**Microservices**

Microservices (or microservice architecture) is a way of designing software systems where an application is broken into small, independent services.

Different components of microservices: <https://chatgpt.com/s/t_68b71cf0c2088191aa62318c6766b0c6>

Different components of microservices: <https://chatgpt.com/s/t_68b71cf0c2088191aa62318c6766b0c6>

Characteristics of microservices:

* Independence: each service can be developed, deployed and scaled separately
* Single responsibility: a service does one thing
* Polyglot → Different microservices can use different tech stacks (Java, .NET, Python, etc.).
* Decentralized Data → Each microservice usually has its own database (avoiding a single monolithic DB).
* Resilience → Failure in one service doesn’t bring down the entire system.

Advantages of microservices:

* Independent Deployment
* Scalability
* Technology Flexibility (Polyglot)
* Resilience & Fault Isolation
* Faster Development & CI/CD
  + Multiple teams can work on different services in parallel.
* Reusability
* Easier Maintenance & Updates
  + Smaller codebases are easier to understand and modify.
* Organizational Agility
  + Aligns with agile teams → each team owns a microservice.
  + Promotes independent team autonomy.
* Improved Time-to-Market
  + New features can be delivered quickly since they’re isolated.
  + Less risk of breaking other modules.

Disadvantages / Challenges of Microservices

* Complex Deployment & Management
  + Multiple services mean more deployment units.
* Increased Operational Overhead
  + Monitoring, logging, and debugging become harder since logs are spread across many services.
  + Requires centralized logging & monitoring
* Network Latency & Communication Issues
  + Services talk via APIs or messaging → adds network overhead compared to in-process calls in a monolith
  + Failure in communication (timeouts, retries) needs proper handling.
* Data Management Complexity
  + Each microservice often has its own database → **data consistency issues** (no single DB transaction across services).
  + Requires distributed transactions or event-driven approaches.
* Testing Challenges
  + Unit testing is simple, but **integration testing** is complex (many dependencies).
  + Need service virtualization/mocking for effective testing.
* Deployment Dependencies
  + Even though services are independent, some features span across multiple services.
  + Coordinating deployments can be tricky.
* Security Concerns
  + More endpoints → larger **attack surface**
  + Requires securing APIs, tokens, gateways, and inter-service communication.
* Requires Strong DevOps & Cloud Setup
  + Microservices are best in environments with **CI/CD, cloud infra, containers**.
  + Without DevOps maturity, it can slow teams down.
* Steeper Learning Curve
  + Developers need to understand distributed systems, API contracts, async messaging, event-driven patterns.
* Higher Infrastructure Costs
  + Each service might run in separate containers/VMs → more servers and resources.
  + Monitoring and orchestration tools also add cost.

\*\*\*\* Network latency - https://chatgpt.com/s/t\_68b44c5ea49481919e9d95f8aa83ea08  
Network latency is the time delay that occurs when data travels from one point to another over a network.

\*\*\*\* Distributed transactions or event-driven approaches: <https://chatgpt.com/s/t_68b44e17be1c81919ef9596d2d25de3d>

How data consistency issue is resolved using distributed transactions/event-driven approaches?

How microservices communicate each other?

* Synchronous communication (req-res)
  + One microservice calls another and waits for a response.
  + Common protocols - **HTTP/HTTPS REST APIs** – Most common, simple, language-agnostic.
  + Pros - Simple to implement, easy to debug
  + Cons - Tight coupling; if one service is down, dependent services may fail (cascading failure).
* Asynchronous Communication (Event-Driven / Messaging)
  + Services communicate via messages/events without waiting for immediate response.
  + Common tools –
    - Message brokers - RabbitMQ, Kafka
    - Event streaming: Kafka, Pulsar
  + Pros - Loosely coupled, resilient, scalable.
  + Cons - More complex, eventual consistency, debugging can be harder.
* Service discovery
* API gateway
  + Acts as a single-entry point for clients to interact with multiple microservices
  + Handles routing, load balancing, authentication, and sometimes response aggregation.
* Other considerations:
  + Load balancing
  + Circuit breakers
  + Retries and timeouts – to handle transient n/w failures

\*\*\*\* Circuit breaker –

* Safety mechanism
* if a service fails too many times, the circuit breaker **opens** to stop requests temporarily.
* <https://chatgpt.com/s/t_68b45491ab04819187ef488da93774bf>

**Message queues:**

* A message queue is a communication mechanism between applications or microservices that allows them to send messages to each other asynchronously
* Key points in simple terms:
  + Asynchronous communication –
    - the sender does not need the receiver to be ready.
  + Queue structure
    - Messages are stored in **order** (FIFO – First In, First Out).
    - The receiver processes messages one by one.
  + Decoupling
    - Sender and receiver don’t need to know about each other’s availability.
    - Makes systems more resilient(able to [withstand](https://www.google.com/search?sca_esv=0f2a8f6f472ba592&q=withstand&si=AMgyJEu0vuRfTngwPFrZh1qV1iGHmRMtUNrrTo4WSz6aPWn90OPRM9-c0nKfRESQA7BT0SJ7Lf8RJ4nlHbTzkd4oKrroeuprFMBX8o5PZTMUbu6LKviRJXw%3D&expnd=1&sa=X&ved=2ahUKEwim34eqmrqPAxXdnmMGHTpvE-IQyecJegQIKBAd) or recover quickly from difficult conditions) and scalable.
  + Durability
    - Messages can be stored until processed successfully.
    - Ensures no data is lost even if a service goes down temporarily.
* Common Use Cases
  + Sending emails after a user signs up.
  + Processing orders in an e-commerce app.
  + Logging events from multiple services
  + Handling notifications in a mobile app.
* \*\*\*\*\* A message queue is like a waiting line for messages, allowing microservices to communicate reliably without blocking each other.

**API Gateway vs Load balancing:** https://chatgpt.com/s/t\_68b6fb5af508819196e2cd820416d037

* API Gateway
  + routes and manages requests across different services, with extra features like security, rate limiting, and monitoring.
  + decides which service (Payments, Loans, Fraud) should handle the request.
* Load Balancer
  + distributes requests among instances of the same service.
  + distributes that request across multiple healthy instances of the chosen service.

Rate limiting:

Auto scaling: