WSO2CON 2024

Architecting Cloud Native Applications



Chintana Wilamuna
Senior Director - Solutions Architecture
WSO2 Inc



Dushan Abeyruwan
Director- Solutions Architecture
WSO2 Inc



Architecture patterns

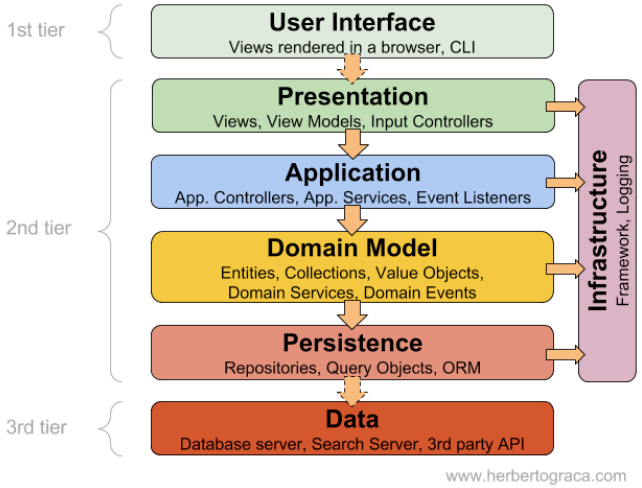
- Background: Architecture patterns
- Evolution of microservices
- Fundamentals of developing and managing microservices
- Cloud native apps
- Analyzing architecture trade-offs
- Focusing on adding business value
- Practical example

Background: Architecture Patterns

Architecture patterns

- Monolithic
  - Layered
  - Pipes and filters
  - Microkernel
- Distributed
  - Service Oriented (SOA)
  - Event-driven
  - Space based architecture
  - Microservices

Architecture patterns > Monolithic > Layered



Pros
Easy to get started for simple projects
Separation of concerns
Lower complexity and cost at the beginning
Cons
Harder to scale
Deployments and modifications are complex and expensive
HA and fault tolerance is harder

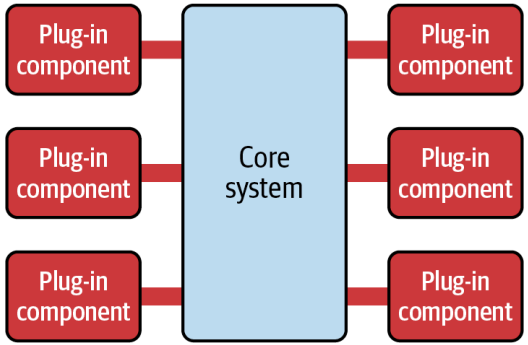
Image - https://herbertograca.com/2017/08/03/layered-architecture/

Architecture patterns > Monolithic > Pipes and filters



Pros
Easy and simple to get started
Comparatively higher modularity
Easy to reuse components
Cons
Harder to scale
Deployments and modifications are complex and expensive
HA and fault tolerance is harder

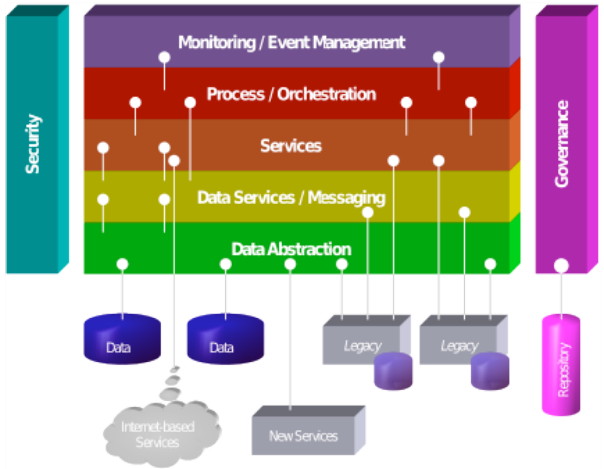
Architecture patterns > Monolithic > Microkernel



Pros
Flexibility in adding dynamic functionality through plugins
Plugins can have their own DB / schema (decoupled data architecture)
Reduced scope in testing, can be isolated to each plugin
Cons
The core is still a monolith, harder to scale
Deployments and modifications are complex and expensive
HA and fault tolerance is harder

Image - https://www.oreilly.com/library/view/software-architecture-patterns/9781098134280/ch04.html

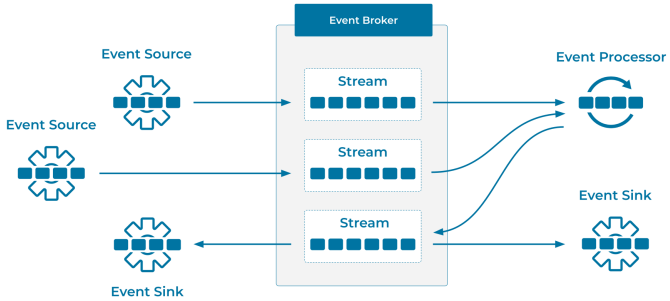
Architecture patterns > Distributed > SOA



Pros
One of the easiest and simplest distributed architecture models
Functionality is isolated and can be dev., test, deploy separately
Easy to scale
Cons
Typically a lot more complexity due to standards
A lot more error conditions due to services over network hops
Interdependent services can be hard to debug

Image - https://en.wikipedia.org/wiki/Service-oriented_architecture

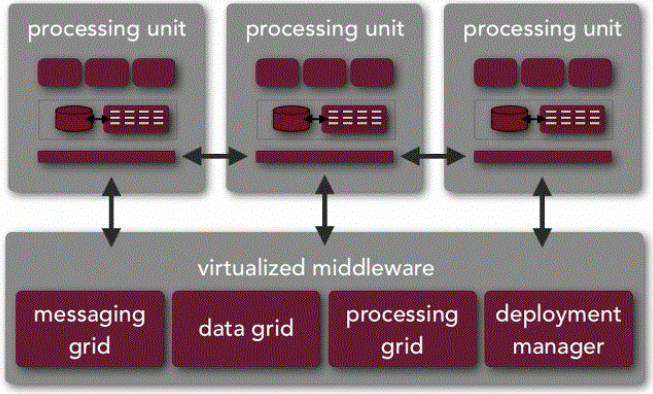
Architecture patterns > Distributed > Event-driven



Pros
Asynchronous comms give higher perf. and overall responsiveness
Higher flexibility and extensibility
Higher reliability - guaranteed delivery patterns
Cons
Harder to test
HA and scalability of brokers are not simple
Handling dynamic events is complex

