ADR 1: Microservices vs. Monolithic Architecture

- **Date**: 2025-03-25
- **Status**: Accepted
- **Context**:
  The LMS platform needs to be scalable, resilient, and maintainable. A choice must be made between using a monolithic or a microservices architecture.
- **Decision**:
  We have decided to implement a **microservices architecture** for the LMS.
- **Consequences**:
  - **PROS**:
    - Scalability: Microservices allow independent scaling of services, such as User, Course, and Payment.
    - Resilience: Failure in one service (e.g., Payment) will not impact the entire platform.
    - Independent Deployment: Each service can be developed, deployed, and maintained independently.
  - **CONS**:
    - Complexity: Microservices introduce the complexity of inter-service communication.
    - Overhead: Managing multiple services requires more infrastructure and monitoring.

ADR 2: Authentication and Authorization using OAuth2

- **Date**: 2025-03-25
- **Status**: Accepted
- **Context**:
  The platform must have secure and reliable user authentication, supporting role-based access control (Admin, Instructor, Student). OAuth2 is widely used, but we need to justify why it's the best choice.

- **Decision**:
  We will implement **OAuth2** for user authentication and role-based access.
- **Consequences**:
  - **PROS**:
    - Secure and scalable authentication, with easy integration of third-party login providers (e.g., Google).
    - Token-based authentication supports role-based access control for different user types.
  - **CONS**:
    - Implementation complexity: OAuth2 setup requires proper management of tokens and security.
    - External dependency: The system relies on the availability and integrity of third-party authentication providers (e.g., Google).

ADR 3: Message Broker for Asynchronous Communication

- **Date**: 2025-03-25
- **Status**: Accepted
- **Context**:
  The platform needs to handle asynchronous tasks such as notification delivery and payment retries. A decision is needed on which message broker to use and whether it should be implemented.
- **Decision**:
  We will use **RabbitMQ** as the message broker for asynchronous communication between services.
- **Consequences**:
  - **PROS**:
    - Allows decoupling of services, enabling asynchronous operations (e.g., notifications, payment retries).
    - Enhances resilience by ensuring reliable message delivery and enabling retries.

  - **CONS**:
    - Requires additional infrastructure and management of the message broker.
    - Complexity in managing and monitoring queues and message processing.

ADR 4: CDN for Dynamic Content Delivery

- **Date**: 2025-03-25
- **Status**: Accepted
- **Context**:
  The platform will serve media-heavy content such as course videos, which need to be delivered efficiently to users globally. A decision is needed on how to distribute this content.
- **Decision**:
  We will implement a **Content Delivery Network (CDN)** for dynamic content delivery (e.g., videos, PDFs).
- **Consequences**:
  - **PROS**:
    - Faster content delivery by caching files at edge servers close to users.
    - Reduces load on the origin server, improving performance.
  - **CONS**:
    - Additional costs for using third-party CDN services.
    - Requires setup and configuration of caching rules.

ADR 5: Retry Mechanism for Payment Enrollment

- **Date**: 2025-03-25
- **Status**: Accepted

- **Context**:
  Payment transactions may occasionally fail due to network issues or other transient problems. A decision is required on how to handle failed payment attempts.
- **Decision**:
  We will implement a **retry mechanism** for failed payment enrollments.
- **Consequences**:
  - **PROS**:
    - Improves user experience by automatically retrying failed transactions, reducing friction for the user.
    - Increases payment success rate for users with intermittent connectivity issues.
  - **CONS**:
    - Potential for delayed responses in cases of frequent retries.
    - Additional logic for managing retries and detecting permanent failures.

ADR 6: Choice of Programming Language – Go (Golang)

- **Date**: 2025-03-25
- **Status**: Accepted
- **Context**:
  A decision is needed on the primary programming language for the microservices. Go (Golang) is a potential candidate due to its performance and concurrency features.
- **Decision**:
  We have decided to use **Go (Golang)** for the implementation of microservices.
- **Consequences**:
  - **PROS**:
    - **Performance**: Go is a statically typed, compiled language with high performance, making it suitable for building fast and efficient microservices.

