KubeCon | CloudNativeCon

North America 2024

Mastering Cell Based Architecture

Practical Solutions and Best Practices

Asanka Abeysinghe | **CTO @ WSO2**

creator and original author of Cell-Based Architecture, Digital Double, the Platformless Manifesto, and the technical definition of the Internal Developer Platform(IDP)

https://blog.architect2architect.com

@asankama

https://www.linkedin.com/in/asankaabeysinghe/

WSO2

https://wso2.com/

R46

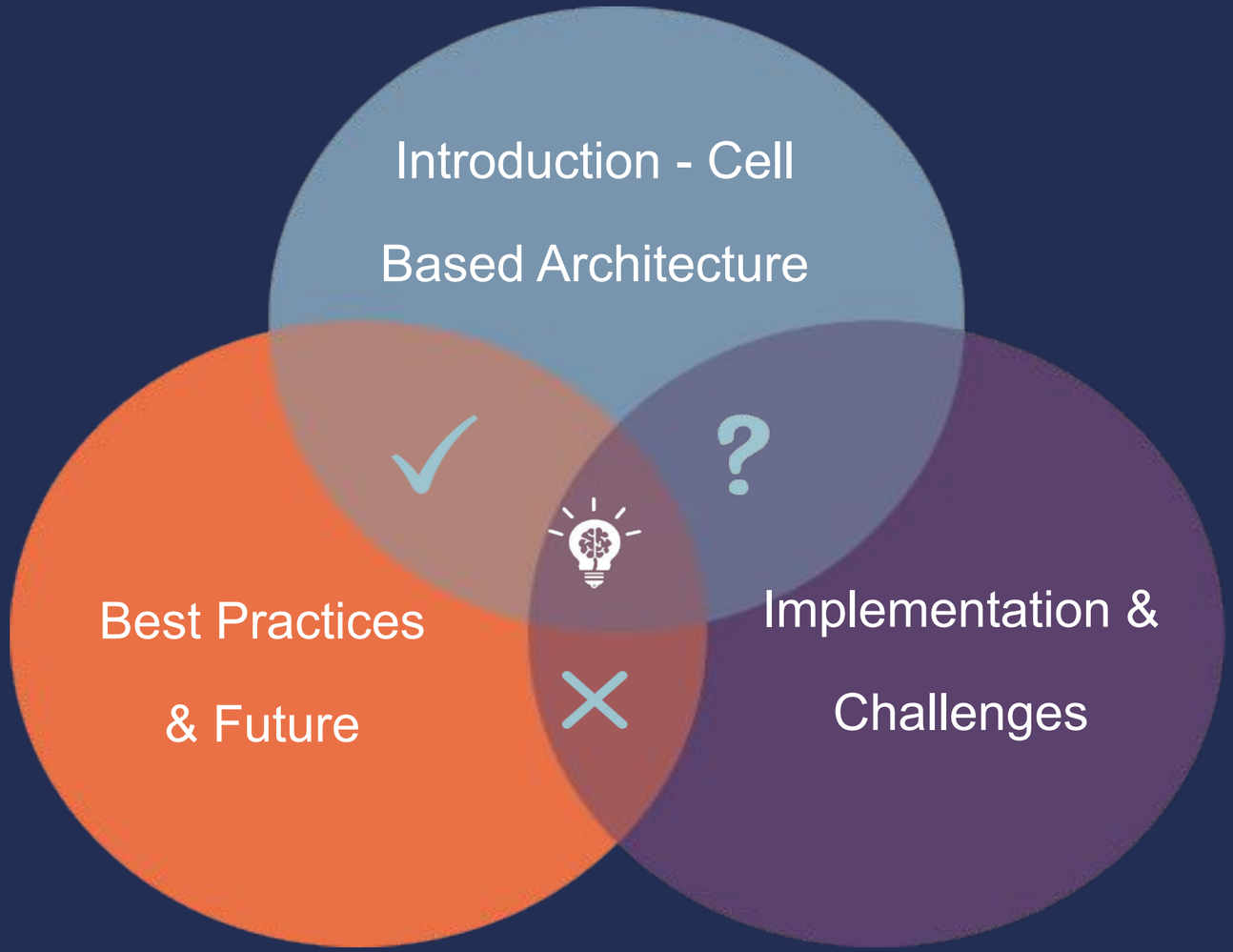
R43

Shweta Vohra

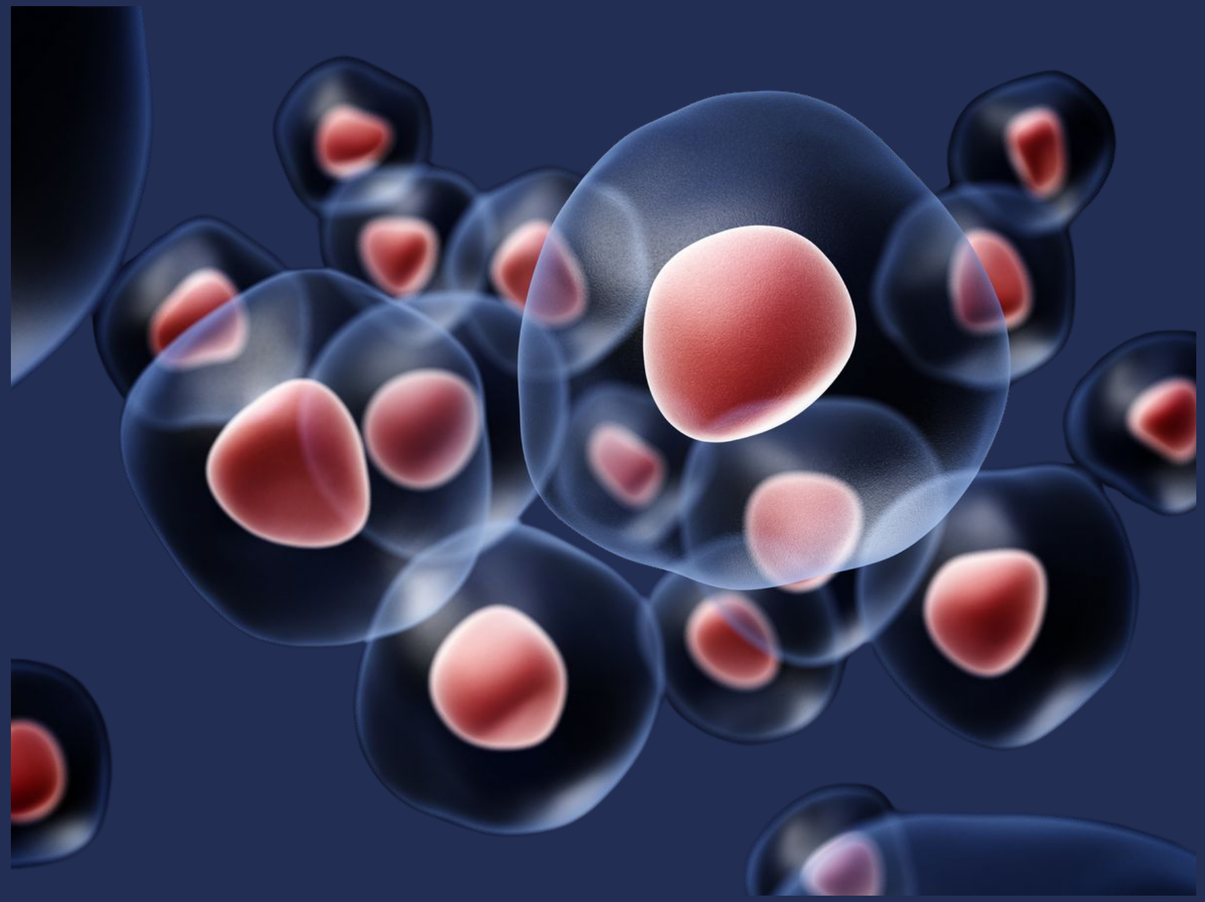
Lead Architect @ Booking.com

Inventor | Author

Agenda


Introduction - Cell
Based Architecture
Best Practices
& Future
Implementation &
Challenges

Cell Based Architecture: Biological and Architectural



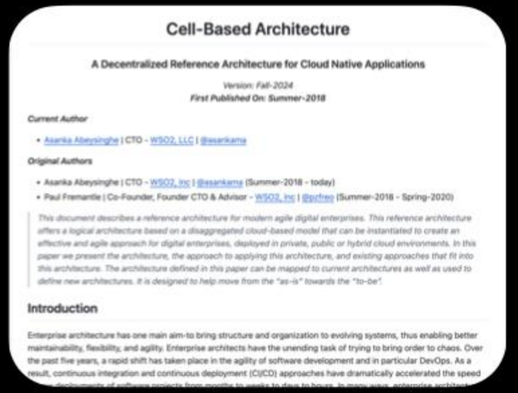
- Smallest Unit
- Self Contained Unit
- Independently Elastic
- Clear Communication Boundaries

Tell us about -

Cell-Based Reference Architecture

Looking back…..

cellbasedarchitecture.org



Cell-Based Architecture

A Decentralized Reference Architecture for Cloud Native Applications

Version: Fall-2024
First Published On: Summer-2018

Current Author

- Asanka Abeysinghe | CTO - WSO2, LLC | @asankama

Original Authors

- Asanka Abeysinghe | CTO - WSO2, Inc | @asankama (Summer-2018 - today)
- Paul Fremantle | Co-Founder, Founder CTO & Advisor - WSO2, Inc | @pzfreo (Summer-2018 - Spring-2020)

This document describes a reference architecture for modern agile digital enterprises. This reference architecture offers a logical architecture based on a disaggregated cloud-based model that can be instantiated to create an effective and agile approach for digital enterprises, deployed in private, public or hybrid cloud environments. In this paper we present the architecture, the