Evolutionary Architecture Topologies

Coupling

- A necessary evil
- **Appropriate coupling**
  - how to identify which dimensions of the architecture should be coupled to provide maximum benefit with minimal overhead and cost

To identify coupling:

1. Connascence
2. Bounded Context

Connascence

Two components are connascent if a change in one would require the other to be modified in order to maintain the overall correctness of the system.

1. Static connascence
2. Dynamic connascence

Static connascence

Source code–level coupling (as opposed to execution-time coupling)

1. **Connascence of Name (CoN)**

   Multiple components must agree on the name of an entity.

2. **Connascence of Type (CoT)**

   Multiple components must agree on the type of an entity.

Static connascence

3. **Connascence of Meaning (CoM)** or **Connascence of Convention (CoC)**

Multiple components must agree on the meaning of particular values.

e.g. if we set `delete` field as either 0 or 1, we need to agree on the meaning of 0(true) and 1(false)

- Enum IS_DELETED: 1
- IS_NOT_DELETED: 0

4. **Connascence of Position (CoP)**

Multiple components must agree on the order of values.

e.g. the order of passing arguments into a function

Static connascence

5. **Connascence of Algorithm (CoA)**

Multiple components must agree on a particular algorithm.

e.g. a security hashing algorithm that must run on both the server and client and produce identical results to authenticate the user.

Dynamic connascence

Analyzes calls at runtime

1. **Connascence of Execution (CoE)**

The order of execution of multiple components is important.

e.g. set email header, then send email

2. **Connascence of Timing (CoT)**

The timing of the execution of multiple components is important.

e.g. race condition

Dynamic connascence

3. **Connascence of Values (CoV)**

This occurs when several values relate to one another and must change together.

e.g. transactions in distributed systems, values need to be update across all the databases

4. **Connascence of Identity (CoI)**

This occurs when multiple components must reference the same entity.

e.g. two independent components that must share and update a common data structure, such as a distributed queue.

Connascence

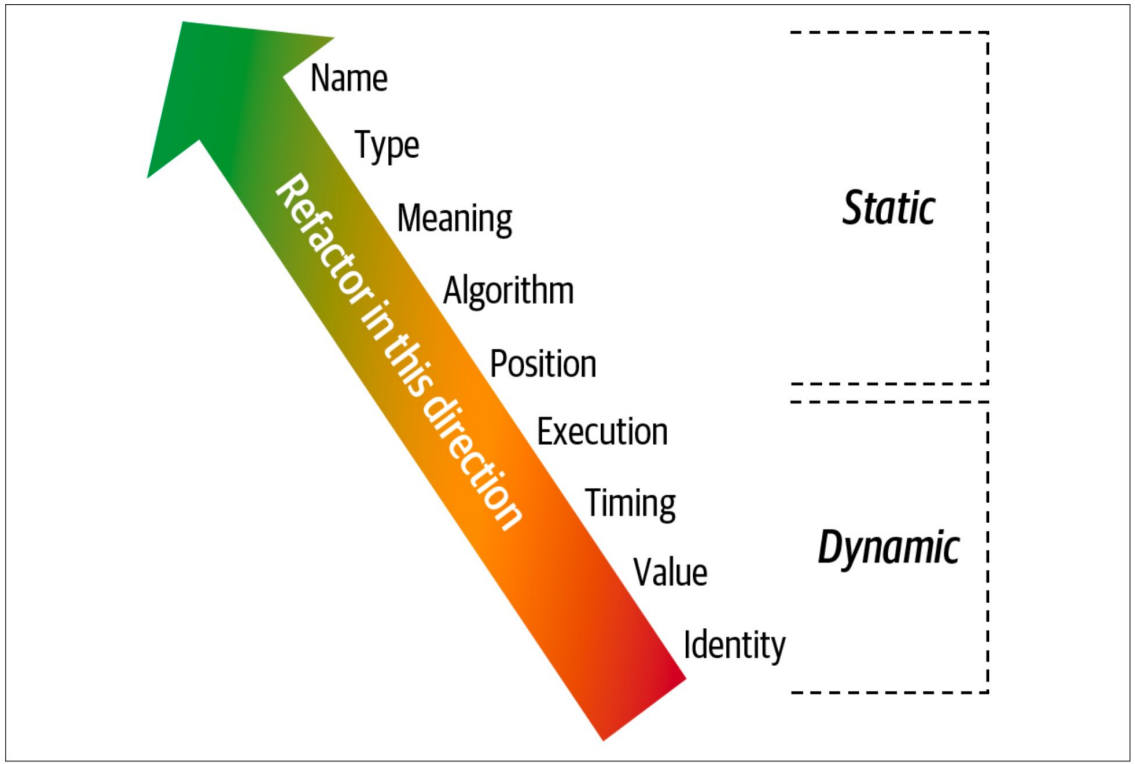

Figure 5-1. The strength of connascence provides a good refactoring guide

Rule of Degree: convert strong forms of connascence into weaker forms of connascence.

Rule of Locality: as the distance between software elements increases, use weaker forms of connascence.

Bounded Context

- **Domain-driven design** (DDD) : modeling technique that allows for organized decomposition of complex problem domains
- Everything related to the domain is visible internally but opaque to other bounded contexts.
- e.g. instead of creating a unified Customer class across the entire organization, each problem domain can create their own and reconcile differences at integration points

Architecture quantum

An **independently deployable** artifact with **high functional cohesion, high static coupling,** and **synchronous dynamic coupling.**

e.g: a well-formed microservice within a workflow.

