Unit-7

Advanced Topics in Software Design and Architecture

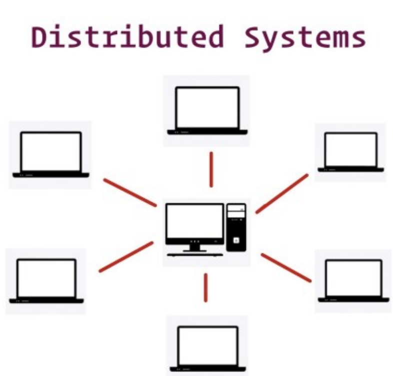
Overview

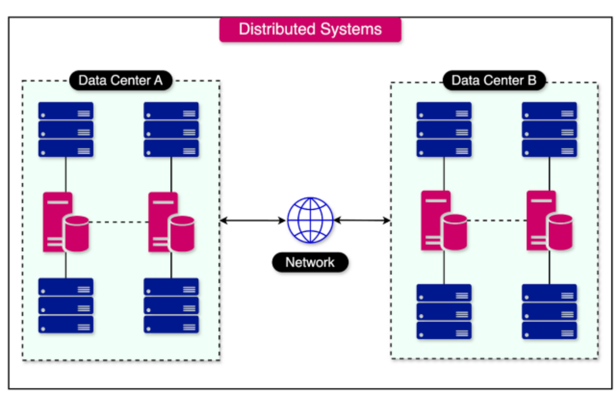
7.1 Designing for distributed systems and cloud-native architecture
7.2 Microservices and serverless architecture
7.3 Domain-Driven Design (DDD)
7.4 Designing for concurrency and parallelism
7.5 Fault tolerance and high availability

Designing for Distributed Systems and Cloud-Native Architecture

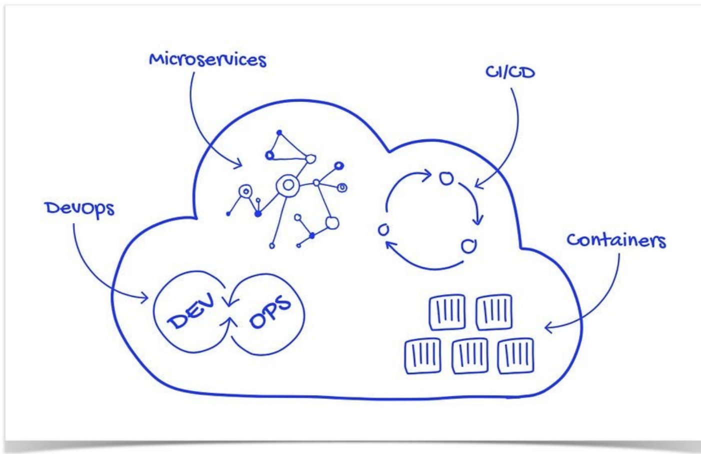
- **Distributed systems:** Decentralized components communicating over a network.
- **Cloud-native architecture:** Built for scalability, resilience, and agility.
- **Key principles:**
  - Loose coupling
  - Statelessness
  - API-first design
  - Automation (CI/CD, infrastructure as code).

Distributed System

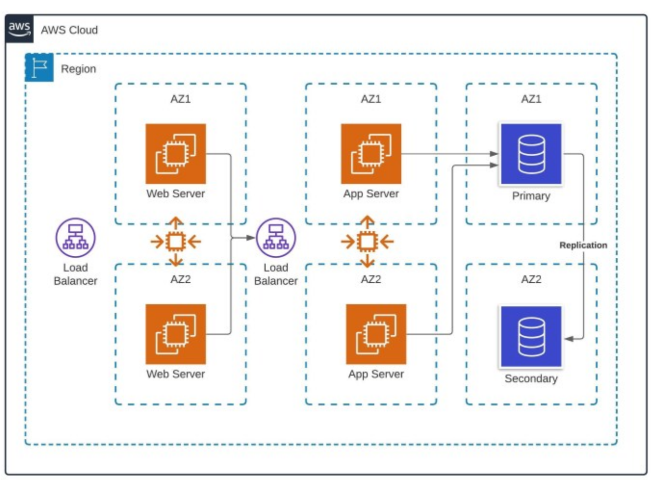




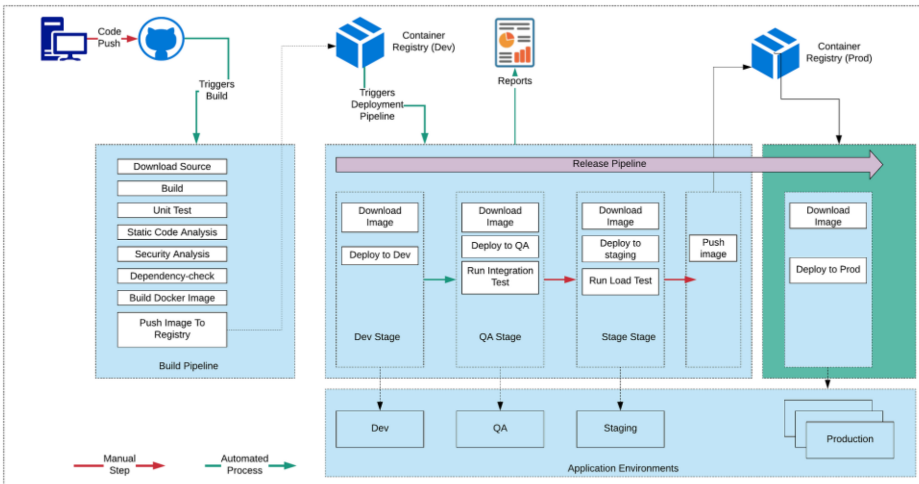
Cloud-Native Architecture



Cloud-Native Architecture



Cloud-Native Architecture- **Agile DevOps & Automation Using CI/CD**



Kubernetes

- **What is Kubernetes?**
  - Container orchestration platform
  - Automates deployment, scaling, and management of containerized applications
- **Why Kubernetes?**
  - Scalability, resilience, and portability
  - Cloud-agnostic infrastructure