approach to applying this architecture, and existing approaches that fit into this architecture. The architecture defined in this paper can be mapped to current architectures as well as used to define new architectures. It is designed to help move from the "as-is" towards the "to-be".

Introduction

Enterprise architecture has one main aim-to bring structure and organization to evolving systems, thus enabling better maintainability, flexibility, and agility. Enterprise architects have the unending task of trying to bring order to chaos. Over the past five years, a rapid shift has taken place in the agility of software development and in particular DevOps. As a result, continuous integration and continuous deployment (CI/CD) approaches have dramatically accelerated the speed

Abeysinghe, A., & Fremantle, P. (2018, June). Cell-based architecture: A decentralized reference architecture for cloud-native applications. https://github.com/wso2/reference-architecture/blob/master/reference-architecture-cell-based.md

56 talks

10+ articles

contribution

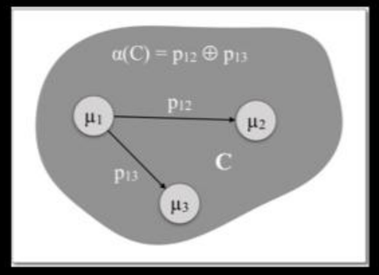
ref. implementation #1

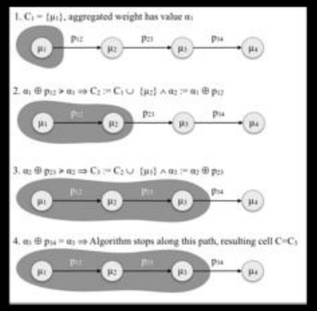
Cellery

architecture as code

```
cellery:Component helloComponent = {
    name: "hello-api",
    src: {
        image: "docker.io/wso2cellery/samples-hello-world-api" // source docker image
    },
    ingresses: {
        helloApi: <cellery:HttpApiIngress>{
            port: 9090,
            context: "hello",
            definition: {
                resources: [
                    {
                        path: "/",
                        method: "GET"
                    }
                ]
            },
            expose: "global"
        }
    },
    envVars: {
        MESSAGE: { value: "hello" }
    }
};
```