Static coupling

- Represents how static dependencies resolve within the architecture via contracts

Dynamic coupling

- Represents how quanta communicate at runtime, either synchronously or asynchronously

High Static Coupling

Elements inside the architecture quantum are tightly wired together.

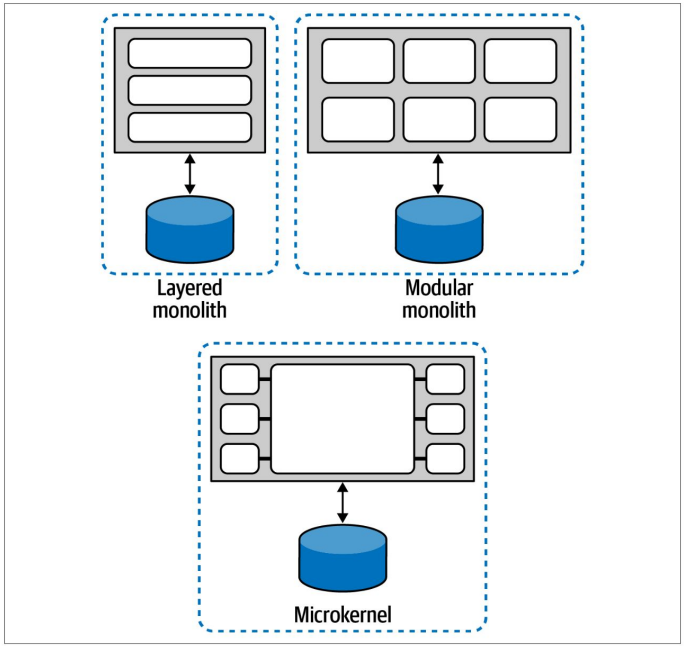

Figure 5-2. Monolithic architectures always have a quantum of one

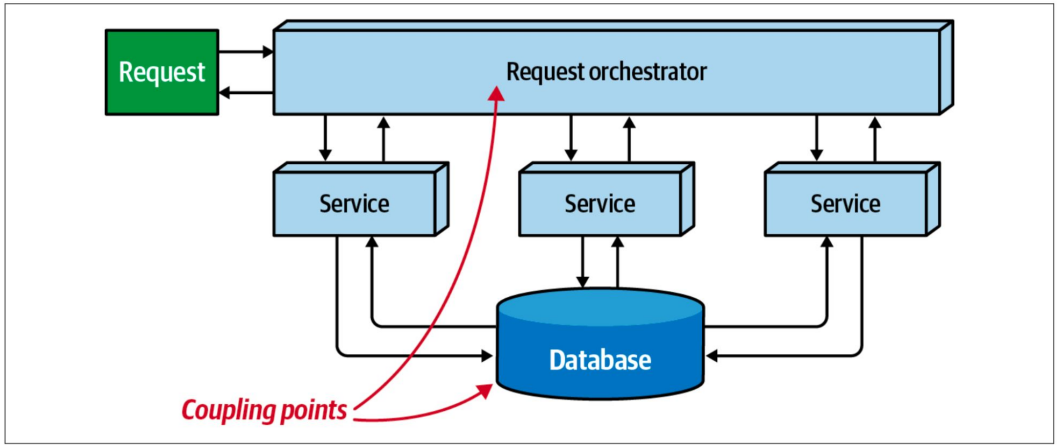

Figure 5-3. A mediated event-driven architecture has a single architecture quantum

Distributed architectures create the possibility of multiple quanta but don’t necessarily guarantee it.

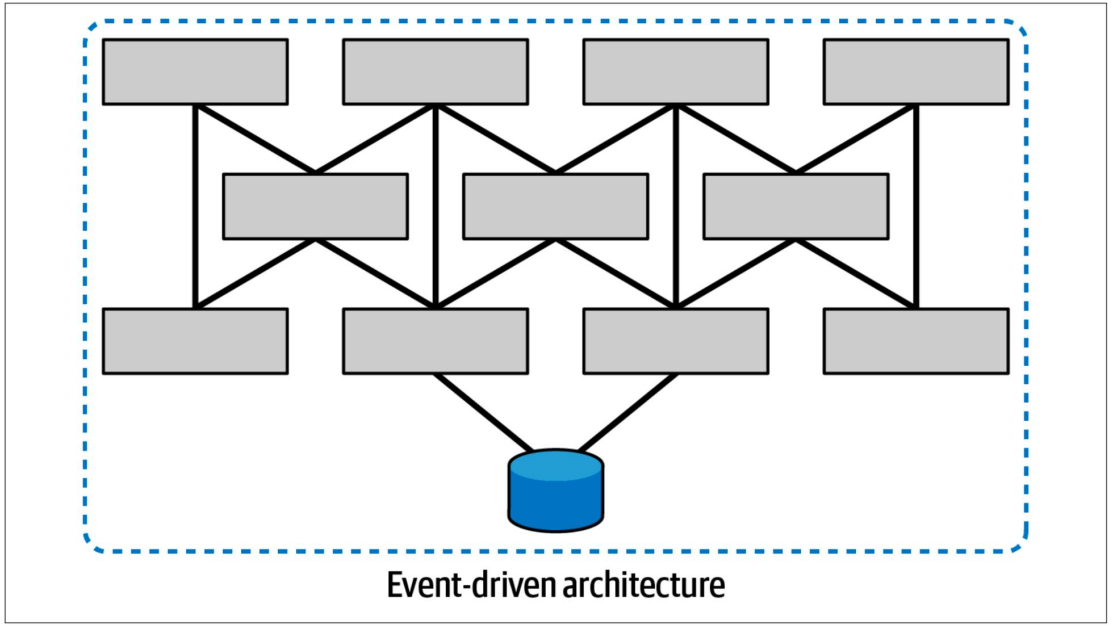

Figure 5-4. Even a distributed architecture such as a broker-style event-driven architecture can be a single quantum

