DDD Fundamentals and Strategic Design

KOENIG
step forward

Introduction to Domain-Driven Design (DDD)

Introduction to Domain-Driven Design (DDD)

- What is DDD and why it matters?
- Understanding domains, subdomains, and bounded contexts

Design Patterns for Microservices

- Microservices design patterns are fundamental to creating robust, scalable, and easily maintainable microservices-based applications.
- The implementation of these patterns streamlines development and significantly improves the quality and maintainability of the resulting applications.
- Recognizing and applying these patterns effectively can often be the difference between the success and failure of a microservices-based project. However, it's important to remember that each pattern comes with its own benefits and drawbacks.

Decomposing Monolith into Microservices

- The process of decomposition entails the partitioning of a monolithic application into microservices that are organized according to functional boundaries.
- The objective of this pattern is to enhance maintainability and resilience by enabling each microservice to operate autonomously.

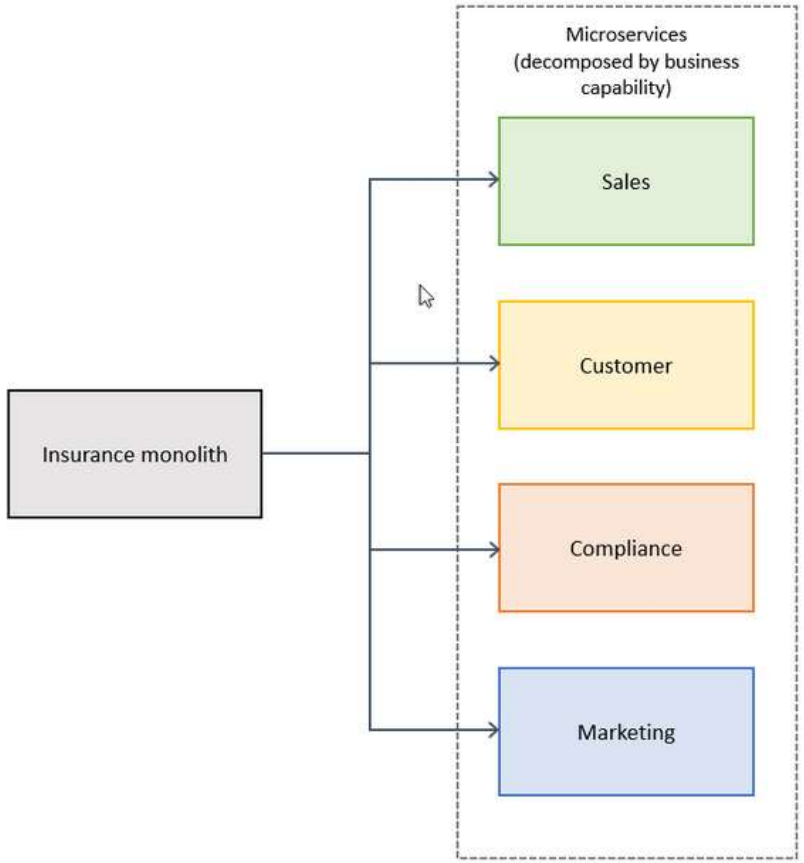
Decomposing Monolith into Microservices

- There are some patterns for decomposing the monolith to microservices:
  - Decompose by Business Capability
  - Decompose by Transactions
  - Strangler fig Pattern
  - Service per team pattern
  - Decompose by Domain-Driven Design

Decompose by Business Capability

- The term "business capability" is a fundamental concept utilized in business architecture modeling.
- A business capability is what a business does to generate value (for example, sales, customer service, or marketing). Typically, an organization has multiple business capabilities and these vary by sector or industry. Use this pattern if your team has enough insight into your organization's business units and you have subject matter experts (SMEs) for each business unit.
- Value generation is a fundamental objective of business operations. A business capability typically aligns with a business object.

Decompose by Business Capability



Decompose by Business Capability

Advantages	Disadvantages
• Generates a stable microservices architecture if the business capabilities are relatively stable. • Development teams are cross-functional and organized around delivering business value instead of technical features. • Services are loosely coupled.	• Application design is tightly coupled with the business model. • Requires an in-depth understanding of the overall business, because it can be difficult to identify business capabilities and services.

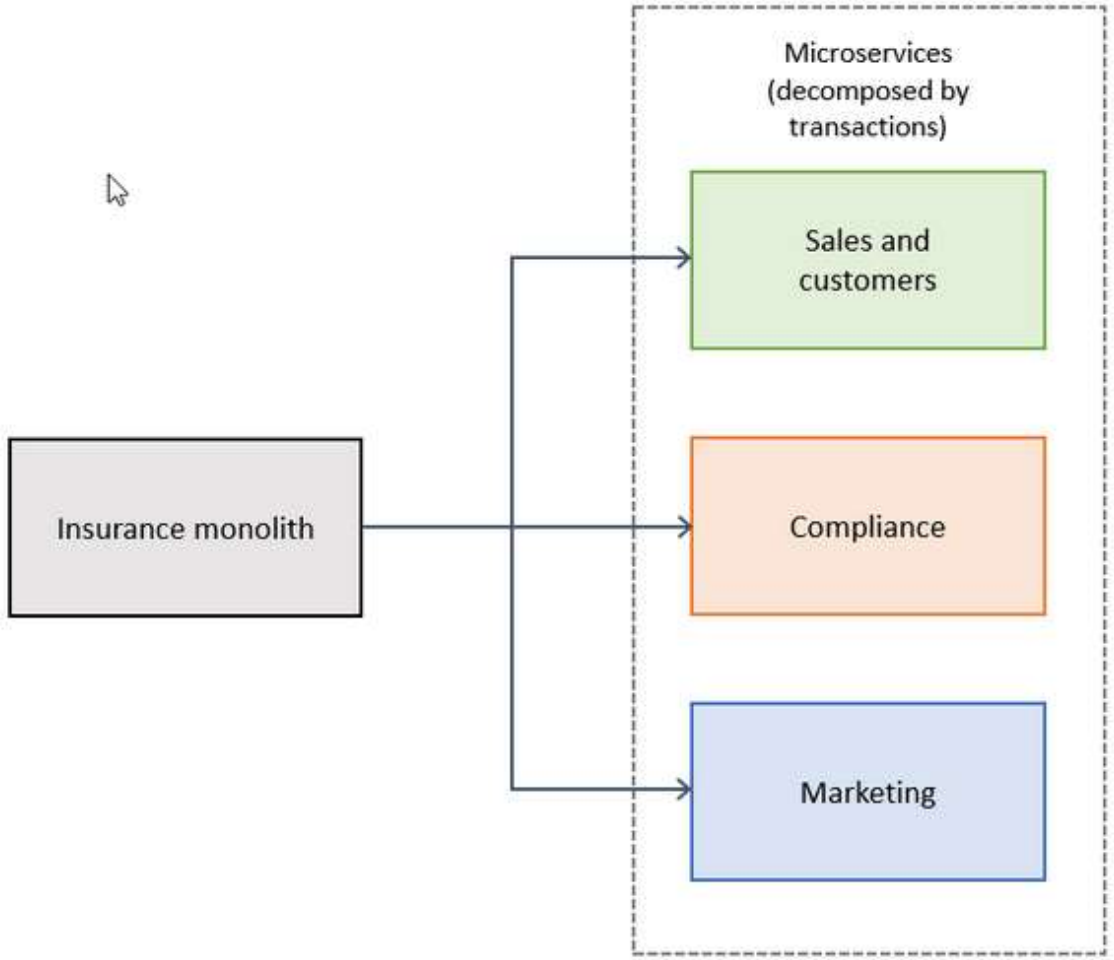
Decompose by Transactions

- In a distributed system, an application typically has to call multiple microservices to complete one business transaction.
- To avoid latency issues or two-phase commit problems, you can group your microservices based on transactions.
- This pattern is appropriate if you consider response times important and your different modules do not create a monolith after you package them.

Decompose by Transactions

- Services can be decomposed based on transactions. A distributed transaction involves two critical steps:
  - Prepare Phase: In this step, all parties involved in the transaction commit and inform the coordinator about their readiness for closure.
  - Commit or Rollback Phase: The transaction coordinator instructs all participants to either commit or rollback.
- It's important to note that the 2PC protocol tends to be slower than single microservice operations, making it less suitable for high-load scenarios.

Decompose by Transactions



Decompose by Transactions

Advantages	Disadvantages
• Faster response times. • You don’t need to worry about data consistency. • Improved availability.	• Multiple modules can be packaged together, and this can create a monolith. • Multiple functionalities are implemented in a single microservice instead of separate microservices, which increases cost and complexity. • Transaction-oriented microservices can grow if the number of business domains and dependencies among them is high. • Inconsistent versions might be deployed at the same time for the same business domain.

