Unit-5

Architectural Decision-Making and Trade-offs

Overview

5.1 Architectural decision-making process

5.2 Balancing functional and non-functional requirements

5.3 Trade-offs in architecture (e.g., performance vs. scalability, security vs. flexibility)

5.4 Risk analysis and mitigation strategies in architecture

5.5 Justifying architectural decisions and documenting trade-offs

5.6 Case study: A system developed using Microservices Architecture

Objectives

- **Understand** the key steps in the architectural decision-making process.
- **Explore** how to balance functional and non-functional requirements.

What is Architectural Decision-Making?

- A **systematic process** to determine the structure and design of a software system.
- Architectural decision-making involves selecting the best design options to meet system requirements while addressing constraints and trade-offs.
- Key aspects:
  - **Strategic**: Influences system qualities and business goals.
  - **Collaborative**: Involves stakeholders like developers, managers, and users.
  - **Iterative**: Requires continuous evaluation and refinement.
- **Purpose:**
  - Ensure the system meets functional and non-functional requirements.
  - Balance competing priorities (e.g., performance, cost, scalability).
  - Mitigate risks and ensure long-term maintainability.

Why is it Important?

1. **Ensures Alignment:**
   Decisions align with business goals and technical constraints.
2. **Reduces Risks:**
   Proactively addresses potential challenges and failures.
3. **Improves Quality:**
   Results in a robust, scalable, and maintainable system.
4. **Facilitates Communication:**
   Provides a clear rationale for decisions, improving stakeholder buy-in.

Architectural Decision-Making Process

1. **Identify Stakeholders and Requirements**
    - **Who:** Business owners, developers, end-users.
    - **What:** Functional and non-functional requirements.
2. **Analyze Trade-offs**
    - **Example:** Security vs. performance.
    - Tools: **Architecture Tradeoff Analysis Method (ATAM).**
3. **Evaluate Constraints**
    - **Technical:** Hardware, frameworks, legacy systems.
    - **Business:** Budget, deadlines.
4. **Model Architectural Alternatives**
    - **Approaches:** Layered, microservices, client-server.
    - Use **prototypes** and **models** for comparison.
5. **Validate and Document Decisions**
    - Create **diagrams** and **specifications**.
    - Use structured documentation like **views and perspectives**.

Functional vs. Non-Functional Requirements

- **Functional Requirements**
  - Define **specific actions** or behaviors.
  - Examples:
    - User login functionality.
    - Data processing workflows.
- **Non-Functional Requirements**
  - Define **system qualities** or constraints.
  - Examples:
    - Performance (e.g., response time < 1 sec).
    - Reliability (e.g., 99.9% uptime).

Balancing Functional and Non-Functional Requirements

Challenges:

- **Conflicting Goals:** High security may reduce usability.
- **Resource Allocation:** Prioritize within time and budget limits.

Strategies:

1. **Prioritize Requirements:** Use tools like **MoSCoW** (Must, Should, Could, Won't).
2. **Iterative Prototyping:** Build incrementally to test feasibility.
3. **Architectural Patterns:** Leverage known solutions (e.g., CQRS for scalability).
4. **Trade-off Analysis:** Document compromises using **decision records**.

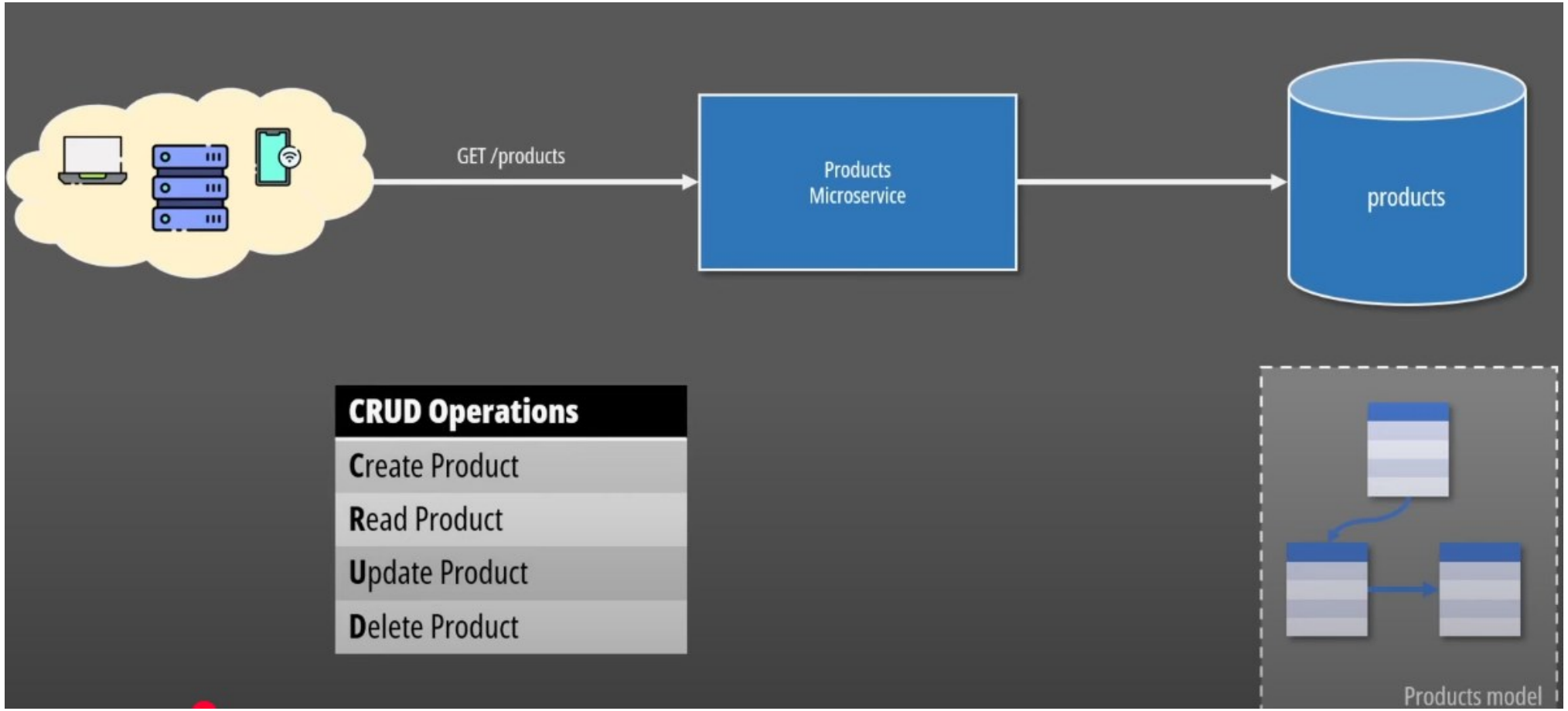
MoSCoW (Must, Should, Could, Won’t)

- **Example:** If you're developing a mobile app:
  - **Must Have:** User login, basic navigation, core functionality.
  - **Should Have:** Push notifications, social media integration.
  - **Could Have:** Customizable themes, advanced search filters.
  - **Won’t Have:** Augmented reality features (planned for a future release).

CQRS (Command Query Responsibility Segregation)

- A design pattern that separates **read** (query) and **write** (command) responsibilities in a system.
- Each is handled by separate models to optimize performance and scalability.
- **Core Idea:**
  - Commands change the state.
  - Queries retrieve data without side effects.

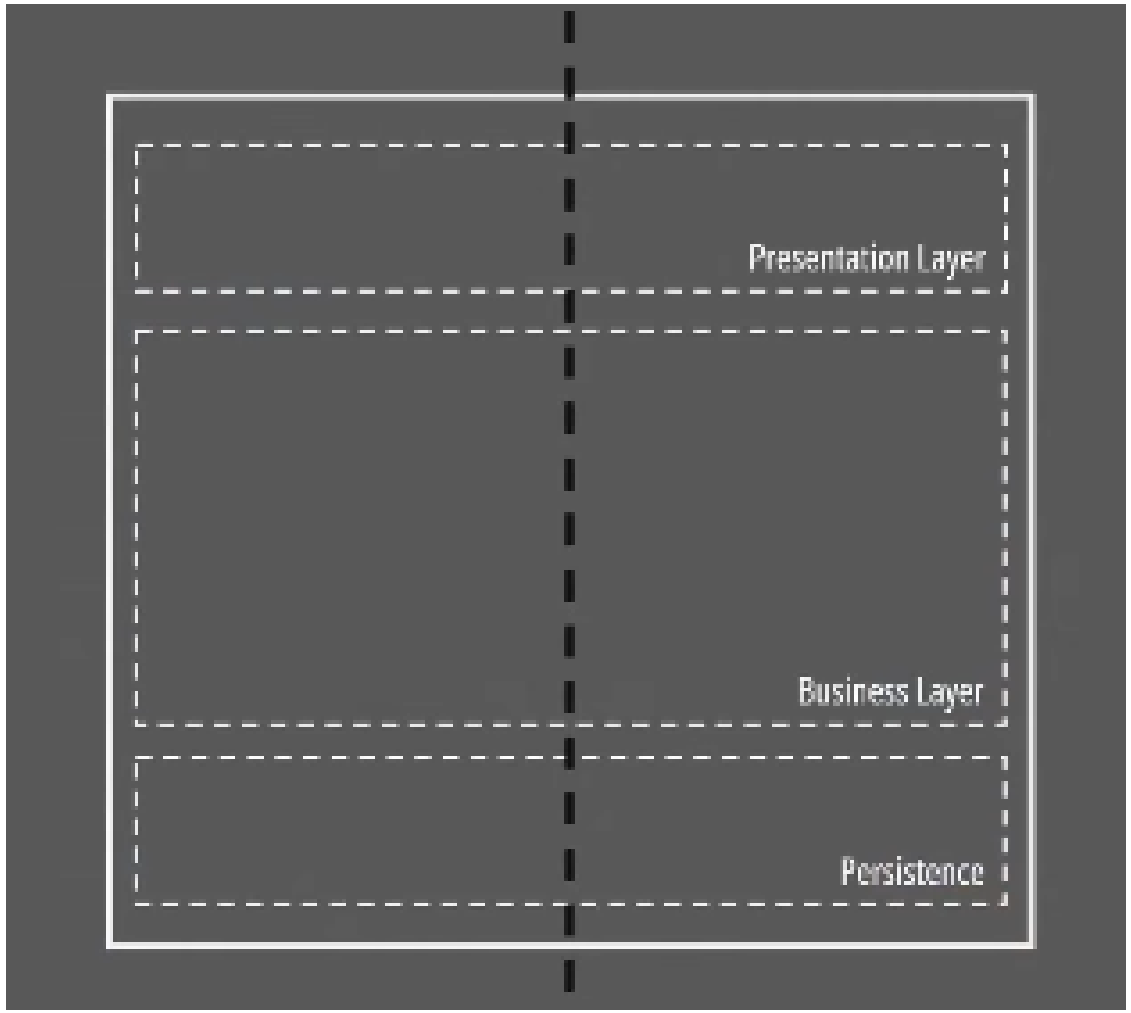
CQRS (Command Query Responsibility Segregation)