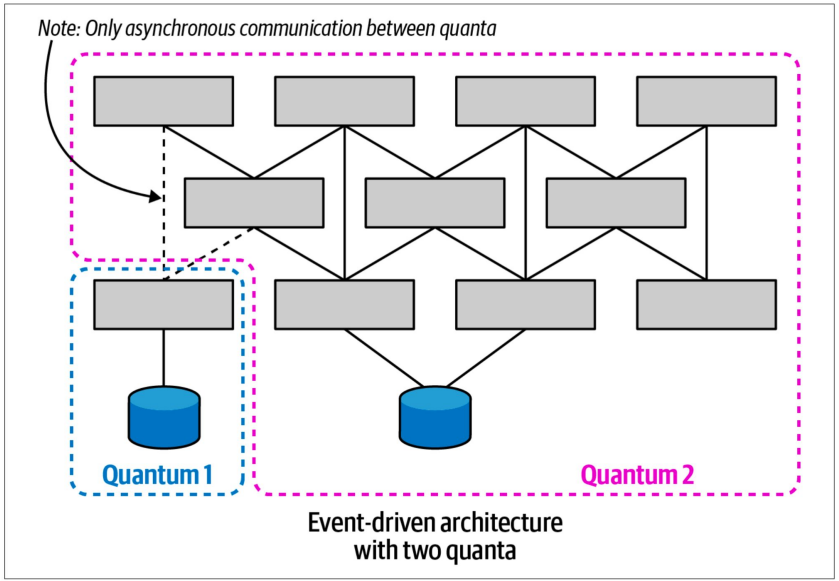

Figure 5-5. An event-driven architecture with two quanta

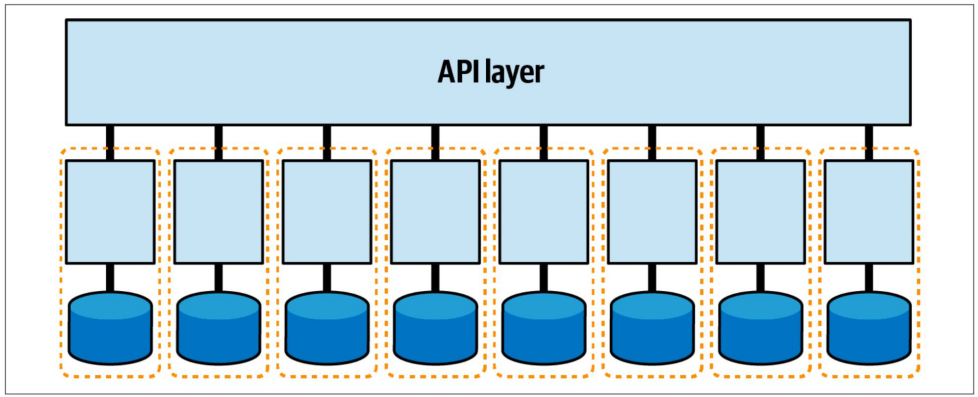

Figure 5-6. Microservices may form their own quanta

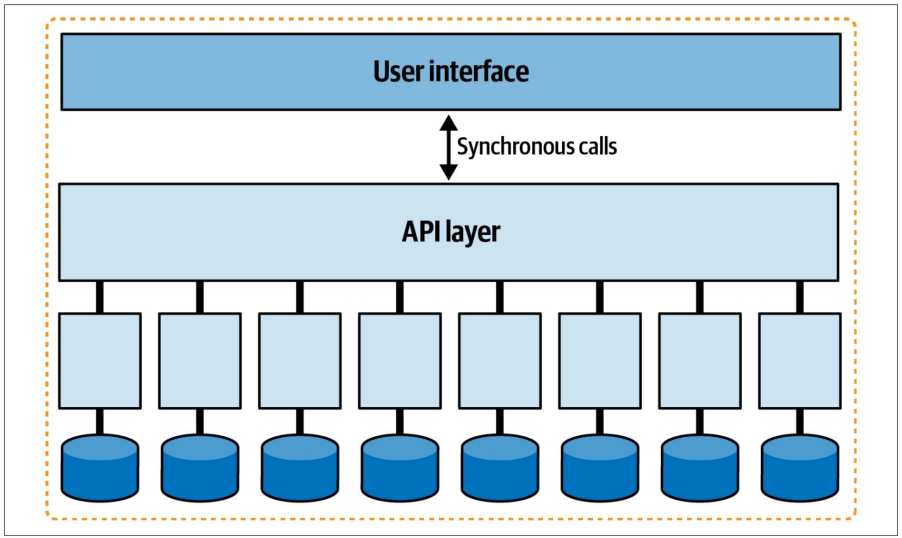

Figure 5-7. A tightly coupled user interface can reduce a microservices architecture quantum to one

