

---

---

# Mastering System Design

## Section 6: Scalability in System Design

### Scalability

---

---

# Scalability - Section Agenda

1. Introduction to Scalability
2. Scaling Strategies: Horizontal, Vertical & Diagonal
3. Understanding Load Balancers: Types, Algorithms & Cloud Solutions
4. Autoscaling & Best Practices in Cloud Environments
5. Summary and Final Thoughts

---

---

# Introduction to Scalability

Scalability In System Design

---

---

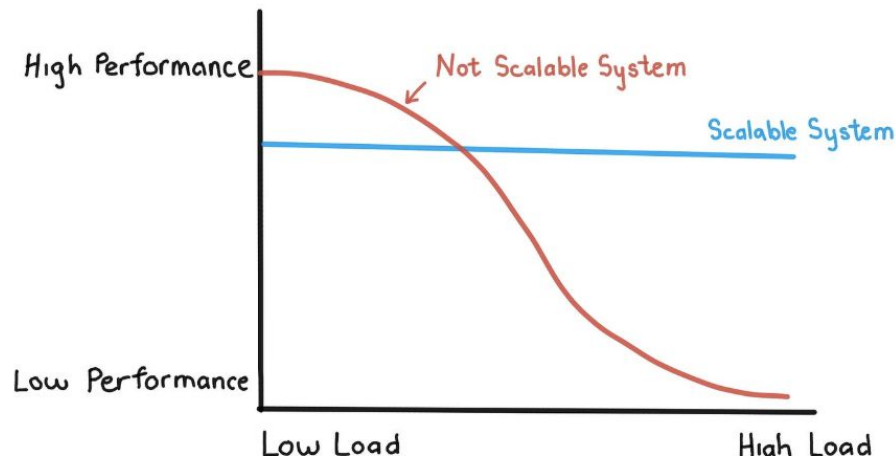
# What is Scalability?

- Scalability is the ability of a system to handle an increasing amount of work, or its potential to accommodate growth.
- It ensures performance, reliability, and availability under growing load.



# Why Do Systems Need to Scale?

- User base growth (e.g., launching in new regions)
- Increasing data volume (e.g., IoT, analytics)
- Peak events (e.g., Black Friday, ticket sales)
- Avoid service degradation or downtime
- Meet performance SLAs

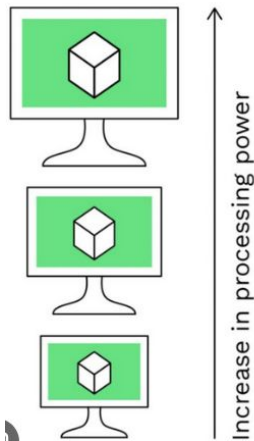


# Types of Scalability (Intro)

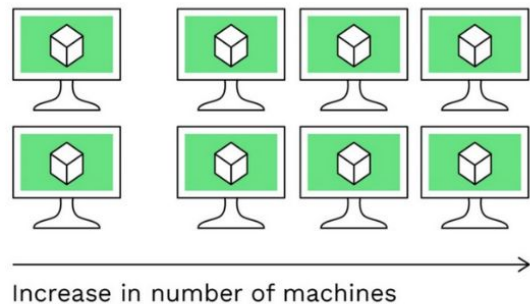
- Vertical Scaling: Add more CPU/RAM to one server
- Horizontal Scaling: Add more servers to distribute the load

🚨 We'll go deep into these types in the next lecture

**Vertical scaling**



**Horizontal scaling**



# Common Challenges in Scaling

- Latency
  - Delay between request and response
  - Causes: network hops, slow DB queries, synchronous calls
  - Amplified in microservices/distributed systems
- Bottlenecks
  - One slow component = system-wide slowdown
  - Examples: DB locks, memory limits, single-threaded processing
  - Hard to predict as load grows
- Downtime
  - More nodes = more failure points
  - Updates, redeployments, scaling events can cause outages
  - High availability becomes harder
- Cost
  - Infrastructure isn't free — CPU, RAM, bandwidth, etc.
  - Autoscaling without limits = budget nightmare
  - Over-provisioning = wasted spend

# Interview Questions - Scaling

- What does scalability mean in the context of system design?
- Can you explain a real-world example where scalability was critical to success or failure?
- What are the main challenges systems face as they scale?
- How would you identify a bottleneck in a scalable architecture?
- Why does latency increase with scale, and how can you mitigate it?
- How do you balance scalability with cost in cloud-based systems?