Image - https://developer.confluent.io/patterns/event-stream/event-broker/

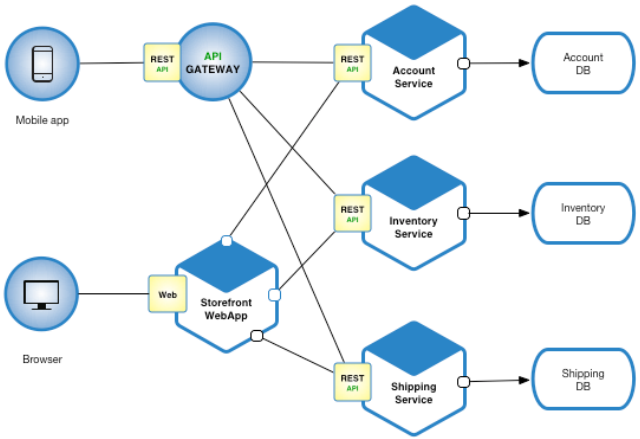
Architecture patterns > Distributed > Spaced based architecture



Pros
Can scale to handle high performance
More responsive - Reads/Writes driven off of a cache
Decoupled components
Cons
Complex and difficult testing process
Complicated architecture
Higher costs to maintain a system with this high complexity

Image - https://en.wikipedia.org/wiki/Space-based_architecture

Architecture patterns > Distributed > Microservices

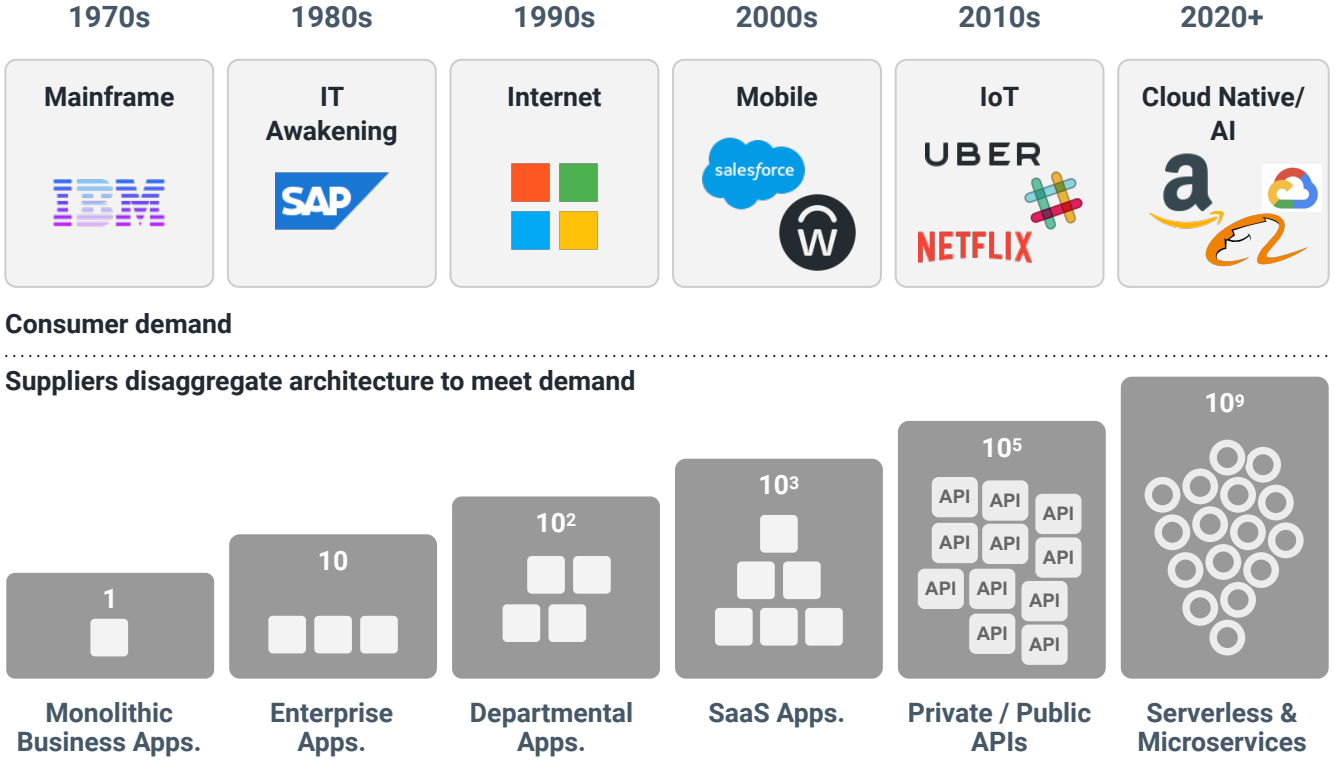


Pros
Often with bounded contexts, clear separation of concerns
Data isolation
Easy to scale
Cons
Testing and debugging is harder
Integration points between services can become harder to manage
Increased cost and communication overhead

Image – https://microservices.io/patterns/microservices.html

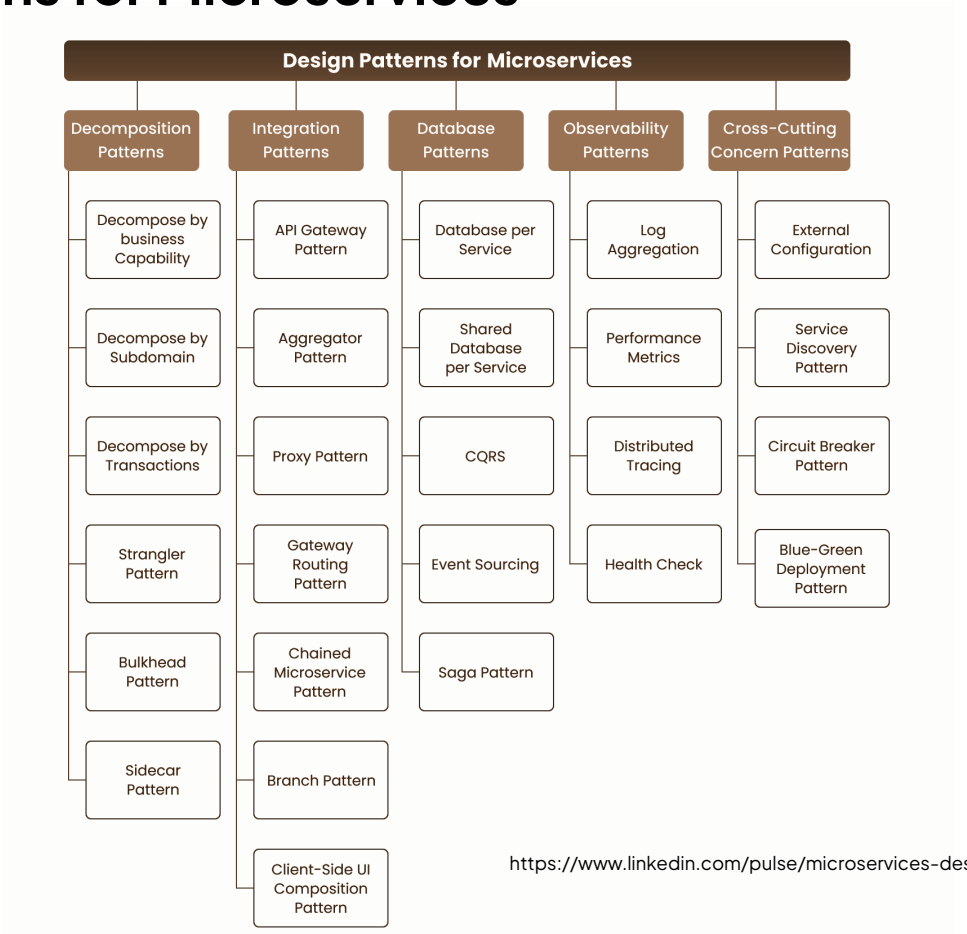
Evolution from monolith to microservices

