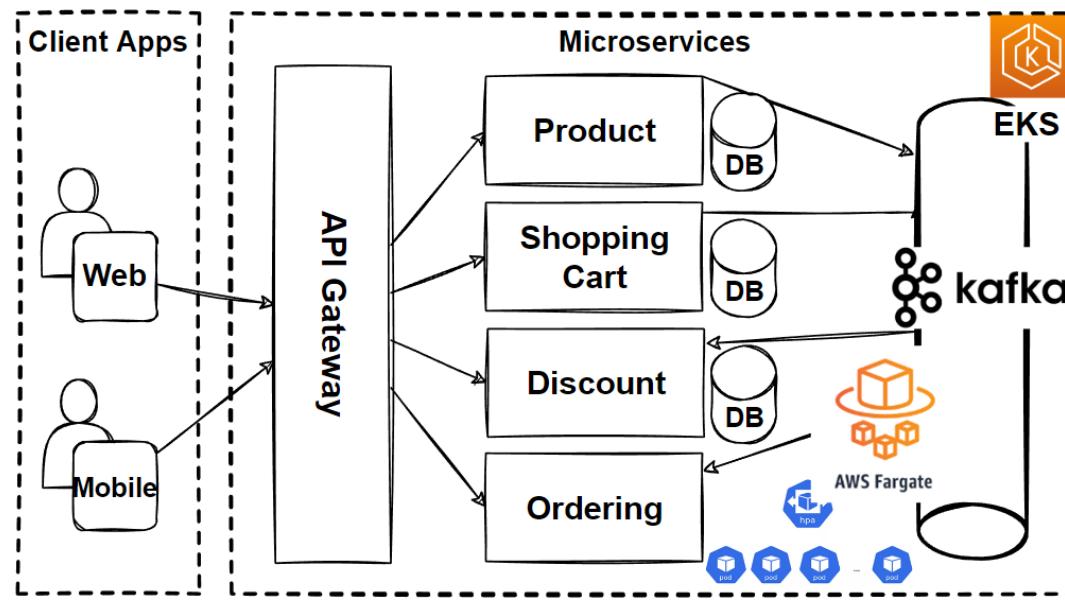
Design: Cloud-Native Scalability w/ AWS Fargate for EKS

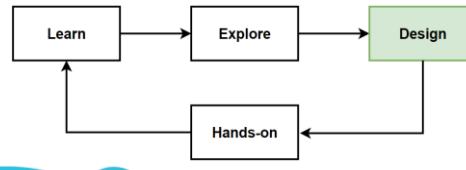




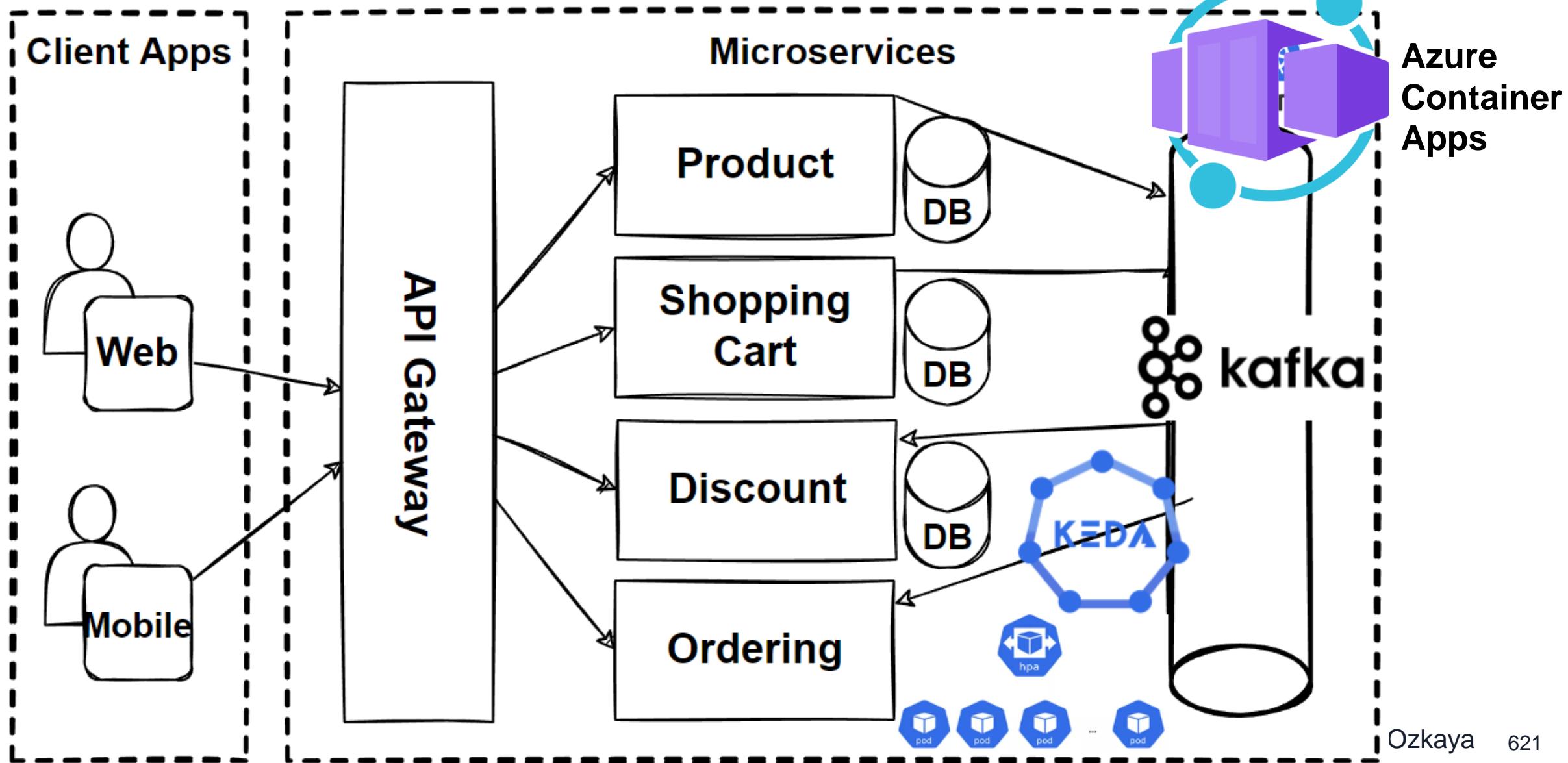
Shopping Cart – Kafka – Ordering – Fargate Workflow

1. Users interact with the shopping cart and eventually proceed to checkout.
2. The Shopping Cart microservice publishes the checkout event to the Kafka topic.
3. **AWS Fargate** is monitoring the metrics (like CPU or custom metrics that indicate load).
4. If there's a surge in events, Fargate starts scaling out the Ordering microservice by launching more tasks (containers).
5. Multiple instances of the Ordering microservice consume messages from Kafka and process the orders.
6. As the load decreases, **AWS Fargate** scales in by **terminating excess tasks**.





Microservice Cloud-Native Scalability w/ Azure



Hands-on: Scale Kubernetes Pods (VPA,HPA,KEDA) on a Kubernetes Cluster with Minikube

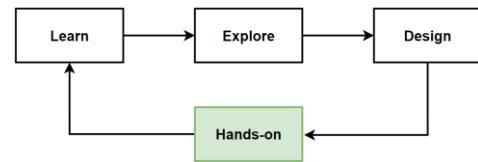
Declarative way scale Kubernetes Pods with Deployment yaml files

Vertical Pod Autoscaler (VPA) on a Kubernetes Cluster with Minikube

Horizontal Pod Autoscaler (HPA) on a Kubernetes Cluster with Minikube

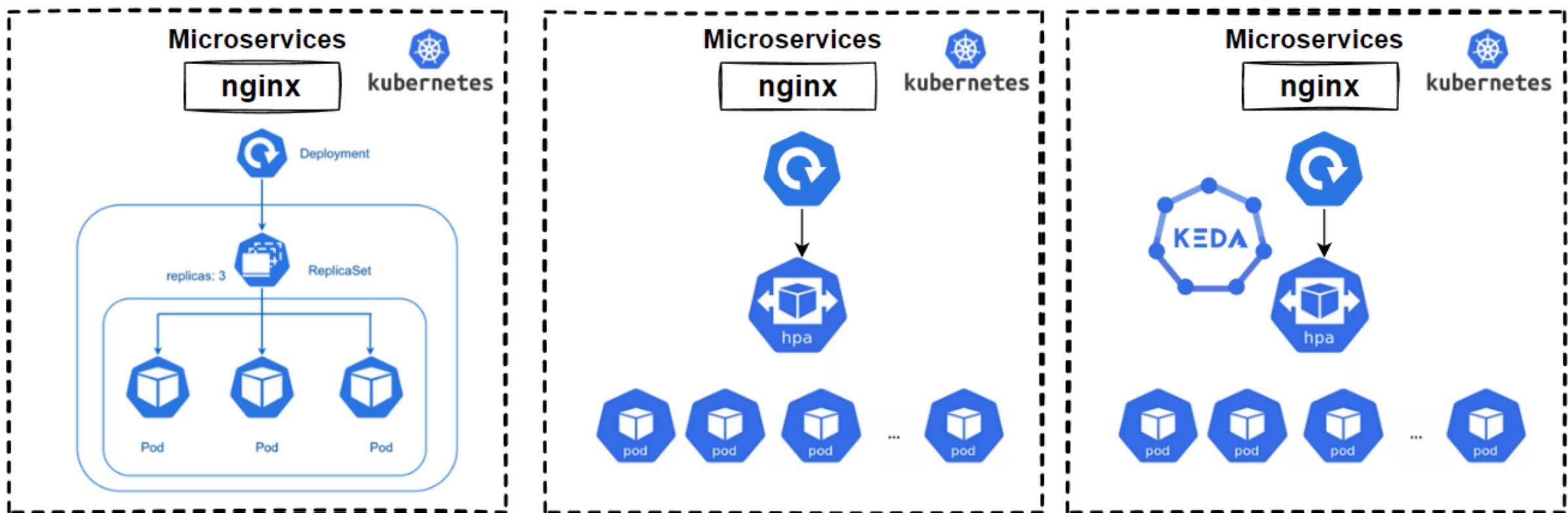
Kubernetes Event-Driven Autoscaling (KEDA) on a Kubernetes Cluster with Minikube

Mehmet Ozkaya

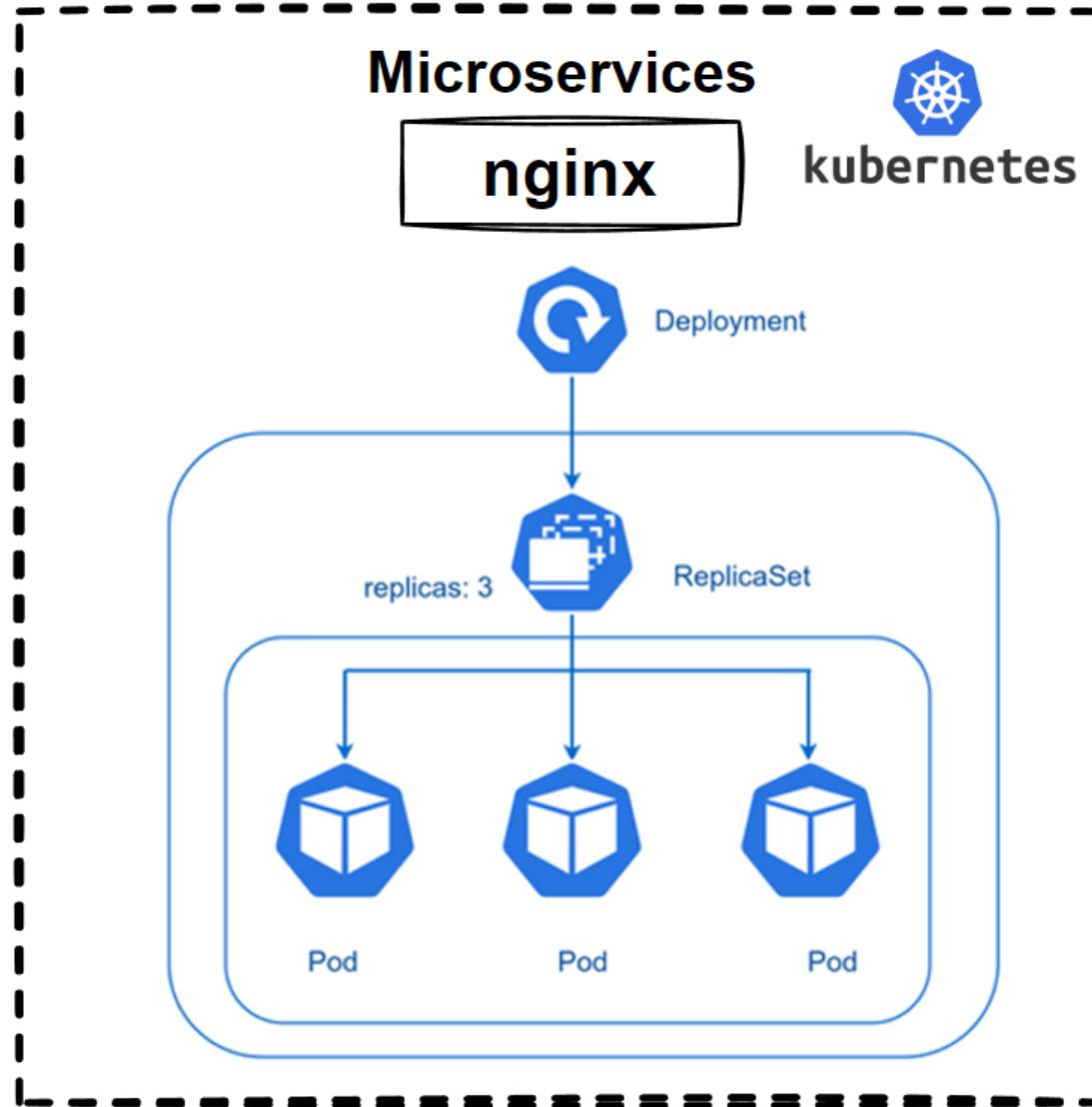
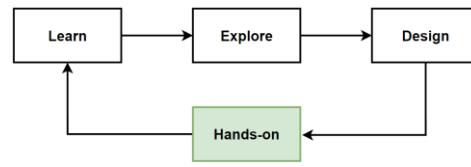


Hands-on: Scale Kubernetes Pods (VPA,HPA,KEDA) with Minikube – Task List

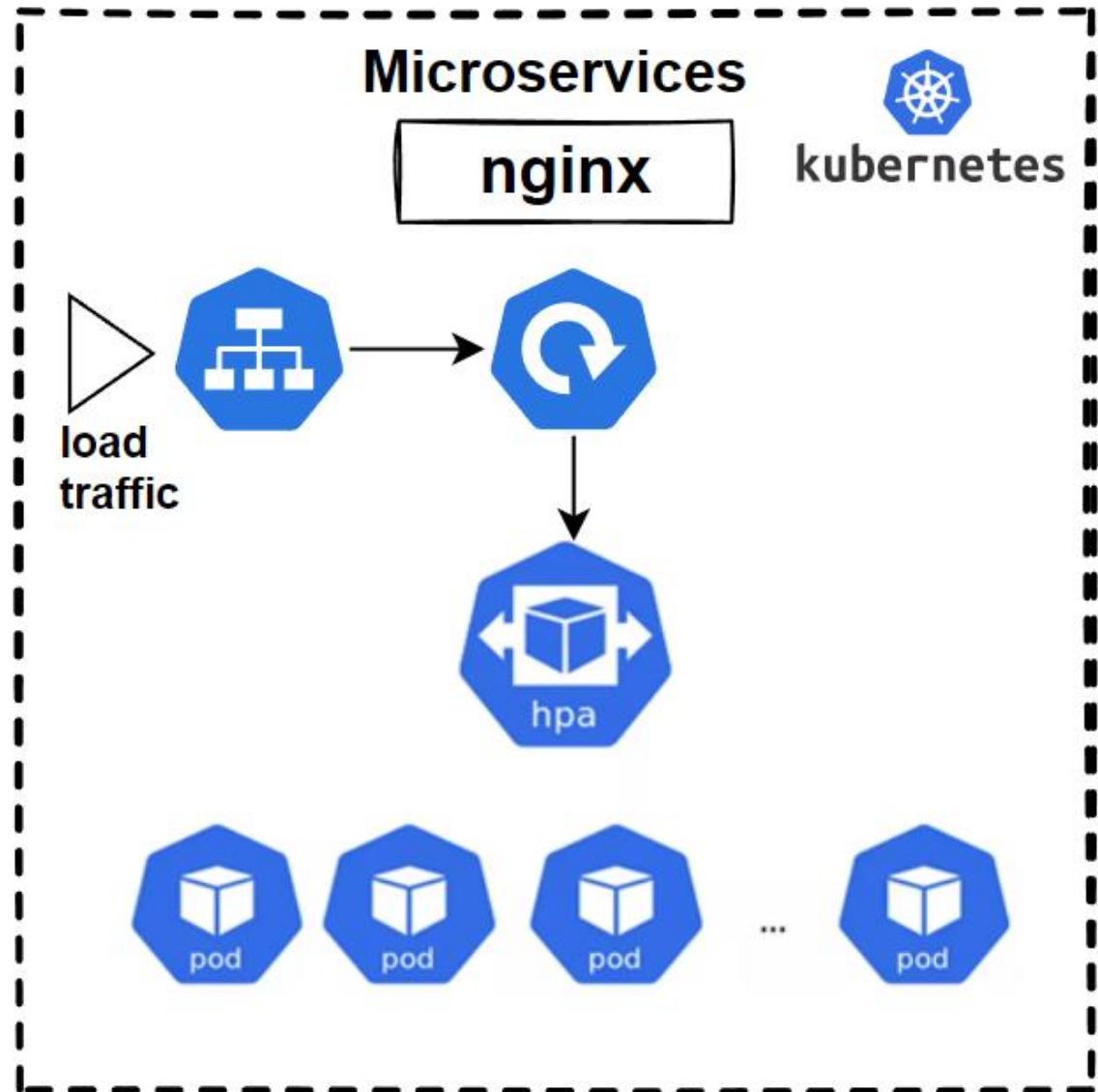
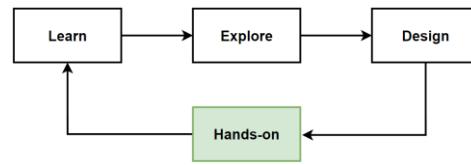
- Step 1. Manually Horizontal and Vertical scaling pods into Kubernetes Cluster with Minikube
- Step 2. Using HPA and VPA for scaling pods into Kubernetes Cluster with Minikube
- Step 3. Using KEDA for scaling pods into Kubernetes Cluster with Minikube

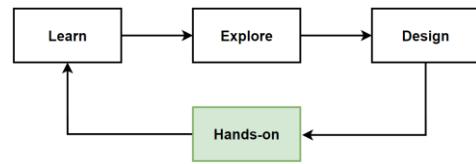


S1: Manually Horizontal and Vertical scaling pods into Kubernetes Cluster with Minikube



S2: Horizontal Pod Autoscaler (HPA) to auto-scale pods on a Kubernetes cluster using Minikube





Why needed KEDA instead of HPA ?

- Kubernetes Event-Driven Autoscaling (KEDA) and Horizontal Pod Autoscaler (HPA) are both used for scaling applications running in Kubernetes clusters, different purposes and cases.

Autoscaling triggers

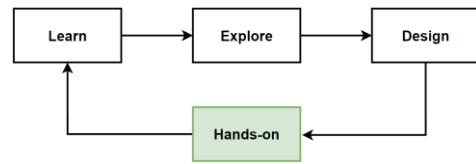
- HPA uses resource-based metrics: CPU and memory utilization, to determine when to scale app.
- Suitable for apps with predictable resource usage patterns.
- KEDA is designed for event-driven apps, uses external metrics: the length of a message queue or the number of messages in a Kafka topic, to scale the application.
- KEDA a better fit for scenarios where the workload depends on the processing of events.



Scale-to-zero Capability

- HPA not support scaling the app to zero replicas. The minimum number of replicas is always at least one.
- Not suitable for scenarios where you want to save resources when there's no work to do.
- KEDA supports scaling the app to zero replicas when there are no events to process.
- Useful for optimizing resource usage in event-driven apps that experience periods of inactivity.





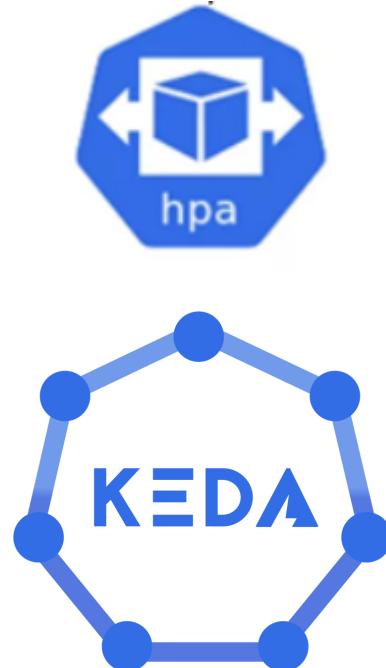
Why needed KEDA instead of HPA ? - 2

Custom Scalers and Integration with External Systems

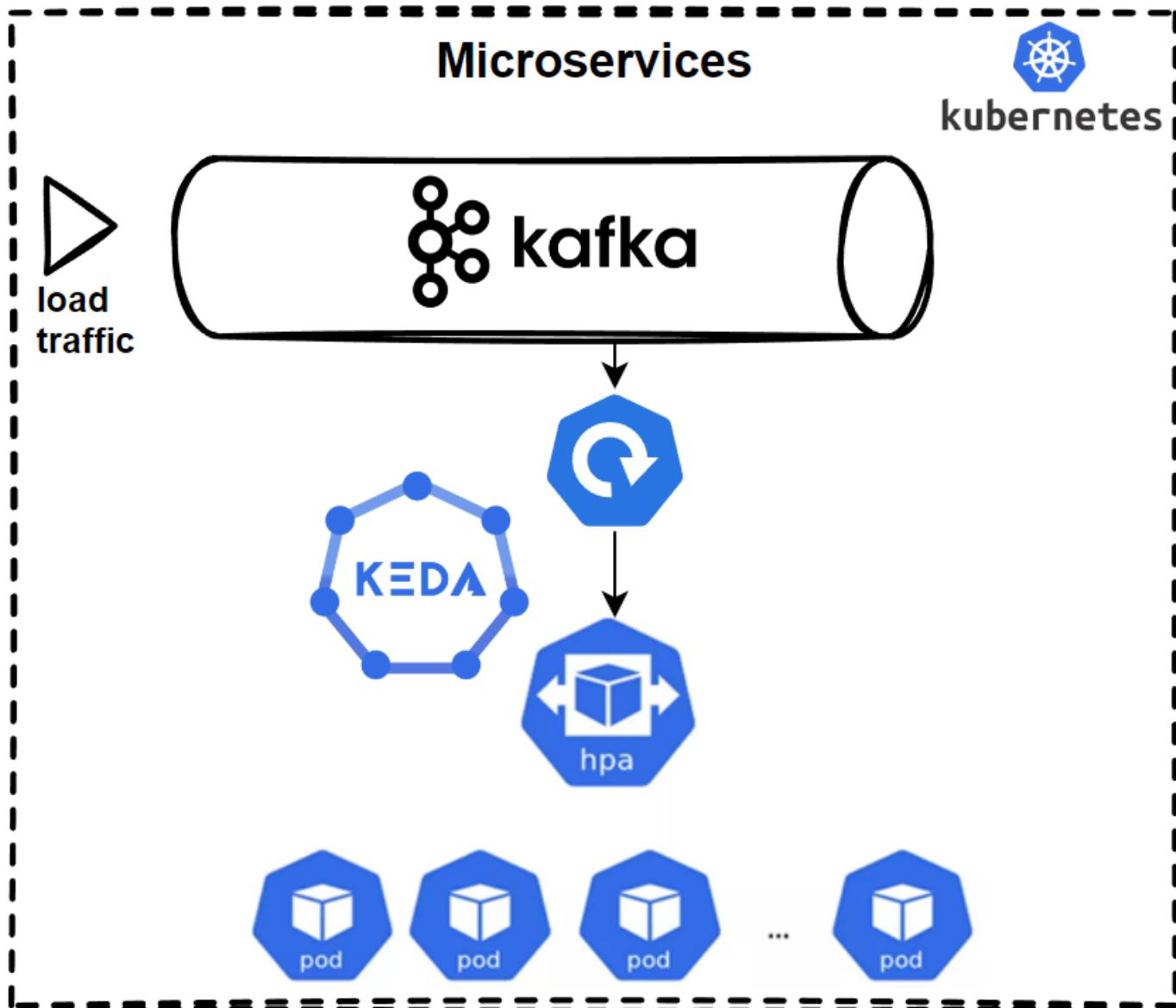
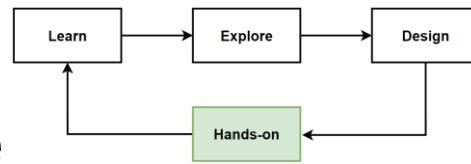
- KEDA supports a wide range of scalers that can be used to integrate with various event sources, such as message queues, databases, and cloud services.
- Allows KEDA to scale applications based on various external factors and events.
- HPA, in contrast, relies on resource-based metrics and custom metrics from the app.
- <https://keda.sh/> -> Scalers -> i.e. Prometheus

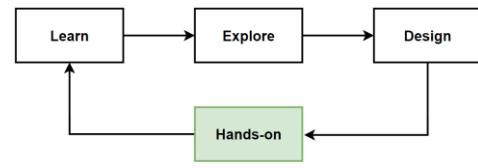
Consider KEDA over HPA

- App is event-driven and needs to scale based on external metrics or events.
- Optimize resource usage by scaling the application to zero replicas during periods of inactivity.
- Need integration with various event sources or external systems for autoscaling.
- If app has predictable resource usage patterns and scales based on CPU or memory utilization, HPA might be a better fit.

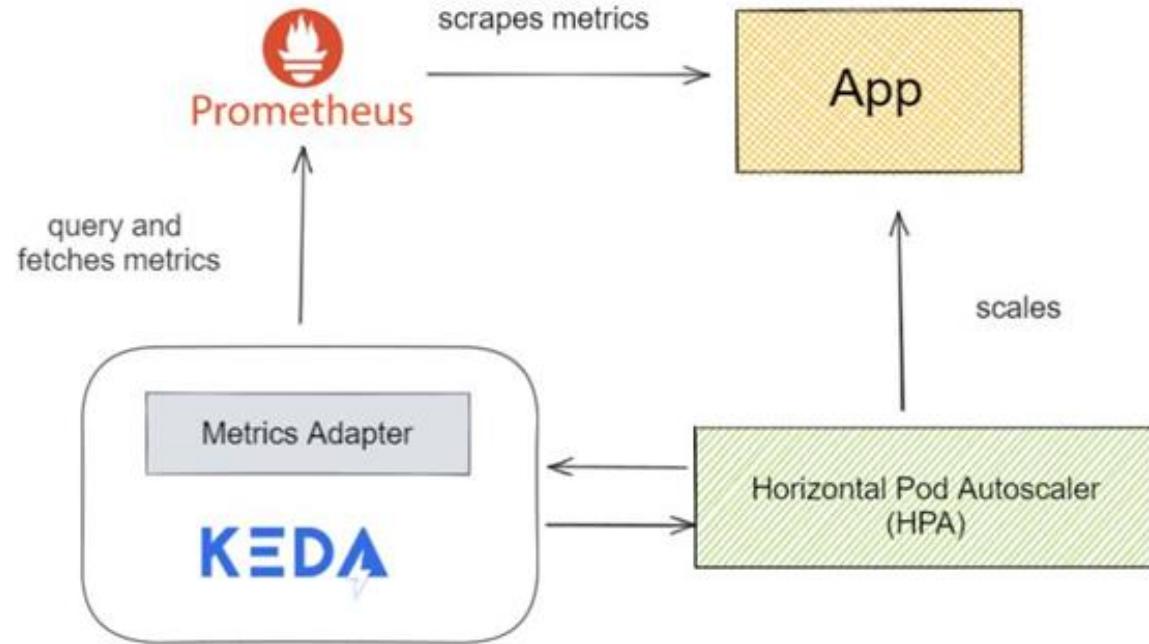


S3: Kubernetes Event-Driven Autoscaling (KEDA) auto-scale pods on a Kubernetes cluster using Minikube





Auto-scaling Spring Boot Microservices in Kubernetes with Prometheus and KEDA using Custom Metrics

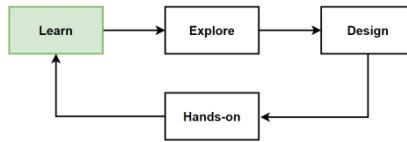


1. [Monitor Spring Boot Custom Metrics with Micrometer and Prometheus using Docker](#)
2. [Monitor Custom Metrics with deploying Kubernetes using Prometheus](#)
3. [Auto-scaling Kubernetes apps with Prometheus and KEDA](#)

Deploy on Cloud Serverless Kubernetes - AWS Fargate for EKS, Azure Container Apps, Google Cloud Run on GKE

Leveraging the power of Kubernetes while avoiding the management of the underlying nodes or servers.

AWS Fargate for EKS, Azure Container Apps, GKE with Cloud Run



Cloud Serverless Kubernetes – Deploy Microservices

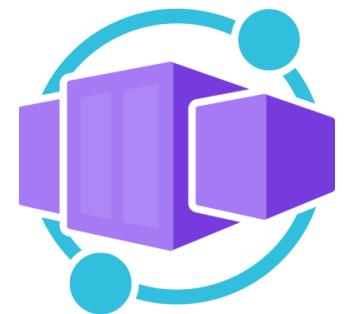
- **Serverless Kubernetes** solutions allow you to build and deploy applications without worrying about the underlying infrastructure.
- **Automatically scale, patch, and manage** the container instances, freeing users from provisioning, configuring, and managing the servers.
- Automatically scale **based on demand** and only charge you for the resources you consume.

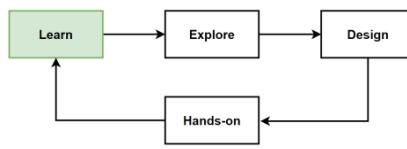
Cloud Serverless Kubernetes that Auto-scale on demand

- AWS Fargate for Amazon EKS
- Azure Container Apps
- Google Cloud Run on Google Kubernetes Engine (GKE)
- Knative on Kubernetes
- OpenShift Serverless



AWS Fargate





Cloud Serverless Kubernetes

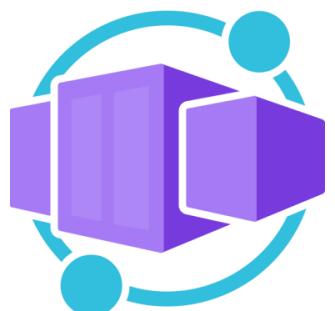
- **AWS Fargate for Amazon EKS**

AWS Fargate is a serverless compute engine for containers. Run Kubernetes workloads without managing the underlying infrastructure. Fargate automatically scales your applications based on demand and takes care of resource provisioning, patching, and maintenance.



- **Azure Container Apps**

ACA is a fully managed serverless platform for running containerized applications. It is built on top of Kubernetes and supports auto-scaling based on demand. Deploy and scale apps without worrying about the underlying infrastructure.



- Dynamically scale based on the following characteristics: HTTP traffic, Event-driven processing, CPU or memory load, Any KEDA-supported scaler

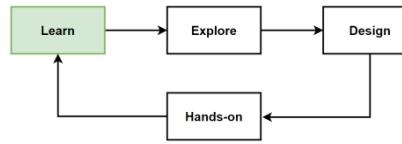
- **Google Cloud Run on Google Kubernetes Engine (GKE)**

Cloud Run is a managed compute platform on Google Cloud that enables you to run containerized applications in a serverless environment. With Cloud Run on GKE, you can deploy serverless workloads on your existing GKE clusters.



- Cloud Run automatically scales your apps based on demand, don't need to worry about managing the underlying infrastructure.

Cloud Run



Cloud Serverless Kubernetes

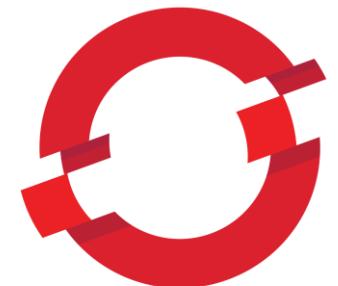
- **Knative on Kubernetes**

Knative is an open-source project: a set of components for building serverless apps on Kubernetes. Knative allows to deploy and scale containerized apps automatically on demand.

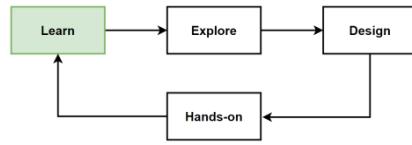
- **OpenShift Serverless**

Red Hat OpenShift Serverless is a serverless platform built on top of Kubernetes and Knative. It enables you to deploy, scale, and manage serverless workloads on OpenShift clusters.

- OpenShift Serverless automatically scales your applications based on demand, allowing you to focus on your application code rather than infrastructure management.

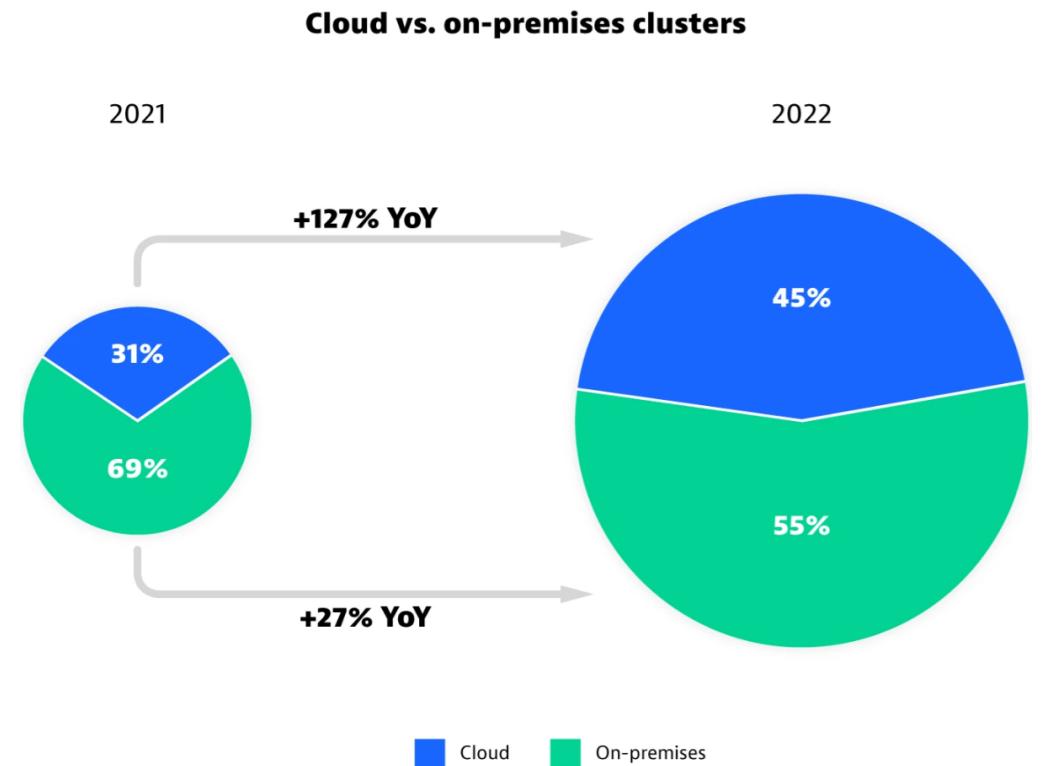


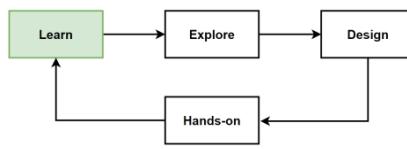
OPENSIFT



Kubernetes moved to the cloud in 2022

- [Dynatrace Kubernetes Report: Kubernetes moved to the cloud](#)
- Kubernetes became the key platform for moving workloads to the public cloud. At an annual growth rate of +127 percent. The number of Kubernetes clusters hosted in the cloud increased five times faster than those hosted on-premises.
- Share of cloud-hosted clusters increased from 31% in 2021 to 45% in 2022.
- Cloud-hosted Kubernetes clusters are surpass on-premises deployments in 2023.
- Significant majority (73%) of Kubernetes clusters in the cloud are built on managed distributions provided by major cloud providers, such as AWS Elastic Kubernetes Service (EKS), Azure Kubernetes Service (AKS), or Google Kubernetes Engine (GKE).
- The remaining 27% of clusters are self-managed by customers on cloud virtual machines.





Why Cloud-hosted Kubernetes ?

- **Cost**

Managed Kubernetes services offered by cloud providers generally have lower infrastructure and operational costs compared to on-premises solutions. Benefits of cloud's pay-as-you-go pricing model, avoiding upfront capital expenses for hardware and infrastructure.

- **Ease of provisioning and scaling**

Managed Kubernetes services provide easy-to-use interfaces and tools for provisioning, scaling, and managing clusters. Enables organizations to quickly deploy and scale their applications to meet demand, resulting in increased agility and faster time to market.

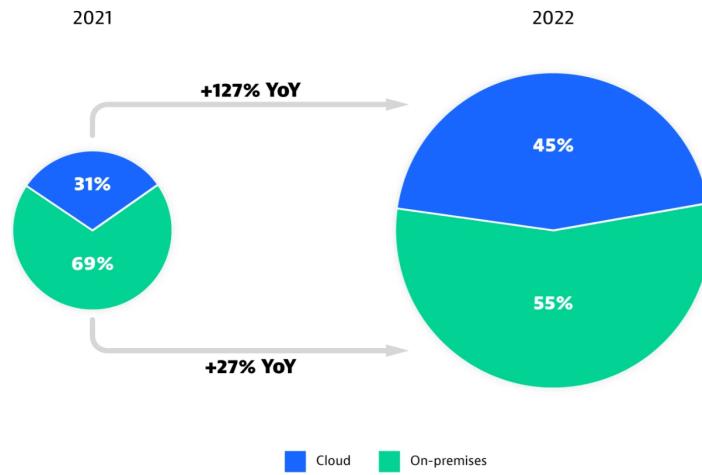
- **Data security**

Cloud providers have stringent security measures in place to protect customer data. They also invest in regular security updates and improvements.

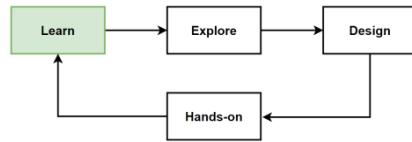
- **Regulatory compliance**

Cloud providers are increasingly focusing on ensuring their services meet various regulatory requirements across different industries and regions. Easier for organizations to maintain compliance while leveraging cloud-based Kubernetes services.

Cloud vs. on-premises clusters

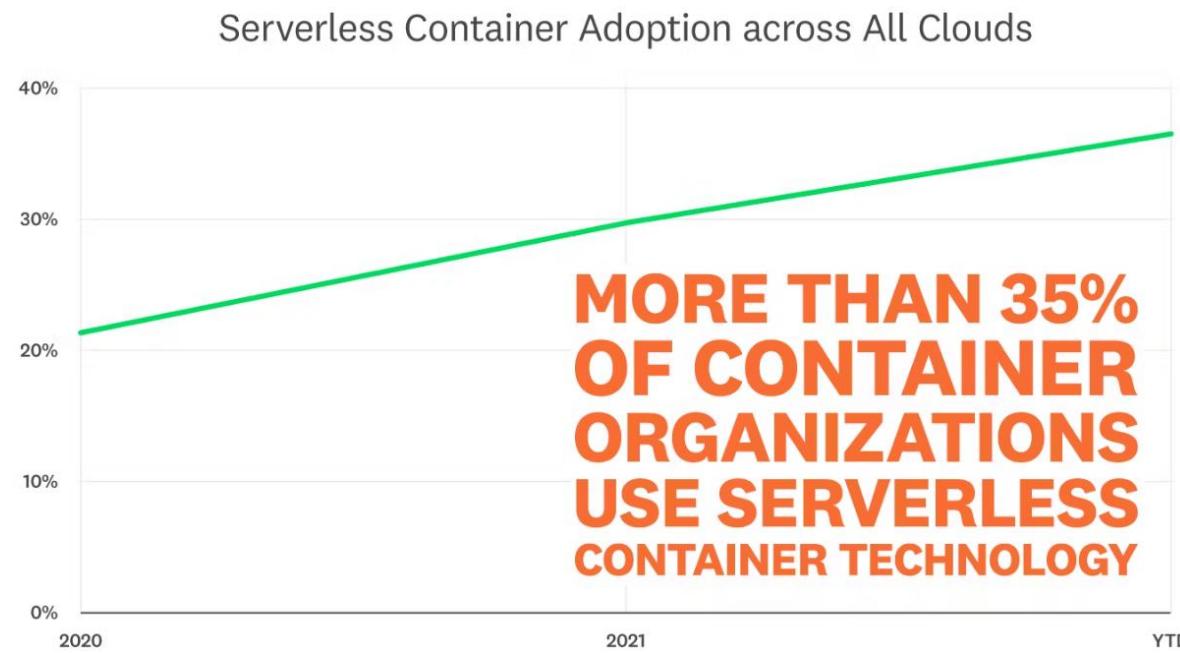


<https://www.dynatrace.com/news/blog/kubernetes-in-the-wild-2023/>

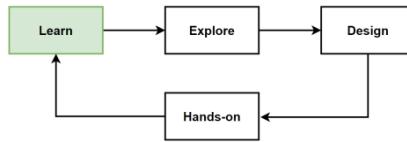


Serverless Kubernetes grow for all major public clouds

- Datadog Report: Serverless Kubernetes grow for all major public clouds
- Growing trend among organizations to adopt serverless container technologies. developers to run and manage applications in containers without having to worry about the underlying infrastructure.
- Usage of serverless container technologies from all major cloud providers—including AWS App Runner, AWS Fargate, Azure Container Apps, Azure Container Instances (ACI), and Google Cloud Run—increased from 21 percent in 2020 to 36 percent in 2022.
- Migration of users from Amazon ECS, toward AWS Fargate, a serverless compute engine for containers.
- Customers said that reduced need to provision and manage underlying infrastructure as one of the primary reasons for adopting serverless.
- Some Customers want to retain more control and flexibility over their infrastructure. This can be critical for certain use cases or regulatory requirements.