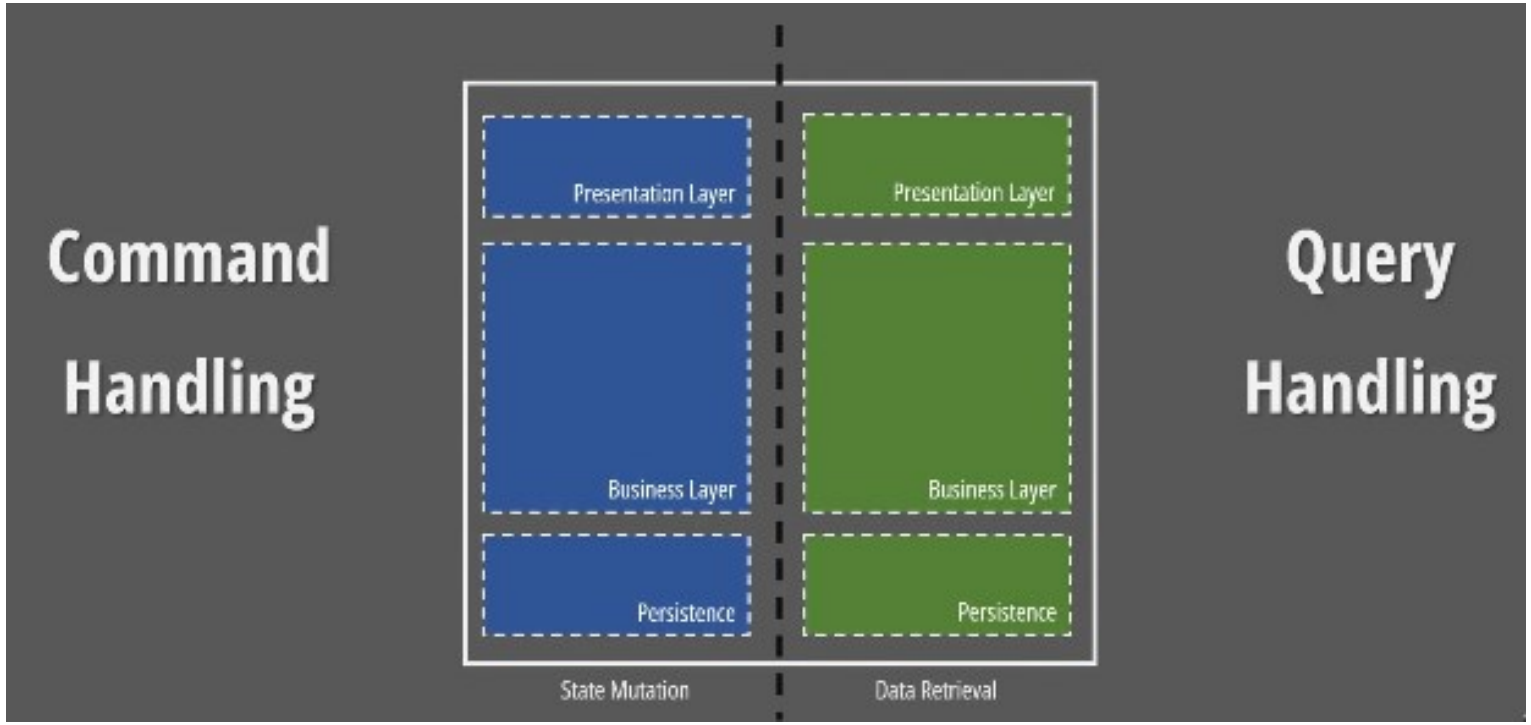
CQRS Document by Greg Young

- He discusses the combination of CQRS (Command Query Responsibility Segregation) and Event Sourcing, particularly within a system that employs Domain-Driven Design (DDD).
    1. **CQRS and Event Sourcing**: These two concepts are most effective when used together. CQRS allows Event Sourcing to serve as the data storage mechanism for the domain.
    2. **Query Limitations in Event Sourcing**: One major limitation of Event Sourcing is the inability to query the system for specific data, such as retrieving all users with a first name of "Greg," because it lacks a representation of the current state. CQRS addresses this by supporting only the GetById query within the domain, which is compatible with Event Sourcing.
    3. **Integration Challenges**: Integrating CQRS and Event Sourcing can be complex, especially when maintaining separate relational models for reading and writing data. This complexity increases when an event model is needed to synchronize the two.
    4. **Cost Efficiency with Event Sourcing**: Using Event Sourcing as the persistence model on the write side significantly reduces development costs by eliminating the need for model conversions.
    5. **System Components**: The text mentions various components such as Domain Objects, Application Services, Remote Facade, Request DTO (Data Transfer Object), and the process of sending commands and receiving status updates between the client and the system.

In a Layered architecture…



CQRS (Command Query Responsibility Segregation)



CQRS vs CRUD

- ***Read*** Operation mapped to ***Queries***
- ***Create Update Delete*** are **replaced** by ***Commands***

Commands are not data centric

CQRS vs CRUD

- A command represents the intention to perform a Task
- A command returns no data

Commands vs. Queries

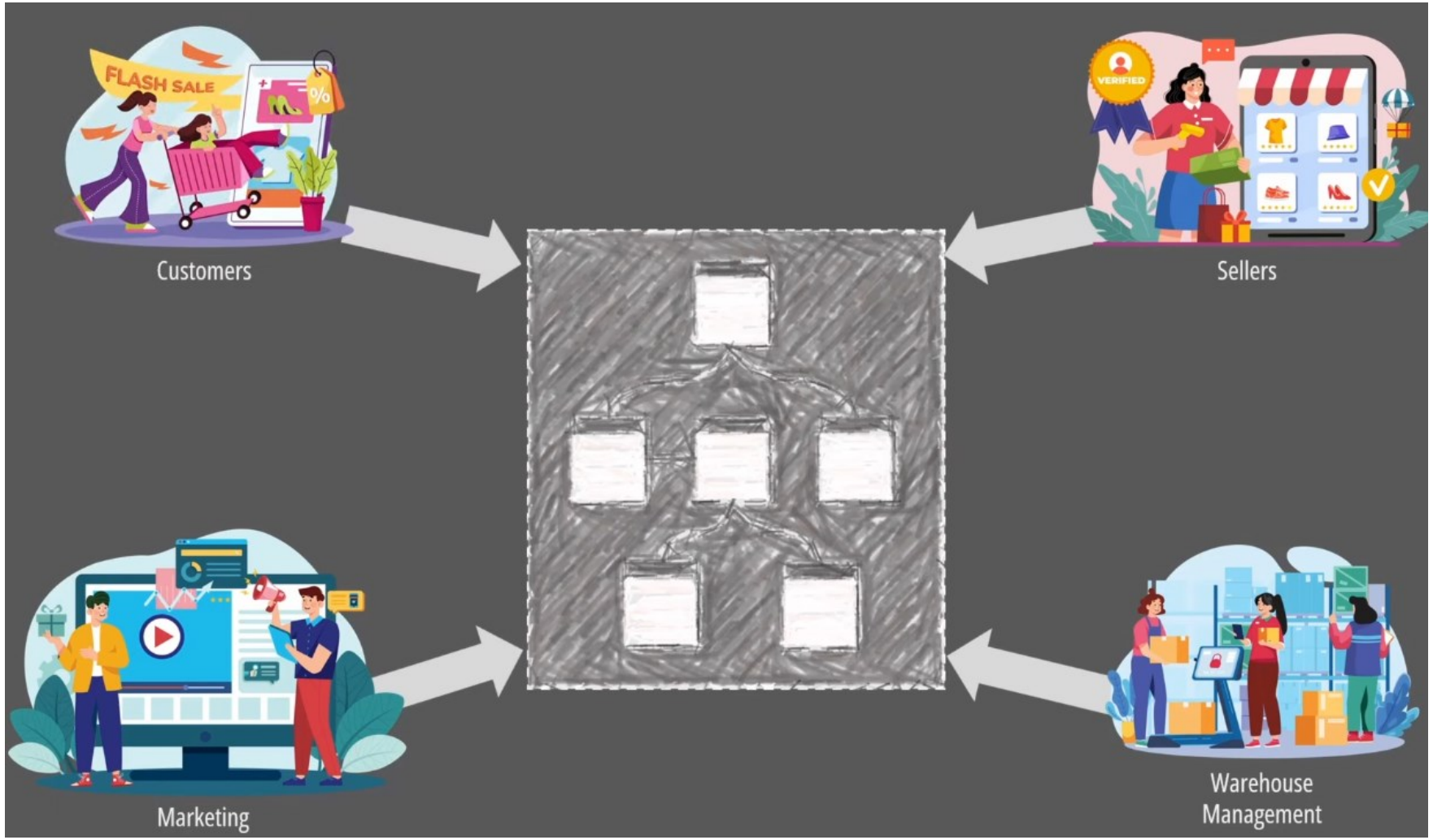
Commands:

- **Purpose:** Write or modify data.
- **Examples:**
  - Add a new user.
  - Update an order status.

