From Monolith to Microservices: Navigating the Journey and Avoiding Pitfalls

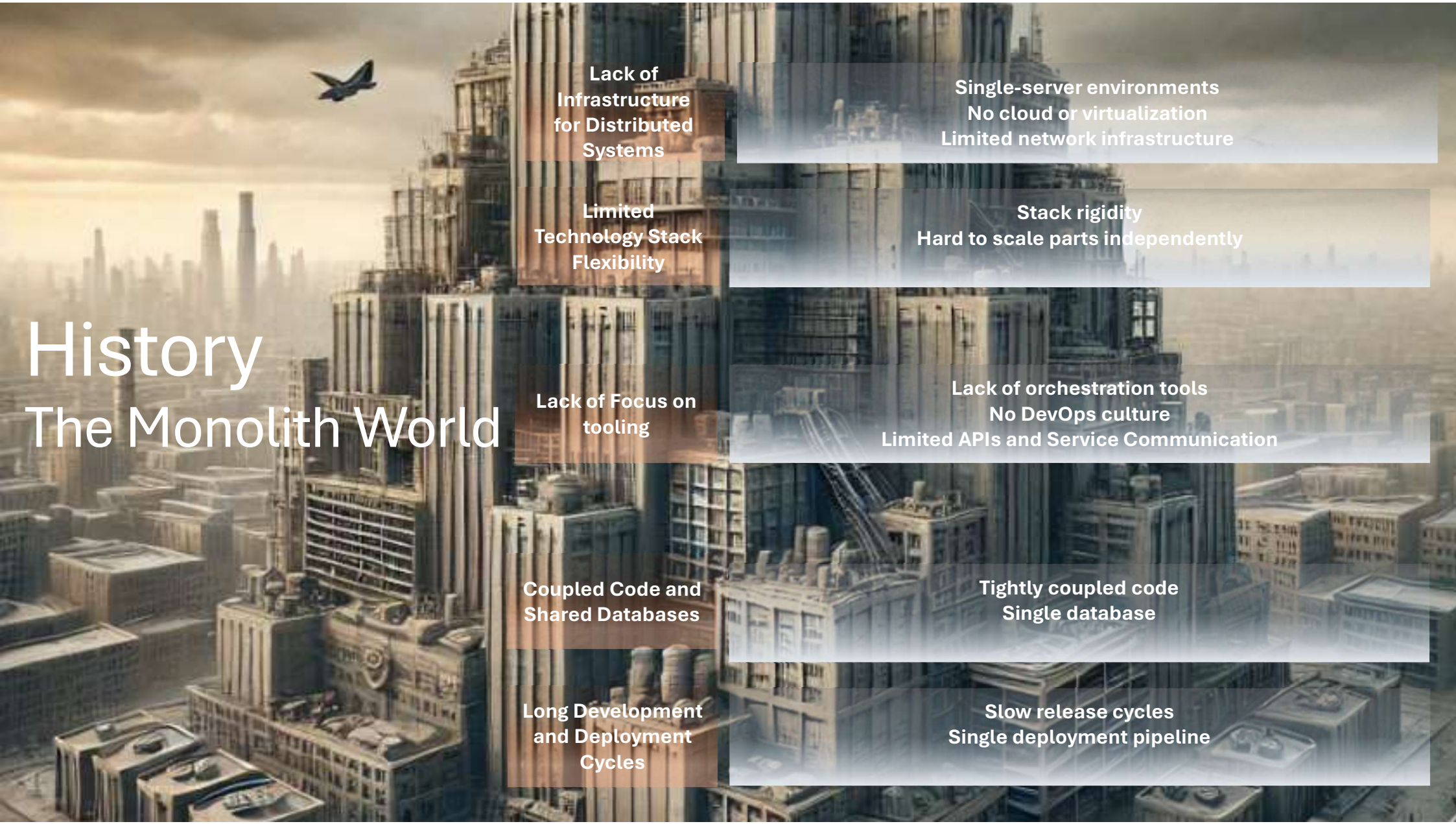
Vitor Paulino

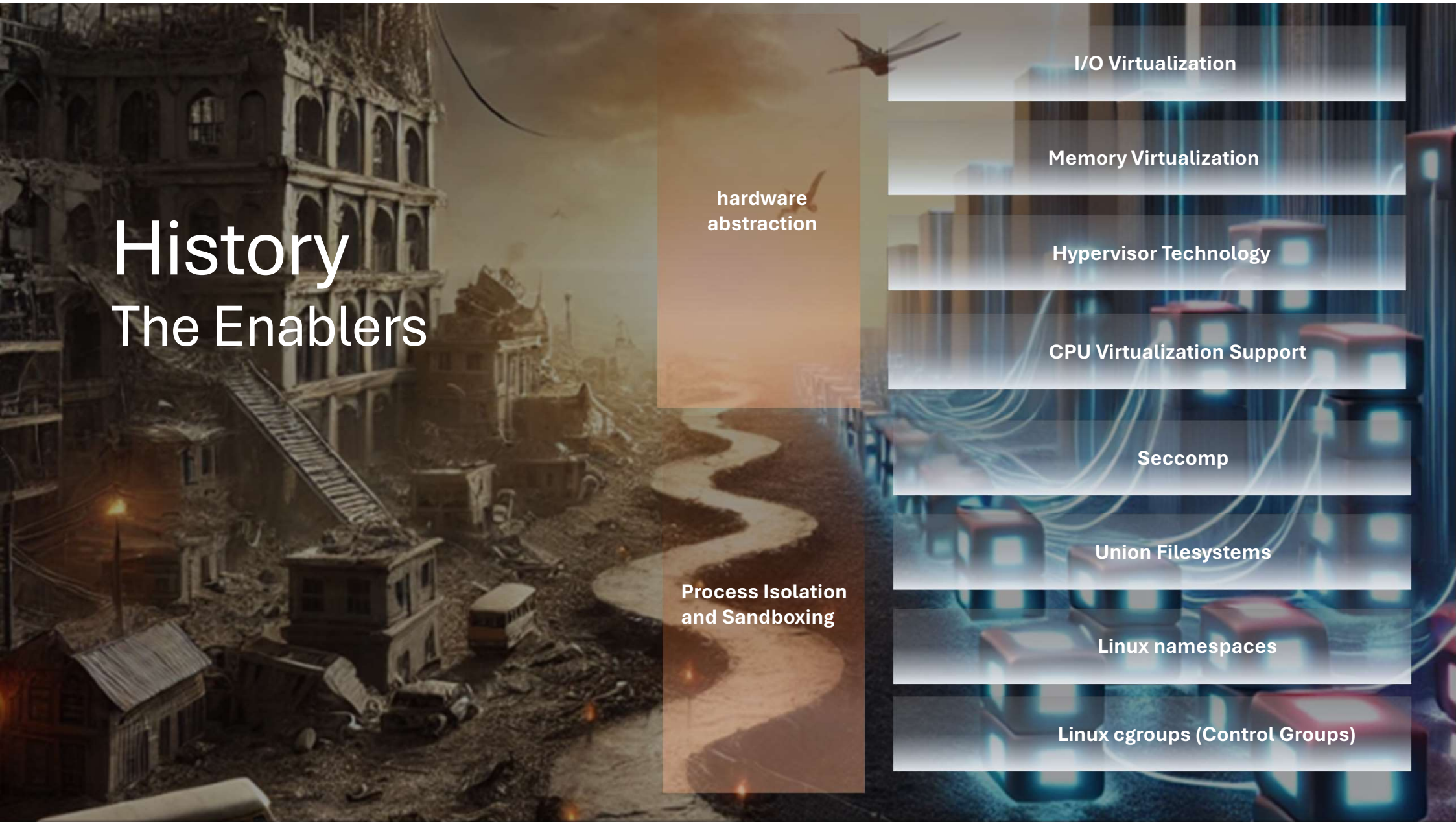
Bio

I'm a Software engineer with almost 15 years of experience on designing, developing, testing and deploying backend distributed applications mainly on .NET, Java and javascript technologies stacks to support a variety of different software designs patterns customized to the needs of the client products requirements. Since the beginning until now I have been working with diferent methodologies and mindsets. From Waterfall to Agile with scrum until more recently working on product development in a BizDevOps team. The mindset that I put in place every time I face a new challenge is aligned with Agile software development best practices such as Agile Architecture, Software design SOLID principles, GoF design patterns and methodologies that enable good software quality such as TDD. Always available to learn, adapt, help, coach and support the team, sharing knowledge an discuss solutions.

About Me


Agenda
• History
• What Can Go Wrong?
• How to Prevent Problems
• Proposed Strategy
• Use Cases


History
The Monolith World
Lack of Infrastructure for Distributed Systems
Single-server environments
No cloud or virtualization
Limited network infrastructure
Limited Technology Stack Flexibility
Stack rigidity
Hard to scale parts independently
Lack of Focus on tooling
Lack of orchestration tools
No DevOps culture
Limited APIs and Service Communication
Coupled Code and Shared Databases
Tightly coupled code
Single database
Long Development and Deployment Cycles
Slow release cycles
Single deployment pipeline


History
The Enablers
hardware abstraction
Process Isolation and Sandboxing
I/O Virtualization
Memory Virtualization
Hypervisor Technology
CPU Virtualization Support
Seccomp
Union Filesystems
Linux namespaces
Linux cgroups (Control Groups)


What Can Go Wrong
Paradigms, Theorems & Laws
Expectations Vs Reality
Pitfalls
RedFlags


What Can Go Wrong
Paradigms Theorems & Laws
CONWAY'S
PASTELS
Monolith First.
CAP THEOREM
OOP & DDD


What Can Go Wrong
The Promissed Land
• Scalability
• Flexibility in Technology Choices
• Fault Isolation
• Faster Time-to-Market
• Improved Maintainability
• Support for Modern Development Practices

What can go wrong?
The Reality of Transformation

- PREMATURE SPLITTING LEADING TO OVER-ENGINEERING.
- COMPLEX INTER-SERVICE COMMUNICATION.
- INCREASED LATENCY AND REDUCED PERFORMANCE.
- LACK OF TEAM EXPERTISE IN NEW TOOLS OR PARADIGMS.
- DATA CONSISTENCY PROBLEMS WITH DISTRIBUTED SYSTEMS.

What Can Go Wrong?

Pitfalls in the Real World

- **Unclear Boundaries:** Poorly defined domains lead to overlapping responsibilities.
- **Operational Overhead:** Each service introduces its own deployment, monitoring, and maintenance.
- **Versioning Chaos:** Breaking changes across APIs and lack of backward compatibility.
- **Observability Gaps:** Insufficient tooling for tracing and debugging issues across services.

What Can Go Wrong

RedFlags

- **Deployment frequency drops**

As complexity increases, deployments become harder to manage and test, causing delays and reducing delivery speed.