Kubernetes Architecture Overview

- **High-Level Diagram**
  - Master Node (Control Plane)
  - Worker Nodes (Data Plane)
- **Key Components:**
  - API Server, Scheduler, Controller Manager, etcd, Kubelet, Kube Proxy

Master Node (Control Plane)

- **Role:** Manages the Kubernetes cluster
- **Components:**
  - **API Server:** Front-end for the control plane
  - **Scheduler:** Assigns workloads to worker nodes
  - **Controller Manager:** Ensures desired cluster state
  - **etcd:** Distributed key-value store for cluster data

Worker Nodes

- **Role:** Run the application workloads
- **Components:**
  - **Kubelet:** Communicates with the master node and manages containers
  - **Kube Proxy:** Maintains network rules for communication
  - **Container Runtime:** Runs containers (e.g., Docker, containerd)

Key Concepts in Kubernetes

- **Pods:** Smallest deployable units in Kubernetes
- **Services:** Enable communication between pods
- **Deployments:** Manage desired state for pods
- **Namespaces:** Logical partitions within a cluster

How Kubernetes Works

- **Step-by-Step Workflow:**
  - User submits a manifest file to the API Server.
  - Scheduler assigns pods to worker nodes.
  - Kubelet ensures containers are running as expected.
  - Controller Manager monitors and maintains the desired state.

Benefit and Challenges

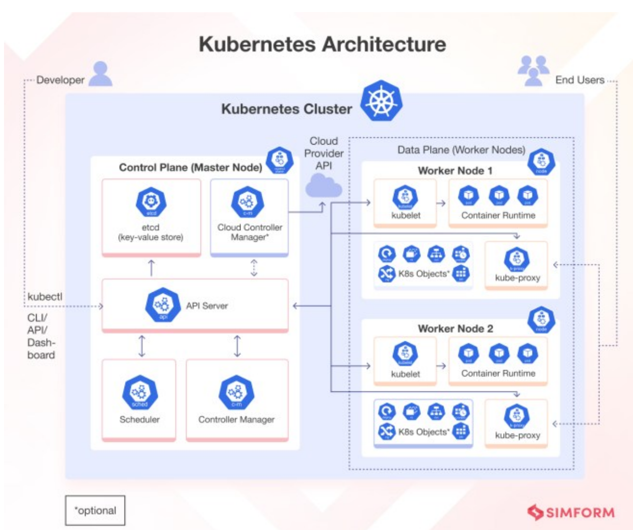
- **Benefits of Kubernetes Architecture**
  - **Scalability:** Automatically scale applications based on demand
  - **High Availability:** Self-healing and fault-tolerant
  - **Portability:** Run applications across multiple environments
  - **Extensibility:** Custom resources and plugins
- **Challenges in Kubernetes**
  - **Complexity:** Steep learning curve
  - **Resource Management:** Requires careful planning
  - **Security:** Ensuring secure configurations
  - **Monitoring and Logging:** Essential for troubleshooting

Use Cases of Kubernetes

- **Microservices Architecture:** Ideal for managing microservices
- **CI/CD Pipelines:** Streamlines deployment processes
- **Hybrid and Multi-Cloud Deployments:** Consistent operations across environments