mathematical model

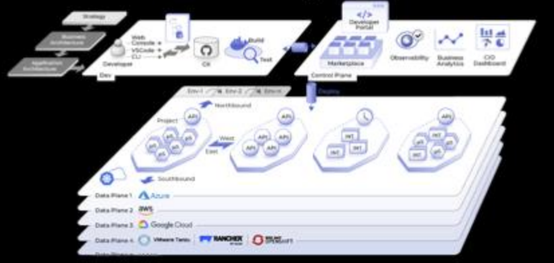




ref. implementation #2

R46

Choreo by WSO2

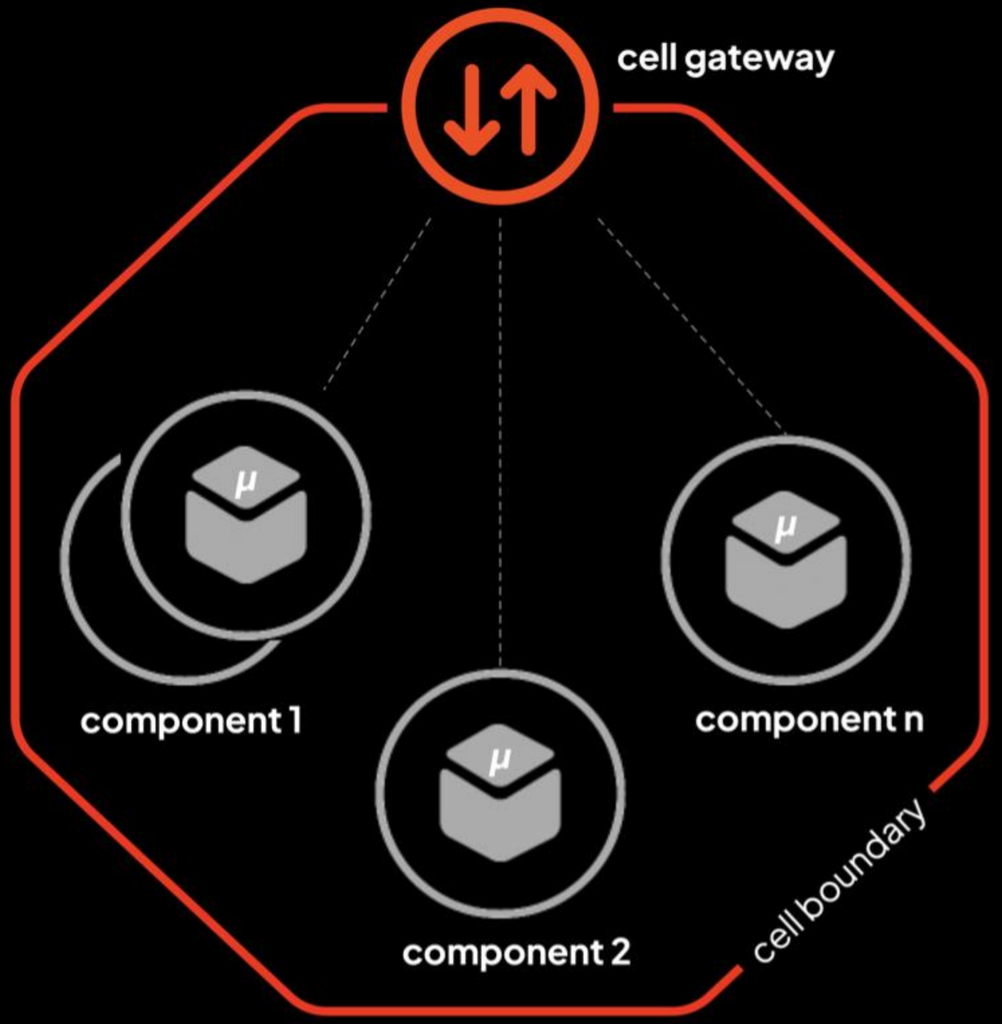


https://wso2.com/choreo

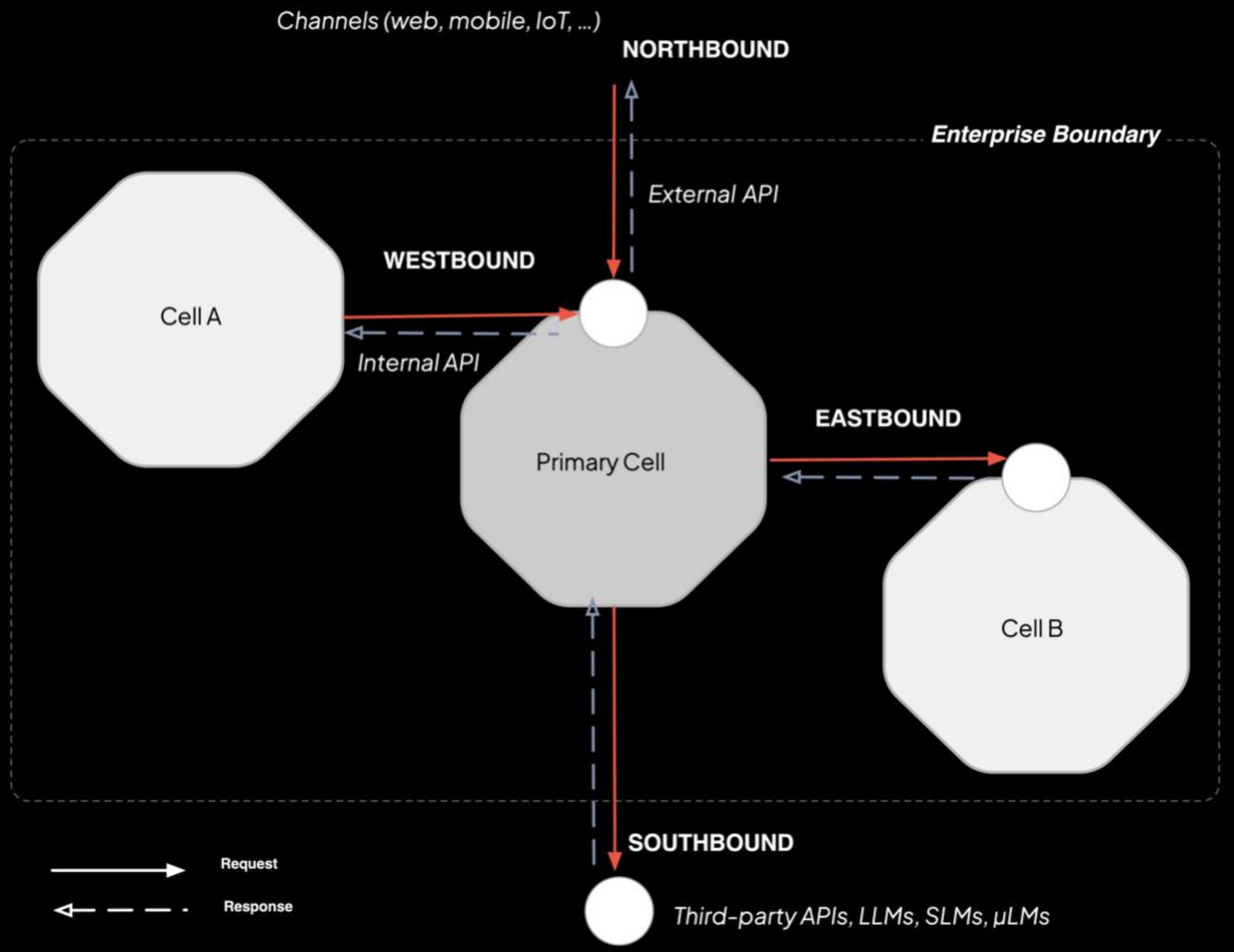
Cell-based Architecture (CBA) is a fusion of application, deployment, and team architecture—engineered to scale and adapt seamlessly to changing technical and business demands.

Cell: units of enterprise architecture

A **cell** is a collection of components, grouped from design and implementation into deployment.
A cell is independently deployable, manageable, and observable.



Cell communication



Tell us interesting use cases with CBA and challenges?