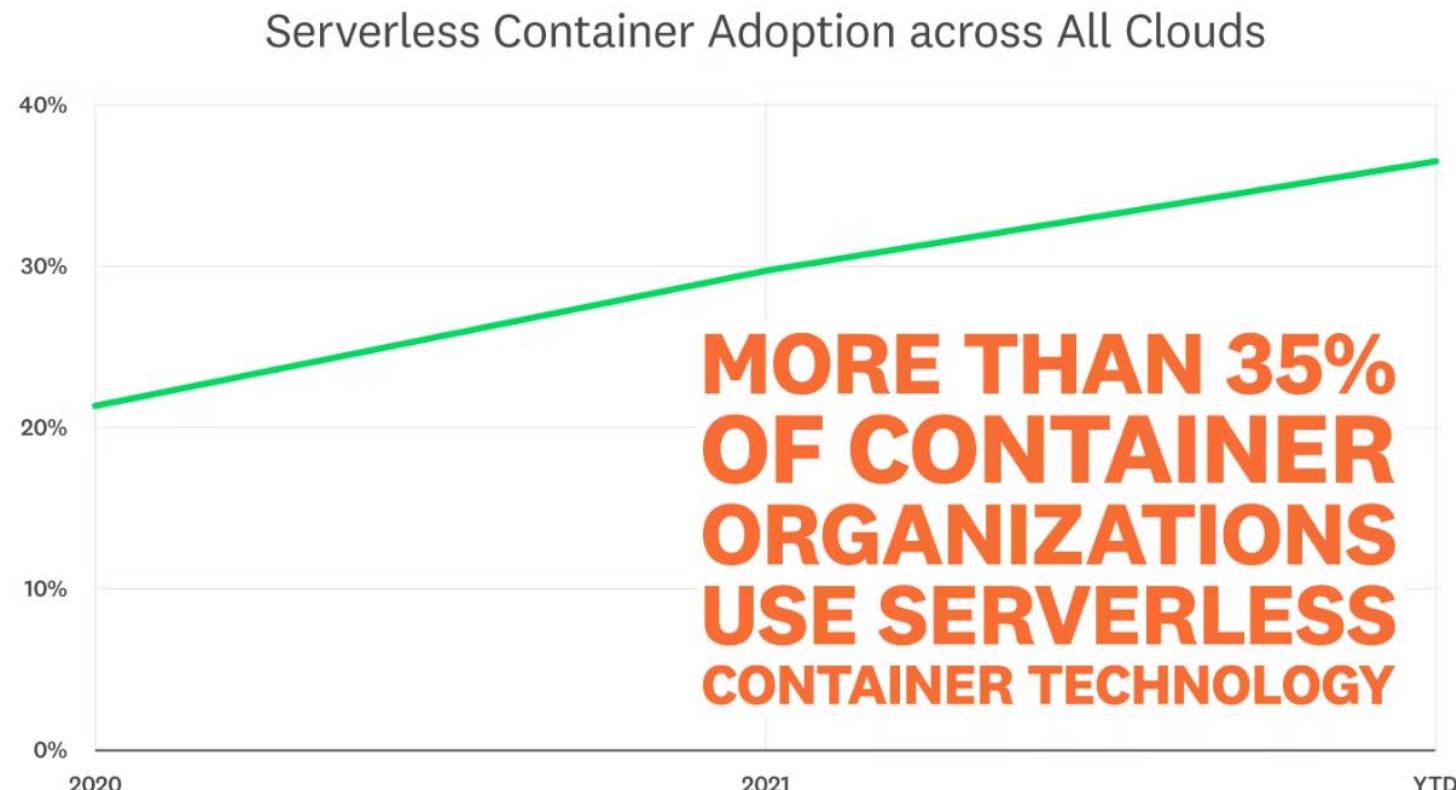


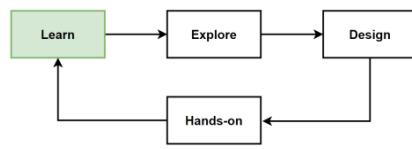
<https://www.datadoghq.com/container-report/#2>



Serverless Kubernetes grow for all major public clouds

- Datadog Report: Serverless Kubernetes grow for all major public clouds
- With over 35% of organizations that deal with containers now using serverless container technology, it's clear that this approach is gaining traction.
- The simplification and cost-effectiveness offered by serverless are proving to be highly attractive.

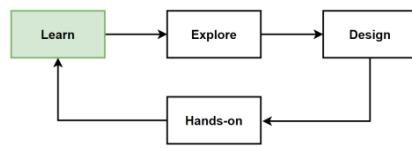




AWS Compute Services: AWS App Runner, ECS, EKS, Fargate, AWS Lambda

- **AWS AppRunner > Amazon ECS > Amazon EKS > AWS Fargate > AWS Lambda**
- Abstraction of managed services on cloud that can deploy and run containers without thinking any server configurations.
- **AWS App Runner**
Fully managed service that makes it easy to build, deploy, and scale containerized applications quickly. You can use App Runner to deploy applications from source code or container images
- **Amazon Elastic Container Service (ECS)**
Fully managed container orchestration service that makes it easy to deploy, manage, and scale containerized applications using Docker containers. Run ECS on Amazon EC2 instances or AWS Fargate, a serverless compute engine for containers.
- **Amazon Kubernetes Service (EKS)**
EKS is a managed Kubernetes service makes it easy to deploy, manage, and scale containerized applications using Kubernetes.
- EKS integrates with various AWS services, such as Elastic Load Balancing, Amazon RDS, and AWS Identity and Access Management, to provide a seamless container management experience.





AWS Compute Services: AWS App Runner, ECS, EKS, Fargate, AWS Lambda

- **AWS AppRunner > Amazon ECS > Amazon EKS > AWS Fargate > AWS Lambda**

- **AWS Fargate**

Fargate is a serverless compute engine for containers that lets you run containers without needing to manage the underlying infrastructure. Focus on building and deploying apps, while AWS takes care of provisioning, scaling, and managing the underlying infrastructure.

- **AWS Lambda**

AWS Lambda is a serverless compute service that can run container images as functions in response to events. You can package and deploy your application code as a container image, enabling you to use familiar container tooling and workflows.



Hands-on: Deploying Microservices on Amazon EKS Fargate

Install and use eksctl

Create Amazon EKS Cluster with eksctl

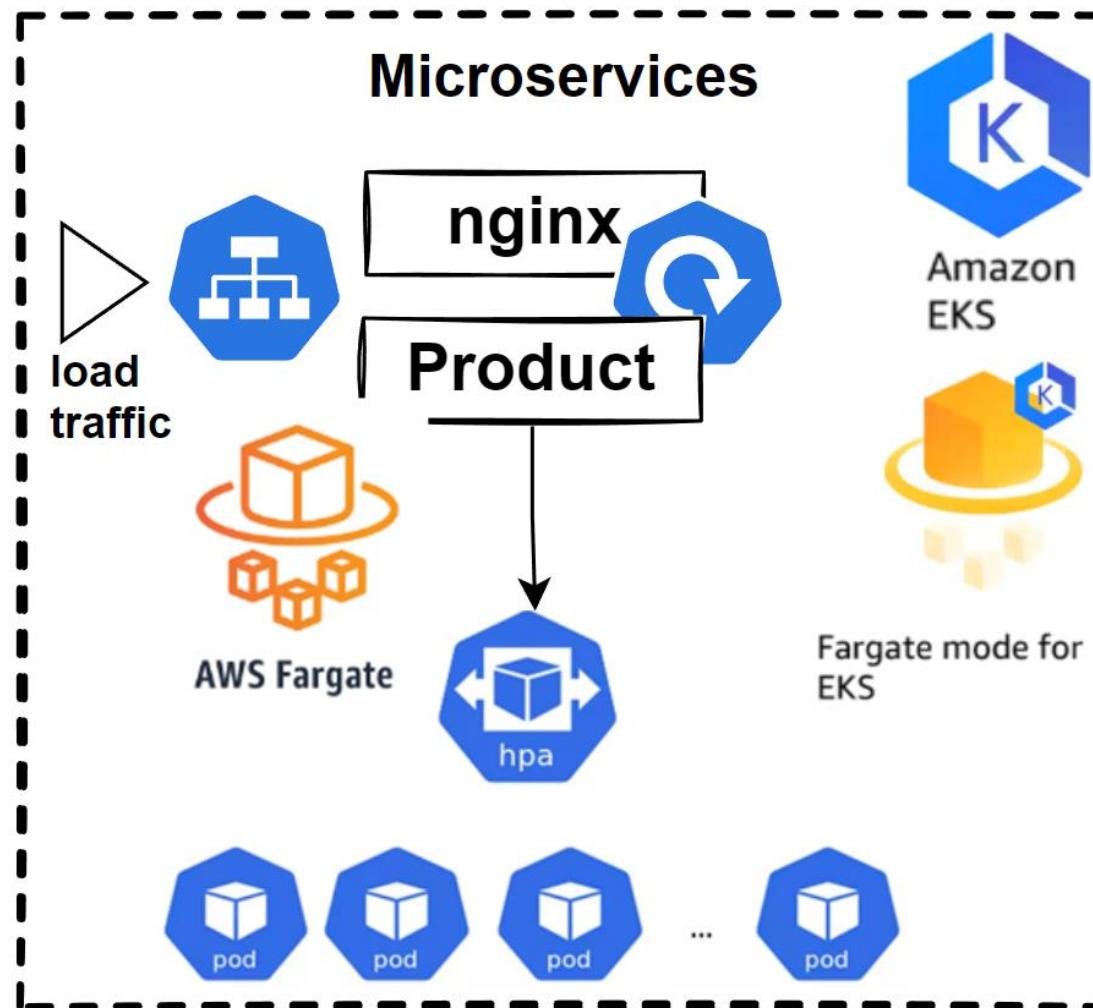
Apply Kubernetes yaml files on EKS Cluster

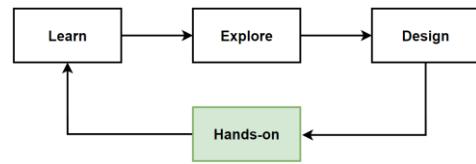
Deploy and Scale Microservices on EKS Cluster with Fargate

Mehmet Ozkaya

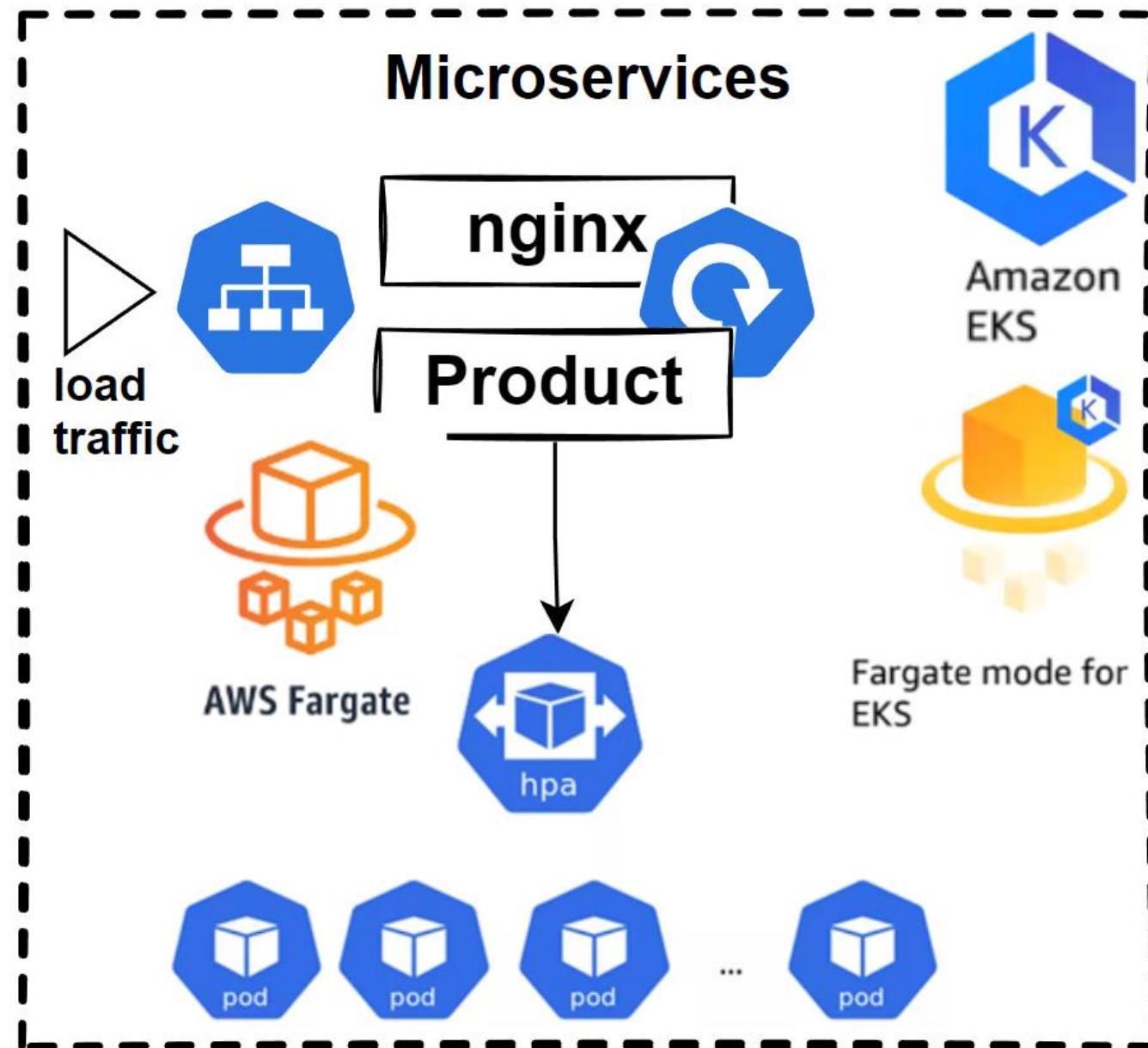
Hands-on: Deploying Microservices on Amazon EKS Fargate— Task List

- Step 1. Installing or updating eksctl to interact Kubernetes Cluster on EKS
- Step 2. Create an EKS Cluster with Fargate using eksctl
- Step 3. Deploy Nginx microservices on EKS Cluster w/ Fargate using eksctl
- Step 4. Create an ECR repository and Push Docker Image to Container Registry
- Step 5. Deploy Product microservices on EKS Cluster w/ Fargate using eksctl
- Step 6. AWS Fargate Auto-scale Deploy Product microservices on EKS
- **Clear Resources - IMPORTANT**





Hands-on: Deploy Amazon EKS Fargate – Architecture



Cloud-Native Pillar7: Devops, CI/CD, IaC and GitOps

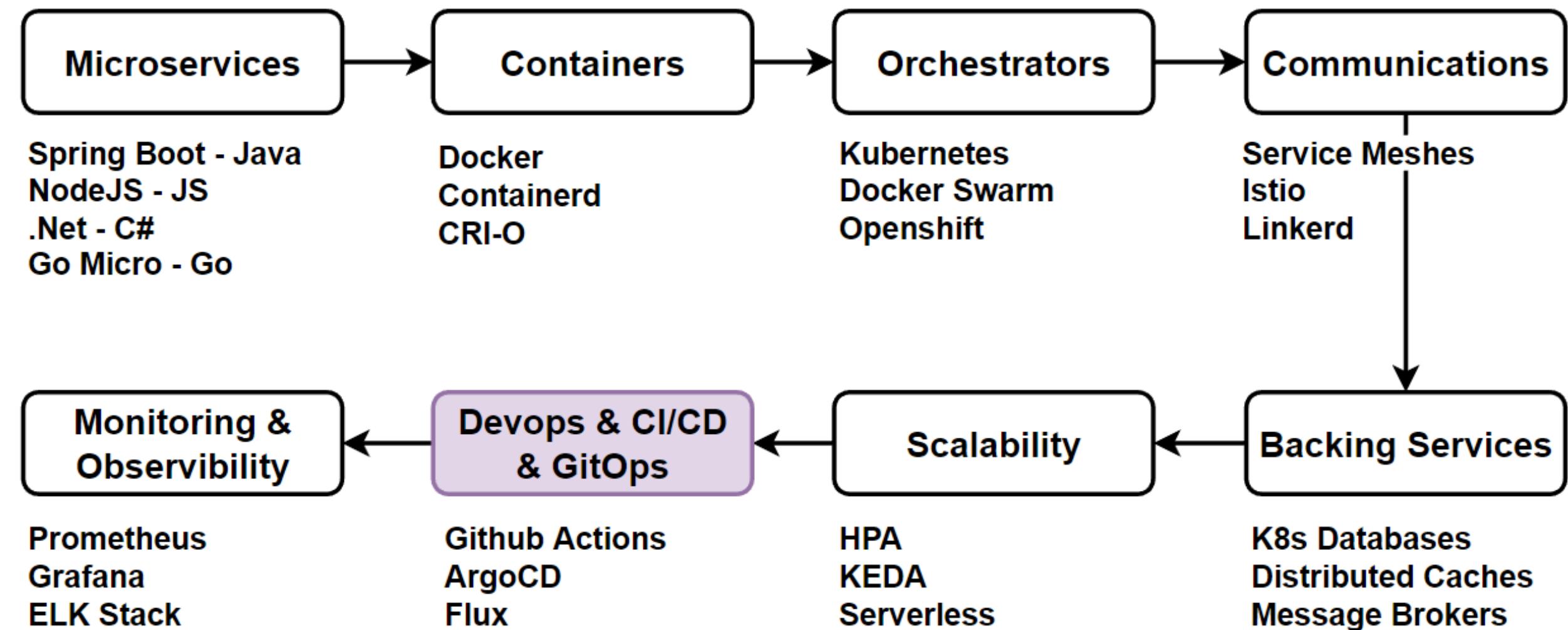
Implementing CI/CD pipelines to automate the building, testing, and deployment of microservices-based apps.

What are Devops CI/CD ?

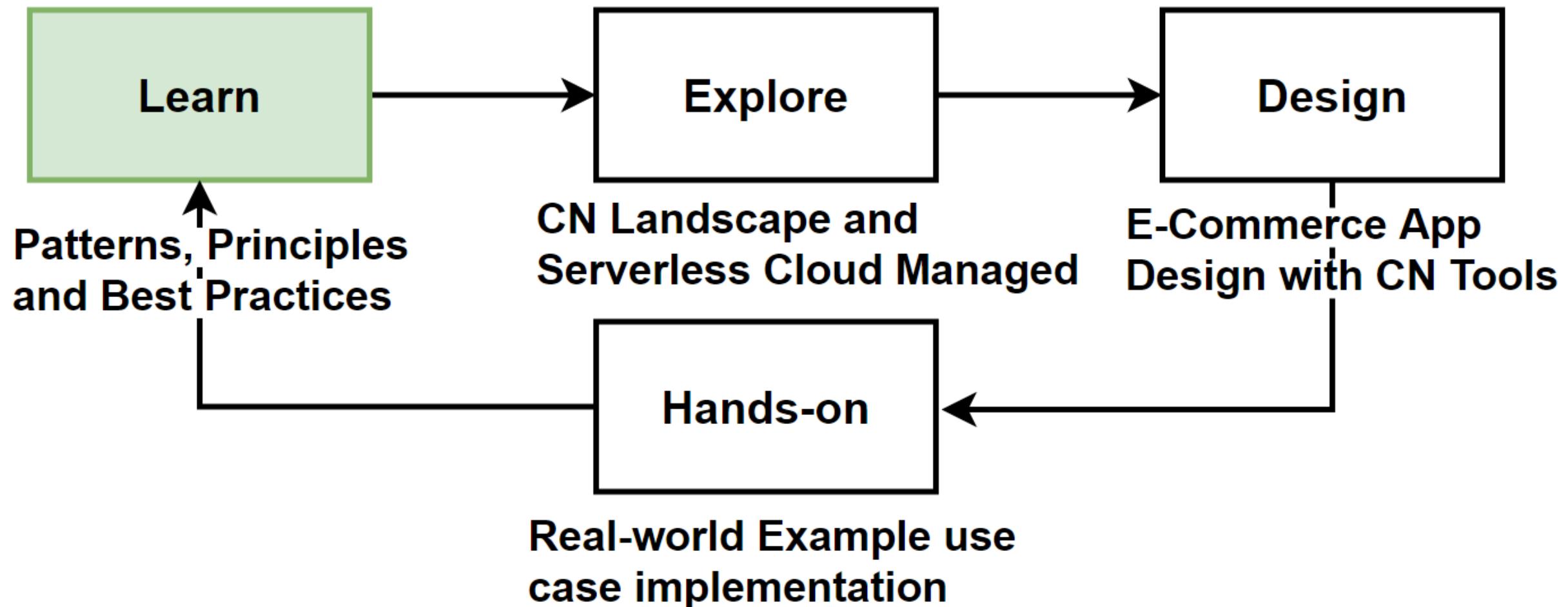
What are patterns & best practices of Devops and CI/CD in CN microservices ?

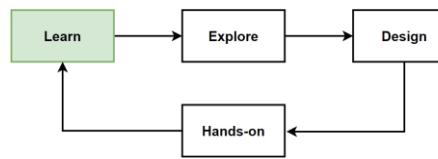
What are IaC ? How IaC uses to create Kubernetes clusters ?

Cloud-Native Pillars Map – The Course Section Map



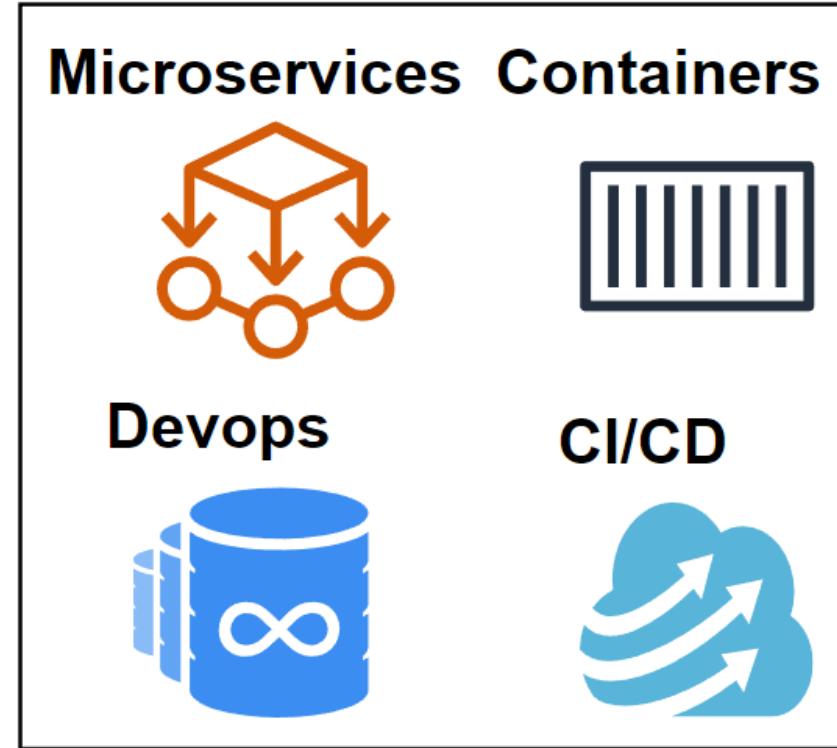
Way of Learning – The Course Flow

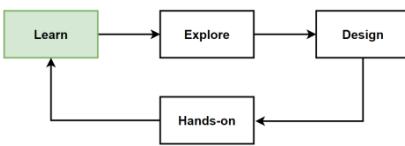




Learn: The 7. Pillar – Devops, CI/CD, IaC and GitOps

- What are Devops CI/CD ?
- How Devops and CI/CD applied in Cloud-Native microservices ?
- What are patterns & best practices of Devops and CI/CD in CN microservices ?
- What are IaC ?
- How IaC uses to create Kubernetes clusters ?
- What is GitOps ?
- How GitOps uses in Cloud-Native Microservices deployments ?
- Explore Devops CI/CD tools
- Implement Hands-on labs for Devops and CI/CD in CN Kubernetes cluster





Where «Devops» in 12-Factor App and CN Trial Map

Twelve-Factor App

- Codebase
- Dependencies
- Config
- Backing Services
- Build, release and run
- Processes
- Port Binding
- Concurrency
- Disposability
- Development/Prod Parity
- Logs
- Admin Processes



CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape <https://github.com/cncf/landscape> has a growing number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator <https://www.cncf.io/training>

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider <http://cncf.io/kcsp>

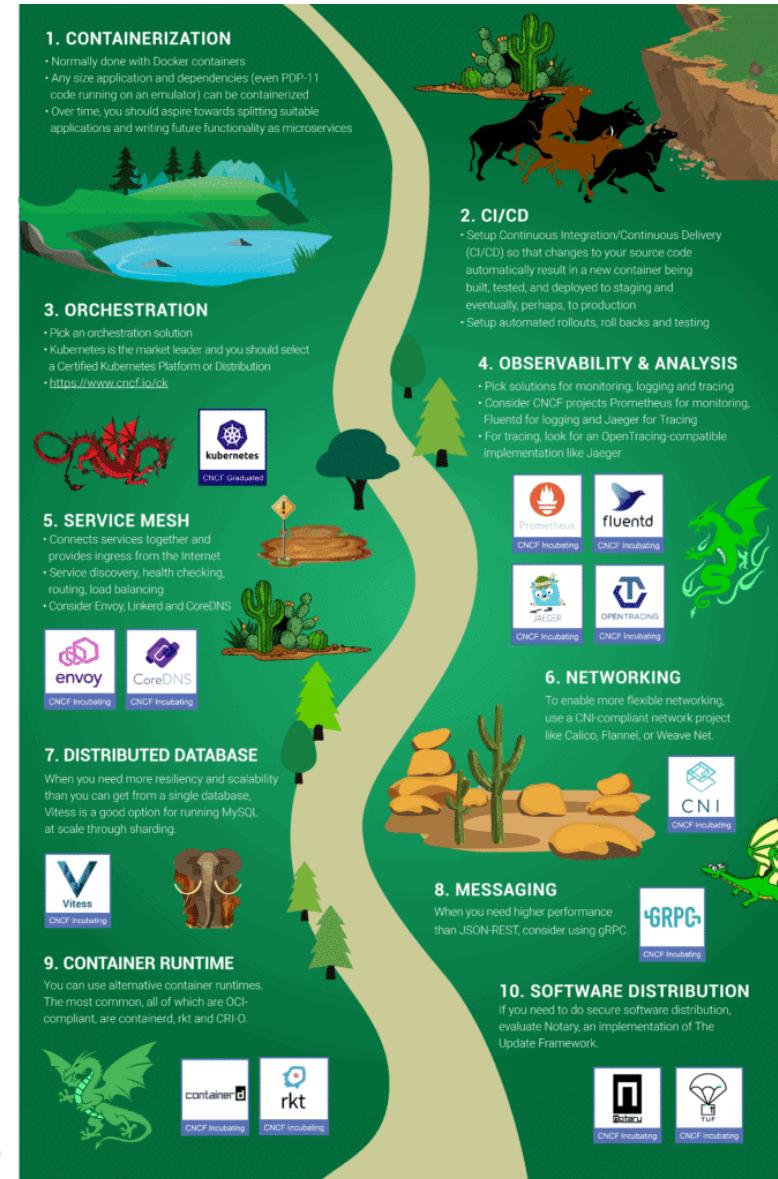
C. Join CNCF's End User Community

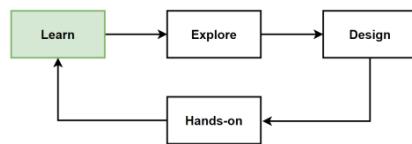
For companies that don't offer cloud native services externally <http://cncf.io/enduser>

WHAT IS CLOUD NATIVE?

- **Operability:** Expose control of application/system lifecycle.
- **Observability:** Provide meaningful signals for observing state, health, and performance.
- **Elasticity:** Grow and shrink to fit in available resources and to meet fluctuating demand.
- **Resilience:** Fast automatic recovery from failures.
- **Agility:** Fast deployment, iteration, and reconfiguration.

www.cncf.io
info@cncf.io





12-Factor App – Cloud-Native Devops and CI/CD

I. Codebase

- A single codebase tracked in version control, with many deploys.
- Importance of using version control systems like Git, which is a foundation for CI/CD pipelines.

II. Dependencies

- Explicitly declare and isolate dependencies.
- Using dependency management tools helps manage dependencies during the build process in a CI/CD pipeline.

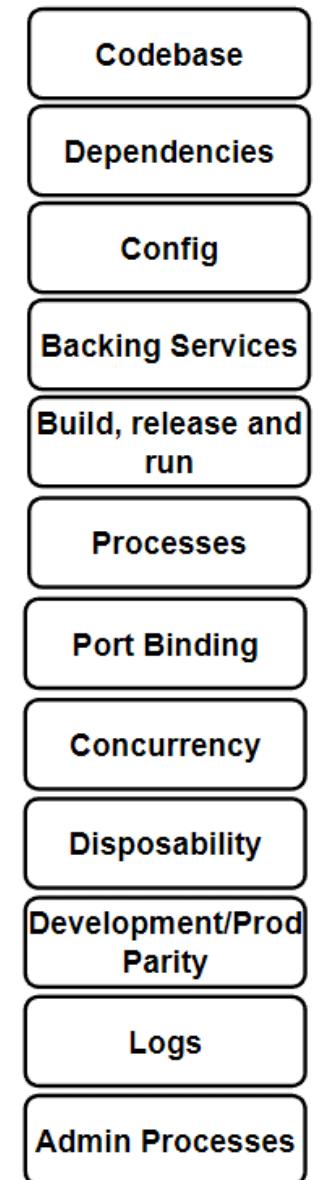
III. Config

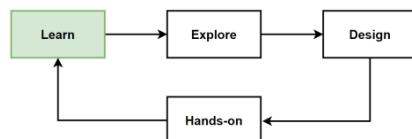
- Store configuration in the environment.
- Separate configuration from the codebase is essential for deploying the same codebase across different env (development, staging, production) in a CI/CD pipeline.

V. Build, release, run

- Strictly separate build and run stages.
- Importance of separating build (compile, package, etc.) and run (deployment) stages in a CI/CD pipeline.

Twelve-Factor App





Cloud-native Trial Map–Devops and CI/CD

Containerization

- Packaging applications into containers is a fundamental step.
- Containers enable consistent deployments across environments and simplify CI/CD pipelines.

CI/CD

- Continuous Integration and Continuous Deployment (CI/CD) is a core practice in cloud-native DevOps.
- CI/CD pipelines automate the process of building, testing, and deploying applications, making it faster and more reliable.

Orchestration & Application Definition

- Kubernetes play a crucial role in automating deployments, scaling, and managing the application lifecycle in cloud-native CI/CD pipelines.

Observability & Analysis

- Monitoring, logging, and tracing tools integrate into CI/CD pipelines helps maintain application health and enables teams to quickly identify and fix problems.



CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape <https://github.com/cncf/landscape> has a growing number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator <https://www.cncf.io/training>

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider <http://cncf.io/kcsp>

C. Join CNCF's End User Community

For companies that don't offer cloud native services externally <http://cncf.io/enduser>

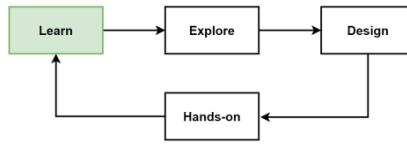
WHAT IS CLOUD NATIVE?

- Operability: Expose control of application/system lifecycle.
- Observability: Provide meaningful signals for observing state, health, and performance.
- Elasticity: Grow and shrink to fit in available resources and to meet fluctuating demand.
- Resilience: Fast automatic recovery from failures.
- Agility: Fast deployment, iteration, and reconfiguration.

www.cncf.io
info@cncf.io



<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>

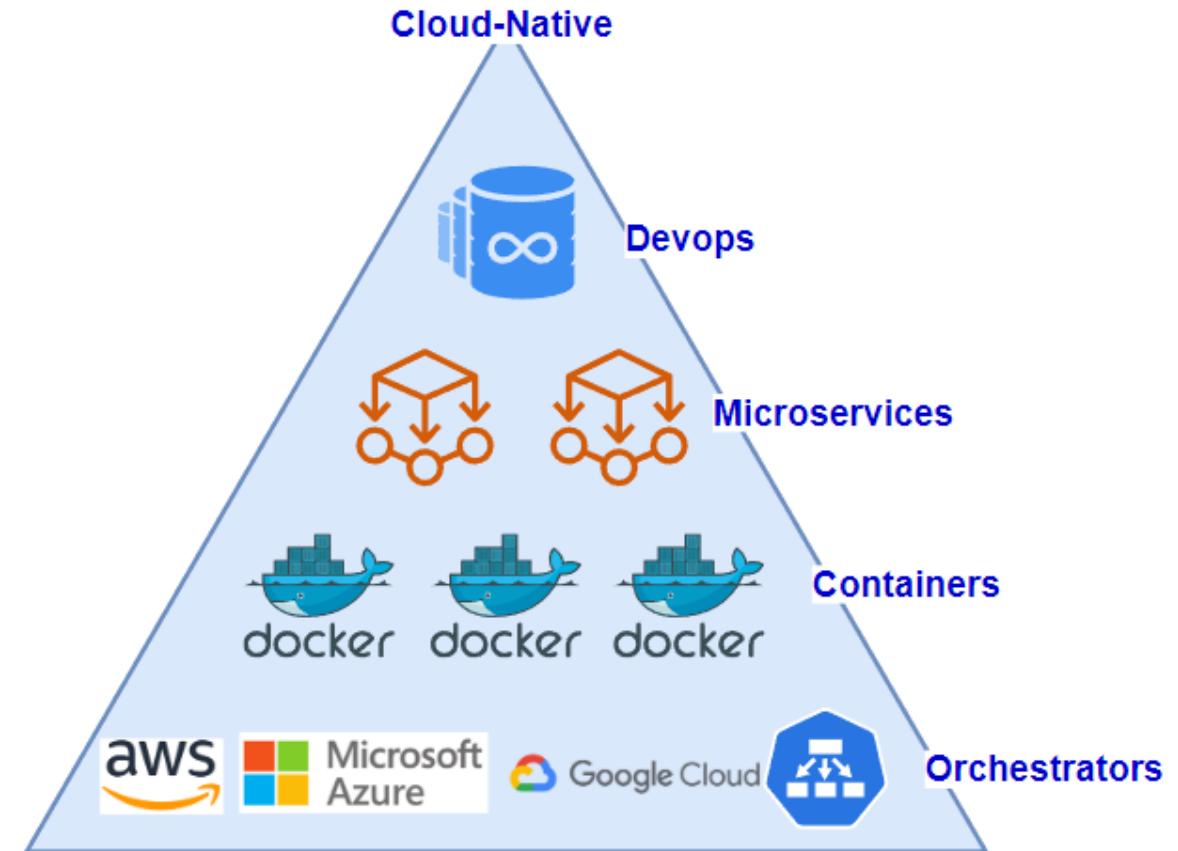


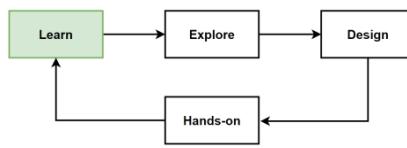
Devops in Cloud-native Applications

- Devops is huge topic in Cloud-native Applications.

Devops Topics divided by 4:

- Devops/DevSecOps
- CI/CD
- IaC
- GitOps





Overview of Devops in Cloud-native Applications

Devops/DevSecOps

- **DevOps** is a set of practices that aim to improve collaboration between development (Dev) and operations (Ops) teams.
- **DevSecOps** extends DevOps by integrating security practices into the lifecycle.

CI/CD

- **Continuous Integration (CI)** involves the practice of integrating code changes from multiple contributors into a shared repository multiple times a day.
- **Continuous Delivery (CD)** is the practice of automatically deploying code to production in small increments.

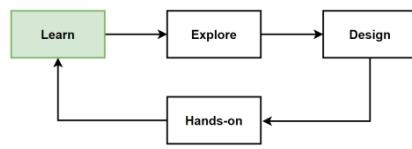
IaC

- **IaC** is the practice of managing and provisioning infrastructure through code.

GitOps

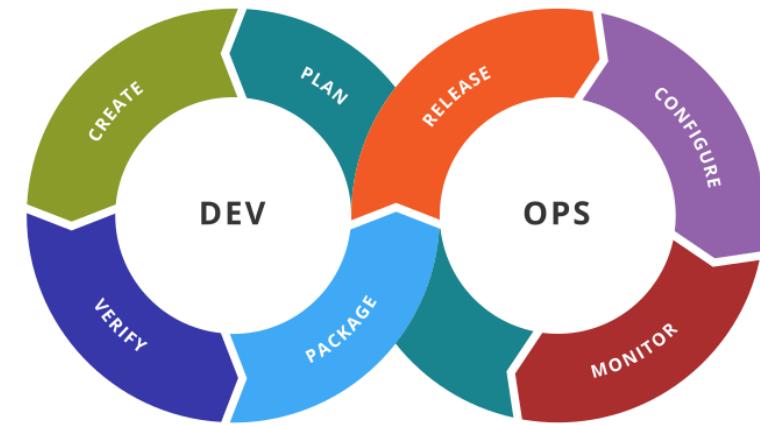
- **GitOps** is a way of implementing Continuous Deployment for cloud-native applications.

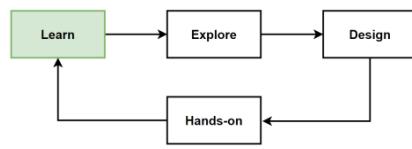
Devops/DevSecOps



What is Devops ?

- **DevOps** is a software development approach that emphasizes collaboration between development and operations teams.
- DevOps is a **set of practices, principles** that aims to improve collaboration and communication between **software development (Dev) and IT operations (Ops) teams**.
- DevOps focuses on **automating** and **streamlining** the **processes** of **software development** and infrastructure management, so that **building**, **testing**, and **releasing** software can happen more rapidly, reliably, and frequently.
- **DevOps** is to **improve the efficiency and speed** of software **development** and **deployment** by **automating** and streamlining the processes involved.
- One common tool used in DevOps is a **Continuous integration/ Continuous delivery (CI/CD)** pipeline.





Why DevOps in Cloud-Native Microservices ?

Accelerated Delivery

- Microservices architecture breaks down applications into small, loosely coupled services.
- DevOps enables faster development and release cycles for each of these services, leading to quicker feature releases and updates.

Reliability and High Availability

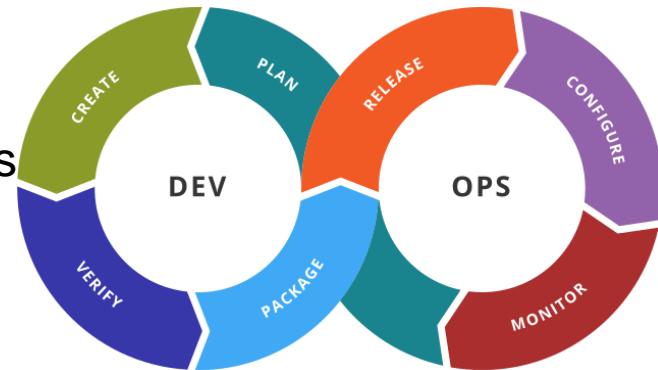
- With CI/CD, and automated testing, DevOps ensures that cloud-native microservices are reliable and available.
- Minimize downtime during app updates and enable rapid recovery in case of failure.

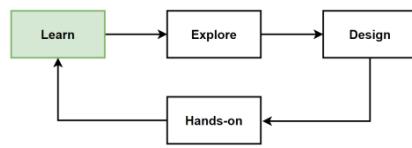
Collaboration and Culture

- In Microservices, it's essential for teams to work together as services are often interdependent.
- A collaborative culture ensures smoother development, deployment, and maintenance processes.

Resource Optimization

- Resources can be allocated and released dynamically. DevOps practices ensure that microservices use resources optimally, leading to cost savings.





How DevOps is Used in Cloud-Native Microservices

Containers and Orchestration

- DevOps uses container orchestration tools like Kubernetes for automating the deployment, scaling, and management of containers.

Continuous Integration/Continuous Deployment (CI/CD)

- Automate the process of code integration, testing, and deployment, ensuring that each service can be developed and deployed independently.

Infrastructure as Code (IaC)

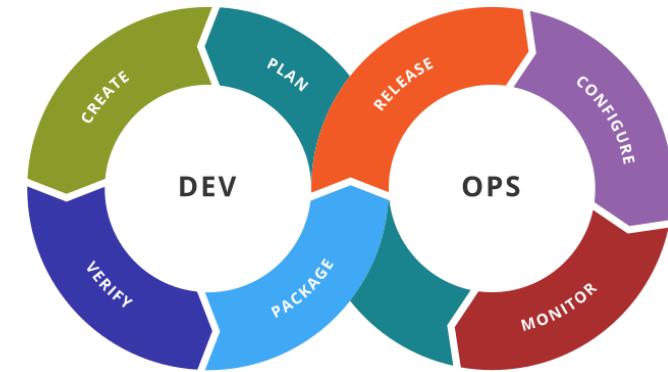
- IaC allows for the provisioning and management of cloud infrastructure using code and automation.

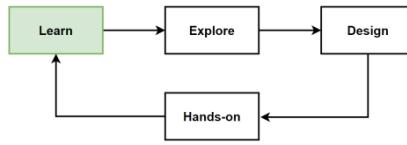
Monitoring and Logging

- With microservices, monitoring and logging are crucial for gaining insights into each service's performance and state. DevOps introduces tools and practices.

Service Discovery and Configuration Management

- DevOps tools for service discovery and configuration management ensure that services are able to communicate and share data efficiently.