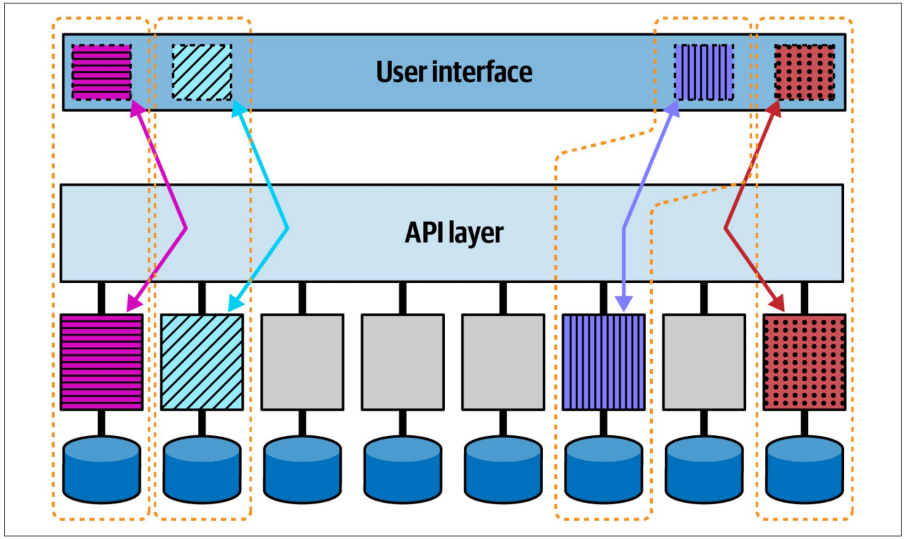

Figure 5-8. In a micro-frontend architecture, each service + user interface component forms an architecture quantum

Any coupling point in an architecture can create static coupling points from a quantum standpoint

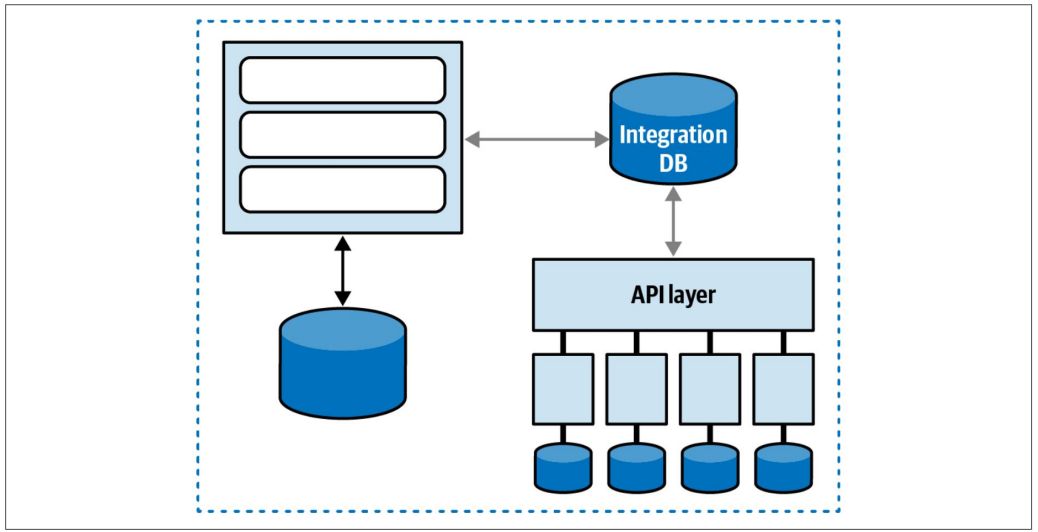

Figure 5-9. A shared database forms a coupling point between two systems, creating a single quantum

Dynamic Quantum Coupling

synchronous coupling at runtime

- the behavior of architecture quanta as they interact with one another to form workflows within a distributed architecture

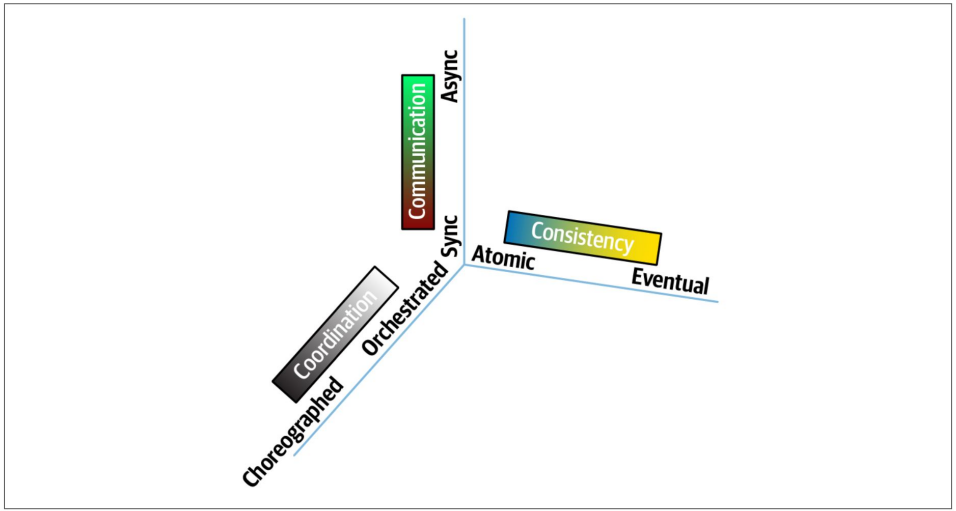

Figure 5-12. The dimensions of dynamic quantum coupling

Contracts

- encompasses all techniques used to “wire together” parts of a system
- including transitive dependencies for frameworks and libraries, internal and external integration points, caches, and any other communication between parts
- loose contracts allows for extremely decoupled systems