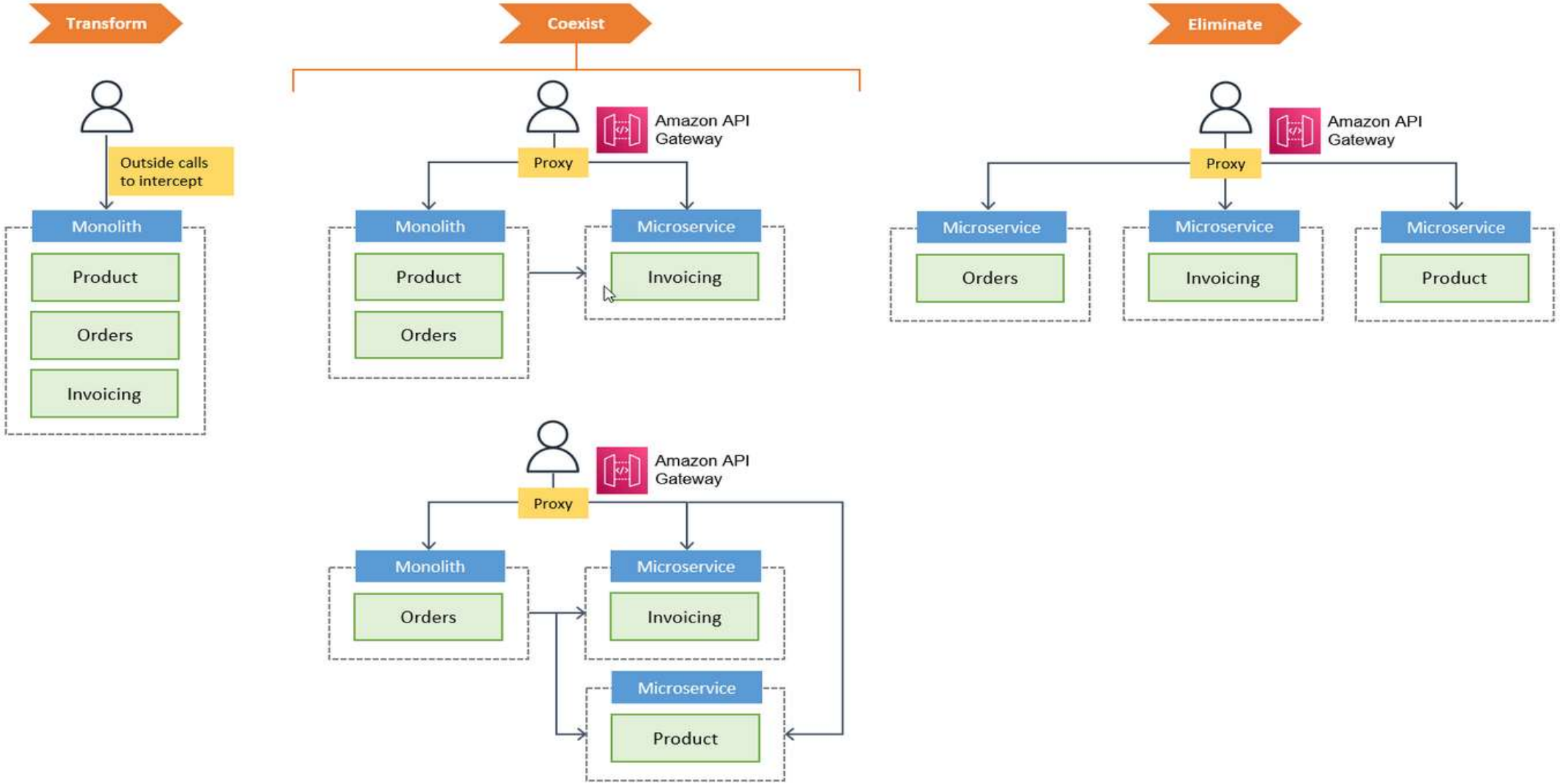
Strangler fig Pattern

- The design patterns discussed so far in this guide apply to decomposing applications for greenfield projects.
- What about brownfield projects that involve big, monolithic applications? Applying the previous design patterns to them will be difficult, because breaking them into smaller pieces while they're being used actively is a big task.
- The strangler fig pattern is a popular design pattern that was introduced by Martin Fowler, who was inspired by a certain type of fig that seeds itself in the upper branches of trees. The existing tree initially becomes a support structure for the new fig. The fig then sends its roots to the ground, gradually enveloping the original tree and leaving only the new, self-supporting fig in its place.

Strangler fig Pattern

- The process to transition from a monolithic application to microservices by implementing the strangler fig pattern consists of three steps: transform, coexist, and eliminate:
  - **Transform** – Identify and create modernized components either by porting or rewriting them in parallel with the legacy application.
  - **Coexist** – Keep the monolith application for rollback. Intercept outside system calls by incorporating an HTTP proxy (for example, Amazon API Gateway) at the perimeter of your monolith and redirect the traffic to the modernized version. This helps you implement functionality incrementally.
  - **Eliminate** – Retire the old functionality from the monolith as traffic is redirected away from the legacy monolith to the modernized service.

Strangler fig Pattern



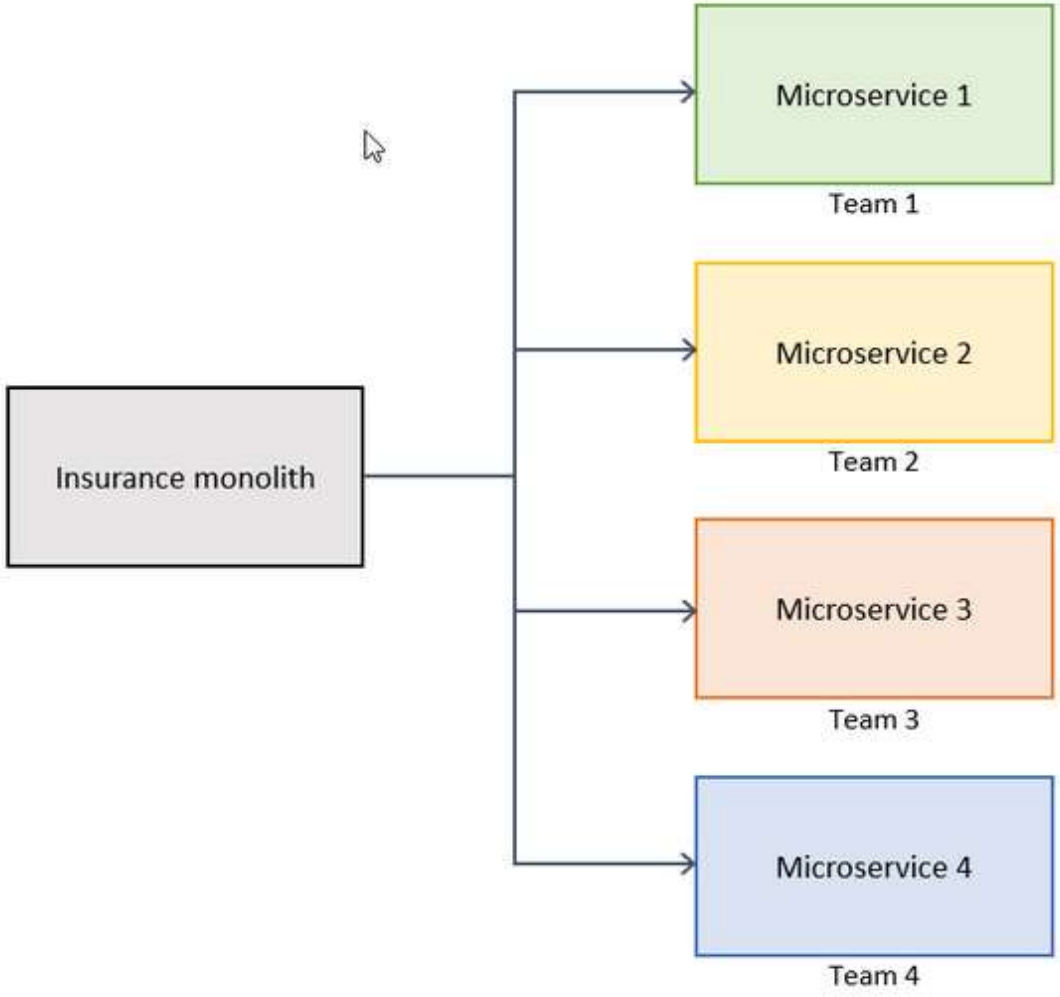
Strangler fig Pattern

Advantages	Disadvantages
• Allows for graceful migration from a service to one or more replacement services. • Keeps old services in play while refactoring to updated versions. • Provides the ability to add new services and functionalities while refactoring older services. • The pattern can be used for versioning of APIs.	• Isn’t suitable for small systems where the complexity is low and the size is small. • Cannot be used in systems where requests to the backend system cannot be intercepted and routed. • The proxy or facade layer can become a single point of failure or a performance bottleneck if it isn’t designed properly.

Service per Team Pattern

- Instead of decomposing monoliths by business capabilities or services, the service per team pattern breaks them down into microservices that are managed by individual teams.
- Each team is responsible for a business capability and owns the capability's code base.
- The team independently develops, tests, deploys, or scales its services, and primarily interacts with other teams to negotiate APIs.
- We recommend that you assign each microservice to a single team. However, if the team is large enough, multiple subteams could own separate microservices within the same team structure.

