Chapter 3: Scalability Fundamentals

Introduction: What is Scalability?

Scalability is the ability of a system to handle growing amounts of work by adding resources.

Think of it like a restaurant:

- **Day 1:** 20 customers → 1 chef, 2 tables ✓
- **Month 6:** 100 customers → Still 1 chef, 2 tables ✗ (chaos!)
- **Solution:** Scale the restaurant!

A scalable system maintains or improves performance as demand increases.

---

1. Vertical Scaling vs Horizontal Scaling

The two fundamental approaches to scaling any system.

Vertical Scaling (Scale Up) - Making the Server Bigger

Definition: Adding more power to your existing machine (more CPU, RAM, storage)

Analogy: Making your car bigger and more powerful

```
Before Scaling:
┌─────────────────┐
│  Web Server     │
│  4 CPU cores    │  Handles 1,000 requests/sec
│  8 GB RAM       │
│  100 GB SSD     │
└─────────────────┘

After Vertical Scaling:
┌─────────────────┐
│  Web Server     │
│  16 CPU cores   │  Handles 4,000 requests/sec
│  64 GB RAM      │
│  1 TB SSD       │
└─────────────────┘
```

Real-World Example: Database Server

```
Current Server:
- Handling 1,000 queries/second
- Response time: 50ms
- CPU at 80% usage
- Problem: Getting slow!

Vertical Scaling Solution:
Old: 8-core CPU, 32 GB RAM → $200/month
New: 32-core CPU, 128 GB RAM → $800/month

Result:
- Handles 4,000 queries/second
- Response time: 20ms
- CPU at 40% usage
```

Pros and Cons

Advantages ✓

- **Simple:** Just upgrade, no code changes
- **No complexity:** Single machine, easier to manage
- **Data consistency:** Everything in one place
- **Fast communication:** No network latency between components

Disadvantages ✗

- **Hardware limits:** Can't scale infinitely (physical limits)
- **Expensive:** High-end servers cost exponentially more
- **Single point of failure:** If server dies, entire system goes down
- **Downtime required:** Need to shut down for upgrades
- **Diminishing returns:** 2x cost ≠ 2x performance

Cost Reality:

Server Specs	Monthly Cost	Performance Gain
4 cores, 16 GB RAM	$100	Baseline
8 cores, 32 GB RAM	$200	2x
16 cores, 64 GB RAM	$400	3.5x (not 4x!)
32 cores, 128 GB RAM	$800	6x (not 8x!)
64 cores, 256 GB RAM	$2,000	9x (not 16x!)

Notice: **Cost grows faster than performance!**

Horizontal Scaling (Scale Out) - Adding More Servers

Definition: Adding more machines to handle the load

Analogy: Instead of one huge truck, use multiple smaller trucks

```
Before Scaling:
┌─────────────────┐
│  Web Server     │  Handles 1,000 requests/sec
│  4 CPU cores    │
│  8 GB RAM       │
└─────────────────┘

After Horizontal Scaling:
┌──────────────┐
│Load Balancer │  Distributes traffic
└──────┬───────┘
       │
  ┌────┴────┬────────┬────────┐
  │    │    │    │
  ↓    ↓    ↓    ↓
┌──────┐ ┌──────┐ ┌──────┐ ┌──────┐
│Server│ │Server│ │Server│ │Server│
│  1   │ │  2   │ │  3   │ │  4   │
└──────┘ └──────┘ └──────┘ └──────┘

Each handles 250 requests/sec
Total: 1,000 requests/sec

If load increases to 2,000:
Just add 4 more servers!
```

Real-World Example: E-Commerce Website

```
Black Friday Sale Scenario:

Normal Day:
- 10,000 concurrent users
- 2 web servers
- Works perfectly

Black Friday:
- 100,000 concurrent users (10x traffic!)
- Vertical scaling: Impossible to get 10x more powerful server
- Horizontal scaling: Add 18 more servers → 20 total
- Result: Handles load successfully!
```

Pros and Cons

Advantages ✓

- **Unlimited scaling:** Add as many servers as needed
- **Cost-effective:** Linear cost growth
- **Fault tolerance:** If one server fails, others continue
- **No downtime:** Add servers without stopping system
- **Geographic distribution:** Servers worldwide reduce latency

