1. Stateless vs Stateful Servers

Stateless Servers:
A stateless server does **not retain any client-specific information** between requests. Each request is independent, and the server relies on the client or external storage to provide context. Stateless servers are simpler, easier to scale, and highly fault-tolerant because any server can handle any request.

Examples:

- **Web servers serving static pages** (e.g., Nginx, Apache)
- **REST APIs** where each request contains authentication tokens and request data

Stateful Servers:
A stateful server **maintains information about the client** across multiple requests, storing session data, user preferences, or transaction states. This allows the server to provide personalized or continuous services but introduces complexity in scaling and failure recovery.

Examples:

- **Database servers** maintaining transactions (e.g., MySQL, PostgreSQL)
- **Online multiplayer game servers** tracking player state

Key Differences:

Feature	Stateless	Stateful
Client context	None	Maintained across requests
Scalability	Easy	Harder; needs session sharing
Fault tolerance	High	Lower; losing a server may lose session

2. Advantages of Containers over VMs in Distributed Deployment

Containers and **Virtual Machines (VMs)** both isolate applications, but containers provide significant advantages for distributed systems:

1. **Lightweight:**
   Containers share the host OS kernel, so they start faster and consume less memory compared to VMs, which include a full guest OS.

2. **Portability:**
   Containers package applications with their dependencies, ensuring consistent behavior across development, testing, and production environments.

3. **Scalability:**
   Due to their lightweight nature, containers can be deployed and replicated quickly across clusters, making horizontal scaling easier.

4. **Efficient Resource Utilization:**
   Multiple containers can run on a single host without the overhead of multiple OS instances, maximizing server utilization.

5. **Microservices Friendly:**
   Containers align perfectly with microservices architecture, allowing independent deployment, updates, and rollback of individual components.

Example: Kubernetes orchestrates containers across nodes, enabling distributed deployment, auto-scaling, and self-healing in large systems, which would be slower and more resource-heavy with VMs.