Service per Team Pattern



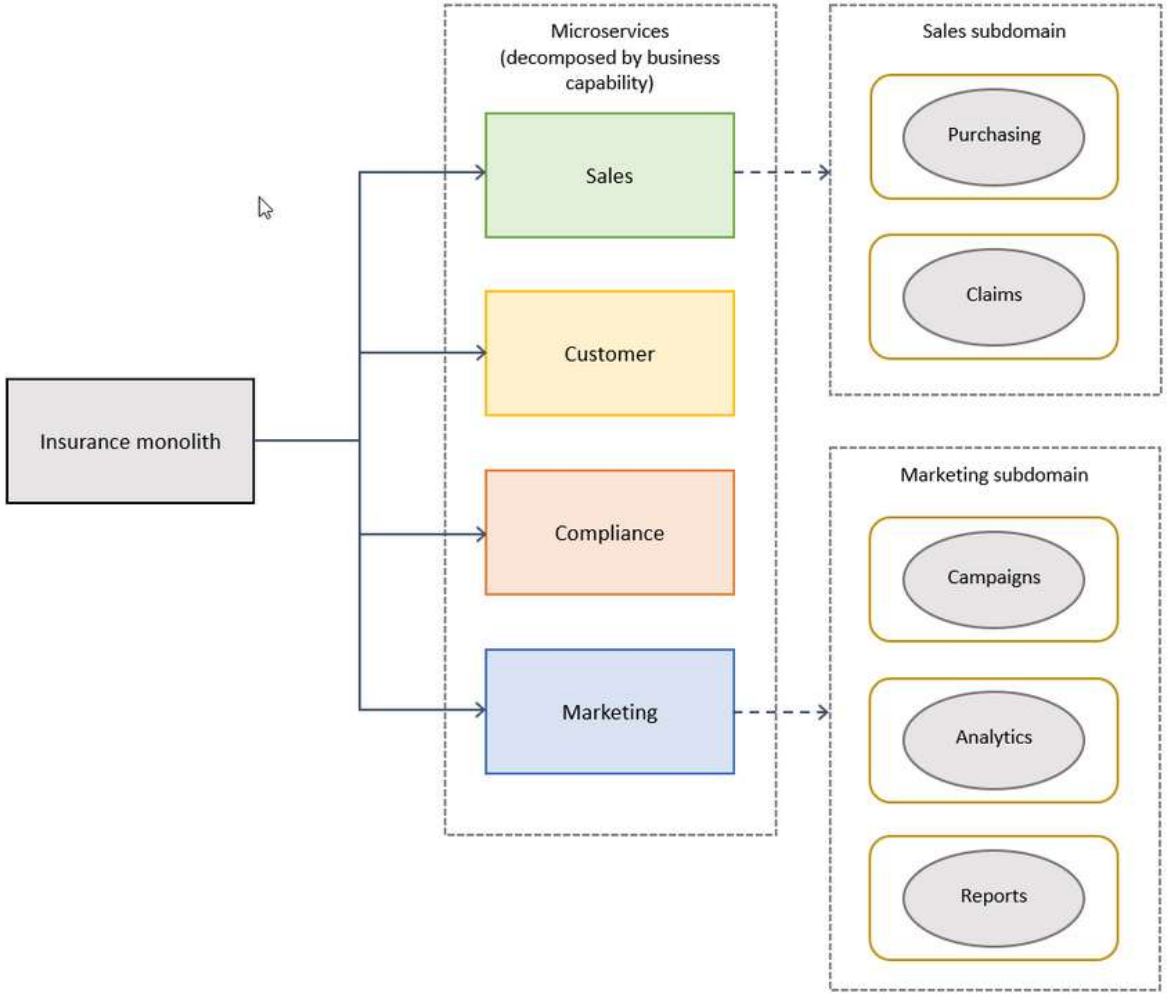
Service per Team Pattern

Advantages	Disadvantages
• Teams act independently with minimal coordination. • Code bases and microservices are not shared by multiple teams. • Teams can quickly innovate and iterate on product features. • Different teams can use different technologies, frameworks, or programming languages..	• It can be difficult to align teams to end-user functionality or business capabilities. • Additional effort is required to deliver larger, coordinated application increments, especially if there are circular dependencies between teams.

Decompose by Domain-Driven Design

- This task involves defining services that align with the subdomains of Domain-Driven Design (DDD).
- Domain-Driven Design establishes the application's domain or problem space. Domains have subdomains. Each subdomain is associated with a distinct segment of the business.

Decompose by Domain-Driven Design



Decompose by Domain-Driven Design

Advantages	Disadvantages
• Loosely coupled architecture provides scalability, resilience, maintainability, extensibility, location transparency, protocol independence, and time independence. • Systems become more scalable and predictable.	• Can create too many microservices, which makes service discovery and integration difficult. • Business subdomains are difficult to identify because they require an in-depth understanding of the overall business.

What is DDD and why it matters?

What is Domain-Driven Design (DDD)?

- Domain-Driven Design (DDD) is a **software development approach** that focuses on modeling software based on **real-world business domains.**
- It emphasizes collaboration between **developers and domain experts** to create a shared understanding of the **problem space,** ensuring that **software solutions** align with **business goals.**

What is DDD and why it matters?

Key Concepts of DDD

- **Domain** – The problem space where the business operates.
- **Ubiquitous Language** – A common, consistent language used by developers and business experts.
- **Bounded Context** – A defined boundary where a specific model applies.
- **Entities & Value Objects** – Fundamental building blocks representing business concepts.
- **Aggregates** – A cluster of domain objects treated as a single unit.
- **Repositories** – Interfaces for accessing domain objects.
- **Domain Events** – Notifications indicating changes in the domain state.
- **Application Services** – Coordinate domain logic and communication between services.

What is DDD and why it matters?

Why DDD Matters?

- **Aligns Software with Business Needs** – Ensures software solutions are built around business goals and domain knowledge.
- **Improves Communication** – Encourages collaboration between developers and domain experts using a shared language.
- **Manages Complexity** – Helps structure complex business logic into modular, maintainable components.
- **Enhances Scalability** – Provides a clear separation of concerns, making systems easier to scale and modify.
- **Encourages Long-Term Maintainability** – Helps reduce technical debt by focusing on business rules rather than just technology.

Understanding domains, subdomains, and bounded contexts

What is a Domain?

- A domain is the area of **knowledge, business, or activity** that your software system is designed to serve.
- It represents the **core problem space of the business**.

- Example:
  - In an **e-commerce application**, the domain is **online retail**, which includes processes like product management, order processing, and customer interactions.

Understanding domains, subdomains, and bounded contexts

What are Subdomains?

- A subdomain is a **smaller, specialized part** of the overall domain.
- Large domains are divided into multiple subdomains, each responsible for a **specific business function**.

- Types of Subdomains in DDD:
  - Core Domain
  - Supporting Subdomain
  - Generic Subdomain

Understanding domains, subdomains, and bounded contexts

Bounded Context

- A **bounded context** defines a **clear boundary** within which a particular domain model is consistent and valid.
- It ensures that different subdomains do not interfere with each other, reducing complexity and conflicts.

Strategic Design Principles

Strategic Design Principles

- Identifying core, supporting, and generic domains
- Designing bounded contexts and context mapping
- Cultivating collaboration between business and technical teams

Identifying core, supporting, and generic domains

- In **Domain-Driven Design (DDD),** strategic design helps in organizing a complex system by categorizing different parts of the business into **core, supporting, and generic domains**.
- This classification helps teams focus their efforts efficiently.

Identifying core, supporting, and generic domains

- **Core Domain**
  - The most important and valuable part of the business. This is where the **competitive advantage** lies and requires **custom development**.
- Key Characteristics:
  - Directly impacts the company's success.
  - Needs deep domain expertise.
  - Often complex and evolving.
  - Must be highly optimized for business needs.
- Example (E-commerce System):
  - **Pricing and discount engine** – Determines special discounts, dynamic pricing, and personalized offers, giving the business a competitive edge.

Identifying core, supporting, and generic domains

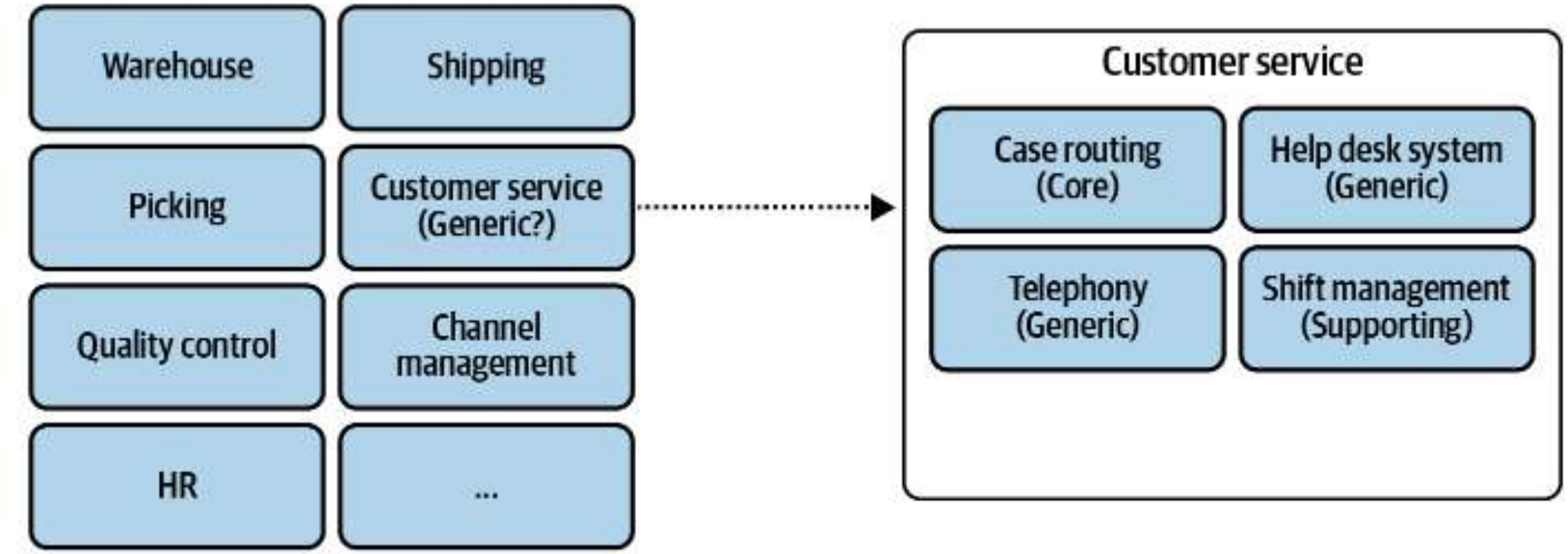
- **Supporting domain**
  - A domain that is necessary for the business but **not its main focus.**
  - These can often be implemented with **standard solutions or customized slightly.**
- Key Characteristics:
  - Supports the core domain but does not differentiate the business.
  - Often developed in-house but doesn’t require deep customization.
  - Can be optimized for efficiency rather than innovation.
- Example (E-commerce System):
  - **Inventory management** – Ensures stock availability but doesn’t directly influence the company's uniqueness.

