

# Horizontal Scaling

The process of adding more servers to your infrastructure to handle increased demand. Unlike vertical scaling, where you upgrade the capacity of single machine, horizontal scaling distributes the workload across multiple servers.

## Key Features of Horizontal Scaling:

1. Improved Fault Tolerance
2. Elasticity
3. Load Balancing
4. Distributed Data management(Often requires sharding, replication, or partitioning of data to manage it across multiple nodes)

## How Horizontal Scaling Works:

- When traffic or load increases, the system:
1. Adds More Nodes: Additional servers are added to the system.
2. Redistributes Load: A load balancer ensures that incoming request are evenly distributed across all available nodes
3. Ensures Data consistency: Using database with distributed capabilities, data consistency is managed across nodes using techniques like replication or sharding.

## Real-world Example: e-Commerce Website

- Imagine an e-commerce websites like amazon handling millions of concurrent users.
1. Scenario: During black Friday sales, traffic spike significantly.
2. Solution: The website adds mutiple application servers to handle the increased traffic.
- Load Balancer: Incoming request are distributed among all the application servers.
- Database scaling:
- For reading data, read replicas of the databse are created and traffi is distributed among them.
- For writing data, the data is partitioned or sharded across multiple database nodes to reduce write bottlenecks.
3. Caching: Popular items or user session data are cached in distributed system like Redis or Memcached, reducing the load on application and database servers.

## System Design Consideration

- When implementing horizontal scaling:
1. Stateless Design: Application servers should be stateless, meaning no session-specific data should be stored locally. Instead, use external session stores like redis.
2. Consistency vs Availability: Decide based on the system's need. e-Commerce system often prioritize availability.
3. Load Balancer: Choose a robust load balancer.
4. Monitoring: Tools like Prometheus, Grafana, AWS cloudwatch monitors systems health and scale it dynamically.

## System Design Interview Example Question:

- Design a URL shortening server like bit.ly.
- Here's how horizontal scaling fits in:
1. Application Layer: Use multiple stateless application servers to handle requests.
2. Database: Implement sharding for storing shortened URLs based on hash prefixes.
3. Cache: Use a distributed cache to store frequently accessed URLs.
4. Load Balancer: Distribute incoming requests across the application servers.

## Challenges with Horizontal scaling

Horizontal scaling offers significant benefits, such as improved fault tolerance, elasticity, and cost efficiency, but it also introduces challenges. Here are some of the key challenges you may face with horizontal scaling:

1. Data Consistency

Challenge: Maintaining data consistency across mutiple server or databases can be difficult, especially in distributed system

Example: In a multi-node database, ensuring all replicas have the latest updates requires complex synchroization protocols like paxos, or Raft.

Solution: Use consistency model like eventual consistency, or strong consistency when required, though it may come with performance trade-offs.
2. Complex Load Balancing

Challenge: Effectively distributing traffic among mutiple nodes requires a robust load balancer, Uneven traffic distribution can lead to overloading certain nodes while other remain underutilized.

Example: A sudden surge in traffic to a specific geographic region might cause localized server strain.

Solution: Use smart load balancers with features like sticky sessions, geolocation routing and real-time health checks.
3. Session Management

Challenge: Maintaining user sessions across multiple stateless servers is tricky.

Example: A user logs in, and their subsequent requests might go to a different server that doesn't recognize their session.

Solution: Implement external session stores like Redis or DynamoDB to ensure session continuity.
4. Data Partitioning and Sharding

Challenge: Distributing data across multiple nodes (e.g., sharding) requires careful design to avoid uneven partitions (hot spots) or data access delays.

Example: A poorly designed shard key can lead to one database node handling the majority of the traffic.

Solution: Use consistent hashing or shard based on access patterns to ensure even data distribution.
5. Increased Latency

Challenge: Adding more nodes might introduce latency due to the additional communication overhead between servers.

Example: A database query requiring cross-node communication (e.g., join operations) can experience delays.

Solution: Optimize queries, denormalize data, or use distributed database systems designed for low-latency operations like Cassandra or CockroachDB.
6. Fault Tolerance and Recovery

Challenge: Although horizontal scaling improves fault tolerance, handling node failures in real-time without impacting the system requires sophisticated failover mechanisms.

Example: If a node goes down during peak traffic, redistributing the load to other nodes can cause temporary overloads.

Solution: Implement health checks, automatic failover, and capacity planning.
7. Cost Management

Challenge: While adding nodes can handle increased traffic, it also increases operational costs (e.g., compute, network, and storage expenses).

Example: Over-provisioning servers during traffic spikes can lead to underutilized resources and wasted costs.

Solution: Use auto-scaling and monitoring tools to scale in and out dynamically based on actual demand.
8. Consistency vs Availability (CAP Theorem)

Challenge: Distributed systems often face trade-offs between consistency and availability.

Example: During a network partition, a horizontally scaled system must choose between serving stale data (availability) or returning errors (consistency).