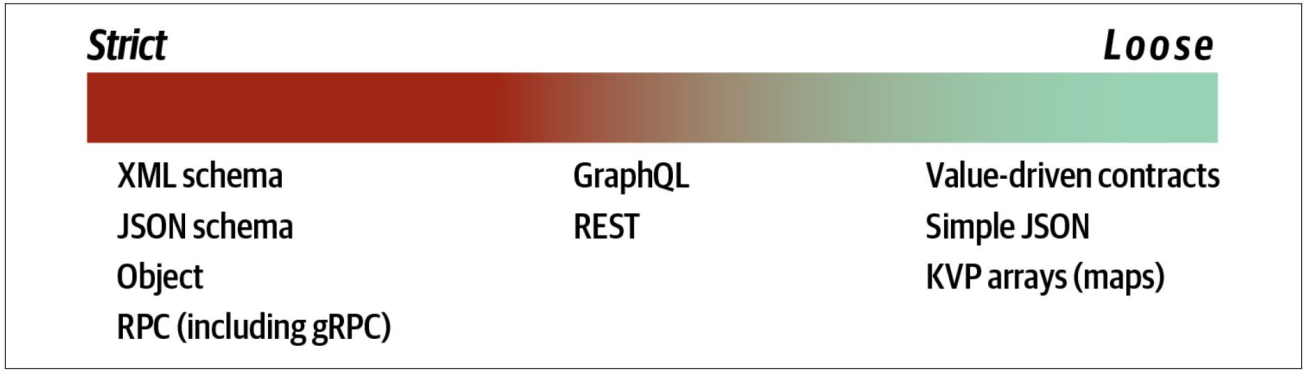

Figure 5-13. The spectrum of contract types from strict to loose

Case Study: Microservices as an Evolutionary Architecture

A microservices architecture defines physical bounded contexts between architectural elements, encapsulating all the parts that might change

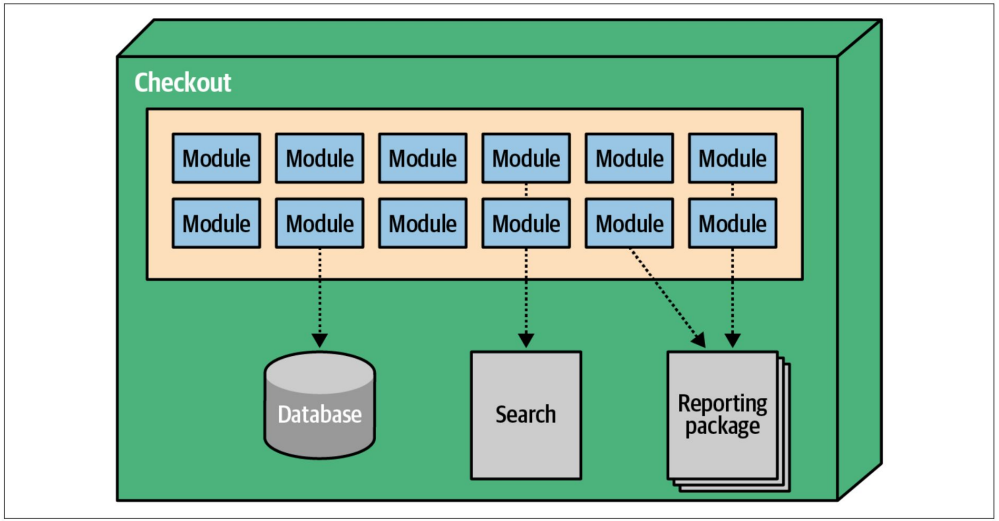

Figure 5-14. The architectural quantum in microservices encompasses the service and all its dependent parts

In a layered architecture, the focus is on the technical dimension, or how the mechanics of the application work