Development has exponentially grown more complicated



Fundamentals of developing and managing microservices

Design Patterns for Microservices



https://www.linkedin.com/pulse/microservices-design-patterns-pranav-kumar-dwivedi/

Challenges in managing microservices

- Packaging microservices - Containers
- Testing - A/B, Blue-Green, Canary
- Version management
- Configuration management
- Scaling
- Discovering services and endpoints
- Monitoring

Cloud Native era


“The cloud native era is transforming how we build and deliver technology, creating more dynamic and immersive digital experiences than ever before.”

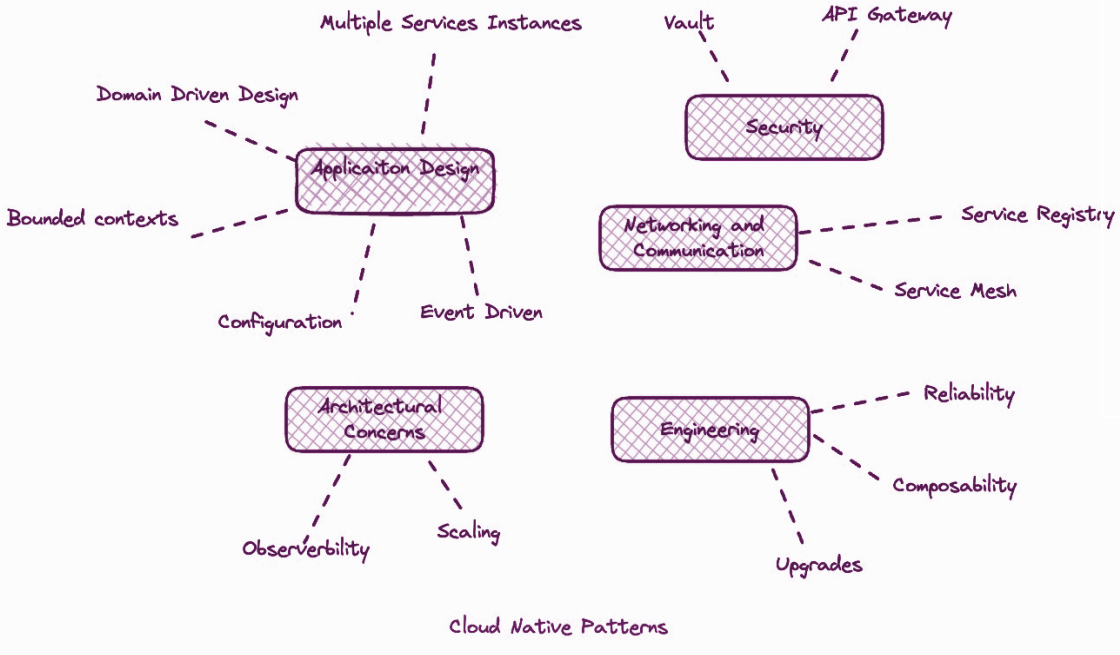
What is Cloud Native?

"Cloud-native architecture and technologies are an approach to designing, constructing, and operating workloads that are built in the cloud and take full advantage of the cloud computing model."

Source: https://learn.microsoft.com/en-us/dotnet/architecture/cloud-native/definition

https://wso2.com/choreo/resources/building-a-cloud-native-twelve-factor-app-on-choreo-just-write-the-code/