Disadvantages ✗

- **Complexity:** Need load balancers, coordination
- **Data consistency:** Harder to keep data in sync
- **Code changes:** Application must support distributed architecture
- **Network overhead:** Communication between servers adds latency

---

Side-by-Side Comparison

```
Aspect            Vertical Scaling      Horizontal Scaling
─────────────────────────────────────────────────────────────
Implementation      Easy                  Complex
Cost at scale       Very expensive        Linear growth
Maximum capacity    Hardware limited      Nearly unlimited
Single point fail   Yes                   No (redundant)
Data consistency    Easy                  Challenging
Setup time          Minutes               Hours/Days
Typical use case    Databases             Web servers, APIs
```

Example	Upgrade to 64-core	Add 10 more servers

When to Use Each?

Use Vertical Scaling When:

✓ Application can't be distributed (legacy monolith)
✓ Database that needs ACID transactions
✓ Quick fix needed (just upgrade hardware)
✓ Small to medium scale (< 10,000 users)
✓ Budget available for expensive hardware

Use Horizontal Scaling When:

✓ Need to handle massive traffic
✓ Require high availability (no downtime)
✓ Want geographic distribution
✓ Cost-conscious growth
✓ Modern cloud-native application

Best Practice: Hybrid Approach

```
Typical Modern Architecture:

Load Balancer
    │
    ├───→ Web Servers (Horizontal) ─┐
    ├───→ Web Servers (Horizontal) ─┤
    └───→ Web Servers (Horizontal) ─┤
                                    │
                                    ↓
                          ┌─────────────────────┐
                          │ Database (Master) │
                          │ Vertical Scaling  │
                          │ 32 cores,128GB    │
                          └──────────┬──────────┘
                                    │
                        ┌────────────┴────────────┐
                        ↓               ↓
                  ┌────────────┐  ┌────────────┐
                  │ Replica  │  │ Replica  │
                  │ (Reads)  │  │ (Reads)  │
                  └────────────┘  └────────────┘

Web layer: Horizontal (stateless, easy to scale)
DB layer: Vertical + replication (stateful, harder to scale)
```

---

2. Understanding Bottlenecks

A **bottleneck** is the component that limits overall system performance.

Analogy: Traffic jam - doesn't matter how wide the highway is if there's a one-lane bridge in the middle!

Identifying Bottlenecks

```
System Performance Chain:
┌────────┐    ┌────────┐    ┌────────┐    ┌────────┐
│ Client │───→│ Server │───→│  DB    │───→│ Disk   │
└────────┘    └────────┘    └────────┘    └────────┘
  1ms     5ms      100ms    500ms

Bottleneck: Disk! (slowest component)
Total response time: 606ms
Even if you make Client 10x faster → only saves 0.9ms!
But if you make Disk 2x faster → saves 250ms!
```

Common Bottlenecks

1. CPU Bottleneck

Symptoms:

- CPU usage consistently > 80%
- Slow response times for compute-heavy tasks
- Server gets hot/loud

Example:

```python
# CPU-intensive task
import time

def cpu_bottleneck_example():
    start = time.time()

    # Heavy computation
    total = 0
    for i in range(10_000_000):
        total += i ** 2

    elapsed = time.time() - start
    print(f"Took {elapsed:.2f} seconds")
    # Output: Took 3.45 seconds

    # CPU at 100% during this time!

# Solutions:
# 1. Optimize algorithm
# 2. Add caching for repeated calculations
# 3. Add more CPU cores (vertical scaling)
# 4. Distribute work across multiple servers (horizontal)
```

Real Example: Video Encoding Service

```
Scenario: Users upload videos to be encoded

Single Server (4 cores):
- Can encode 4 videos simultaneously
- Each video takes 10 minutes
- Throughput: 24 videos/hour
- Problem: 100 users uploading → 4+ hour wait!

Solution 1 - Vertical Scaling:
- Upgrade to 16 cores
- Throughput: 96 videos/hour
- Better, but still limited

Solution 2 - Horizontal Scaling:
- Add 10 servers (4 cores each)
- Throughput: 240 videos/hour
- Much better! And can add more as needed
```

---

2. Memory (RAM) Bottleneck

Symptoms:

- Frequent "out of memory" errors
- System using swap space (disk as RAM)
- Performance suddenly drops 100x
- Server becomes unresponsive

Example:

```python
```

```python
import sys

def memory_bottleneck_example():
    # Each user session stored in memory
    user_sessions = {}

    # Simulate 100,000 users
    for user_id in range(100_000):
        user_sessions[user_id] = {
            'data': 'x' * 50_000, # 50 KB per user
            'timestamp': time.time(),
            'preferences': [i for i in range(1000)]
        }

    # Total memory: 100,000 users × 50 KB = 5 GB
    print(f"Memory usage: {sys.getsizeof(user_sessions) / 1024 / 1024:.2f} MB")

    # If server only has 4 GB RAM → CRASH!

# Solutions:
# 1. Use Redis for session storage (separate server)
# 2. Implement session cleanup (delete old sessions)
# 3. Vertical scaling (more RAM)
# 4. Horizontal scaling (distribute users across servers)
```

Real Example: Social Media Feed

```
Problem: Loading user's feed

Bad Approach (Memory bottleneck):
- Load all 10,000 posts into memory
- 10,000 posts × 2 KB each = 20 MB per user
- 1,000 concurrent users = 20 GB RAM needed!

Good Approach:
- Load only 20 posts at a time (pagination)
- 20 posts × 2 KB = 40 KB per user
- 1,000 concurrent users = 40 MB RAM needed!
- 500x less memory!
```

---

3. Database Bottleneck

Symptoms:

- Slow query response times
- Database CPU/memory maxed out
- Queries timing out
- Connection pool exhausted

Example Scenario:

```sql
-- Slow query (bottleneck)
SELECT * FROM users
WHERE email LIKE '%@gmail.com%'
ORDER BY created_at DESC;

-- Scans entire table (millions of rows)
-- Takes 5+ seconds!

-- Performance:
Query time: 5,000ms
Throughput: 0.2 queries/second
With 100 concurrent users → Queue backs up!

-- Solution: Add index
CREATE INDEX idx_users_email ON users(email);
CREATE INDEX idx_users_created ON users(created_at);

-- Now query takes 10ms
-- Throughput: 100 queries/second
-- 500x improvement!
```

Common Database Bottleneck Patterns:

```
1. Missing Indexes
   Problem: Full table scans
   Solution: Add appropriate indexes

2. N+1 Query Problem
   Problem: Making 1000 queries instead of 1

   Bad:
   posts = get_all_posts()        # 1 query
   for post in posts:
       author = get_author(post.author_id)  # 1000 queries!

   Good:
   posts = get_all_posts_with_authors()  # 1 query with JOIN

3. Expensive Joins
   Problem: Joining 5 large tables
   Solution: Denormalize data, use caching

4. Too Many Connections
   Problem: 10,000 clients trying to connect
   Solution: Connection pooling, read replicas
```

---

4. Network Bottleneck

Symptoms:

- High latency between services
- Timeouts
- Packet loss
- Bandwidth saturation

Example:

```
Scenario: Microservices Architecture

Service A (API) → Service B (User Service) → Service C (Database)

Each arrow adds latency:
- A → B: 50ms (network)
- B → C: 100ms (network)
- Total: 150ms just for network!

If user request needs 10 service calls:
150ms × 10 = 1,500ms = 1.5 seconds!

Solutions:
1. Reduce service calls (batch requests)
2. Use caching (avoid repeated calls)
3. Deploy services closer together
4. Use faster network (upgrade bandwidth)
5. Async processing (don't wait for all responses)
```

Real Example: Image Loading

```
Problem: Website loads 50 images per page

Naive approach:
- Each image: 2 MB
- User bandwidth: 10 Mbps
- Time to load ONE image: (2 MB × 8) / 10 Mbps = 1.6 seconds
- Time to load 50 images: 80 seconds!

Optimized approach:
1. Compress images: 2 MB → 200 KB (10x smaller)
2. Use CDN: Serve from nearby server (300ms → 30ms)
3. Lazy loading: Load only visible images first
4. Result: Page loads in 3 seconds instead of 80!
```

---

5. Disk I/O Bottleneck

Symptoms:

- High disk queue length
- Slow read/write operations
- System feels sluggish

Example:

```python
import time

# Writing 100,000 records to disk
def disk_bottleneck_example():
    # Bad: Write one at a time
    start = time.time()
    with open('data.txt', 'w') as f:
        for i in range(100_000):
            f.write(f"Record {i}\n")
            f.flush()  # Force write to disk each time

    elapsed = time.time() - start
    print(f"Unbuffered: {elapsed:.2f} seconds")
    # Output: 45 seconds (disk bottleneck!)

    # Good: Batch writes (buffering)
    start = time.time()
    with open('data.txt', 'w') as f:
        for i in range(100_000):
            f.write(f"Record {i}\n")
        # Writes happen in batches automatically

    elapsed = time.time() - start
    print(f"Buffered: {elapsed:.2f} seconds")
    # Output: 0.5 seconds (90x faster!)
```

Bottleneck Identification Strategy

```
Step 1: Monitor Everything

┌─────────────────────────────────────┐
│ System Metrics Dashboard            │
├─────────────────────────────────────┤
│ CPU Usage:       45%                │
│ Memory Usage:    78%                │
│ Disk I/O:        95% ← BOTTLENECK!  │
│ Network:         20%                │
│ Database Queries: 50ms avg          │
└─────────────────────────────────────┘

Step 2: Trace a Request
User Request → Load Balancer (2ms)
```

```
        → Web Server (10ms)
        → Cache Miss
        → Database Query (500ms) ← BOTTLENECK!
        ← Return Response
Total: 512ms

Step 3: Profile the Code
Function          Time    % of Total
─────────────────────────────────────
load_user_data()    450ms   88% ← BOTTLENECK!
render_template()   30ms    6%
send_response()     32ms    6%

Step 4: Optimize the Bottleneck
- Add caching: 450ms → 5ms
- Overall: 512ms → 67ms (7.6x faster!)
```

---

3. Performance Metrics

The Three Pillars of System Performance

```
┌─────────────────────────────────────────────┐
│           PERFORMANCE                  │
│                                   │
│  ┌──────────┐ ┌──────────┐ ┌──────────┐ │
│  │ Latency  │ │ Throughput │ │Availability│ │
│  │(Response │ │(Capacity)  │ │(Uptime)   │ │
│  │ Time)    │ │            │ │           │ │
│  └──────────┘ └──────────┘ └──────────┘ │
│    ↓          ↓          ↓      │
│ "How fast?"  "How many?"  "How reliable?" │
└─────────────────────────────────────────────┘
```

---

1. Latency (Response Time)

Definition: Time from request to response

Types of Latency:

```
┌──────────────────────────────────────────────┐
│              Request Journey                 │
├──────────────────────────────────────────────┤
│                                              │
│ Client → Network → Server → Processing → Response │
│                                              │
│  5ms    50ms     10ms     100ms     50ms     │
│                                              │
│ Total Latency: 215ms                         │
└──────────────────────────────────────────────┘

Breakdown:
- Network Latency: 50ms + 50ms = 100ms (46%)
- Processing Latency: 100ms (47%)
- Client Overhead: 5ms + 10ms = 15ms (7%)
```

Latency Percentiles (Critical Concept!):

```
Why Percentiles Matter:

Average latency: 100ms ← Misleading!
Could mean:
- Scenario A: All requests take 100ms (good!)
- Scenario B: 99% take 10ms, 1% take 9 seconds (bad!)

Better Metrics:
┌─────────────────────────────────┐
│ Latency Distribution            │
├─────────────────────────────────┤
│ P50 (median):   50ms    │ 50% of requests
│ P90:            150ms   │ 90% of requests
│ P95:            300ms   │ 95% of requests
│ P99:            1,000ms │ 99% of requests
│ P99.9:          5,000ms │ 99.9% of requests
└─────────────────────────────────┘

Real Example:
If you have 1 million requests/day:
- P99.9 = 5s means 1,000 users experience 5+ second delays!
```