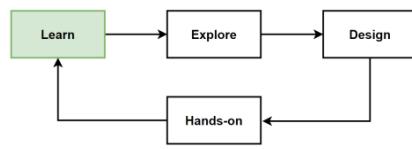




What is DevSecOps ?

- **DevSecOps** is a practice that integrates security into the DevOps process, combination of "Development," "Security," and "Operations."
- **DevSecOps** is the **integration of security practices** into the DevOps process, involves implementing **security at every stage**.
- Aims to **embed security practices** and principles into the **entire software development and deployment lifecycle**.
- **Contrast to traditional approaches** where security was often considered as an afterthought





DevSecOps Key Aspects

Shift-Left Security

- Integrating security early in the development process, often referred to as “shifting security to the left”
- Security considerations are taken into account from the beginning, during the planning and design phases.

Automated Security Checks

- Automated security scanning and testing tools, teams can detect vulnerabilities and security issues early and efficiently.

Continuous Monitoring and Response

- Ongoing monitoring of apps and infrastructures for security incidents is essential.

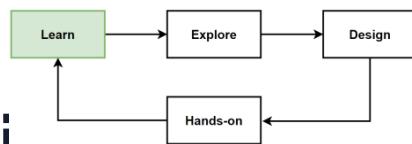
Policy as Code

- Codifying security policies and configurations, and managing them as code.
- Allows for version control, peer reviews, and automation of policy enforcement.

Infrastructure as Code Security

- Ensuring the security of the infrastructure by managing infrastructure as code and integrating security checks and policies into the infrastructure provisioning process.





DevOps Stages: Software Development and Deployment Lifecycle

Plan

- Define the goals and requirements for the app what needs to be built, define tasks, and allocate resources.
- Jira and Trello are often used for project management and planning.

Code

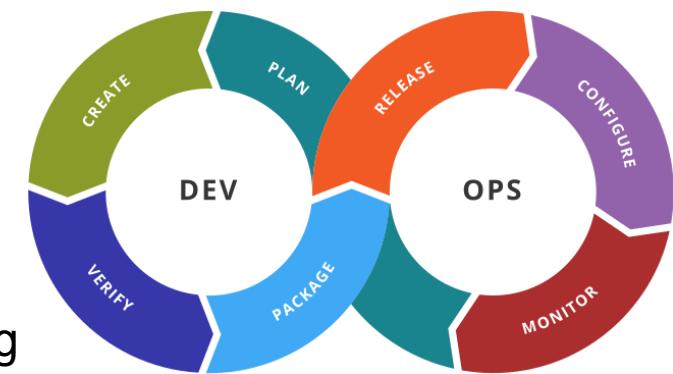
- This stage involves the actual writing of source code.
- Version control systems like Git are essential in this stage to manage changes.

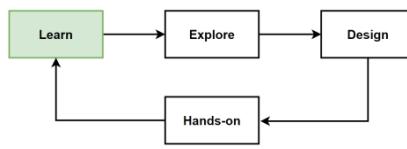
Build

- Compiling the source code into executable code, involve packaging code, managing dependencies, and creating builds or containers.
- Maven, Gradle, and Docker are common in this stage.

Test

- Running various tests (unit tests, integration tests, performance tests, etc.) to identify and fix bugs and ensure the software meets requirements.
- Common testing tools include JUnit, Selenium, and TestNG.





DevOps Stages - 2

Release

- Continuous Integration (CI) systems automate the build and testing processes and help prepare a release.
- Jenkins, GitLab CI/CD, and CircleCI are used to automate the release process.

Deploy

- Deploying the application to a production environment.
- Kubernetes, Ansible, and Terraform are often used for deployment.

Operate

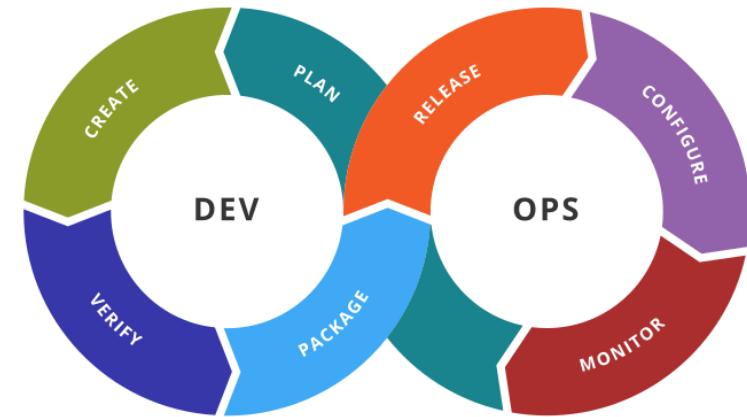
- Monitoring the application, scaling resources, and ensuring high availability.
- Kubernetes is a popular tool for managing deployed applications.

Monitor and Observe

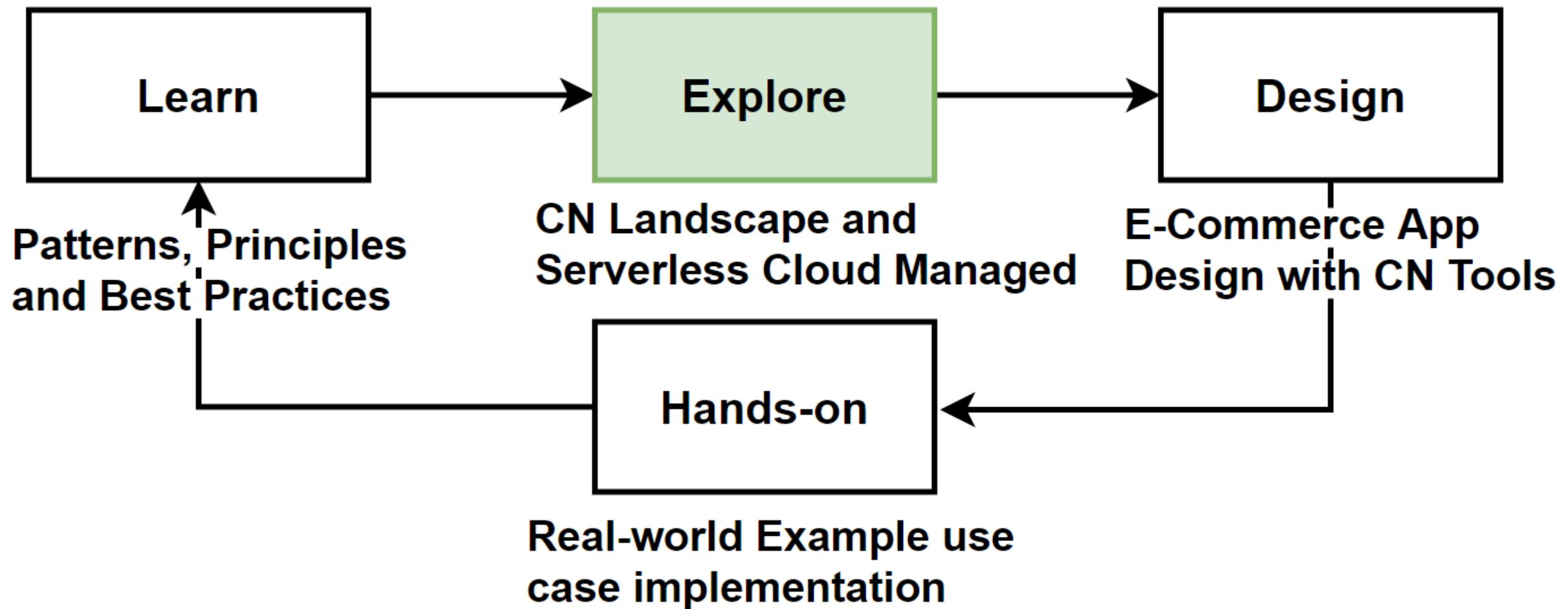
- Monitoring involves keeping an eye on the app's performance and functionality.
- Prometheus, Grafana, and the ELK Stack are used for monitoring and logging.

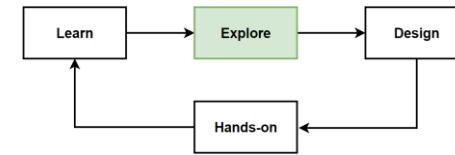
Security (DevSecOps)

- Code analysis, vulnerability scanning, and implementing security policies.
- Tools like SonarQube and OWASP Zap can be used for security analysis.



Explore: Devops Tools





Explore: Devops Tools

Code

- **Git**: Version control system used for source code management.
- **Bitbucket**: Git code management and collaboration tool.
- **GitHub**: Web-based hosting service for version control and collaboration.

Build

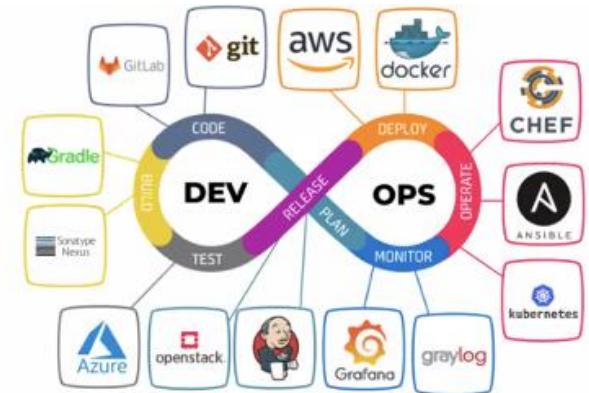
- **Maven**: Build automation tool mainly used for Java projects.
- **Gradle**: Build tool based on the concepts of Apache Ant and Apache Maven.
- **Docker**: Tool for creating containers that package app code and its dependencies

Test

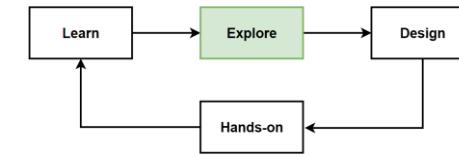
- **JUnit**: Framework to write repeatable tests in Java.
- **Selenium**: Tool for automating web browsers, widely used for testing web applications.

Release

- **Jenkins**: Automation server used for implementing continuous integration and continuous delivery.
- **GitLab CI/CD**: A built-in CI/CD feature of GitLab to automate the software delivery process.



<https://medium.com/clarusway/popular-devops-tools-review-ee0cffea14ec>



Explore: Devops Tools - 2

Deploy

- **Kubernetes:** Container orchestration platform for the deployment, scaling, and management of containerized apps.
- **Terraform:** Infrastructure-as-Code (IaC) tool for provisioning cloud infrastructure.

Operate

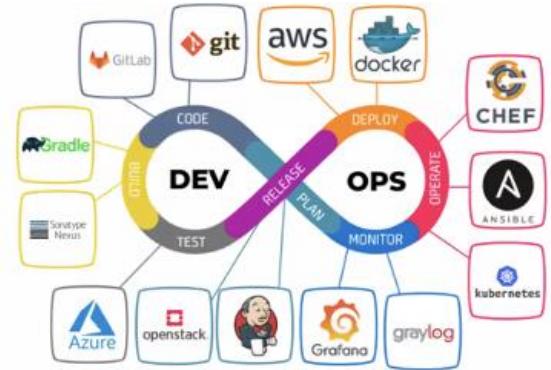
- **Kubernetes:** Automates various operations: scaling, load balancing, and rolling updates.
- **Ansible:** Configuration management and application deployment tool.

Monitor and Observe

- **Prometheus:** Monitoring tool and time-series database.
- **Grafana:** Analytics and monitoring dashboard used for querying, visualizing, and understanding metrics.

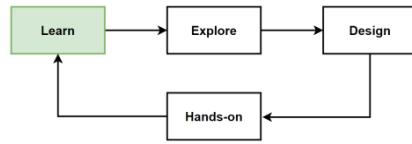
Security (DevSecOps)

- **SonarQube:** Continuous inspection of code quality for detecting bugs, code smells, and security vulnerabilities.
- **Aqua:** Security tool specialized for containers, used to find vulnerabilities and enforce security policies.



<https://medium.com/clarusway/popular-devops-tools-review-everything-you-need-to-know-about-them-1000f3a2a2>

CI/CD Pipelines

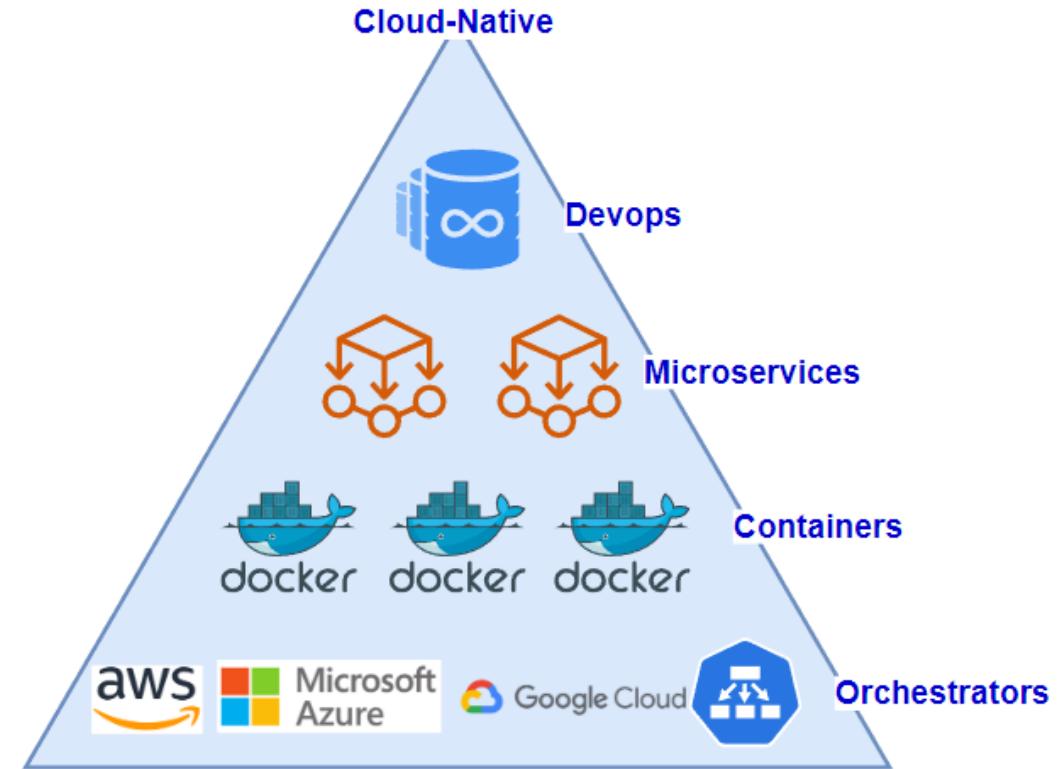


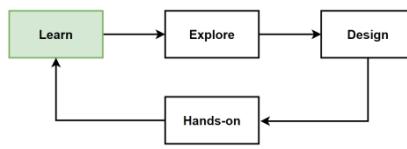
Devops in Cloud-native Applications

- Devops is huge topic in Cloud-native Applications.

Devops Topics divided by 4:

- Devops/DevSecOps
- **CI/CD**
- IaC
- GitOps





What is CI/CD ?

- CI/CD stands for **Continuous Integration** and **Continuous Deployment**, sometimes extended to **CI/CD/CD** including **Continuous Delivery** as well.

Continuous Integration (CI)

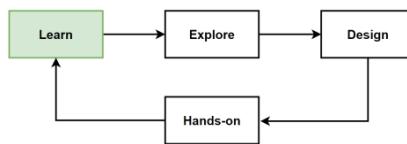
- Developers frequently merge their code changes back to a central repository, after which automated builds and tests are run.
- Find and address bugs quicker, improve software quality, and reduce the time to validate and release new software updates.

Continuous Delivery (CD)

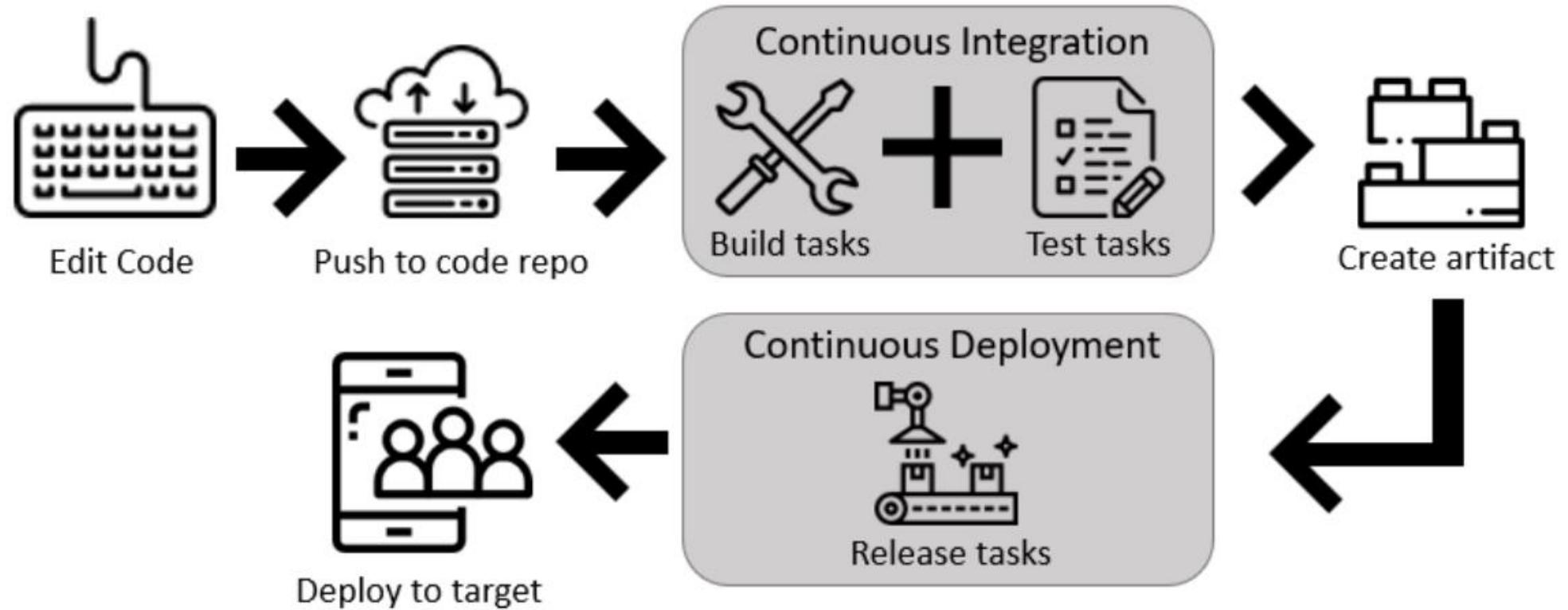
- Continuous Delivery is an extension of continuous integration ensures that the code is always in a deployable state.
- Automated testing beyond unit tests (e.g., user acceptance tests, load tests, vulnerability scans), and automated deployment to staging environments that mirror production.

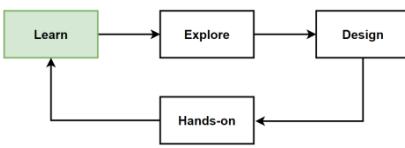
Continuous Deployment (CD)

- Every change that passes all stages of your production pipeline is released to your customers. There's no human intervention.
- Suitable for very mature teams with a high degree of confidence in their automated tests and deployment process.

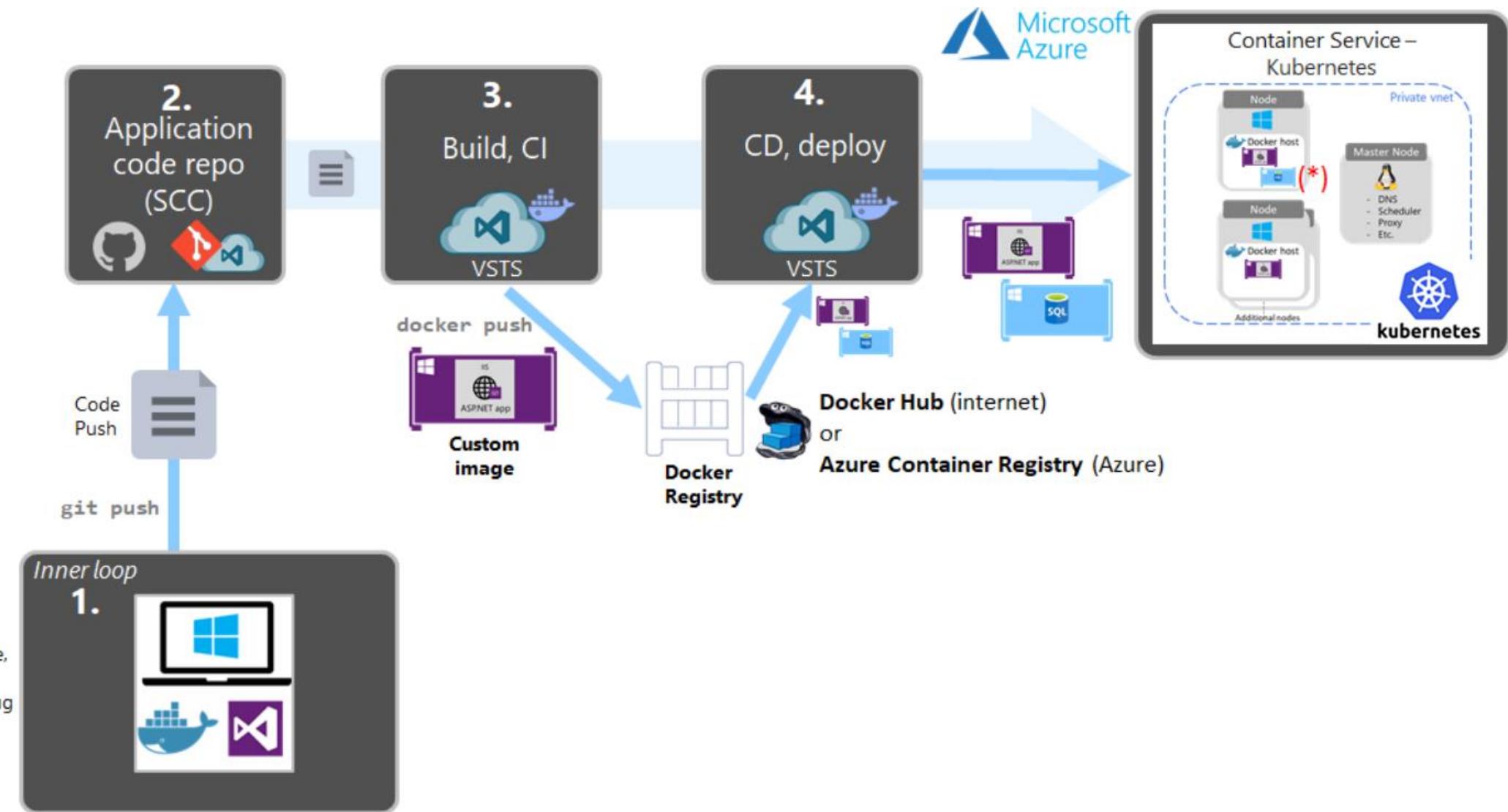


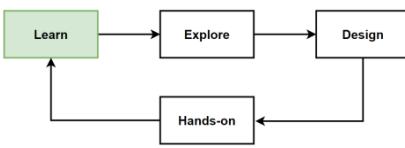
CI/CD Pipeline Steps for Microservices Deployments



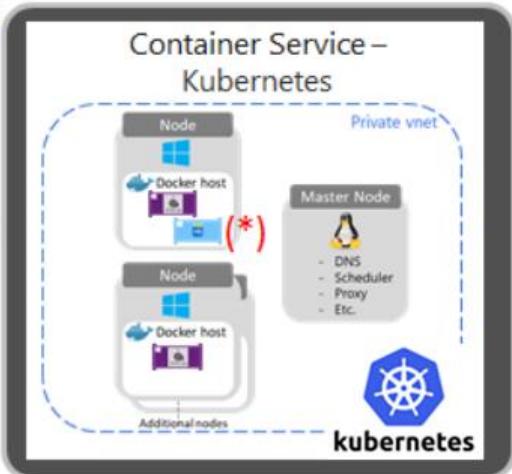
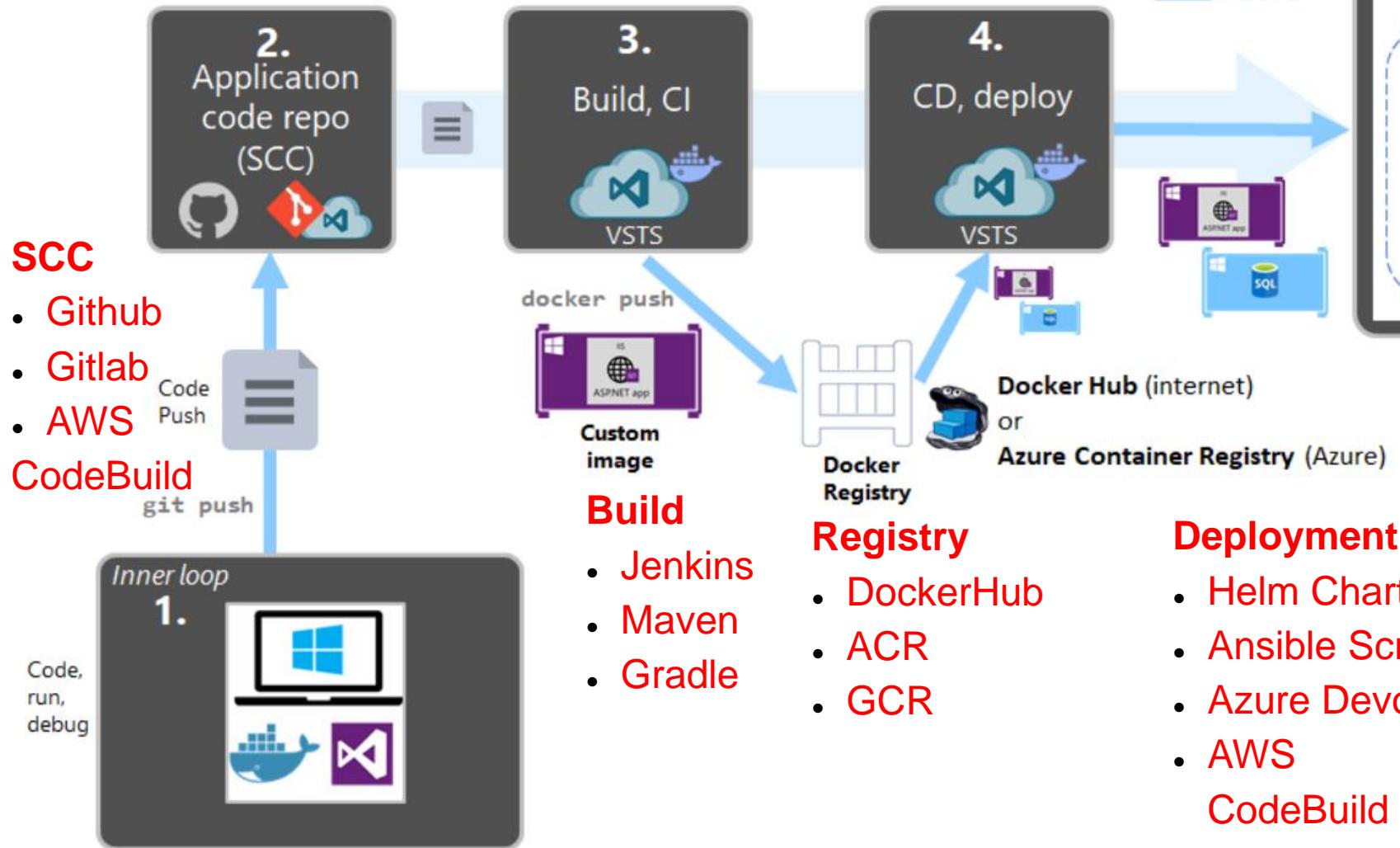


CI/CD Tools for Microservices Deployments

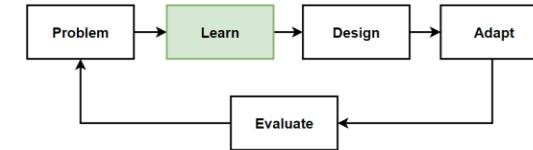




CI/CD Tools for Microservices Deployments



Deployment Strategies for Microservices



- **Blue-green deployment**

Deploying updates to a new set of microservices (the "green" deployment), while the old version of the microservices (the "blue" deployment) remains running.

- **Rolling deployment**

Deploying updates to a subset of the microservices at a time, and then rolling the updates out to the rest of the microservices over time.

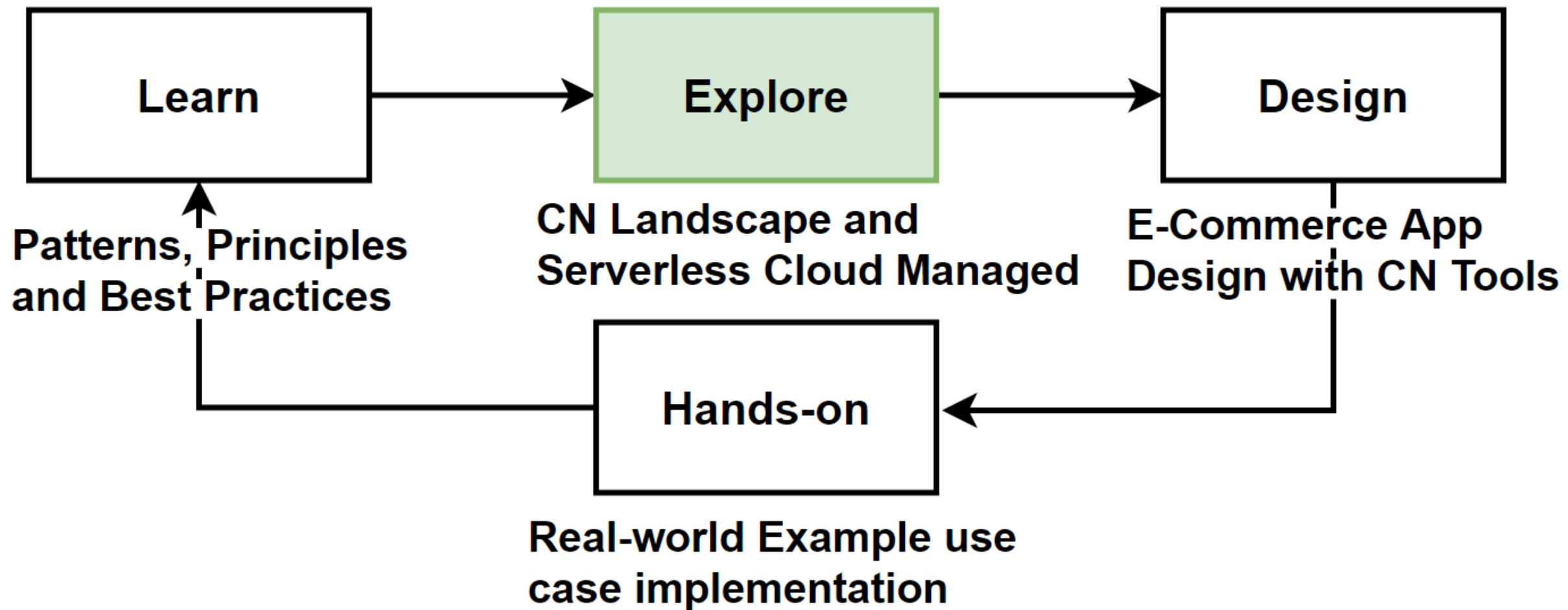
- **Canary deployment**

Deploying updates to a small subset of the microservices, and then gradually rolling the updates out to the rest of the microservices over time.

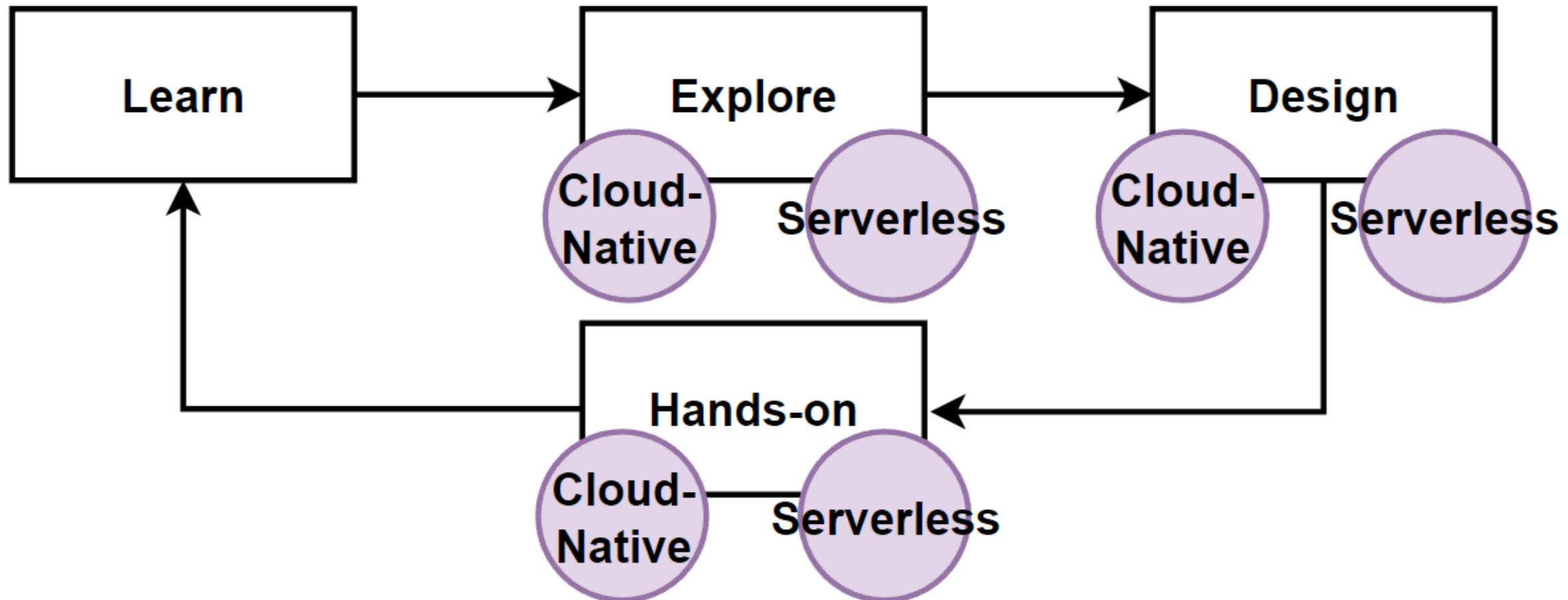
- **A/B testing**

Deploying updates to a subset of the microservices, and then comparing the performance of the updated microservices with the performance of the unmodified microservices.

Explore: Cloud Managed and Serverless Microservices Frameworks

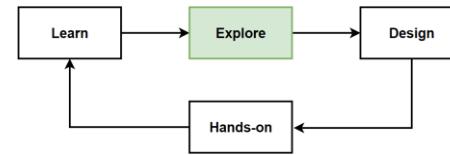


Way of Learning – Cloud-Native & Serverless Cloud Managed Tool



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

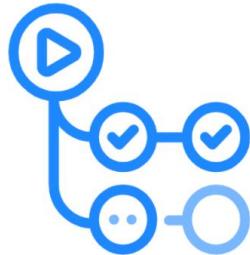


Explore: Cloud-Native CI/CD Pipelines

- Cloud-native CI/CD pipelines are designed to work seamlessly with cloud-based services and are optimized for deploying applications in cloud environments.

Cloud-Native CI/CD Pipelines

- GitHub Actions
- GitLab CI/CD
- Jenkins X
- CircleCI
- Travis CI



GitHub Actions

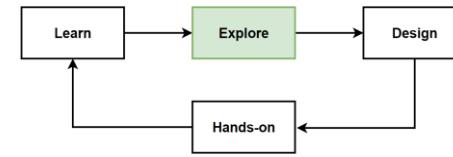
Cloud Serverless CI/CD Pipelines

- AWS CodePipeline
- Azure Pipelines
- Google Cloud Build



Azure Pipelines

Goto -> <https://landscape.cncf.io/>



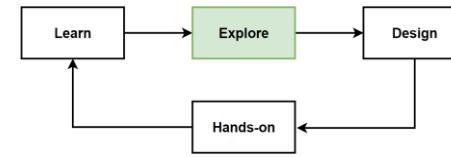
What is GitHub Actions ?

- **GitHub Actions** is a CI/CD (Continuous Integration/Continuous Deployment) and automation service provided by GitHub.
- It allows you to **automate**, **customize**, and **execute** your software development workflows **directly** in your **GitHub repository**.
- **Handle various tasks**: building, testing, and deploying applications, automating issue responses, and much more.
- **Create workflows** that build and test **every pull request** to your repository, or deploy **merged pull requests** to production.
- **Run workflows** when other **events happen** in your repository: i.e. run a workflow to automatically add the valid labels whenever someone creates a new issue in your repository.
- **GitHub Actions** is particularly powerful because it is **deeply integrated with GitHub**.
- **Supports a wide range of languages** and **frameworks** and integrates with many tools through the use of actions.
- Provides **Linux**, **Windows**, and **macOS** virtual machines to run your workflows.



GitHub Actions

	All checks have passed
	4 successful checks
	build Successfully in 59s — build
	test Successfully in 59s — build
	publish Successfully in 59s — build



Components of GitHub Actions

- **Workflows**

Custom automated processes that you can set up in your repository to build, test, package, or deploy any code project on GitHub.

- **Events**

Workflows can start in different events, including pushing code, submitting pull requests, or creating issues.

- **Jobs**

A workflow run is made up of one or more jobs. Jobs run in parallel by default. Each job runs in its own fresh instance of a virtual environment.

- **Steps**

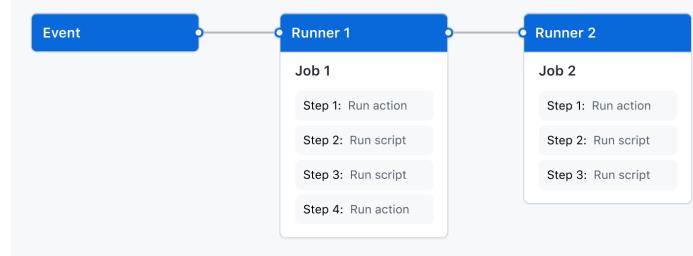
Jobs consist of a series of steps. Each step can be a set of shell commands, a script, or an action. Steps are executed in the same virtual environment.

- **Actions**

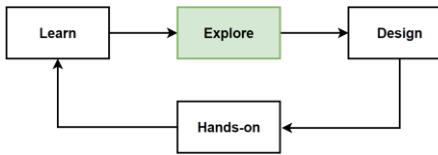
Smallest portable building blocks of a workflow. Can create own actions, use actions shared from the GitHub community, or reuse actions in public repos.

- **Runners**

Servers where the jobs in your workflows run. GitHub provides runners with a standard Linux, Windows, or macOS environment.

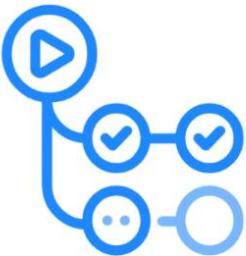
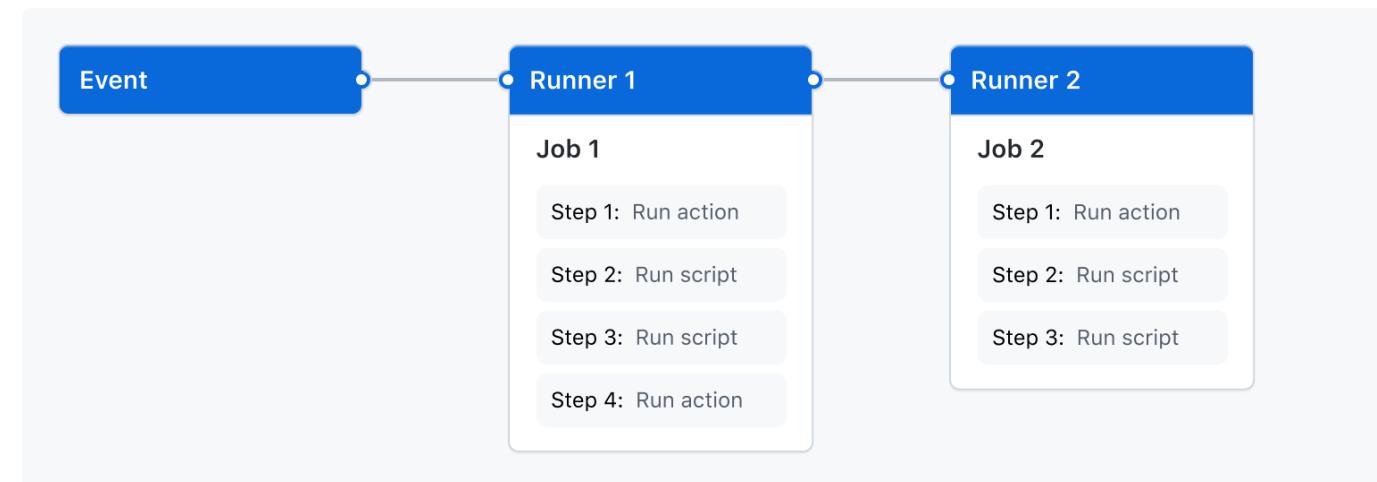


GitHub Actions

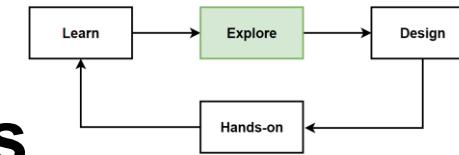


How GitHub Actions Works ?

- Trigger Workflow:** Configure a GitHub Actions workflow to be triggered when an event occurs in your repository, such as a pull request being opened or an issue being created.
- Execute Jobs:** Workflow contains one or more jobs which can run in sequential order or in parallel. Each job will run inside its own virtual machine runner, or inside a container.
- Run Steps Within Jobs:** Each job has one or more steps that either run a script that you define or run an action, which is a reusable extension that can simplify your workflow.