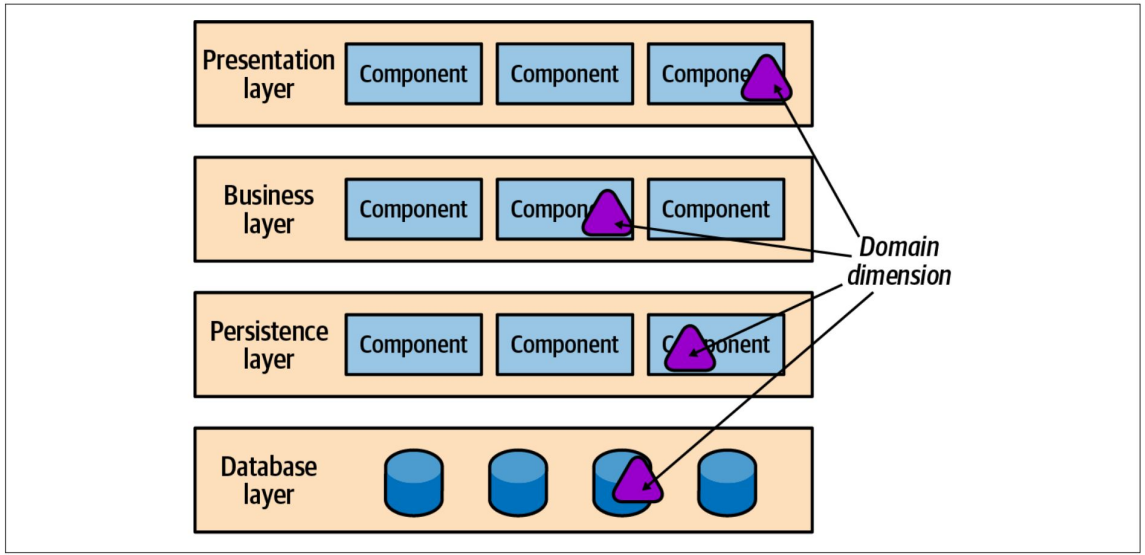

Figure 5-15. The domain dimension is embedded within technical architecture

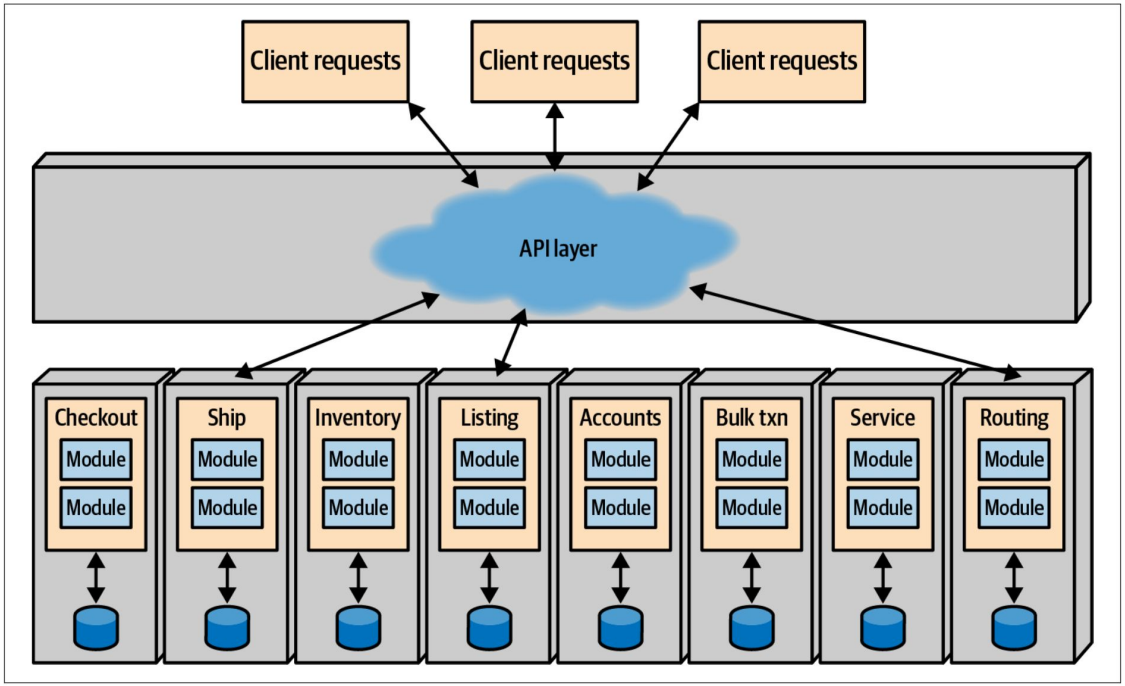

Figure 5-16. Microservices architectures partition across domain lines, embedding the technical architecture

Seven principles of microservices

1. Modeled around the business domain
2. Hide implementation details
3. Culture of automation
4. Highly decentralized
5. Deployed independently
6. Isolate failure
7. Highly observable

Why is microservice a good evolutionary architecture

- Incremental change
  - Each service forms a bounded context around a domain concept, making it easy to make changes that only affect that context
- Guided change with fitness functions
  - Each service has a well-defined boundary, allowing a variety of levels of testing within the service components

Why haven’t developers embraced Microservices before

- Limited Automation in the Past
  - VMs required manual setup, long provisioning times, and lacked automation support.
- Budget Constraints
  - Led to shared resource architectures and workarounds like Enterprise Service Bus (ESB).
- Expensive & Cumbersome Operations
  - Architects designed around operational complexity.
- DevOps & Continuous Delivery Shift
  - Version-controlled infrastructure, automation, and parallel test environments enabled safe deployments.
- Open-Source Adoption
  - Reduced licensing concerns, enabling flexible architectures.
- Domain-Centric Evolution
  - Microservices integrate technical and domain concerns, avoiding architectural rigidity (Big Ball of Mud).

Reuse Patterns

- **SOA Approach**: Encouraged reuse—e.g., a shared Customer service for Checkout & Shipping.
- More reusable code = harder to use due to added complexity.
- Microservices Philosophy: **Prefer duplication over coupling** to maintain independence.
- **Domain Isolation**: Each service maintains its own Customer representation.
- Collaboration Over Consolidation: Services **exchange relevant data instead of sharing entities**.
- Balancing Reuse & Coupling: Use **service meshes** for concerns like logging & monitoring without tightly coupling business logic.

Sidecars and Service Mesh: Orthogonal Operational Coupling

The Sidecar pattern leverages the same concept as hexagonal architecture in that it decouples the domain logic from the technical (infrastructure) logic.