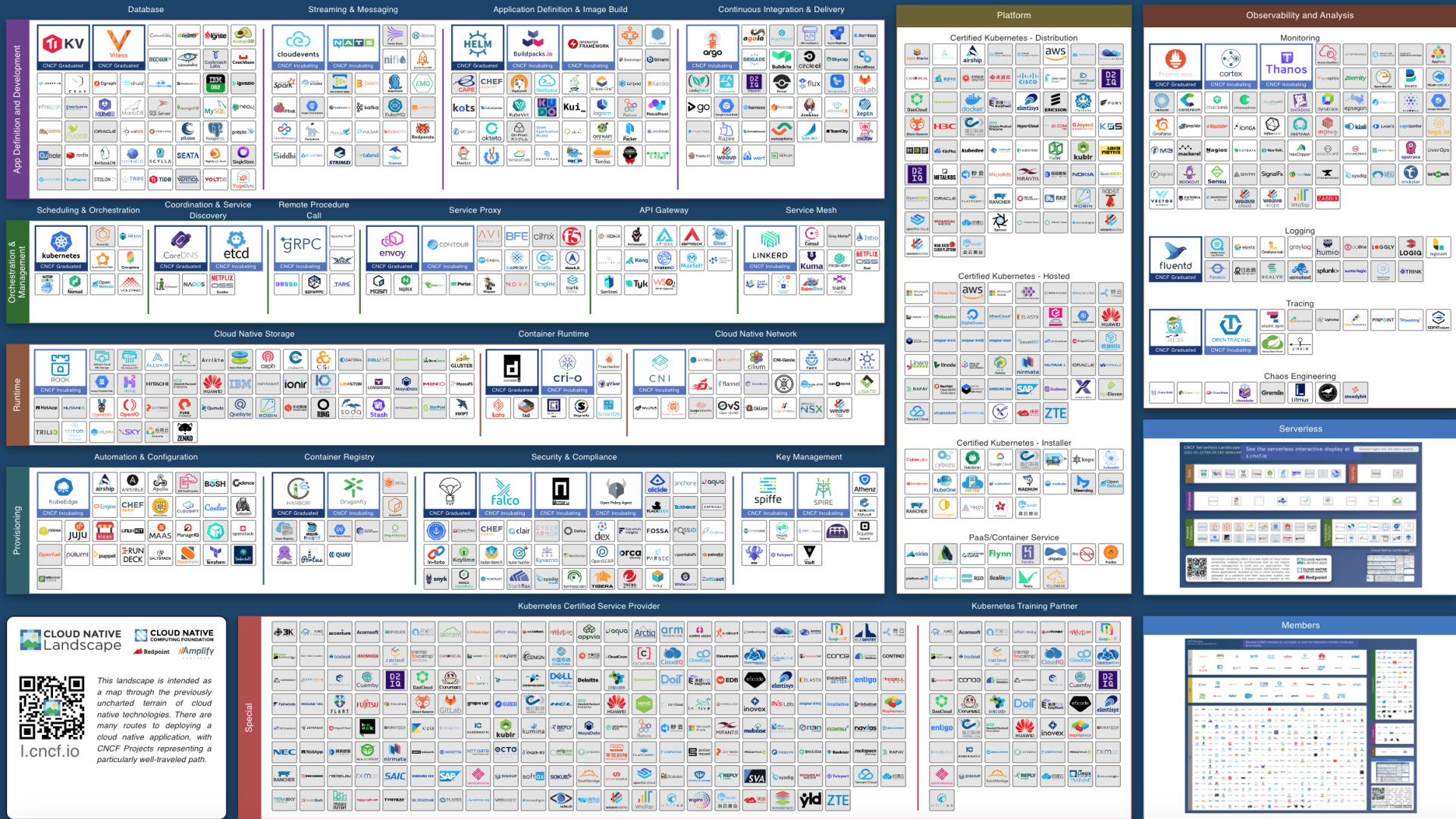
Cloud Native Application Design



Cloud-native architecture harnesses cloud benefits through key practices:

- **Microservices:** Splits apps into small services for flexible scaling and maintenance.
- **Containers/Orchestration:** Ensures consistent environments and automates operations with tools like Kubernetes.
- **Immutable Infrastructure:** Uses replaceable components for reliability, avoiding direct changes.
- **Declarative APIs:** Defines desired states, ensuring consistent application behavior.
- **Stateless Design:** Improves scalability and fault tolerance; stores state externally for better load management.

Above practices enable ***resilient, scalable, and manageable*** cloud applications, optimizing deployment and operations.


App Definition and Development
Database
Streaming & Messaging
Application Definition & Image Build
Continuous Integration & Delivery
Orchestration & Management
Scheduling & Orchestration
Coordination & Service Discovery
Remote Procedure Call
Service Proxy
API Gateway
Service Mesh
Runtime
Cloud Native Storage
Container Runtime
Cloud Native Network
Provisioning
Automation & Configuration
Container Registry
Security & Compliance
Key Management
Platform
Certified Kubernetes - Distribution
Certified Kubernetes - Hosted
Certified Kubernetes - Installer
PaaS/Container Service
Observability and Analysis
Monitoring
Logging
Tracing
Chaos Engineering
Serverless
Kubernetes Certified Service Provider
Special
Kubernetes Training Partner
Members
CLOUD NATIVE Landscape
CLOUD NATIVE COMPUTING FOUNDATION
Redpoint
Amplify
This landscape is intended as a map through the previously uncharted terrain of cloud native technologies. There are many routes to deploying a cloud native application, with CNCF Projects representing a particularly well-traveled path.
l.cncf.io

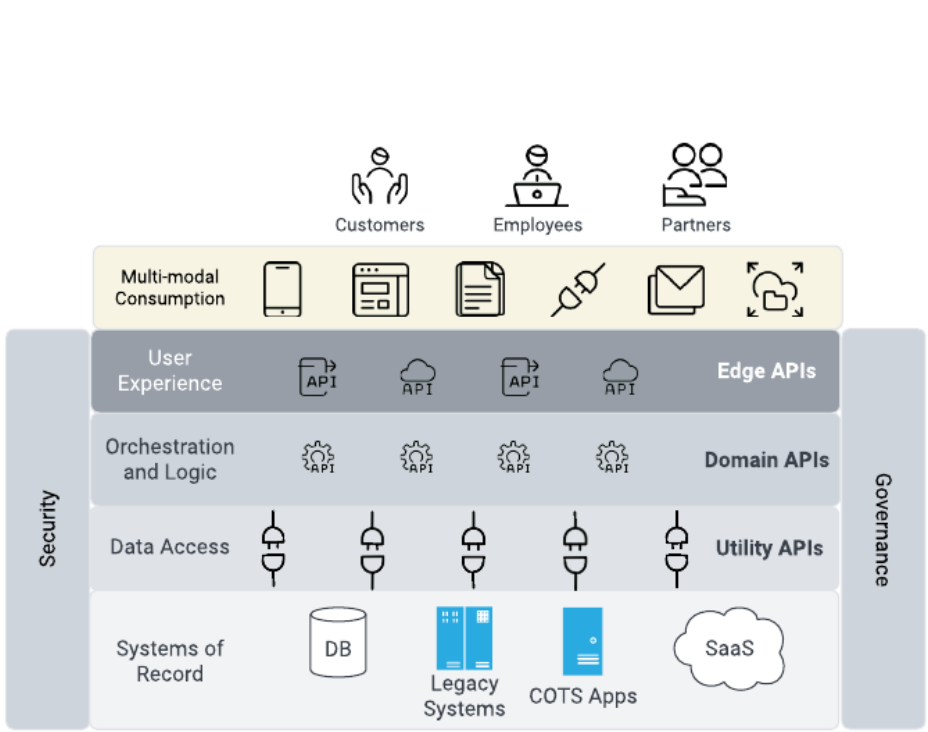
What is a cloud native application and what are the benefits?