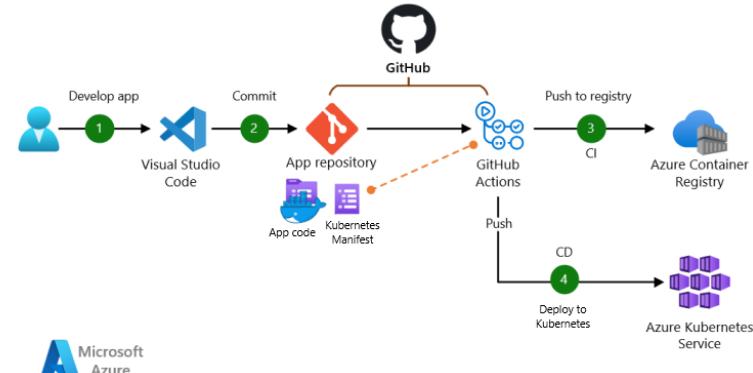


GitHub Actions

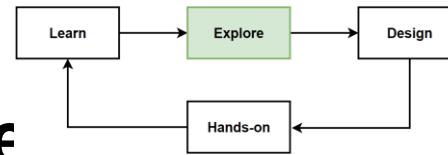


Deploy Microservices to Kubernetes with GitHub Actions

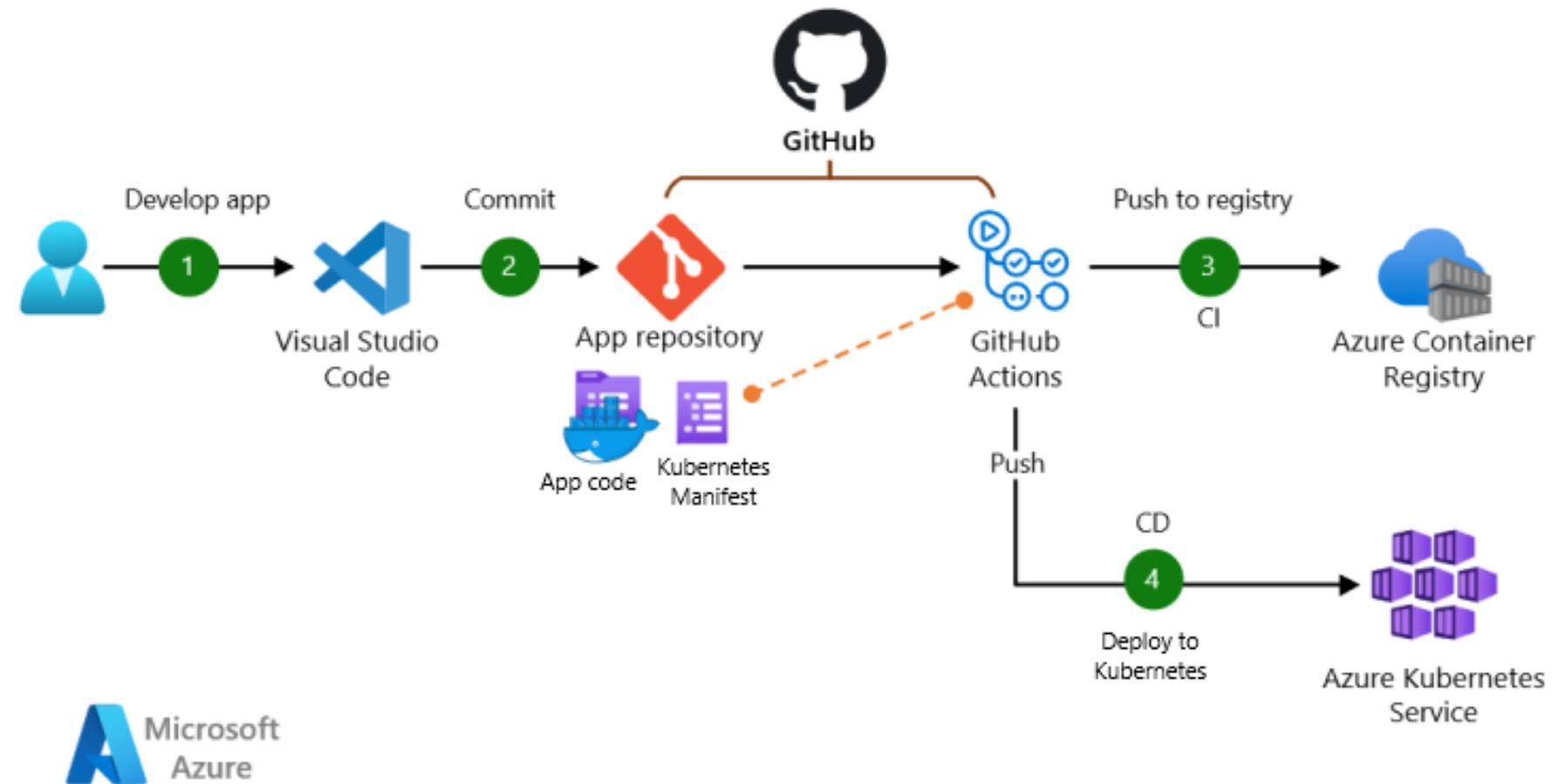
- **Create a Workflow File:** Create a .github/workflows directory in your repository. This file will define the GitHub Actions workflow.
- **Define Workflow Steps:** Inside the workflow file, define the steps GitHub Actions should take. I.e. Checkout code, build container image, push registry..
 - Check Out Code: Use the actions/checkout action
 - Set Up Environment: Docker to build container images.
 - Build Container Image: Build the Docker image for your microservice
 - Push Image to Container Registry: push it to a container registry like Docker Hub
 - Configure Kubernetes CLI: Set up kubectl in the runner environment.
 - Deploy to Kubernetes: Apply your Kubernetes manifests to your cluster.
- **Commit and Push:** Commit the workflow file to your GitHub repository. That trigger the workflow.
- **Monitor the Workflow:** Watch the progress and view the logs in the “Actions” tab of your GitHub repository.



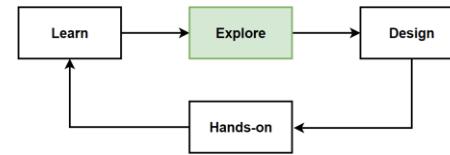
<https://learn.microsoft.com/en-us/azure/architecture/guide/aks/aks-cicd-github-actions-and-gitops>



Deploy Microservices to AKS with GitHub Actions - Azure



<https://learn.microsoft.com/en-us/azure/architecture/guide/aks/aks-cicd-github-actions-and-gitops>

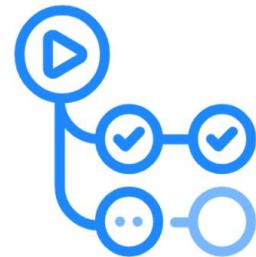


Explore: Cloud-Native CI/CD Pipelines

- Horizontally Scalable Distributed Message Brokers that designed to work seamlessly with cloud-native architectures and scale out by adding more nodes to the system.

Cloud-Native CI/CD Pipelines

- GitHub Actions
- GitLab CI/CD
- Jenkins X
- CircleCI
- Travis CI



GitHub Actions

Cloud Serverless CI/CD Pipelines

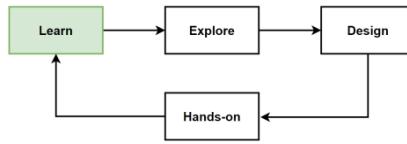
- AWS CodePipeline
- Azure Pipelines
- Google Cloud Build



Azure Pipelines

Goto -> <https://landscape.cncf.io/>

IaC – Infrastructure as Code

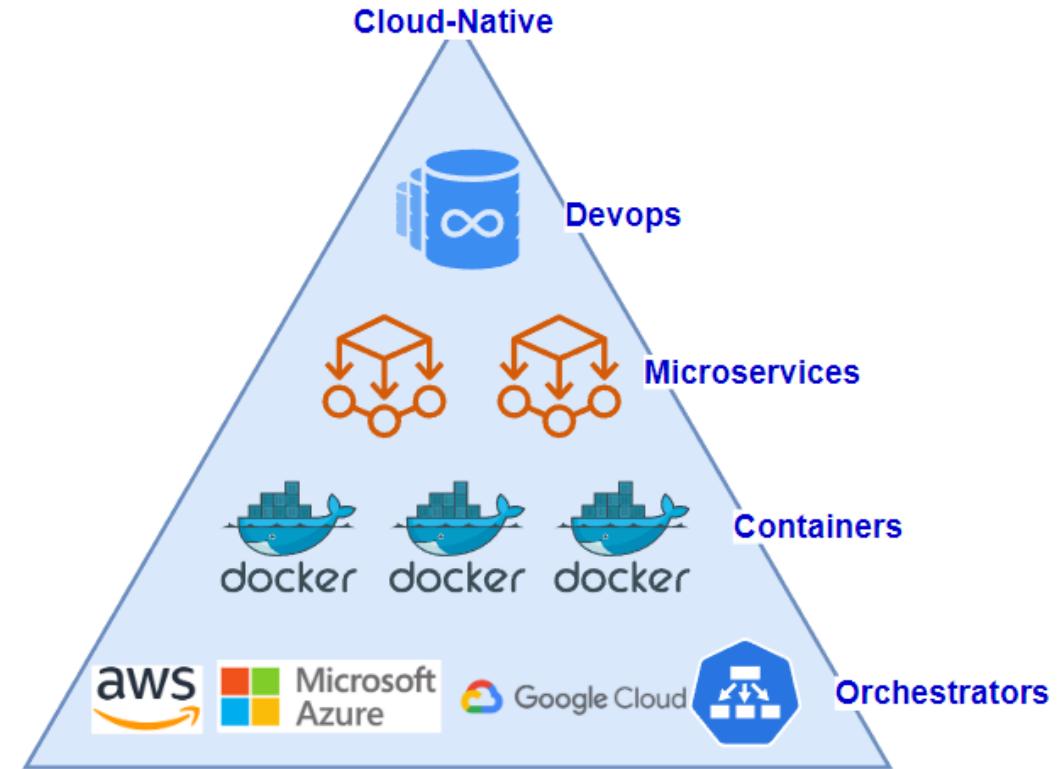


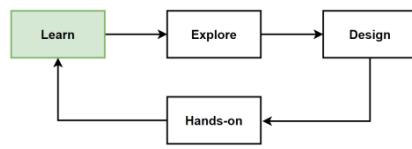
Devops in Cloud-native Applications

- Devops is huge topic in Cloud-native Applications.

Devops Topics divided by 4:

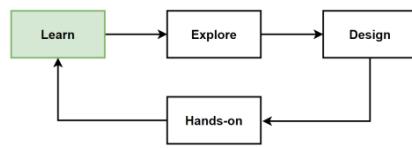
- Devops/DevSecOps
- CI/CD
- **IaC**
- GitOps





Infrastructure as Code (IaC)

- Infrastructure as a codebase that can be managed and versioned in the same way as application code.
- IaC is to enable teams to manage their infrastructure in a more automated and repeatable way.
- IaC can be used to automate the process of deploying and managing the infrastructure needed to run the microservices.
- I.e. define the infrastructure needed to run your microservices in a Kubernetes cluster, including the pods, services, and other resources required.
- Terraform is a tool that allows you to define and manage infrastructure as code.
- Ansible is a tool that allows you to automate the deployment and management of infrastructure.
- They can be used to define the infrastructure needed to run your microservices in a Kubernetes cluster.
- IaC is a useful tool for automating the process of deploying and managing the infrastructure.
- Teams can manage their infrastructure in a more repeatable and automated way, making it easier to update and maintain their microservices over time.



IaC usage in Cloud-native Microservices

- **Rapid Provisioning of Infrastructure**

Microservices need various infrastructure components like databases, message brokers, and caches. IaC allows to script and automate the provisioning of components rapidly.

- **Consistency Across Environments**

Microservices are deployed in different environments like development, staging, and prod. IaC provides all environments are consistent by using the same code to provision the infra.

- **Immutable Infrastructure**

Instead of updating the existing infra, new infra is provisioned for each release, and the old infra is terminated. Reduces inconsistencies and makes deployments more predictable.

- **Rollbacks and Version Control**

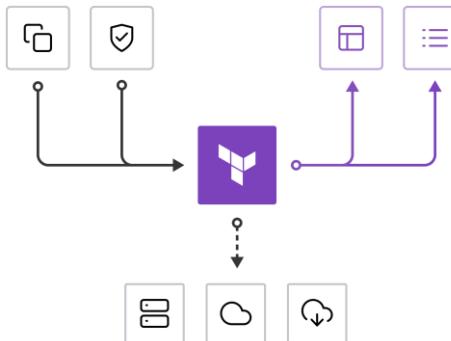
Versioning of infrastructure configurations, which is vital for rollback in case of failure.

- **Integration with CI/CD Pipelines**

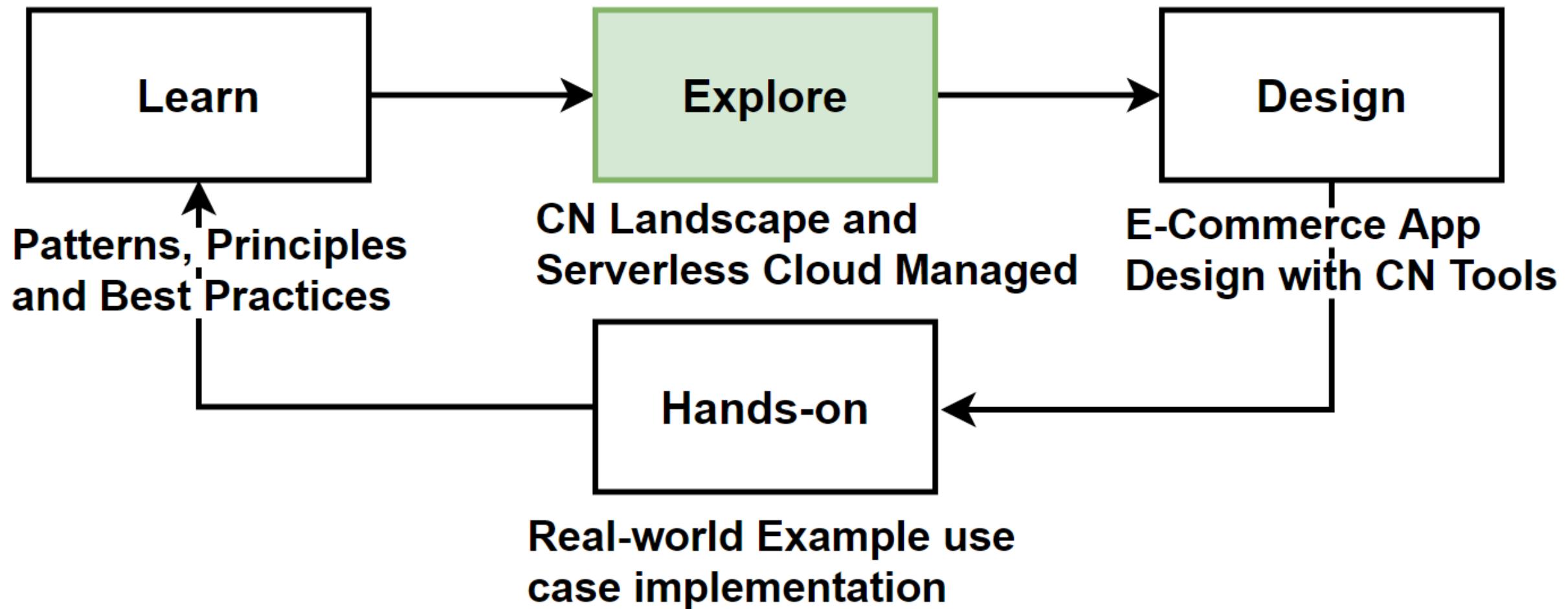
IaC is integrated into CI/CD pipelines to handle infrastructure setup and configuration as part of the automated release process.

- **Managing Configuration and Secrets**

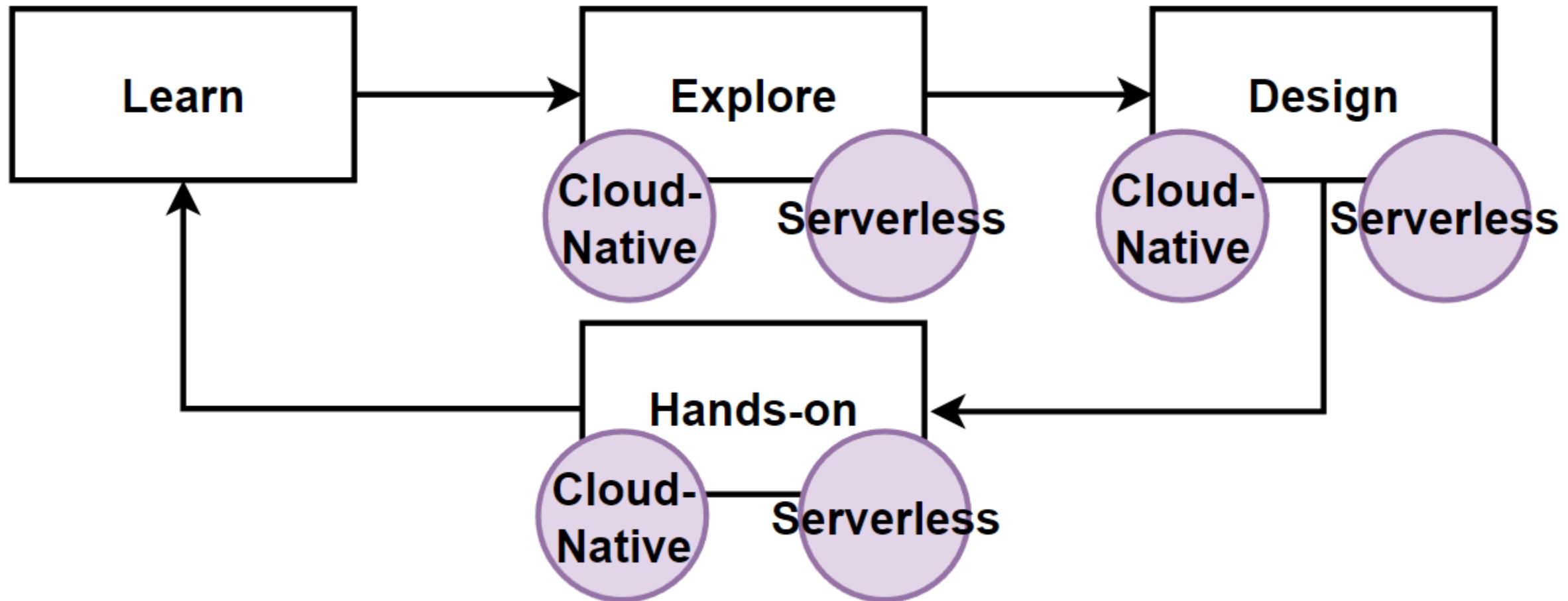
IaC used to automate the management of configurations and secrets, ensuring that they are securely injected into the services when needed.



Explore: Cloud Managed and Serverless Microservices Frameworks

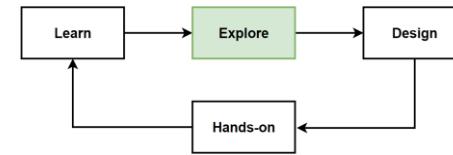


Way of Learning – Cloud-Native & Serverless Cloud Managed Tool



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Explore: Cloud-Native IaC Tools

- Automate the provisioning and management of infrastructure through code, making it easier to maintain consistency, scalability, and reliability in cloud environments.

Cloud-Native IaC tools

- Terraform
- Ansible
- Puppet
- Chef
- Pulumi

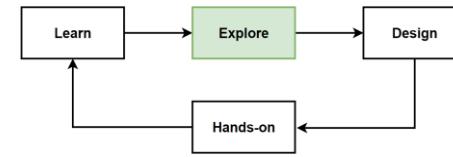
Cloud Serverless IaC tools

- AWS CloudFormation
- AWS SAM (Serverless Application Model)
- AWS Cloud Development Kit (CDK)
- Azure Resource Manager (ARM) Templates
- Azure Bicep
- Google Cloud Deployment Manager

Goto -> <https://landscape.cncf.io/>

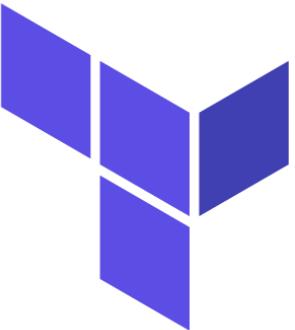


AWS CloudFormation

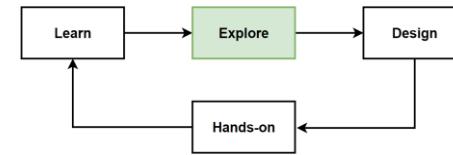


What is Terraform ?

- **Terraform** is an **open-source Infrastructure as Code (IaC)** tool developed by HashiCorp.
- Define and provision data center infrastructure using a **declarative configuration** language known as **HashiCorp Configuration Language (HCL)**, or optionally JSON.
- Terraform can **manage infrastructure** across **various cloud providers**, as well as on-premises resources.
- **Terraform** enables you to safely and predictably provision and manage infrastructure in any cloud.
- Terraform is used **primarily by DevOps teams** to automate various infrastructure tasks like provisioning of cloud resources.



HashiCorp
Terraform



How Terraform uses in Cloud-native ?

- **Provisioning Kubernetes Clusters**

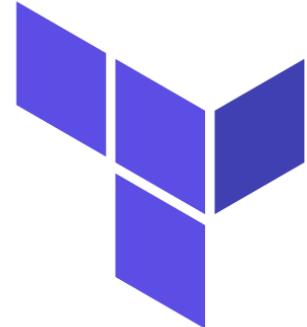
Write scripts to automate the creation of Kubernetes clusters across various cloud providers such as AWS (EKS), Azure (AKS), Google Cloud (GKE), or even on-premises.

- **Managing Kubernetes Resources**

Terraform has a Kubernetes provider which allows you to manage Kubernetes resources such as namespaces, deployments, and services.

- **Microservices Deployment**

Defining Kubernetes deployments and services in Terraform scripts, ensure microservices have the necessary configurations, scaling policies, and networking.



HashiCorp

Terraform

- **Scaling and Auto-scaling**

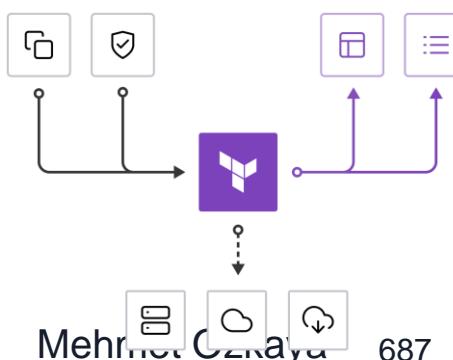
Define auto-scaling policies for your microservices.

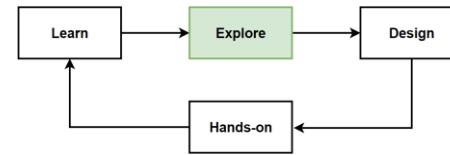
- **Integration with Continuous Delivery Pipelines**

Terraform scripts can be integrated into CI/CD pipelines, allows for automated deployment and management of infrastructure and applications.

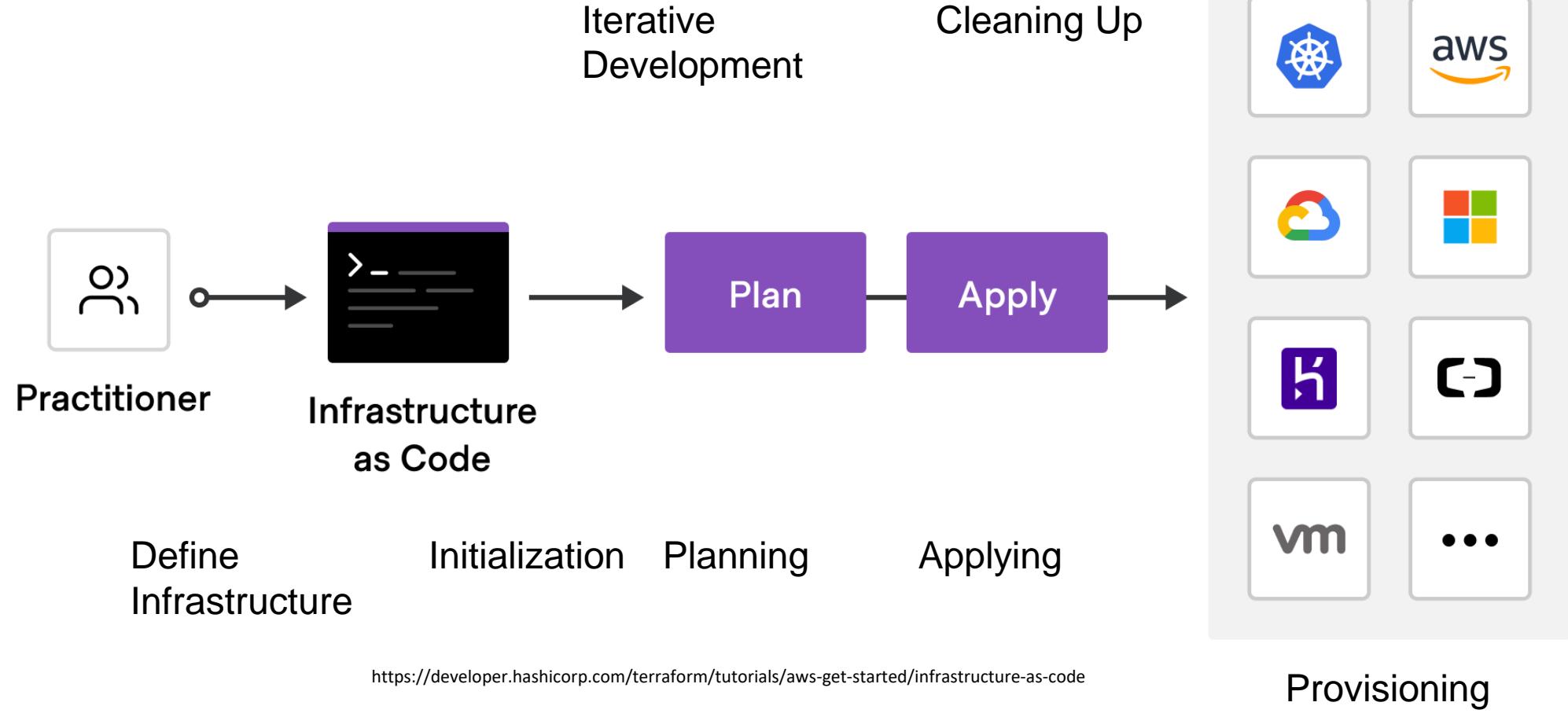
- **State Management**

Terraform maintains a state file which tracks the current state of the infrastructure. Crucial for managing configurations over time and is beneficial in a microservices env.

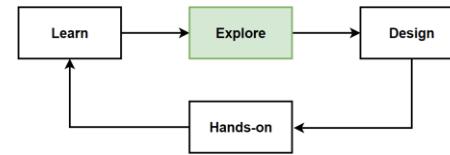




Terraform IaC Steps - How Terraform Works ?



<https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code>



Explore: Cloud-Native IaC Tools

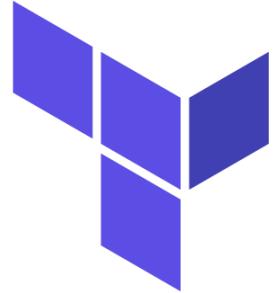
- Automate the provisioning and management of infrastructure through code, making it easier to maintain consistency, scalability, and reliability in cloud environments.

Cloud-Native IaC tools

- Terraform
- Ansible
- Puppet
- Chef
- Pulumi

Cloud Serverless IaC tools

- [AWS CloudFormation](#)
- [AWS SAM \(Serverless Application Model\)](#)
- [AWS Cloud Development Kit \(CDK\)](#)
- [Azure Resource Manager \(ARM\) Templates](#)
- [Azure Bicep](#)
- [Google Cloud Deployment Manager](#)

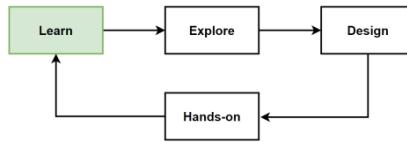


HashiCorp
Terraform



AWS CloudFormation

GitOps

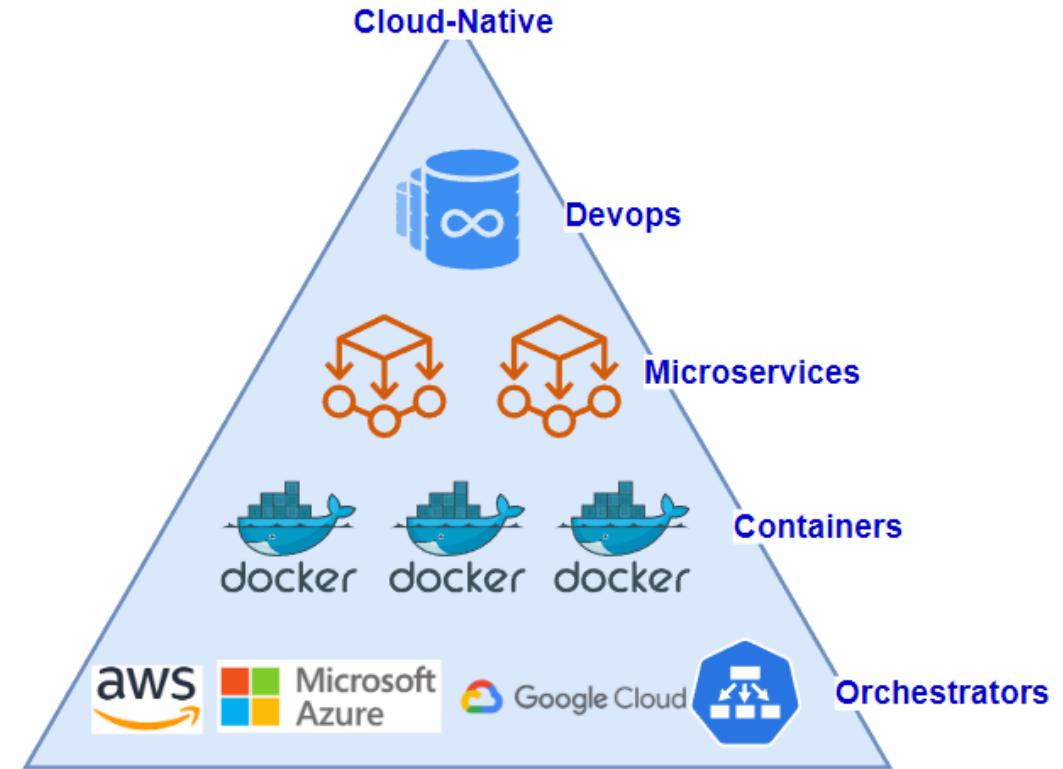


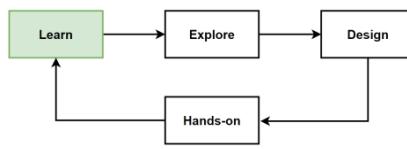
Devops in Cloud-native Applications

- Devops is huge topic in Cloud-native Applications.

Devops Topics divided by 4:

- Devops/DevSecOps
- CI/CD
- IaC
- **GitOps**

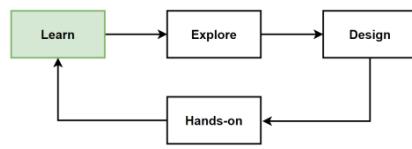




What is GitOps ?

- **GitOps** is a modern approach to continuous delivery and infrastructure management.
- Uses **Git** as the **single source of truth** for both application code and infrastructure configuration.
- **GitOps** aims to make the **process of deploying and managing applications** more streamlined, consistent, and auditable by using Git repositories.
- **GitOps** takes **DevOps best practices** used for application development such as version control, collaboration, compliance, and CI/CD, and applies them to infrastructure automation.
- **GitOps** is evolution of **Infrastructure as Code (IaC)** and a **DevOps best practice** leverages Git as the single source of truth, and control mechanism for creating, updating, and deleting system architecture.
- It is the practice of using **Git pull requests** to verify and automatically deploy system infrastructure modifications.
- GitOps ensures that a system's cloud infrastructure is **immediately reproducible** based on the state of a Git repository.





What is GitOps ? - 2

Why GitOps ?

- Modern infrastructures require to be elastic to manage cloud resources that are needed for continuous deployments.
- GitOps is used to automate the process of provisioning infrastructure for modern cloud infra.
- Similar to use application source code, operations teams adopt GitOps to use infrastructure as code and configuration files into Git repositories.

How to get started with GitOps ?

- Requires infrastructure that can be declaratively managed.
- GitOps is often used with Kubernetes and cloud-native application development and enables continuous deployment for Kubernetes.

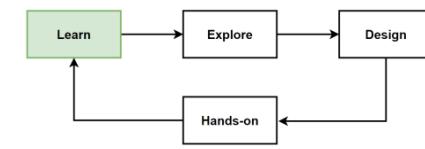


Differences with GitOps and DevOps

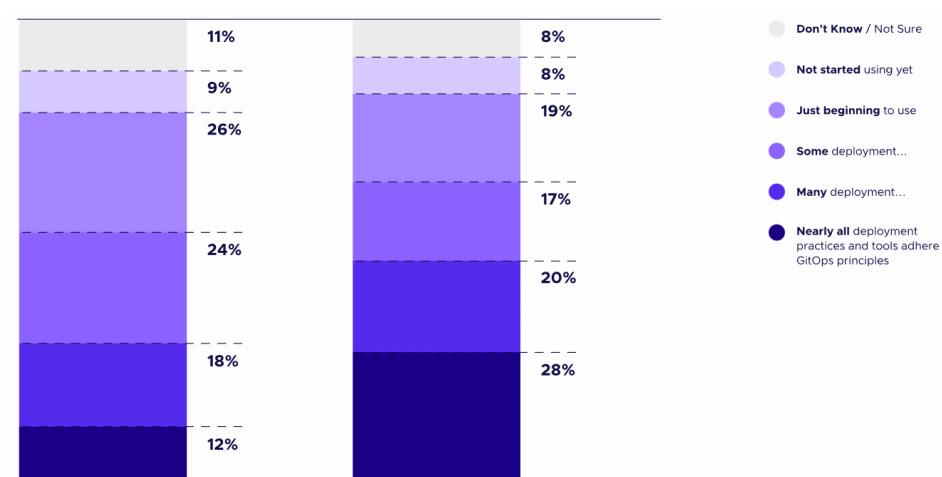
- DevOps is cultural change and providing a way for development teams and operations teams to work together collaboratively.
- GitOps takes DevOps practices, like collaboration, CI/CD, and version control, and applies them to infrastructure automation and application deployment.

CNCF 2022 Annual Survey: Cloud-native Organizations Favor GitOps

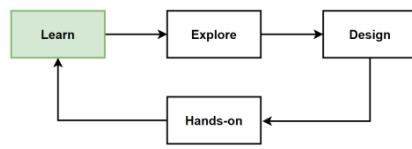
- [**CNCF 2022 Annual Survey**](#)
The year cloud native became the new normal
- **Key Finding:** Cloud-native Organizations Favor GitOps
- Organizations that have fully embraced cloud native techniques are more likely to be releasing applications and using **GitOps**.
- **GitOps** is maturing as a technology base with the **Argo** and **Flux** projects recently graduating in **CNCF**.
- **GitOps principles** are 4x as likely to be followed at mature cloud native organizations, versus those that have not embraced cloud native techniques.



CNCF 2022 ANNUAL SURVEY



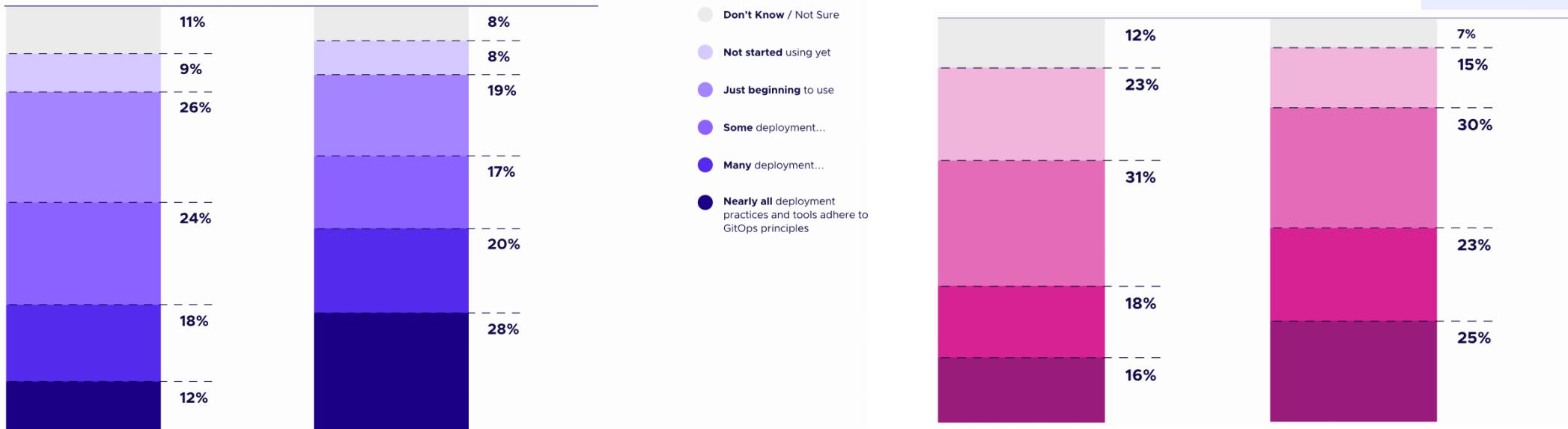
<https://www.cncf.io/reports/cnfc-annual-survey-2022/#findings>



CNCF 2022 Annual Survey: Cloud-native Organizations Favor GitOps - 2

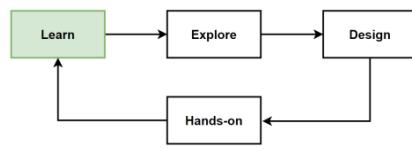
- **Q1:** To what extent has your organization adopted practices and tools that adhere to GitOps principles ?
- **Q2:** How often are your organization's release cycles?

CNCF 2022 ANNUAL SURVEY



- **GitOps becomes crucial practice for organizations which embrace Cloud-native principles that is combination of Devops, CI/CD, IaC and Git principles.**

<https://www.cncf.io/reports/cncf-annual-survey-2022/#findings>



Key Principles of GitOps

- **Declarative Configuration**

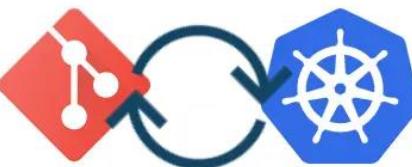
The entire system; applications, environments, and infrastructure is described declaratively. Configuration files are used to describe the desired state of resources.

- **Version Control as Single Source of Truth**

All configurations and code are stored in a Git repository. Acts as the single source of truth for the desired state of the system. Any change must be made by modifying the repository.

- **Pull-Based Deployments**

Changes to the system are automatically deployed from the Git repository. The system pulls in changes from the Git repository when there are updates.



- **Automated Synchronization**

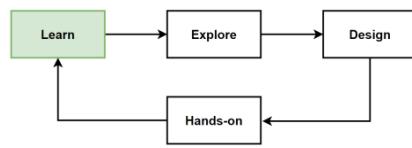
GitOps pipeline continuously compares the actual state of system with the desired state defined in the Git repository. The pipeline automatically syncs the system to match the desired state.

- **Immutable and Auditable Changes**

Changes made through Git, makes an immutable history of who made what change and when. Tracking changes, auditing, and even rolling back to a previous state.

- **Reproducible Infrastructure**

Easily recreate the same infrastructure across various stages of development and even in disaster recovery scenarios, ensuring reliability and stability.

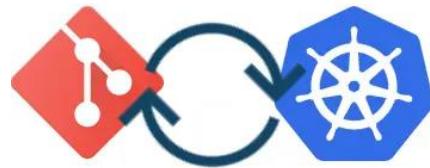


The Steps of GitOps Workflow

- **GitOps** works by using **Git** as the **single source of truth** for declarative infrastructure. Its an operational framework that takes DevOps, CI/CD and IaC best practices. **4 main steps:**

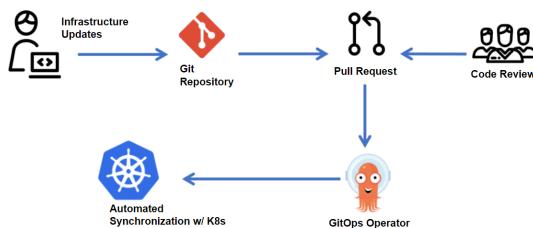
1. Define the Desired State in Git

Developers and operation teams define the desired state of their infrastructure and applications using declarative configuration files in Git repositories as a single source of truth.



2. Submit Change through Pull Request

When needs to make a change to the infrastructure, they submit a pull request to the Git repository. Allows for collaboration, as team members can review the changes, provide feedback, and eventually approve or decline the request.

