AI Systems Engineering

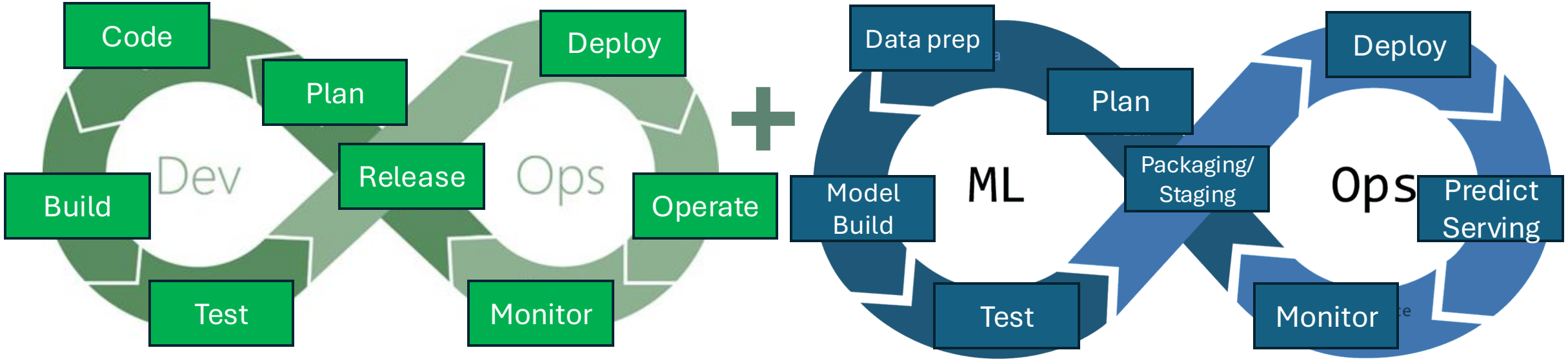
Prof. Roberto Pietrantuono

Lecture 6-7

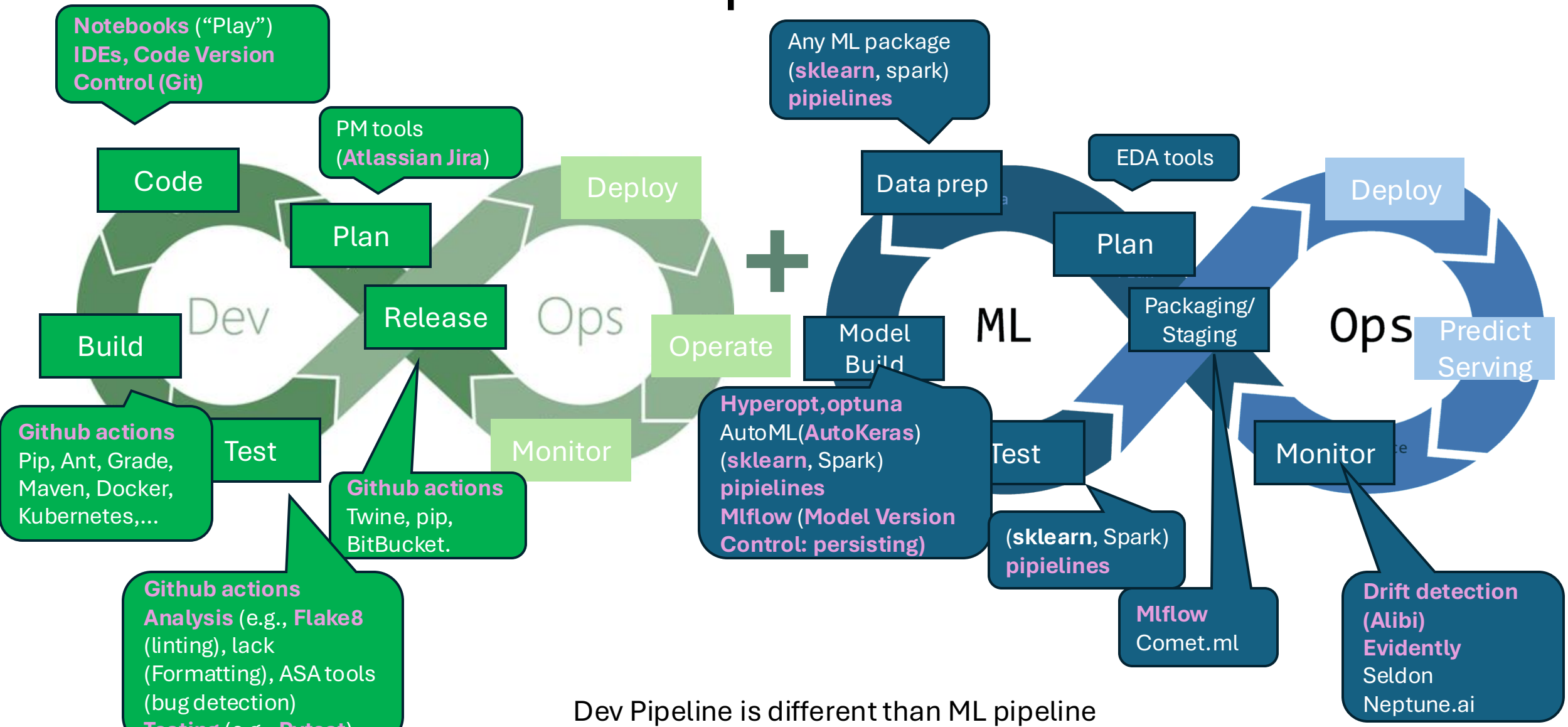
Outline

- From **Development to Deployment**: Architectural Patterns
  - Batch vs Stream Processing
  - Microservices and Containers
- Deployment Example
  - Microsoft Azure
- Pipelining

Where we are in the process...



Where we are in the process…



Dev Pipeline is different than ML pipeline

Useful Architectural (and ML) patterns

- …to increase quality while transitioning from development to deploy
- The role of software architectures
  - Separation of concerns
  - Modularity
  - Reuse
  - Extensibility
  - Technologies/tools
  - From problem to solution: tuning cost-time-quality trade-offs

Design choices

- User-system interaction, interfaces design
- Persistence model, consistency model (recall: “Data conditioning” in our functional architecture)
- Machine learning algorithm decisions (learning strategy, type of algorithm, hyperparameters, training system, …; Recall: “ML” in our functional architecture)
- Performance, efficient resource usage
- Distributed architectures (**architectural patterns**)
- Deployment options
- Software-to-hardware allocation (Recall: “Modern computing” in our functional architecture)
- …

Architectural patterns

- Client/Server
- Two e Three-tiers
- SOA, MSA
- Shared data-Repository, Publish-Subscribe
- …

Principles

- **SOLID principles:**
  - **Single responsibility**
  - **Open/Closed** (“*Open for extension but Closed for modification*”)