Queries:

- **Purpose:** Retrieve data without modifying it.
- **Examples:**
  - Get a list of all orders.
  - Fetch user details.

In an Online Store..



CQRS Applicability

Writes

- Complex Validation
- Complex Business Rules
- Orchestration
- Asynchronous Flows
- Pessimistic Locks with High Contention

Reads

- Complex queries
- Multiple Table Joins
- Multiple views
- Cross-Domain Aggregation

CQRS Misconceptions

It is a complex system level architecture

Separate Databases for Reads and Writes are compulsory

It can be used only with DDD and Event Sourcing

CQRS Misconceptions

It is a complex system level architecture

Separate Databases for Reads and Writes are compulsory

It can be used only with DDD and Event Sourcing

MYTHS

Real-World Use Cases

1. **E-commerce Platforms:**
   - Handling inventory updates (commands) and product searches (queries).
2. **Financial Systems:**
   - Processing transactions (commands) and generating reports (queries).
3. **IoT Systems:**
   - Updating device settings(commands) and retrieving device data separately(queries).

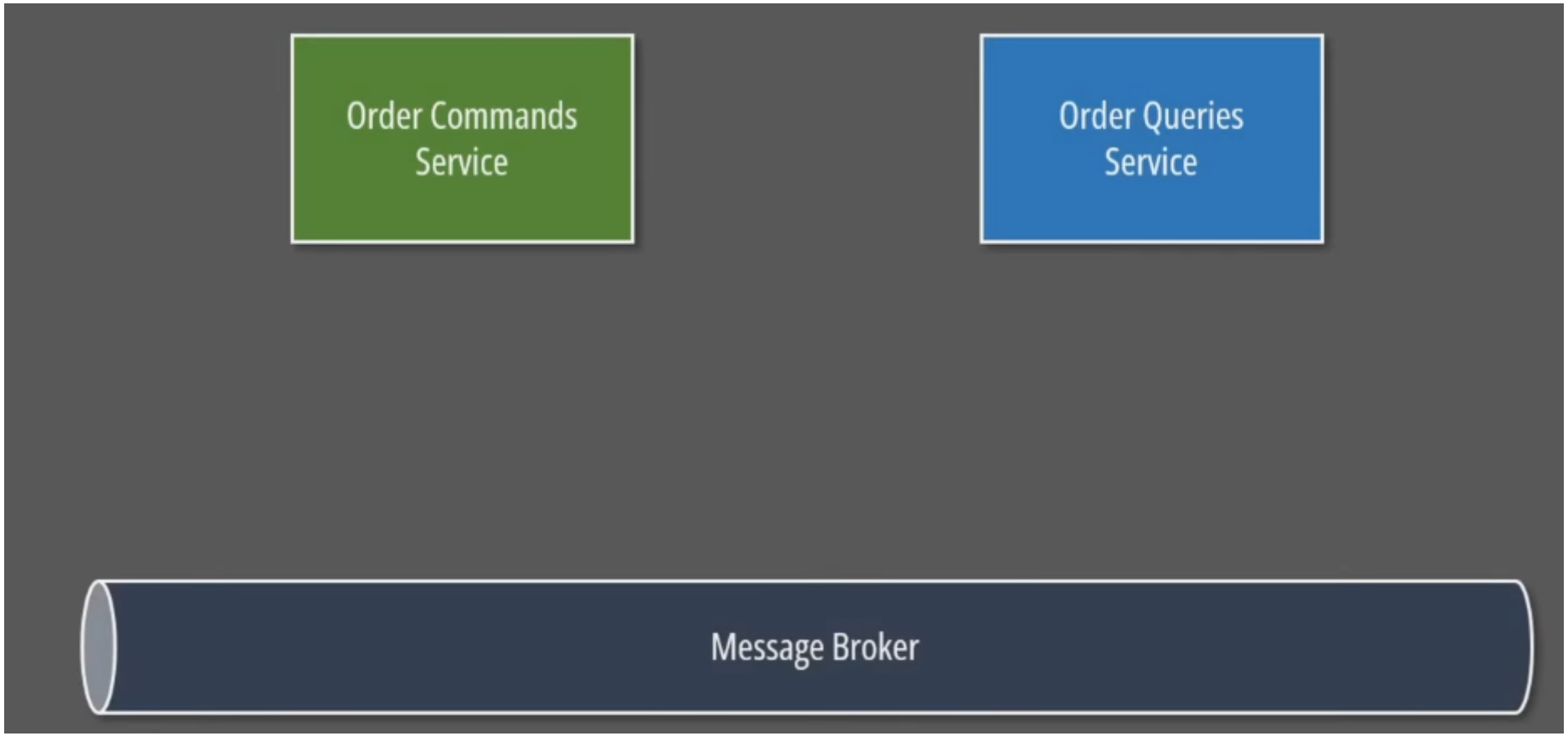
How CQRS Works

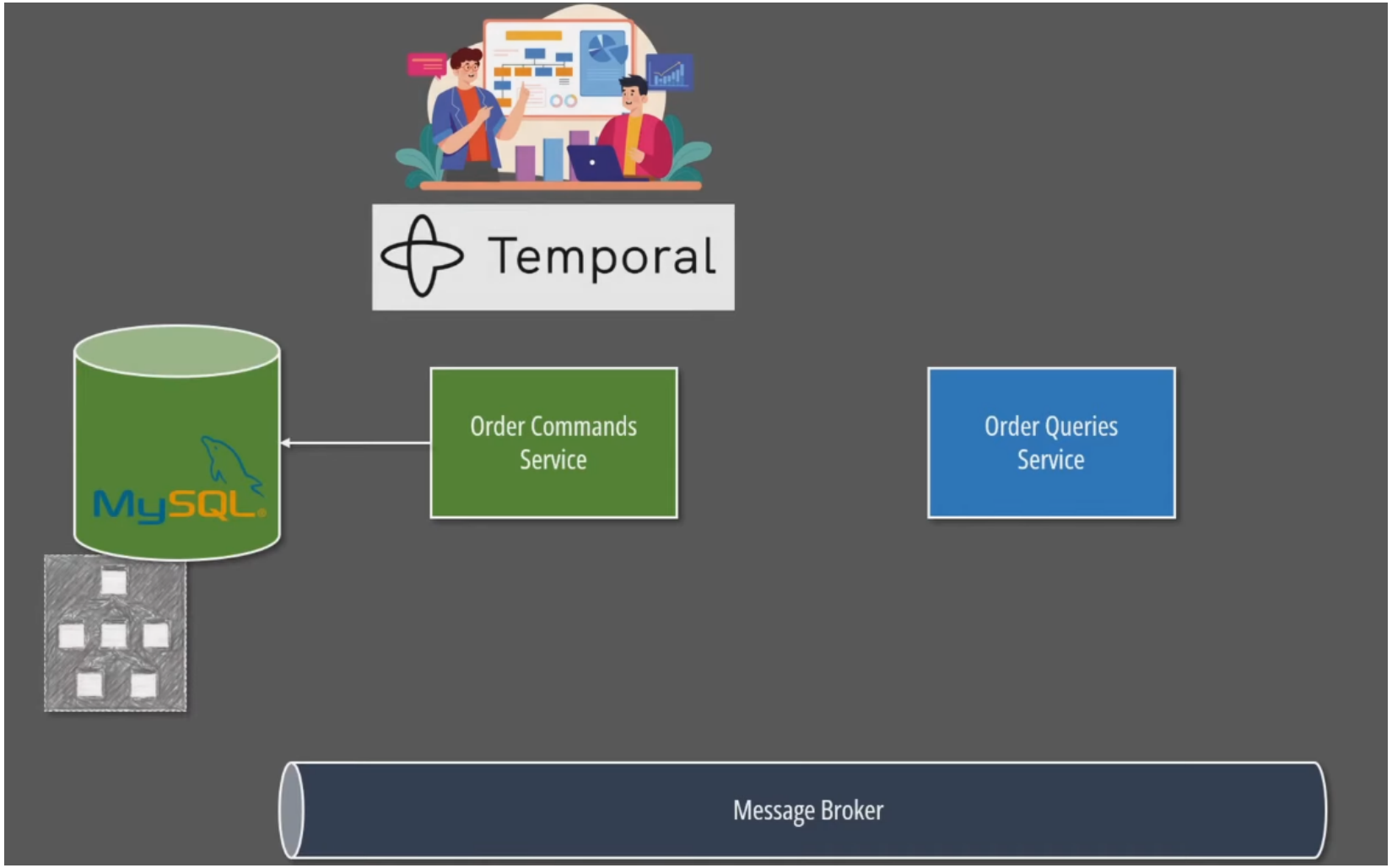
1. **User Action:**
   - A user sends a **command** to modify data.
2. **Write Model:**
   - Processes the command and updates the database.
3. **Event Notification:**
   - A message/event updates the **read model**.
4. **Query Model:**
   - Optimized for reading and serves user requests.

