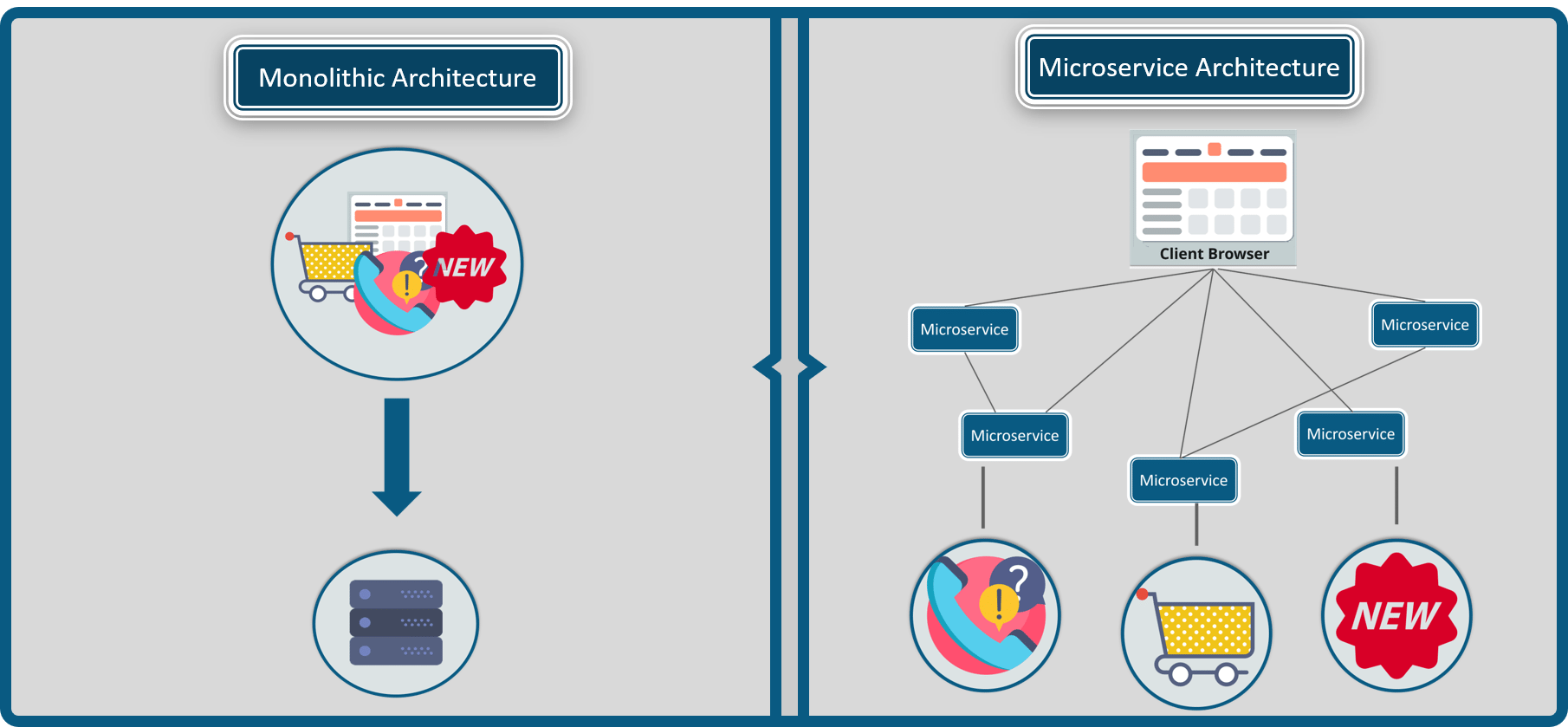
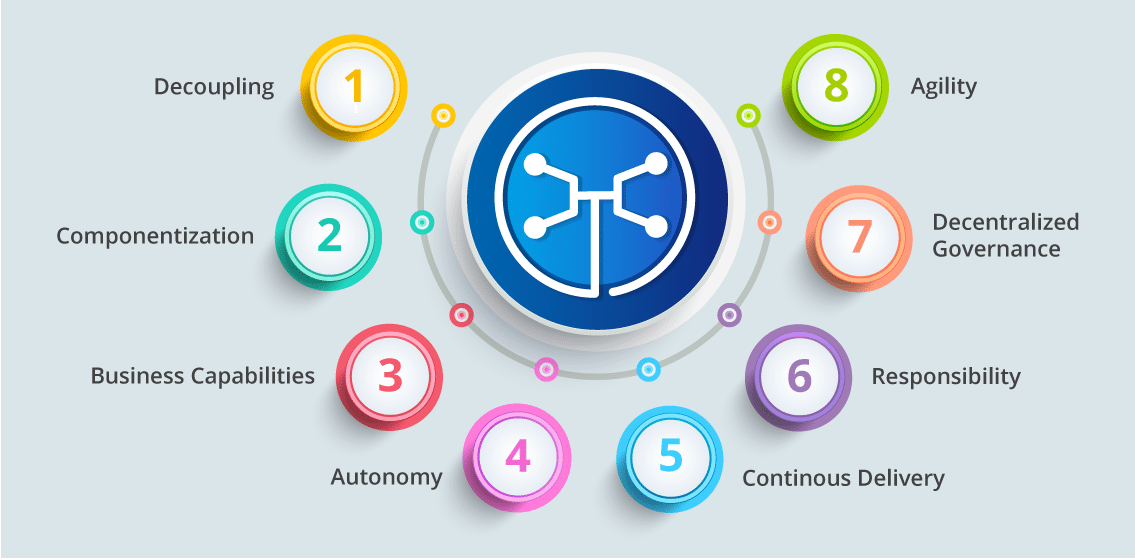
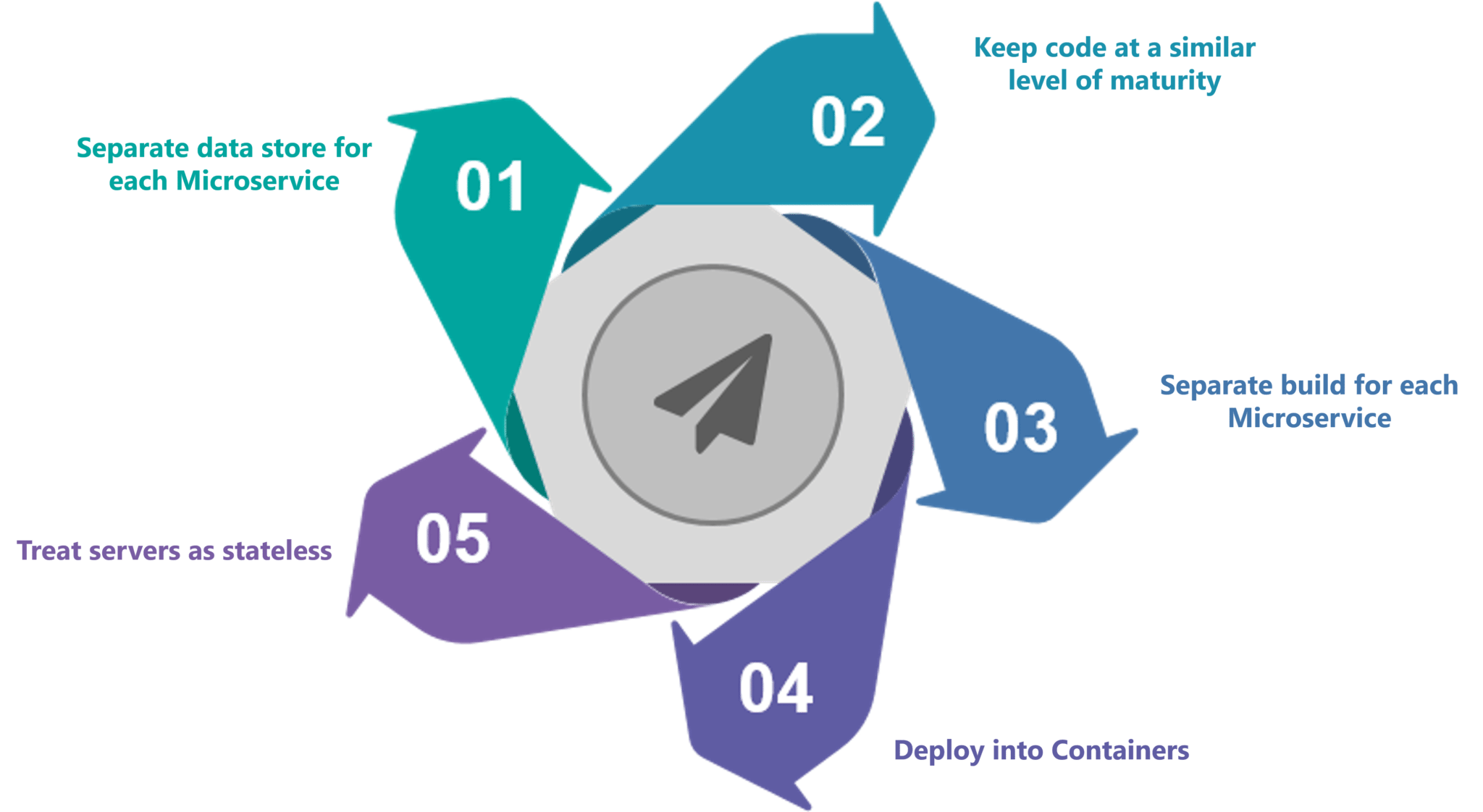
**What are Micro services**

1. An architectural style in which large complex software applications are composed of one or more services
2. Each of these micro services focuses on completing one task only and does that task really well
3. Communicate with each other using language-neutral APIs e.g. REST
4. Have a bounded context i.e. one micro service does not need to know anything about another micro service
5. Individual micro services can be developed using different technologies that best suit to their need
6. Can be deployed and scaled independent of each other
7. Better isolation for error detection and damage control



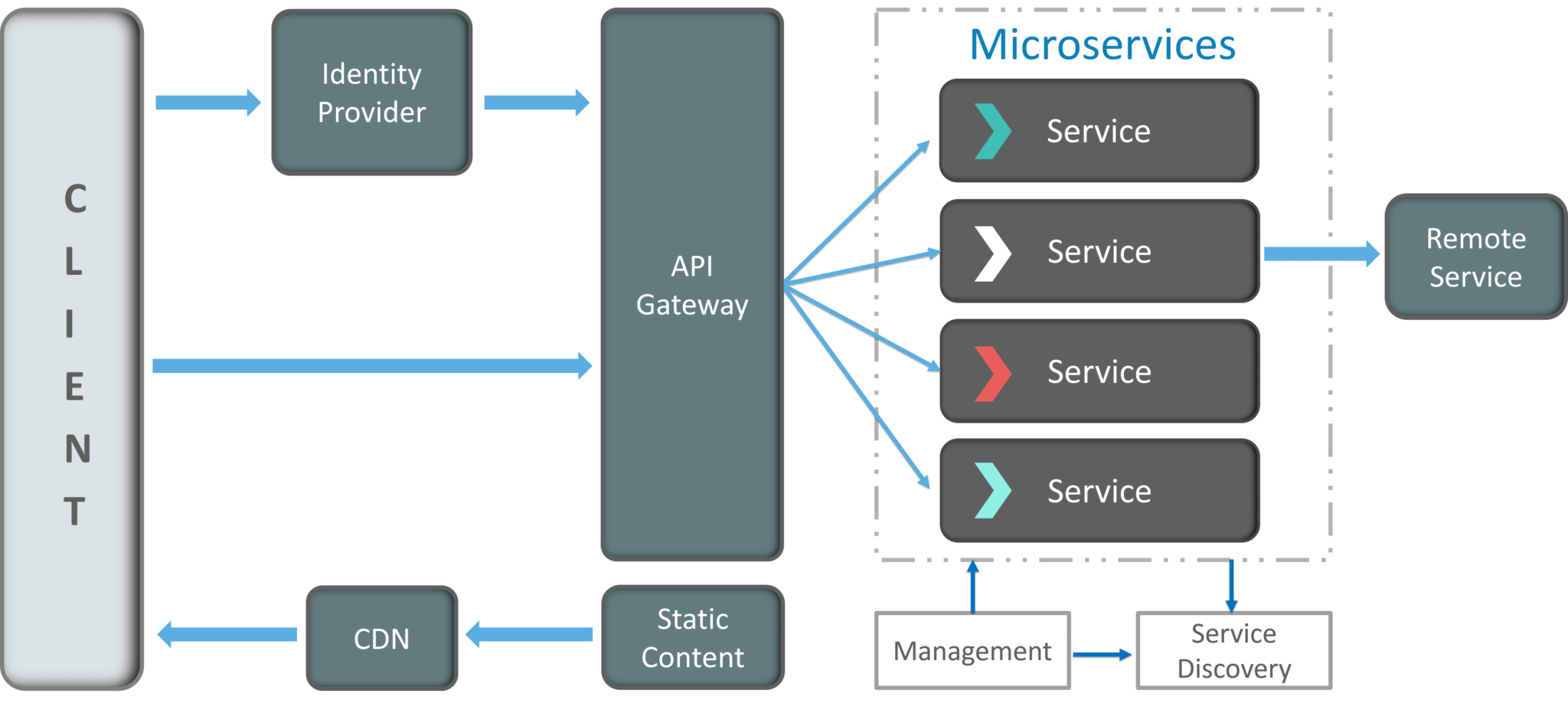


* **Decoupling** Services within a system are largely decoupled. So the application as a whole can be easily built, altered, and scaled
* **Componentization** Microservices are treated as independent components that can be easily replaced and upgraded
* **Business Capabilities** Microservices are very simple and focus on a single capability
* **Autonomy** Developers and teams can work independently of each other, thus increasing speed
* **Continuous Delivery** Allows frequent releases of software, thru systematic automation of software creation, testing, and approval
* **Responsibility** Microservices do not focus on applications as projects, rather as products for which they are responsible
* **Decentralized Governance** The focus is on using the right tool for the right job. That means there is no standardized pattern or any technology pattern. Developers have the freedom to choose the best useful tools to solve their problems
* **Agility** Microservices support agile development. Any new feature can be quickly developed and discarded again



**Microservice best practices**

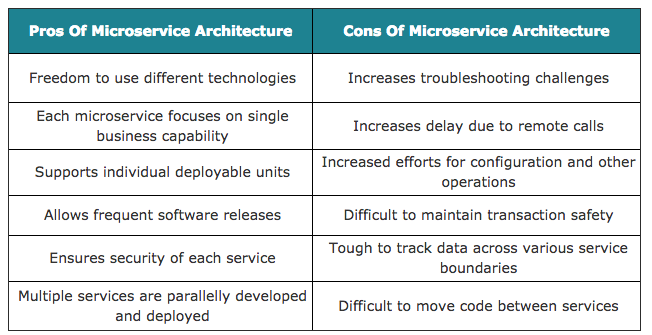
**Microservices Architecture**



**A typical Microservice architecture consists of the following components:**

|  |  |
| --- | --- |
| **Clients** | The architecture starts with different types of clients, from different devices trying to perform various management capabilities such as search, build, configure etc. |
| **Identity providers** | These requests from the clients are then passed on the identity providers who authenticate the requests of clients and communicate the requests to API Gateway. The requests are then communicated to the internal services via well-defined API Gateway. |