- **Team productivity declines due to complexity**

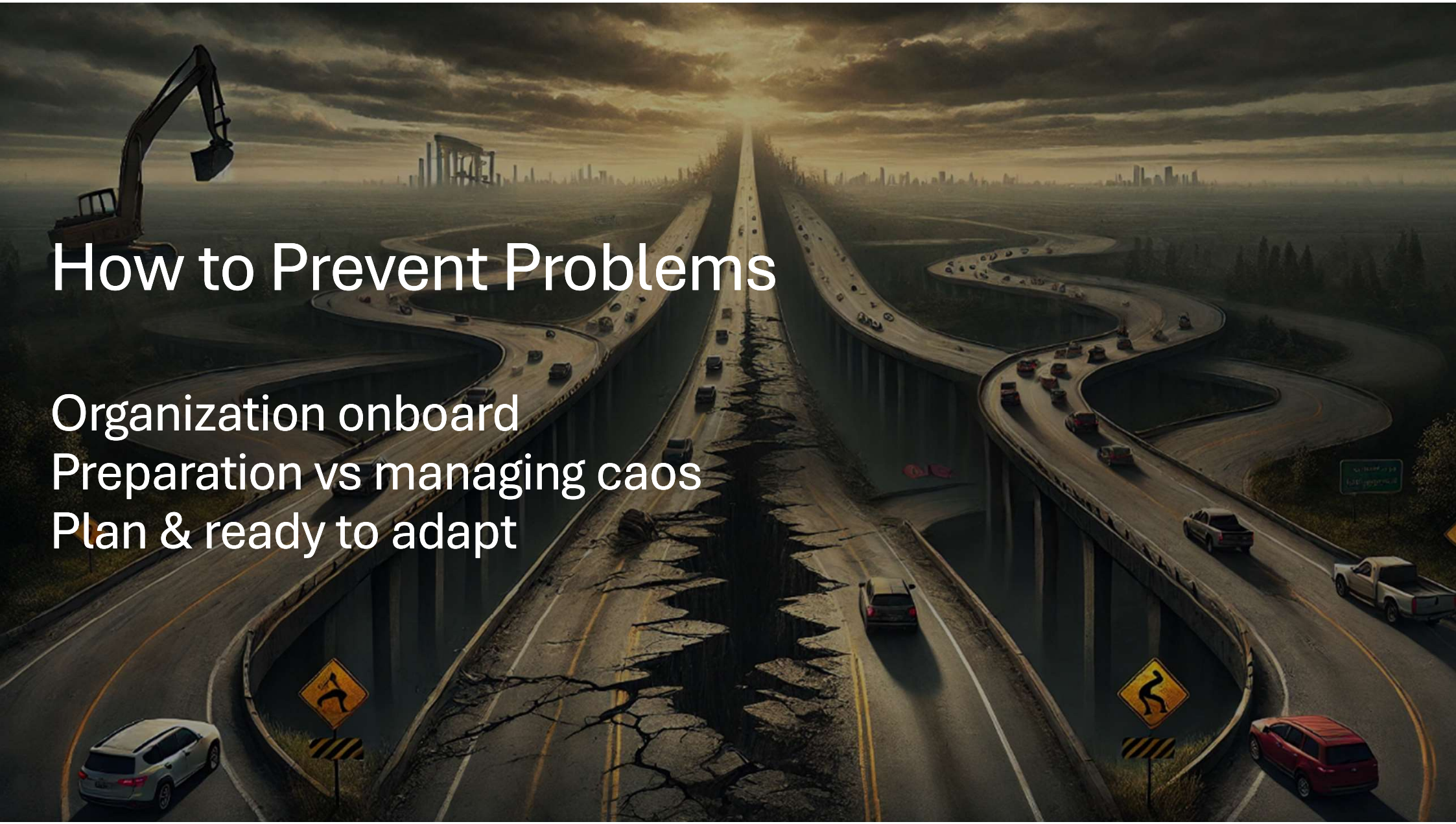
Teams spend more time dealing with inter-service dependencies, communication issues, and debugging rather than delivering features.

- **frequent service outages or slow response times**

Poorly designed systems introduce latency and availability issues, as services depend on each other for critical operations.

- **Developer frustration and burnout.**

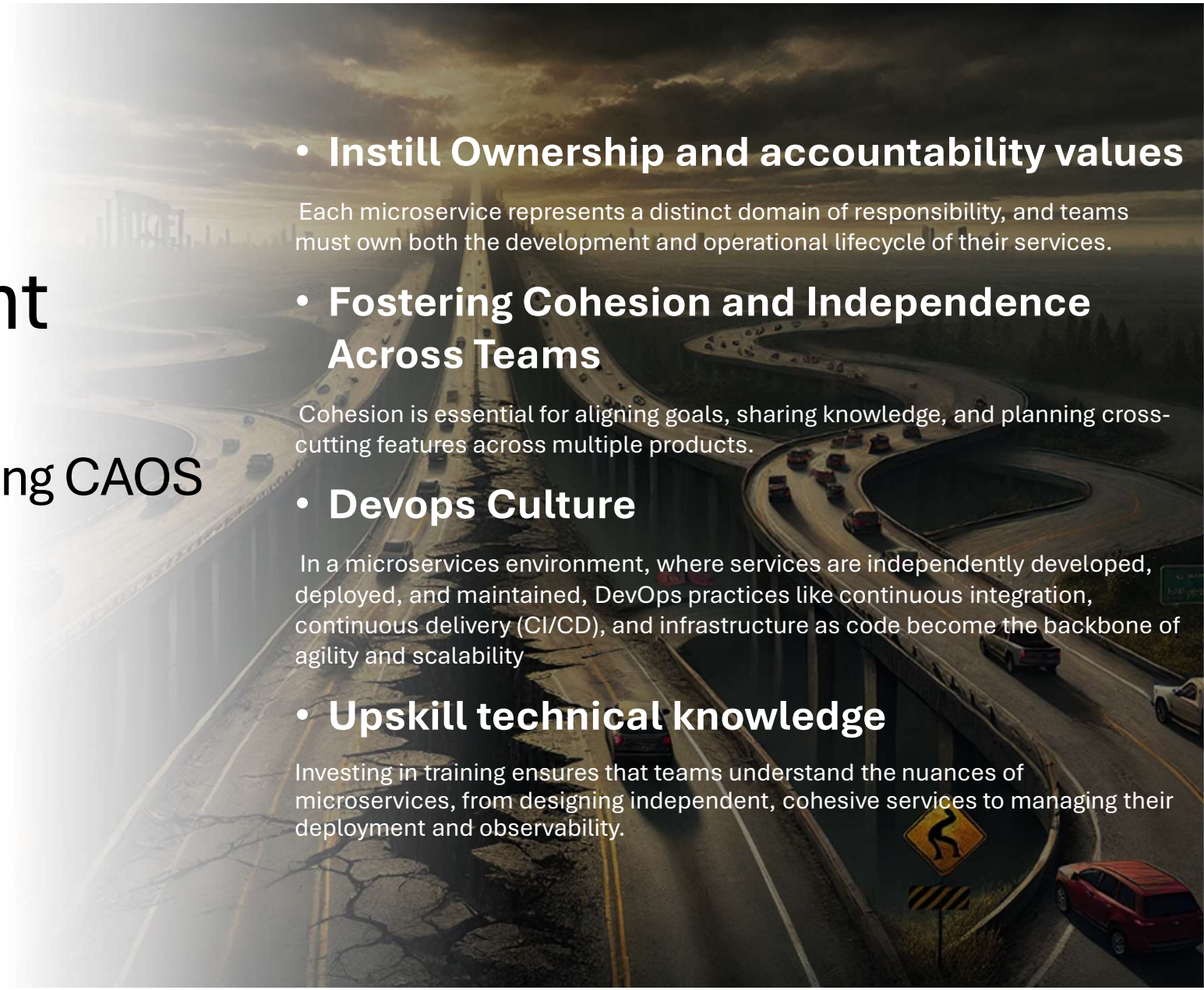
Increased cognitive load, unclear ownership, and recurring issues lower morale and lead to burnout over time.


How to Prevent Problems
Organization onboard
Preparation vs managing caos
Plan & ready to adapt

How to Prevent Problems

How to Prevent Problems

Preparation vs managing CAOS



- **Instill Ownership and accountability values**

Each microservice represents a distinct domain of responsibility, and teams must own both the development and operational lifecycle of their services.

- **Fostering Cohesion and Independence Across Teams**

Cohesion is essential for aligning goals, sharing knowledge, and planning cross-cutting features across multiple products.

- **Devops Culture**

In a microservices environment, where services are independently developed, deployed, and maintained, DevOps practices like continuous integration, continuous delivery (CI/CD), and infrastructure as code become the backbone of agility and scalability

- **Upskill technical knowledge**

Investing in training ensures that teams understand the nuances of microservices, from designing independent, cohesive services to managing their deployment and observability.

Strategy

Preparation vs caos management

- **Cover Use cases with Automated Tests**

Focusing on key use cases ensures critical workflows are protected, reduces the risk of regression, and builds confidence in the system's ability to handle real-world scenarios. This practice is vital for maintaining quality and consistency in an architecture designed for continuous evolution.

- **Add Observability early as possible**

Starting early ensures that as the system grows, visibility remains intact, preventing blind spots and reducing mean time to recovery (MTTR). Observability not only improves debugging but also fosters a culture of proactive monitoring, enabling teams to optimize performance and maintain confidence in the evolving architecture.

- **Evolve Technology before spliting**

By addressing technical debt and introducing scalable, maintainable solutions early, teams can reduce migration risks and streamline the transition. This preparation enables a smoother evolution of the architecture, ensuring that each new service is built on a robust and future-ready stack.

Strategy

Evolve Without breaking

- **First non critical readonly services**

This approach builds confidence, allows for gradual learning, and establishes foundational practices for more critical services later in the migration. teams can experiment with microservices patterns, infrastructure, and deployment pipelines without impacting core functionality.

- **Incrementally Decouple the Database**

This approach allows teams to refactor application logic, identify cross-domain dependencies, and prepare for eventual migration to fully independent databases.

- **Introduce API Gateway**

The goal is to abstract the complexities of managing multiple services, ensuring a seamless experience for consumers while enabling smooth service migration.

YARP or **Ocelot** provide flexibility and scalability

- **Design Patterns Adoption that allows evolution architecture**

Patterns like the **Strangler Fig** enable incremental migration from monoliths to microservices, while **Event-Driven Architecture** decouples services for scalability and flexibility. Implementing a **Proxy Pattern** or API Gateway ensures smooth transitions by abstracting underlying complexities.