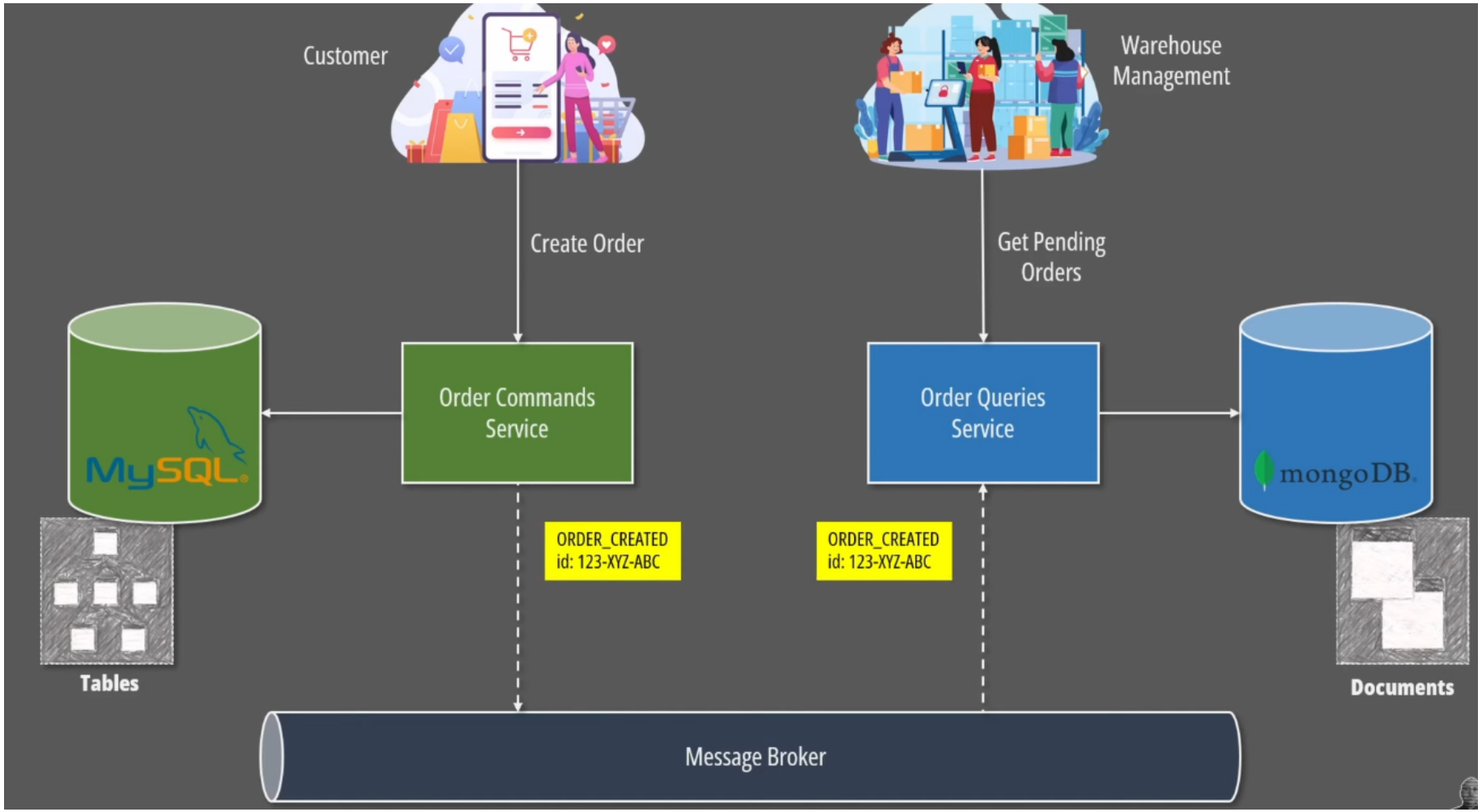
Tools for CQRS

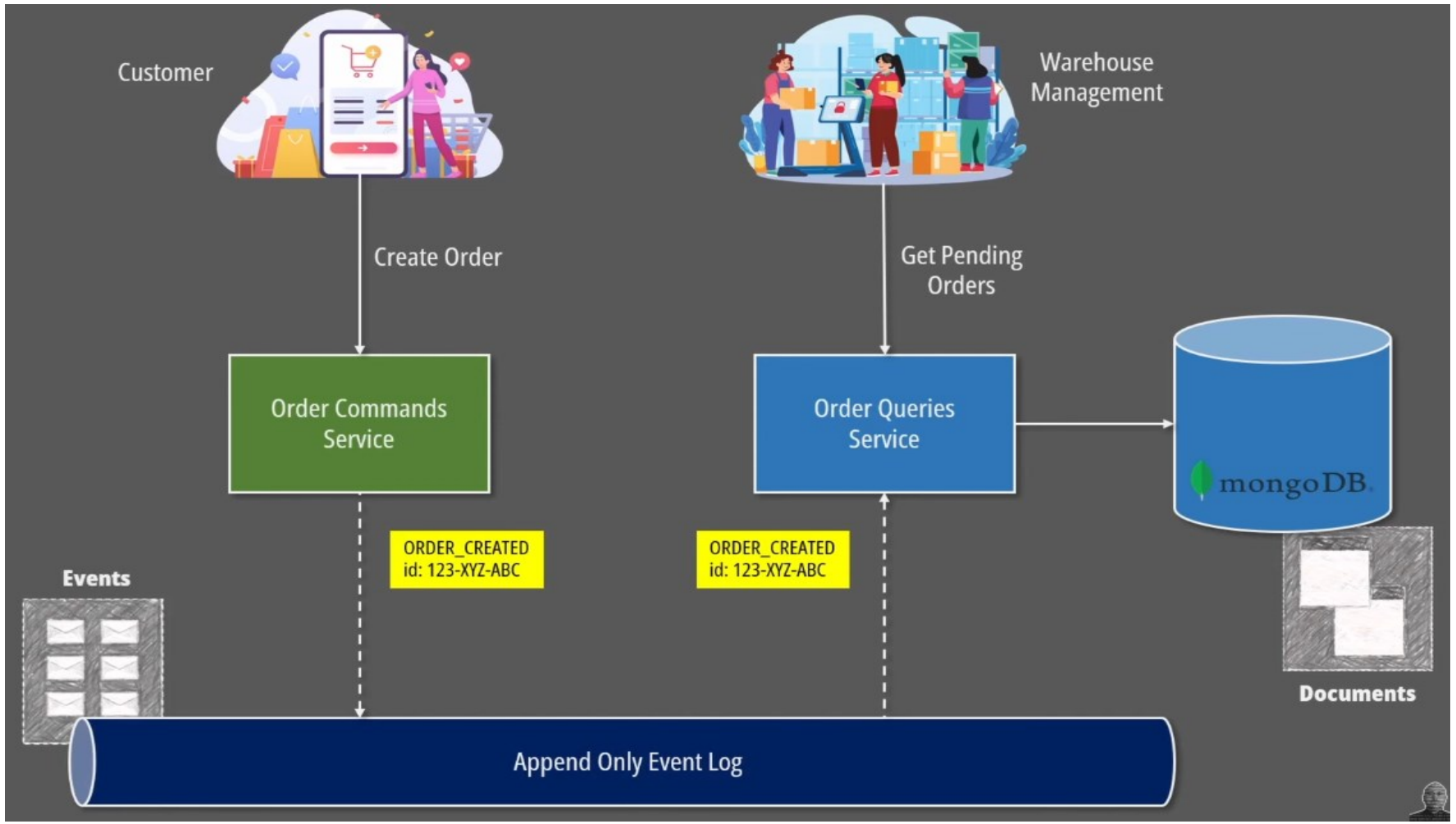
1. **Messaging Systems:**
   - RabbitMQ, Kafka for event communication.
2. **Frameworks:**
   - Axon Framework, MediatR.
3. **Databases:**
   - SQL for commands, NoSQL for queries.

https://www.axoniq.io/products/axon-framework


Order Commands Service
Order Queries Service
Message Broker


Temporal
MySQL
Order Commands Service
Order Queries Service
Message Broker


Customer
Warehouse Management
Create Order
Get Pending Orders
Order Commands Service
Order Queries Service
MySQL
mongoDB
ORDER_CREATED
id: 123-XYZ-ABC
ORDER_CREATED
id: 123-XYZ-ABC
Tables
Documents
Message Broker


Customer
Warehouse Management
Create Order
Get Pending Orders
Order Commands Service
Order Queries Service
mongoDB.
ORDER_CREATED
id: 123-XYZ-ABC
ORDER_CREATED
id: 123-XYZ-ABC
Events
Documents
Append Only Event Log

Benefits of CQRS

1. **Performance Optimization:**
   - Specialized models for reads and writes improve efficiency.
2. **Scalability:**
   - Write and read workloads can scale independently.
3. **Clear Separation:**
   - Simplifies complex domain logic by separating concerns.
4. **Flexibility in Databases:**
   - Allows using different databases or storage mechanisms for commands and queries.

Best Practices in Decision-Making

1. **Engage Stakeholders** early and regularly.
2. Use **reference architectures** to accelerate decisions.
3. Apply **Design Tactics**:
   - Availability: Redundancy.
   - Modifiability: Encapsulation.
4. Document **decision rationale** to ensure traceability.
5. Continuously **review and refine** decisions as the system evolves.

Real-World Example: E-Commerce Platform

- **Functional Requirements:**
  - User authentication.
  - Product search.
- **Non-Functional Requirements:**
  - Scalability for high traffic.
  - Security compliance (e.g., **Payment Card Industry(PCI), Data Security Standard(DSS)**.

Solution:

Architecture: Microservices with API Gateway.

Trade-offs: Performance optimized by caching; security ensured via OAuth.