Industry Use Cases

Cells - Intuitive & Relatable

Domain Segmentation

Microservices/ Containerized

Security & Separation

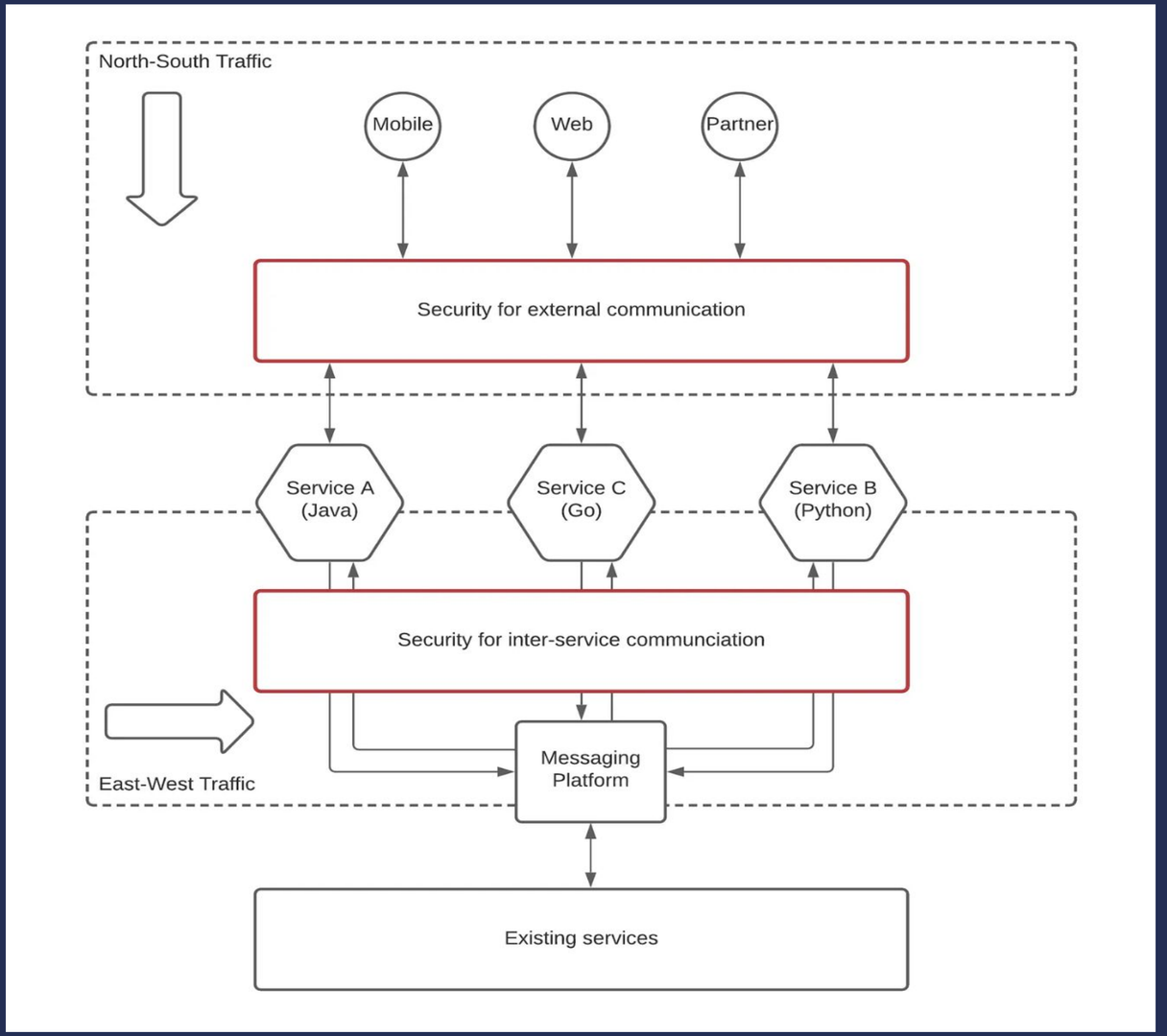
Resiliency

Three Prominent Challenges

- Complexity in Cell Coordination
- Security and Isolation
- Operational Overhead

Complex Cell Coordination

Cells must coordinate to ensure seamless inter-cell communication, which can add complexity in large systems.



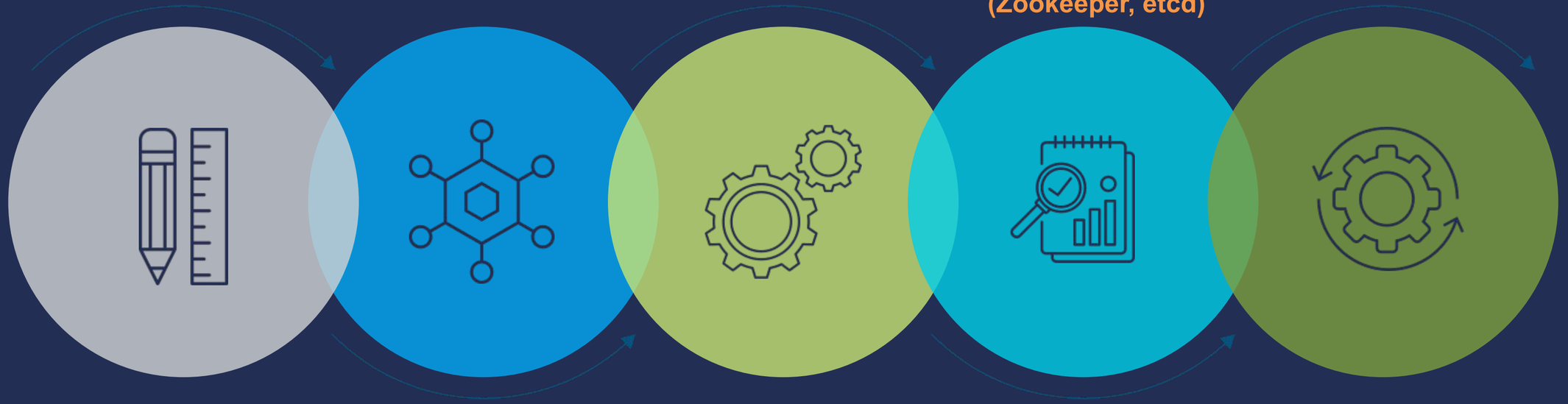
PC: https://medium.com/microservices-learning/how-to-implement-security-for-microservices-89b140d3e555

Complex Cell Coordination

Key Considerations

Event-Driven Architecture for Decoupled Communication (Kafka, Messaging Services)

Distributed Coordination for State Management & Leader Election (Zookeeper, etcd)