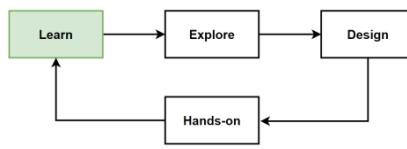


3. Automated Synchronization

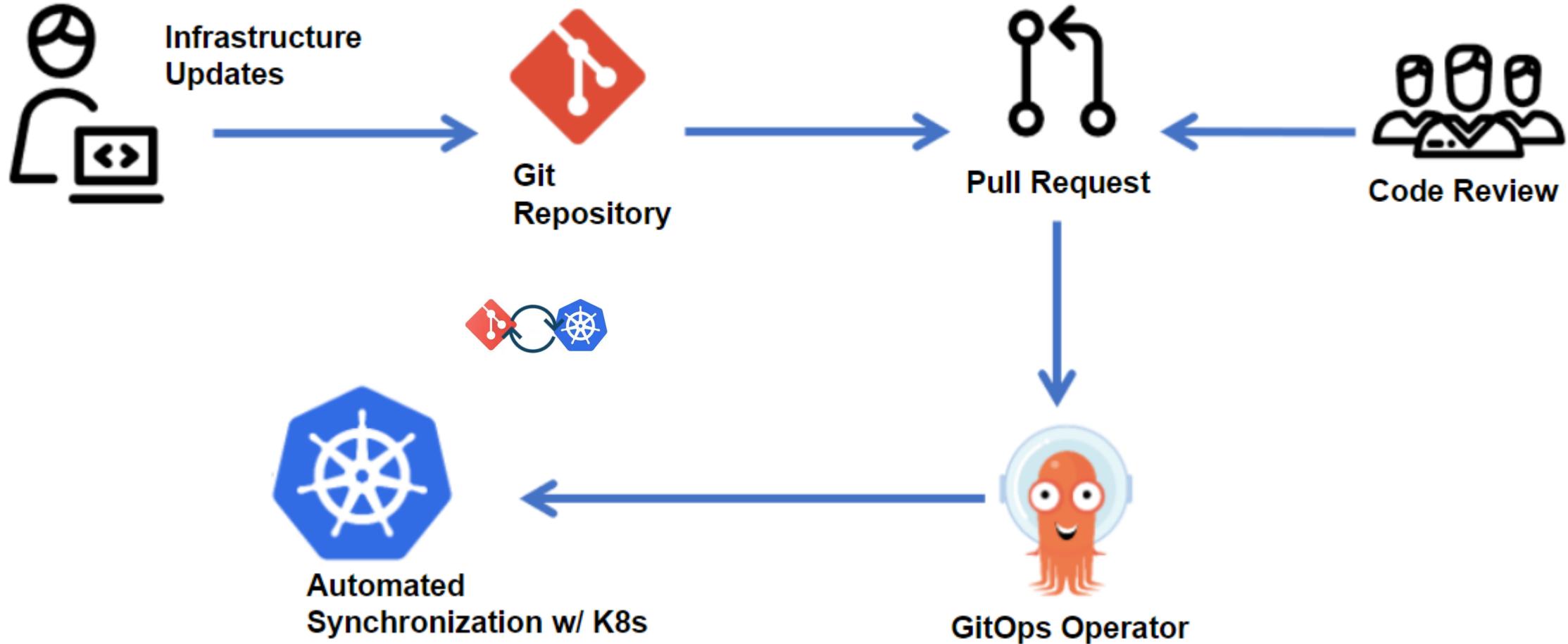
The operator is ensures that the current state of the system matches the desired state in Git repository. It continuously compares the actual state of the env to the desired state defined in Git, and if any differences, it automatically synchronize them by apply changes.

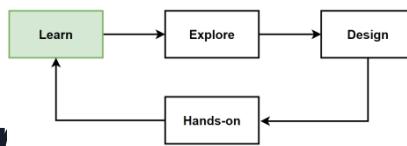
4. Continuous Feedback and Monitoring

Logging, monitoring, and alerting tools that can notify the teams if there is an issue during the synchronization process or if the actual state diverges from the desired state.

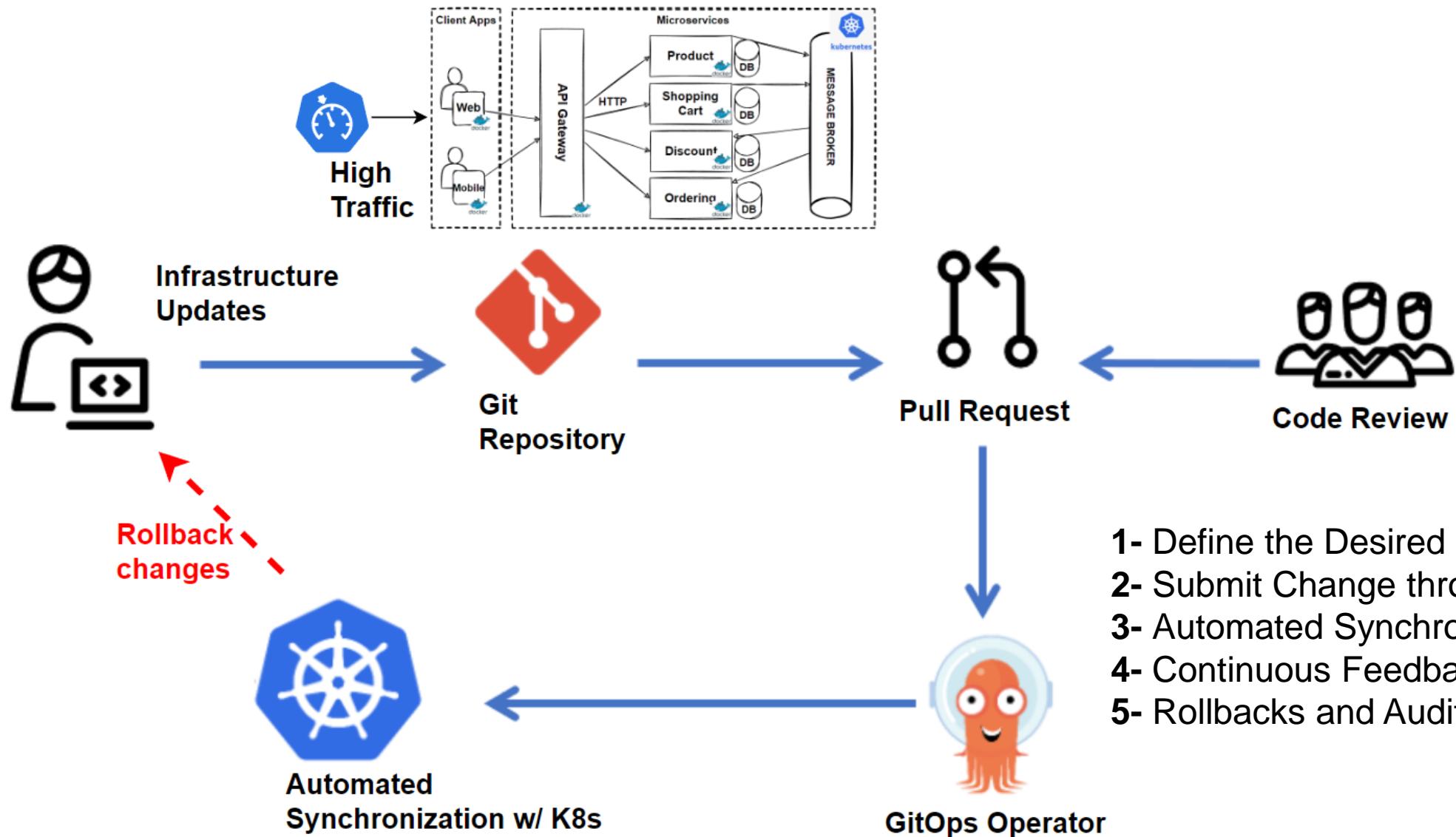


GitOps Workflow Architecture

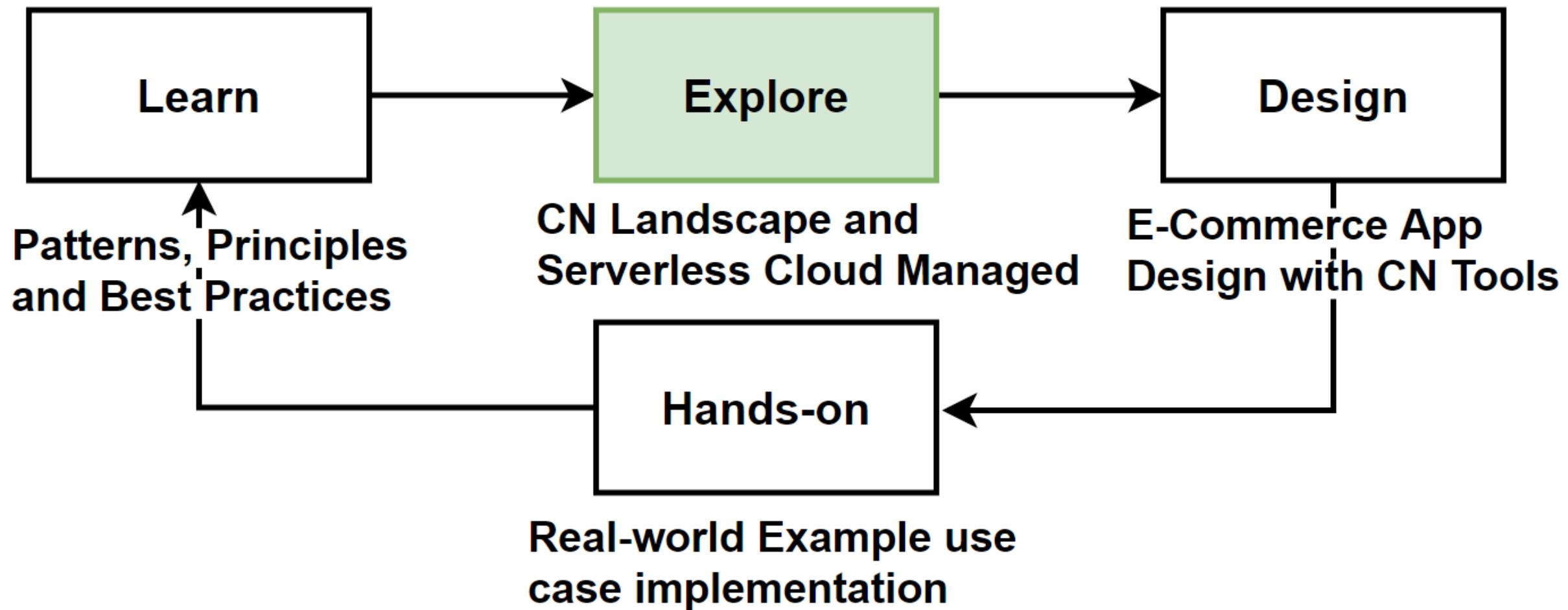




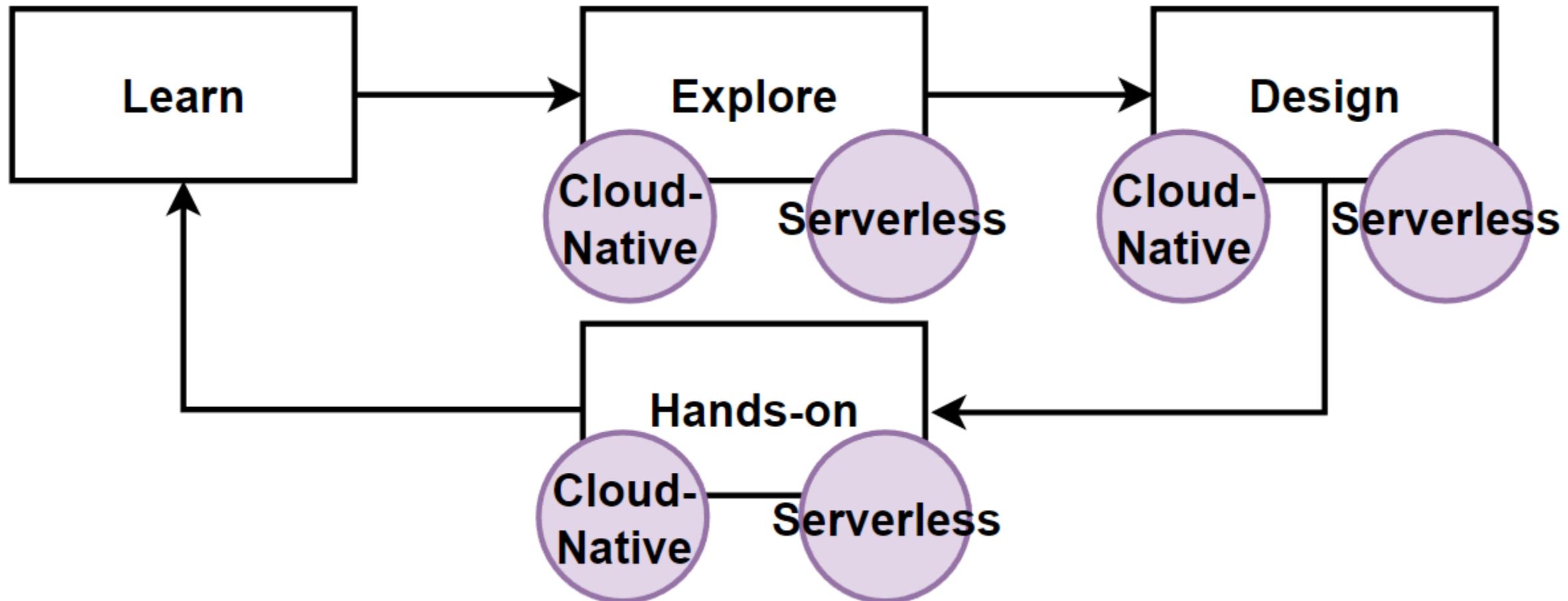
GitOps Real-world Use Case - Black Friday Sale E-commerce



Explore: Cloud Managed and Serverless Microservices Frameworks

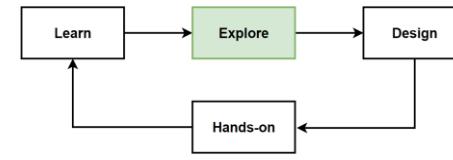


Way of Learning – Cloud-Native & Serverless Cloud Managed Tool



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Explore: Cloud-Native GitOps Tools

- **Cloud-native GitOps tools** are designed to facilitate the GitOps workflow, particularly in Kubernetes ensuring that the state of your cluster matches the configuration defined in a Git repository.

Cloud-Native GitOps tools

- ArgoCD
- Flux
- Jenkins X
- Codefresh
- Tekton



Cloud Serverless GitOps tools

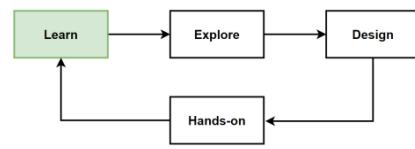
- AWS Proton
- Azure Arc
- Google Config Sync



Goto -> <https://landscape.cncf.io/>

CNCF 2022 Annual Survey: Cloud-native Organizations Favor GitOps

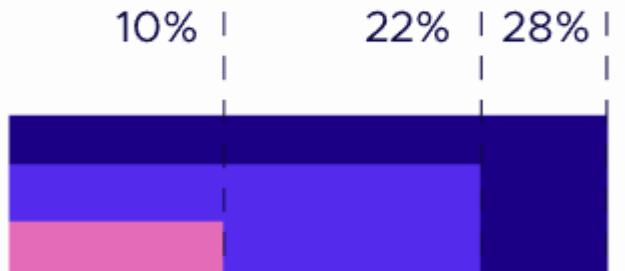
- [CNCF 2022 Annual Survey](#)
The year cloud native became the new normal
- Key Finding: Cloud-native Organizations Favor GitOps
- Organizations that have fully embraced cloud native techniques are more likely to be releasing applications and using GitOps.
- **GitOps** is maturing as a technology base with the **Argo** and **Flux** projects recently **graduating** in CNCF.
- GitOps principles are 4x as likely to be followed at mature cloud native organizations, versus those that have not embraced cloud native techniques.
- The **adoption of CNCF incubated and graduated projects** once again **increased** in 2022, with OpenTelemetry and **Argo** scoring the largest jumps in usage. The former rose from 4% in 2020 to 20% in 2022 and the later from 10% to 28%.



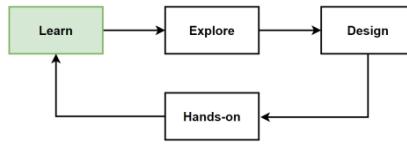
CNCF 2022 ANNUAL SURVEY

● 2020 ● 2021 ● 2022

ARGO
(Graduated)

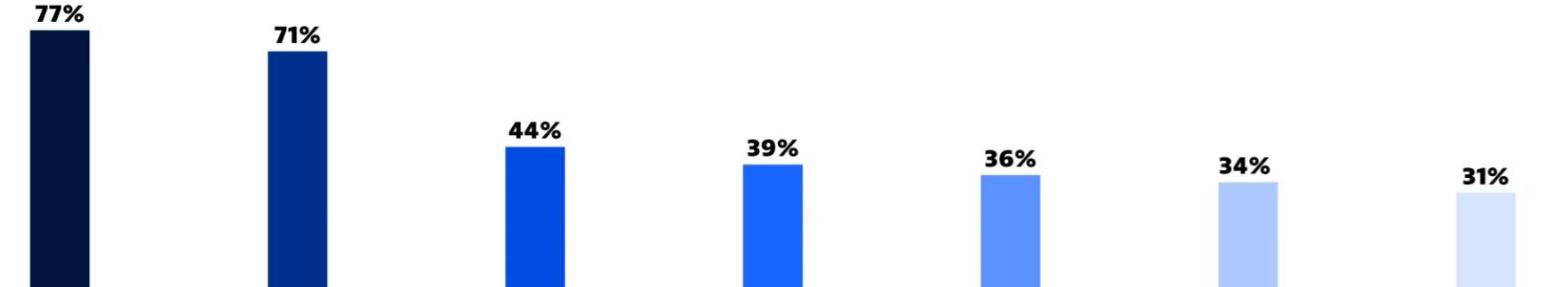


<https://www.cncf.io/reports/cncf-annual-survey-2022/#findings>

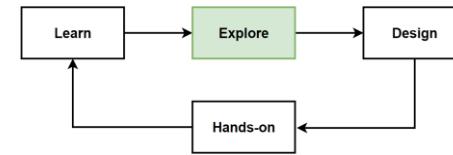


Open source software drives a vibrant Kubernetes

- [Dynatrace Kubernetes Report: Strongest Kubernetes growth areas](#)
- Open-source projects in the Kubernetes across various categories, including observability, databases, messaging, CI/CD, big data, security, and service meshes.
- Open-source solutions: Prometheus, Redis, RabbitMQ, Kafka, **ArgoCD**, **Flux**, GitLab, Jenkins, Elasticsearch, Gatekeeper, and Istio, are widely used as backing services or tools to enhance the capabilities of cloud-native applications.



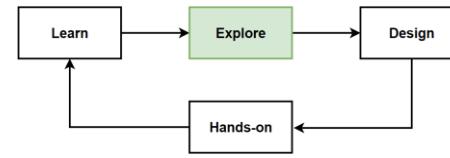
Open source observability		Database		Messaging		Continuous delivery		Big data		Security		Service mesh	
Prometheus	65%	Redis	60%	Kafka	32%	ArgoCD	17%	Elasticsearch	32%	Gatekeeper	16%	Istio	24%
Kubestatemetrics	47%	PostgreSQL	29%	RabbitMQ	23%	Flux	15%	Solr	8%	Twistlock	11%	Kong Mesh	7%
Fluentd	31%	MongoDB	28%	ActiveMQ	6%	GitLab	15%	Spark	2%	Tigera	9%	Linkerd	3%
FluentBit	27%	MySQL	18%			Jenkins	8%			Aqua	4%	Anthos Service Mesh	1%
Thanos	26%	Cassandra	10%			Argo Rollouts	4%			Qualys	2%		
Jaeger	18%	Oracle	7%			GoCD	3%			CrowdStrike	2%		
Kiali	18%	MariaDB	6%			Keptn	2%			Sysdig	2%		
		Memcached	6%			Tekton	2%			CIS	2%		
		CouchDB	3%			GitHub Actions	1%			Snyk	1%		
		Liquibase	3%							NeuVector	1%		
		Couchbase	2%										



What is Argo CD ?

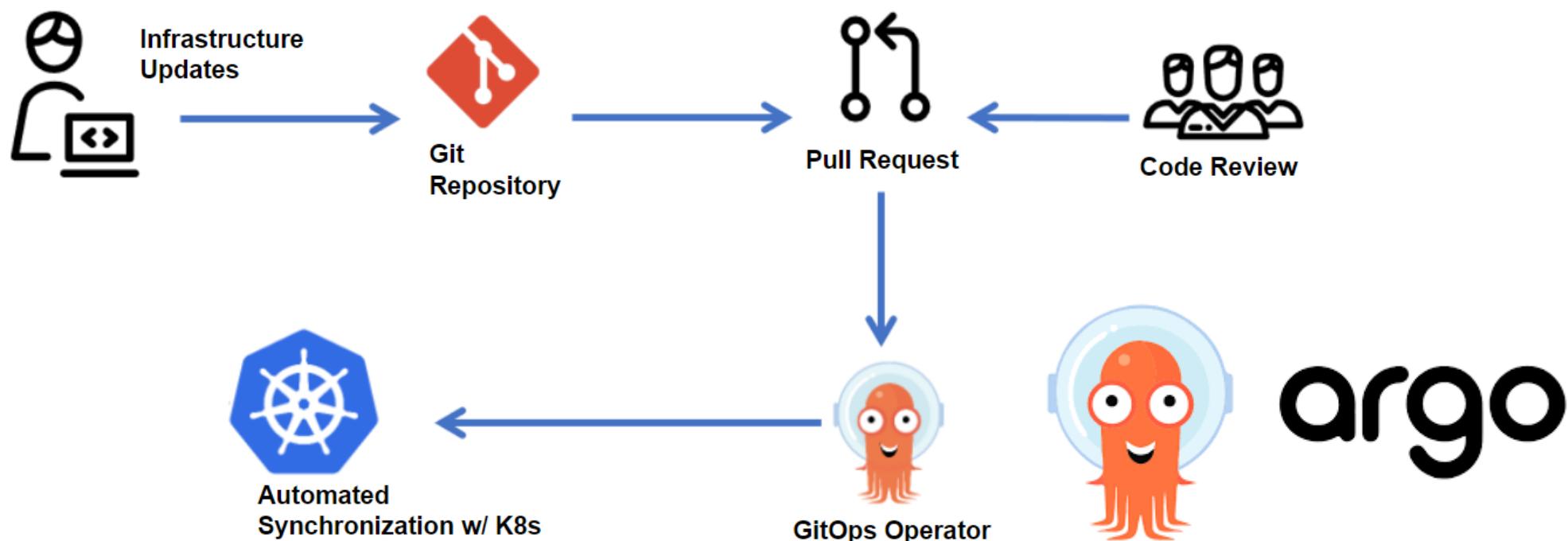
- Argo CD is an open-source **continuous delivery (CD) tool** that facilitates the management and deployment of apps within Kubernetes using GitOps principles.
- Argo CD **automates the synchronization** of the desired application states defined in a Git repository with the live state in a Kubernetes cluster.
- Argo CD uses **Git repositories** as the **source of truth** for the desired state of apps and environments. Everything is defined declaratively and version-controlled.
- Argo CD **continuously monitors** the Git repository and automatically syncs the changes to the Kubernetes cluster. Users can choose between automatic sync or manually trigger sync for changes.
- Through its **user-friendly dashboard**, Argo CD provides **visualization** of the applications, their configuration, health status, and synchronization status.
- Argo CD can **easily perform rollbacks** to previous versions and provide a history of all changes that have been made.
- Argo CD can **deploy** applications across **multiple Kubernetes clusters**, allowing for more complex and scalable architectures.

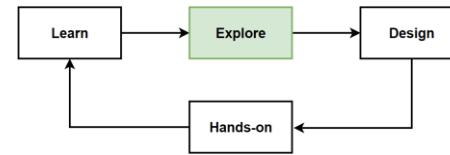




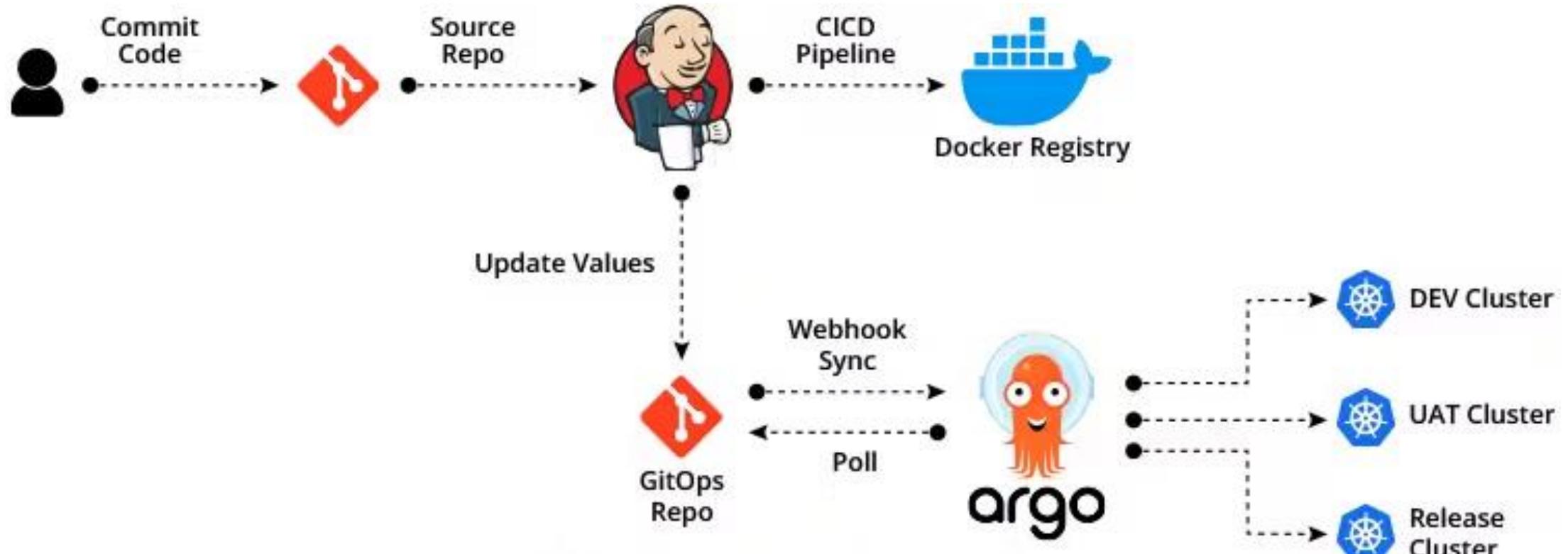
Practical Use of Argo CD

- Teams **define Kubernetes manifests, Helm charts**, or other declarative configurations in a Git repository.
- Argo CD** is configured to **track this repository** and ensure the Kubernetes cluster **matches the desired state** defined in the repository.
- Changes made to the repository**, such as a new version of a container image or a configuration change, will be automatically applied to the cluster. This ensures a consistent and auditable deployment process.





Argo CD deploy microservices in Kubernetes with CI/CD



Step 1: Define Application Code and Infra Code in Git

Step 2: Integrate CICD Pipelines with GitOps

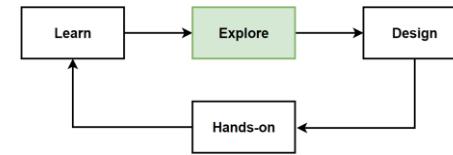
Step 3: Submit Change in Application Code

Step 4: CICD Update Docker Image Tag into Infra Code

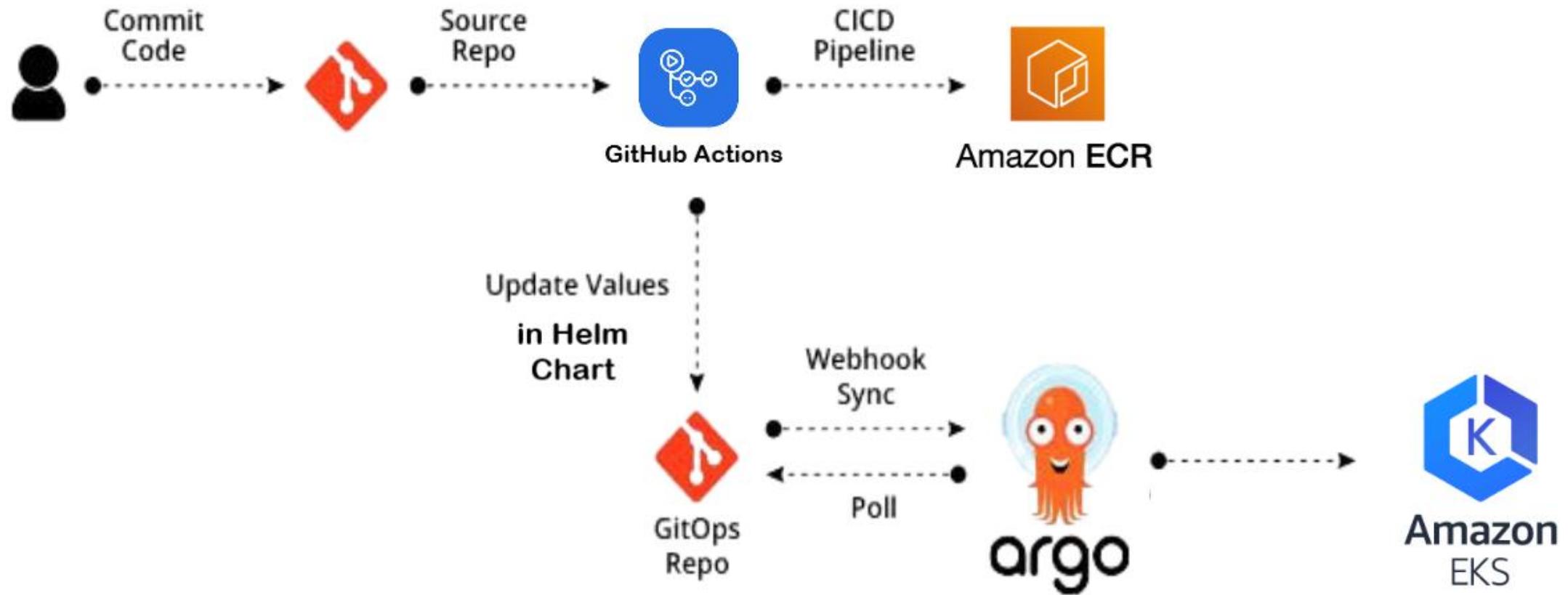
Step 5: GitOps Automated Synchronization

Step 6: Continuous Monitoring and Audit Trails

<https://www.gspann.com/resources/blogs/continuous-delivery-for-kubernetes-with-gitops-and-argo-cd/>



Argo CD 2 deploy EKS with GitHub Actions



Step 1: Define Application Code and Infra Code in Git

Step 2: Integrate CICD Pipelines with GitOps

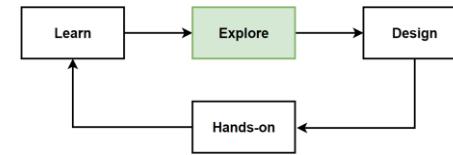
Step 3: Submit Change in Application Code

Step 4: CICD Update Docker Image Tag into Infra Code

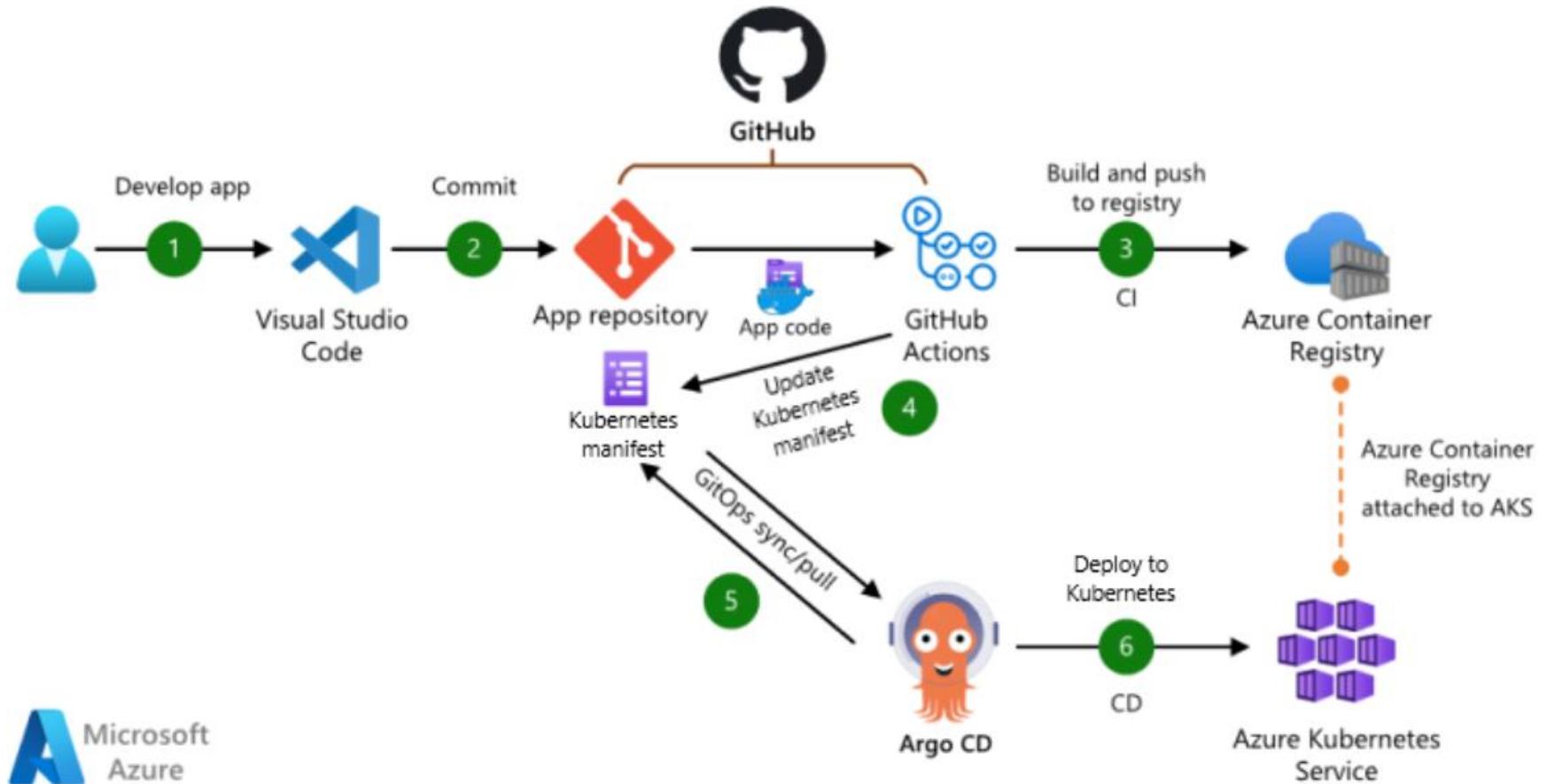
Step 5: GitOps Automated Synchronization

Step 6: Continuous Monitoring and Audit Trails

<https://levelup.gitconnected.com/gitops-ci-cd-using-github-actions-and-argocd-on-kubernetes-909d85d37746>



Argo CD 3 deploy AKS with GitHub Actions



Step 1: Define Application Code and Infra Code in Git

Step 2: Integrate CICD Pipelines with GitOps

Step 3: Submit Change in Application Code

Step 4: CICD Update Docker Image Tag into Infra Code

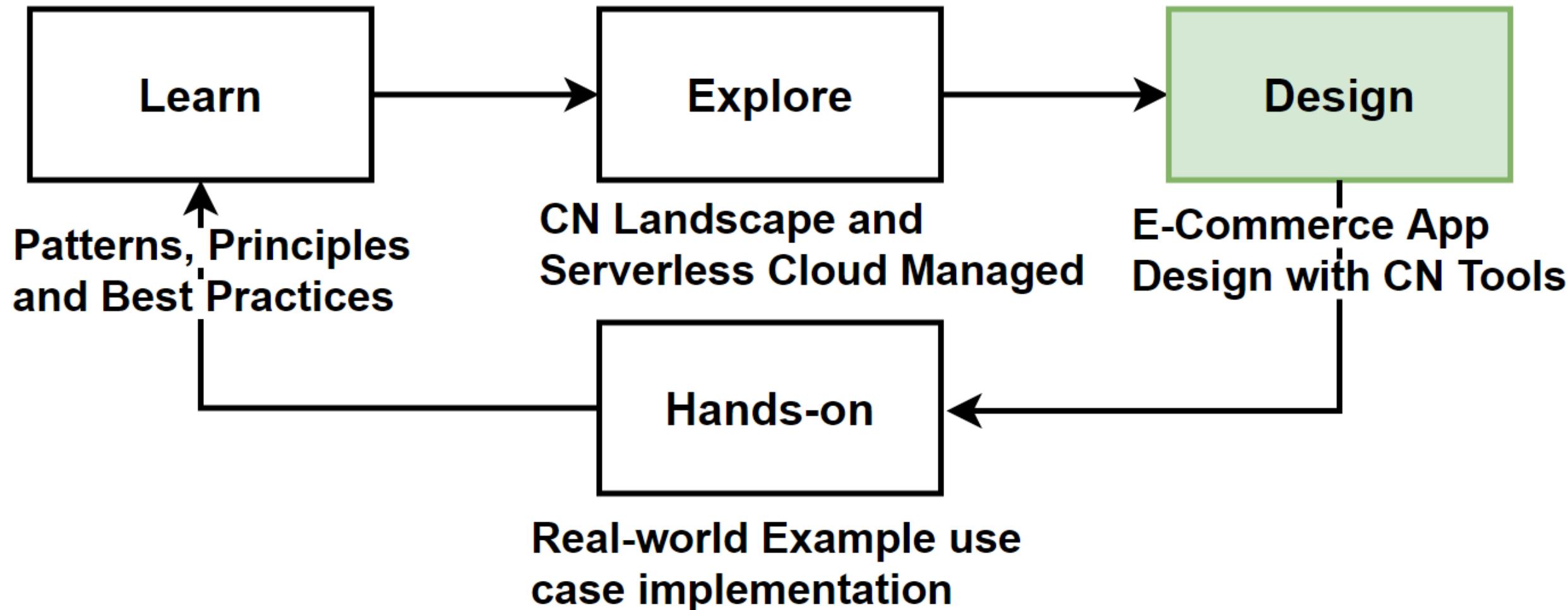
Step 5: GitOps Automated Synchronization

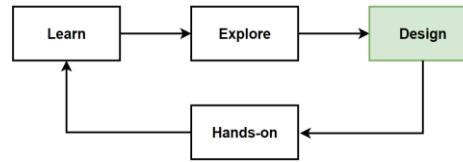
Step 6: Continuous Monitoring and Audit Trails

<https://learn.microsoft.com/en-us/azure/architecture/guide/aks/aks-cicd-github-actions-and-gitops>

Design with Devops, CI/CD, IaC and GitOps

Way of Learning – The Course Flow





Tools: Cloud-Native DevOps, CI/CD, IaC and GitOps

- Cloud-native DevOps tools that we picked for designing e-commerce microservices application:

CI/CD Pipeline

- GitHub Actions

IaC

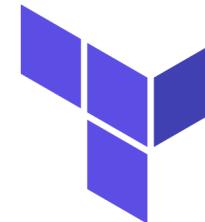
- Terraform

GitOps

- ArgoCD



GitHub Actions

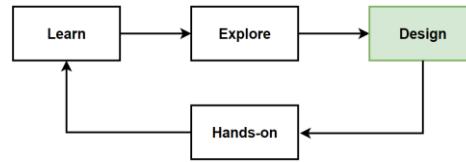


HashiCorp

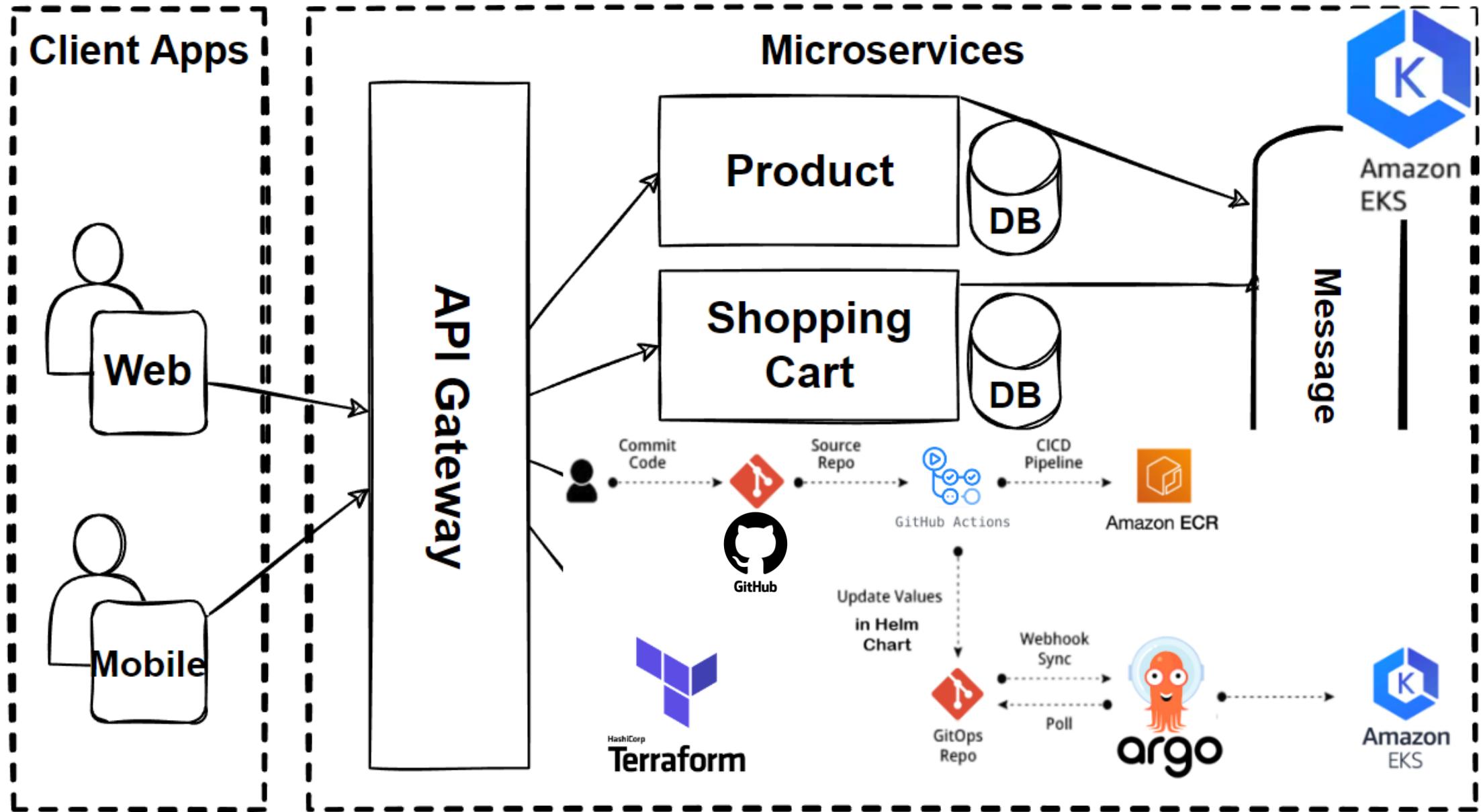
Terraform

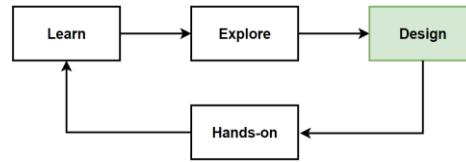


argo



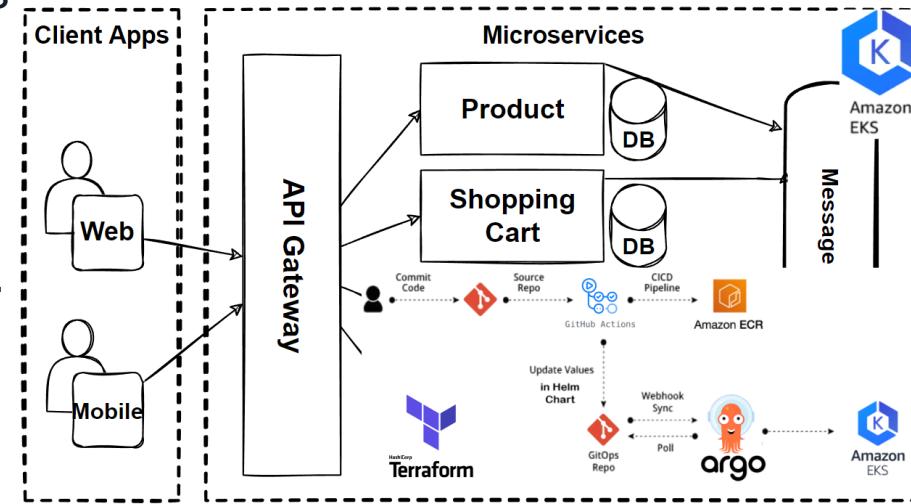
Microservices with DevOps, CI/CD, IaC and GitOps