# Summary & What's Next

- Scalability helps systems grow without breaking
- Two main types: vertical and horizontal
- But it introduces latency, cost, and other risks
- What's Next:
  - Scaling Strategies — how to actually implement horizontal, vertical, and diagonal scaling with trade-offs and real-world choices

---

---

# Scaling Strategies: Horizontal, Vertical & Diagonal

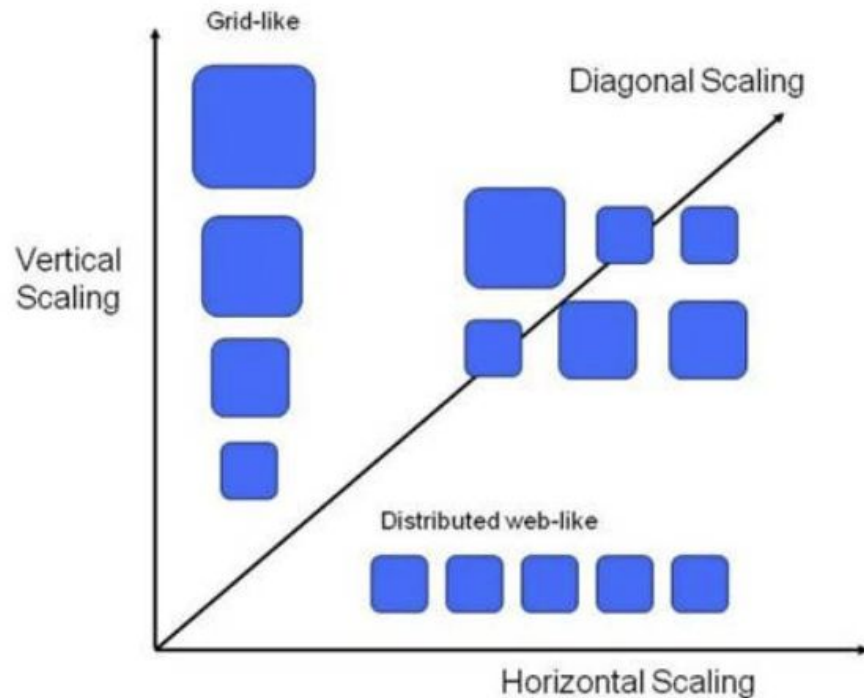
Scalability In System Design

---

---

# Quick Recap – Types of Scalability

- Vertical: Upgrade one machine
- Horizontal: Add more machines
- Diagonal: Start vertical, then go horizontal



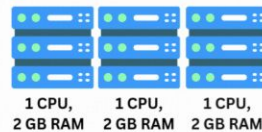
# Types of Scalability - Deep Dive

- Vertical: Upgrade one machine
  - Upgrade server's CPU, RAM, Disk
  - Easy to implement (less complexity)
  - Limits: Physical cap, risk of single point of failure
- Horizontal: Add more machines
  - Add more nodes to distribute traffic/load
  - Requires load balancer, stateless design
  - Complex setup (coordination, replication)
- Diagonal: Start vertical, then go horizontal
  - Hybrid approach: start vertical, add horizontal as needed
  - Common in cloud-native apps
  - Cost-effective + long-term ready