Identifying core, supporting, and generic domains

- **Generic domain**
  - A domain that is **common across industries** and can be handled using **third-party solutions** or open-source frameworks.
- Key Characteristics:
  - Doesn’t require custom development.
  - Can be outsourced or implemented using off-the-shelf solutions.
  - Provides a **non-differentiating** service to the business.
- Example (E-commerce System):
  - **Payment processing** – Most businesses integrate **Stripe, PayPal, or Square** instead of building their own payment system.

Understanding Domains & Sub-Domains

- Core Domain & Core Subdomain
- Supporting Domain & Supporting Subdomain
- Generic Domain & Generic Subdomain



Identifying core, supporting, and generic domains

- Why Does This Classification Matter?
  - **Optimized Resource Allocation** – Focus development efforts on the core domain.
  - **Better Scalability** – Supporting and generic domains can use third-party solutions.
  - **Improved Maintainability** – Separates concerns, making the system more manageable.
  - **Cost Efficiency** – Reduces unnecessary custom development for generic solutions.

Designing bounded contexts and context mapping

- When building complex systems using **Domain-Driven Design (DDD),** it's essential to break the domain into manageable **bounded contexts** and define how they interact.
- This helps in organizing teams, maintaining clear domain boundaries, and reducing complexity.

Activity – 1 Identifying Domains and

Gigmaster

- Ticket sales and distribution company. Its mobile app analyzes users' music libraries, streaming service accounts, and social media profiles to identify nearby shows that its users would be interested in attending.
- Gigmaster's users are conscious of their privacy. Hence, all users' personal information is encrypted. Moreover, to ensure that users' guilty pleasures won't leak out under any circumstances, the company's recommendation algorithm works exclusively on anonymized data.
- To improve the app's recommendations, a new module was implemented. It allows users to log gigs they attended in the past, even if the tickets weren't purchased through Gigmaster.

Activity – 1 Identifying Domains and

Business domain and subdomains

Gigmaster's business domain is ticket sales. That's the service it provides to its customers.

Core subdomains: Gigmaster's main competitive advantage is its recommendation engine. The company also takes its users' privacy seriously and works only on anonymized data.

Finally, although not mentioned explicitly, we can infer that the mobile app's user experience is crucial as well. As such, Gigmaster's core subdomains are:

- Recommendation engine
- Data anonymization
- Mobile app

Activity – 1 Identifying Domains and

Generic subdomains. We can identify and infer the following generic subdomains:

- Encryption, for encrypting all data
- Accounting, since the company is in the sales business
- Clearing, for charging its customers
- Authentication and authorization, for identifying its users

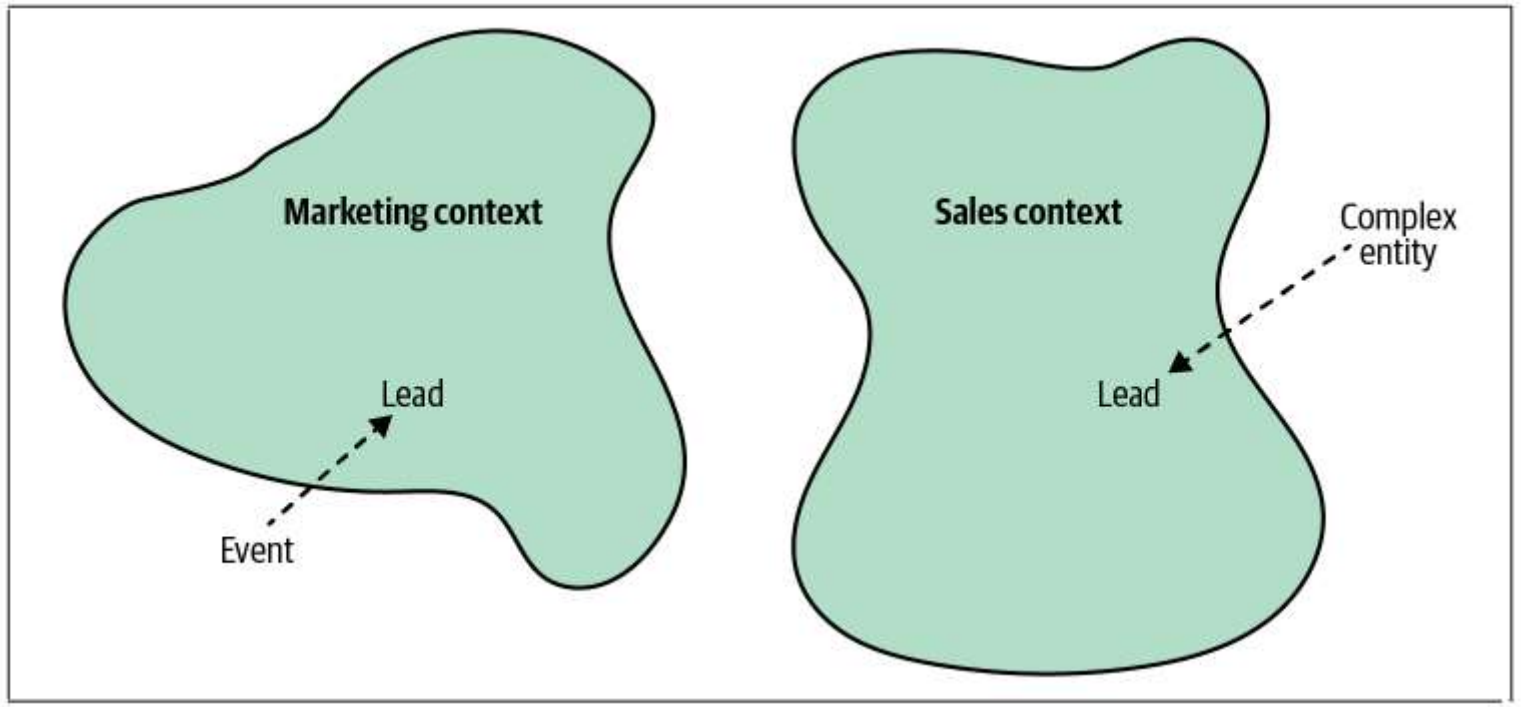
Supporting subdomains. Finally, the following are the supporting subdomains. Here the business logic is simple and resembles ETL processes or CRUD interfaces:

- Integration with music streaming services
- Integration with social networks
- Attended-gigs module

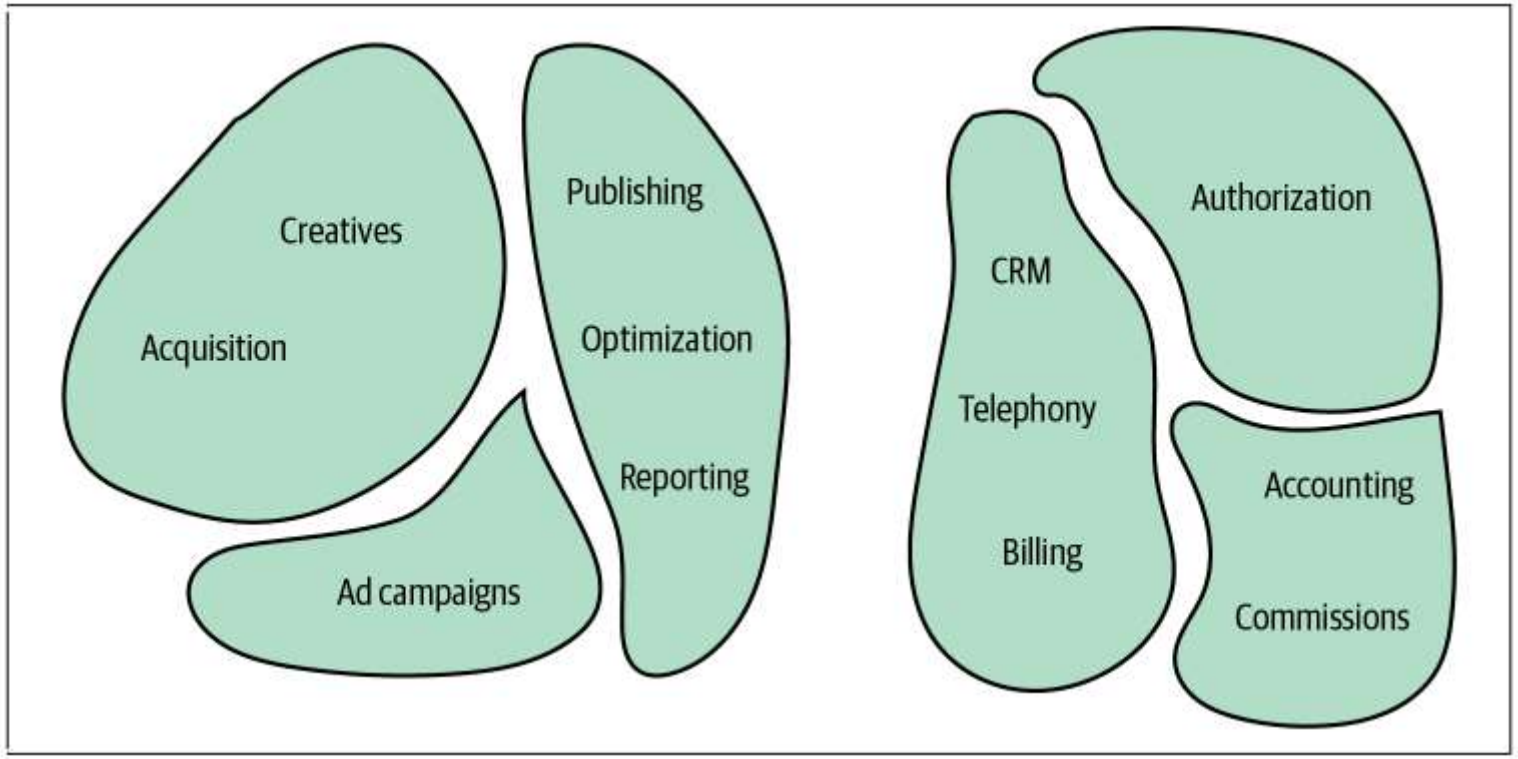
Designing bounded contexts and context mapping