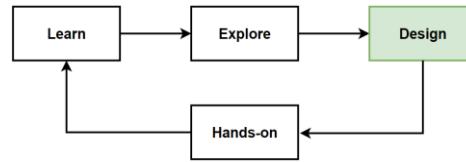




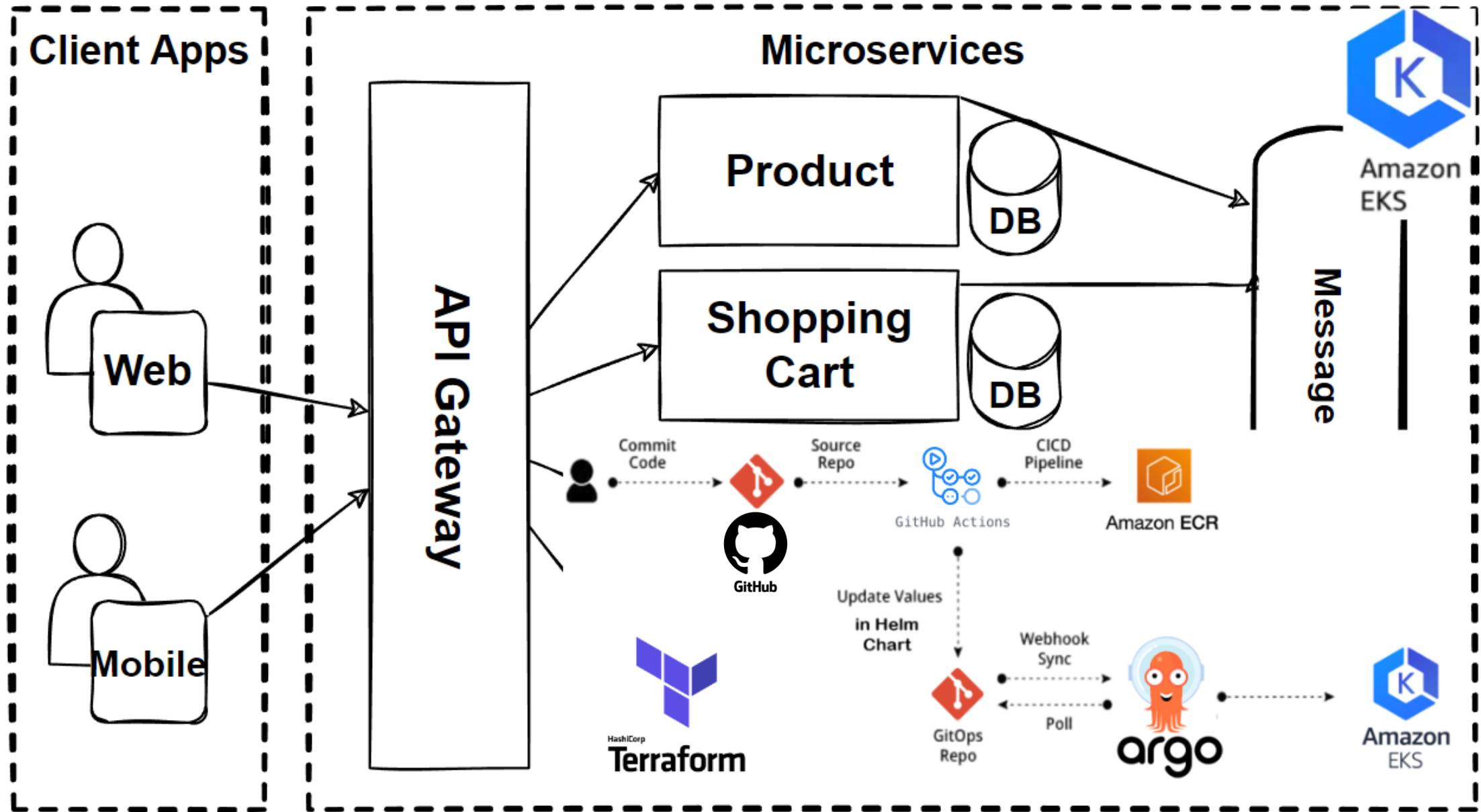
Steps of DevOps, CI/CD, IaC and GitOps Architecture

- 1. Infrastructure Provisioning:** Terraform configuration files to describe your AWS infrastructure including the EKS cluster, ECR repositories.
- 2. Set Up ArgoCD:** Deploy ArgoCD in your EKS cluster. Configure ArgoCD to monitor your GitHub repositories for changes in Kubernetes manifests.
- 3. Develop Microservices:** Writes code for the microservices and pushes to GitHub.
- 4. GitHub Actions Trigger:** It builds a Docker image of the microservice. Pushes the Docker image to ECR. Updates the Kubernetes manifest files with the new image tag.
- 5. ArgoCD Observes and Deploys:** ArgoCD notices the changes in Kubernetes manifests. It pulls these manifest files. Deploys the updated microservice to the EKS cluster by applying the manifests.
- 6. Monitor and Observe:** Set up monitoring and logging solutions like Prometheus and Grafana, AWS CloudWatch for observing the performance and health of your microservices.





Microservices with DevOps, CI/CD, IaC and GitOps



Hands-on: Terraform IaC provision AWS EC2 instance

Terraform Workflow and Development Loop, Terraform Providers

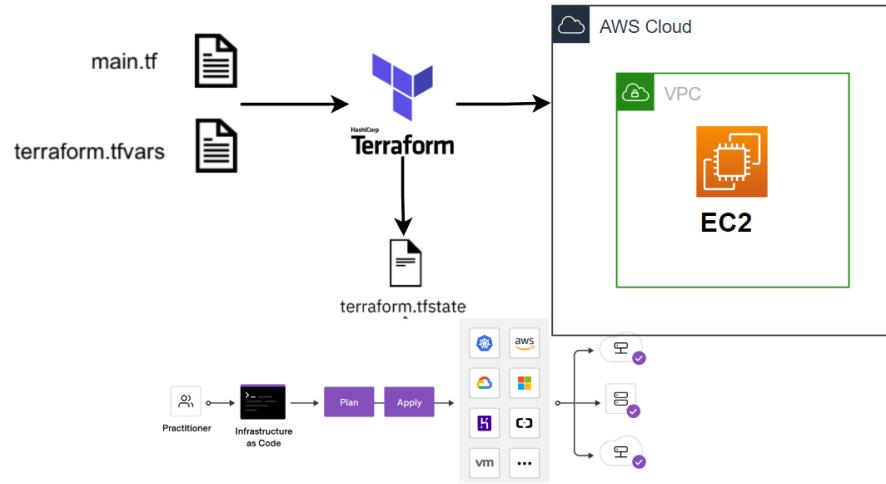
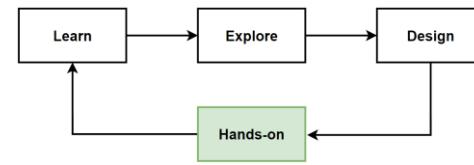
Install and Setup Terraform

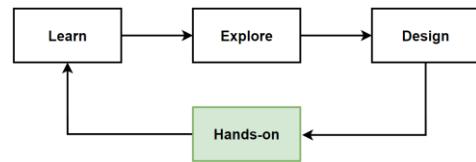
Build, Change and Destroy Infrastructure

Parameterizing the Configuration with Input Variables

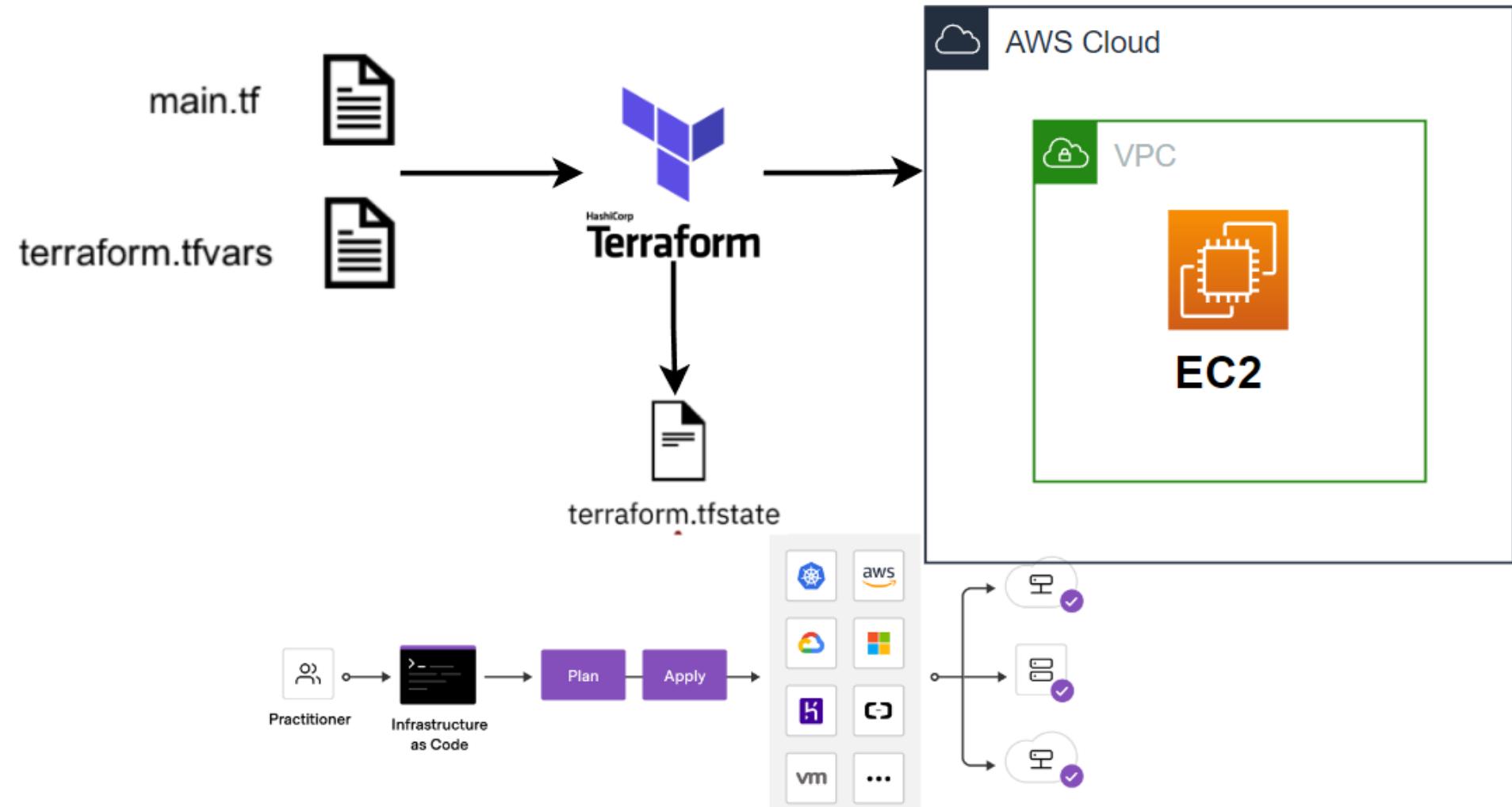
Hands-on: Deploying Microservices on Amazon EKS Fargate— Task List

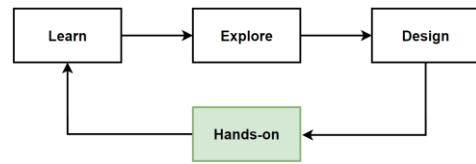
- Step 1. Install and Setup Terraform
- Step 2. Terraform Develop IaC: Write configuration the Desired State
- Step 3. Terraform Init: Initializing the Working Directory
- Step 4. Terraform Plan: Validate and format the configuration
- Step 5. Terraform Apply: Create infrastructure w/ Applying the Plan
- Step 6. Change Infrastructure w/ Config Changes and Apply Changes
- Step 7. Terraform Variables: Parameterizing the Configuration with Input Variables
- **Step 8. Terraform Destroy: Destroy Infrastructure**



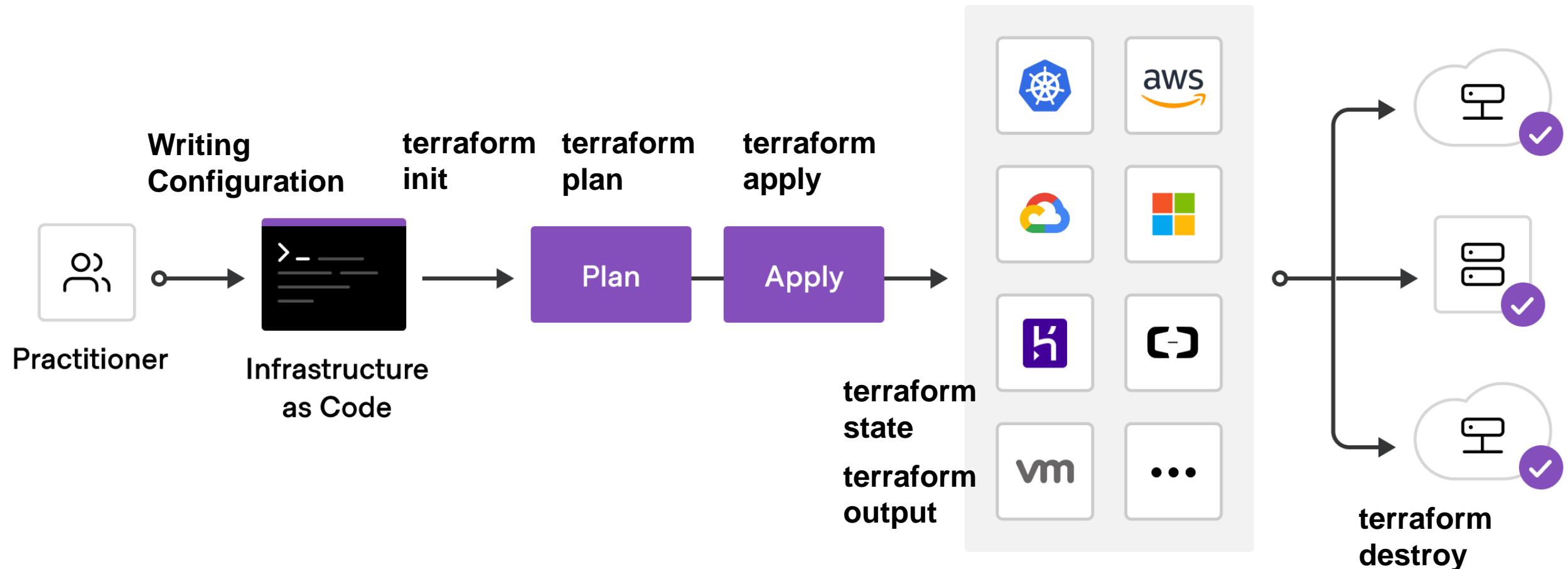


Hands-on: Terraform IaC provision AWS EC2 instance

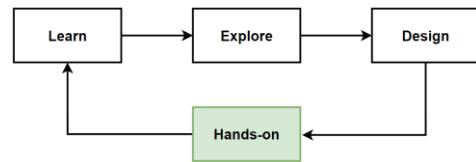




Terraform IaC Steps - How Terraform Works ?



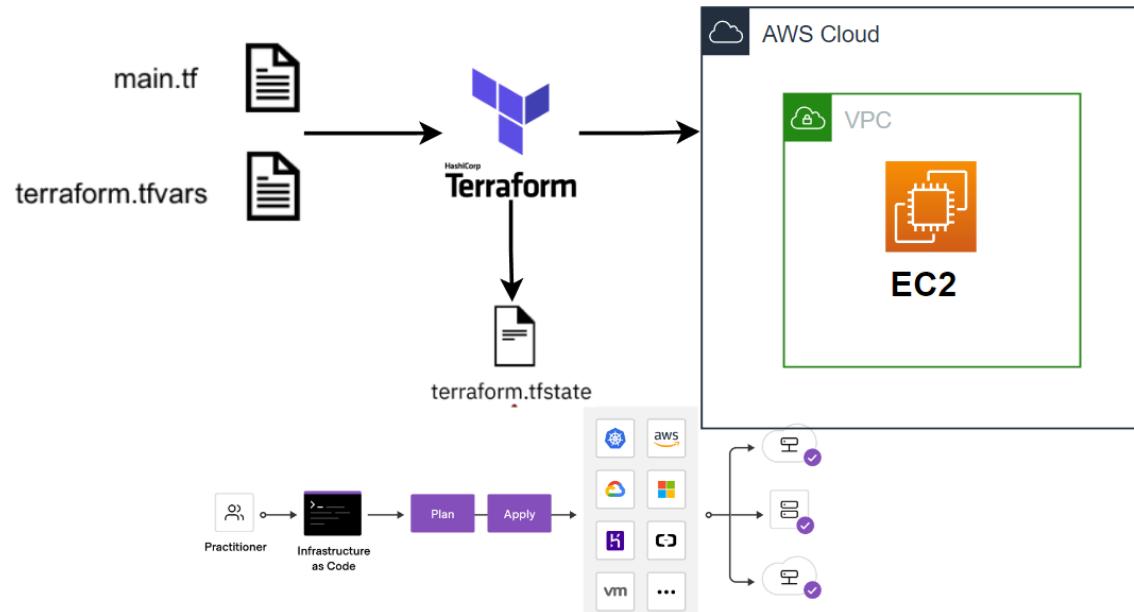
<https://developer.hashicorp.com/terraform/tutorials/aws-get-started/infrastructure-as-code>

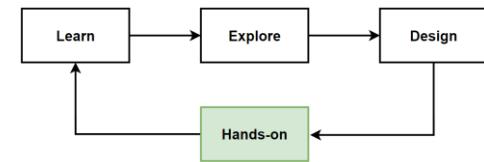


Step 1. Install and Setup Terraform

- [Get AWS Free Trier Account](#)
- [Configure AWS CLI](#)
- [Install Terraform on your local machine](#)

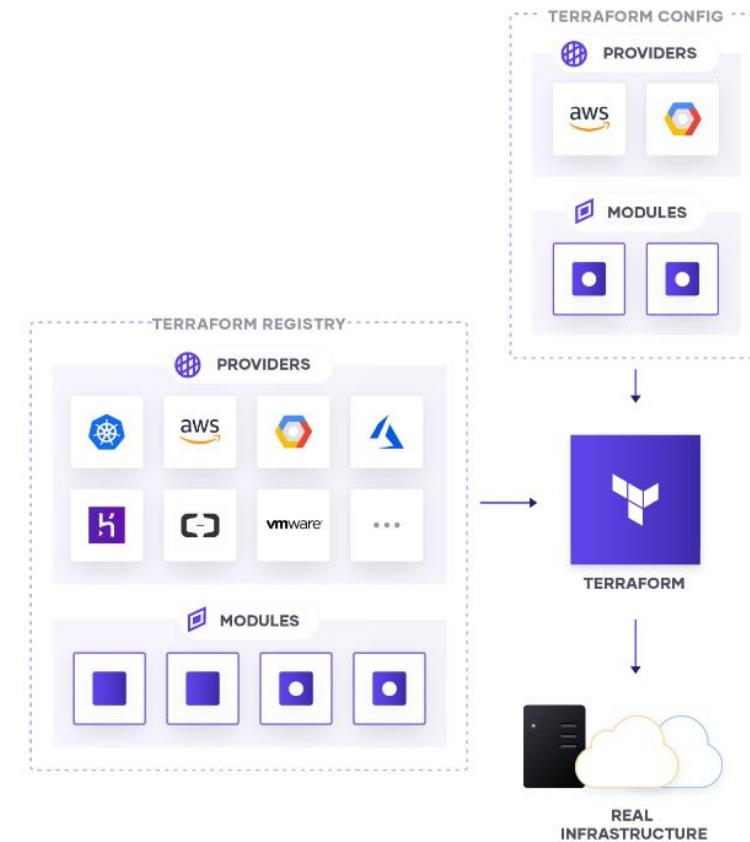
- **Run Commands:**
- `aws --version`
- `terraform -help plan`



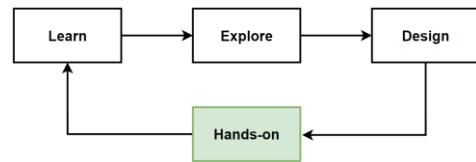


Terraform Providers

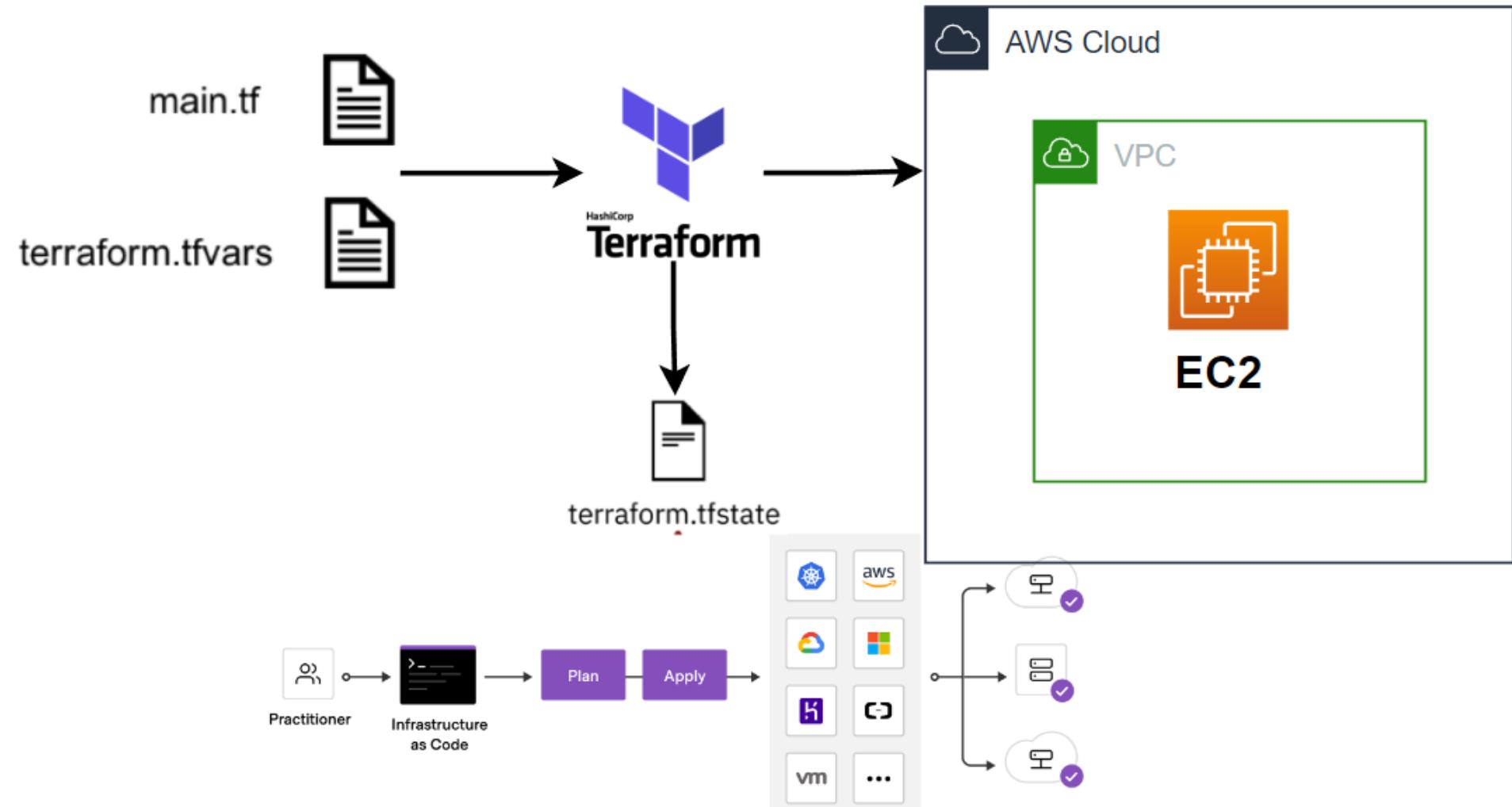
- Terraform implements a **modular approach** in its application architecture.
- The Terraform binary we **downloaded** is the **core module** required to perform core Terraform functions.
- **Terraform Providers** are responsible for **understanding API interactions** and **exposing resources** to Terraform.
- They essentially **act as a bridge** between the **Terraform configuration files** and the **target platform's API**.
- [Terraform Providers from Registry](#)
- <https://registry.terraform.io/>



<https://registry.terraform.io/>



Hands-on: Terraform IaC provision AWS EC2 instance



Hands-on: GitHub Actions CI/CD for Build & Push Docker Images to DockerHub

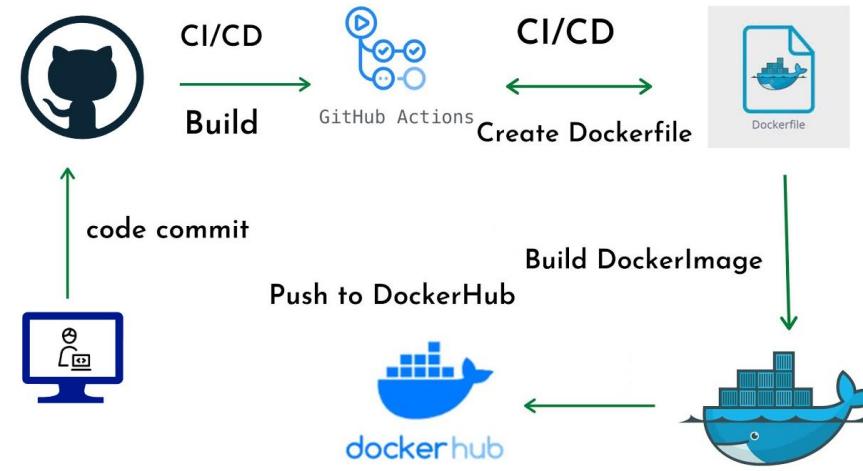
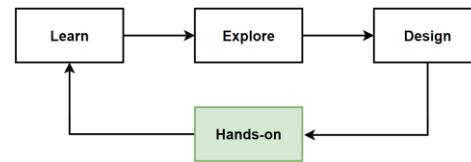
Create Repository on GitHub and push Web API project with Dockerfile

Create GitHub Actions workflow

Develop Workflow for Build & Push Docker Images to DockerHub

Hands-on: GitHub Actions CI/CD for Build & Push Docker Images to DockerHub

- Step 1. Create Repository on GitHub
- Step 2. Clone Repository and push Product microservices codes with Dockerfile
- Step 3. Create GitHub Secrets for DockerHub Username and Token
- Step 4. Create and Define a GitHub Actions Workflow File
- Step 5. Commit Push and Monitor Workflow, check DockerHub
- Step 6. Change code and commit push to monitor new GitHub Actions workflow



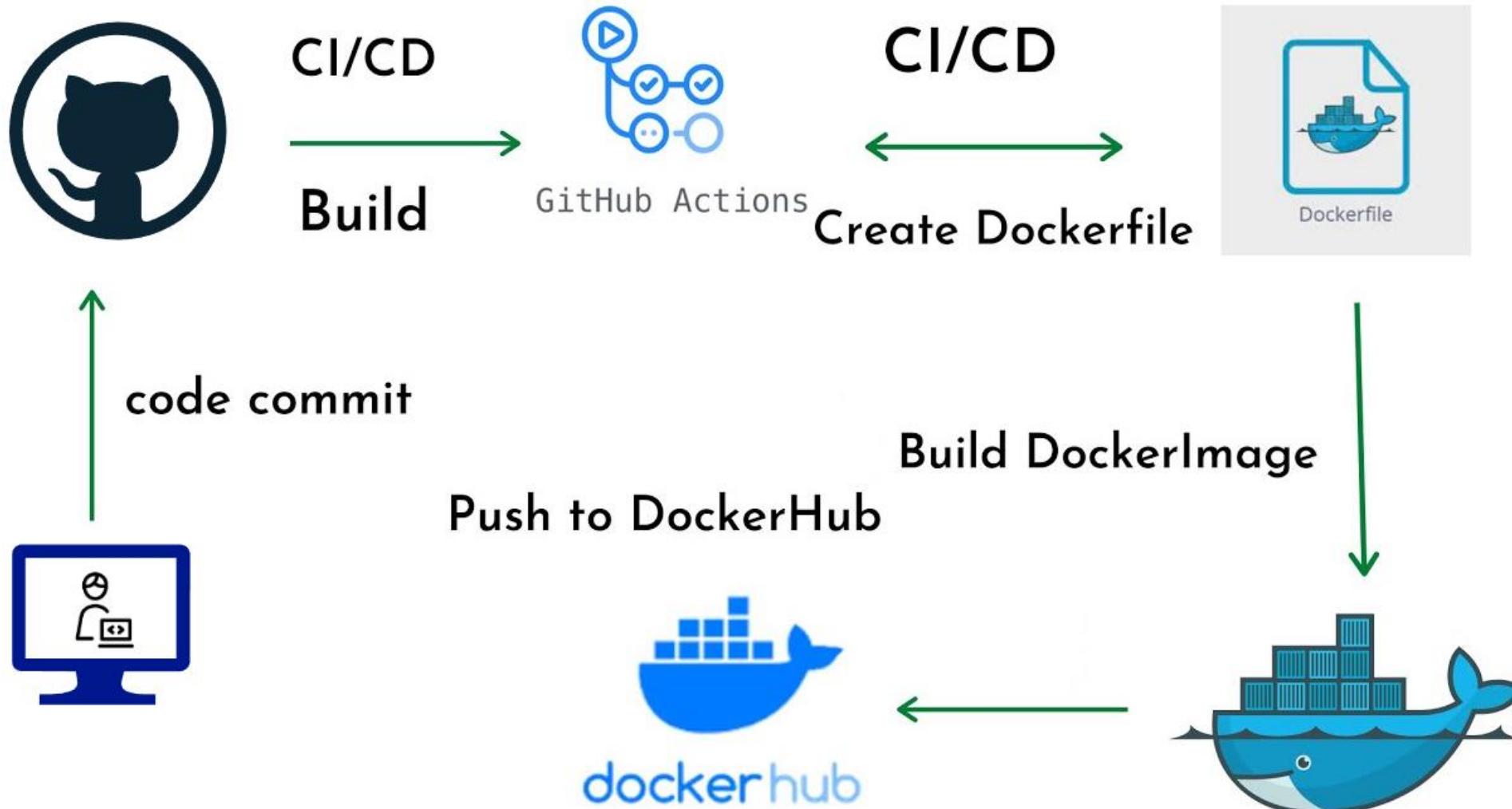
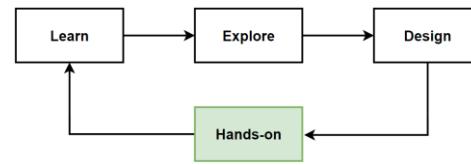
Prerequisites:

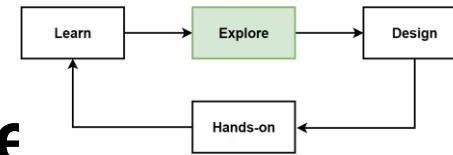
- A GitHub account
- A Docker Hub account
- A repository on GitHub containing Product Microservices with a Dockerfile

GitHub Repo:

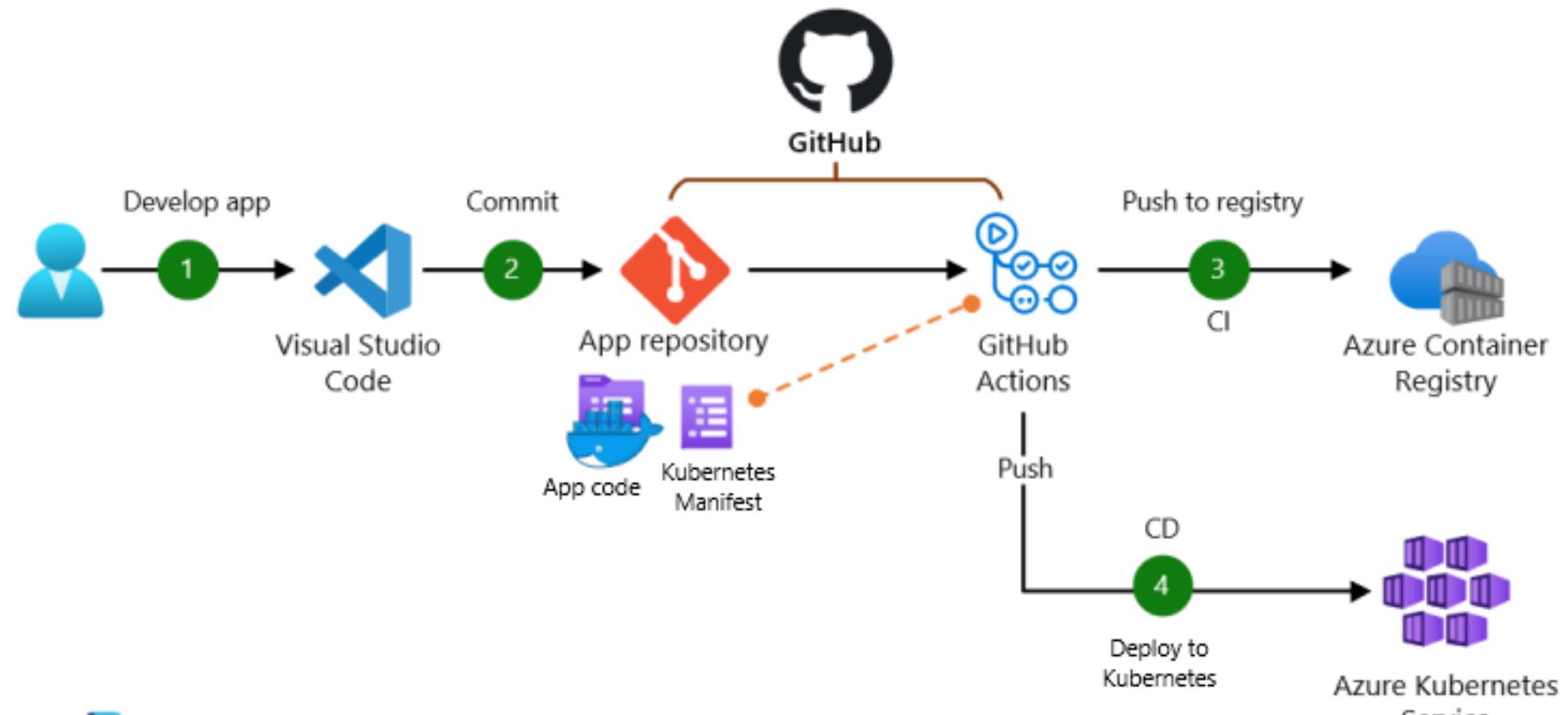
<https://github.com/mehmetozkaya/ProductService>

Hands-on: GitHub Actions CI/CD for Build & Push Docker Images to DockerHub





Deploy Microservices to AKS with GitHub Actions - Azure



<https://learn.microsoft.com/en-us/azure/architecture/guide/aks/aks-cicd-github-actions-and-gitops>

GitHub Marketplace Actions Deploy to Kubernetes cluster:

<https://github.com/marketplace/actions/deploy-to-kubernetes-cluster>

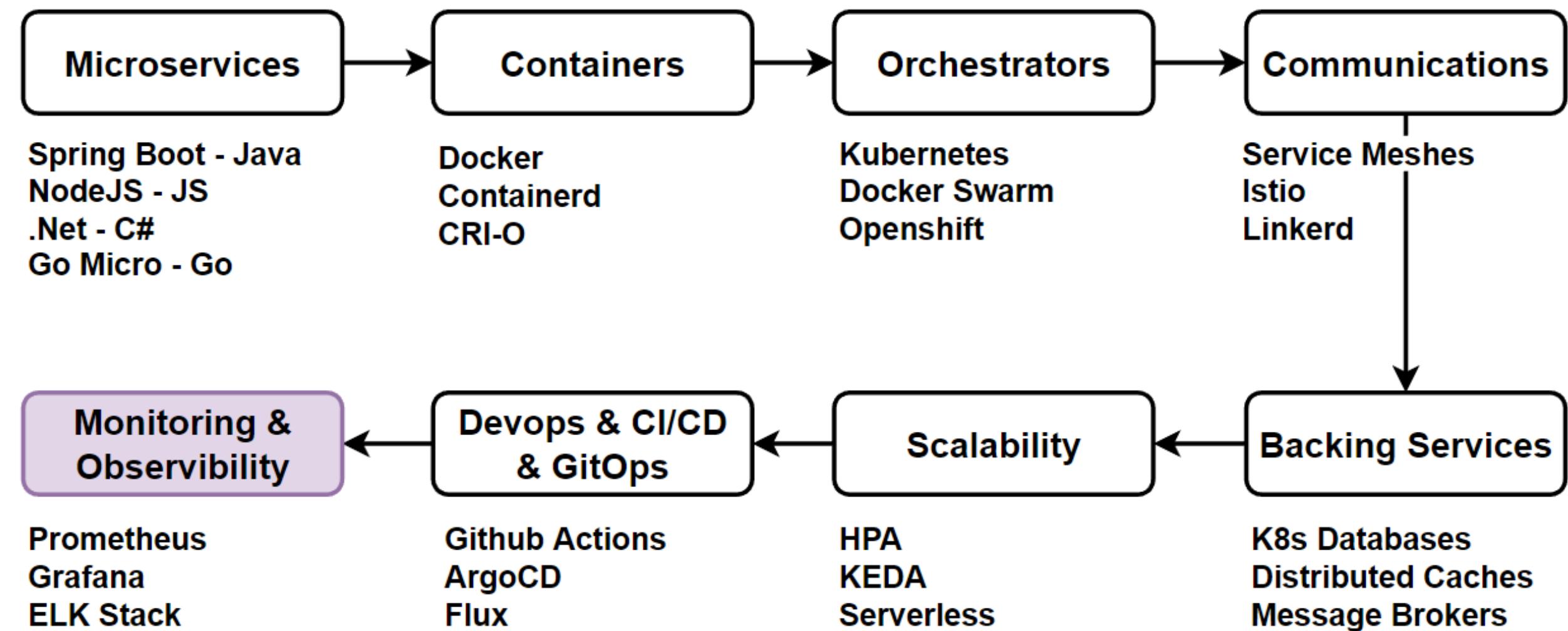
Cloud-Native Pillar8: Monitoring & Observability with Distributed Logging and Tracing

Monitoring and troubleshooting microservices-based applications using observability tools.

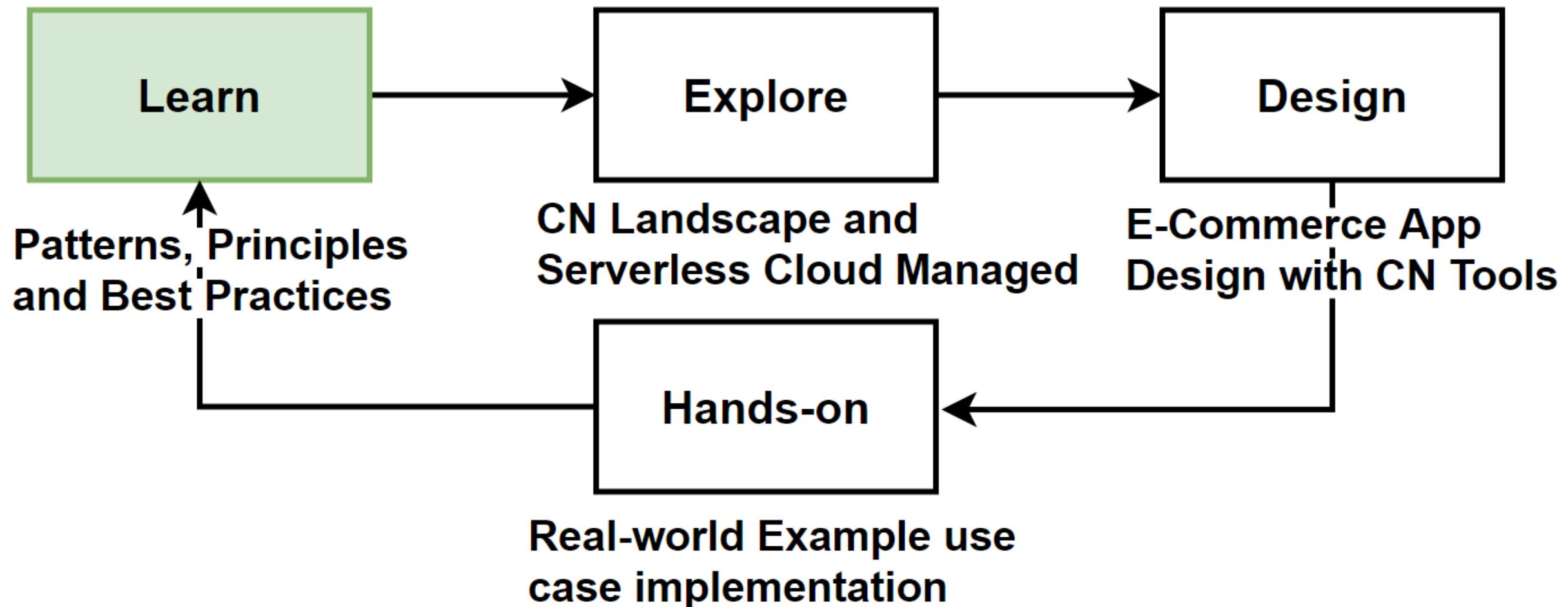
What are Monitoring & Observability in Cloud-native Microservices ?