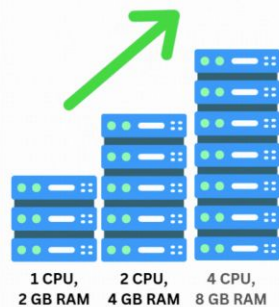
## VERTICAL SCALING



## HORIZONTAL SCALING



## DIAGONAL SCALING



# Trade-Offs: Cost vs Complexity vs Performance

Strategy	Cost	Complexity	Performance
Vertical	Low-mid	Low	Medium
Horizontal	High	High	High
Diagonal	Medium	Medium	High

# Real-World Examples and When to Choose What?

- Examples:
  - Twitter: Moved from monolith → horizontal scaling (microservices)
  - Small startups: Vertical scaling for MVP
  - AWS Lambda apps: Start diagonal with autoscaling
- When to choose:
  - Startups: vertical (cheaper, simpler)
  - Scaling apps: horizontal (resilience + capacity)
  - Cloud-native: diagonal (flexibility + cost balance)

# Interview Questions - Scaling Strategies

- What is the difference between horizontal and vertical scaling?
- What is diagonal scaling and when is it a good idea?
- What are the trade-offs between horizontal and vertical scaling in terms of performance and complexity?
- Can you describe a scenario where horizontal scaling wouldn't help?
- When would you choose vertical scaling over horizontal?
- What challenges arise in horizontal scaling and how would you solve them?

# Summary & What's Next

- Scaling Strategies:
  - Vertical = simpler, but hits limits
  - Horizontal = scalable, but needs planning
  - Diagonal = hybrid approach for cloud success
- What's next:
  - Understanding Load Balancers: Types, Algorithms & Cloud Solutions



---

---

# Understanding Load Balancers

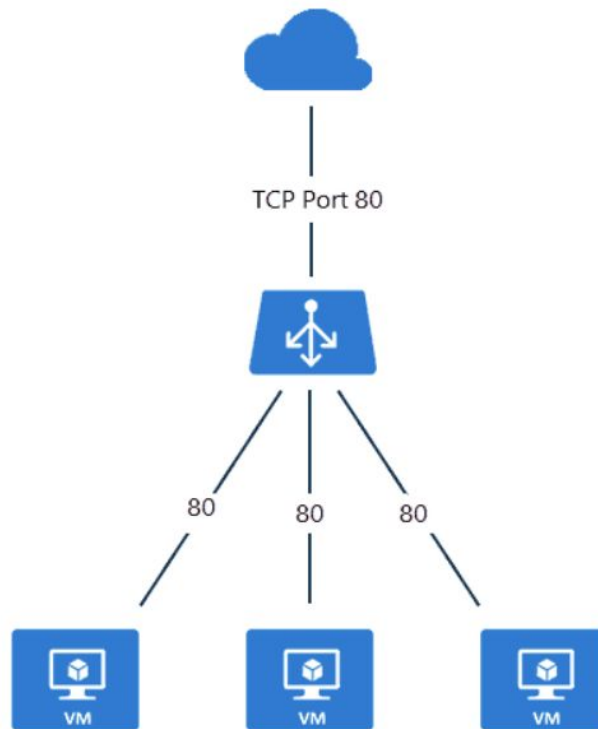
Scalability In System Design

---

---

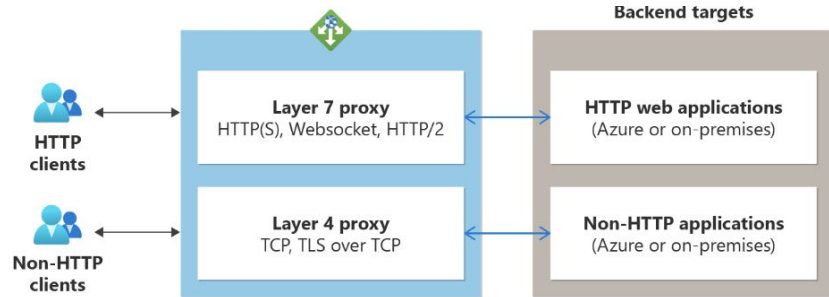
# Why Load Balancing is Needed

- High Availability: Ensures system uptime even under high traffic.
- Traffic Distribution: Spreads requests evenly across servers.
- Prevents Overload: Avoids overburdening a single server.
- Improves Performance: Reduces latency and enhances response times.
- Handles Failures Gracefully: Redirects traffic in case of server failure.
- Supports Scalability: Helps scale systems efficiently.
- Example: A high-traffic e-commerce site uses load balancing to handle peak-hour requests.



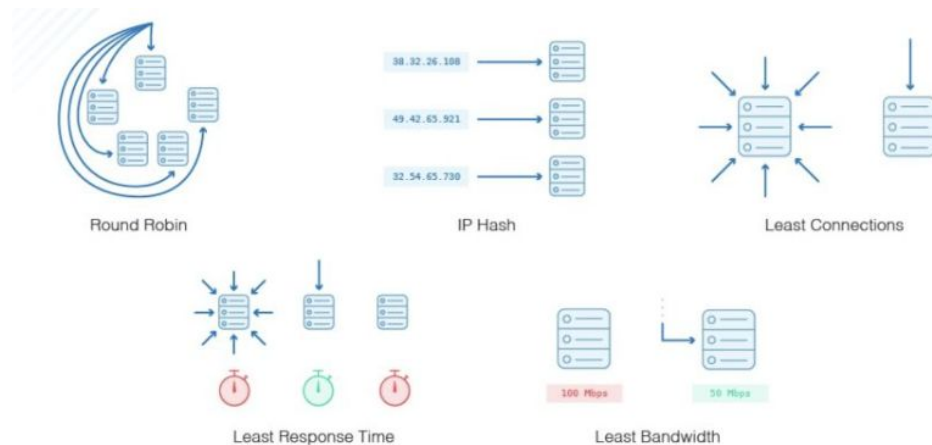
# Types of Load Balancers

- Based on Layer
  - Layer 4 (Transport Layer): Operates at TCP/UDP level, distributing requests based on network-level data.
  - Layer 7 (Application Layer): Operates at HTTP/HTTPS level, making routing decisions based on request content.
- Based on Deployment
  - Hardware Load Balancers: Specialized devices (e.g., F5, Citrix NetScaler).
  - Software Load Balancers: Nginx, HAProxy, Envoy.
  - Cloud-based Load Balancers: AWS Elastic Load Balancer (ELB), Google Cloud Load Balancing.



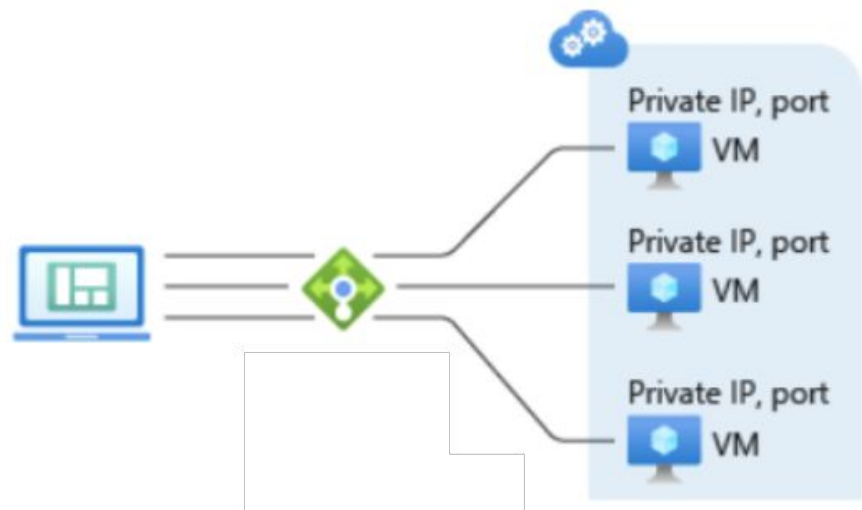
# Load Balancing Strategies

- Static Load Balancing
  - Round Robin: Distributes requests sequentially to each server.
  - Least Connections: Directs traffic to the server with the fewest connections.
  - IP Hashing: Routes requests based on client IP.
- Dynamic Load Balancing
  - Least Response Time: Sends requests to the server with the fastest response.
  - Adaptive Load Balancing: Uses real-time monitoring to make decisions.
  - Weighted Load Balancing: Assigns different weights to servers based on capacity.