| **API Gateway** | Since clients don’t call the services directly, API Gateway acts as an entry point for the clients to forward requests to appropriate microservices.  The advantages of using an API gateway include:   * All the services can be updated without the clients knowing. * Services can also use messaging protocols that are not web-friendly. * The API Gateway can perform cross-cutting functions such as providing security, load balancing etc.   After receiving the requests of clients, the internal architecture consists of microservices which communicate with each other through messages to handle client requests. |
| **Messaging formats** | There are two types of messages through which they communicate:   * Synchronous Messages: In the situation where clients wait for the responses from a service, microservices usually tend to use REST (Representational State Transfer) as it relies on a stateless, client-server, and the HTTP protocol. This protocol is used as it is a distributed environment each and every functionality is represented with a resource to carry out operations * Asynchronous Messages: In the situation where clients do not wait for the responses from a service, microservices usually tend to use protocols such as AMQP, STOMP, MQTT.These protocols are used in this type of communication since the nature of messages is defined and these messages have to be interoperable between implementations.   The next question that may come to your mind is how do the applications using microservices handle their data? |
| **Data handling** | Well, each micro service owns a private database to capture their data and implement the respective business functionality. Also, the databases of microservices are updated through their service API only.  The services provided by microservices are carried forward to any remote service which supports inter-process communication for different technology stacks. |
| **Static content** | After the microservices communicate within themselves, they deploy the static content to a cloud-based storage service that can deliver them directly to the clients via **Content Delivery Networks (CDNs)**. |
| **Management** | This component is responsible for balancing the services on nodes and identifying failures. |
| **Service discovery** | Acts as a guide to microservices to find the route of communication between them as it maintains a list of services on which nodes are located. |