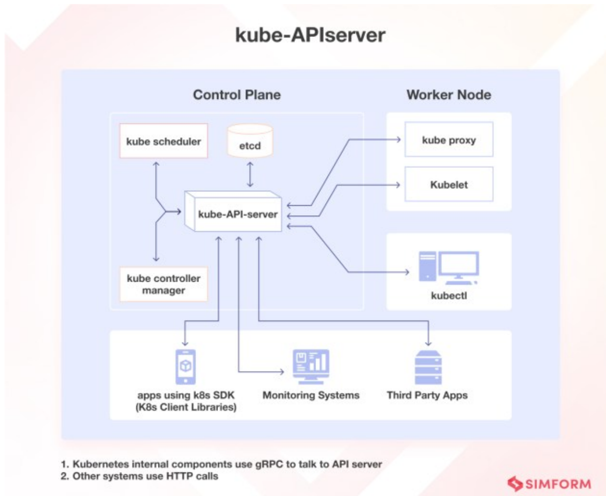
15

Kubernetes Architecture



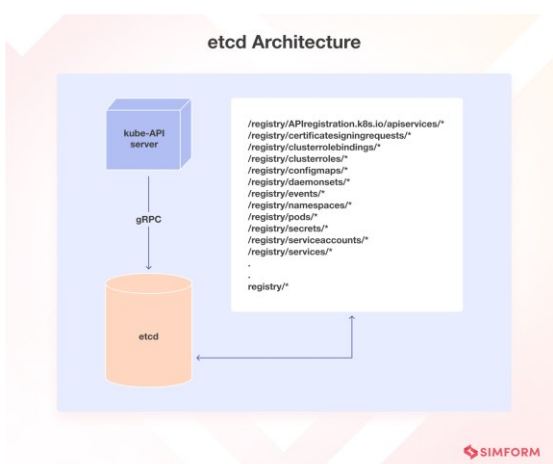
16

Kube-APIserver



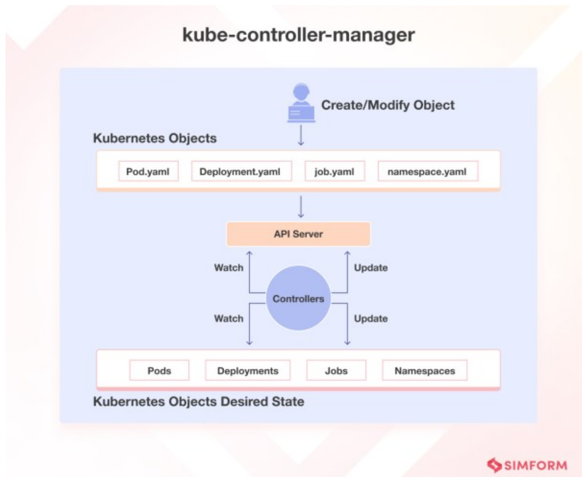
17

etcd



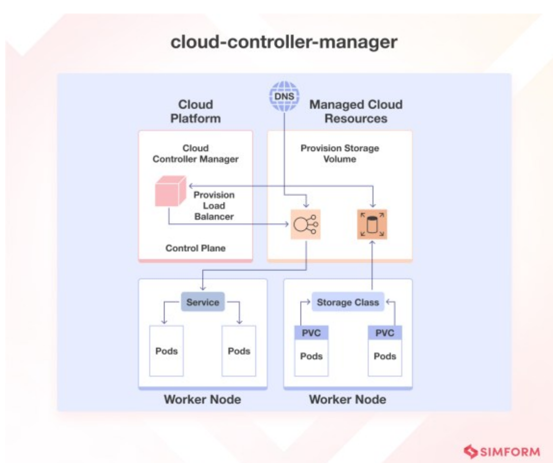
18

kube-controller-manager



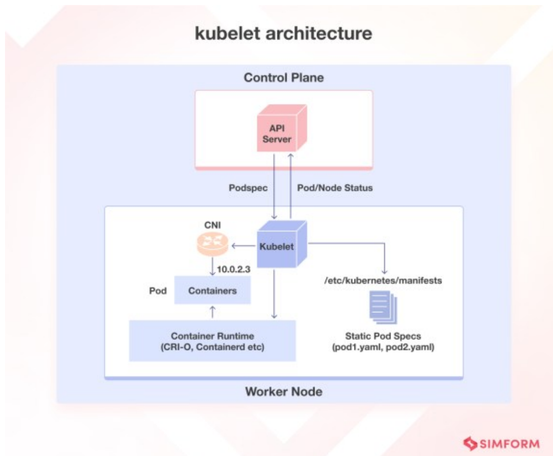
19

cloud-controller-manager (CCM)



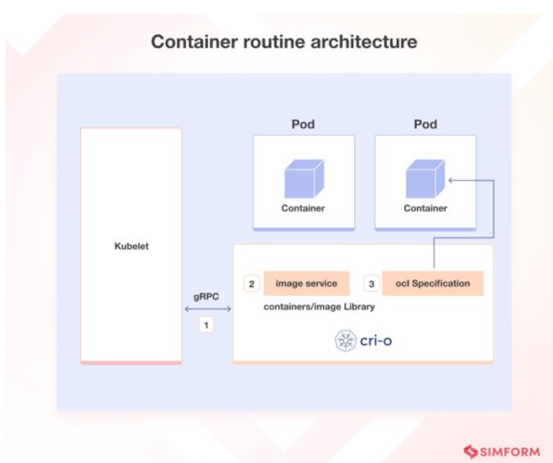
20

Kubelet | kube-proxy



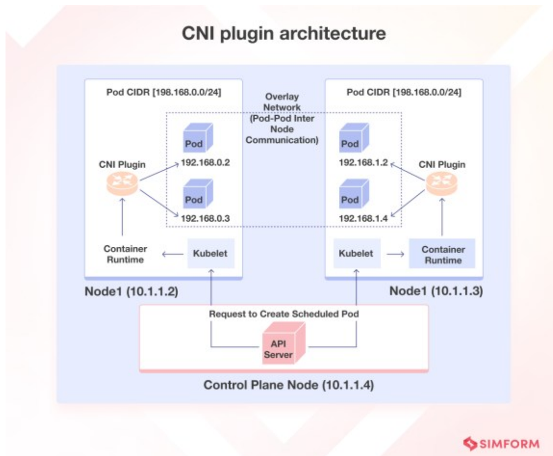
21

Container runtime



22

CNI Plugin



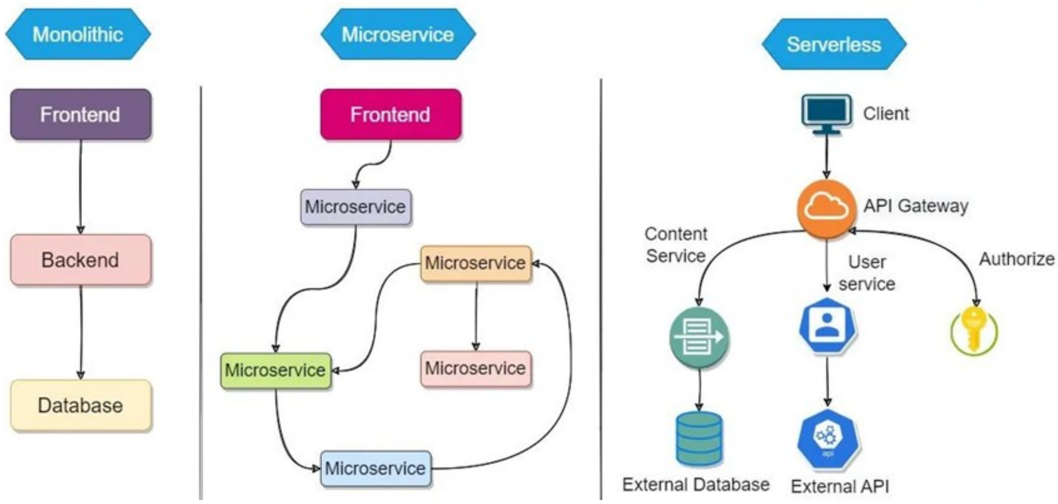
23

Microservices and Serverless Architecture

- **Microservices:**
  - Small, independent services with single responsibilities.
  - Benefits: Scalability, flexibility, and easier maintenance.
  - Challenges: Complexity in management and communication.
- **Serverless:**
  - Event-driven, auto-scaling, and pay-as-you-go.
  - Examples: AWS Lambda, Azure Functions.
  - Benefits: Reduced operational overhead, cost efficiency.

24

Monolith | Microservice | Serverless



25

Monolithic Architecture

A single, unified codebase where all components are tightly coupled.

Characteristics:

- Single deployment unit.
- Shared database and resources.
- Easier to develop and test initially.