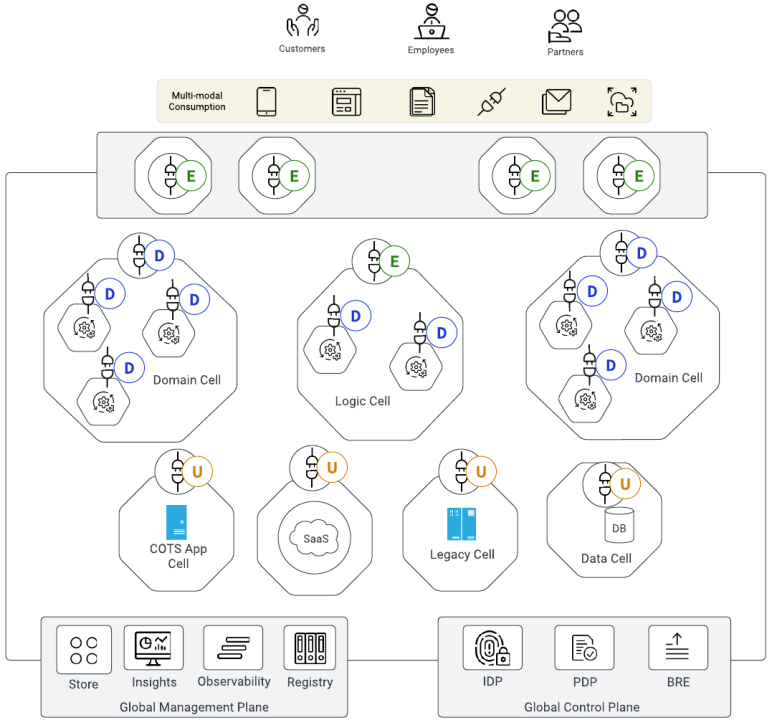
- Cloud native apps are specifically designed and built to run on cloud computing platforms by using technologies that are optimized for the cloud environment.
- These apps gain benefits of the cloud including
  - Scalability: Ability to scale up and down as needed
  - Resiliency: Can continue to function event if individual components fail
  - Flexibility: Can be deployed in any environment and easily integrated with other services and apps
  - Faster time-to-market: Modular, containerized architecture makes it easier to develop, test, and deploy new features.
  - Cost-effective: You only have to pay for resources that you use

Cloud native app journey

- The business leader aims to deliver awesome digital experiences rapidly.
- **Architects and app developers should adopt generic patterns to their use case and unique constraints**
- Application developers must understand underlying infrastructure concerns
  - DevOps
  - GitOps
  - CI/CD
  - API management
  - Analytics & Insights
  - Observability and Monitoring
  - Security

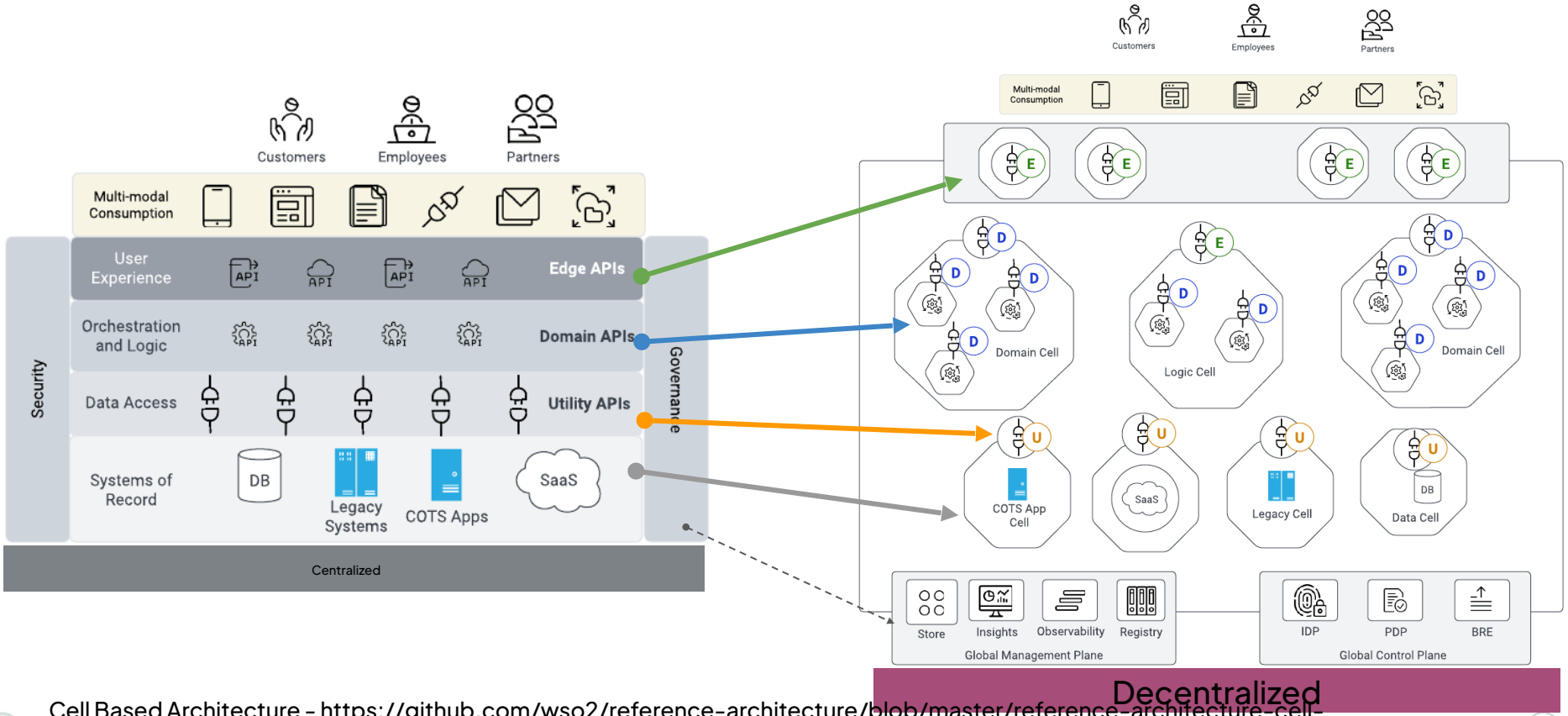
Typical brown field app transformation





Centralized | Decentralized

Typical brown field app transformation



Cell Based Architecture - https://github.com/wso2/reference-architecture/blob/master/reference-architecture-cell-based.md

A typical path of a cloud native application

- They face several challenges along the way:
  - **Challenge 1:** Technology complexity
  - **Challenge 2:** Hard to find technical resources
  - **Challenge 3:** Slow to go-to-market
- The organization must now make a decision
  - Continue to build a platform on their own
  - Buy a ready made platform with all the tools they need

Analyzing architecture trade-offs

Application Architecture – Analyzing trade-offs

- Use case - Hotel Reservation System
  - Room reservation
  - Room search
  - Cancel reservation

Analyzing trade-offs > Room reservation

Hotel Reservation System Architecture

CU What's the best architecture pattern to use for reserving a room for a hotel reservation system?