- What is a Bounded Context?
  - A **bounded context** is a well-defined boundary within which a particular domain model is consistent and applicable. Each bounded context has:
    - Its own **domain logic** and **data model**
    - A **Ubiquitous Language** (common terminology used by developers and domain experts)
    - Clear **interfaces** for communication with other bounded contexts
- Example (E-Commerce System):
  - **Product Catalog Context** → Manages product details and descriptions.
  - **Order Management Context** → Handles order placement, tracking, and fulfillment.
  - **Payment Processing Context** → Processes payments and transactions.
- Each context has its own database, models, and services.

Designing bounded contexts and context mapping

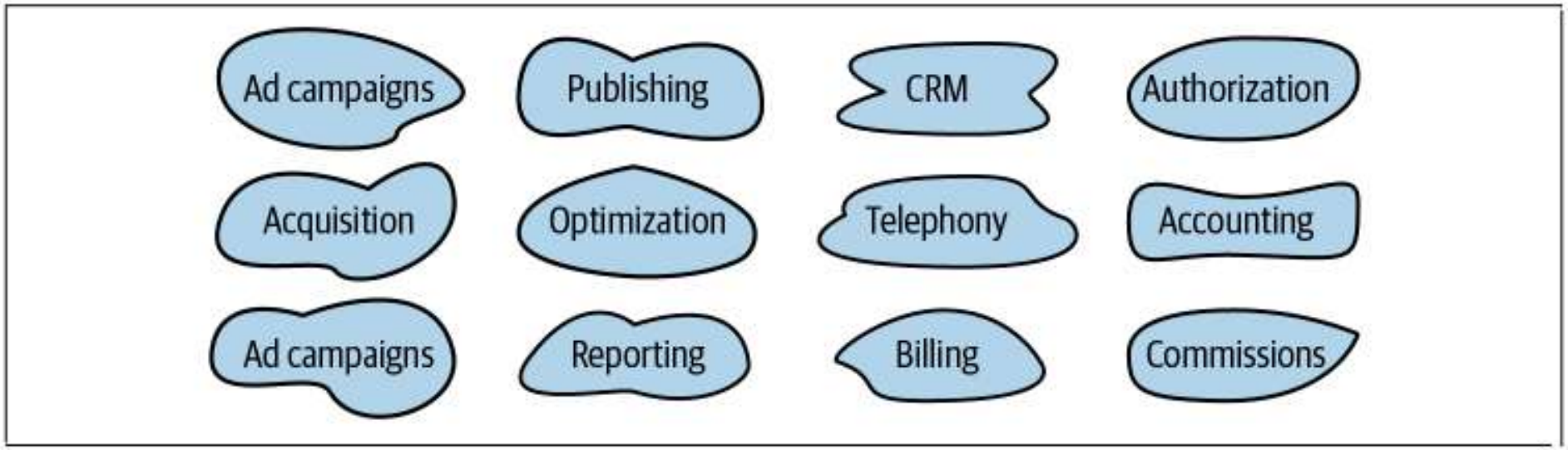


Designing bounded contexts and context mapping



Designing bounded contexts and context mapping

- Having a one-to-one relationship between contexts and subdomains can be perfectly reasonable in some scenarios. In others, however, different decomposition strategies can be more suitable.



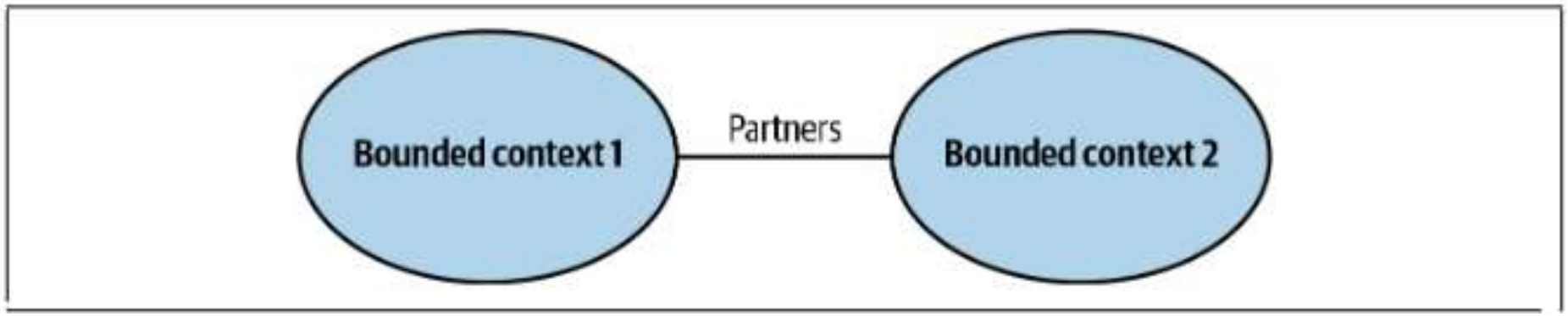
Integrating Bounded Contexts [Communication]

- Domain-driven design patterns for defining relationships and integrations between bounded contexts.
- These patterns are driven by the nature of collaboration between teams working on bounded contexts.
- It is to divide the patterns into three groups, each representing a type of team collaboration:
  - **Cooperation** [Partnership, Shared Kernel]
  - **Customer–supplier** [Conformist, Anti-Corruption Layer, Open-Host Service]
  - **Separate ways**

Integrating Bounded Contexts [Communication]

Cooperation: Partnerships

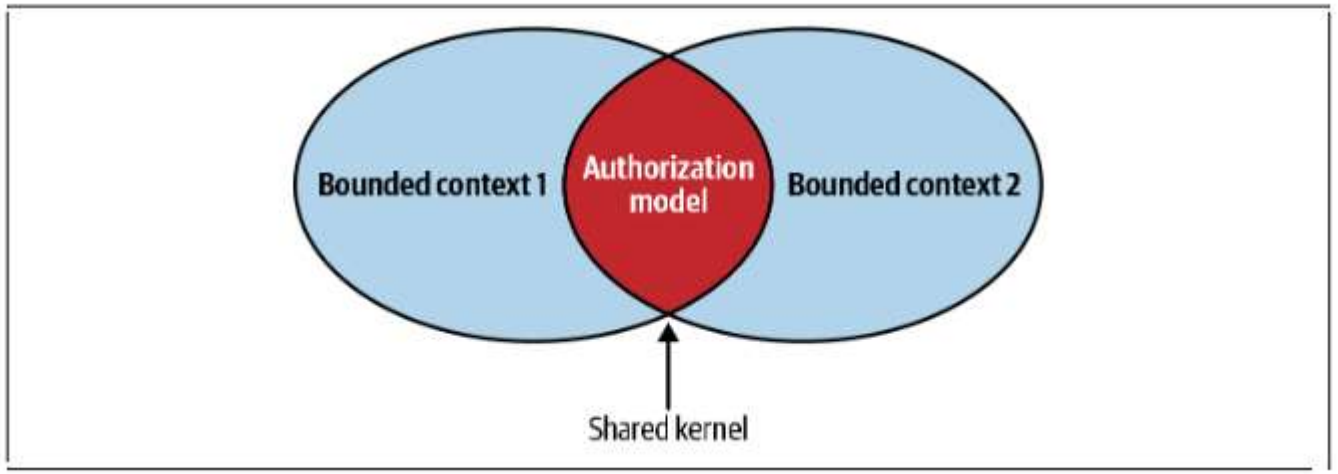
- In the partnership model, the integration between bounded contexts is coordinated in an ad hoc manner. One team can notify a second team about a change in the API, and the second team will cooperate and adapt—no drama or conflicts



Integrating Bounded Contexts [Communication]

Cooperation: Shared Kernel Module

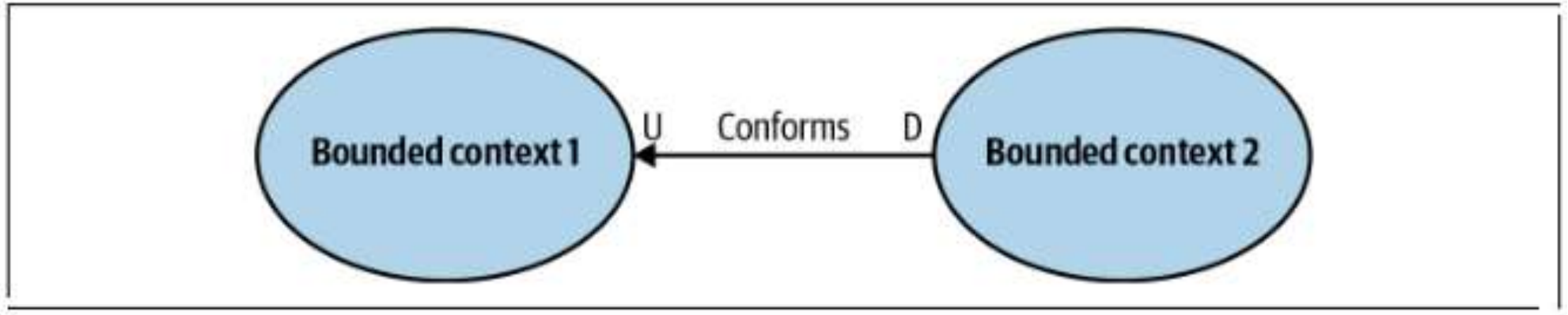
- Despite bounded contexts being model boundaries, there still can be cases when the same model of a subdomain, or a part of it, will be implemented in multiple bounded contexts.
- It's crucial to stress that the shared model is designed according to the needs of all of the bounded contexts. Moreover, the shared model has to be consistent across all of the bounded contexts that are using it.