Tools for Architectural Decision-Making

- **Architecture Tradeoff Analysis Method (ATAM).**
- **Quality Attribute Workshops (QAW).**
- **Modeling Tools:** UML, ArchiMate.
- **Documentation Tools:** C4(**context, containers, components, and code**) model, ADR (**Architectural Decision Records**).

https://insights.sei.cmu.edu/library/quality-attribute-workshop-collection/

A Quality Attribute Workshop (QAW)

- **A Quality Attribute Workshop (QAW)** is a facilitated method for evaluating a software-intensive system's architecture against critical quality attributes. QAWs are held early in the system development life cycle to identify the system's driving quality attributes.
- Some quality attributes include:
  - **Availability**: A common quality attribute for most systems
  - **Performance**: A common quality attribute for most systems
  - **Security**: A common quality attribute for most systems
  - **Interoperability**: A critical quality attribute
  - **Modifiability**: A critical quality attribute
  - **Usability**: A quality attribute that defines how easy it is for users to perform a task on the system
  - **Testability**: A quality attribute that matters when building and automating tests for individual components, interactions, and the system as a whole
  - **Portability**: A quality attribute that may be more specific to some systems
  - **Cost effectiveness**: A quality attribute that may be more specific to some system

C4 Model (Context, Containers, Components, Code)

- C4 stands for "***Context, Container, Component, and Code***," representing different levels of abstraction used to visualize a software system's architecture, allowing for detailed diagrams at various levels of granularity depending on the audience's needs.
- **Levels of Abstraction:**
  - **Context:** A high-level view showing the system in relation to external actors and other systems it interacts with.
  - **Container:** Zooms in to show the major building blocks of the system, like applications, databases, and external services.
  - **Component:** Further breaks down containers into individual components and their relationships within the system.
  - **Code:** Displays the detailed implementation level, focusing on specific classes and methods.

Architectural Decision Records (ADRs)

- An ADR is a document that captures the reasoning behind a significant architectural decision made during development, including the context, considered options, pros and cons, and rationale for the chosen solution.
- **Key Elements:**
  - **Decision Title:** A concise description of the architectural decision being documented.
  - **Context:** Explains the situation or problem that led to the decision.
  - **Decision:** States the chosen solution clearly.
  - **Rationale:** Details the reasoning behind the selected option, including pros and cons compared to other considered alternatives.
  - **Consequences:** Describes the potential impacts and trade-offs of the decision.

How they work together:

- **Visualizing Decisions:**
  - C4 diagrams can be used to visually represent the architectural components impacted by a specific ADR, providing a clear visual context for the decision made.
- **Documentation Alignment:**
  - By linking ADRs to relevant C4 diagrams, developers can easily access the rationale behind architectural choices while exploring the system's structure.

Summary

- **Architectural decision-making** aligns design with goals and constraints.
- Balancing functional and non-functional requirements is key to system success.
- Use structured methods, tools, and collaboration to achieve optimal outcomes.

Discussion Questions

1. How do you prioritize requirements in a high-pressure project?
2. What are common trade-offs you've experienced between performance and security?
3. Which tools are available for architectural modeling and analysis?

Trade-Offs in Software Architecture

Objectives

- **Understand** the nature of trade-offs in architectural decisions.
- **Explore** examples of common trade-offs (e.g., performance vs. scalability, security vs. flexibility).
- **Learn** strategies to evaluate and address trade-offs effectively.

What are Trade-Offs in Architecture?

- A situation where improving one quality attribute results in the degradation of another.
- **Why it Matters:**
  - Systems often have **conflicting goals**.
  - Resources (time, budget, hardware) are **limited**.
- **Examples of Conflicts:**
  - Performance vs. Scalability.
  - Security vs. Flexibility.
  - Usability vs. Security.

Common Trade-Off Examples

1. Performance vs. Scalability

- **Performance:** Focuses on reducing response times and maximizing throughput under current load conditions.
- **Scalability:** Ensures the system can handle increased loads by adding resources without degrading performance.
- **Conflict Example:**
  - Transitioning from a monolithic architecture to microservices can enhance scalability but may introduce network latencies, affecting performance.

Common Trade-Off Examples…

2. Security vs. Flexibility

- **Security:** Strong authentication and access controls.
- **Flexibility:** Ease of integration and user accessibility.
- **Conflict Example:**
  - Strict security measures may limit dynamic integrations with third-party systems.

Common Trade-Off Examples…

3. Usability vs. Security

- **Usability:** Simple, intuitive user experience.
- **Security:** Complex authentication and validation processes.(Multi-Factor Authentication)
- **Conflict Example:**
  - Multi-factor authentication may frustrate users but enhances security.

Common Trade-Off Examples…

4. Maintainability vs. Performance:

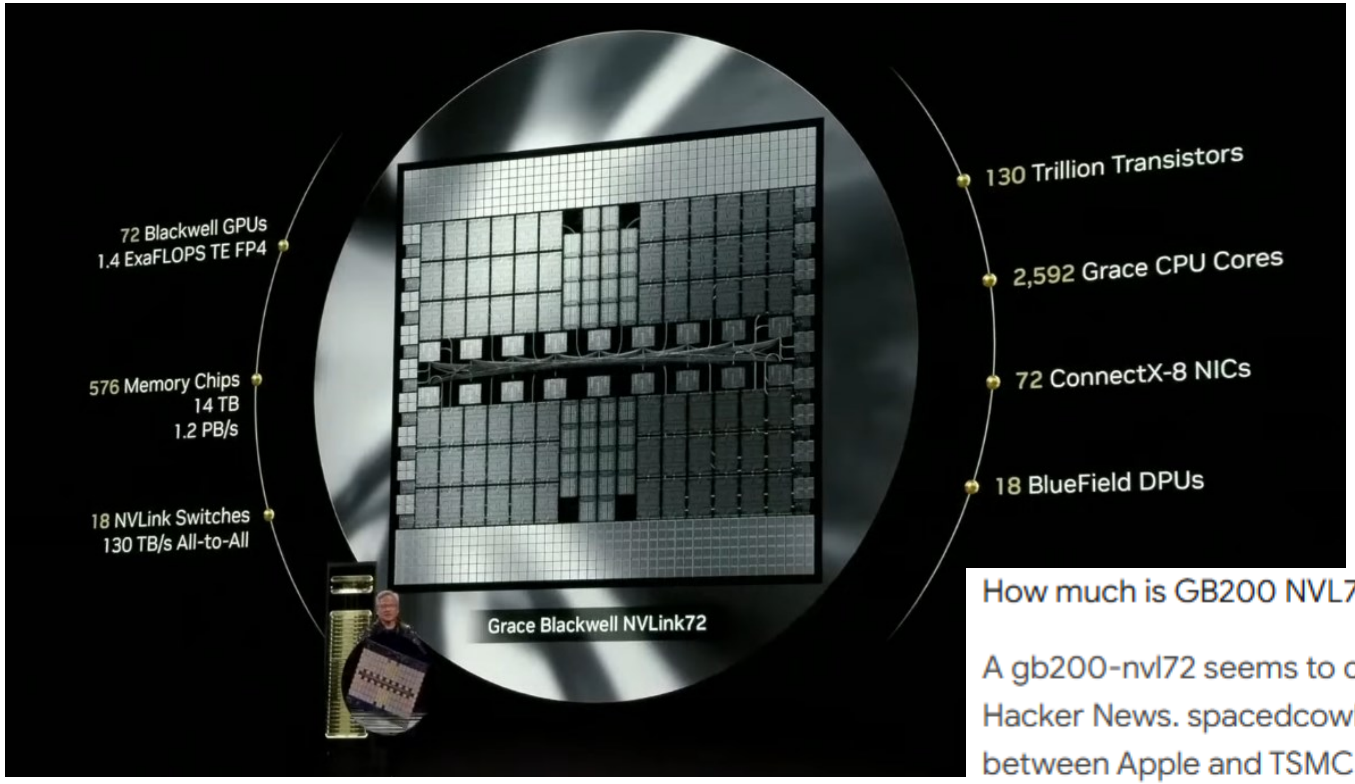
- **Maintainability:** Emphasizes modular, well-documented, and easily modifiable systems.
- **Performance:** Prioritizes speed, sometimes at the expense of clean design, leading to complex codebases.
- **Conflict Example:**
  - Highly optimized systems may achieve speed through low-level optimizations but become harder to maintain due to reduced readability.

Common Trade-Off Examples…

5. Cost vs. Reliability:

- **Cost:** Budget constraints may limit the implementation of redundant systems.
- **Reliability:** Ensuring high availability often requires significant investment in infrastructure.
- **Conflict Example:**
  - Startups may opt for single-server models to save costs, risking downtime during failures.

Nvidia Blackwell Megachip



How much is GB200 NVL72?

A gb200-nvl72 seems to cost [1] **about $3,000,000**. I expect Apple could throw tog... | Hacker News. spacedcowboy 7 months ago | parent | context | favorite | on: Secret meeting between Apple and TSMC reported to ... A gb200-nvl72 seems to cost [1] about $3,000,000.

Hacker News
https://news.ycombinator.com › item

A gb200-nvl72 seems to cost [1] about $3000000. I expect ... - Hacker News

Factors Influencing Trade-Offs

1. **Stakeholder Priorities:**
   - Different stakeholders value attributes differently (e.g., end-users prioritize usability, while IT teams prioritize security).
2. **System Context:**
   - The domain and usage scenarios (e.g., financial systems prioritize security over flexibility).
3. **Resource Constraints:**
   - Limited time, budget, or technology capabilities.

Strategies for Managing Trade-Offs

1. **Define Priorities:**
   - Use frameworks like the **Quality Attribute Utility Tree** to rank quality attributes.
2. **Use Architectural Patterns:**
   - Patterns like **CQRS** (Command Query Responsibility Segregation) can balance performance and scalability.
3. **Perform Trade-Off Analysis:**
   - Methods like **ATAM (Architecture Tradeoff Analysis Method)**.
4. **Iterative Prototyping:**
   - Test trade-offs with small-scale prototypes.