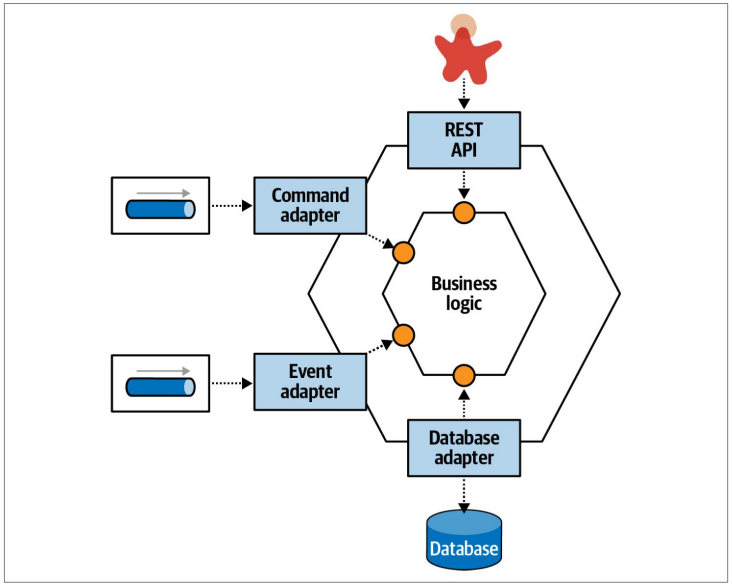

Figure 5-17. The Hexagonal Pattern separated domain logic from technical coupling

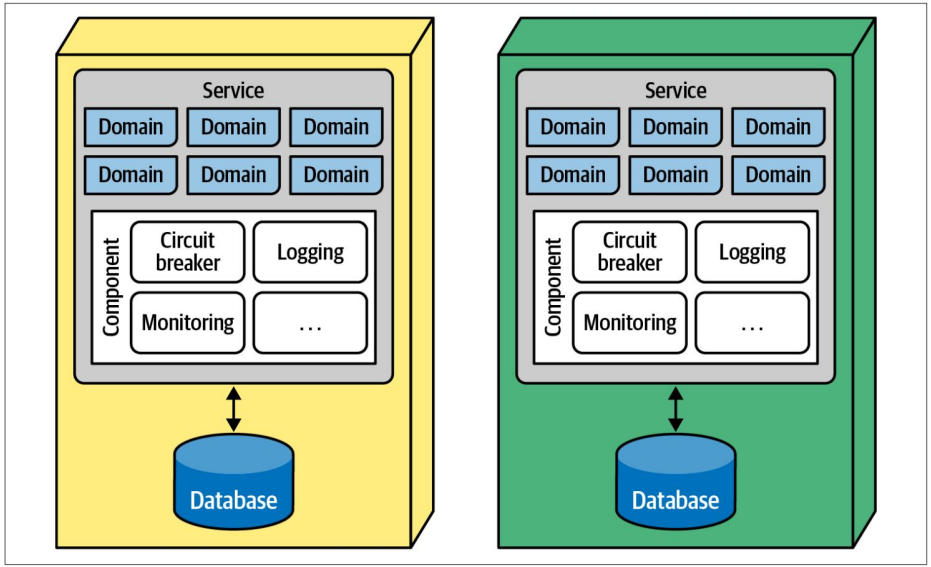

Figure 5-18. Two microservices that share the same operational capabilities

Sidecars and Service Mesh: Orthogonal Operational Coupling

Service mesh enables architects and DevOps to create dashboards, monitoring health and performance, control operational characteristics such as scaling, traffic routing, security enforcement, and fault tolerance

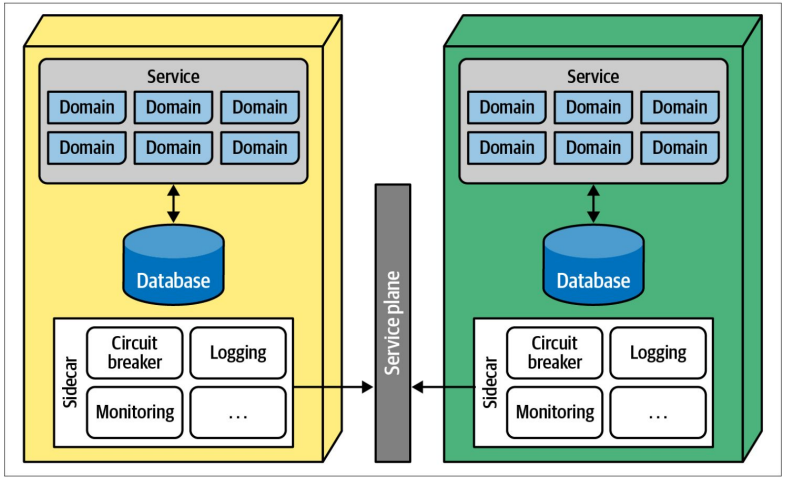

Figure 5-19. When each microservice includes a common component, architects can establish links between them for consistent control

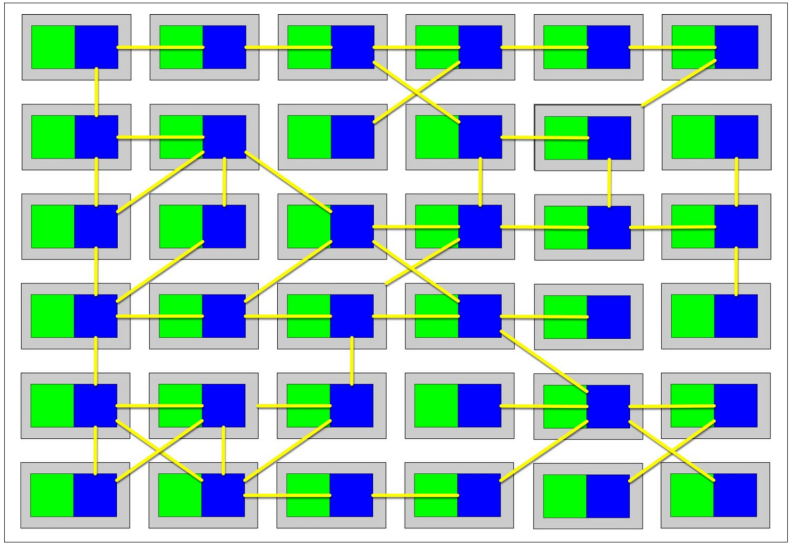

Figure 5-20. A service mesh is a set of operational links between services

Data Mesh: Orthogonal Data Coupling

An approach to sharing, accessing, and managing analytical data in a decentralized fashion.