Advantages:

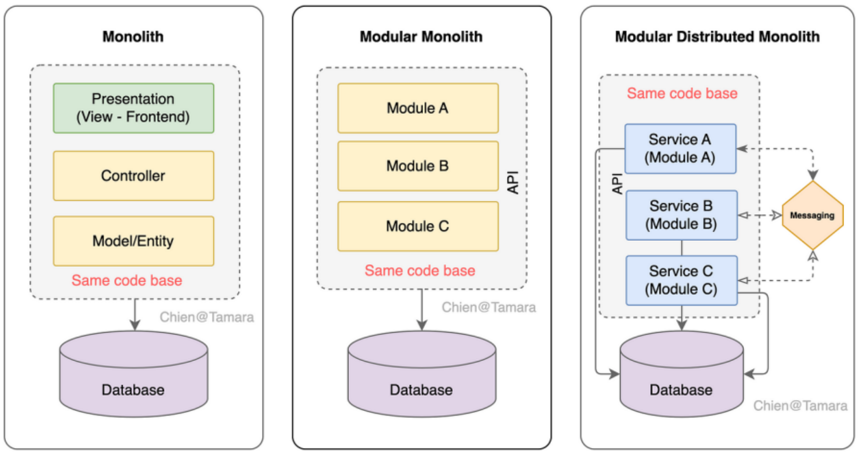
- Simplicity in development and deployment.
- Easier debugging and testing.

Disadvantages:

- Scalability challenges.
- Difficult to maintain as the codebase grows.
- Limited flexibility for technology stack changes.

26

Monolithic Architecture



27

Microservices Architecture

A collection of loosely coupled, independent services that communicate via APIs.

Characteristics:

- Each service has its own database and logic.
- Services can be developed, deployed, and scaled independently.

Advantages:

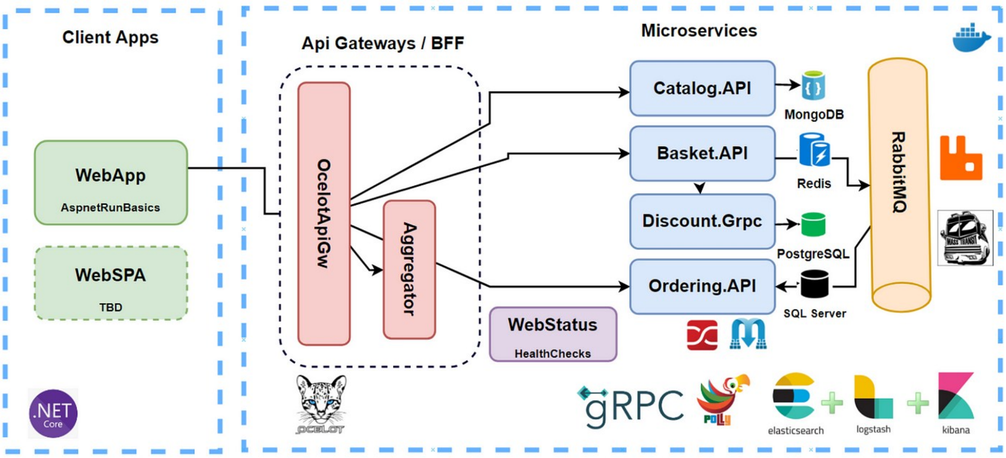
- Scalability and flexibility.
- Easier to adopt new technologies.
- Improved fault isolation.

Disadvantages:

- Increased complexity in development and deployment.
- Requires robust DevOps practices.
- Potential latency due to inter-service communication.

28

Microservices Architecture



29

Serverless Architecture

- A cloud-native model where developers build and run applications without managing servers.

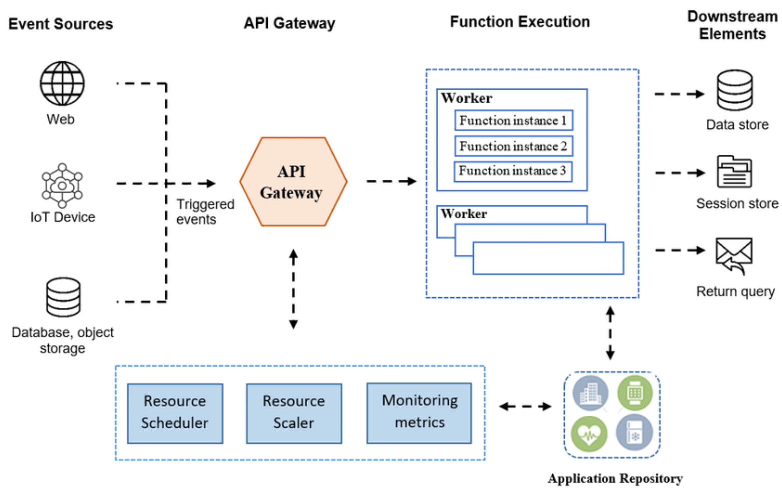
Characteristics:
- Event-driven execution (e.g., AWS Lambda, Azure Functions).
- Pay-as-you-go pricing model.
- Automatic scaling.

Advantages:
- No server management required.
- Cost-effective for sporadic workloads.
- High scalability.

Disadvantages:
- Limited control over the runtime environment.
- Cold start latency issues.
- Vendor lock-in risks.