**Pros and Cons**



**API Gateway**

**Problems that comes into picture if a client directly calls a micro service:**

* Number of calls made to different micro services for a page to render with all information can be a limiting factor
* So many requests over a public network is not at all prudent. This would also make the client much more complex
* For clients directly calling micro services, protocols might not be web friendly. Applications should use only HTTP/web sockets outside the firewall
* This approach is also problematic for refactoring micro services

**Need for an API gateway:**

1. Services may be updated and the address of the service could change
2. Dealing with different kinds of clients that have different kinds of requirements
3. Systems might have centralized authentication and security

**An API gateway provides a single, unified entry-point across one or more internal APIs:**

1. Enables support for mixing communication protocols
2. Decreases micro services complexity by providing authorization using API tokens
3. Provides request routing. All requests from clients at first must go thru API gateway. Then those are routed to appropriate micro services

**Microservice design patterns**

Principles for designing microservice architecture

1. High cohesion along with loose coupling
2. Seamless API integration
3. A unique source of identification for every service
4. Real time traffic management
5. Minimizing data tables to optimize load
6. Performing constant monitoring over external and internal APIs
7. Isolated data storage for each microservices
8. Decentralization
9. Scalability
10. Continuous delivery through DevOps integration

**Collaboration patterns:**

1. Aggregator MDP:

With multiple services involved, fetching the output and combining it for the end user is necessary. To aggregate this data, we make use of the Aggregator pattern. The solution can be forwarded to the end user through two major components

* + 1. Composite microservice
    2. API gateways

1. Branch MDP:

You can simultaneously process the request and response from two independent or two mutually exclusive chains of microservices

1. Backend for frontend/API gateway:

API gateway acts as the single-entry point for all interactions that take place within the architecture. It also helps to establish security by client authorization and exposing relevant APIs with respect to the client. It also aggregates results from multiple services and send the output to user. It can also handle multiple protocol requests and convert

**Performance monitoring patterns:**

1. Log aggregation MDP:

Every service generates an entry in the logs regarding its execution. The master logging service is responsible for aggregating the logs from all the microservice instances. This centralized log should be searchable making it easier to monitor

1. Synthetic monitoring/ Semantic monitoring

Helps to regularly map the results in comparison to the production environment

1. API health check

Every service running must have a specific health check API endpoint. A service registry periodically appeals to the health check API endpoint to perform a health scan

**Decomposing monolith as per business capabilities:**

1. Unique microservice for each BC

This needs classification into various business domains that are responsible to generate value in their own capabilities

1. Microservices around similar business capabilities

Domain Driven Design (DDD) comes into play here to decompose common classes or God classes across business capabilities. We use subdomains here. These subdomains have defined scope of functionality known as bounded context. This is the parameter to overcome the issue of common classes

1. Strangler Vine MDP

Service is broken down into different domains as possible. Once the service have been reformed it strangles the existing version of the application. (came from an analogy in Australia)

**Optimizing Database storage:**

1. Individual Database per service

1. Shared Database per service
2. Event Sourcing design pattern
3. Command Query Responsibility Segregation

**Seamless deployment:**

1. Service discovery

Involves the use of a service registry. A service instance registers itself on this registry while coming up and deregister while shutting down. Health check on this registry ensures the availability of only working instances

1. Blue Green deployment

Two identical environments run parallelly known as blue and green. At a time only one is live and processing all the production traffic.