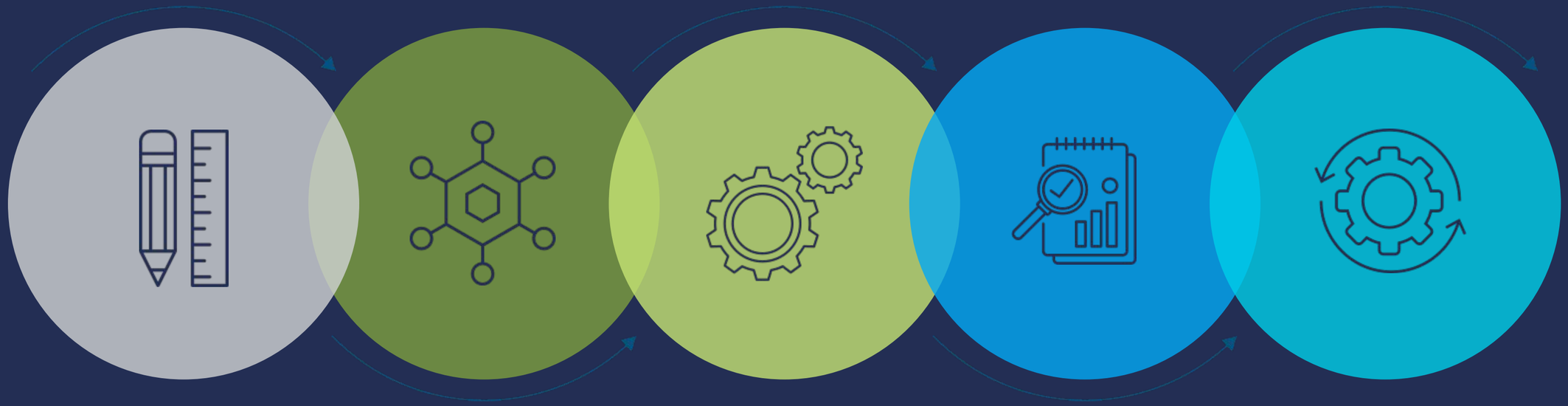
Standardized Communication Protocols (gRPC, Rest APIs, Cell Gateways)

Service Mesh for Communication Management (Istio, Mesh Services)

Prepare for Optimization (Versioning, Automated Retries)

Security & Isolation

Providing robust security and isolation for each cell, particularly in multi-tenant systems.

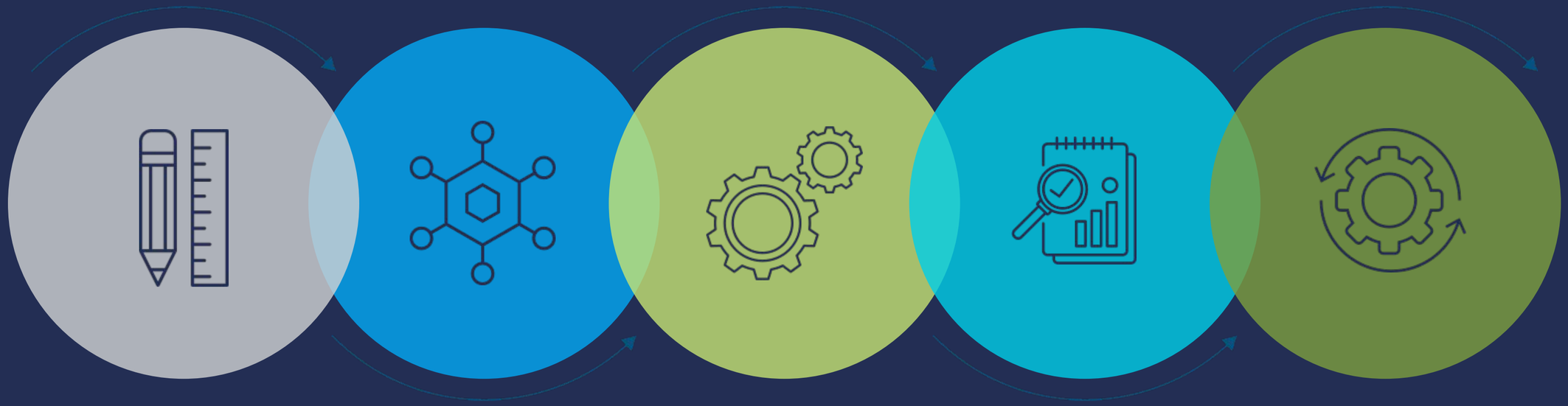


Operational Overhead

Increased overhead for managing separate cells, including monitoring, logging, and deploying updates across cells.

Configuration Management and Consistency (Centralized vs. Distributed)

Self-Healing and Autoscaling Mechanisms



Automated CI/CD for Streamlined Deployment and Updates

Centralized Monitoring and Alerting

Additional Considerations (Access control, health check)

Cell Boundaries best practices - Do’s and Don’t

Defining cell boundaries

Pattern + Context

The design of systems has always required an approach to the clustering of functionality, and it remains an open computer science problem - so don't expect a definitive answer!