  - **Liskow substitution**: components with the same interface/contract can be swapped
  - **Interface segregation** (“*no code should be forced to depend on methods it does not use*”, “*don’t have multiple ways for components to talk to one another*”, contract)
  - **Dependency Inversion** (absrtact the interaction between high-level and low-level modules)
    1. High-level modules should not import anything from low-level modules. Both should depend on abstractions (e.g., interfaces)
    2. Abstractions should not depend on details. Details (concrete implementations) should depend on abstractions

Useful Architectural (and ML) patterns

- **IDEALS** principles (for Microservices)
  - **Interface segregation**
  - **Deployability:** design and technology choices for packaging, deploying, running MS
  - **Event-driven:** whenever possible, services activated by asynchronous message/event
  - **Availability over consistency:** often, users value availability over strong consistency
  - **Loose-coupling**
  - **Single responsibility:** modeling microservices for one or few cohesive functionalities

Useful Architectural (and ML) patterns
Microservices Architecture (MSA)

Definition: Architectural style that structures an application as a collection of loosely coupled services.

Key Characteristics:

- Independent deployment
- Focus on a single business capability
- Communicate via APIs (RESTful architectural style)
  - REST is stateless (no session) and based on HTTP
  - GET, POST, PUT, DELETE for CRUD operations (plus otehrs)