30

Serverless Architecture



31

AWS Lambda Best Practices and Event-driven Architecture

Fig:1

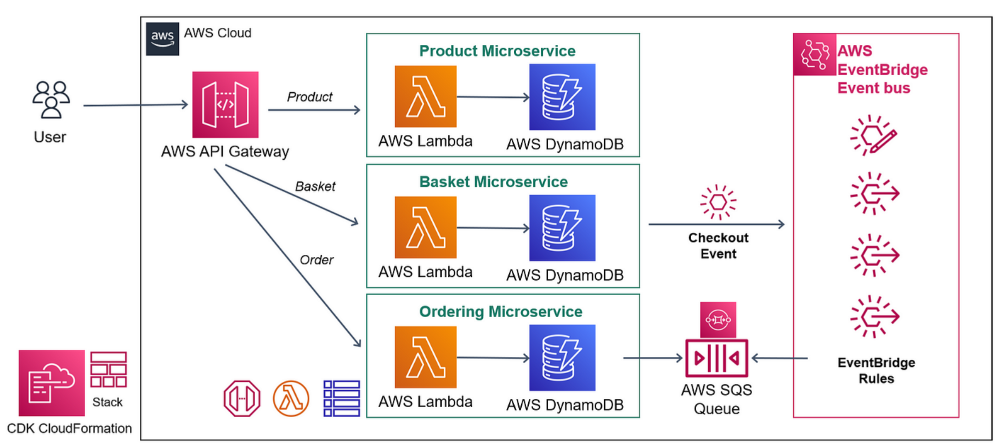
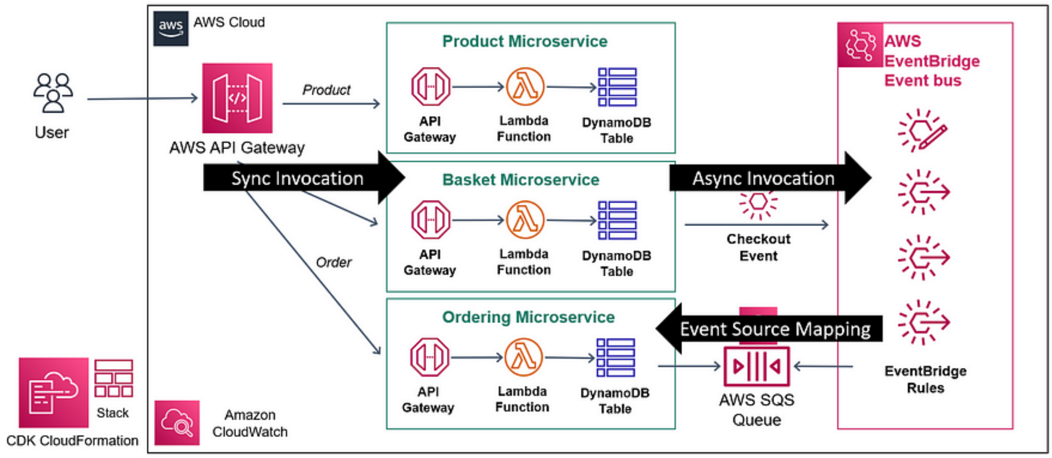


Fig:2

32

AWS Lambda Best Practices and Event-driven Architecture



33

Comparison

Aspect	Monolith	Microservices	Serverless
Complexity	Low	High	Medium
Scalability	Limited	High	Very High
Cost	Low (initially)	Medium	Pay-as-you-go
Maintenance	Hard (as it grows)	Easier (per service)	Managed by provider
Deployment Speed	Slow	Faster	Fastest

34

When to Use Each Architecture

- **Monolith:**
  - Small teams or startups.
  - Simple applications with limited scalability needs.
- **Microservices:**
  - Large, complex applications.
  - Teams with strong DevOps expertise.
- **Serverless:**
  - Event-driven or sporadic workloads.
  - Applications requiring rapid scaling.

35

Example

- **Monolith:**
  - Legacy systems like early versions of Shopify.
- **Microservices:**
  - Netflix, Amazon, and Uber.
- **Serverless:**
  - AWS Lambda for event-driven tasks (e.g., image processing).

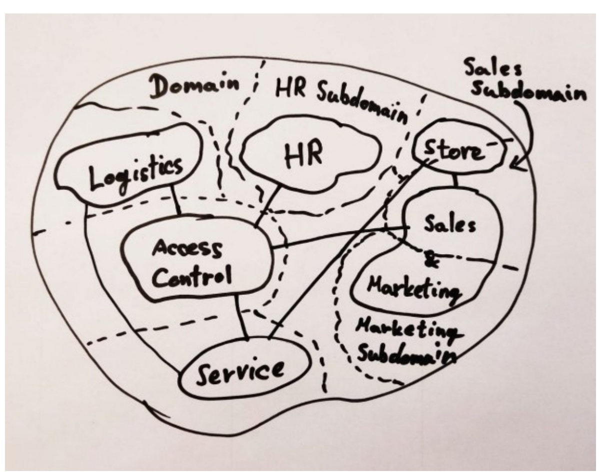
36

Domain-Driven Design (DDD)

- Focus on the core domain and domain logic.
- Key concepts:
  - ***Bounded Context***
  - ***Ubiquitous Language***
  - ***Entities, Value Objects, Aggregates.***
- Benefits: Aligns software design with business needs.

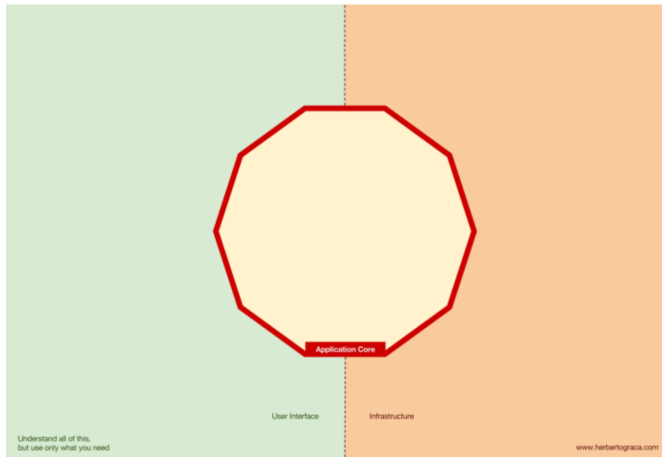
37

business domain with bounded contexts

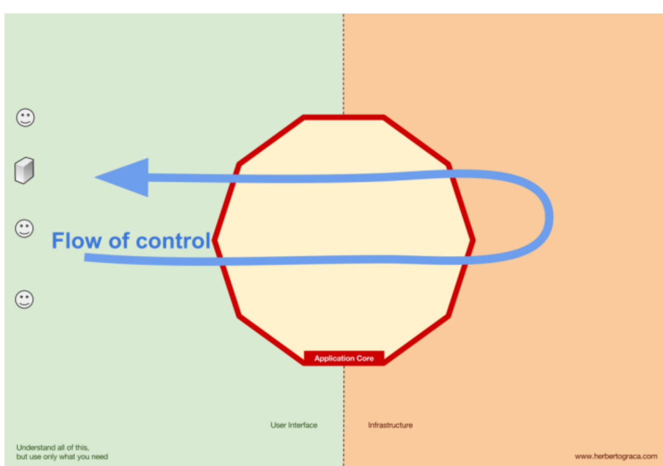


38

•What makes it possible to run a **user interface**, whatever type of user interface it might be;
•The system **business logic**, or **application core**, which is used by the user interface to actually make things happen;
•**Infrastructure** code, that connects our application core to tools like a database, a search engine or 3rd party APIs.