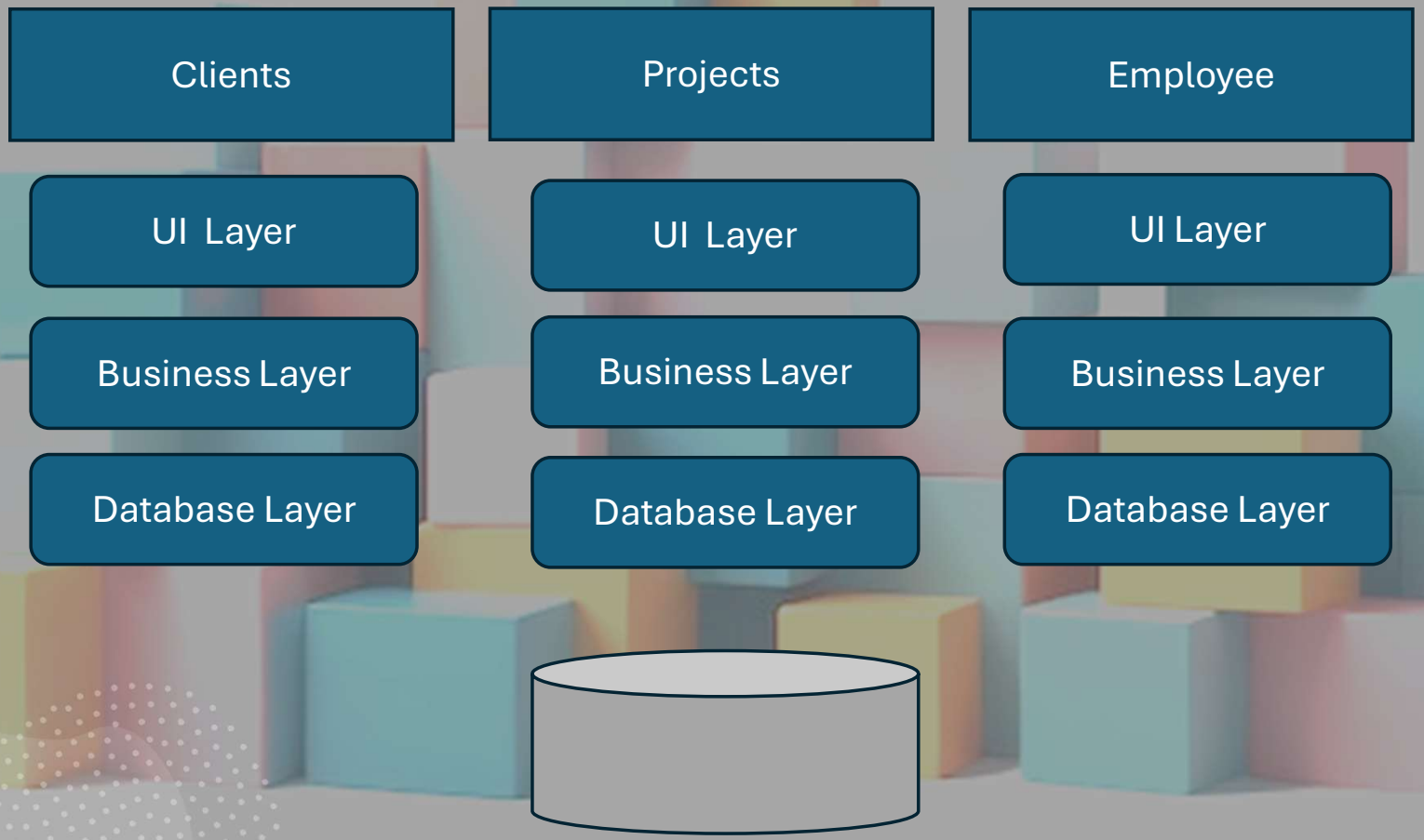
Use Cases

Use Case Modular Monolith

- CRUD Client
- Manage Client Account

- CRUD Project
- Project Management

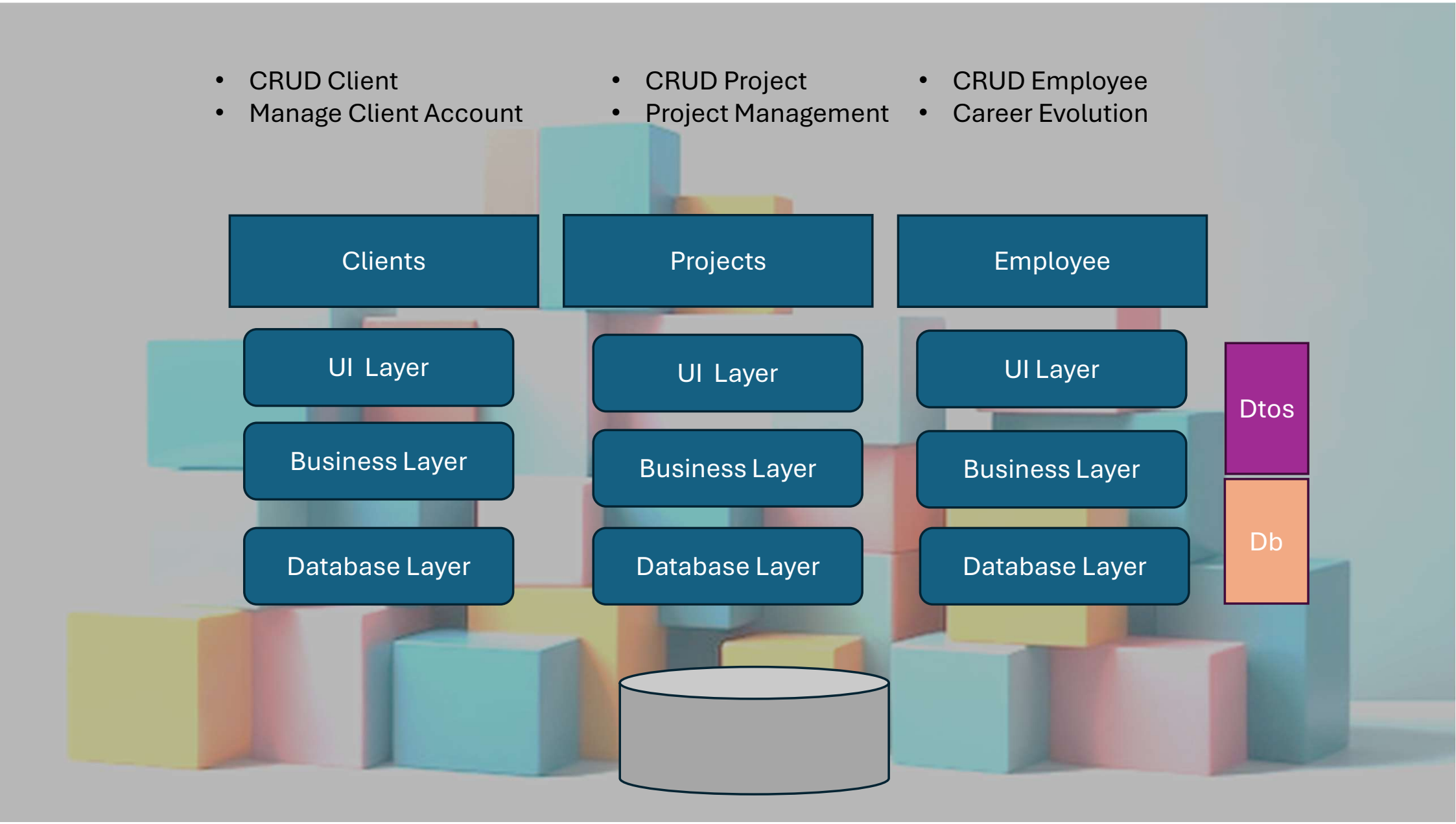
- CRUD Employee
- Career Evolution

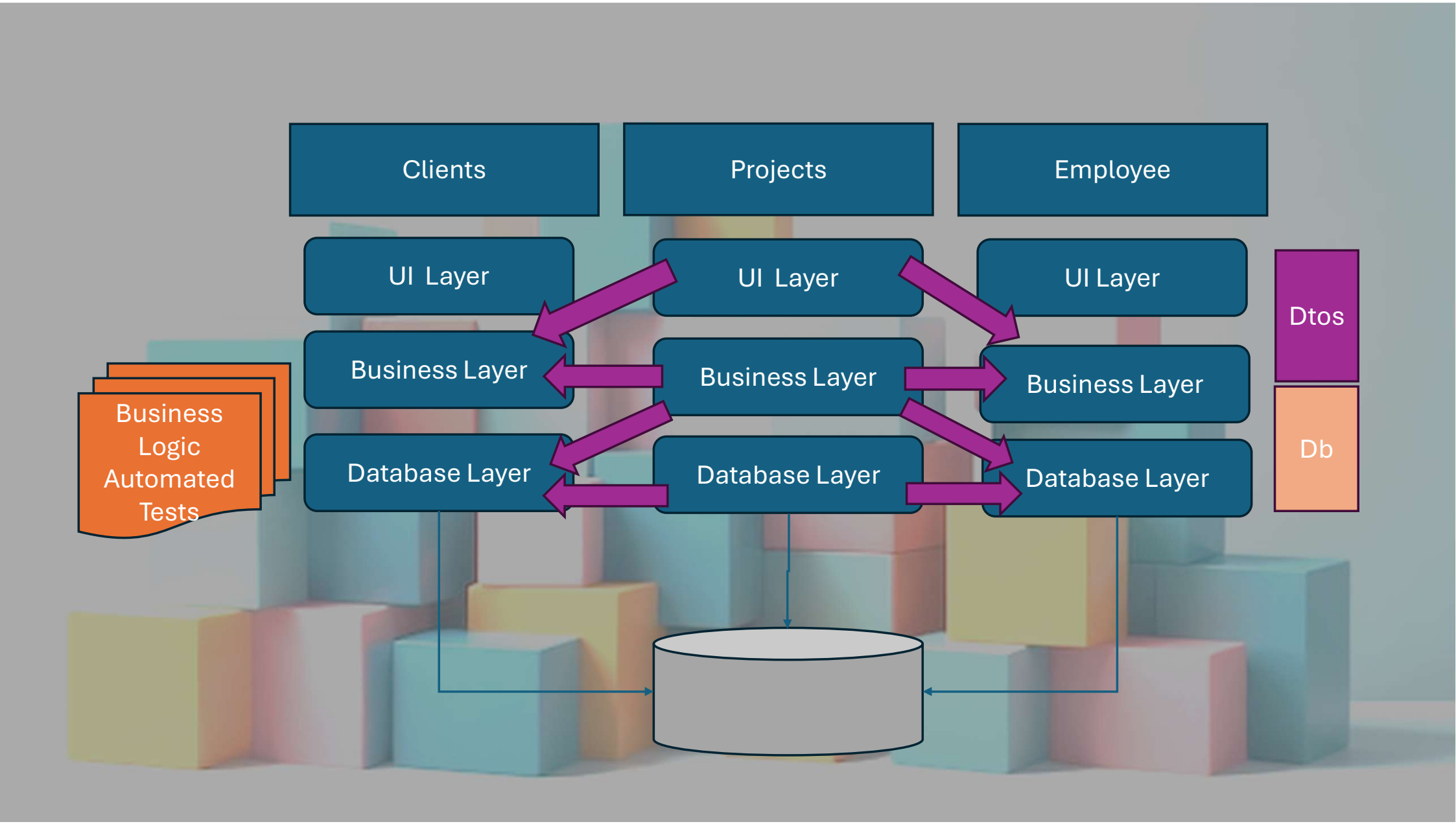


- CRUD Client
- Manage Client Account

- CRUD Project
- Project Management

- CRUD Employee
- Career Evolution

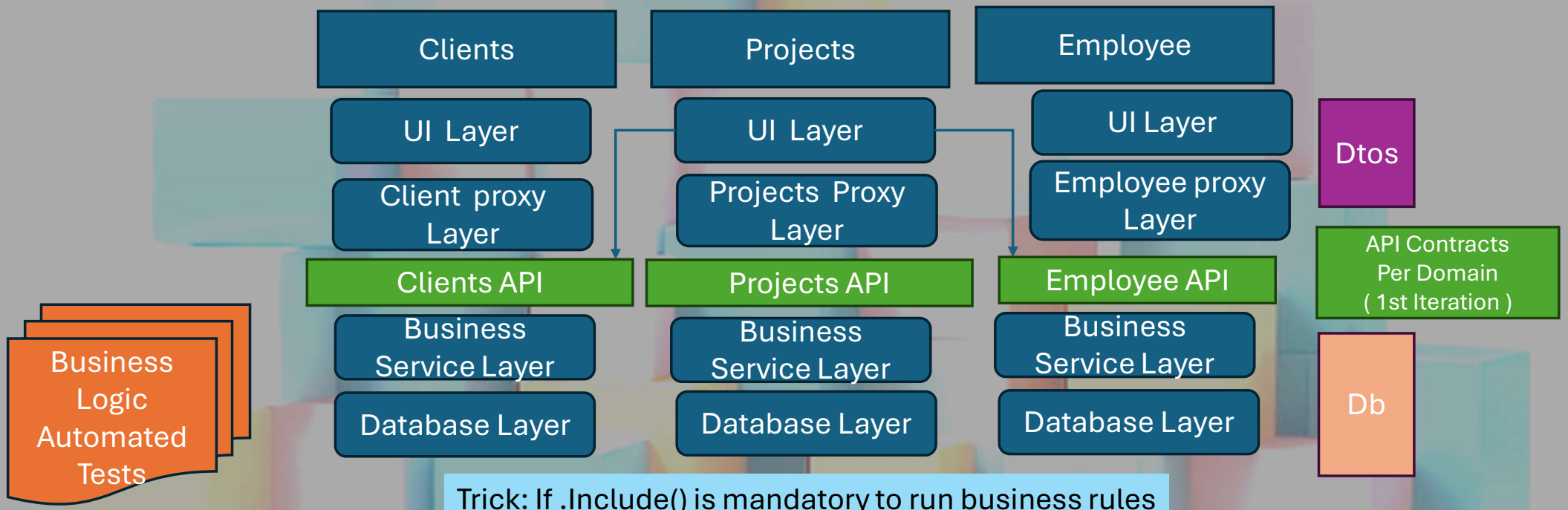