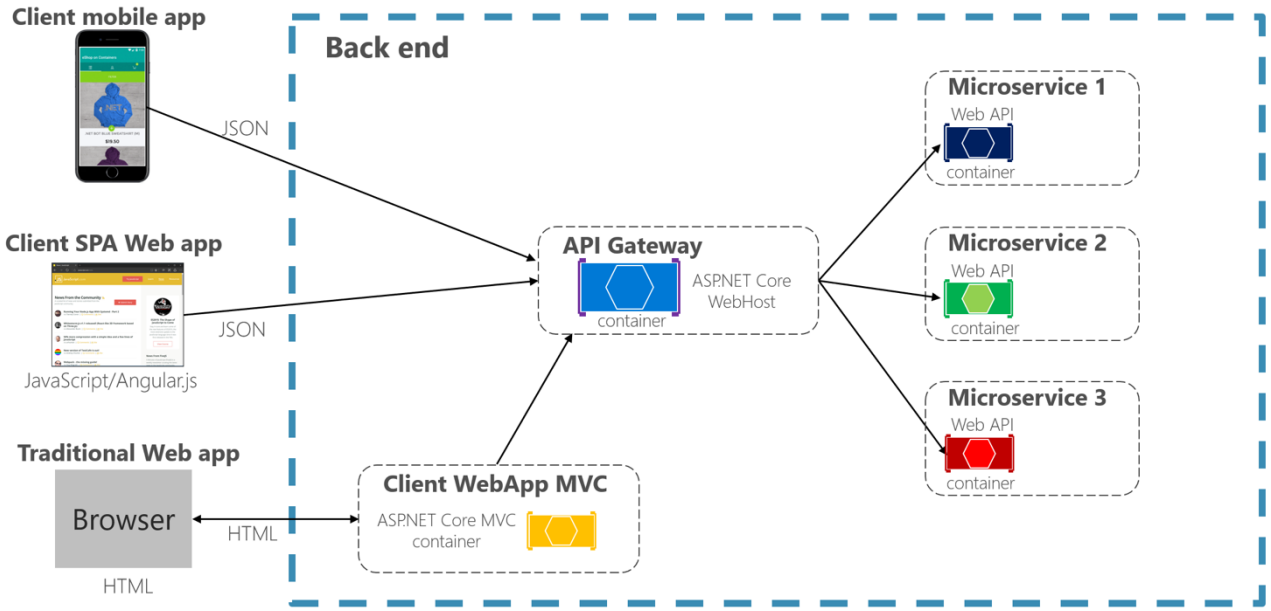
Why Microservices?

- Not scalable
- Unreliable
- Not flexible
- Blocks continuous development
- Slow development
- Large and complex applications

- Loosely coupled
- Run in their own processes
- Interact via lightweight (REST) mechanisms
- Independently development, debug, patch, or deploy individual services rather than tearing down the whole system.
- Avoid a single point of failure
- Fault isolation
- Increase maintainability.
- Allow separate services to be owned by distinct teams with clearer responsibilities.
- Accelerate development
- Scalability
- Techonology diversity

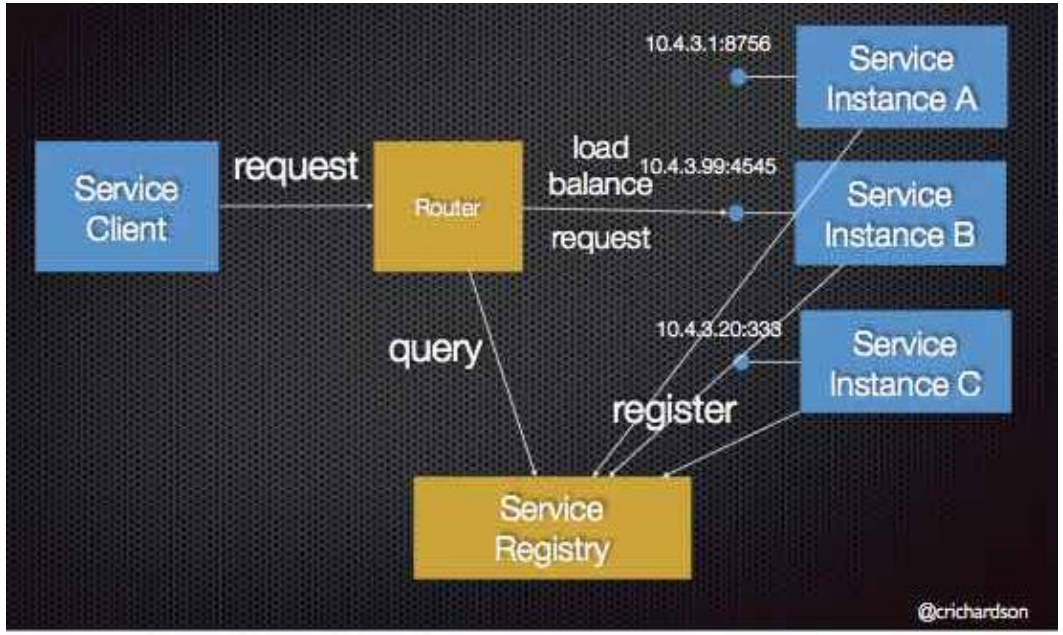
Microservices: key design patterns

- **API Gateway**
  - A single entry point for clients.
- Benefits
  - Request routing
  - Load balancing
  - Security and authentication