39

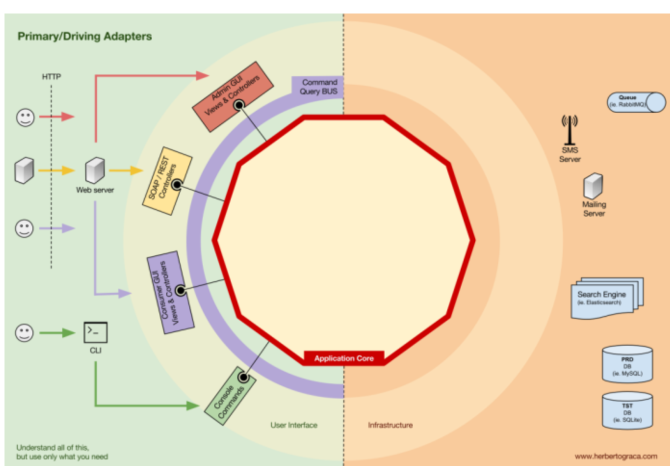


40

Tools

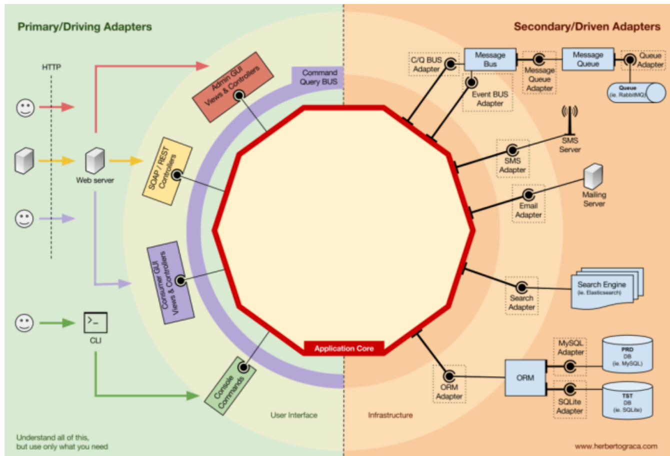
41

Primary or Driving Adapters



42

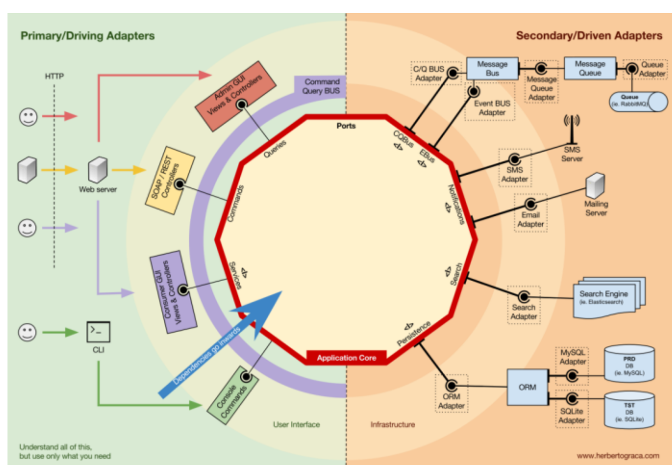
Secondary or Driven Adapters



43

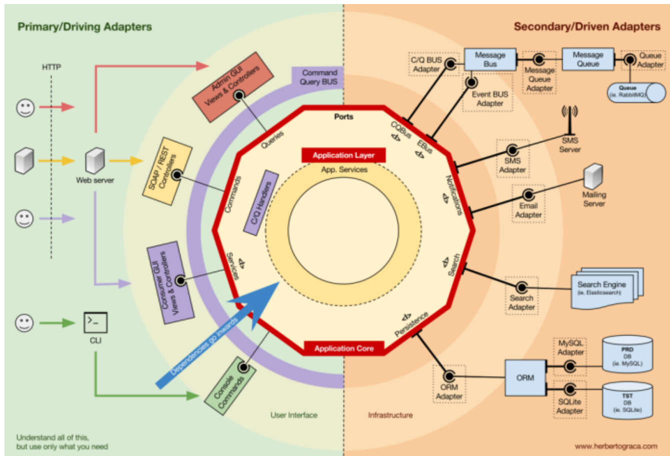
Inversion of control

A characteristic to note about this pattern is that the adapters depend on a specific tool and a specific port (by implementing an interface). But our business logic only depends on the port (interface), which is designed to fit the business logic needs, so it doesn't depend on a specific adapter or tool.