Solution: Make trade-offs based on system requirements (e.g., prioritize availability in e-commerce, prioritize consistency in financial systems).
9. Deployment and Configuration Management

Challenge: Ensuring consistent deployment and configuration across all nodes can be complex.

Example: A configuration mismatch between nodes can lead to inconsistent application behavior.

Solution: Use tools like Kubernetes, Terraform, or Ansible for orchestration and configuration management.
10. Monitoring and Debugging

Challenge: Monitoring a distributed system with many nodes generates a large volume of logs and metrics, making it harder to pinpoint issues.

Example: Debugging a failed transaction that spans multiple servers and databases can be challenging.

Solution: Use centralized logging (e.g., ELK Stack, Splunk) and monitoring tools (e.g., Prometheus, Datadog) for visibility and debugging.
11. Network Bottlenecks

Challenge: As nodes increase, network communication between them becomes a potential bottleneck.

Example: A distributed database system might face delays due to high inter-node communication traffic.

Solution: Optimize network usage by reducing inter-node communication or using higher-bandwidth networking solutions.
12. Distributed Systems Complexity

Challenge: Horizontally scaled systems inherently become more complex to design, implement, and maintain.

Example: Synchronizing tasks or operations across multiple nodes requires distributed algorithms, which are more complex than centralized ones.

Solution: Invest in training, use well-documented frameworks, and modularize system components to reduce overall complexity.

## System Design Question: Design as real-time video processing sytem for live sports streaming

- Problem Requirements:
- Key Features:
- Accept video feeds from cameras in sports stadiums.
- Process these feeds to add real-time overlays (e.g., score, stats).
- Stream the processed video to thousands of users watching live.
- Non-Functional Requirements:
- Low Latency: Real-time processing should not introduce noticeable delays.
- High Reliability: Downtime during live matches is unacceptable.
- Scalability: The system should handle increased processing demands during high-profile events.
- Constraints:
- Real-time video processing is computationally intensive.
- Budget constraints limit the ability to immediately distribute tasks across multiple machines.

## Horizontal Scaling for Real-time video processing

- Core Idea:
- Instead of relying on single high-performance server, distribute the workload acrosss mutiple server or nodes. Each node handle a portion of video processing pipeline.
- Redesigned System Architecture
1. Video input module:

Distribute incoming video feeds across multiple nodes

Use a load balancer to assign specific video feeds to individual servers.
2. Processing Module:

Each server processes one or more video streams independently.

Implement stateless processing pipeline, so any server can handle any video feed.

Use GPUs or hardware accelerators on each node to maximize processing power.
3. Streaming Module:

After processing, the nodes send their output to central server for delivery to end users.

## Key Components with Horizontal Scaling

1. Load Balancer

Distributs video feeds among processing nodes to ensure even workload distribution.

Adjusts dynamically if a node become overloaded or fails.
2. Distributed Processing:

Split processing tasks(decoding, overlay rendering, encoding) acrosss mutiple servers.

Use a distributed task manager like Apache Kafka, RabbitMQ, or custom routing logic.
3. Database for metadata:

Use a distributed database (e.g., Cassandra or DymamodB) to store metadata about video feeds and processing state.

Ensure this metadata is accessible to all nodes for fault tolerance.
4. Fault Tolerance:

Add redundancy: if a node fails, its tasks are reassigned to other nodes.

Use health checks and auto-scaling groups (e.g., AWS Auto Scaling, Kubernetes) to monitor and manage nodes.

## How Horizontal Scaling Handles Scenarios

- Normal Game Scenario:
- 5 video feeds are distributed across 5 nodes, with each node handling one feed.
- The system remains underutilized, ready to scale up if needed.
- High Profile Event Scenario
- A surge to 10 feeds triggers automatic horizontal scaling. The system spins up 10 additional nodes, ensuring each feed is processed on a separate node.
- Load balancing ensures no single node is overloaded.

## How Horizontal Scaling Handles Scenarios

1. Scalability:

Add more nodes as traffic or processing demands increase.

No upper limit compared to hardware constraints of vertical scaling
2. Fault Tolerance

If one node fails, others can take over, minimizing disruption.
3. Cost Efficiency

Use smaller, cost-effective servers for distributed processing, scaling only when needed.
4. Geographical Distribution:

Deploy processing nodes closer to video sources or end-users to reduce latency.

## Challenges of Horizontal Scaling

1. Data Distribution:

Distribution of video feeds across nodes introduces complexity, You'll need to ensure smooth task reallocation if a node fails
2. Latency

Splitting tasks across nodes can introduce inter-node communication delays.