Integrating Bounded Contexts [Communication]

Customer(U)-Supplier(D): Conformist

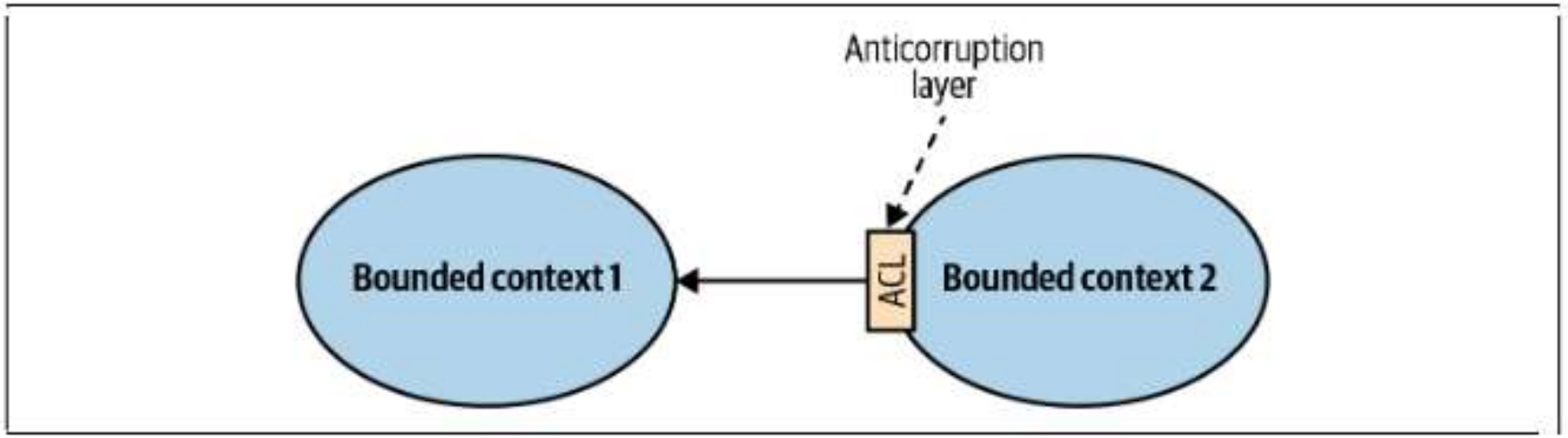
- In some cases, the balance of power favors the upstream team, which has no real motivation to support its clients' needs. Instead, it just provides the integration contract, defined according to its own model—take it or leave it.
- If the downstream team can accept the upstream team's model, the bounded contexts' relationship is called conformist.



Integrating Bounded Contexts [Communication]

Customer(U)-Supplier(D): Anti-Corruption Layer

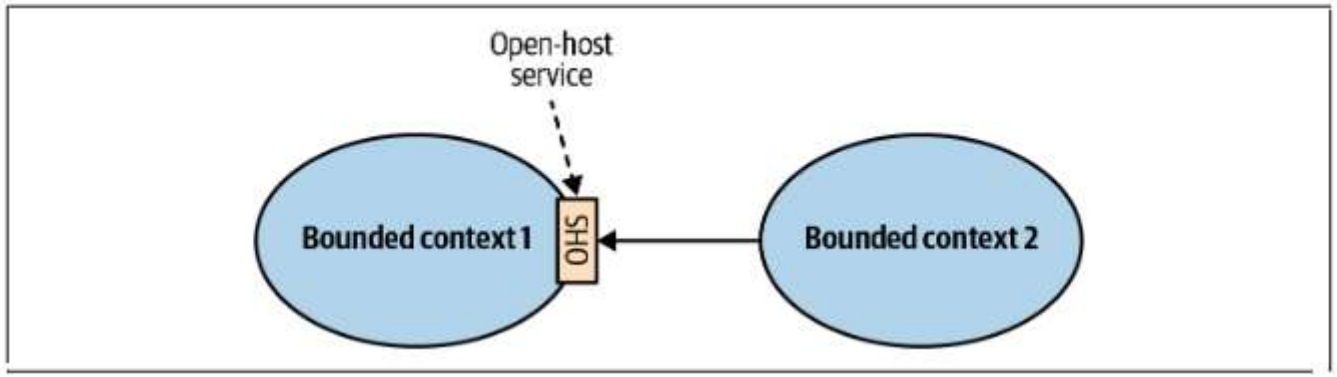
- As in the conformist pattern, the balance of power in this relationship is still skewed toward the upstream service. However, in this case, the downstream bounded context is not willing to conform.
- Instead, it can translate the upstream bounded context's model into a model tailored to its own needs via an anticorruption layer.



Integrating Bounded Contexts [Communication]

Customer(U)-Supplier(D): Open-Host Service

- This pattern addresses cases in which the power is skewed toward the consumers. The supplier is interested in protecting its consumers and providing the best service possible.
- To protect the consumers from changes in its implementation model, the upstream supplier decouples the implementation model from the public interface. This decoupling allows the supplier to evolve its implementation and public models at different rates,



Summary: Designing bounded contexts and context mapping

- How to Design Bounded Contexts?
- **Step 1: Identify Business Subdomains**
  - Analyze the business and break it into **core, supporting, and generic subdomains.**
  - Example: In an e-commerce system, the **order management** and **payment processing** subdomains have different rules and must be separate.
- **Step 2: Define Context Boundaries**
  - Ensure that **each context has a clear purpose** and does not overlap with others.
  - Use **Ubiquitous Language** specific to each bounded context.
  - Example: “Order” might mean **a customer purchase** in **Order Management** but **a financial transaction** in **Payment Processing.**

Summary: Designing bounded contexts and context mapping

- **Step 3: Establish Communication Between Contexts**
  - Use APIs, messaging, or events for communication between contexts.
  - **Anti-Corruption Layer (ACL):** A pattern that translates between two models to avoid direct dependencies.
  - Example: **Order Management** requests **Payment Processing** to charge a customer using an event-driven architecture.

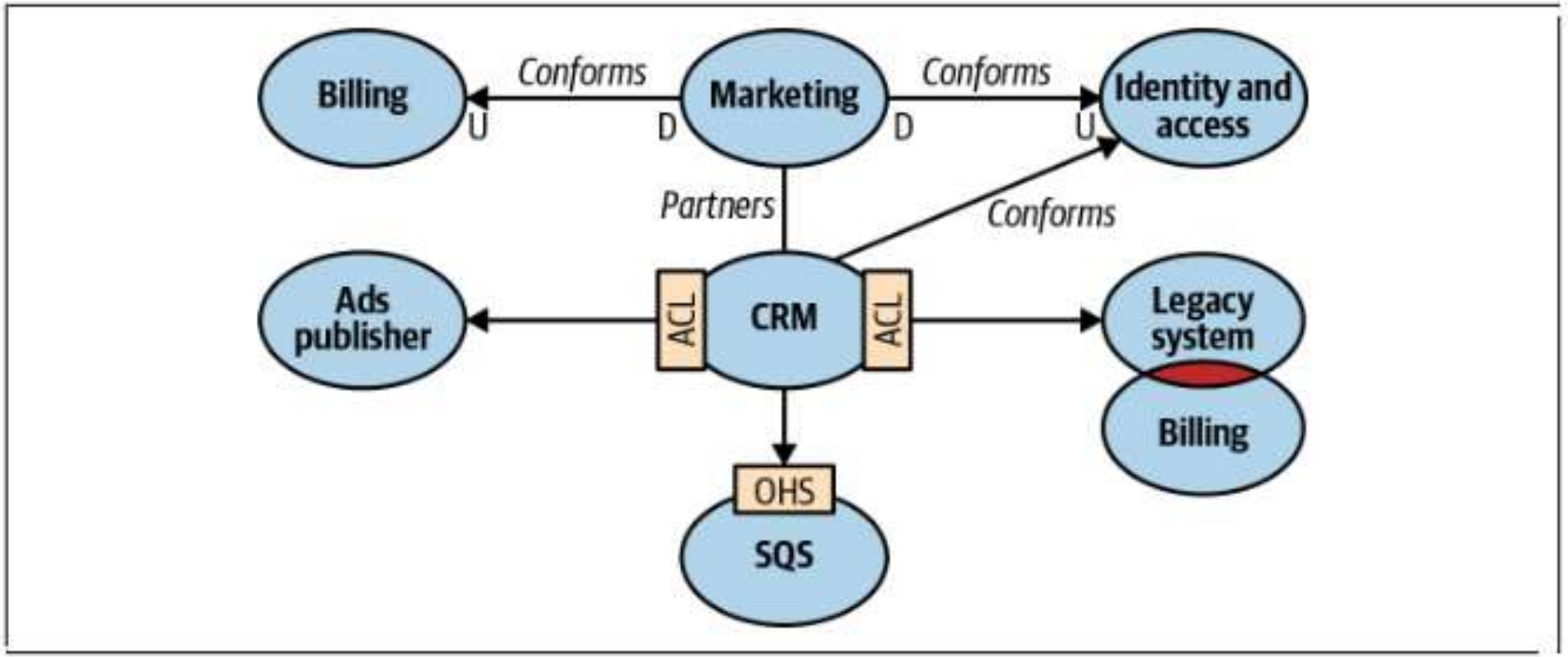
Summary: Designing bounded contexts and context mapping

- What is Context Mapping?
  - Context Mapping defines how multiple bounded contexts interact within a system. It visualizes dependencies and relationships between different parts of the domain.
- Common Context Mapping Patterns:
  - **Shared Kernel** – Two contexts share a common part of the model but evolve independently.
  - **Customer-Supplier** – One context depends on another (e.g., **Order Management** depends on **Product Catalog**).
  - **Anti-Corruption Layer (ACL)** – Translates models between contexts to prevent dependencies.
  - **Separate Ways** – Two contexts are independent and do not share data directly.