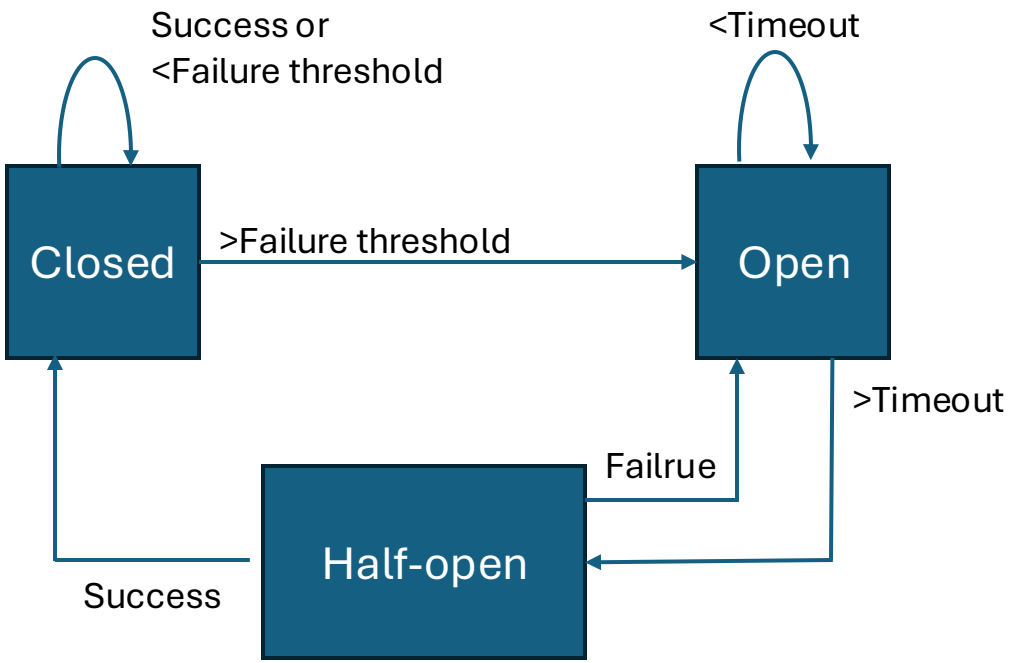
Microservices: key design patterns

- **Service Discovery**
  - Locating service instances dynamically.
- Types
  - Client-side discovery
  - Server-side discovery
- Tools
  - Eureka
  - Consul
  - Zookeeper



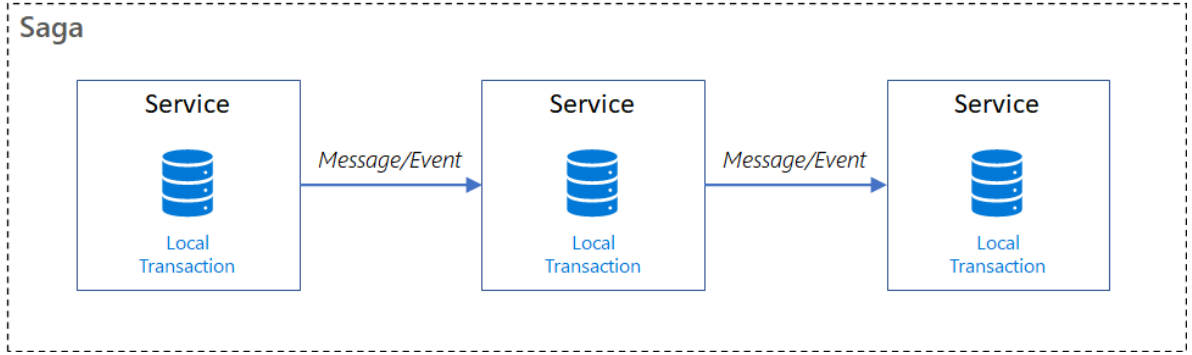
Microservices: key design patterns

- **Circuit Breaker**
  - Prevents cascading failures.
- How it Works:
  - Opens the circuit when failures exceed a threshold
- Benefits:
  - Resilience
  - Improved user experience