Clients
Projects
Employee
UI Layer
UI Layer
UI Layer
Business Layer
Business Layer
Business Layer
Database Layer
Database Layer
Database Layer
Business Logic Automated Tests
Dtos
Db

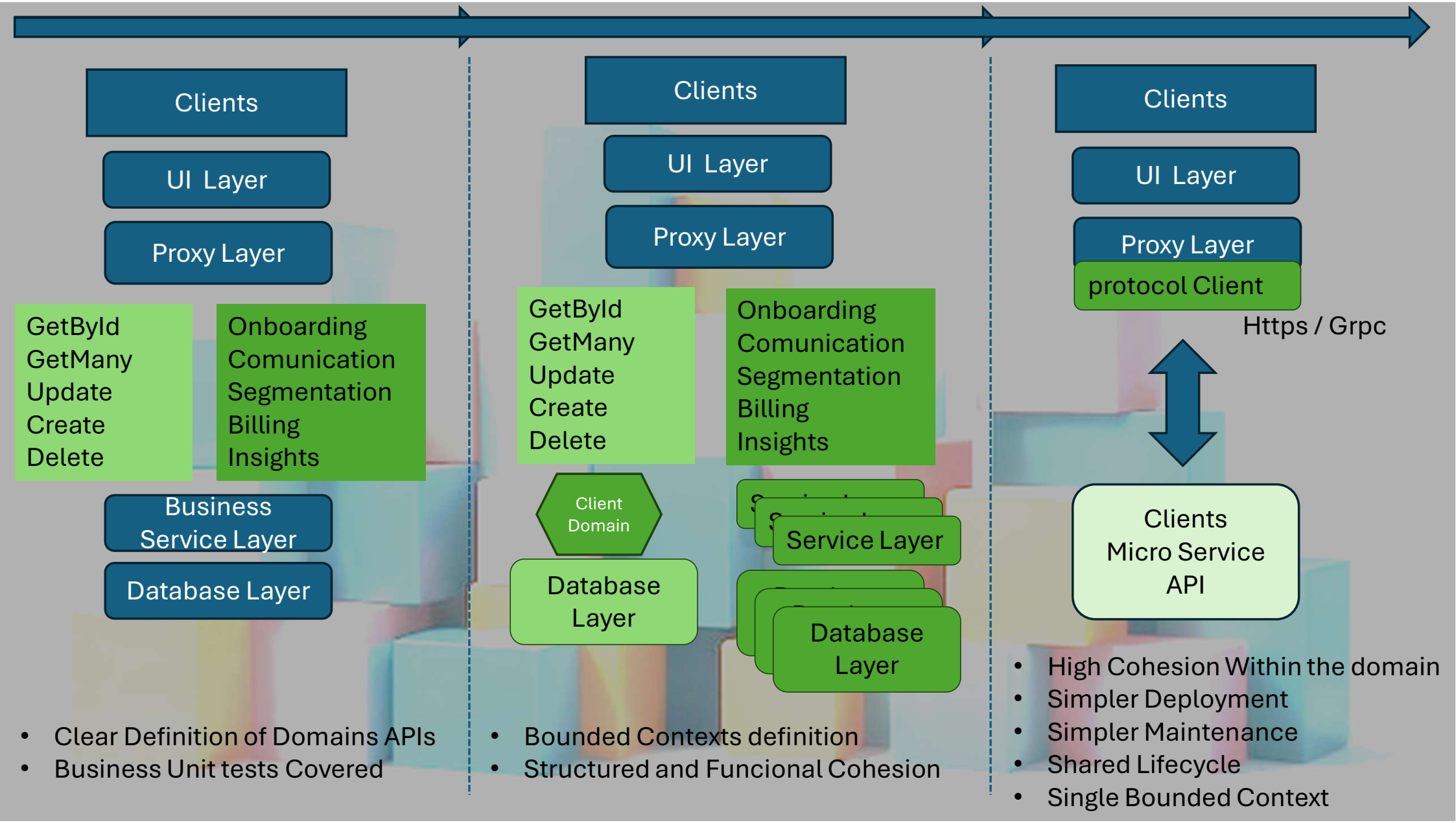
Define Clearn Bounds to each Domain

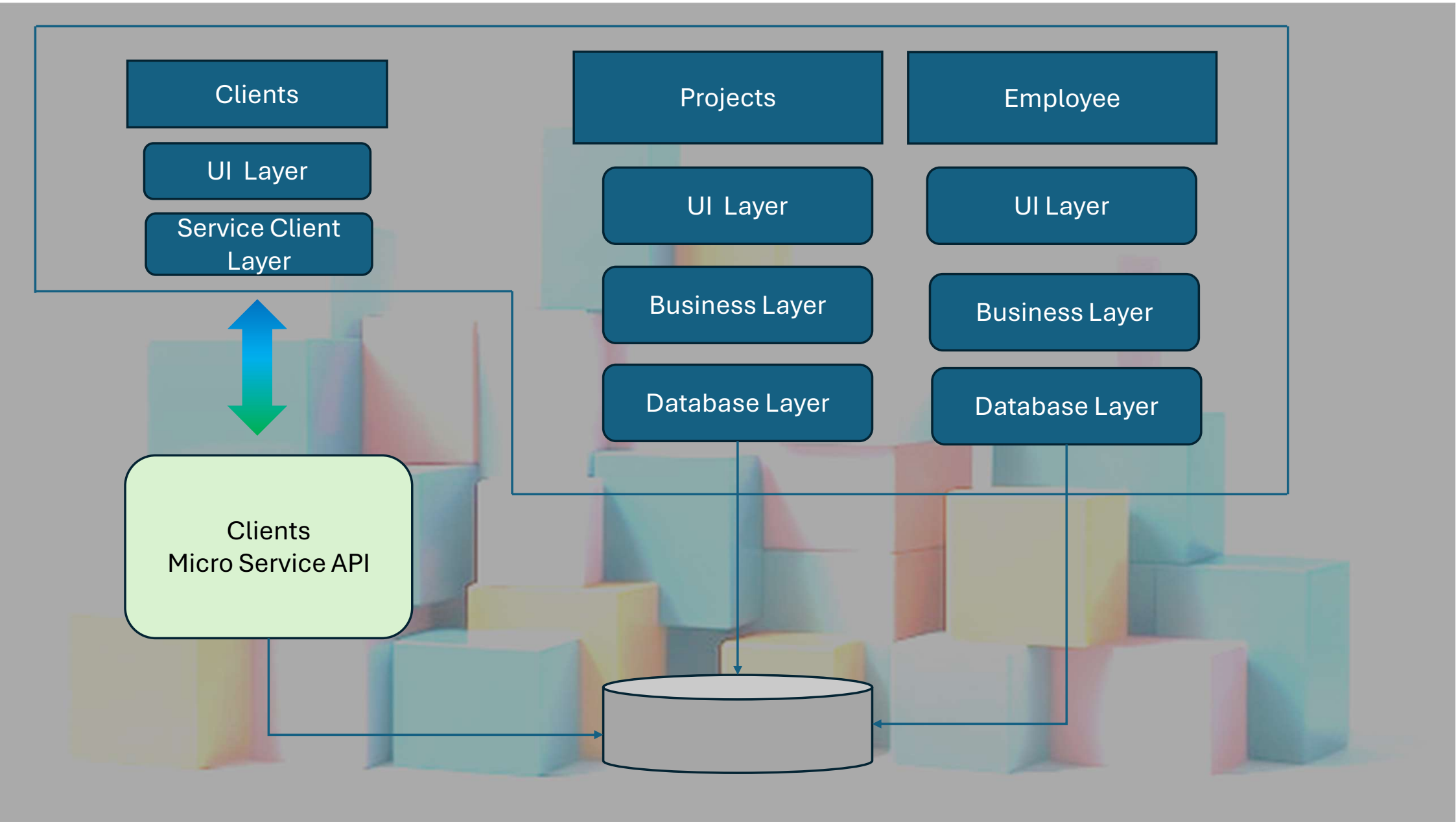
- Domain Driven Design
  - Unbiguitus Language
  - Bounded Context, Aggregates & Entities