Designing bounded contexts and context mapping

- Example (E-Commerce System Context Map):
  - **Product Catalog Context** provides data to **Order Management** (Customer-Supplier).
  - **Order Management Context** interacts with **Payment Processing** using an **Anti-Corruption Layer (ACL).**
- **Why is This Important?**
  - **Reduces Complexity –** Clear boundaries prevent overlapping responsibilities.
  - **Improves Maintainability –** Changes in one context don’t break the entire system.
  - **Enhances Scalability –** Different teams can manage their own bounded contexts independently.
  - **Enables Distributed Architecture –** Microservices can be built based on bounded contexts.

Activity 2: Context Map for Bounded Context with Integration Patterns



Cultivating collaboration between business and technical teams

- In **Domain-Driven Design (DDD),** successful software development depends on **strong collaboration** between **business experts** and **technical teams.**
- Bridging the gap ensures that the software aligns with real-world business needs and is built with a deep understanding of the domain.

Cultivating collaboration between business and technical teams

- Why Collaboration is Critical?
  - **Reduces Miscommunication** – Ensures both teams speak the same language.
  - **Aligns Software with Business Goals** – Helps developers build features that truly matter.
  - **Improves Domain Knowledge** – Developers gain insights from business experts.
  - **Accelerates Decision-Making** – Faster problem-solving through shared understanding.

Cultivating collaboration between business and technical teams

- Strategies to Enhance Collaboration

1. Establish a Ubiquitous Language
    - A shared language that both business and technical teams understand and use.
    - Helps avoid misunderstandings and misinterpretations.
    - Defines **key business terms** clearly and consistently.
- Example:
    - Instead of calling it a "Purchase Order" in one part of the system and an "Invoice" in another, both teams agree to use "**Order**."
- Action Step: Maintain a **glossary** of domain terms and ensure they are used in code, discussions, and documentation.

Cultivating collaboration between business and technical teams

2. Engage Business Experts in Modeling Sessions
    - Regular **collaborative modeling** sessions where both teams shape the domain model together.
    - Use **Event Storming, Domain Storytelling, or Context Mapping** to visualize business processes.
    - Business experts provide real-world insights; developers translate them into domain models.
- Action Step: Schedule **frequent domain workshops** where business users explain real-world scenarios while developers translate them into software models.

Cultivating collaboration between business and technical teams

3. Involve Developers in Business Discussions
    - Developers participate in business meetings to understand real business challenges.
    - Helps them design better models and make informed technical decisions.
- Action Step: Invite engineers to **product strategy meetings**, so they can contribute ideas early and align technical feasibility with business goals.

Cultivating collaboration between business and technical teams

4. Use Bounded Contexts to Define Clear Responsibilities
    - Splitting a large system into **bounded contexts** helps each team focus on a specific business area.
    - Encourages **domain-driven team structures**, making it easier for teams to own and understand their areas.
- Action Step: Assign **cross-functional teams** to different **bounded contexts** to ensure each domain gets the right expertise.

Cultivating collaboration between business and technical teams

5. Implement Continuous Feedback Loops
    - **Regular check-ins** between business and development teams to refine the domain model.
    - Helps in **adjusting requirements** and ensuring the software stays aligned with business needs.
    - Encourages early issue detection before they become costly.
- Action Step: Set up **weekly sync meetings** and use **feedback tools** like Slack channels or internal forums for ongoing discussions.

Cultivating collaboration between business and technical teams

6. Tools to Facilitate Collaboration

- **Event Storming** – A visual technique to map out business processes.
- **Domain Storytelling** – A method to describe domain scenarios in a simple way.
- **Collaboration Boards (Miro, MURAL)** – To visualize workflows and relationships.
- **API Documentation & Contracts** – Ensures clarity between business logic and technical implementation.

Cultivating collaboration between business and technical teams

- Final Takeaways
  - DDD is not just a technical approach—it requires strong business involvement.
  - **Ubiquitous Language** ensures a shared understanding of domain concepts.
  - **Regular workshops** and feedback loops keep development aligned with business needs.
  - **Bounded Contexts** help create focused teams with clear responsibilities.

Tactical Design and Practical Applications

Tactical Design Principles

Tactical Design Principles

- Entities, Value Objects, and Aggregates
- Domain Events and Repositories
- Leveraging factories and application services

Entities, Value Objects, and Aggregates

- Tactical design in **Domain-Driven Design (DDD)** provides **practical building blocks** to implement domain models effectively.
- The key elements include **Entities, Value Objects, and Aggregates**, which help in structuring complex business logic while maintaining consistency.

Entities, Value Objects, and Aggregates

- Entities
  - An **Entity** is a domain object that has a distinct **identity** and **persists over time.**
  - Identified by a **unique identifier (ID)** rather than just its attributes.
  - Can change over time while maintaining the same identity.
- Key Characteristics:
  - Has a unique identity that remains consistent.
  - Encapsulates business logic and state changes.
  - Mutability is allowed, as entities evolve.
- Example
  - **Orders, Customers, and Employees** are typical **Entities** because they must be **uniquely identified**.

Entities, Value Objects, and Aggregates

- Value Objects
  - A **Value Object** represents **a descriptive characteristic** of a domain without a unique identity.
  - **Immutable** – Once created, its state cannot change.
  - Used for modeling **concepts like addresses, money, and measurements**.
- Key Characteristics:
  - **No identity** – Two value objects with the same attributes are considered **equal**.
  - **Immutable** – Cannot be modified after creation.
  - **Reusable** – Used across multiple entities.
- Example
  - Address, Money, and Dimensions are perfect **Value Objects** because they **only store data and don’t require an identity.**

Entities, Value Objects, and Aggregates

- Aggregates
  - An **Aggregate** is a cluster of related domain objects (Entities + Value Objects) that should be treated as **a single unit.**
  - Defines a **root entity (Aggregate Root)** that ensures data consistency within the boundary.
  - Other objects within the aggregate can only be accessed via the **aggregate root.**
- Key Characteristics:
  - **Ensures data consistency**
  - **Encapsulates business rules**.
  - **Restricts direct access**.

Entities, Value Objects, and Aggregates

- Why Use Aggregates?
  - Protects **data integrity** by ensuring all related objects are modified together.
  - Prevents **direct access** to nested entities (e.g., **OrderItem cannot be modified outside Order**).
  - Helps manage **transaction boundaries** efficiently.
- **Best Practices for Using Tactical Design in DDD**
  - Keep Entities Small
  - Prefer Value Objects When Possible
  - Design Aggregates Carefully
  - Avoid Large Aggregates

Domain Events and Repositories

- Domain Events
  - A **Domain Event** represents something **important that happened** in the domain that business stakeholders care about.
  - Events capture **changes in state** and help **decouple business logic** from the rest of the system.
- Why Use Domain Events?
  - **Decouples different parts of the system** (e.g., notifying users when an order is placed).
  - **Improves scalability** by enabling **event-driven architectures**.
  - **Ensures business rules are followed** by broadcasting events.

Domain Events and Repositories

- How to Handle Domain Events?
  - **Publish Event** – When an order is placed, trigger **OrderPlacedEvent**.
  - **Listen to Event** – The **OrderPlacedEventHandler** listens and performs actions (e.g., send an email).
- Best Practices for Domain Events:
  - **Use immutable event objects** to ensure consistency.
  - **Keep event handling logic separate** from the main business logic.
  - **Use an event bus (like Spring Events or Kafka)** for real-world applications.

Domain Events and Repositories

- Repositories
  - A **Repository** provides an **interface** for accessing domain objects, abstracting away database interactions.
- Why Use Repositories?
  - **Separates database logic from business logic.**
  - **Encapsulates queries and persistence logic** for domain objects.
  - **Simplifies testing** by allowing **mocking** of database calls.
- Best Practices for Repositories:
  - **Expose only necessary methods** (avoid generic CRUD methods).
  - **Keep repositories focused on aggregates** (handle one aggregate per repository).
  - **Use Specification Pattern** for complex queries instead of bloating repositories.

Leveraging factories and application services

- Factories in DDD
  - A **Factory** is a design pattern used to **create complex domain objects** while hiding the construction logic.
- Why Use Factories?
  - **Encapsulates complex creation logic** in one place.
  - **Ensures object consistency** by applying business rules at creation time.
  - **Simplifies object creation** by avoiding large constructors in entities.
- Factory Benefits:
  - **Keeps services clean** by offloading creation logic.
  - **Ensures domain rules** are applied before object creation.
  - **Improves maintainability** by centralizing object construction.