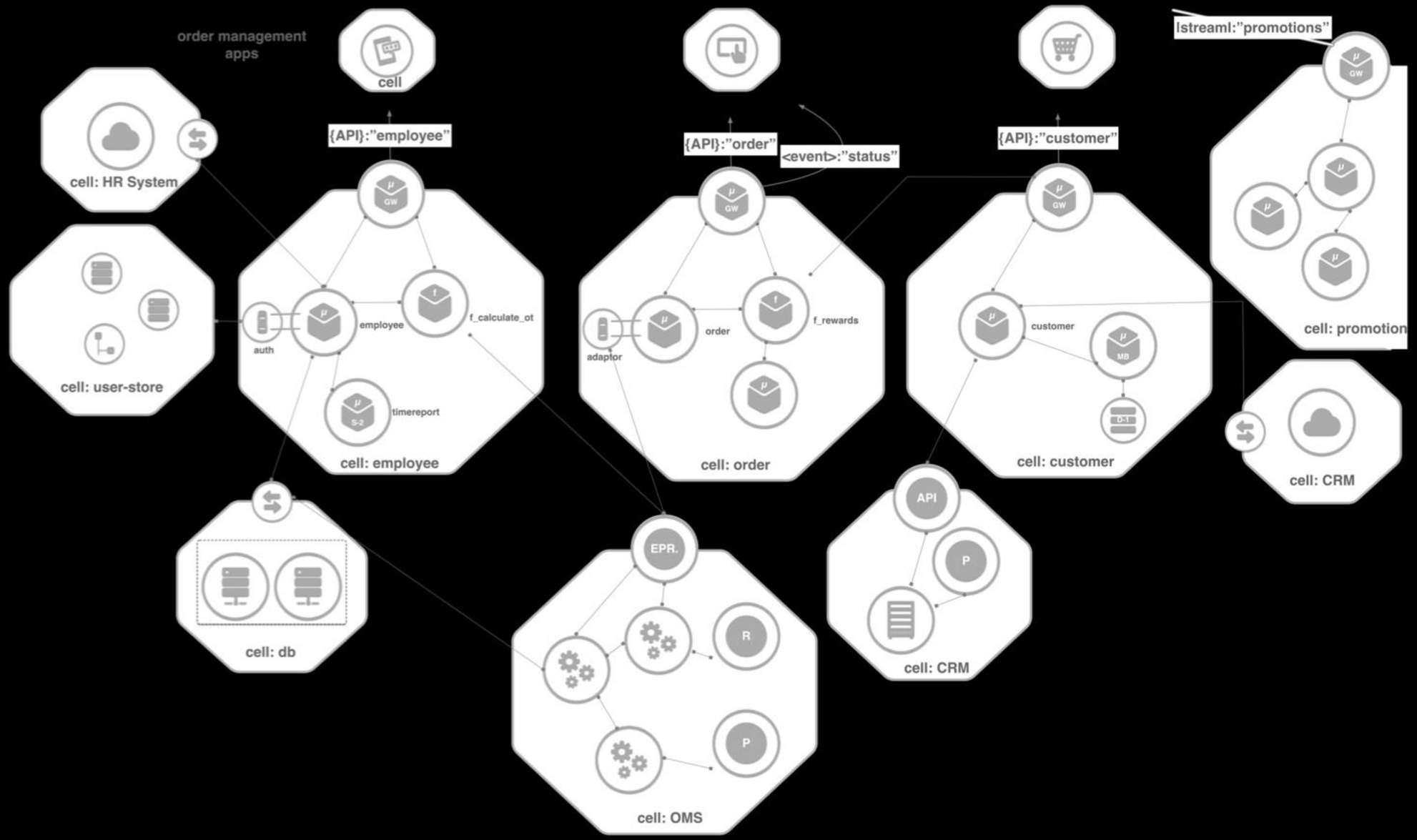
Defining cell boundaries

The number of component-component connections within a cell should be higher than the number that crosses the cell boundary

Approaches such as domain-driven design (DDD) help, but fundamentally the cell model is there to provide team boundaries

Hence the size of a cell should be based on the size, responsibility, and output of a team.

Applying CBA with DDD in action



Can I use CBA for both greenfield and brownfield implementations?

Greenfield CBA Implementations

Greenfield Design - Use Case & Approach

Step 1

Define Cells Based on Functional Boundaries

- Cell 1: Stateless microservice on Kubernetes.
- Cell 2: Stateful microservice on Kubernetes with persistent storage.
- Cell 3: Serverless function (AWS lambda)

Step 2

Isolate Stateful & Stateless Services

- Use separate Kubernetes namespaces for each microservice
- Ensure isolation at the networking level and
- Handle State with Robust Data Management (for stateful)
- Apply resource quotas based on Cells needs

Greenfield Design - Use Case & Approach

Step 3

Use Service Mesh for Cross-Cell Communication

- CBA isolates cells, but services may need inter-communication
- Service mesh enables secure, efficient cell communication
- Provides traffic management, observability, and security
- Good for Cells 1 type of Cell

Step 4

Apply API Gateways & Lambda Invocation

- Use an API Gateway to expose stateful and stateless microservices.
- API Gateway handles routing and external access.
- AWS Lambda (Cell 3) can be invoked via API Gateway or event triggers (e.g., S3 upload, SNS).
- Set rate-limiting, authentication, and access policies in the API Gateway for secure access.

Step 5

Security and Isolation

- Apply RBAC at the Kubernetes cluster to restrict microservice access.
- Use IAM roles/policies to control Lambda access for authorized services.
- Set API Gateway policies to enforce authentication (e.g., OAuth, JWT) for requests.

Greenfield Design - Use Case & Approach

Step 6

Resource & Scaling Management

- Enable horizontal pod autoscaling for the stateless microservice (Cell 1) to manage traffic spikes.
- Manage resources for the stateful microservice (Cell 2) to scale while keeping state consistency; consider using a cluster autoscaler.
- Lambda scales automatically; set memory and timeout limits based on workloads.

Step 7

Ensure Fault Tolerance

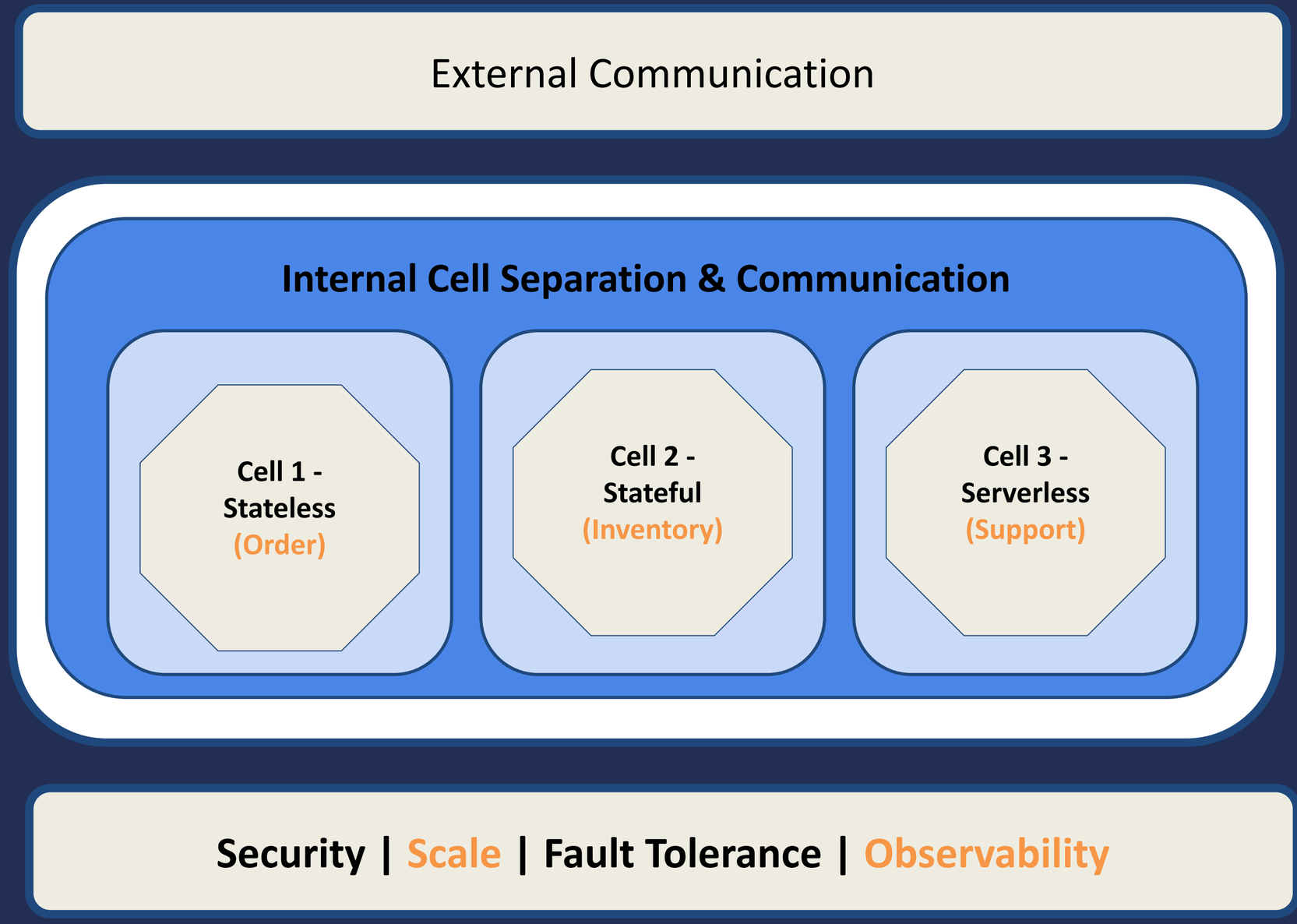
- Use circuit breakers to prevent cascading failures if a microservice is unavailable.
- Implement graceful degradation in the API Gateway to manage partial failures.
- Set failover strategies for the stateful service, like data replication across zones or regions.