Quality Attribute Utility Tree

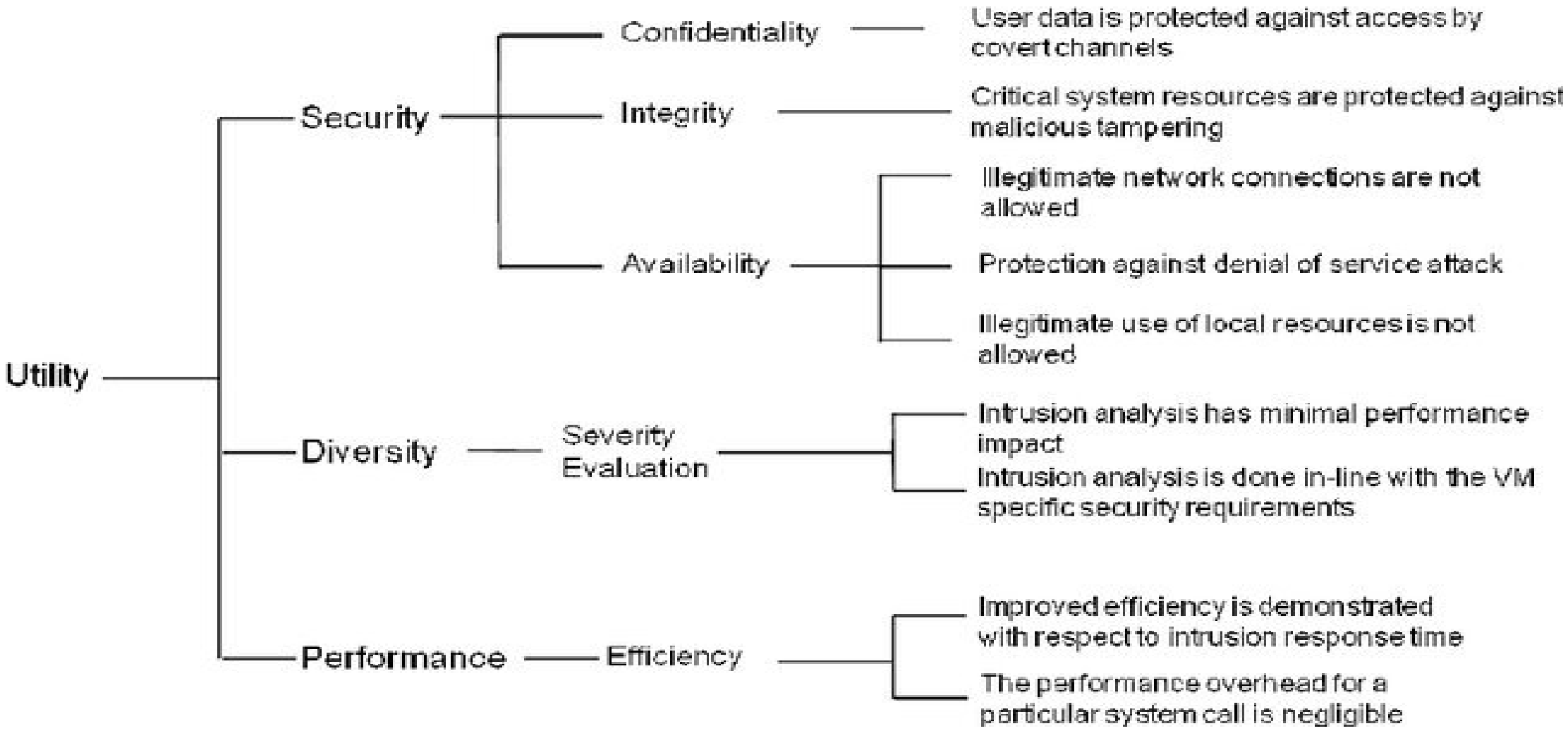


Fig: Quality Attribute Utility Tree

Real-World Case Study

Example: Online Video Streaming Service

- **Trade-Off:**
  - Performance: Low latency for video playback.
  - Scalability: Millions of concurrent users.
- **Solution:**
  - Implement **Content Delivery Networks (CDNs)** to cache content closer to users.
  - Trade-off resolved by optimizing resource placement.

Tools for Trade-Off Analysis

1. **ATAM:** Architecture Tradeoff Analysis Method.
2. **Scenario-based Evaluation:**
    - Create scenarios for performance, scalability, and security.
3. **Modeling Tools:** UML, ArchiMate.
4. **Prototyping and Simulation Tools:**
    - Tools for testing scalability and performance.

Architecture Tradeoff Analysis Method (ATAM)

- The Architecture Tradeoff Analysis Method (ATAM) is a structured approach for evaluating software architectures, focusing on how well they satisfy quality attribute requirements such as ***performance, modifiability, security, and availability***.
- Developed by the Software Engineering Institute at Carnegie Mellon University, ATAM helps identify potential risks and trade-offs in architectural decisions.

Objectives of ATAM:

- Assess the consequences of architectural decisions relative to quality attribute requirements.
- Identify risks early in the development process.
- Facilitate communication among stakeholders.
- Clarify quality attribute requirements.
- Improve architectural documentation.

ATAM Process Overview:

- ATAM consists of nine steps, typically divided into **two** phases:

Phase 1: Evaluation Preparation

1. **Present ATAM:** Introduce the ATAM process to stakeholders, outlining objectives and procedures.
2. **Present Business Drivers:** Stakeholders discuss system functionality, goals, constraints, and desired quality attributes.
3. **Present Architecture:** The architect provides an overview of the system's architecture, detailing key components and interactions.
4. **Identify Architectural Approaches:** Discuss and document different architectural strategies considered for the system.
5. **Generate Quality Attribute Utility Tree:** Define and prioritize quality attributes, creating scenarios that exemplify these attributes.
6. **Analyze Architectural Approaches:** Evaluate how well the architecture supports each scenario, identifying sensitivity and trade-off points.

Phase 2: Evaluation Analysis

7. Brainstorm and Prioritize Scenarios: Engage a broader stakeholder group to generate additional scenarios and prioritize them based on importance.

8. Analyze Architectural Approaches (Revisited): Reassess the architecture with the new scenarios, refining the analysis of risks and trade-offs.

9. Present Results: Compile findings, including identified risks, sensitivity points, trade-offs, and recommendations, and present them to stakeholders.

Benefits of ATAM:

- Early identification of architectural risks.
- Enhanced stakeholder communication and understanding.
- Improved alignment between architecture and business goals.
- Informed decision-making regarding trade-offs between quality attributes.

Follow this link: download the paper and study
https://insights.sei.cmu.edu/library/atam-method-for-architecture-evaluation/

Creating Scenarios for Performance, Scalability, and Security:

1.Performance:

- *Scenario:* A sudden surge in user activity during a promotional event.
- *Evaluation Focus:* Assess the system's response time and throughput under increased load.

2.Scalability:

- *Scenario:* The user base doubles over six months.
- *Evaluation Focus:* Determine if the system can accommodate growth without degradation.

3.Security:

- *Scenario:* An attempted breach using SQL injection.
- *Evaluation Focus:* Evaluate the system's ability to detect and prevent such attacks.

Modeling Tools: UML and ArchiMate:

- **Unified Modeling Language (UML):**
  - A standardized modeling language used to visualize the design of a system.
  - Supports various diagrams like class, sequence, and activity diagrams.
  - *Tools:* StarUML, Enterprise Architect.
- **ArchiMate:**
  - An open-standard modeling language for enterprise architecture.
  - Provides a comprehensive approach to describe, analyze, and visualize relationships among business domains.
  - *Tools:* Archi, Enterprise Architect.

Prototyping and Simulation Tools for Testing Scalability and Performance:

- **Load Testing Tools:**
  - Simulate multiple users to assess system performance under load.
  - *Examples:* Apache JMeter, LoadRunner.
- **Stress Testing Tools:**
  - Evaluate system behavior under extreme conditions.
  - *Examples:* Stress-ng, Locust.
- **Performance Monitoring Tools:**
  - Monitor system metrics during testing to identify bottlenecks.
  - *Examples:* New Relic, Dynatrace.

Discussion Questions

1. How do you prioritize trade-offs in a time-sensitive project?
2. Can you think of a scenario where security should take precedence over usability?
3. Which trade-offs are most relevant in cloud-based architectures?

Summary

- Trade-offs are **inevitable** in software architecture.
- Effective trade-off management requires **prioritization, analysis, and testing**.
- Use structured frameworks and tools to make **informed decisions**.

Risk Analysis and Mitigation Strategies in Architecture

- Ensuring Robust and Reliable Software Systems

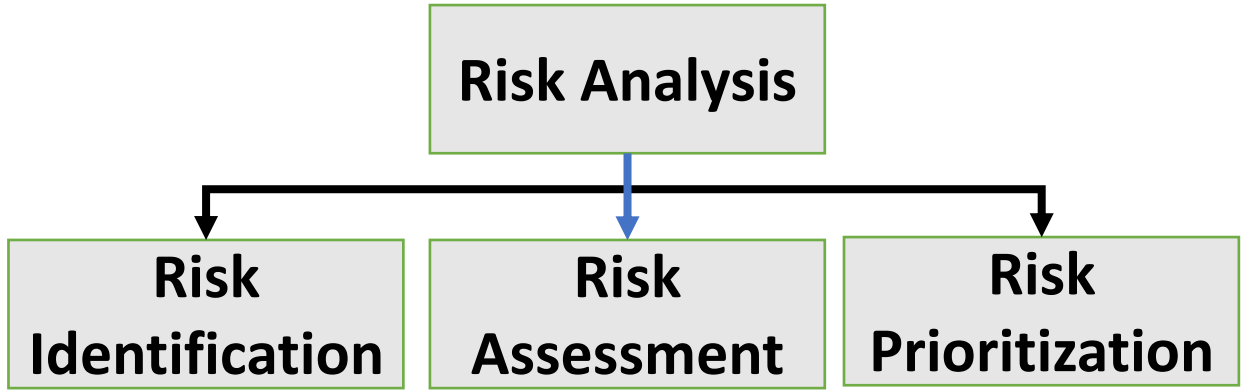
Objectives

- **Understand** the importance of risk analysis in software architecture.
- **Learn** key risk analysis techniques and tools.
- **Explore** effective mitigation strategies.
- **Discuss** real-world examples of risk management.