# Load Balancer in Action

- Example Scenario:
  - A web application with multiple servers.
  - Users send requests, and the load balancer distributes them efficiently.



# Choosing the Right Load Balancer

- Layer 4 vs. Layer 7: When to use each type.
- Scalability Needs: Matching the right load balancer to traffic volume.
- Security Concerns: SSL termination and DDoS protection.
- Use Cases:
  - Nginx/HAProxy for web applications.
  - AWS ELB for cloud-native applications.
  - Hardware Load Balancers for enterprise data centers.

# Interview Questions on Load Balancing

- Fundamentals
  - What is load balancing, and why is it important?
  - Explain the difference between Layer 4 and Layer 7 load balancing.
  - How does a load balancer handle high availability and failover?
- Strategies & Use Cases
  - Compare Round Robin and Least Connections strategies.
  - What are the advantages of Weighted Load Balancing?
  - When would you use a software load balancer over a hardware one?
- Real-World Scenarios
  - How would you design a scalable load balancing solution for a large e-commerce site?
  - What factors should be considered when choosing a load balancing strategy?
  - How does a load balancer improve security?

# Summary & Key Takeaways

- Load balancing enhances scalability, reliability, and performance.
- Different types exist based on layers and deployment models.
- Choosing the right strategy depends on traffic patterns and system needs.
- Essential for highly available and resilient architectures.
- What's next:
  - Autoscaling & Best Practices in Cloud Environments