When designing a hotel reservation system, a common and effective architecture pattern to use is the Domain-Driven Design (DDD) pattern, along with a layered architecture approach. Here's a high-level overview of how you could structure your system using these patterns:

1. **Domain Layer**: This layer represents the core business logic of your system. It should encapsulate the domain concepts, such as `Hotel`, `Room`, `Reservation`, and the rules that govern them. This layer should be independent of any external concerns, such as data persistence or user interfaces.
2. **Application Layer**: This layer coordinates the application's activity and acts as a bridge between the domain layer and the infrastructure layer. It contains services that orchestrate the domain objects to perform specific use cases, such as "Reserve a Room" or "Cancel a Reservation".
3. **Infrastructure Layer**: This layer handles technical concerns, such as data persistence (e.g., database access), external system integrations (e.g., payment gateways), and other cross-cutting concerns like logging or caching.
4. **Presentation Layer**: This layer handles user interfaces, such as a web application or a mobile app, and communicates with the application layer to perform user actions.

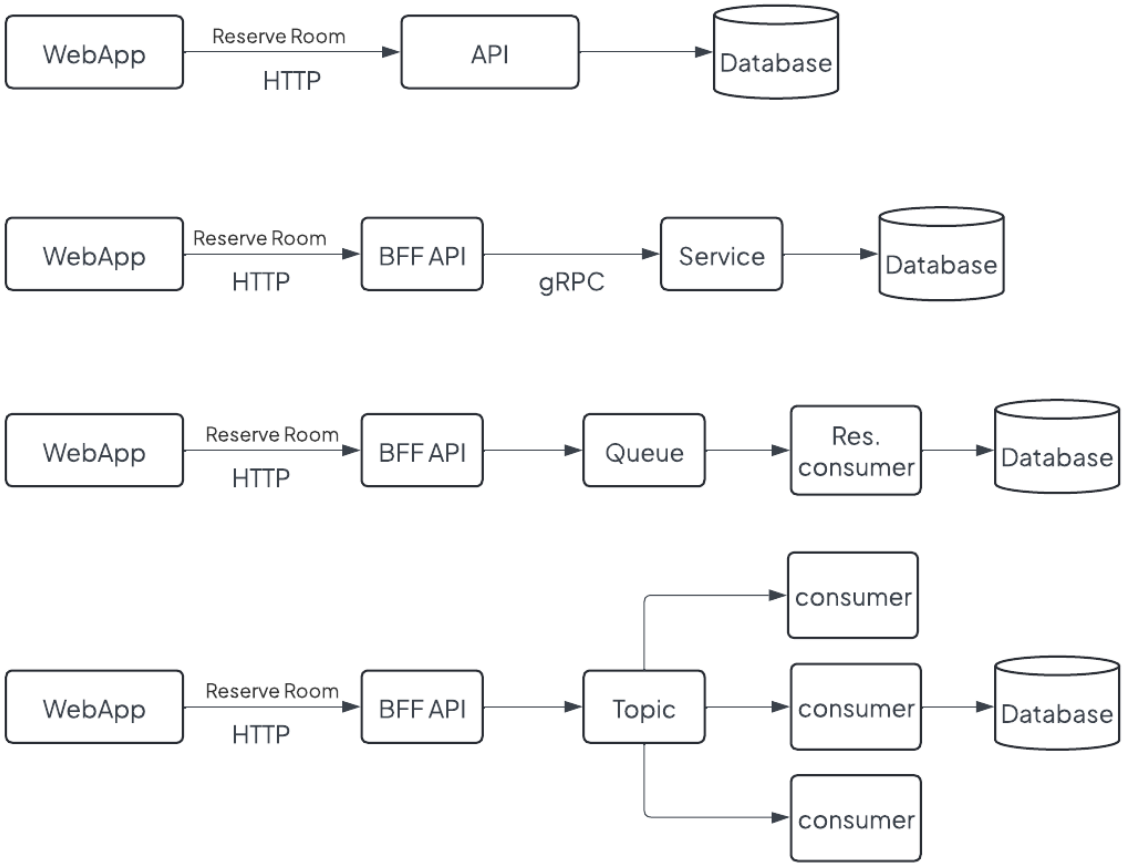
You
What's the best architecture pattern to use for reserving a room for a hotel reservation system?

ChatGPT
The best architecture pattern for reserving a room in a hotel reservation system often depends on various factors such as scalability, flexibility, and the specific requirements of the system. However, a commonly used pattern for this scenario is the "Repository Pattern" along with the "Unit of Work" pattern, often implemented within a layered architecture such as the "Clean Architecture" or "Hexagonal Architecture".

Here's a breakdown of how these patterns could be applied:

1. **Repository Pattern**: This pattern abstracts the data access layer, providing a clean separation between the domain logic and the data storage mechanism. You would typically have a repository interface defining methods like `` `reserveRoom()` ``, `` `cancelReservation()` ``, `` `getRoomAvailability()` ``, etc. Multiple implementations (e.g., for different databases) can exist behind these interfaces without affecting the rest of the system.
2. **Unit of Work Pattern**: This pattern helps in managing transactions and ensuring consistency across multiple repository operations. It's particularly useful when you need to coordinate multiple data operations within a single transaction, such as reserving a room and updating inventory

Analyzing trade-offs > Room reservation



Trade off analysis at a granular level

- REST vs GraphQL vs gRPC
- Choosing data storage options
  - MySQL, MongoDB, Cassandra
  - GCP - Bigtable, Datastore, Firestore, Spanner, AlloyDB
- Federated GraphQL API for all services of my enterprise

Still a lot of things left to think about as a architect/developer

- Endpoint management
- Secret management
- Endpoint/API credentials / authentication
- User authentication and authorization
- ...

Tooling make life easy … or does it?



frontend.yaml

```yaml
apiVersion: ap
kind: Deployme
metadata:
  labels:
    run: fronte
  name: fronte
spec:
  selector:
    matchLabel
      app: fro
  replicas: 2
  template:
    metadata:
      labels:
        app: n
    spec:
      containe
      - name:
        image:
        ports:
        - cont
```

frontend-src.yaml

```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    run: frontend
  name: frontend
spec:
  ports:
  - port: 80
    protocol: TCP
    targetPort: 8
  selector:
    run: frontend
  type: LoadBalanc
status:
  loadBalancer: {}
```

bff-api.yaml

```yaml
apiVersion: apps
kind: Deployment
metadata:
  labels:
    run: bff
  name: bff
spec:
  replicas: 2
  selector:
    matchLabels:
      run: bff
  template:
    metadata:
      labels:
        run: bff
    spec:
      containers
      - image:
        name: b
        ports:
        - contai
```

bff-api-svc.yaml

```yaml
apiVersion: v
kind: Service
metadata:
  labels:
    run: bff
  name: bff
spec:
  ports:
  - port: 808
    protocol:
    targetPort
  selector:
    run: bff
  type: LoadBa
status:
  loadBalancer
```