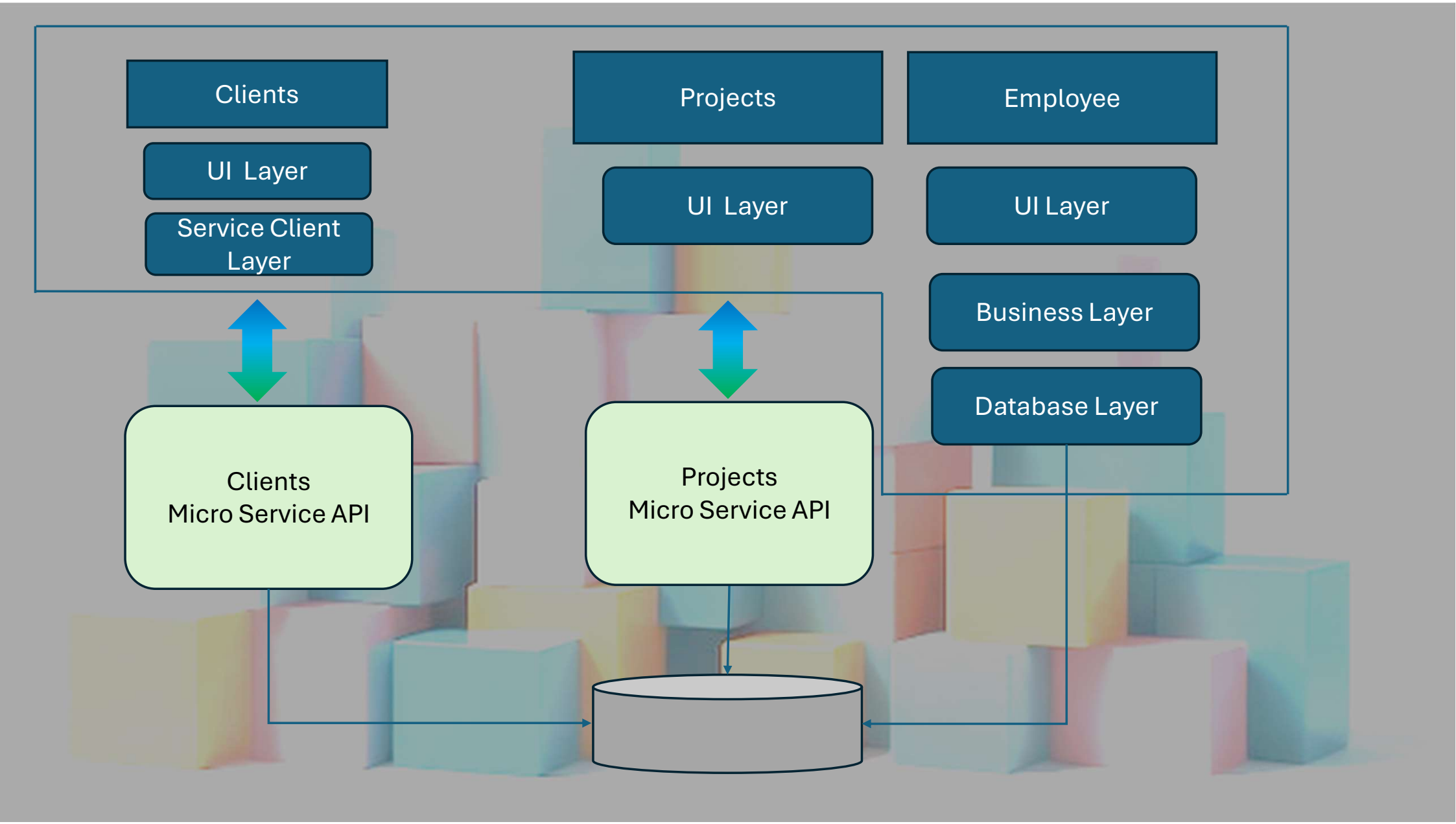
- Strangler Fig
  - Define APIs
  - Remove Cross Reference between domains
  - 1st Iteration we can maintain database as IS

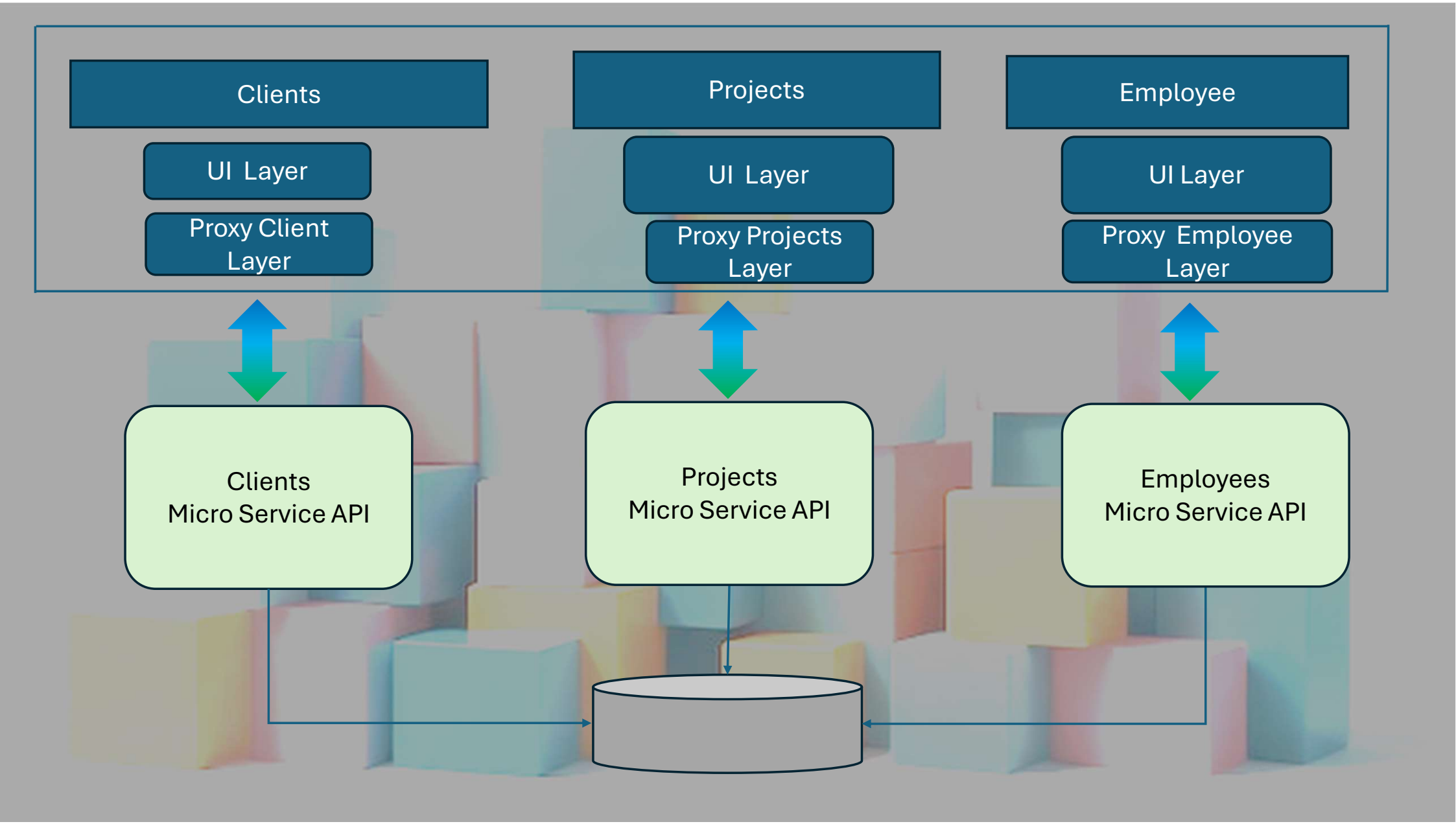


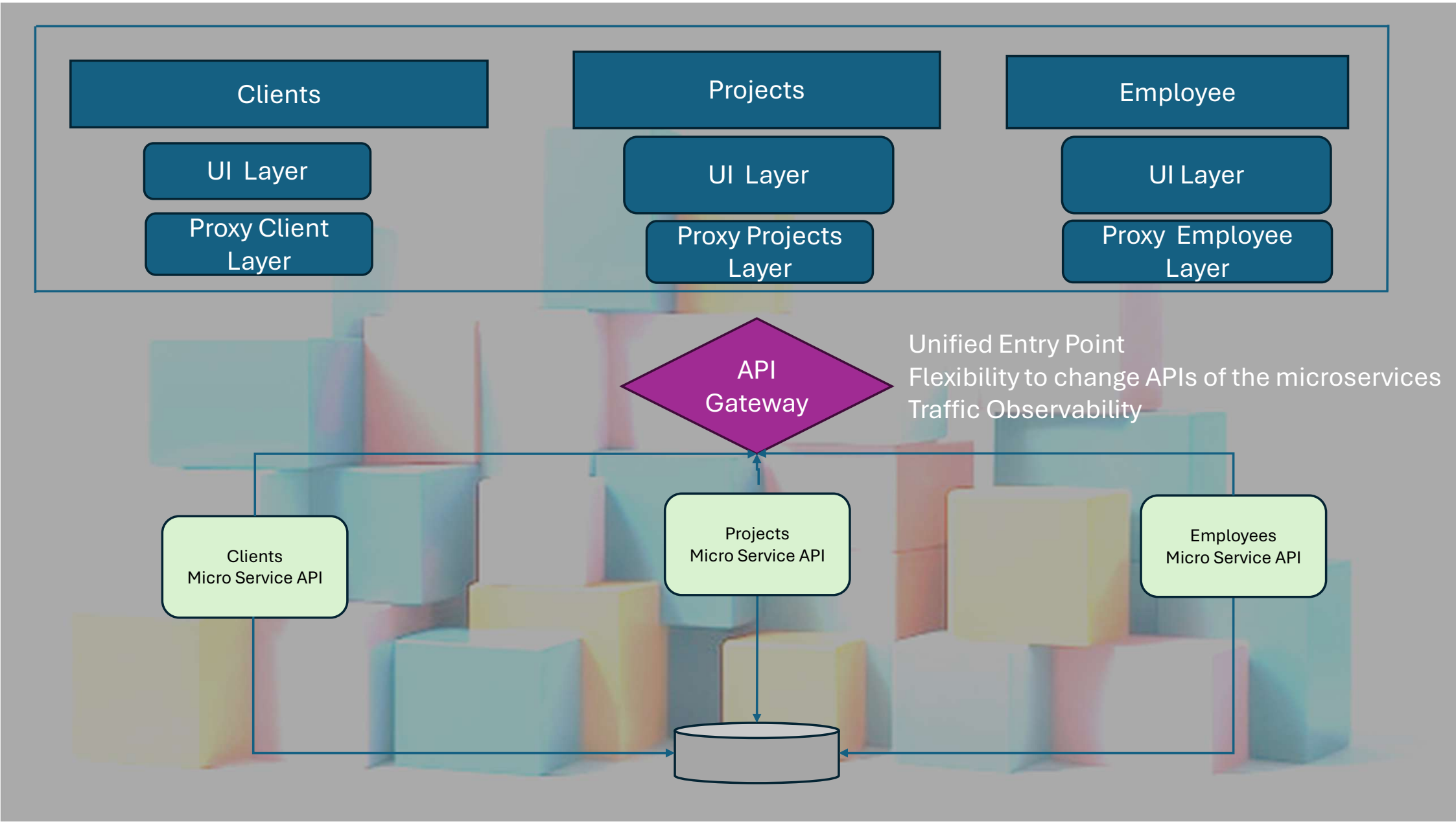
Trick: If .Include() is mandatory to run business rules it means High Coesion between such entities