---

---

# Autoscaling & Best Practices in Cloud Environments

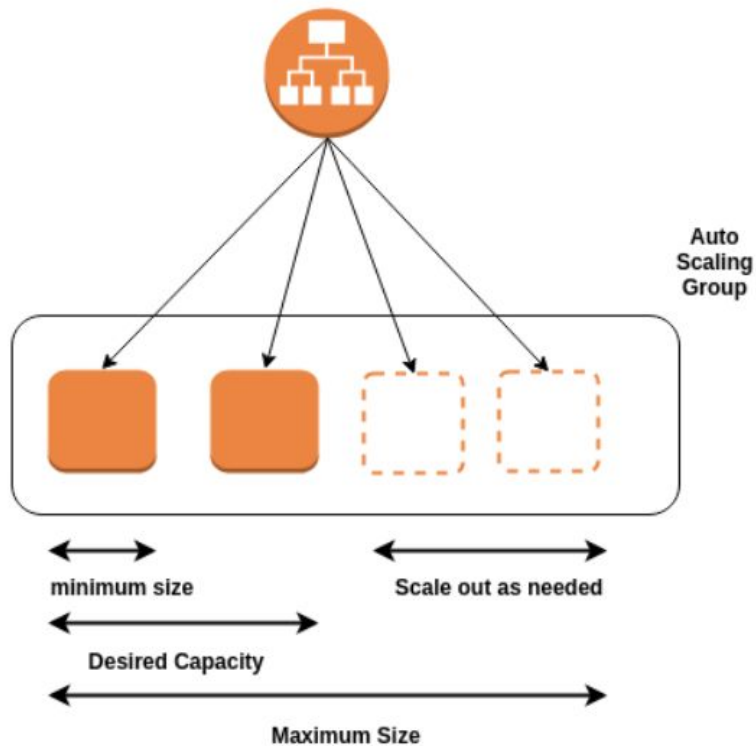
Scalability In System Design

---

---

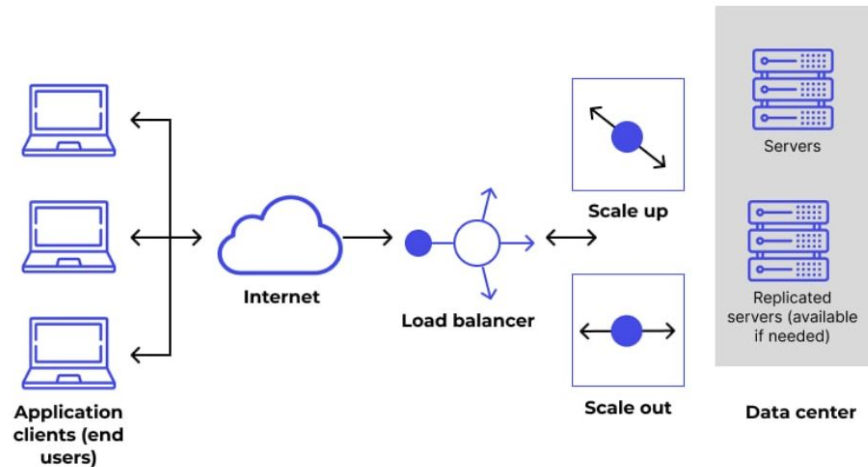
# What is Autoscaling?

- Autoscaling = automatic adjustment of compute resources based on load
- Ensures performance, availability, and cost-efficiency
- Common in microservices, web apps, and event-driven systems



# How Autoscaling Works

- Triggers:
  - CPU usage
  - Memory
  - Request rate
  - Queue length
- Types:
  - Horizontal Scaling: Add/remove instances
  - Vertical Scaling: Resize a single instance
- Scaling Policies:
  - Reactive (based on thresholds)
  - Predictive (based on trends)



# Autoscaling Across Cloud Providers

- All major cloud providers support built-in autoscaling across compute and container services.
  -  AWS – Auto Scaling for EC2, Lambda, ECS, EKS
  -  Azure – Autoscaling via VM Scale Sets, App Services, AKS
  -  GCP – Autoscaling with MIGs, Cloud Run, GKE, Functions

# Monitoring & Proactive Scaling

- Use metrics like:
  - CPU, Memory, Network
  - Queue depth, Custom KPIs
- Proactive Scaling:
  - Predictive algorithms (based on ML or trends)
  - Scheduled scaling (known traffic patterns)
- Tools: CloudWatch, Prometheus + Grafana, Azure Monitor, GCP Operations

# Cost Optimization Strategies

- Avoid over-provisioning—scale just enough
- Use spot/preemptible instances for batch workloads
- Apply resource limits & quotas
- Rightsize regularly based on actual usage
- Use auto-pausing or scale-to-zero features for idle services

# Interview Questions





- Conceptual Understanding
  - What is autoscaling, and why is it important in distributed systems?
  - What's the difference between horizontal and vertical scaling?
  - How does predictive autoscaling work?
- Cloud-Specific Scenarios
  - How does autoscaling work in AWS/Azure/GCP?
  - How would you set up autoscaling for a containerized application?
  - What metrics would you monitor for effective autoscaling?
- Best Practices
  - How can you ensure cost optimization when implementing autoscaling?
  - What are some challenges with autoscaling in real-time systems?

# Summary and Key Takeaways

- Autoscaling ensures agility, availability, and cost-efficiency
- Choose the right scaling strategy (horizontal, vertical, predictive)
- Monitor proactively to stay ahead of load
- Align autoscaling with cost goals and traffic patterns
- What's next:
  - Summary and recap - Scalability in system design



# Section Summary - Scalability in System Design

- In this section, we explored the core building blocks of scalability in modern systems:
  -  Introduction to Scalability – Why systems need to scale and the challenges that come with it
  -  Scaling Strategies – Deep dive into Horizontal, Vertical, and Diagonal scaling approaches
  -  Load Balancing – How different types of load balancers and algorithms distribute traffic efficiently
  -  Autoscaling – Best practices and how cloud platforms like AWS, Azure, and GCP support autoscaling
- What's Next:
  - Database and Storage