consumer.yaml

```yaml
apiVersion: apps
kind: Deployment
metadata:
  labels:
    run: consume
  name: consumer
spec:
  replicas: 2
  selector:
    matchLabels:
      run: consu
  template:
    metadata:
      labels:
        run: con
    spec:
      containers
      - image: d
        name: co
        ports:
        - contai
```

db.yaml

```yaml
apiVersion:
kind: Deploy
metadata:
  labels:
    run: mys
  name: mysc
spec:
  replicas:
  selector:
    matchLab
      run: m
  template:
    metadata
      labels
        run:
    spec:
      contai
      - imag
        name
        port
        - containerPort: 3306
```

db-svc.yaml

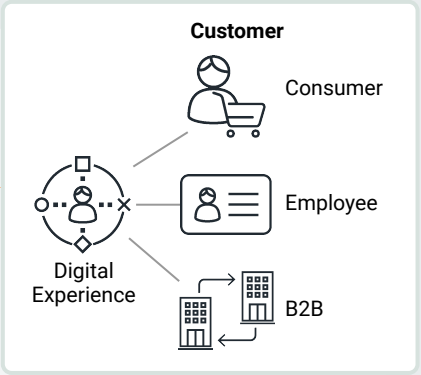
```yaml
apiVersion: v1
kind: Service
metadata:
  labels:
    run: mysql
  name: mysql
spec:
  ports:
  - port: 3306
    protocol: TCP
    targetPort: 3306
  selector:
    run: mysql
  type: LoadBalancer
status:
  loadBalancer: {}
```

Because manufacturing software in the cloud-native era is very hard



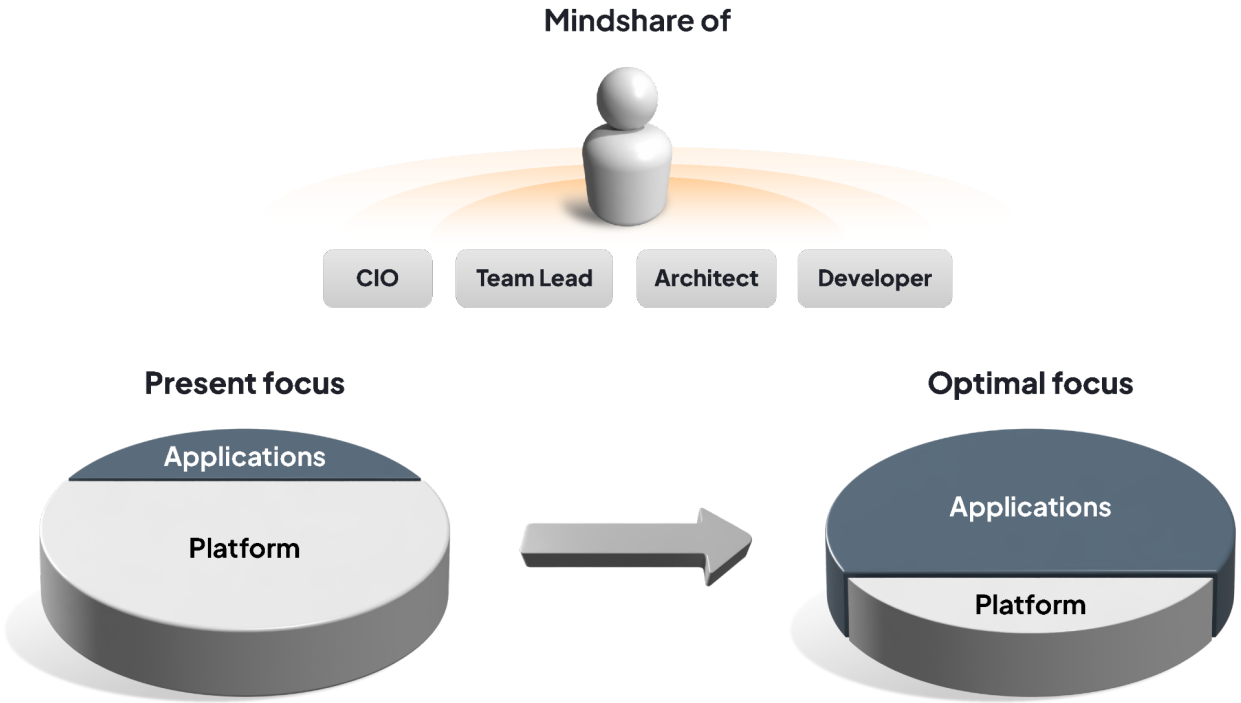
You write code. You want to get it to the hands of the customer ASAP.

- Runtime architecture, CI/CD, DevOps, environments, SecOps, configuration management, version management, testing, observability, analytics, and SRE
- New features in development needs re-use of existing code/libs/APIs
- DevOps goes to Blue/Green, etc.
- Self-service and policy-driven
- Zero-trust environment