        - **Concurrency**: Go's goroutines provide lightweight concurrency, which is ideal for handling multiple requests in parallel (important for scalable services).
        - **Ease of Deployment**: Go generates a single binary, simplifying deployment and reducing dependency management.
        - **Strong Ecosystem**: Go has a strong ecosystem for building REST APIs and working with cloud services.
    - **CONS**:
        - **Learning Curve**: While Go is simple, developers familiar with other languages (e.g., Python, JavaScript) may face a learning curve.
        - **Lack of Libraries**: Compared to languages like Python or JavaScript, Go may have fewer ready-to-use libraries, requiring more development effort for certain tasks.

ADR 7: Framework for RESTful API – Gin for Go

- **Date**: 2025-03-25
- **Status**: Accepted
- **Context**:
  A decision is needed regarding the framework for building RESTful APIs in Go. Several frameworks exist, and Gin is a popular choice for high-performance applications.
- **Decision**:
  We will use **Gin** as the web framework for building RESTful APIs in Go.
- **Consequences**:
    - **PROS**:
        - **Performance**: Gin is one of the fastest Go web frameworks, which is crucial for handling high traffic in a microservices architecture.
        - **Simplicity**: Gin has a simple and intuitive API for defining routes and handling requests.

        - **Middleware Support**: Gin provides easy-to-use middleware support for tasks like authentication, logging, and error handling.
        - **Active Community**: Gin has a large community and good documentation, which accelerates development.
    - **CONS**:
        - **Limited Features**: While Gin is lightweight and fast, it might not provide as many built-in features as heavier frameworks, requiring more custom code.
        - **Potential Overhead**: As with any framework, additional layers can introduce overhead in terms of both memory usage and execution time.

ADR 8: Database – PostgreSQL for Relational Data

- **Date**: 2025-03-25
- **Status**: Accepted
- **Context**:
  A decision is needed regarding the choice of database for storing structured data, including user profiles, courses, and enrollments.
- **Decision**:
  We will use **PostgreSQL** as the relational database for the LMS platform.
- **Consequences**:
    - **PROS**:
        - **ACID Compliance**: PostgreSQL provides strong transactional guarantees (ACID), which is essential for financial transactions (e.g., payments).
        - **Scalability**: PostgreSQL supports horizontal scaling through replication and sharding.
        - **Rich Features**: It supports advanced features like JSONB for handling unstructured data and full-text search.
    - **CONS**:

        - **Performance for Large Datasets**: While PostgreSQL is fast, its performance may degrade with very large datasets or complex queries.
        - **Operational Complexity**: Setting up and maintaining a scalable PostgreSQL setup can be complex, especially in distributed systems.

ADR 9: Caching – Redis for Session Management and Caching

- **Date**: 2025-03-25
- **Status**: Accepted
- **Context**:
  A decision is required on the caching solution for the LMS platform. Caching is crucial to improve performance and reduce database load. Redis is often used for session management and caching.
- **Decision**:
  We will use **Redis** for caching and session management.
- **Consequences**:
    - **PROS**:
        - **Speed**: Redis is an in-memory data store that offers extremely fast read and write operations, ideal for caching.
        - **Persistence**: Redis supports persistence, ensuring data is retained even after restarts.
        - **Flexibility**: Redis can be used for various caching strategies, including session data and frequently accessed course content.
    - **CONS**:
        - **Memory Constraints**: Being an in-memory data store, Redis can become costly when dealing with large amounts of data that need to be cached.
        - **Data Loss**: If Redis is configured without persistence, data loss can occur in case of failures.

ADR 10: Monitoring and Logging – OpenSearch and Prometheus

- **Date**: 2025-03-25
- **Status**: Accepted
- **Context**:
  A decision is needed for monitoring, observability, and logging tools. The system must be observable for troubleshooting, performance monitoring, and debugging.
- **Decision**:
  We will use **OpenSearch** for logging and **Prometheus** for monitoring the system's metrics.
- **Consequences**:
  - **PROS**:
    - **Centralized Logging**: OpenSearch provides a powerful search and analytics engine to manage logs from all microservices in one place.
    - **Metrics Collection**: Prometheus provides excellent support for scraping and storing metrics, with easy integration into Go applications.
    - **Alerting**: Both OpenSearch and Prometheus allow setting up alerts to notify developers about issues like service downtime or slow performance.
  - **CONS**:
    - **Complex Setup**: Setting up and configuring OpenSearch and Prometheus may require additional effort.
    - **Operational Overhead**: Both tools require proper maintenance and scaling as the platform grows.