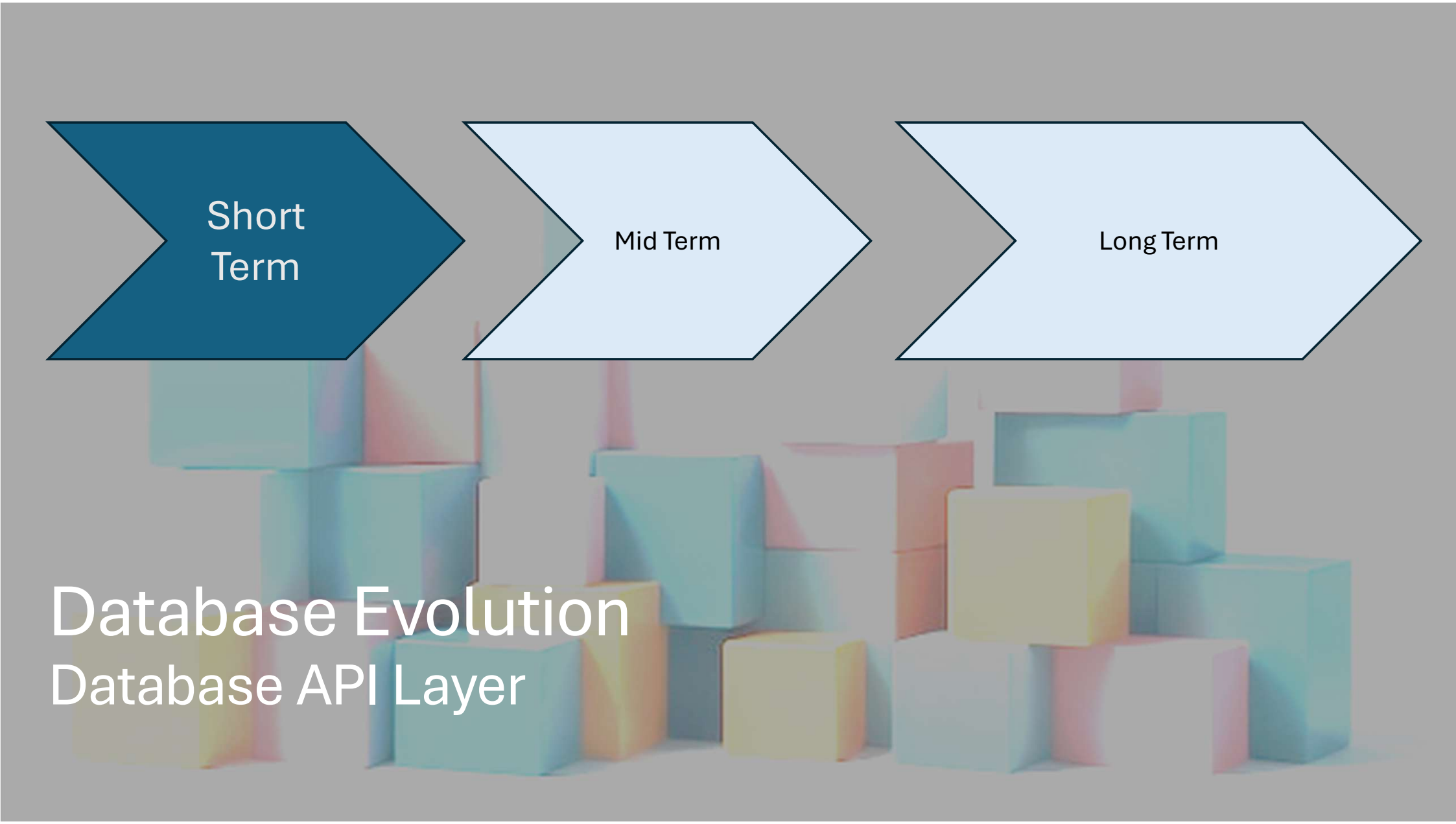

Clients
UI Layer
Proxy Layer
GetById
GetMany
Update
Create
Delete
Onboarding
Comunication
Segmentation
Billing
Insights
Business Service Layer
Database Layer
Clients
UI Layer
Proxy Layer
GetById
GetMany
Update
Create
Delete
Onboarding
Comunication
Segmentation
Billing
Insights
Client Domain
Service Layer
Database Layer
Database Layer
Clients
UI Layer
Proxy Layer
protocol Client
Https / Grpc
Clients
Micro Service
API
• Clear Definition of Domains APIs
• Business Unit tests Covered
• Bounded Contexts definition
• Structured and Funcional Cohesion
• High Cohesion Within the domain
• Simpler Deployment
• Simpler Maintenance
• Shared Lifecycle
• Single Bounded Context

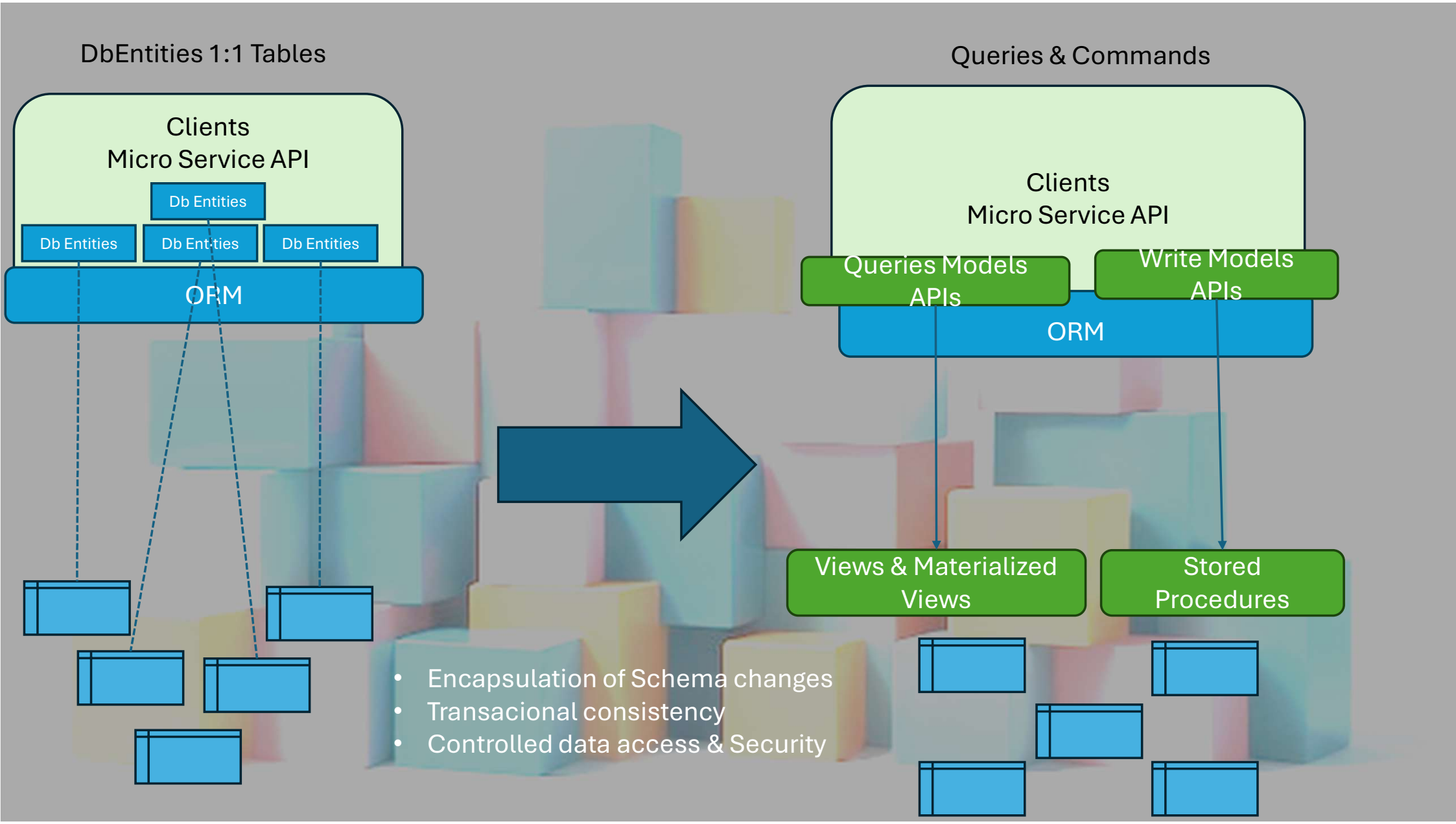

Clients
UI Layer
Service Client Layer
Projects
UI Layer
Business Layer
Database Layer
Employee
UI Layer
Business Layer
Database Layer
Clients
Micro Service API