Leveraging factories and application services

- Application Services in DDD
  - **Application Services** handle use cases by coordinating domain logic, repositories, and domain events.
- Why Use Application Services?
  - **Separates application logic from domain logic.**
  - **Manages transactions and security.**
  - **Orchestrates multiple domain objects and repositories.**
- Application Service Benefits:
  - **Keeps domain models clean** by handling external concerns.
  - **Encapsulates transactions and event publishing**.
  - **Simplifies testing** by isolating business logic from infrastructure.

Implementing DDD

Implementing DDD

- Real-world examples of DDD in action
- Transitioning from a legacy system to a DDD approach
- Common pitfalls and how to avoid them

Real-world examples of DDD in action

Scenario: **Order Fulfillment System in a Supply Chain**

Define the Business Problem

- A supply chain company needs an efficient order fulfillment system that ensures:
  - Customers place orders for products.
  - Inventory is checked for availability.
  - Orders are packed and shipped.
  - Customers receive tracking updates.

Real-world examples of DDD in action

Business Goal: Food Delivery System

- To enhance operational efficiency, the company aims to redesign the food delivery system using Domain-Driven Design (DDD) principles. The objective is to:
- **Streamline Order Processing** – Reduce delays in order confirmation and preparation.
- **Improve Payment Reliability** – Ensure smooth transactions and order validation.
- **Optimize Delivery Assignment** – Assign drivers based on availability and proximity.
- **Enhance Customer Experience** – Provide real-time order tracking and notifications.
- **Improve Communication Between Teams** – Integrate restaurant, delivery, and payment workflows efficiently.

Transitioning from a legacy system to a DDD approach

- Migrating from a **monolithic, tightly coupled legacy system** to a **DDD-based architecture** is challenging but rewarding.
- The goal is to **incrementally** refactor and **modernize** the system while ensuring business continuity.

Transitioning from a legacy system to a DDD approach

- **Key Steps for Transitioning to DDD**
- **Understanding the Legacy System**
  - Identify **core business processes** and their dependencies.
  - Analyze **existing domain models** (if any).
  - Recognize **pain points** (e.g., high coupling, scalability issues, slow changes).
  - Document **business rules** and workflows.
- Example:
  - A legacy **E-commerce Monolith** has the following tightly coupled modules:
    - Order Management
    - Payment Processing
    - Inventory Management
    - Customer Service

Transitioning from a legacy system to a DDD approach

- **Define Bounded Contexts**
  - **Bounded Contexts** define the **scope** of each business domain to reduce complexity and dependency issues.
  - Identify **subdomains** (Core, Supporting, Generic).
  - Define **clear boundaries** where business logic applies.
  - Establish **ubiquitous language** for each context.
- Example:
  - **Order Context** → Manages orders, order statuses, and customer purchases.
  - **Payment Context** → Handles transactions, refunds, and invoicing.
  - **Inventory Context** → Tracks product stock levels.

Transitioning from a legacy system to a DDD approach

- **Gradual Refactoring Using the Strangler Pattern**
  - **The Strangler Pattern** allows **incremental migration** by replacing legacy components **one by one.**
    - Start **small** with a non-critical service.
    - Introduce a **new microservice** implementing DDD principles.
    - Route new functionality **to the modern service** while keeping the old system running.
    - Decommission the **legacy module** after a full transition.
- Example:
  - Phase 1: Introduce a new **OrderService** using DDD principles alongside the legacy system.
  - Phase 2: Move **Payments** to a new service, integrating with Orders.
  - Phase 3: Migrate **Inventory Management**, finally deprecating the monolith.

Transitioning from a legacy system to a DDD approach

- **Implement Tactical Design Patterns**
  - Use **Entities & Value Objects** to model domain concepts.
  - Define **Aggregates** to maintain data integrity.
  - Introduce **Repositories** to abstract database access.
  - Implement **Domain Events** for decoupled communication.
- Example:
  - Migrating **Orders** from the monolith to a DDD-based service.

Transitioning from a legacy system to a DDD approach

- **Introduce Application Services**
  - Application Services help orchestrate domain logic while keeping the domain layer clean.
- **Integrate with the Legacy System**
  - During the transition, **legacy and DDD services must co-exist**.
  - Use **Event Sourcing** to track domain changes.
  - Implement **Anti-Corruption Layer (ACL)** to prevent legacy pollution in new services.
  - Leverage **Message Queues (Kafka, RabbitMQ)** for communication between services.
  - Example: Implementing **Anti-Corruption Layer**

Transitioning from a legacy system to a DDD approach

- **Modernize Data Access with Repositories**
  - Legacy systems often have **direct database access** across modules. Use **repositories** to **abstract** database interactions.
  - Introduce **Repository Pattern** to handle persistence.
  - Use **CQRS (Command Query Responsibility Segregation)** to separate read and write operations.
  - Example: Separate Read and Write Repositories

Transitioning from a legacy system to a DDD approach

- **Fully Transition to a DDD-Based Architecture**
  - Gradually **decommission** the legacy monolith.
  - Ensure **all services** follow DDD principles.
  - Implement **CI/CD pipelines** for efficient deployments.
  - Conduct **domain-driven refactoring** as the business evolves.

Transitioning from a legacy system to a DDD approach

- Summary of Transition Approach

Step	Action
Analyze Legacy System	Identify pain points, business rules, and dependencies.
Define Bounded Contexts	Separate domain models and align with business.
Apply Strangler Pattern	Incrementally replace monolith modules with microservices.
Implement Tactical DDD	Use Aggregates, Entities, Repositories, and Domain Events.
Introduce Application Services	Separate domain logic from infrastructure concerns.
Integrate with Legacy System	Use Event Sourcing, ACL, and message queues.
Modernize Data Access	Implement Repository and CQRS patterns.
Fully Transition to DDD	Decommission monolith and embrace modular architecture.

Common pitfalls and how to avoid them

1. Treating DDD as Just Another Technical Framework

- Pitfall:
  - Many teams **adopt DDD purely as a technical approach** without focusing on its real purpose—**aligning software design with business needs.**
- How to Avoid:
  - **Understand the Business First** – Engage domain experts before writing code.
  - **Focus on Ubiquitous Language** – Ensure both **business and technical teams** use the same terms.
  - **DDD is a Mindset, Not a Framework** – Use it for problem-solving, not just coding patterns.

Common pitfalls and how to avoid them

2. Ignoring Ubiquitous Language

- Pitfall:
  - Developers and business experts use different terminologies, leading to misunderstandings and inconsistent models.
- How to Avoid:
  - **Develop a shared glossary** – Ensure **everyone** speaks the same language.
  - **Use the Ubiquitous Language in Code** – Class and method names should reflect real-world domain terms.
  - **Hold regular meetings** between developers and domain experts to refine the model.

Common pitfalls and how to avoid them

3. Poorly Defined Bounded Contexts

- Pitfall:
  - Teams either **create too many small contexts** or **one giant context**.
- How to Avoid:
  - **Identify Core, Supporting, and Generic Domains** before defining contexts.
  - **Use Context Mapping** to visualize relationships between services.
  - **Apply Anti-Corruption Layer (ACL)** when integrating legacy systems to prevent contamination.

Common pitfalls and how to avoid them

4. Overusing Entities Instead of Value Objects

- Pitfall:
  - Developers treat **every object as an Entity**, leading to **unnecessary complexity and bloated databases.**
- How to Avoid:
  - **Use Value Objects for immutable data** (e.g., Money, Address, Coordinates).
  - **Reserve Entities for objects with unique identity** (e.g., User, Order).

Common pitfalls and how to avoid them

5. Incorrect Aggregate Design

- Pitfall:
  - Aggregates are either **too large** (causing performance issues) or **too small** (leading to data consistency problems).
- How to Avoid:
  - **Follow the Single Responsibility Principle** – Aggregates should manage only what they own.
  - **Use Factories for Aggregate Creation** – Prevent invalid object states.
  - **Reference Other Aggregates by ID** instead of embedding full objects.

Common pitfalls and how to avoid them

6. Treating Repositories as CRUD Services

- Pitfall:
  - Repositories should **only handle domain objects**, but teams often turn them into **generic database access layers**.
- How to Avoid:
  - Repositories should retrieve Aggregates, not just raw data.
  - Use Specification Pattern for complex queries.
  - Keep business logic inside Aggregates, not Repositories.

Common pitfalls and how to avoid them

8. Trying to Apply DDD Everywhere

- Pitfall:
  - Applying **DDD principles to every module**, even in **simple CRUD applications**, increases complexity.
- How to Avoid:
  - **Use DDD only for Complex Business Domains** (e.g., finance, logistics).
  - **For simple modules, use traditional CRUD or Service-Oriented approaches**.
  - **Follow the 80/20 Rule:** Focus DDD on **core business logic**, not generic services.

Hands-on Exercises and Case Studies

Hands-on Exercises and Case Studies

- Building a domain model for a sample business scenario
- Context mapping workshop to align team understanding

Hands-on Exercises and Case Studies

Problem Statement: Digital Payment System

- A **fintech company** wants to build a **digital payment system** that allows users to:
  - **Transfer money** between wallets or bank accounts.
  - **Top up wallets** via bank transfers or cards.
  - **Process payments** for merchants securely.
  - **Detect fraudulent transactions** based on user behavior.
- The system must be **secure, scalable, and maintainable** while ensuring **seamless transactions** across different services.