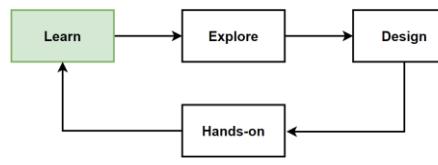
What are patterns & best practices of Monitoring & Observability in CN microservices ?

Cloud-Native Pillars Map – The Course Section Map



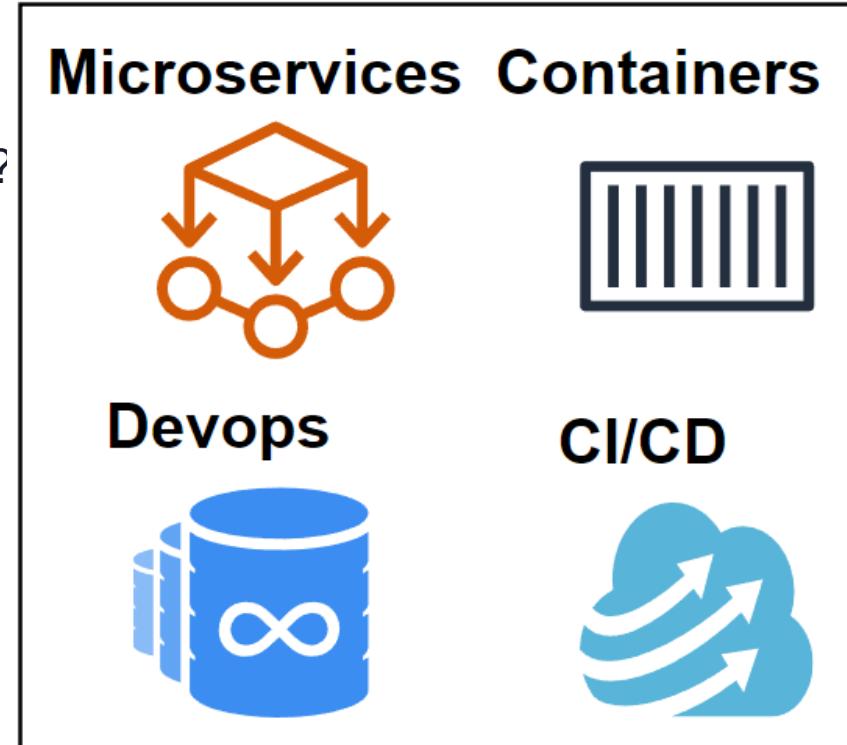
Way of Learning – The Course Flow

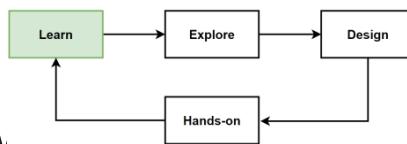




Learn: The 8. Pillar – Monitoring & Observability

- What are Monitoring Observability ?
- What are best practices of Monitoring and Observability in CN microservices ?
- How Monitoring and Observability applied in Cloud-Native microservices ?
- What are Health Monitoring ?
- How Health Monitoring uses to create Kubernetes clusters ?
- What are Distributed Logging-Tracing ?
- How Distributed Logging-Tracing uses to create Kubernetes clusters ?
- Design our E-Commerce application with Monitoring and Observability
- Implement Hands-on labs for Monitoring and Observability in CN Kubernetes cluster with Prometheus, Grafana, ELK ...





Where «Monitor&Observe» in 12-Factor App and CN Trial Map

Twelve-Factor App

- Codebase
- Dependencies
- Config
- Backing Services
- Build, release and run
- Processes
- Port Binding
- Concurrency
- Disposability
- Development/Prod Parity
- Logs
- Admin Processes



CLOUD NATIVE TRAIL MAP

The Cloud Native Landscape <https://github.com/cncf/landscape> has a growing number of options. This Cloud Native Trail Map is a recommended process for leveraging open source, cloud native technologies. At each step, you can choose a vendor-supported offering or do it yourself, and everything after step #3 is optional based on your circumstances.

HELP ALONG THE WAY

A. Training and Certification

Consider training offerings from CNCF and then take the exam to become a Certified Kubernetes Administrator <https://www.cncf.io/training>

B. Consulting Help

If you want assistance with Kubernetes and the surrounding ecosystem, consider leveraging a Kubernetes Certified Service Provider <http://cncf.io/ksp>

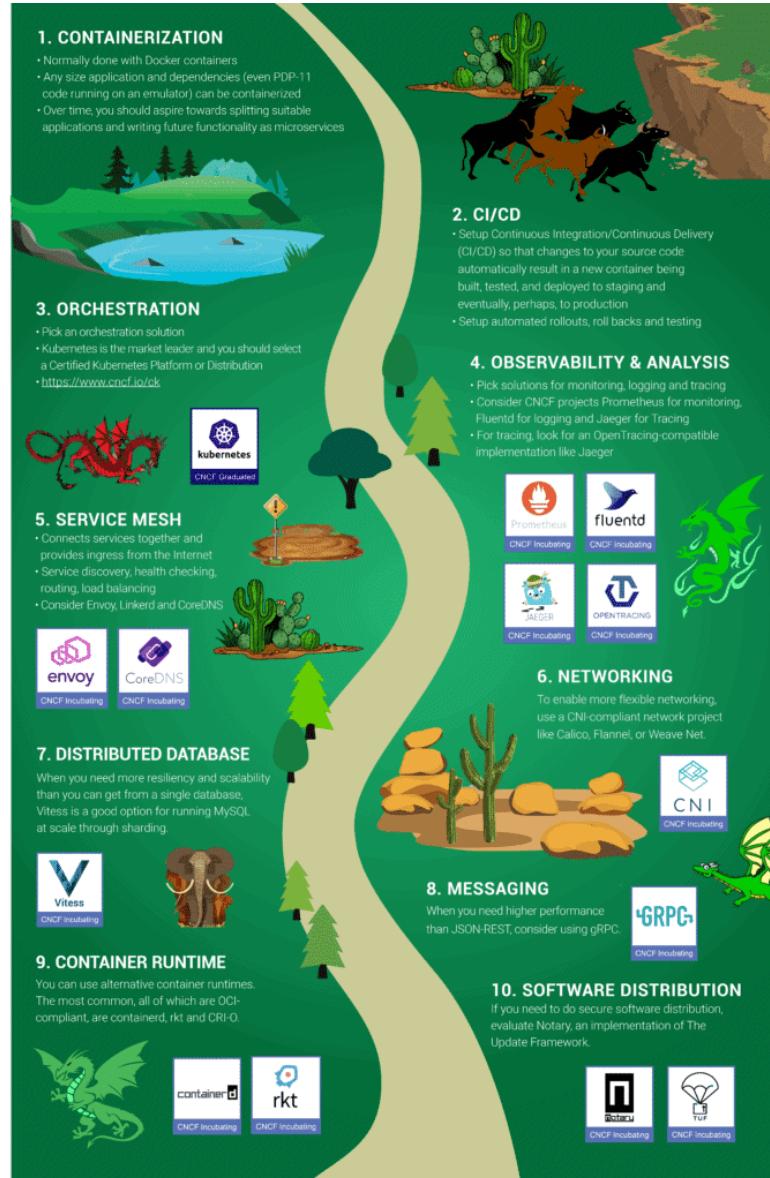
C. Join CNCF's End User Community

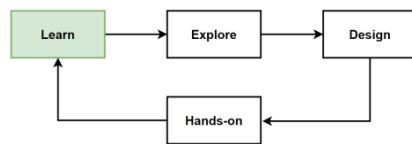
For companies that don't offer cloud native services externally <http://cncf.io/enduser>

WHAT IS CLOUD NATIVE?

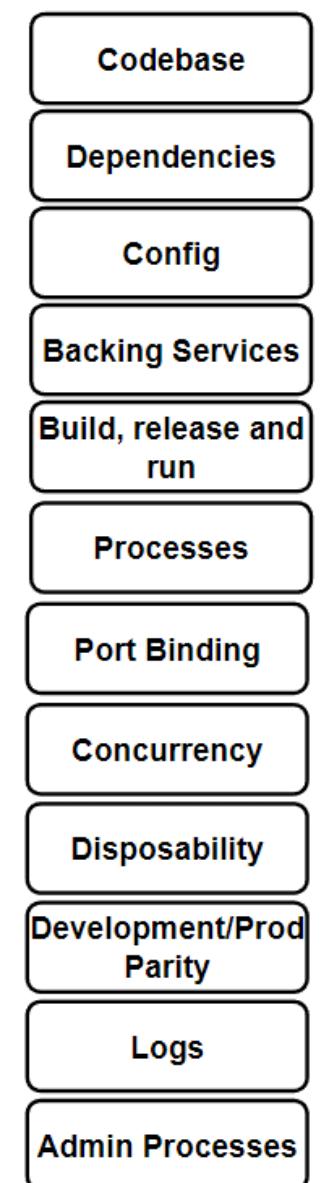
- **Operability:** Expose control of application/system lifecycle.
- **Observability:** Provide meaningful signals for observing state, health, and performance.
- **Elasticity:** Grow and shrink to fit in available resources and to meet fluctuating demand.
- **Resilience:** Fast automatic recovery from failures.
- **Agility:** Fast deployment, iteration, and reconfiguration.

www.cncf.io
info@cncf.io





Twelve-Factor App



12-Factor App – Cloud-Native Monitoring & Observability

VI. Processes

- Each process should be independent and isolated.
- Simplifies monitoring and observability since you can focus on each process separately and understand its behavior and performance.

XI. Logs

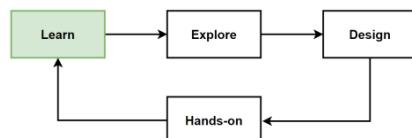
- Apps should treat logs as event streams and not be concerned with the storage and management of logs.
- Focusing on generating logs in a structured and standardized format.
- Easier to collect, aggregate, and analyze logs for monitoring and observability purposes.

VIII. Concurrency

- Scaling applications by running multiple instances of the application (process model).
- Track the health, performance, and resource usage of each instance, as well as aggregating the data to have a holistic view of the application.

IX. Disposability

- Fast startup and graceful shutdown apps that recover from crashes and failures quickly.
- Detecting such issues and ensuring the proper functioning of the application.



Cloud-native Trial Map—Monitoring & Observability

Monitoring and Observability

- This is 4. specific category in the trail map, which highlights the importance of monitoring and observability
- Emphasizes the need to monitor application and infrastructure health, performance, and behavior.

Logging and Tracing

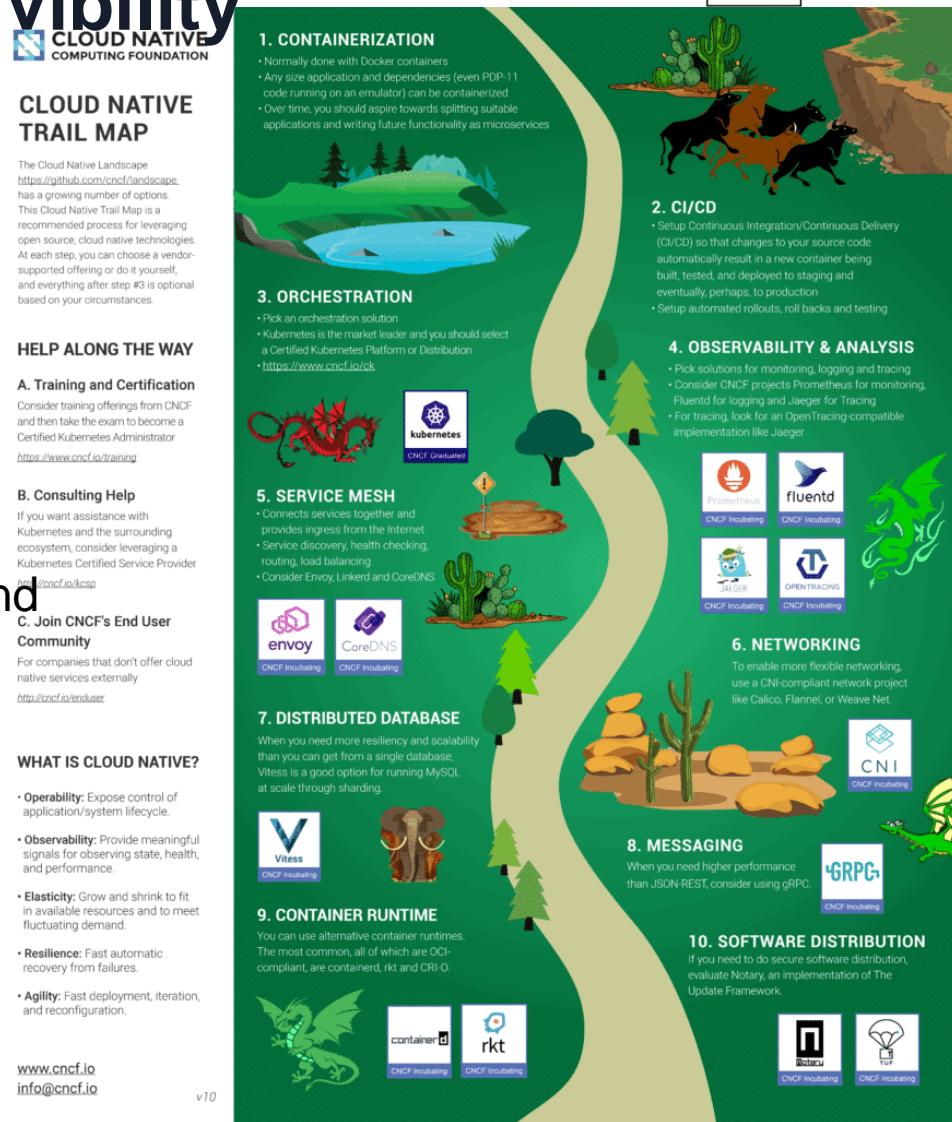
- Essential aspects of observability.
- Tools like Fluentd and Jaeger, which help in collecting, aggregating, and analyzing log data and tracing information across microservices.

Metrics and Analysis

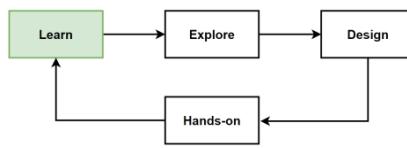
- Vital for monitoring the performance and health of cloud-native apps.
- Tools like Prometheus and Grafana, which provide a robust way to collect, store, and visualize metrics.

Service Mesh

- Tools like Linkerd and Istio, which provide observability features like distributed tracing, monitoring, and logging for microservices.



<https://www.cncf.io/blog/2018/03/08/introducing-the-cloud-native-landscape-2-0-interactive-edition/>



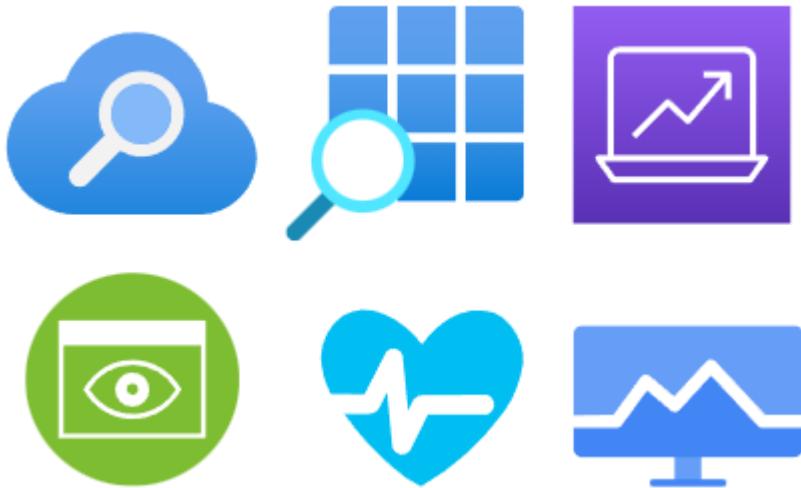
Monitoring & Observability in Cloud-native Applications

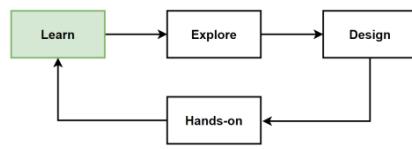
- Monitoring & Observability is huge topic in Cloud-native Applications.

Topics divided by 2 main and sub topics:

- Monitoring
 - Health Monitoring
 - Tools: Prometheus, Grafana
- Observability
 - Distributed Logging
 - Distributed Tracing
 - Chaos Engineering

- How CNCF divided Monitoring & Observability tools ?
- Goto-> <https://landscape.cncf.io/card-mode?category=observability-and-analysis&grouping=category>





CNCF 2022 Annual Survey: Observability tools show biggest growth

- [CNCF 2022 Annual Survey](#)

The year cloud native became the new normal

- Key Finding

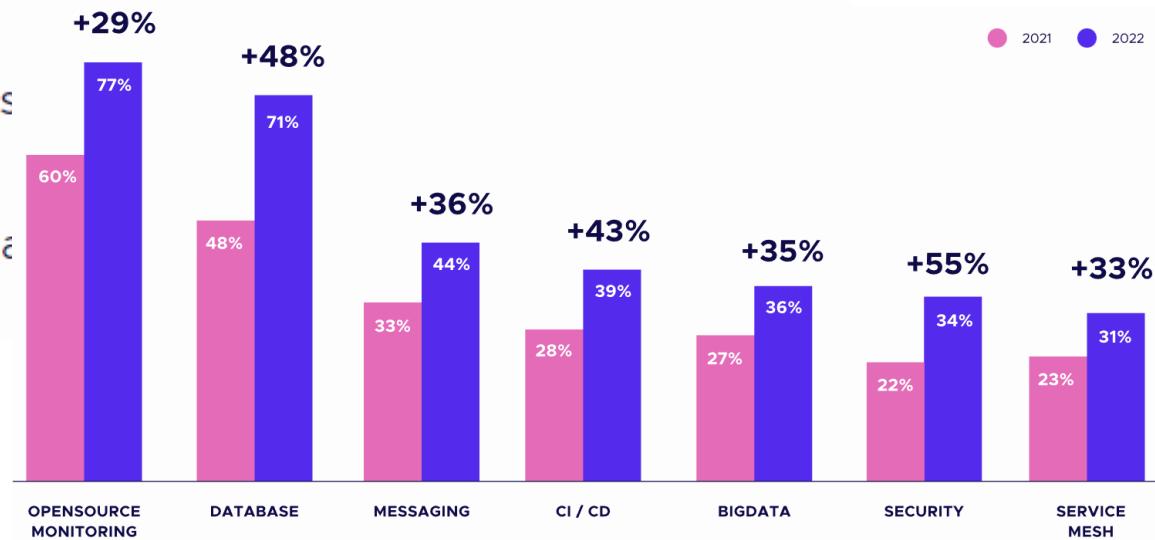
CNCF 2022 ANNUAL SURVEY

OBSERVABILITY TOOLS SHOW BIGGEST GROWTH IN PRODUCTION

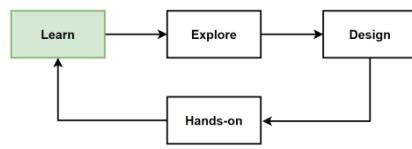
Dynatrace's data revealed a similar picture to CNCF survey responses

Kubernetes security tools increased from 22% in 2021 to 34% in 2022

55%. That trend will likely continue as security awareness grows and more tools become available.

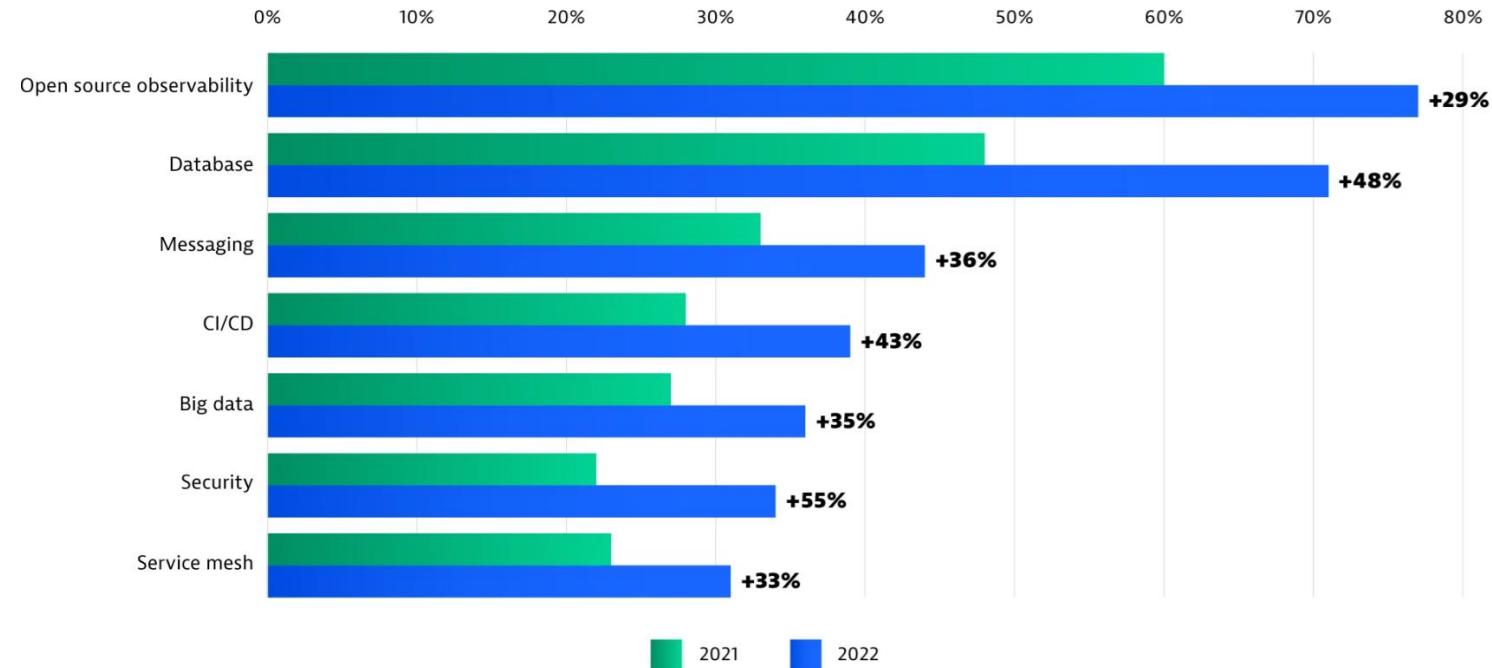


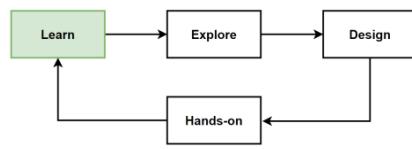
<https://www.cncf.io/reports/cncf-annual-survey-2022/#findings>



Top Kubernetes Workloads by Categories

- [Dynatrace Kubernetes Report: Strongest Kubernetes growth areas](#)
- Open-source projects in the Kubernetes across various categories, including **observability**, databases, messaging, CI/CD, big data, security, and service meshes.
- Open-source solutions: **Prometheus**, Redis, RabbitMQ, Kafka, ArgoCD, Flux, GitLab, Jenkins, **Elasticsearch**, Gatekeeper, and Istio, are widely used as backing services or tools to enhance the capabilities of cloud-native applications.





Why Monitoring Tools increases in K8s Workloads ?

Complexity of Microservices Architecture

- Organizations increasingly adopt microservices so the complexity of managing and monitoring microservices grows.
- Communicate with one another over a network, and it's crucial to understand the health and performance of these interactions.

Scalability and Dynamic Nature

- Containers can be spun up and down quickly, and applications can be scaled on demand.
- This dynamic nature requires sophisticated monitoring tools to ensure that the system is performing as expected and to catch any issues before they impact the users.

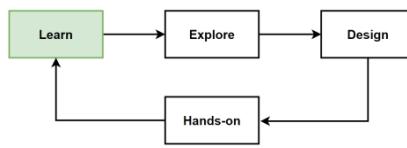
Troubleshooting and Root Cause Analysis

- In complex distributed system like K8s, understanding the root cause of an issue is challenging.
- Observability tools help in collecting data and metrics that can be critical in troubleshooting issues.

Compliance and Auditing

- In regulated industries, monitoring is essential not just for performance but for compliance reasons.
- They need to have detailed logs and metrics to satisfy regulatory requirements.

Monitoring



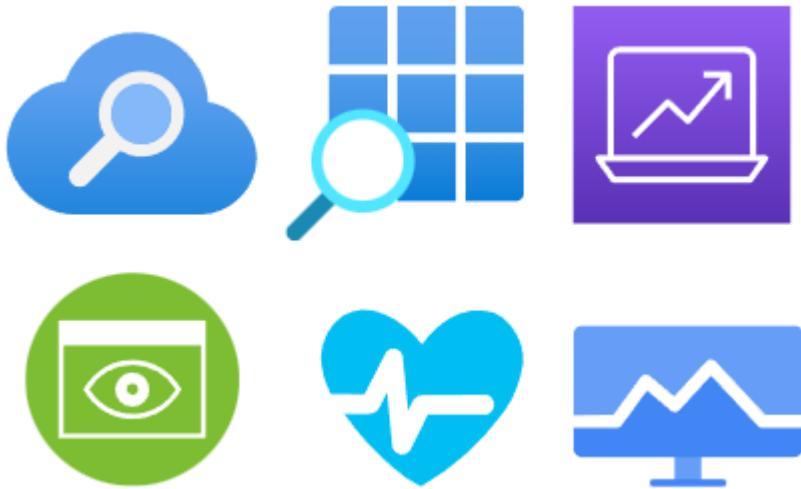
Monitoring & Observability in Cloud-native Applications

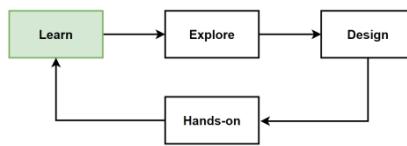
- Monitoring & Observability is huge topic in Cloud-native Applications.

Topics divided by 2 main and sub topics:

- Monitoring
 - Health Monitoring
 - Tools: Prometheus, Grafana
- Observability
 - Distributed Logging
 - Distributed Tracing
 - Chaos Engineering

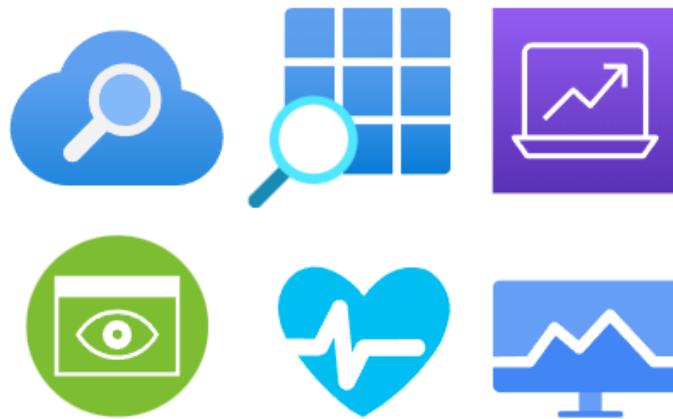
- How CNCF divided Monitoring & Observability tools ?
- Goto-> <https://landscape.cncf.io/card-mode?category=observability-and-analysis&grouping=category>

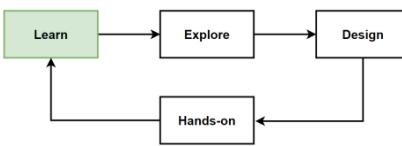




What is Monitoring ?

- **Monitoring** for cloud-native involves the **collection, analysis, and visualization** of **data** from microservices and the surrounding infrastructure in a cloud environment.
- This **data** can include **metrics, logs, and traces**.
- The goal of monitoring is to **gain insight into the performance, availability, and health of microservices**, as well as to **detect and troubleshoot issues**.
- In **highly dynamic and distributed** cloud-native environments, **effective monitoring** is particularly **critical**.





Key Components of Monitoring

Metrics

- Numerical data points that represent the state of a system at a point in time.
- Common metrics in microservices monitoring include request rate, error rate, response times, CPU usage, memory usage, and network throughput.

Logs

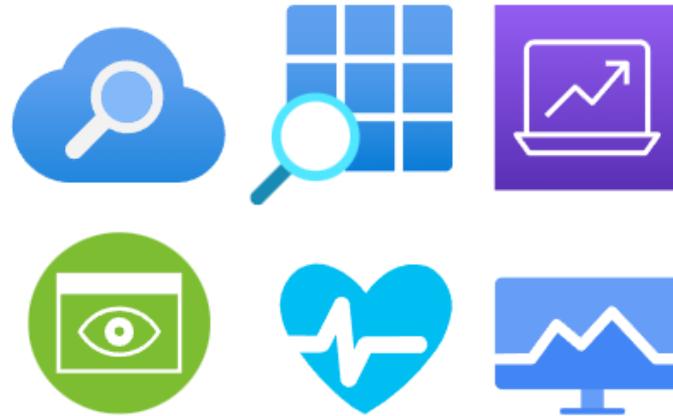
- Records of events that occur within an application or system.
- Invaluable when it comes to debugging and understanding the behavior of microservices.

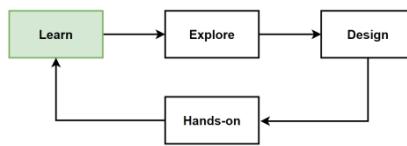
Traces

- Tracking requests useful in a microservices architecture where a single user request might involve multiple services.
- Understanding the flow of requests and identifying performance bottlenecks.

Alerting

- Receive alerts when something goes wrong or when certain thresholds are crossed.
- Notified of issues in real-time, allowing for quick resolution.





Key Components of Monitoring - 2

Visualization

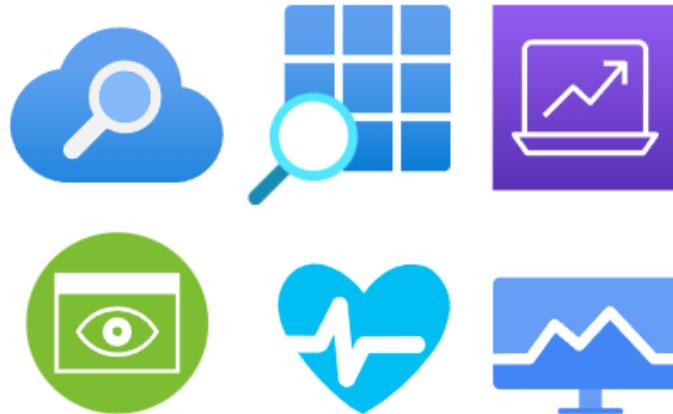
- Creating dashboards that can display metrics, logs, and traces in a user-friendly manner.
- Gaining insights at a glance and makes it easier to understand the state of the microservices and the underlying infrastructure.

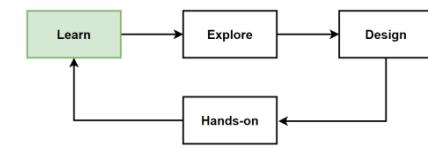
Health Checks

- Regularly checking the health of microservices is important.
- Involve simple checks to see if a service is up and running, or more complex checks like testing if a service can connect to a database.

Integration with CI/CD

- Monitoring should be integrated with CI/CD pipelines.
- Allows for a feedback loop where issues detected in monitoring can be addressed in continuous integration and deployment processes.





Microservices Health Checks: Liveness, Readiness and Performance Checks

- The **process of monitoring** the **health** and **performance** of individual microservices in a system.
- The **failure of a single microservice** can have **cascading effects** on the rest of the system, **important to identify** and **address issues**.
- What is **Health Checks** for microservices ?
- **Health Checks for microservices** are a way to **monitor the health and performance of individual microservices** in a system.
- **Health checks** used to determine whether a microservice is functioning properly and is able to handle requests.
- There are **3 types of health checks** that can be used for microservices.

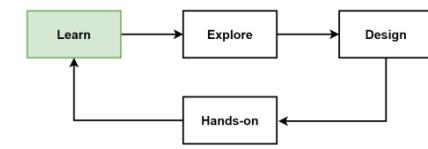
Health Checks UI

localhost:5107/hc-ui#/healthchecks

Health Checks status

Refresh every 10 seconds Change

	NAME	HEALTH	ON STATE FROM	LAST EXECUTION
+	WebMVC HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	WebSPA HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Web Shopping Aggregator GW HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Mobile Shopping Aggregator HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
-	Ordering HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
	NAME	HEALTH	DESCRIPTION	DURATION
self	✓ Healthy			00:00:00.0000031
orderingDB-check	✓ Healthy			00:00:00.0008153
ordering-rabbitmqbus-check	✓ Healthy			00:00:00.0097614
+	Basket HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Catalog HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Identity HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Marketing HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Locations HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:18 PM
+	Payments HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:18 PM
+	Ordering SignalRHub HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:18 PM



Microservices Health Checks: Liveness, Readiness and Performance Checks

▪ Liveness Checks

Determine whether a microservice is still running. If a liveness check fails, it may indicate that the microservice has crashed.

▪ Readiness Checks

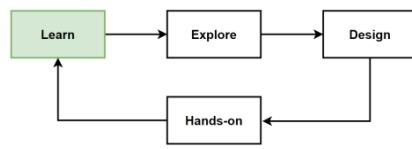
Determine whether a microservice is ready to handle requests. If a readiness check fails, it may indicate that the microservice is not yet ready to handle traffic.

▪ Performance Checks

Monitor the performance of a microservice, such as response times or error rates. If the results of a performance check indicate that a microservice is not performing as expected.

Health Checks status				Refresh every 10 seconds	Change
	NAME	HEALTH	ON STATE FROM	LAST EXECUTION	
+	WebMVC HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM	
+	WebSPA HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM	
+	Web Shopping Aggregator GW HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM	
+	Mobile Shopping Aggregator HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM	
-	Ordering HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM	

	NAME	HEALTH	DESCRIPTION	DURATION
self	✓ Healthy			00:00:00.000031
orderingDB-check	✓ Healthy			00:00:00.0008153
ordering-rabbitmqbus-check	✓ Healthy			00:00:00.0097614
+	Basket HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Catalog HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Identity HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Marketing HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:17 PM
+	Locations HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:18 PM
+	Payments HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:18 PM
+	Ordering SignalRHub HTTP Check	✓ Healthy	2 minutes ago	12/12/2019, 3:41:18 PM



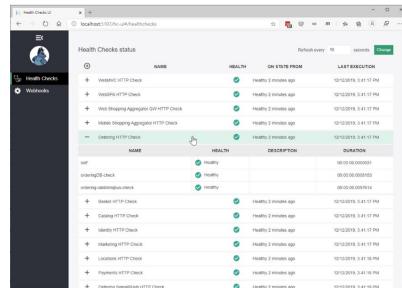
Microservices Health Checks: Liveness, Readiness and Performance Checks

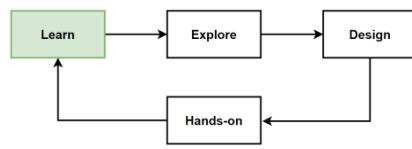
Liveness Checks

- Determine whether the microservice is running.
- If a liveness probe fails, the Kubernetes scheduler will automatically restart the affected container, which often resolves temporary issues like a deadlock.
- Use liveness checks to catch situations where the application is running but unable to make progress.
- **Example:** A simple HTTP GET endpoint within the microservice that returns a 200 OK status code.

Readiness Checks

- Determine whether the microservice is ready to accept requests.
- If a readiness probe fails, Kubernetes removes the pod from the service load balancer but doesn't restart it. This is useful for cases where the application is temporarily unable to serve, for instance during startup.
- Use readiness checks when the application needs to do some initialization work like caching data or connecting to databases before it can serve requests.
- **Example:** An HTTP endpoint that checks if the application can establish a connection to a database or if configuration files are fully loaded.





Microservices Health Monitoring with Kubernetes, Prometheus and Grafana

- **Use Liveness and Readiness Probes**

Kubernetes provides liveness and readiness probes that can be used to monitor the health of individual microservices.

- Liveness probes check to see if a microservice is still running, and readiness probes check to see if a microservice is ready to receive traffic.

- **Use Monitoring Tools**

Monitoring tools can be used to monitor the health and performance of microservices that can be integrated with Kubernetes to provide alerts or notifications when issues arise. I.e. Prometheus, Grafana, Datadog.

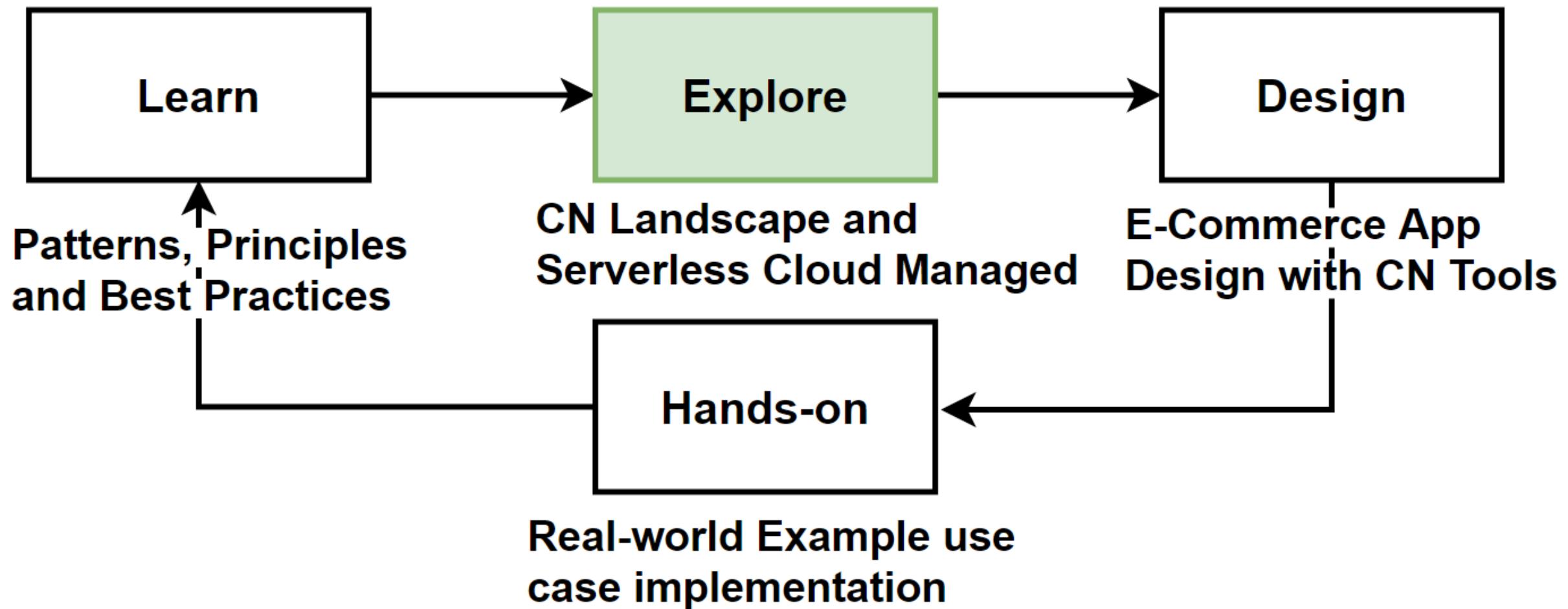
- **Use Log Analysis Tools**

Analyze log messages generated by your microservices and identify issues or trends. Elastic Stack (Elasticsearch, Logstash, Kibana), Fluentd, Splunk.