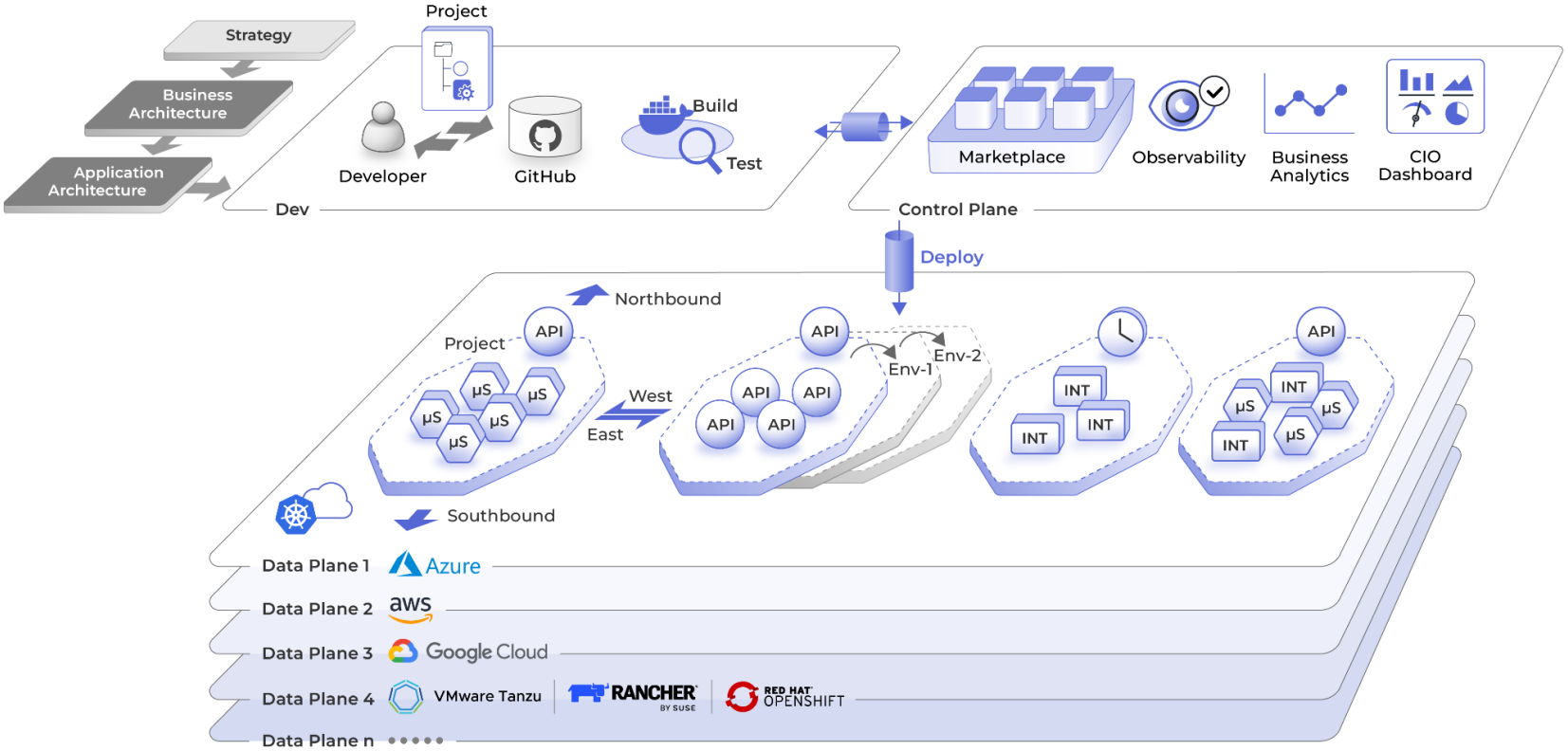
Only digital natives have software manufacturing down to a science; all others are struggling.

Enterprises should focus on engineering applications, not platforms



Focusing on adding business value

What is Choreo?



Let’s build it (practical session)

Demo: Building Hotel Reservation System

Use case

"The Luxury Hotel wants to build a reservation system for the public website"

Room Search: Users can find rooms by entering check-in and check-out dates and can apply filters based on the guest count.

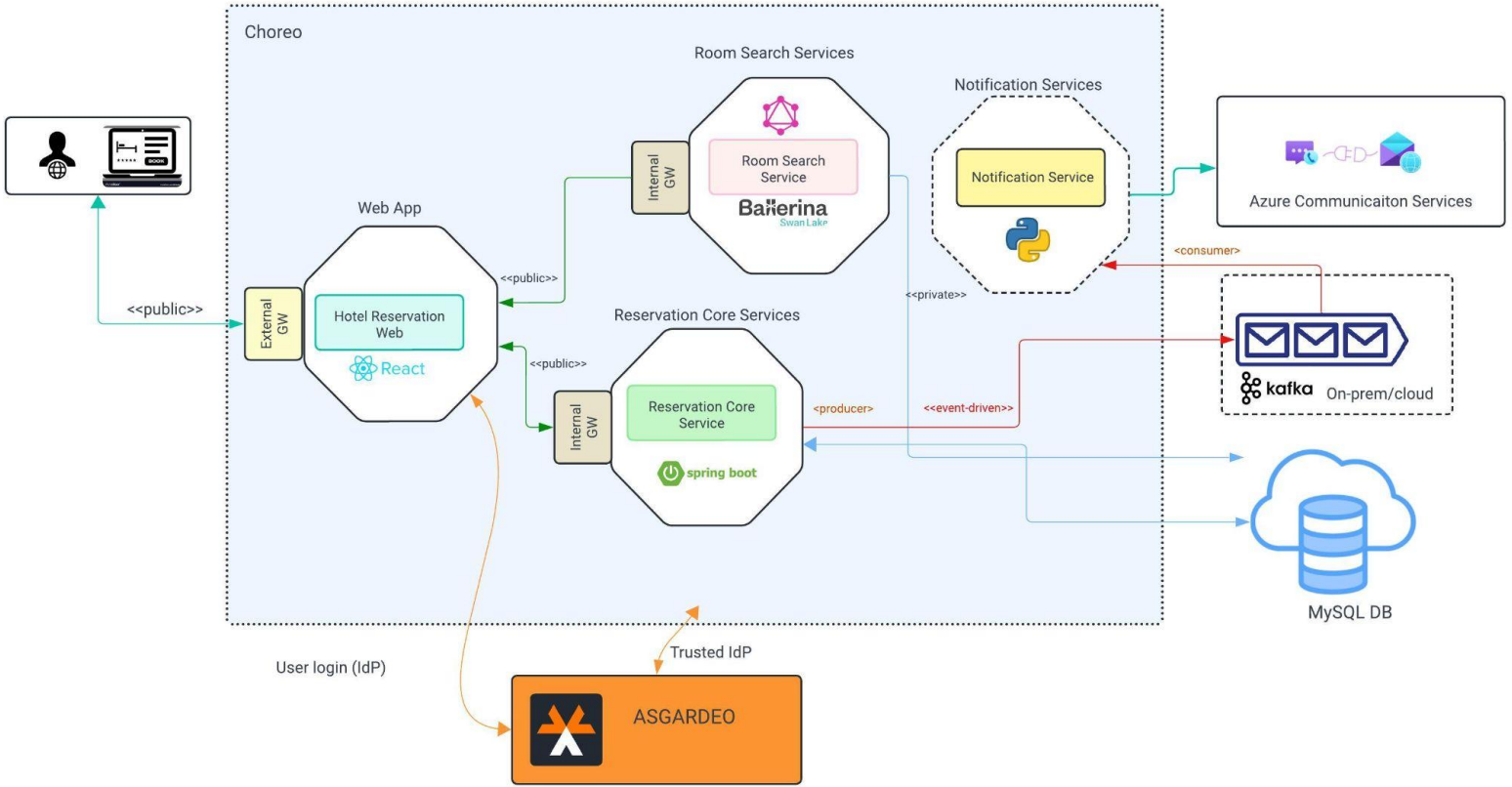
Room Reservation: With requires entering personal details (full name, contact number, email)

Update Reservations: Users can change their booking dates

Cancel Reservations: Offers an easy option for users to cancel their bookings at any time.

Please refer to the instructions guide for more details.
https://github.com/wso2con2024/architecture-tutorial

Solution architecture view



Azure Communication Configs



Thank You!

Provide Feedback



WSO2CON2024