Step 8

Observability and Monitoring

- Use centralized logging tools like ELK or Prometheus/Grafana for metrics and observability.
- Leverage the service mesh for inter-cell metrics and resource tracking.
- Enable CloudWatch for Lambda to monitor execution, errors, and performance.

Greenfield Design - Use Case & Approach



Brownfield CBA Implementations

Greenfield Design - Use Case & Approach

Step 1

Understand the Architectural Complexity

- Portability: Enable movement between on-prem and cloud environments.
- Stateful Nature: Handle persistent storage and data consistency for monolithic and microservices.
- Legacy Considerations: Monolithic on-prem app has tightly coupled components, complicating service isolation.

Step 2

Analyzing the Portability of Each Component in CBA

- Monolithic App (On-Prem): Difficult to port; refactor gradually into microservices for future CBA fit.
- Stateful Microservice (On-Prem): Containerize with Kubernetes for portability; suitable for CBA with service mesh.
- Stateful App (Cloud/Hybrid): Strong CBA fit; use Kubernetes for cloud-native storage and hybrid flexibility.
- Stateless App (Cloud/Hybrid): Highly portable; ideal for CBA with independent scaling and API-based communication.

Brownfield Design - Use Case & Approach

Step 3

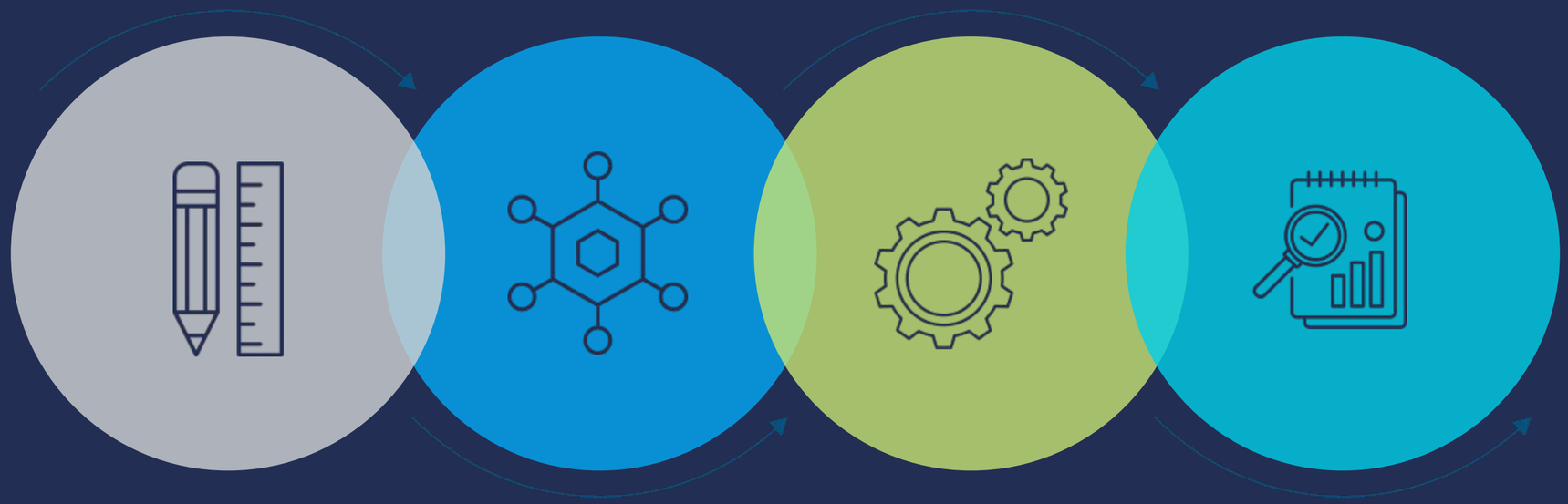
To use or Not to Use CBA?

- Reasons to Use CBA:
  - Hybrid Flexibility: Encapsulates services into cells for on-prem/cloud portability, starting with stateful/stateless microservices.
  - Monolith Decomposition: Gradually refactor monolith into smaller services and move to the cloud.
  - Incremental Migration: Migrate stateful services in phases, ensuring consistency and resilience.
  - Security & Isolation: Zero-trust boundaries protect on-prem/cloud services from breaches or failures.

- Reasons Not to Use CBA:
  - Monolithic Constraints: Large, complex monoliths may limit CBA benefits due to refactoring challenges.
  - Operational Overhead: Hybrid CBA adds complexity in managing varied on-prem/cloud requirements without a clear strategy.