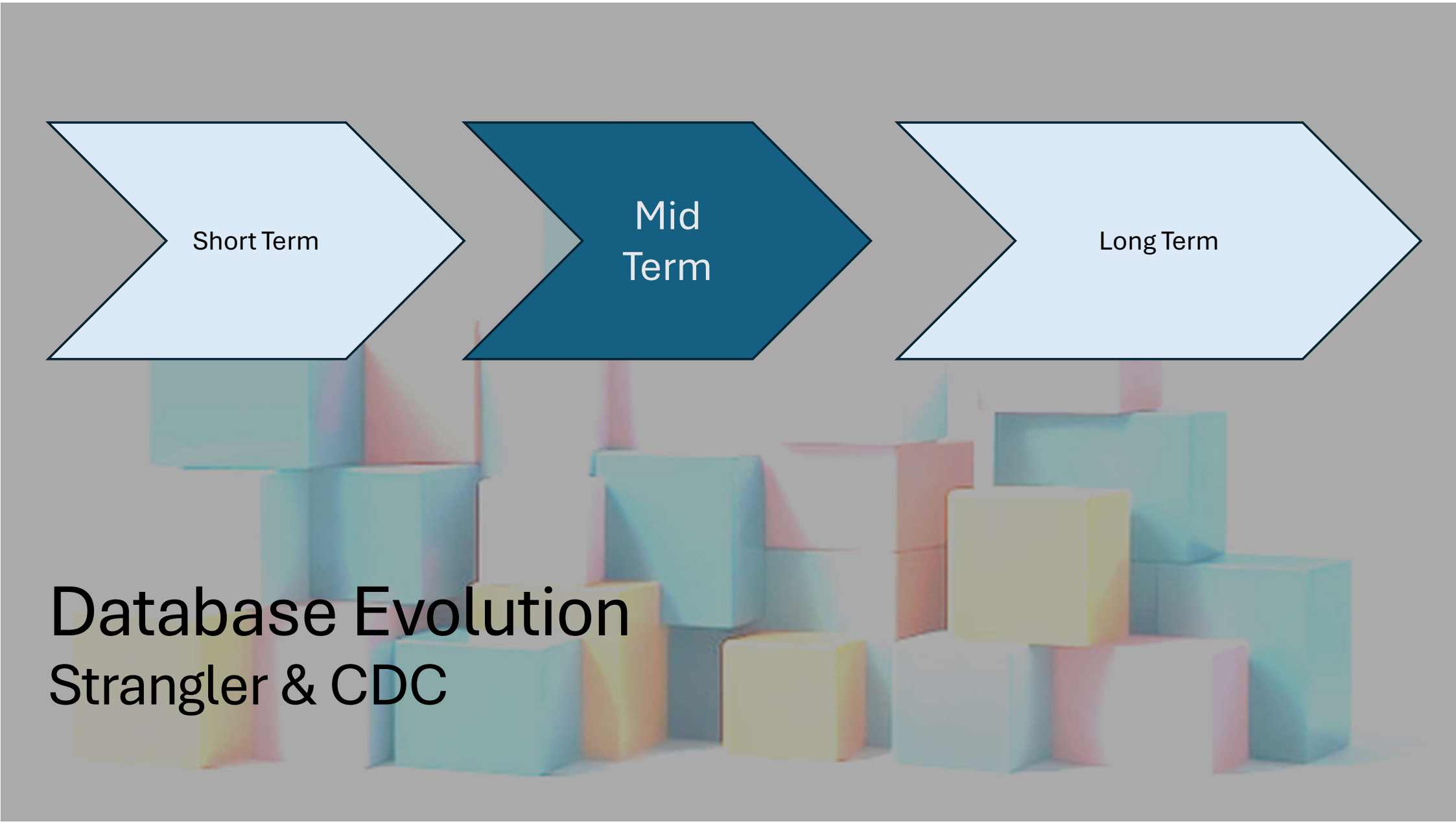

Clients
UI Layer
Service Client Layer
Projects
UI Layer
Employee
UI Layer
Business Layer
Database Layer
Clients Micro Service API
Projects Micro Service API


Clients
UI Layer
Proxy Client Layer
Projects
UI Layer
Proxy Projects Layer
Employee
UI Layer
Proxy Employee Layer
Clients Micro Service API
Projects Micro Service API
Employees Micro Service API


Clients
UI Layer
Proxy Client Layer
Projects
UI Layer
Proxy Projects Layer
Employee
UI Layer
Proxy Employee Layer
API Gateway
Unified Entry Point
Flexibility to change APIs of the microservices
Traffic Observability
Clients Micro Service API
Projects Micro Service API
Employees Micro Service API


Short Term
Mid Term
Long Term
Database Evolution
Database API Layer


DbEntities 1:1 Tables
Clients
Micro Service API
Db Entities
Db Entities
Db Entities
Db Entities
ORM
Queries & Commands
Clients
Micro Service API
Queries Models
APIs
Write Models
APIs
ORM
Views & Materialized
Views
Stored
Procedures
• Encapsulation of Schema changes
• Transacional consistency
• Controlled data access & Security

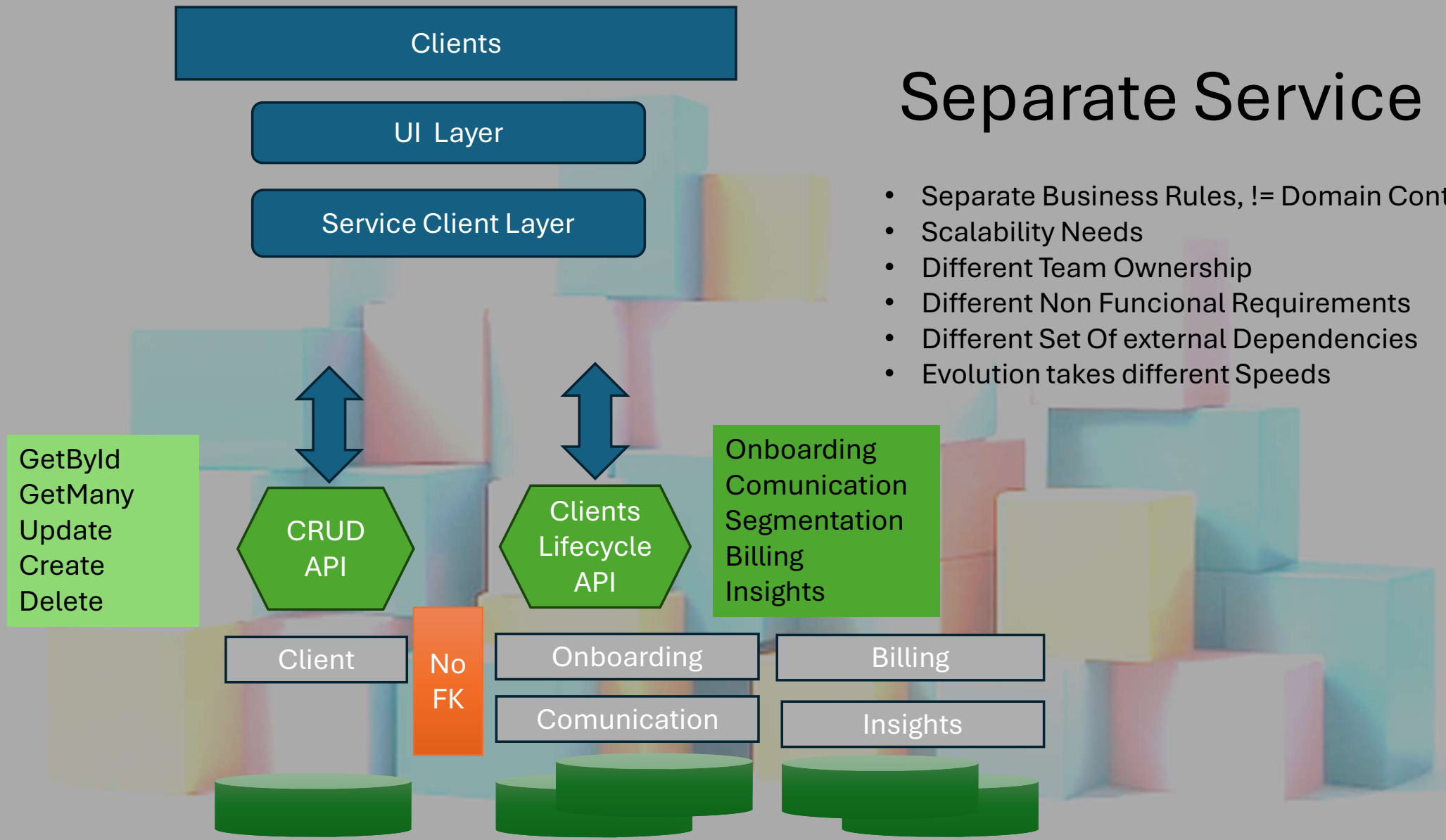


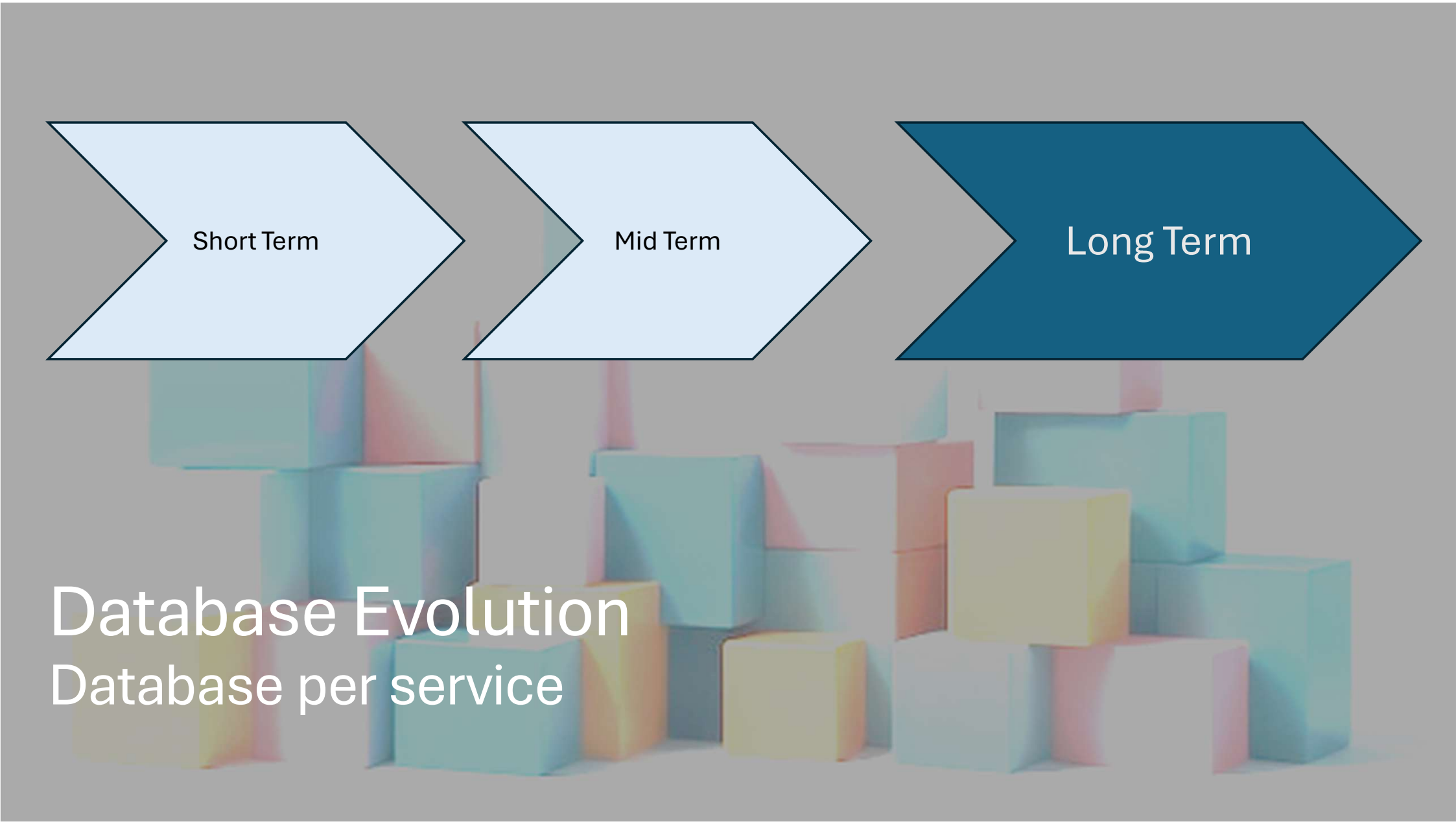
Database Evolution

Strangler & CDC

Separate Service

- Separate Business Rules, != Domain Contexts
- Scalability Needs
- Different Team Ownership
- Different Non Funcional Requirements
- Different Set Of external Dependencies
- Evolution takes different Speeds