Solution: Optimize by reducing communication overhead and using fast networks.
3. Consistency

Ensuring all nodes operate on consistent metadata or configuration requires careful design.
4. Monitoring and Debugging:

Monitoring a distributed system requires sophisticated tools

## Trade-offs: Vertical vs Horizontal Scaling

Aspect	Vertical Scaling (Hardware Upgrades)	Horizontal Scaling
Scalability	Limited by physical capacity	Unlimited by adding more nodes
Fault Tolerance	Low(Singl point of failure)	High(Distributed across nodes)
Cost	Expensive upfront hardware costs	Scales cost-efficiently with demand
Latency	Low(no inter-node communication)	May increase due to distributed tasks
Complexity	Simple to implement	More complex to design and manage

## Why use Horizontal Scaling in This system?

1. High-profile Events: The unpredictable nature of traffic during popular games demands elastic scaling, which is easier to achieve with horizontal scaling.
2. Global Expansion: As the system grows to handle international streams, horizontal scaling enables geo-distributed processing for reduced latency.
3. Reliability: Fault tolerance becomes crucial for live events; horizontal scaling ensures failover capabilities without affecting the user experience.

## Further Tradeoff between vertical and Horizontal Scaling

1. Scalability

Vertical Scaling:

Trade-off: Limited by physical Hardware capacity of a single machine. Once you hit the limit, scaling further is impossible

Scenario: Works well for applications with predictable, steady growth.

Horizontal Scaling:

Trade-off: Virtually unlimited; you can add more nodes as demand increases. However software and architecture must support distributed scaling.

Scenario: Ideal for dynamic or unpredictable growth patterns, such as handling traffic spikes in real time system.
2. Fault Tolerance

Vertical Scaling:

Trade-off: A single point of failure. If the upgrade server goes down, the entire system fails.

Scenario: Vertical scaling requires expensive failover system.

Horizontal Scaling:

Trade-off: High fault tolerance. If one node fails other can take over. Requires redundancy and failover mechanism

Scenario: Horizontal scaling provides built in redundancy, making it better for mission critical systems.
3. Cost Efficiency

Vertical Scaling:

Trade-off: Upfront hardware cost are high. For example, upgrading to high performance GPUs or adding terabytes of RAM can be expensive.

Scenario: Cost-effective for small-scale system or early stage applications.

Horizontal Scaling:

Trade-off: Incremental costs. You pay only for the additional nodes as needed.

Scenaiio: Cost-effective for system with variable demand, as you can scale in and out dynamically.
4. Latency

Vertical Scaling:

Trade-off: Low latency within a single powerful machine, as no inter-node communication is required.

Scenario: Vertical scaling is ideal for latency-sensive systems like in-memory databases or financial trading.

Horizontal Scaling:

Trade-off: Potential latency due to inter-node communication and data distribution overhead.

Scenario: Horizontal scaling works for systems that tolerate slight delays, such as content delivery or batch processing.
5. Complexity

Vertical Scaling:

Trade-off: Simpler to design, manage, and maintain. Focuses on optimizing a single machine

Scenario: Good for small teams or when simplicity is a priority.

Horizontal Scaling:

Trade-off: High complexity. Requires orchestration tools and robust distributed algorithm for consistency and fault tolerance.

Scenario: Necessary for large-scale distributed systems requiring global availability.
6. Performance Bottlenecks

Vertical Scaling:

Trade-off: Single machine bottleneck can limit performance. Upgrades may not address software inefficiencies.

Scenario: Suitable for applications with limited concurrency and processing demands

Horizontal Scaling:

Trade-off: Bottlenecks can shift to network communication or load balancers, requiring careful tuning.

Scenario: Better for handling high-concurrency workloads, such as large-scale user interactions or data processing.
7. Time to Scale:

Vertical Scaling:

Trade-off: Scaling up requires downtime to upgrade hardware, which may interrupt the service

Scenario: Works when planned downtime is acceptable.

Horizontal Scaling:

Trade-off: Scaling out can be seamless using automation, but requires pre-planned architecture.

Scenario: Perferred for system needing uninterrupted service. Cloud-based auto-scaling helps here.
8. Data Consistency

Vertical Scaling:

Trade-off: Easier to maintain consistency because all operations happen on a single machine.

Scenario: Best for systems requiring strong consistency, such as relational databases.

Horizontal Scaling:

Trade-off: Requires distributed data managemtn techniques which complicate consistency guarantees.

Scenario: Suitable for systems where eventual consistency is acceptable, such as no-sql database or cache layer.
9. Energy Efficiency:

Vertical Scaling:

Trade-off: High-end servers consume significant energy and cooling resources

Scenario: Efficient for small-scale setups

Horizontal Scaling:

Trade-off: A large number of smaller nodes may consume more energy collectively.

Scenario: Better for cloud environments using optimized energy efficient instances.
10. Deployment Model

Vertical Scaling:

Trade-off: Limited to on-premises or fixed infrastructures, Cloud providers have limited vertical scalability options.

Scenario: On-premises deployments benefit from vertical scaling

Horizontal Scaling:

Trade-off: Fully supported in cloud environments, allowing dynamic provisioning.

Scenario: Cloud-native applications are designed for horizontal scaling.