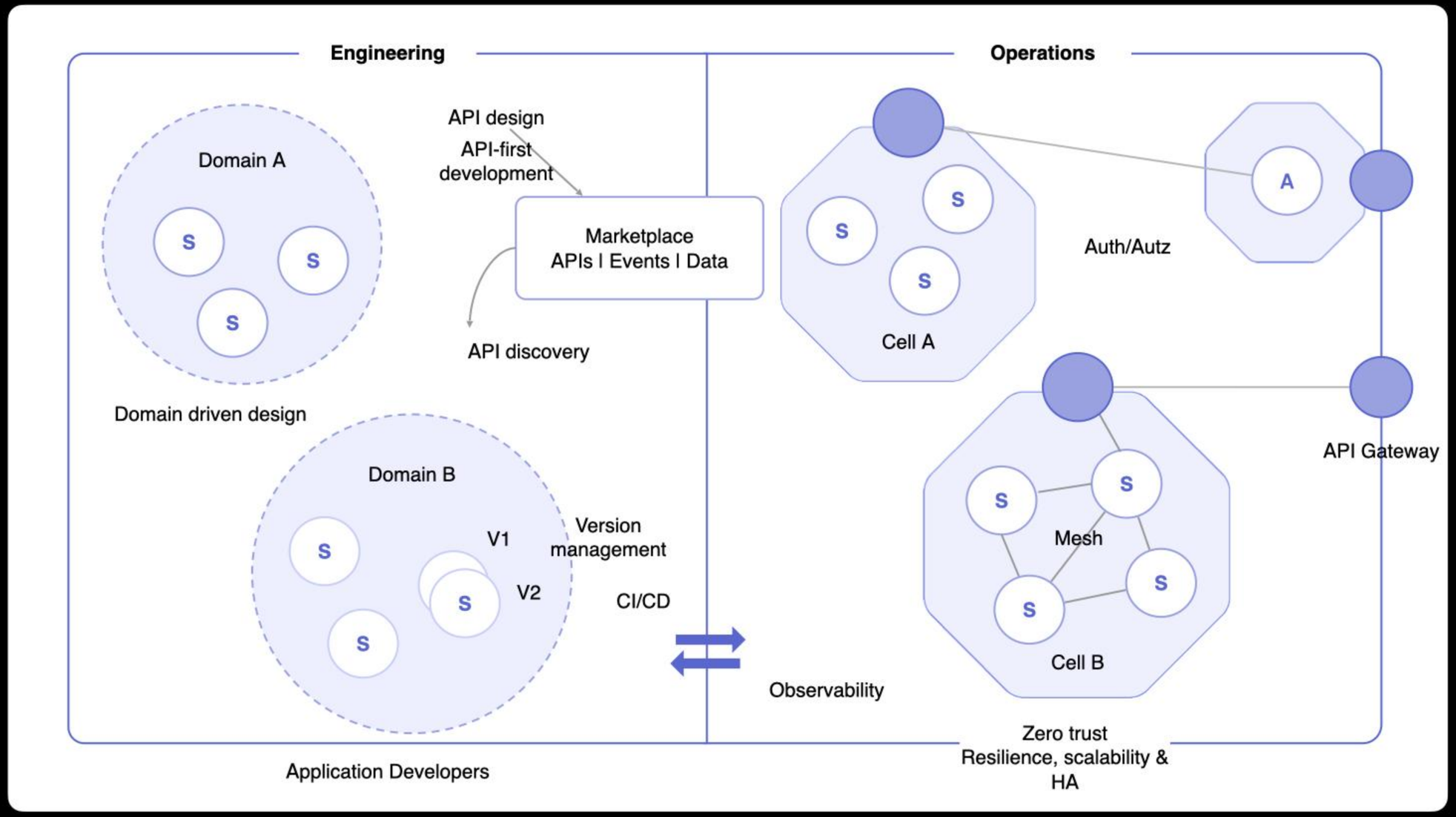
CBA - Best Practices

Data Practices



Tell us some interesting CBA - Industry Use cases & Trends

Push it down to the internal developer platform



Modularity

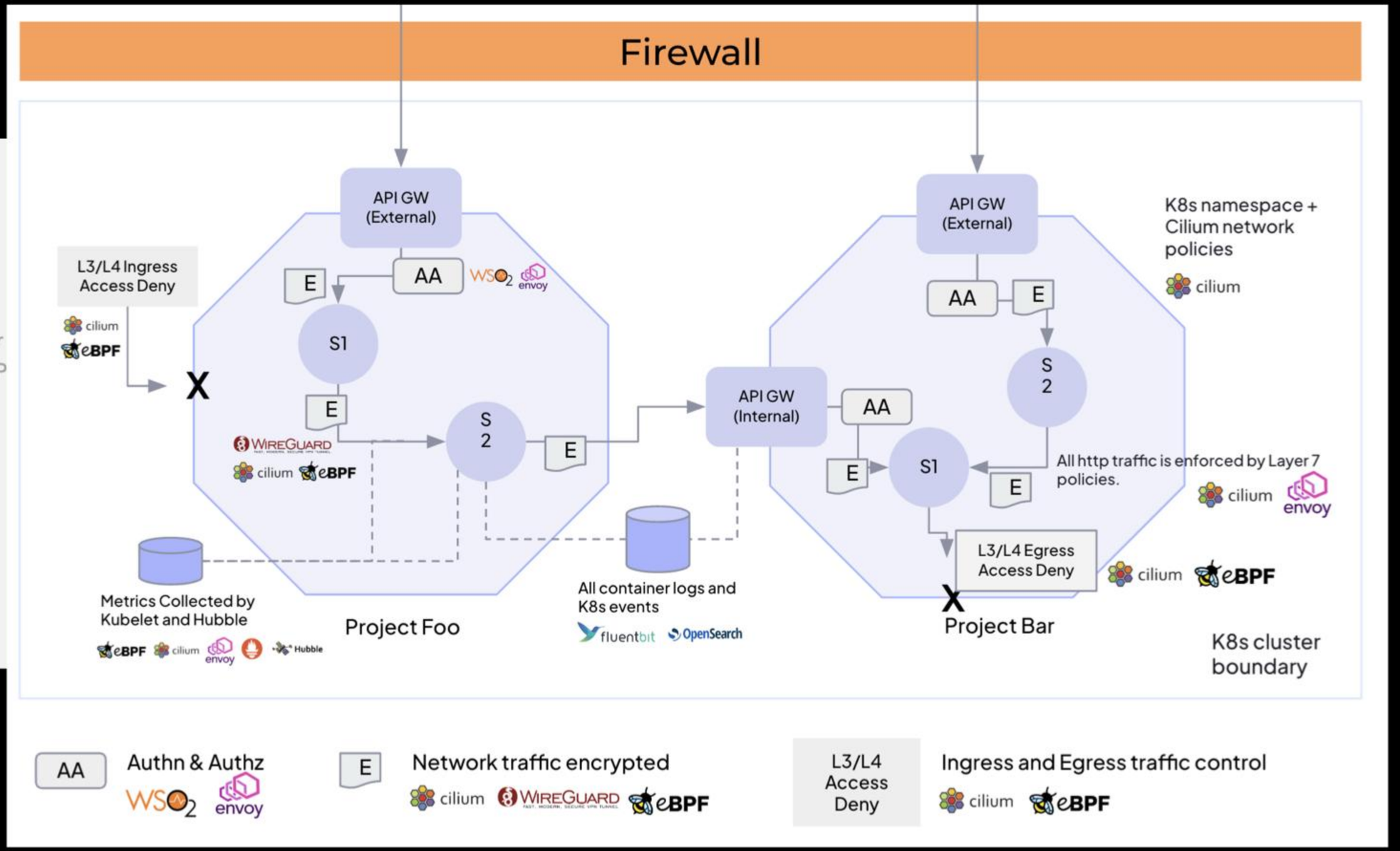
Discovery

Orchestration

Autonomy

CBA implementation in k8s ecosystem

```
{
  "group": "core",
  "version": "v1",
  "kind": "Project",
  "metadata": {
    "name": "expense-tracker_01",
    "displayName": "Expense Tracker
    "description": "Expense Tracker P
    "labels": {
        "type": "web",
        "category": "finance"
      }
  },
  "spec": {
    "region": "US"
  }
}
```



Cells give structure to life, just like the foundation of modern software systems. Without them, everything would be a ‘big blob of mush”!

Q&A


Grab Signed Copy
DECODING
PLATFORM PATTERNS
Shweta Vohra
Complete Blueprint to Harness Technology Platforms Power from Strategy to Engineering for Business Success
Foreword By Priyanka Sharma (Executive Director, CNCF)
kindle
Devtron Booth
@3:30 pm
Today only!