A microservice is a software development approach where an application is built as a collection of small, loosely coupled services that are independently deployable and scalable. ***Each microservice focuses on a specific business capability*** and ***communicates with other microservices through lightweight protocols like HTTP or messaging queues.*** Here are some common design patterns associated with microservices:

1. **Single Responsibility Principle (SRP):** Each microservice is designed to have a single responsibility or business capability. This ensures that each service remains focused, maintainable, and independently deployable.
2. **Decentralized Data Management:** Microservices often have their dedicated data storage or database, ensuring that each service has its own data model and can manage its data independently. This enables service autonomy and reduces inter-service dependencies.
3. **API Gateway:** An API gateway acts as a single entry point for client applications to interact with multiple microservices. It handles requests from clients, performs authentication and authorization, and routes requests to the appropriate microservices. This pattern provides a unified interface to the clients and helps to decouple the client applications from the underlying microservices.
4. **Service Discovery:** Microservices need a way to discover and communicate with each other dynamically. Service discovery patterns involve a service registry or a centralized mechanism where microservices can register themselves and obtain information about other available services. This enables dynamic scaling, load balancing, and fault tolerance.
5. **Event-Driven Architecture (EDA):** In an event-driven microservices architecture, services communicate with each other by producing and consuming events. Events represent significant changes or actions within the system. This pattern promotes loose coupling, scalability, and flexibility as services can react to events asynchronously.
6. **Circuit Breaker:** The circuit breaker pattern is used to handle failures and prevent cascading failures in a distributed system. It monitors the calls to remote services and can open the circuit when the service is unavailable or experiencing issues. It helps to isolate and contain failures, improving overall system resilience.
7. **Containerization:** Microservices are often deployed using containerization technologies like Docker. Containerization provides an isolated and lightweight runtime environment for each microservice, allowing easy deployment, scaling, and management of services across different environments.
8. **Continuous Integration and Deployment (CI/CD):** Microservices architectures often embrace CI/CD practices. Each microservice is developed, tested, and deployed independently, enabling rapid iteration and deployment of new features or bug fixes.

These are just a few examples of design patterns associated with microservices. The actual design patterns used can vary depending on the specific requirements and technologies used in a given implementation.