What is Risk Analysis in Architecture?

- The process of identifying, assessing, and prioritizing risks that could impact system architecture.
- **Why It Matters:**
  - Prevent cost overruns and delays.
  - Ensure system quality attributes like reliability and security.
- **Common Sources of Risk:**
  - Changing requirements.
  - Technology obsolescence.
  - Scalability and performance challenges.

Risk Analysis Process



1.Risk Identification

- Tools: Brainstorming, checklists, stakeholder interviews.
- Examples:
  - Scalability risks for high-traffic systems.
  - Security risks for sensitive data handling.

2.Risk Assessment

- **Likelihood:** Probability of occurrence.
- **Impact:** Severity of consequences.
- Tools: Risk matrices, FMEA (Failure Modes and Effects Analysis).

3.Risk Prioritization

- Categorize risks as High, Medium, or Low.
- Focus on high-priority risks first.

Common Risks in Software Architecture

1.Performance Risks:

- Poor response times under load.
- Mitigation: Load testing, caching strategies.

2.Security Risks:

- Vulnerabilities to attacks (e.g., SQL injection, DDoS).
- Mitigation: Penetration testing, secure coding practices.

3.Scalability Risks:

- Inability to handle increased users.
- Mitigation: Cloud-based scaling, microservices.

4.Integration Risks:

- Issues with third-party systems or APIs.
- Mitigation: Use integration testing and modular design.

5.Maintainability Risks:

- High complexity and poor documentation.
- Mitigation: Modular architecture, proper documentation.

Mitigation Strategies

1. **Risk Avoidance**
   - Use proven frameworks instead of custom solutions.
2. **Risk Reduction**
   - Implement redundancy to reduce downtime risk.
3. **Risk Sharing**
   - Outsource non-core functionalities to reliable vendors.
4. **Risk Acceptance**
   - Accept minor risks that have low impact and likelihood.

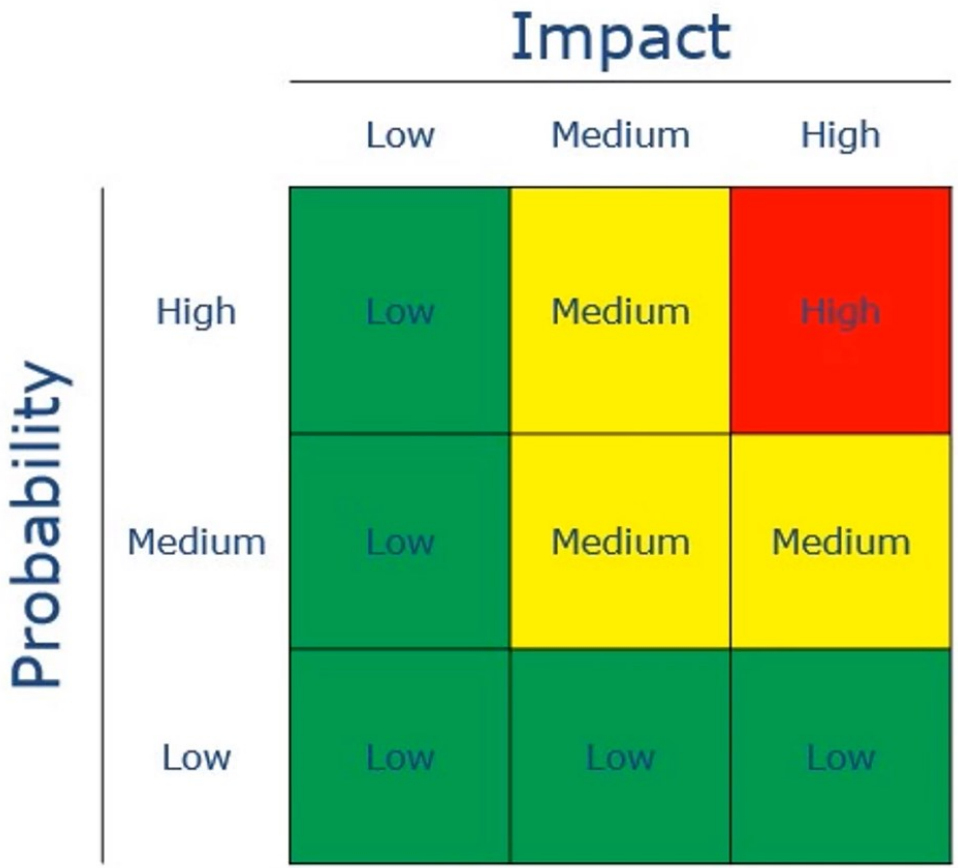
Risk Analysis Tools

1. **Risk Matrix:**
   - Visualize risks based on likelihood and impact.
   - High-priority risks in the red zone.
2. **ATAM (Architecture Tradeoff Analysis Method):**
   - Evaluate architecture’s ability to meet quality attributes.
3. **FMEA:**
   - Analyze potential failure modes and their effects.
4. **Fault Tree Analysis (FTA):**
   - Identify root causes of potential failures.

3 X 3 Risk Assessment Matrix

- **Likelihood (Probability)**
  - High (3): This risk is highly likely to occur.
  - Medium (2): This risk may occur occasionally.
  - Low (1): This risk is unlikely to occur.
- **Severity (Consequence)**
  - High (3): This risk would result in significant harm or damage.
  - Medium (2): This risk would cause moderate harm or damage.
  - Low (1): This risk would result in minor harm or damage.

3 x 3 Risk Assessment Matrix

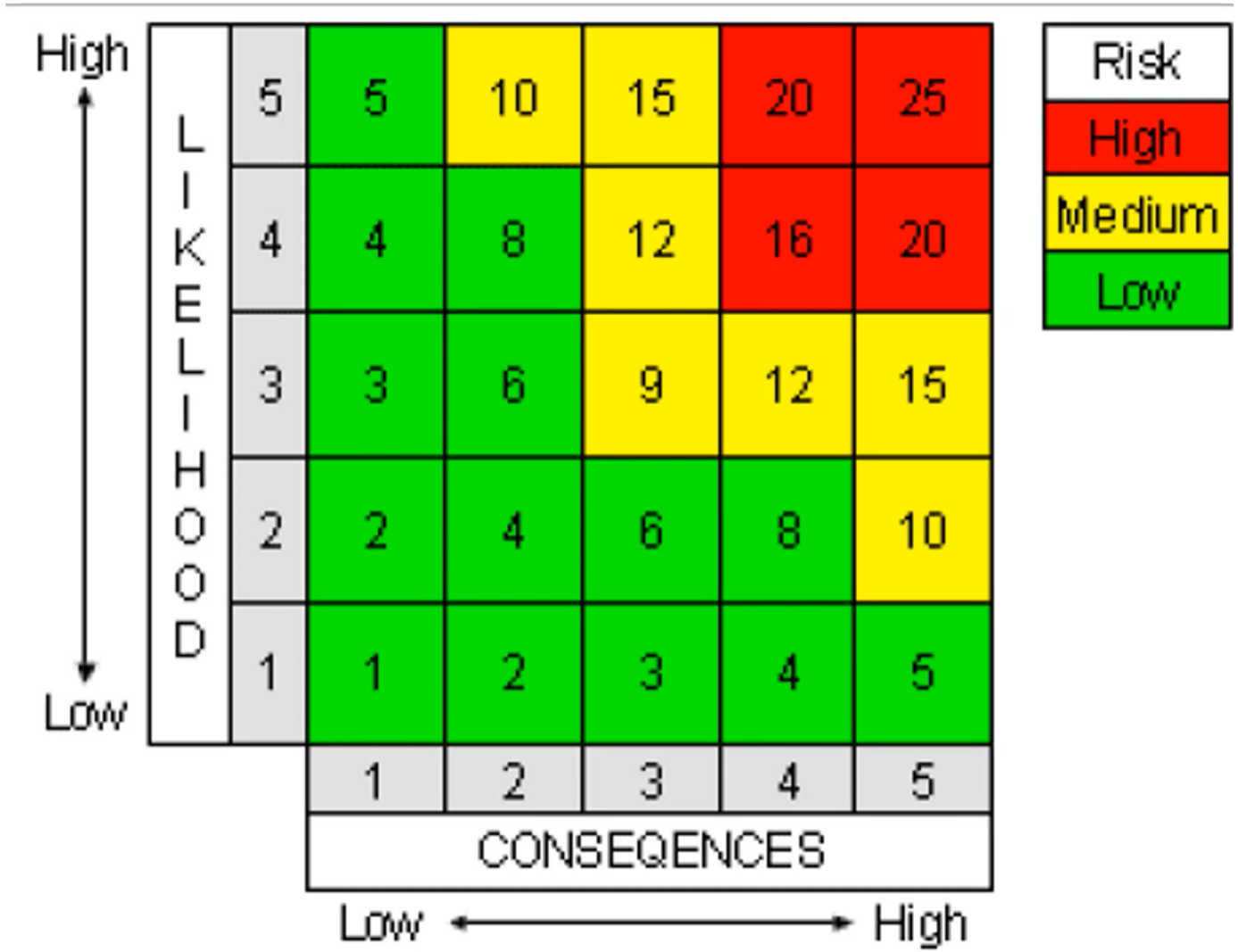


- "Low Risk" represents risks that are not very likely to occur and, if they do, would result in minimal harm or damage.
- "Medium Risk" represents risks that have a moderate likelihood of occurring and could cause moderate harm or damage.
- "High Risk" represents risks that are highly likely to occur and would result in significant harm or damage.

5 X 5 Risk Assessment Matrix

- Likelihood (Probability)
  - Very Low (1): This risk is extremely unlikely to occur.
  - Low (2): This risk has a low probability of occurring.
  - Moderate (3): This risk has a moderate probability of occurring.
  - High (4): This risk is likely to occur.
  - Very High (5): This risk is highly likely to occur.
- Severity (Consequence)
  - Very Low (1): This risk would result in minimal or no harm or damage.
  - Low (2): This risk would cause minor harm or damage.
  - Moderate (3): This risk would cause moderate harm or damage.
  - High (4): This risk would result in significant harm or damage.
  - Very High (5): This risk would result in severe or catastrophic harm or damage.

5 X 5 Risk Assessment Matrix



5 X 5 Risk Assessment Matrix

- "Very Low Risk" represents risks that are extremely unlikely to occur and, if they do, would result in minimal or no harm or damage.
- "Low Risk" represents risks that have a low likelihood of occurring and would cause minor harm or damage.
- "Moderate Risk" represents risks that have a moderate likelihood of occurring and would cause moderate harm or damage.
- "High Risk" represents risks that are likely to occur and would result in significant harm or damage.
- "Very High Risk" represents risks that are highly likely to occur and would result in severe or catastrophic harm or damage.

Real-World Case Study

Example: Financial Transaction System

- **Risk Identified:**
  - High latency during peak hours.
- **Mitigation Strategy:**
  - Adopt a cloud-based architecture with auto-scaling.
  - Implement caching for frequently accessed data.
- **Outcome:**
  - Reduced downtime and improved user experience.

Best Practices in Risk Management

1. **Engage Stakeholders Early:**
   - Gather inputs from developers, users, and business owners.
2. **Use Iterative Development:**
   - Identify and address risks incrementally.
3. **Monitor Continuously:**
   - Regularly reassess risks during the project lifecycle.
4. **Document Risks:**
   - Maintain a risk register for transparency and traceability.

Discussion Questions

1. What techniques do you use to identify architectural risks?
2. How do you prioritize risks in a time-sensitive project?
3. Can you share an example where a mitigation strategy failed? How did you adapt?

Summary

- Risk analysis is **critical** to ensuring system reliability and robustness.
- Use structured processes and tools to identify, assess, and prioritize risks.
- Apply mitigation strategies tailored to the system context.
- Continuously monitor and adapt to evolving risks.

Justifying Architectural Decisions and Documenting Trade-Offs

- Key Practices for Effective Software Architecture

Objectives

- **Understand** the importance of justifying architectural decisions.
- **Learn** techniques for documenting trade-offs.
- **Explore** examples and tools to support decision-making and communication.

Why Justify Architectural Decisions?

- **Build Stakeholder Confidence:**
  - Ensure transparency and trust in the design process.
- **Improve Decision Quality:**
  - Make informed choices aligned with project goals.
- **Facilitate Communication:**
  - Provide clear reasoning to team members and stakeholders.
- **Enable Maintenance:**
  - Help future teams understand and evolve the system.

Framework for Justifying Decisions

1. **Define Context**
    - Identify **problem statements** and constraints.
    - Align decisions with **business goals** and quality attributes.
2. **List Alternatives**
    - Evaluate potential architectural solutions.
    - Examples:
        - Monolithic vs. Microservices.
        - Relational vs. NoSQL databases.

Framework for Justifying Decisions…

3. Assess Trade-Offs

- Analyze impacts on functional and non-functional requirements.
- Tools: **ATAM (Architecture Tradeoff Analysis Method).**

4. Make the Decision

- Use quantitative and qualitative data.
- Engage stakeholders for feedback and consensus.

5. Document the Rationale

- Record the "**why**" behind the choice.
- Include benefits, drawbacks, and rejected alternatives.

Documenting Trade-Offs

- **What Are Trade-Offs?**
  - Balancing competing priorities to meet system objectives.
  - **Examples:**
    - Performance vs. Scalability.
    - Security vs. Usability.

Why Document?

- Ensure **transparency** and traceability.
- Provide a **historical record** for future reference.
- Support **audit and compliance** requirements.

Techniques for Documenting Trade-Offs

1.Architectural Decision Records (ADR):

- Template:
  - Title.
  - Context.
  - Decision.
  - Consequences.

2.Decision Trees:

- Visual representation of alternatives and outcomes.

3.Trade-Off Matrices:

- Compare alternatives based on key criteria.

4.Scenarios and Quality Attribute Workshops:

- Use scenarios to evaluate impacts of trade-offs.

Real-World Example

Case: Cloud Migration

- **Context:**
  - A legacy system needs to be moved to the cloud.
- **Alternatives Considered:**
  - Lift-and-shift vs. re-architecting for microservices.
- **Trade-Offs:**
  - Lift-and-shift:
    - Low cost, faster migration.
    - Less scalability and cloud optimization.
  - Microservices:
    - High scalability and flexibility.
    - Longer development time and higher cost.
- **Decision:**
  - Opted for a phased microservices migration based on long-term goals.
- **Documentation:**
  - ADRs created for each migration phase.

Tools for Decision Documentation

1. **ADRs (Architectural Decision Records):**
   - Markdown-based templates for simple, effective documentation.
2. **C4 Model:**
   - Context, Containers, Components, and Code diagrams for architectural views.
3. **Trade-Off Analysis Tools:**
   - Tools like ATAM or scenario-based evaluation.
4. **Modeling Software:**
   - Tools like ArchiMate, UML diagrams, etc.

Discussion Questions

1. How do you decide which trade-offs are acceptable for a given project?
2. Can you share an example where documenting a decision helped resolve a conflict?
3. What tools do you use to justify and document architectural decisions?

Summary

- Justifying architectural decisions is **critical** for transparency and alignment.
- Documenting trade-offs ensures **accountability** and provides a foundation for future improvements.
- Use structured processes, tools, and collaboration to make informed and defensible decisions.

Case Study – A System Developed Using Microservices Architecture

- Exploring the Design, Challenges, and Benefits of Microservices

Objectives

- **Understand** the principles of Microservices Architecture (MSA).
- **Examine** a real-world case study of a system developed using MSA.
- **Learn** about the challenges, solutions, and outcomes in implementing MSA.

Microservices Architecture

- An architectural style that structures an application as a collection of loosely coupled, independently deployable services.
- **Key Principles:**
  - **Single Responsibility:** Each service focuses on a specific business function.
  - **Independence:** Services are self-contained and communicate over APIs.
  - **Scalability:** Independent scaling for different services.
  - **Resilience:** Failure in one service doesn’t bring down the entire system.

Case Study Overview

System: E-Commerce Platform

- **Goals:**
  - Provide a scalable and resilient platform for online shopping.
  - Enable independent deployment and scaling of features like search, payment, and inventory.
- **Architectural Style:** Microservices Architecture
- **Technologies:**
  - **Backend:** Java and Node.js services.
  - **Frontend:** Angular.
  - **Infrastructure:** Kubernetes for orchestration, Docker for containerization.
  - **Communication:** REST APIs, asynchronous messaging with RabbitMQ.

System Design

Microservices Overview

1. **User Service:**
   - Handles authentication, user profiles.
2. **Product Service:**
   - Manages product catalog and inventory.
3. **Order Service:**
   - Processes orders, tracks delivery.
4. **Payment Service:**
   - Integrates with payment gateways.
5. **Search Service:**
   - Full-text search for products.
6. **Notification Service:**
   - Sends email and SMS notifications.

Challenges in Development

1. Communication Overhead

- **Problem:** Increased complexity due to service interactions.
- **Solution:**
  - Use API gateways for external communication.
  - Implement asynchronous communication using RabbitMQ.

2. Data Consistency

- **Problem:** Maintaining consistency across distributed services.
- **Solution:**
  - Use **event sourcing** and **CQRS**.
  - Implement eventual consistency where necessary.

Challenges in Development…

3. Deployment and Monitoring

- **Problem:** Managing multiple services in production.
- **Solution:**
  - Use Kubernetes for orchestration.
  - Monitor with tools like Prometheus and Grafana.

Benefits Achieved

1.Scalability:

- Services scaled independently based on load (e.g., Search and Order services).

2.Resilience:

- Failures in Payment Service didn’t impact Product or Search services.

3.Faster Development:

- Teams worked in parallel on different services.

4.Improved Deployment:

- Continuous Deployment enabled rapid updates for individual services.

Key Takeaways

- **Microservices Architecture** enables scalability, resilience, and modularity.
- Effective **communication and monitoring** strategies are essential for success.
- **Trade-offs:**
  - Increased complexity and operational overhead.
  - Requires careful planning for data consistency and inter-service communication.

Tools and Best Practices

1. **Orchestration:** Kubernetes.
2. **Containerization:** Docker.
3. **Communication:** REST APIs and RabbitMQ for messaging.
4. **Monitoring:** Prometheus, Grafana, and Elastic Stack.
5. **Documentation:** OpenAPI for API documentation.

Discussion Questions

1. What are the primary benefits of using Microservices Architecture for large-scale systems?
2. How would you handle the trade-offs of complexity in microservices?
3. Can you think of scenarios where microservices may not be the best choice?

Summary

- Microservices Architecture allows building scalable, resilient, and modular systems.
- Proper tools and strategies can address challenges like data consistency and monitoring.
- The e-commerce case study demonstrates the real-world application of MSA principles.