Microservices: key design patterns

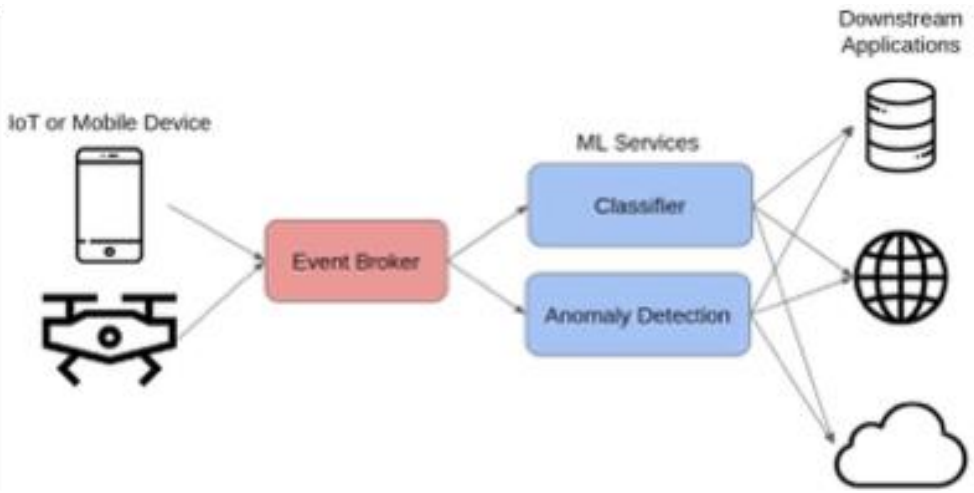
- **Saga Pattern**
  - Managing distributed transactions.
- Types:
  - Choreography
  - Orchestration



- **Other patterns:**
  - **Event Sourcing**: Storing state changes as a sequence of events.
  - **CQRS (Command Query Responsibility Segregation)**: Separating read and write operations.
  - **Strangler Fig Pattern**: Gradually replacing parts of a monolith with microservices.
  - **Asynch messages**

Event-based design

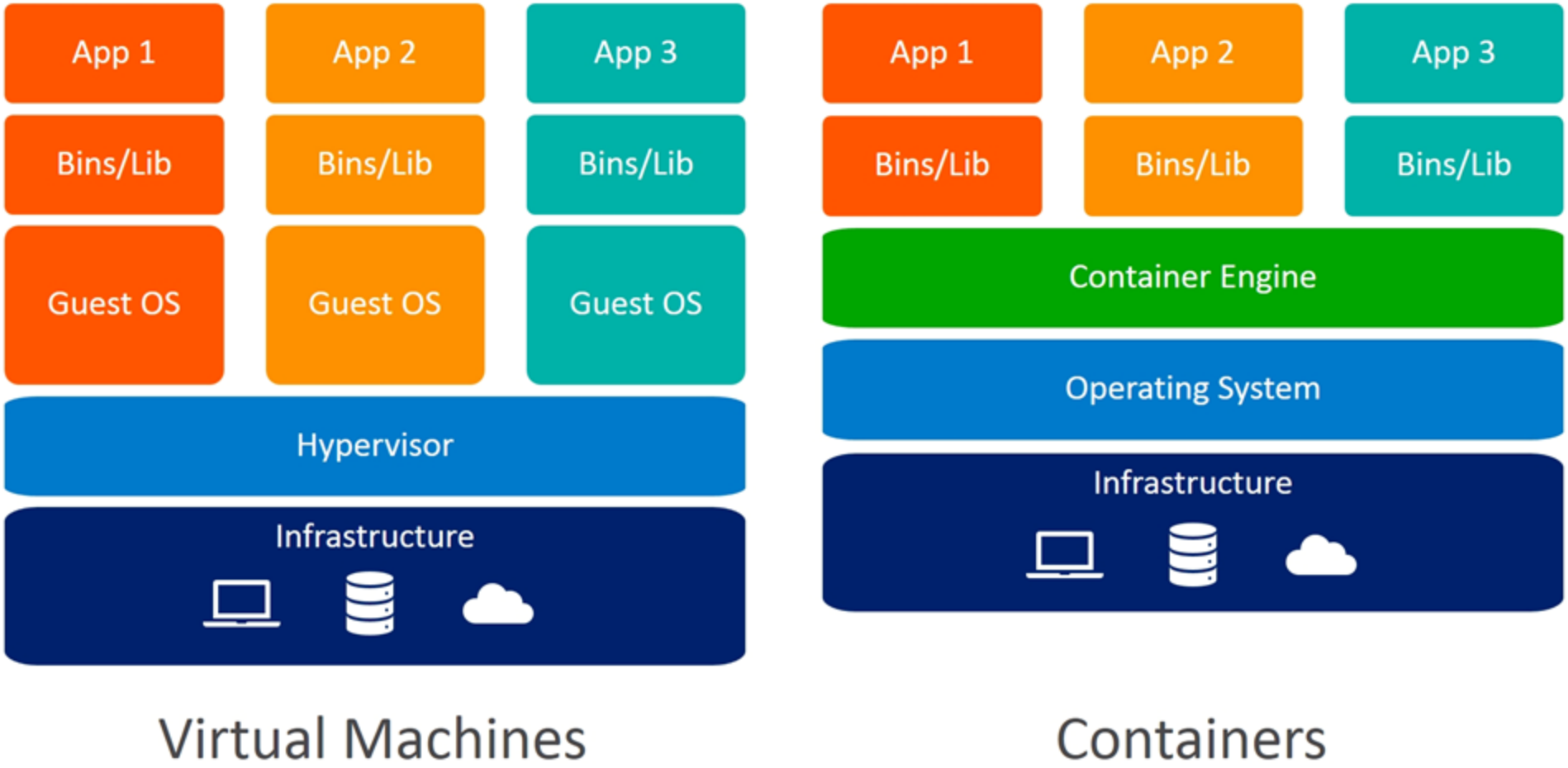
- **Pub/sub**: Event data is published to a message broker or event bus to be consumed by other applications, e.g. **Apache Kafka**.
- **Event streaming**: Process a continuous flow of data in something very close to real time, it is not persisted *at rest* in a database but processed as it is created or received by the streaming solution, e.g. **Apache Storm**.