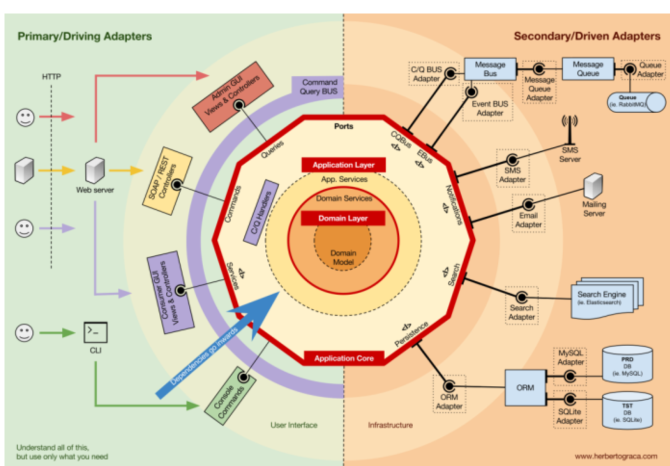


44

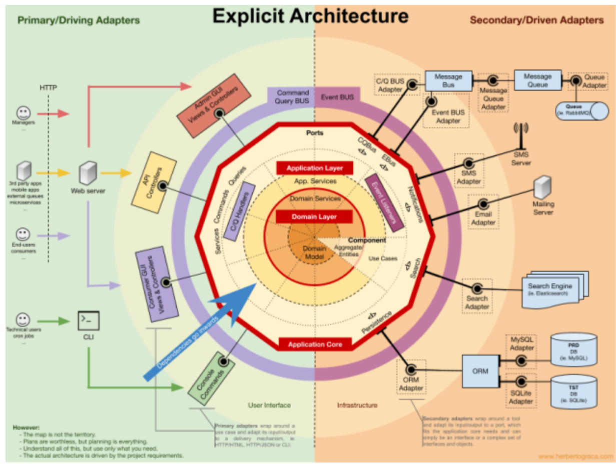
Application Layer



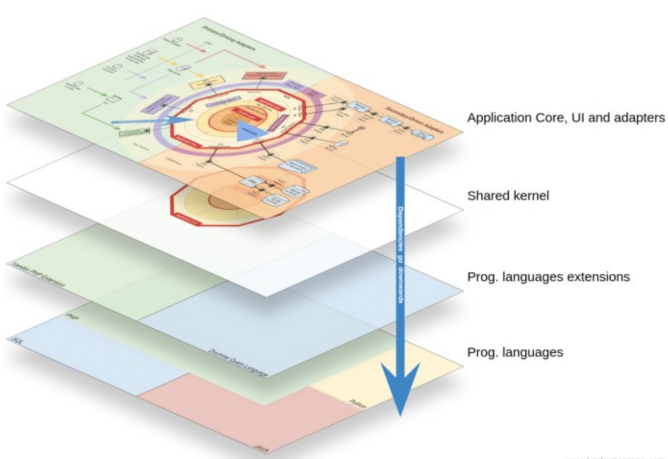
Domain Layer



Decoupling the components



Triggering logic in other components

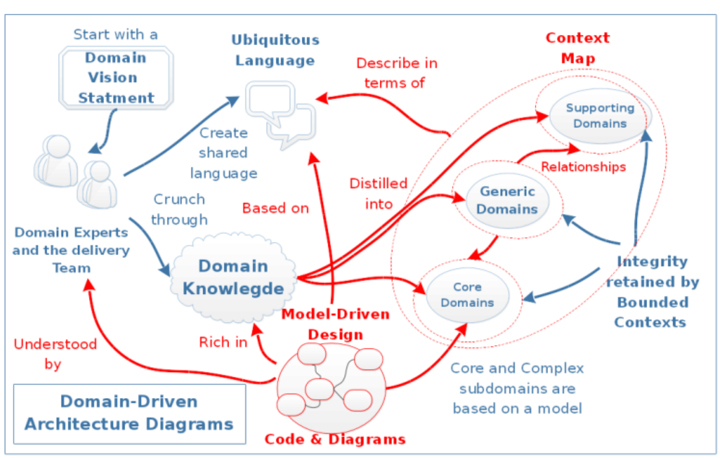


What Are Domain-Driven Architecture Diagrams?

- Diagrams that align software architecture with business domains.
- Focus on both **strategic** (high-level) and **tactical** (detailed) design.
- Inspired by Domain-Driven Design (DDD) principles.

49

Bounded Context Map



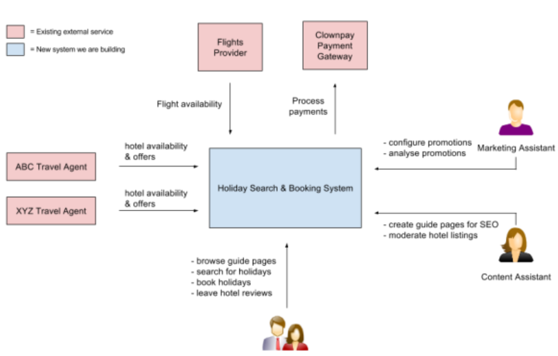
50

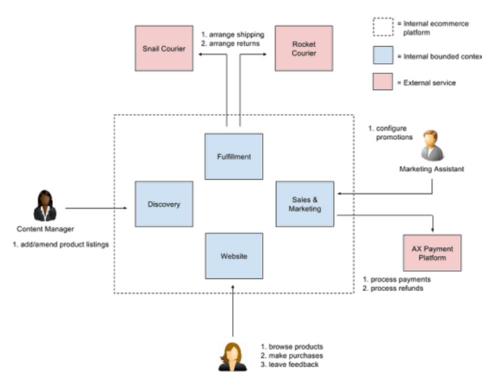
Why Use Domain-Driven Architecture Diagrams?

Why Are They Important?

- **Clarity:** Helps teams understand complex systems.
- **Alignment:** Ensures software design matches business needs.
- **Communication:** Bridges the gap between technical and non-technical stakeholders.
- **Decision-Making:** Guides architectural and design choices.

System Context Diagram | Bounded Context Map





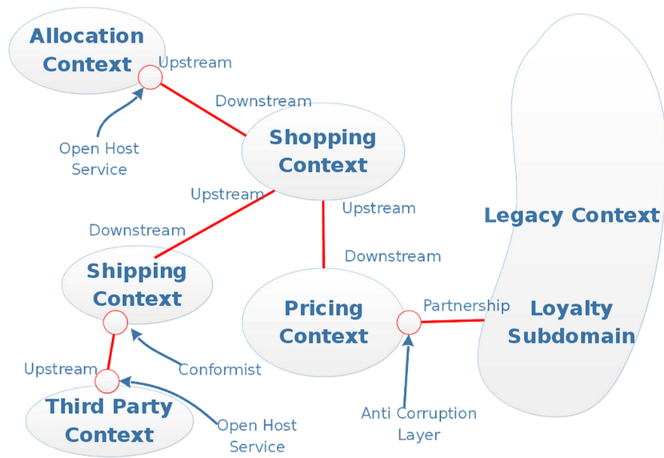
Key Concepts in Domain-Driven Design (DDD)

- Foundational DDD Concepts
  - **Bounded Contexts:** Clear boundaries around specific domains.
  - **Ubiquitous Language:** Shared vocabulary between developers and domain experts.
  - **Strategic Design:** High-level patterns like Context Maps.
  - **Tactical Design:** Detailed patterns like Entities, Value Objects, and Aggregates.

Types of Domain-Driven Architecture Diagrams

- Diagram Types
  - **Context Maps:** Show relationships between bounded contexts.
  - **Domain Diagrams:** Focus on core domains and subdomains.
  - **Component Diagrams:** Detail internal components within a bounded context.
  - **Event Storming Diagrams:** Visualize workflows and domain events.