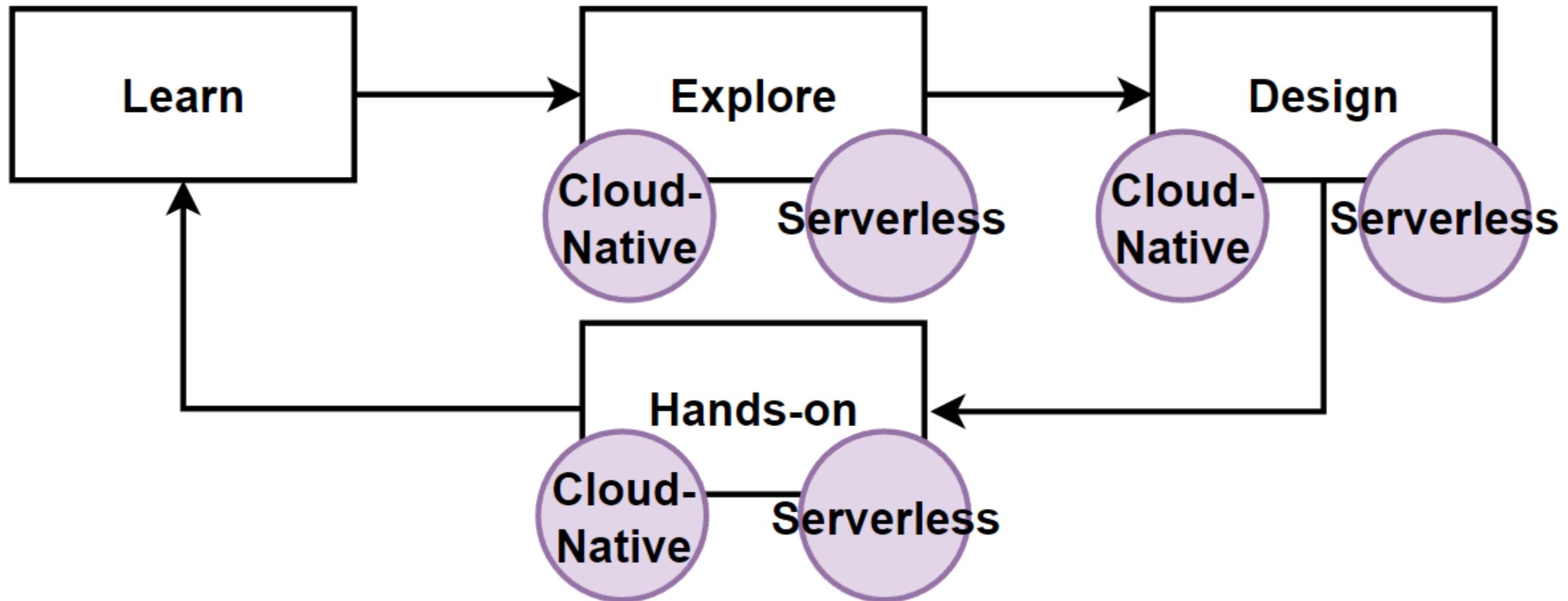
- **Set up Alerts and Notifications**

Setting up alerts and notifications can help to ensure that relevant parties are notified when issues arise, allowing them to be addressed quickly. Slack, Teams, Email, SMS.

Explore: Cloud Managed and Serverless Microservices Frameworks

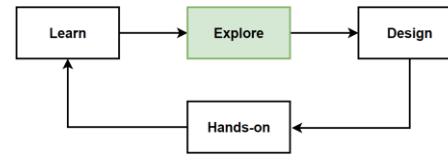


Way of Learning – Cloud-Native & Serverless Cloud Managed Tool



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs

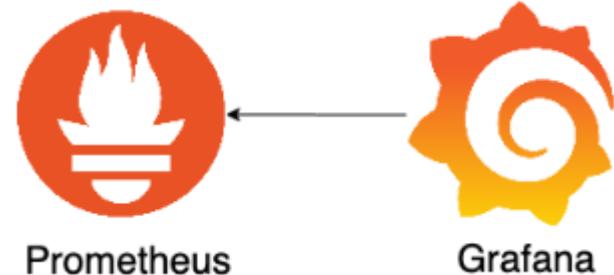


Explore: Cloud-Native Monitoring Tools

- Designed to support the dynamic nature of cloud environments, especially those built with microservices. Scalable, API-driven, and work well with containerized environments like Kubernetes.

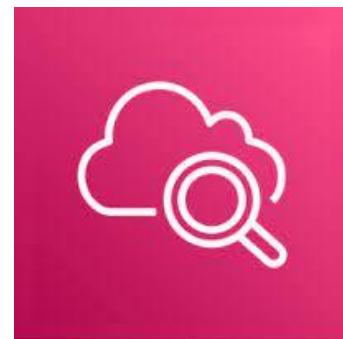
Cloud-Native Monitoring tools

- Prometheus
 - Grafana
 - Dynatrace
 - Datadog
 - Cortex
 - OpenMetrics



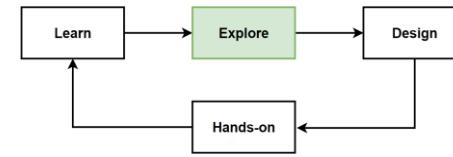
Cloud Serverless Monitoring tools

- Amazon CloudWatch and AWS X-Ray
 - Google Stackdriver
 - Microsoft Azure Monitor



Amazon CloudWatch

Goto -> <https://landscape.cncf.io/>



What is Prometheus ?

- **Prometheus** is an **open-source monitoring** and **alerting** toolkit designed for reliability and scalability.
- Widely adopted in cloud-native environments and is especially popular for monitoring Kubernetes clusters and microservices.
- How Prometheus is used in cloud-native Kubernetes ?

Data Collection and Storage

- Scrapes metrics from instrumented jobs
- Stores scraped samples locally
- Runs rules over data for aggregation or alerts

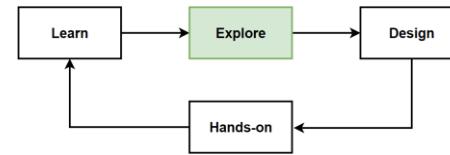
Service Discovery

- Automatically discovers scraping targets in Kubernetes
- Adapts to dynamically scheduled services

Metrics Exposition

- Microservices expose metrics via HTTP endpoint (e.g. /metrics)
- Uses libraries like client_golang, client_java





▪ How Prometheus is used in Cloud-native Kubernetes ?

Flexible Query Language (PromQL)

- Select and aggregate metric data
- Powerful for querying large number of services

Visualization

- Basic visualization via web UI
- Often used with Grafana for advanced dashboards

Alerting

- Define alert conditions based on Prometheus expressions
- Sends notifications through Alertmanager

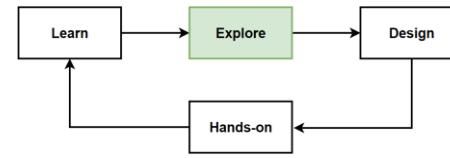
Integration with Kubernetes

- Works with Kubernetes labels for flexible queries
- Aligns with Kubernetes metadata definitions

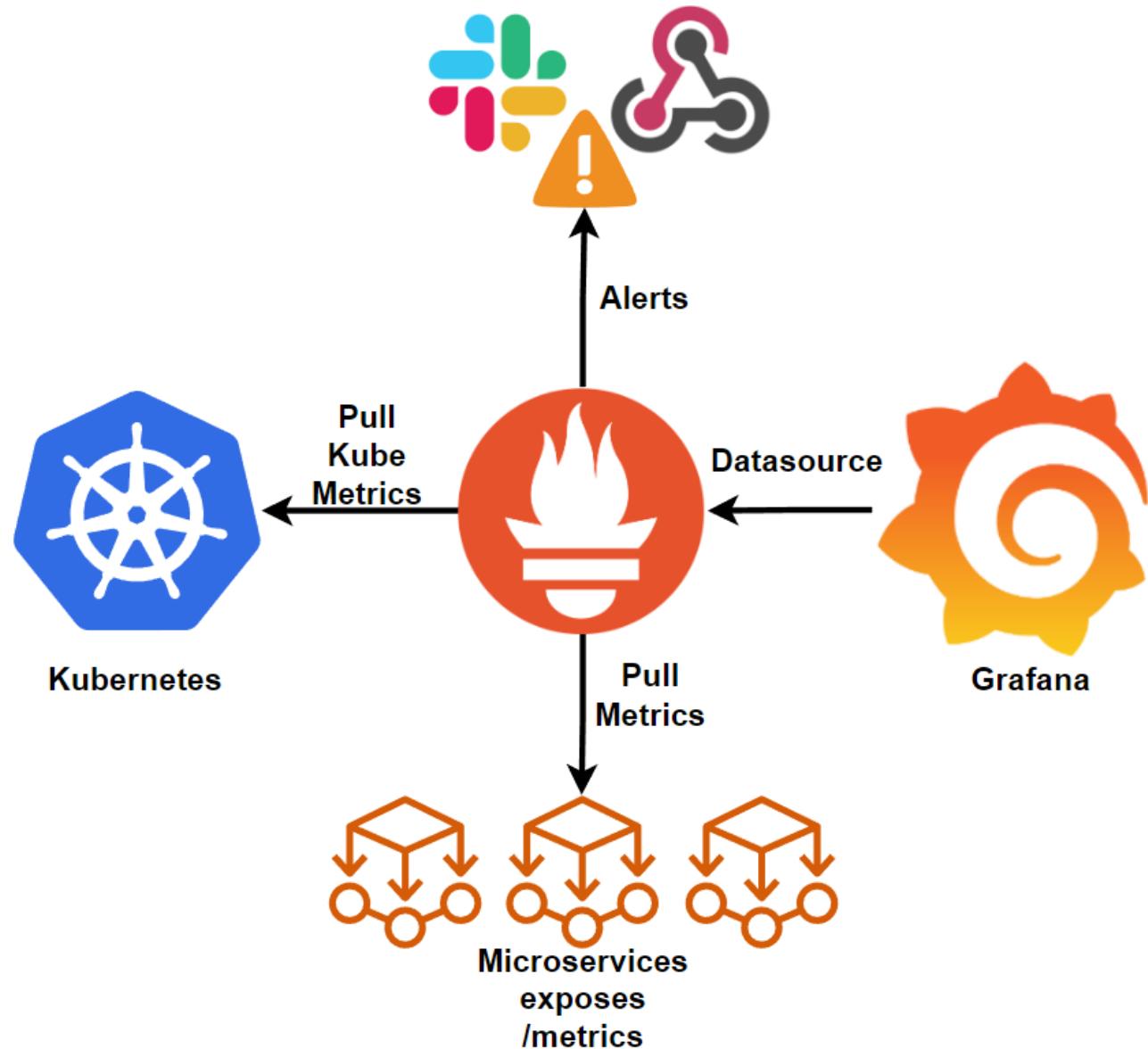
Use Cases

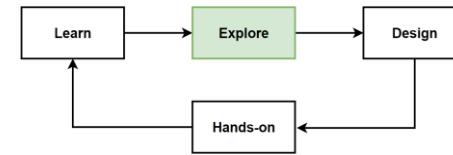
- Comprehensive observability of microservices in Kubernetes
- Integrated with Grafana for visualization and Alertmanager for notifications





How Prometheus Works ?





How Prometheus Works ?

Setup and Configuration of Prometheus in Kubernetes

- Deploy Prometheus to your Kubernetes cluster.
- Configure Prometheus to discover and scrape metrics from your microservices.

Instrumenting Microservices

- Implement metric exposition in your microservices using Prometheus client libraries.
- Expose metrics through an HTTP endpoint (usually /metrics).

Service Discovery

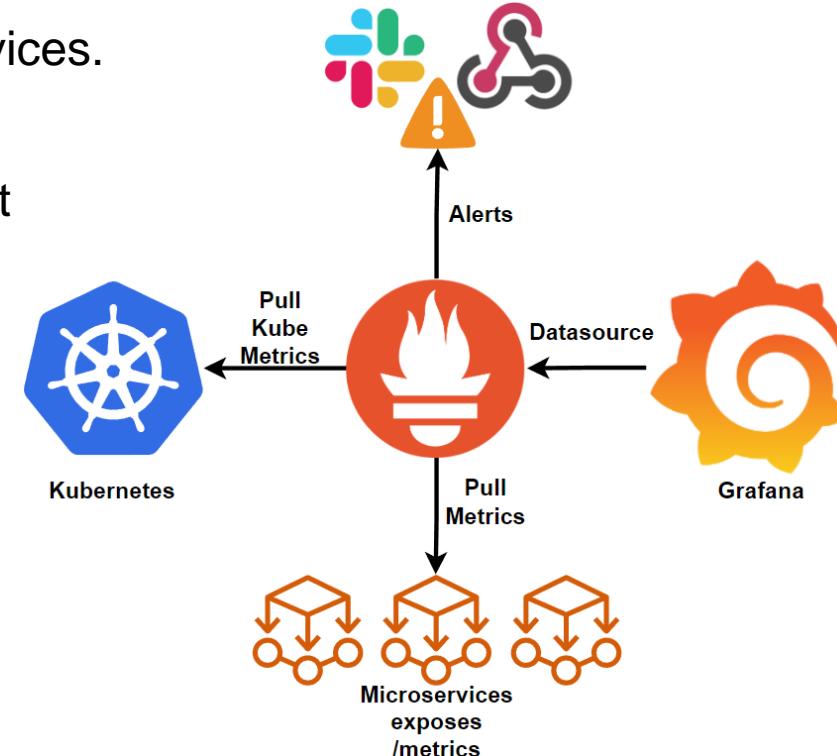
- Use Prometheus's Kubernetes service discovery to dynamically discover services to monitor.

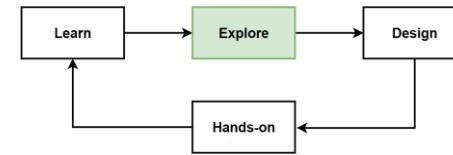
Defining Recording Rules and Alerts

- Use recording rules to precompute frequently needed or computationally expensive expressions.

Querying Metrics with PromQL

- Use PromQL, Prometheus's flexible query language, to query your metrics.





How Prometheus Works ? - 2

Visualizing Data

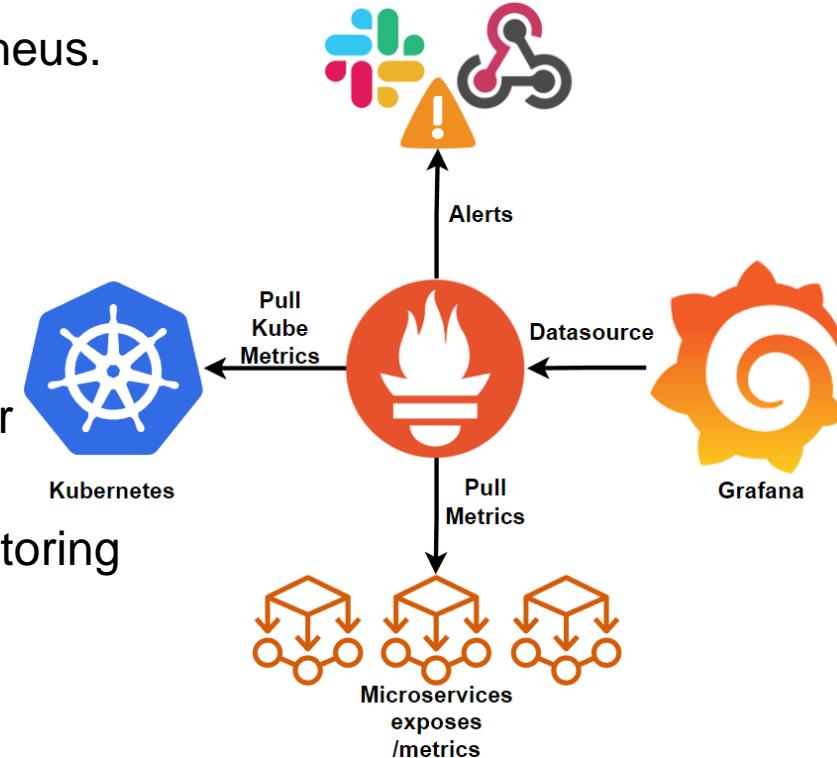
- Integrate Prometheus with Grafana.
- Setup dashboards in Grafana to visualize the metrics collected by Prometheus.

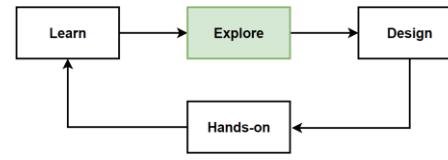
Setting up Alertmanager

- Configure Alertmanager to group, suppress, and route the alerts to the appropriate channel like email, Slack, PagerDuty, etc.

Gaining Insights and Taking Action

- Use the insights gained from monitoring to understand the behavior of your microservices.
- Take actions like scaling, optimizing, or troubleshooting based on the monitoring data.



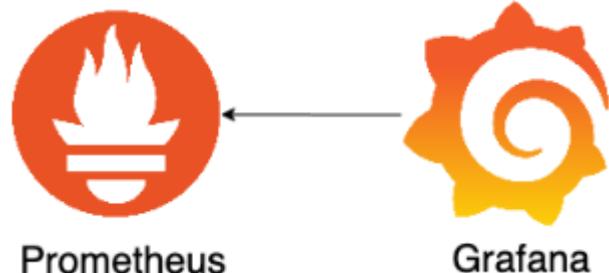


Explore: Cloud-Native Monitoring Tools

- Designed to support the dynamic nature of cloud environments, especially those built with microservices. Scalable, API-driven, and work well with containerized environments like Kubernetes.

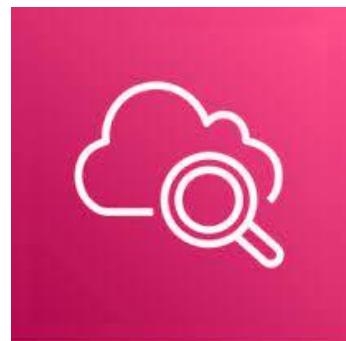
Cloud-Native Monitoring tools

- Prometheus
 - Grafana
 - Dynatrace
 - Datadog
 - Cortex
 - OpenMetrics



Cloud Serverless Monitoring tools

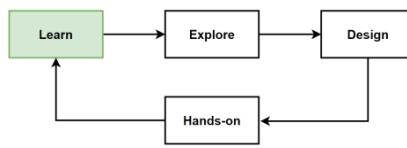
- Amazon CloudWatch and AWS X-Ray
 - Google Stackdriver
 - Microsoft Azure Monitor



Amazon CloudWatch

Goto -> <https://landscape.cncf.io/>

Distributed Logging

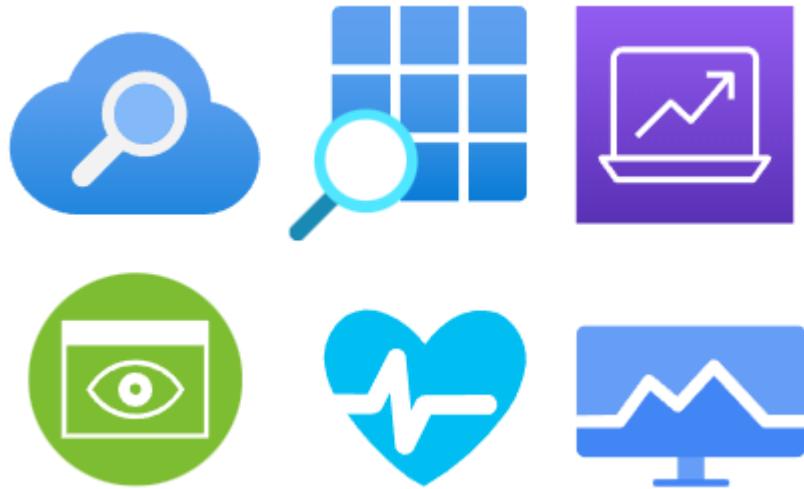


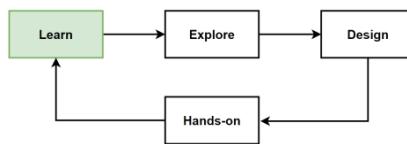
Monitoring & Observability in Cloud-native Applications

- Monitoring & Observability is huge topic in Cloud-native Applications.

Topics divided by 2 main and sub topics:

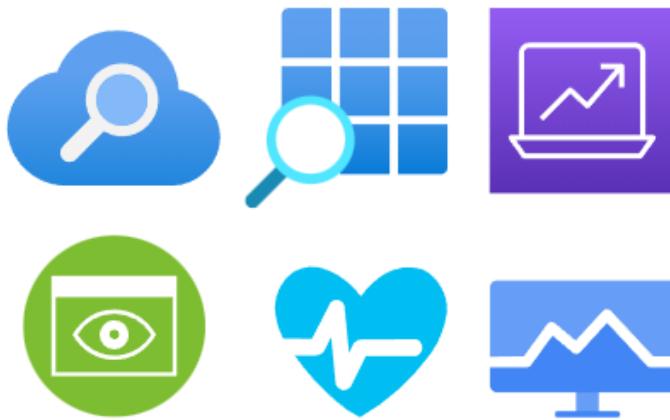
- Monitoring
 - Health Monitoring
 - Tools: Prometheus, Grafana
- Observability
 - Distributed Logging
 - Distributed Tracing
 - Chaos Engineering
- How CNCF divided Monitoring & Observability tools ?
- Goto-> <https://landscape.cncf.io/card-mode?category=observability-and-analysis&grouping=category>



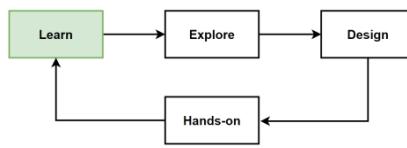


What is Distributed Logging ?

- **Monitoring** for cloud-native involves the **collection, analysis, and visualization** of **data** from microservices and the surrounding infrastructure in a cloud environment.
- This **data** can include **metrics, logs, and traces**.
- The goal of monitoring is to **gain insight into the performance, availability, and health of microservices**, as well as to **detect and troubleshoot issues**.
- In **highly dynamic and distributed** cloud-native environments, **effective monitoring** is particularly **critical**.



Distributed Tracing



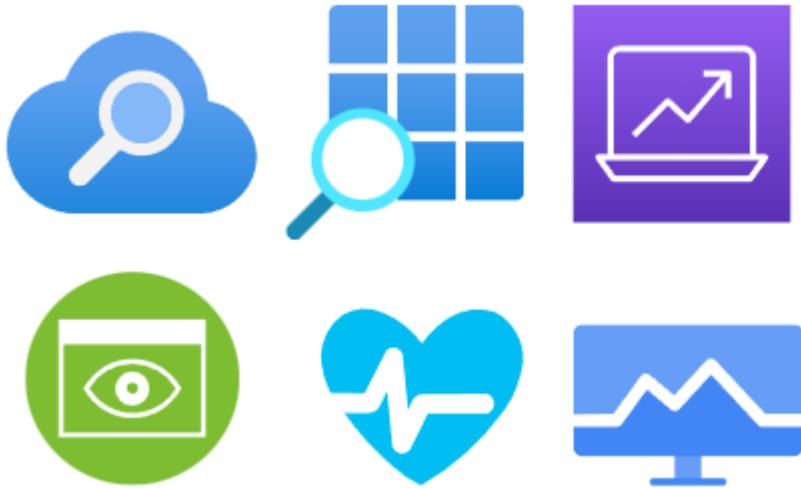
Monitoring & Observability in Cloud-native Applications

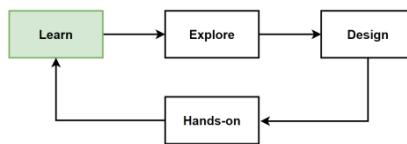
- Monitoring & Observability is huge topic in Cloud-native Applications.

Devops Topics divided by 2 main and sub topics:

- Monitoring
 - Health Monitoring
 - Tools: Prometheus, Grafana
- **Observability**
 - Distributed Logging
 - **Distributed Tracing**
 - Chaos Engineering

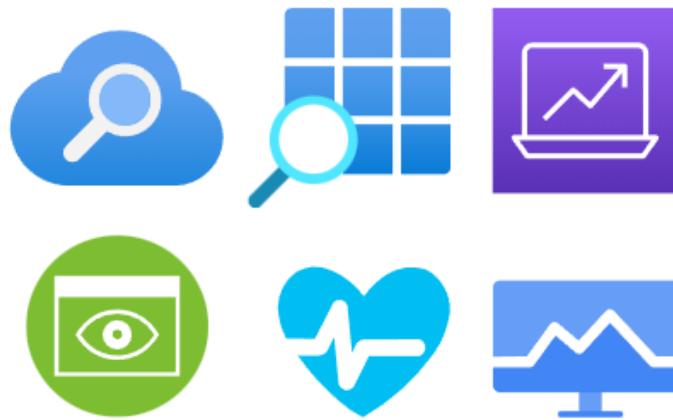
- How CNCF divided Monitoring & Observability tools ?
- Goto-> <https://landscape.cncf.io/card-mode?category=observability-and-analysis&grouping=category>





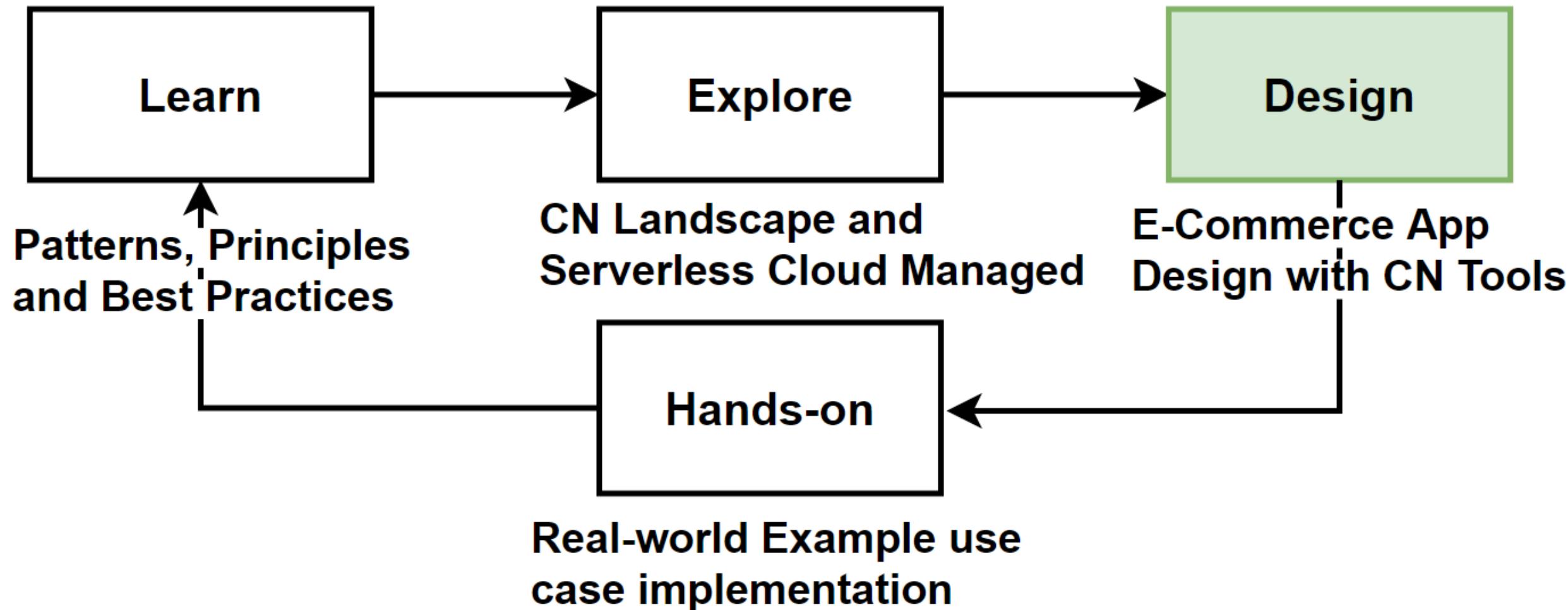
What is Distributed Logging ?

- **Monitoring** for cloud-native involves the **collection, analysis, and visualization** of **data** from microservices and the surrounding infrastructure in a cloud environment.
- This **data** can include **metrics, logs, and traces**.
- The goal of monitoring is to **gain insight into the performance, availability, and health of microservices**, as well as to **detect and troubleshoot issues**.
- In **highly dynamic and distributed** cloud-native environments, **effective monitoring** is particularly **critical**.

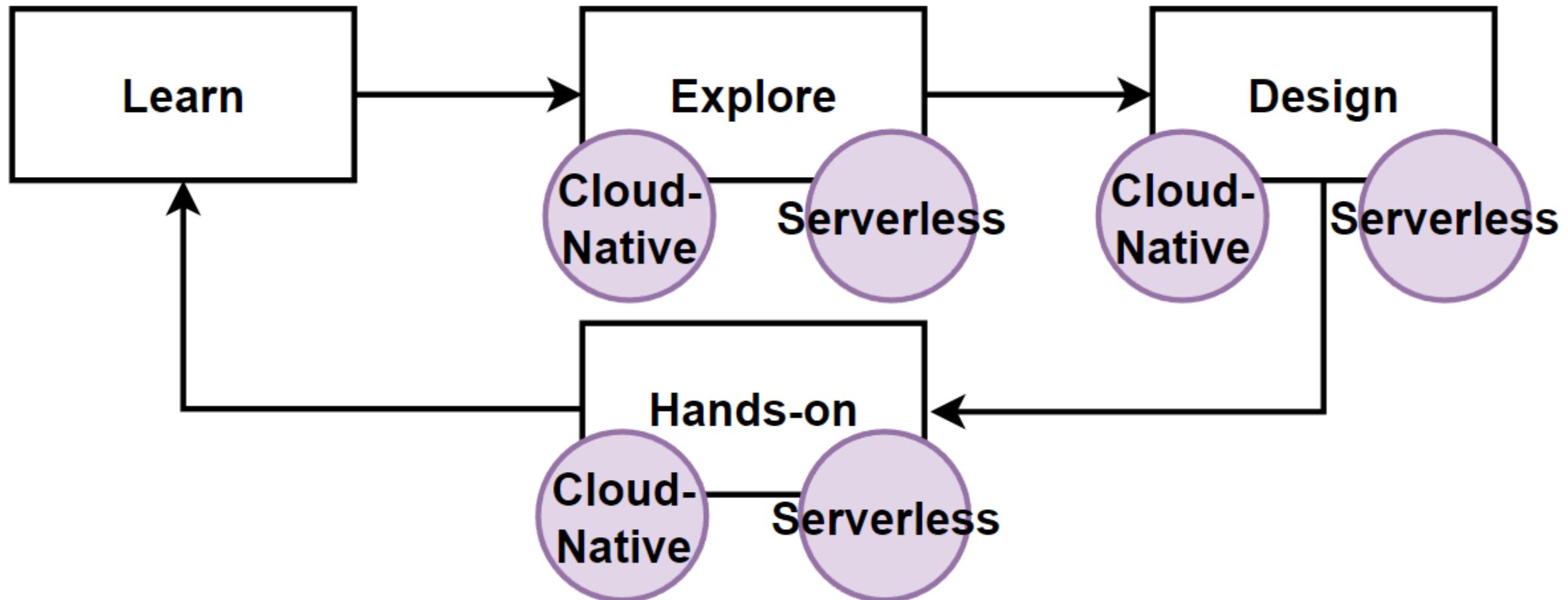


Design with Monitoring & Observability Tools

Way of Learning – The Course Flow

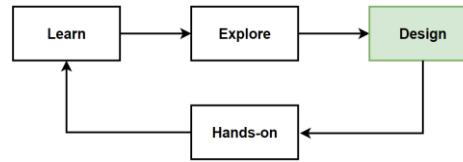


Way of Learning – Cloud-Native & Serverless Cloud Managed



Examples of CN vs Serverless Cloud Tools:

- Kubernetes - EKS, AWS Fargate
- Kafka - Azure Service Bus
- Prometheus - Amazon Cloudwatch Logs



Tools: Cloud-Native Monitoring & Observability

- Cloud-native DevOps tools that we picked for designing e-commerce microservices application:

CI/CD Pipeline

- GitHub Actions

IaC

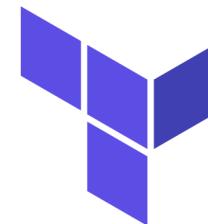
- Terraform

GitOps

- ArgoCD



GitHub Actions

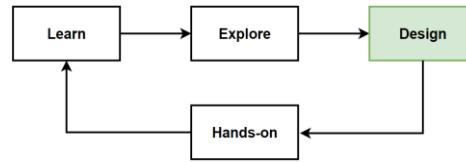


HashiCorp
Terraform

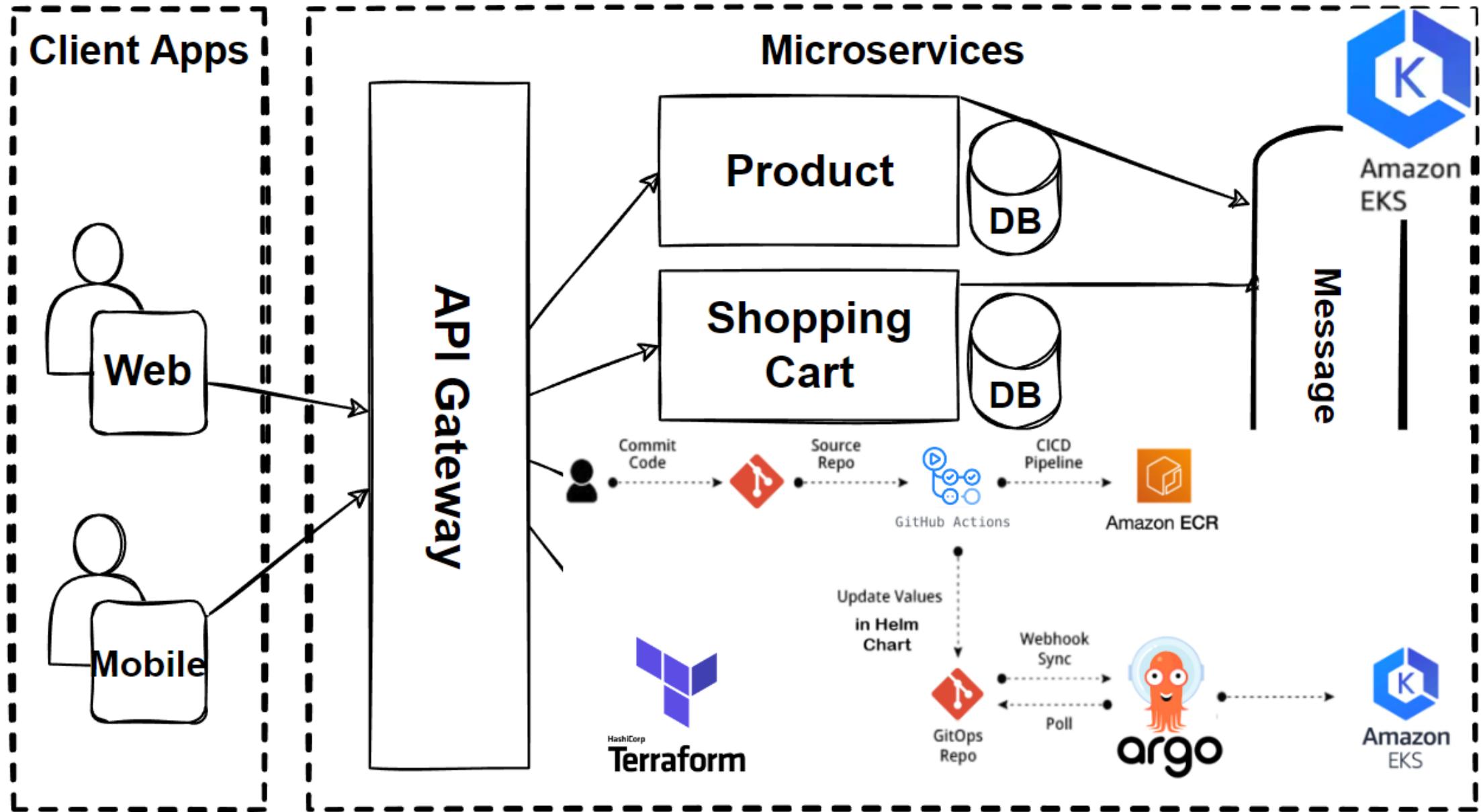


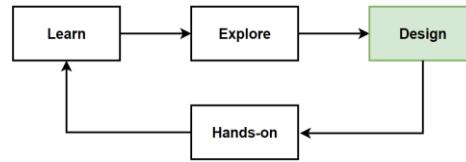
argo

Goto -> <https://landscape.cncf.io/>



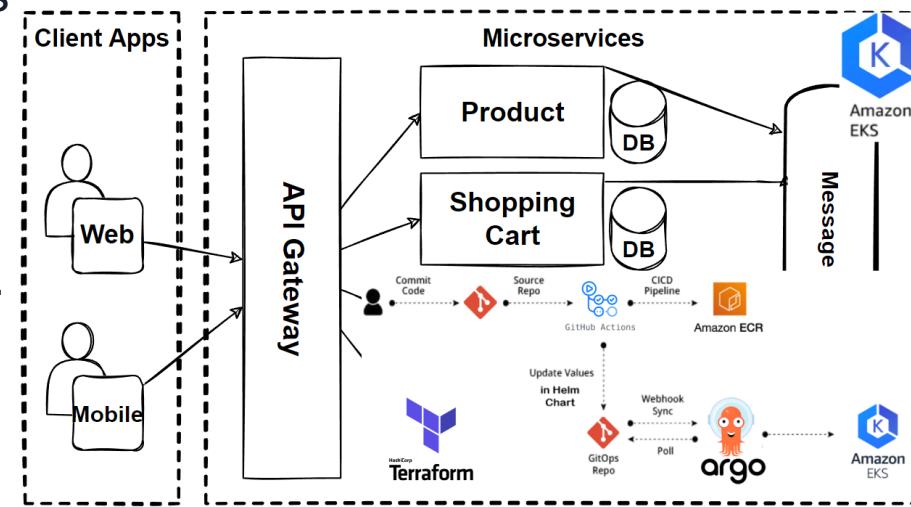
Microservices with DevOps, CI/CD, IaC and GitOps

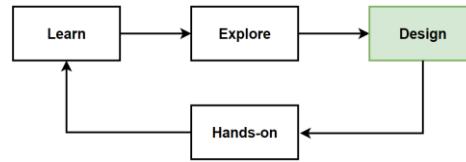




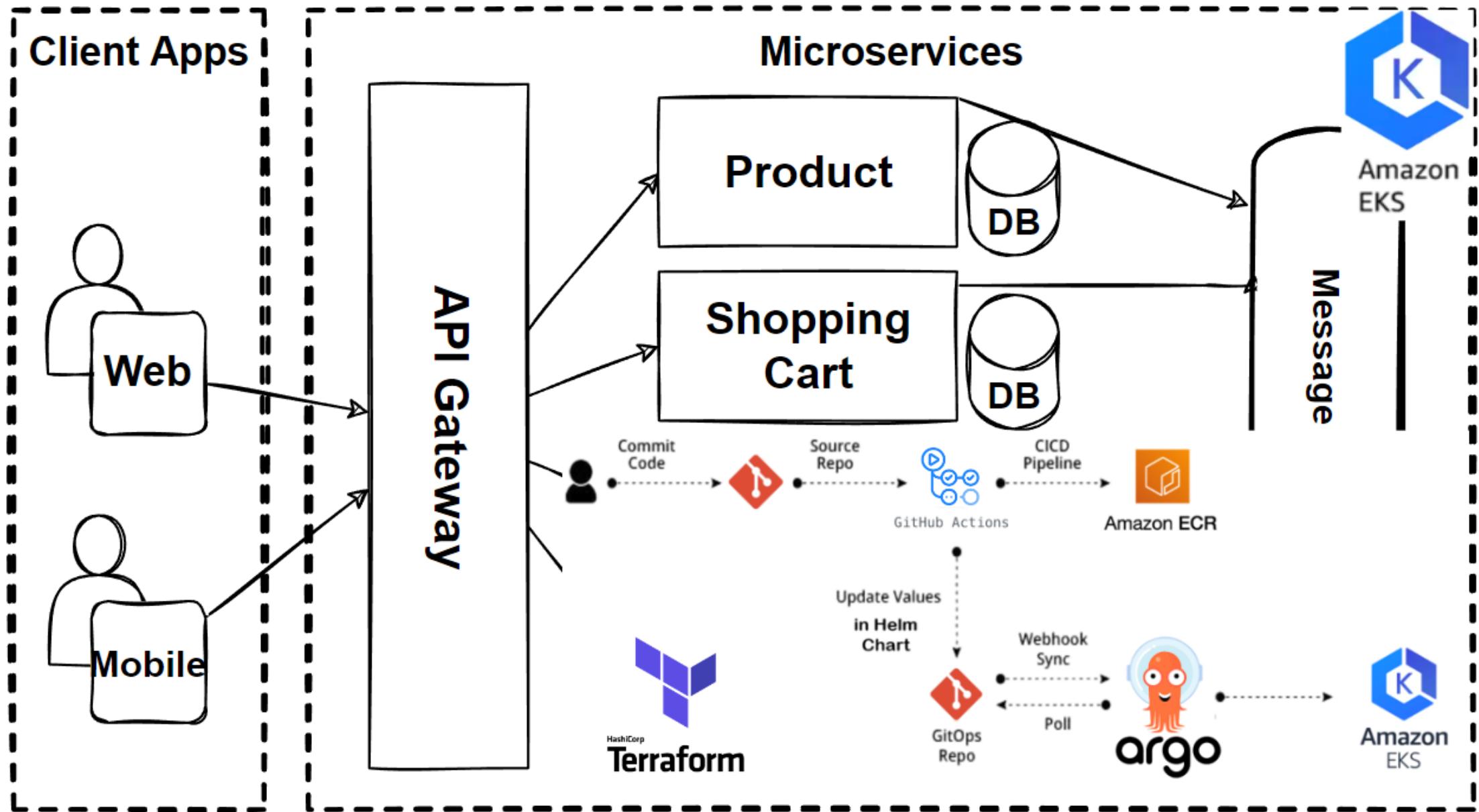
Steps of DevOps, CI/CD, IaC and GitOps Architecture

- 1. Infrastructure Provisioning:** Terraform configuration files to describe your AWS infrastructure including the EKS cluster, ECR repositories.
- 2. Set Up ArgoCD:** Deploy ArgoCD in your EKS cluster. Configure ArgoCD to monitor your GitHub repositories for changes in Kubernetes manifests.
- 3. Develop Microservices:** Writes code for the microservices and pushes to GitHub.
- 4. GitHub Actions Trigger:** It builds a Docker image of the microservice. Pushes the Docker image to ECR. Updates the Kubernetes manifest files with the new image tag.
- 5. ArgoCD Observes and Deploys:** ArgoCD notices the changes in Kubernetes manifests. It pulls these manifest files. Deploys the updated microservice to the EKS cluster by applying the manifests.
- 6. Monitor and Observe:** Set up monitoring and logging solutions like Prometheus and Grafana, AWS CloudWatch for observing the performance and health of your microservices.





Microservices with DevOps, CI/CD, IaC and GitOps



Thanks

Thank you so much for being with me on this journey.

Reviews and feedback is really encourage to me for pushing forward to create new courses like this.

Mehmet Ozkaya