Containers
Importance of **Environment** in ML Engineering

- Software deployment is a core aim of ML engineers
- Awareness of environmental requirements is crucial
- Different environments can impact code execution
- **Challenges with Python Deployment**
  - Python lacks built-in support for standalone executables
  - Requires a Python interpreter to run
  - Needs relevant libraries and packages installed
- **The Containerization Solution**
  - Containerization: Isolate applications and dependencies
  - Create standalone units for diverse computing platforms
  - Simplifies deployment and reduces compatibility issues

Containers


App 1
App 2
App 3
Bins/Lib
Bins/Lib
Bins/Lib
Guest OS
Guest OS
Guest OS
Hypervisor
Infrastructure
Virtual Machines
App 1
App 2
App 3
Bins/Lib
Bins/Lib
Bins/Lib
Container Engine
Operating System
Infrastructure
Containers

Containers

- **Containers:** Lightweight, share OS kernel, faster startup
- **VMs:** Heavier, run separate OS, longer startup time
- **Popular Tools:**
  - Docker
  - Kubernetes
  - OpenShift
  - Podman

Containers
Docker

- Most popular container technology
- Open-source and user-friendly
- Ideal for deploying Python applications

Containers Benefits

- Make it easy to build, modify, publish, search and run containers
  - A container has an application along with all its dependencies
  - Created though a Docker file
  - Subsequent changes to a baseline image can be committed and pushed t oa central registry
- Containers can be found in a Docker registry, using Docker search
- Containers can be pulled from a Docker registry
- Build, ship and run any app, anywhere
- Very suitable for DevOps cycles and Microservice Architectures

Containers and images: Terminology

- **Image:** Persisted snapshot that can be run
  - *images:* List all local images
  - *run:* create a container from an image
  - *tag:* Tag an image
  - *pull:* Download an image
  - *rmi:* delete a local image
- **Container:** Runnable instance of an image
  - *ps:* List all running containers
  - *start/stop:* Start/stop a container
  - *rm:* delete a container
  - *commit:* create an image from a contaienr

Containers and images: Terminology

- **Dockerfile**
  - To create images automatically using a build script
  - Can be versioned
- **Docker Hub**
  - Public repository of docker images

Example: Containerizing a Flask Application

- **Objective**:
  - Containerize a simple Flask RESTful web app
  - Application will serve as an interface to a forecasting model
  - Uses a skeleton app returning random numbers for forecasts

Check the repo out and see the Notebook

https://github.com/rpietrantuono/AISE_Ch5/