Bounded Context Map



Context Maps

- Context Maps
  - Show how different bounded contexts interact.
  - Common patterns: **Partnership**, **Customer-Supplier**, **Conformist**, **Anti-Corruption Layer**.
  - Helps identify integration challenges.

Domain Diagrams

- Focus on **core domains** (most critical to the business).
- Identify **supporting** and **generic subdomains**.
- Helps prioritize development efforts.

Component Diagrams

- Detail the internal structure of a bounded context.
- Show relationships between entities, value objects, and aggregates.
- Useful for tactical design and implementation.

Event Storming Diagrams

- Visualize workflows and domain events.
- Collaborative process involving domain experts and developers.
- Helps identify key events, commands, and policies.

Best Practices

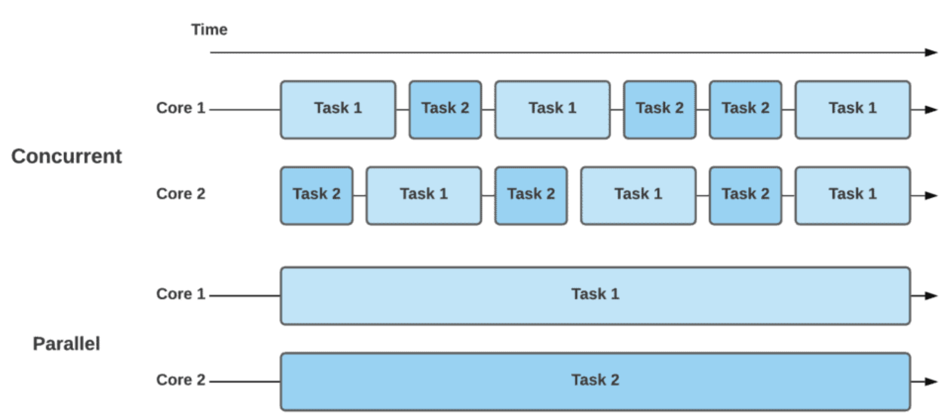
- Best Practices for Creating Domain-Driven Diagrams
  - Collaborate with domain experts and stakeholders.
  - Use a consistent **ubiquitous language**.
  - Iterate and refine diagrams as the system evolves.
  - Focus on both strategic and tactical levels.
  - Keep diagrams simple and understandable.

Designing for Concurrency and Parallelism

- **Concurrency:** Managing multiple tasks at once (e.g., threads, async programming).
- **Parallelism:** Executing multiple tasks simultaneously (e.g., multi-core processing).
- Techniques:
  - Thread pools, async/await, message queues.
  - Distributed computing frameworks (e.g., Apache Kafka, Spark).
- Challenges: Race conditions, deadlocks, and resource contention.

61

Concurrent | Parallel



62

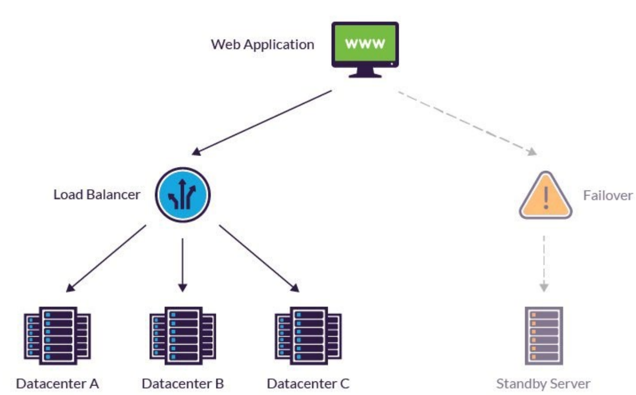
Fault Tolerance and High Availability

- **Fault Tolerance:** System's ability to continue operating despite failures.
  - Techniques: Redundancy, retries, circuit breakers.
- **High Availability:** Ensuring minimal downtime.
  - Techniques: Load balancing, failover, replication.
- **Tools**: Kubernetes, Istio, Hystrix.

63

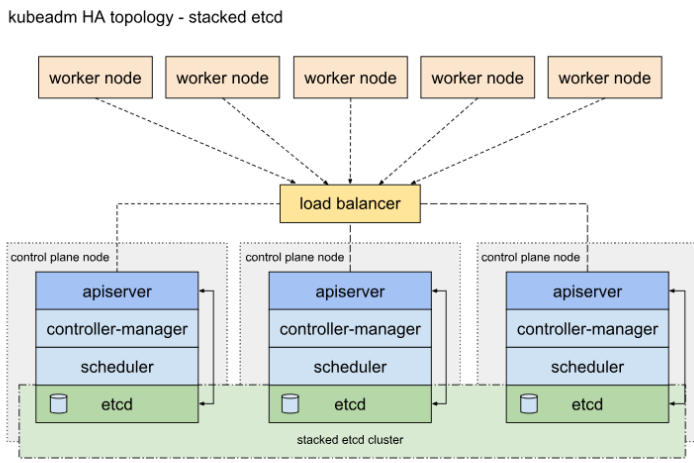
Fault Tolerant System

In the context of web application delivery, fault tolerance relates to the use of load balancing and failover solutions to ensure availability via redundancy and rapid disaster recovery.



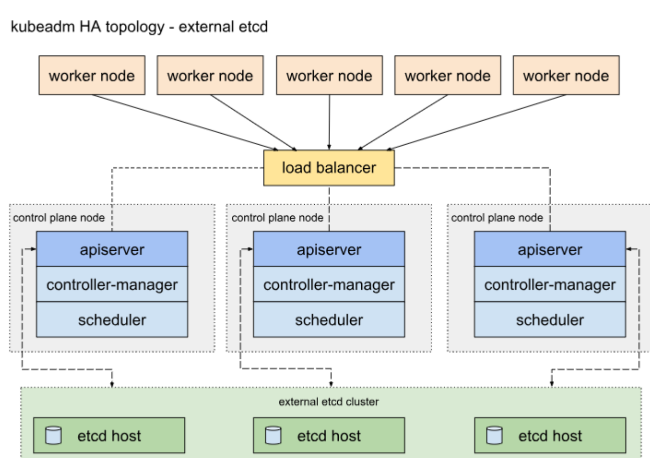
64

Stacked etcd



65

External etcd



66