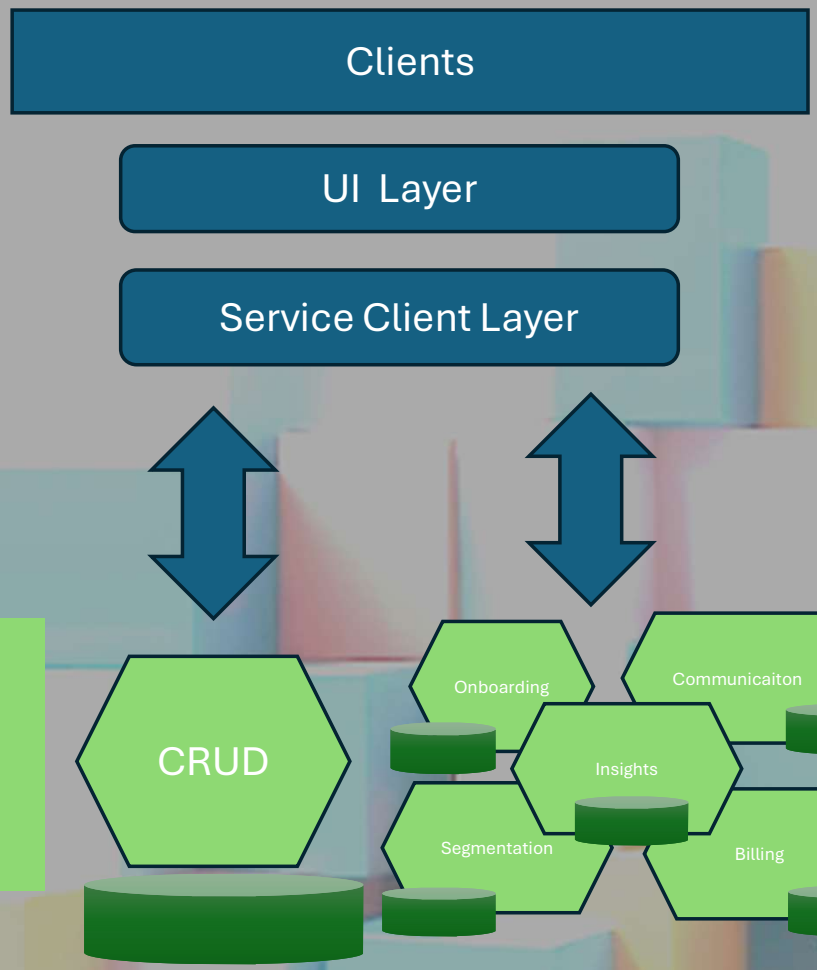


Database Evolution

Database per service

Separate even more

- Separate Business Rules, != Domain Contexts
- Scalability Needs
- Different Team Ownership
- Different Non Funcional Requirements
- Evolution takes different Speeds

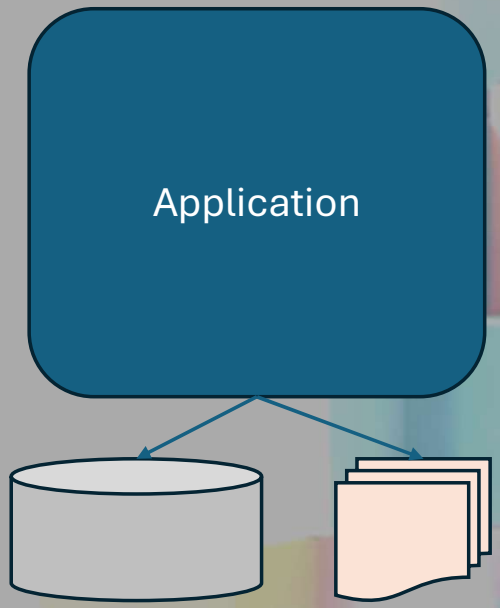


GetById
GetMany
Update
Create
Delete

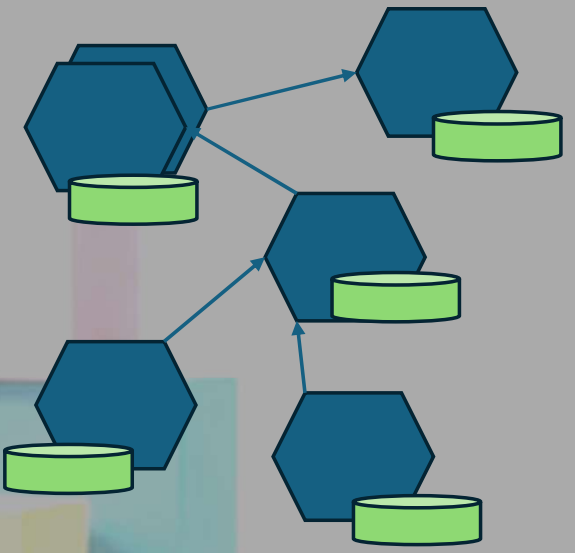
Onboarding
Comunication
Segmentation
Billing
Insights

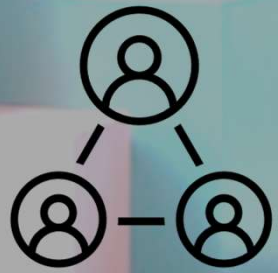
Use Case

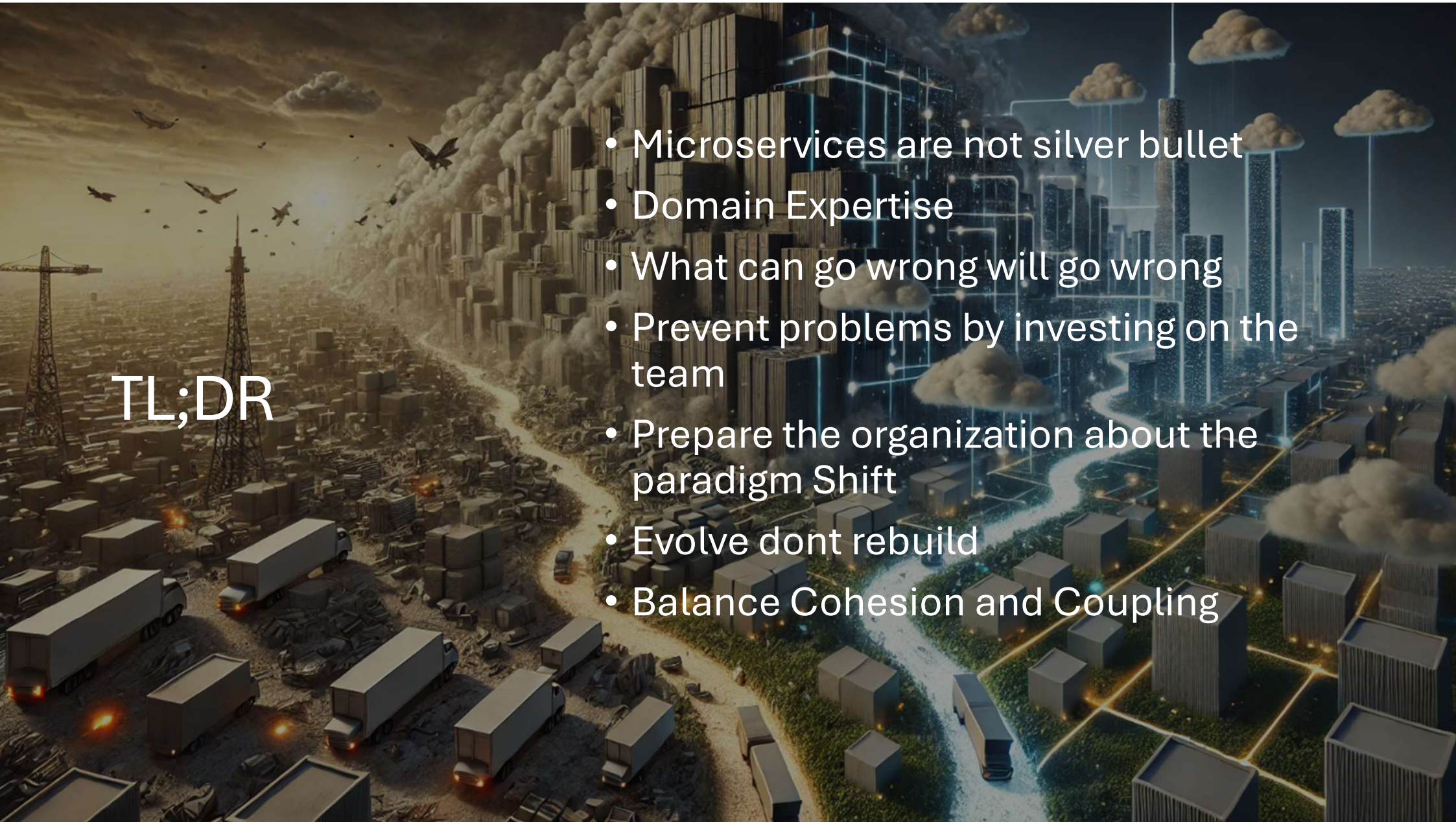
Summary



- Real world is never linear
- Technical decisions are tied to organizational realities
- Balance Flexibility and complexity
- Architecture are tradeoff decisions to serve the business






TL;DR
• Microservices are not silver bullet
• Domain Expertise
• What can go wrong will go wrong
• Prevent problems by investing on the team
• Prepare the organization about the paradigm Shift
• Evolve dont rebuild
• Balance Cohesion and Coupling