Satisfies analytical use cases: reporting, training ML models, generating insights

- **Domain Ownership of Data** – Data is owned and managed by the domains that generate or consume it, enabling decentralized data sharing.
- **Data as a Product** – Data is treated as a product, ensuring discoverability, usability, and high quality for consumers.
- **Self-Serve Data Platform** – Provides tools for domain teams to create, manage, and discover data products efficiently.
- **Computational Federated Governance** – Embeds security, compliance, and quality policies into each data product through automation.

Data product quantum

- Just as in the service mesh, teams build a data product quantum (DPQ) adjacent but coupled to their service
- Operationally independent but highly coupled set of behaviors and data.

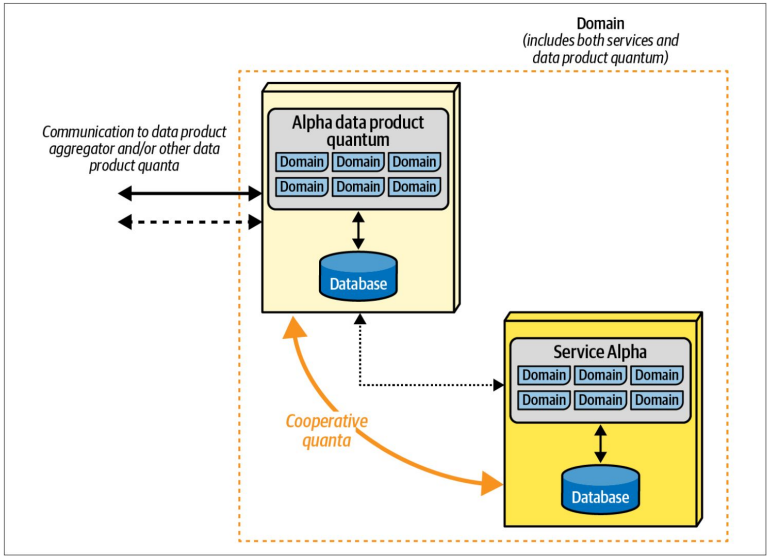

Figure 5-21. Structure of a data product quantum

Cooperative quantum

An operationally separate quantum that communicates with its cooperator via **asynchronous** communication and **eventual consistency**

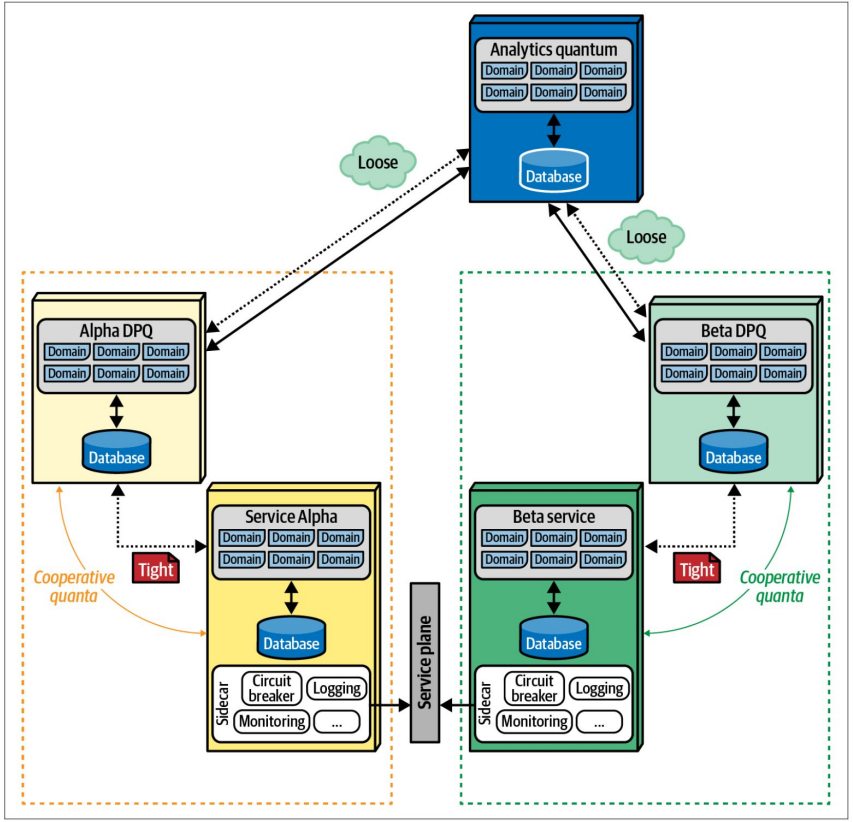

Figure 5-22. The data product quantum acts as a separate but highly coupled adjunct to a service

References

Building Evolutionary Architecture, by Neal Ford, Patrick Kua, and Rebecca Parsons