# 4. Microservices & Scalability & Performance & Reliability

## Summary

- Decompose the system into independent, cloud-native services (e.g. auth, payments, communication, bookkeeping) and use gRPC/HTTP plus message queues for inter-service calls.
- Apply resiliency patterns (retries, circuit breakers, fallbacks), define SLAs/SLOs/SLIs, and ensure redundancy to guarantee reliability.
- Leverage platform autoscaling (KEDA), load balancing, API gateway/service discovery, and caching layers for performance and scalability.

**Pending Decisions**

- Select and configure the API gateway/service discovery solution (Azure API Management vs Application Gateway vs custom).
- Finalize the data consistency pattern (event sourcing, sagas, single-service DB) and caching strategy.

**Open Items**

- Document full microservice boundaries and integration workflows beyond the initial examples.
- Establish CI/CD and deployment patterns (e.g. blue-green/ Canary releases).
- Clarify security boundaries, token management, and RBAC between services.
- Review and align with detailed integration requirements with the other topics.

## Overview

A microservices-architected distributed system ensures scalability, reliability, and performance by breaking down the system into smaller, independent services. Each service should follow industry best practices, prioritize resiliency, and leverage cloud-native solutions.

## Key guidelines

Trust the lead developer / tech lead and overall team experience for application architecture - with secondary knowledge boost brought in through AI

- **Place a high importance on inter-service communication:**
    - consider gRPC for efficient communication between services, especially for high-performance needs. Use HTTP for external APIs where compatibility is more important
    - implement message queues for asynchronous communication and background processes (even one file's download should be queued up in most cases, especially if it takes away resources from a front-end application)
    - always keep in mind data consistency questions and avoid designing solutions on platforms which are not a good fit for such purpose (*cough cough* Kafka with equal-hierarchy topics)

- **Resiliency & reliability:**
  - design services to handle failures gracefully using retry policies, circuit breakers, and fallback mechanisms.
    - for example Dapr - Distributed Application Runtime (Microservice APIs powered by Dapr | Microsoft Learn) and inbuilt runtime specific APIs (Build resilient HTTP apps: Key development patterns - .NET | Microsoft Learn)
  - redundancy**:** ensure critical resources have a degree of redundancy to minimize downtime in unforeseen events
  - try to provide SLAs, SLOs and SLIs (promise, objective, measurement): to measure and guarantee reliability
  - consider short-circuit strategies to isolate failing services, preventing cascading failures
- **Scalability:**
  - prefer platform-provided auto scaling options
    - like KEDA (used natively also by various cloud platforms, e.g. Scaling in Azure Container Apps | Microsoft Learn)
  - load balancing, gateway and routing options: again, *prefer platform-provided options over coupling such things into core business APIs*:
    - which can also handle rate limiting (Rate Limiting pattern - Azure Architecture Center | Microsoft Learn), throttling, caching, traffic distributing, etc..
    - load balancing happens on various layers, keep in mind. See Load-balancing options - Azure Architecture Center | Microsoft Learn
    - Azure API Management - Overview and key concepts | Microsoft Learn & What is Azure Application Gateway | Microsoft Learn
  - authentication & authorization: again, split it out of the core as much as possible
    - see also 1. Authentication & Authorization
- **Data management:**
  - when using SQL databases, ensure only one application owns and manages it to maintain consistency and enable proper scalability
  - utilize various caching layers, with cloud-native approach (emphasis on distributed systems)
- **Cloud native approach:**
  - see 16. Hosting, especially understand implicit dependencies and reliance on third-party and that the cloud is a shared responsibility
  - think about service discovery, cataloging (e.g. Azure API Center - Overview - Azure API Center | Microsoft Learn)

## Example separation into microservices

- Payment gateway microservice → docs started at: LISASPORTS/new-payment-gateway
- Communication microservice → LISASPORTS/communication-service: A centralized communication service for Lisa
- Auth service → LISASPORTS/OAuthServer, see also 1. Authentication & Authorization | OAuthServer Responsibilities
- Bookkeeping system microservice
- see also further integrations requirements at 6. Communication & Notifications & Integrations