Measuring Latency:

python

```python
import time
import statistics

def measure_latency():
    latencies = []

    for i in range(1000):
        start = time.time()

        # Simulate operation (e.g., API call)
        response = make_api_call()

        latency = (time.time() - start) * 1000  # Convert to ms
        latencies.append(latency)

    # Calculate percentiles
    latencies.sort()

    print(f"Average latency: {statistics.mean(latencies):.2f}ms")
    print(f"P50 (median): {latencies[499]:.2f}ms")
    print(f"P90: {latencies[899]:.2f}ms")
    print(f"P95: {latencies[949]:.2f}ms")
    print(f"P99: {latencies[989]:.2f}ms")
    print(f"P99.9: {latencies[998]:.2f}ms")

# Example output:
# Average latency: 45.23ms
# P50 (median): 40.15ms
# P90: 75.30ms
# P95: 120.45ms
# P99: 450.80ms
# P99.9: 2,100.25ms
```

Real-World Latency Targets:

Application Type	P50	P95	P99	User Experience
Search Engine	<50ms	<100ms	<200ms	Feels instant
Social Media Feed	<100ms	<300ms	<800ms	Smooth scrolling
E-commerce Checkout	<200ms	<500ms	<1s	Acceptable
Video Streaming	<2s	<5s	<10s	Tolerable
Batch Processing	N/A	N/A	N/A	Doesn't matter

---

2. Throughput (Capacity)

Definition: Number of operations completed per unit of time

Measuring Throughput:

```
Requests per Second (RPS):

┌─────────────────────────────────────┐
│ Time Period: 1 second               │
│ Requests Completed: 1,000           │
│ Throughput: 1,000 RPS               │
└─────────────────────────────────────┘

Queries per Second (QPS):

┌─────────────────────────────────────┐
│ Database                            │
│ Queries Completed: 5,000/sec        │
│ Throughput: 5,000 QPS               │
└─────────────────────────────────────┘

Bits per Second (bps):

┌─────────────────────────────────────┐
│ Network                             │
│ Data Transferred: 100 Megabits/sec  │
│ Throughput: 100 Mbps                │
└─────────────────────────────────────┘
```

Throughput vs Load:

```
System Capacity Curve:

Throughput
  ↑
  │              ┌────── Saturation Point
  │             ╱
10K │          ╱
  │          ╱
 5K │       ╱
  │       ╱
  │      ╱
  │     ╱
  │    ╱
  └──────────────────────────────→ Load
    5K   10K   15K   20K

Observations:
- Up to 10K: Throughput increases linearly
- 10K-15K: System approaching limits
- Beyond 15K: Throughput plateaus (saturated!)
- May even decrease due to overhead
```

Calculating Maximum Throughput:

```python
```

```python
def calculate_max_throughput():
    """
    Calculate theoretical maximum throughput
    """

    # Server specs
    num_servers = 5
    cores_per_server = 8
    total_cores = num_servers * cores_per_server  # 40 cores

    # Request handling
    avg_request_time_ms = 100  # 100ms per request
    requests_per_second_per_core = 1000 / avg_request_time_ms  # 10 RPS

    # Maximum throughput
    max_throughput = total_cores * requests_per_second_per_core

    print(f"Theoretical Max Throughput: {max_throughput} RPS")
    # Output: 400 RPS

    # Real-world (accounting for overhead)
    efficiency = 0.7  # 70% efficiency (realistic)
    actual_max_throughput = max_throughput * efficiency

    print(f"Actual Max Throughput: {actual_max_throughput} RPS")
    # Output: 280 RPS

    return actual_max_throughput
```

Load Testing Example:

```python
```

```python
import concurrent.futures
import requests
import time

def load_test(url, num_requests, concurrent_users):
    """
    Simulate load and measure throughput
    """

    start_time = time.time()
    successful_requests = 0
    failed_requests = 0

    def make_request(request_num):
        try:
            response = requests.get(url, timeout=5)
            return response.status_code == 200
        except:
            return False

    # Send concurrent requests
    with concurrent.futures.ThreadPoolExecutor(max_workers=concurrent_users) as executor:
        results = executor.map(make_request, range(num_requests))

        for success in results:
            if success:
                successful_requests += 1
            else:
                failed_requests += 1

    elapsed_time = time.time() - start_time
    throughput = successful_requests / elapsed_time

    print(f"Total Requests: {num_requests}")
    print(f"Successful: {successful_requests}")
    print(f"Failed: {failed_requests}")
    print(f"Time: {elapsed_time:.2f}s")
    print(f"Throughput: {throughput:.2f} RPS")
    print(f"Success Rate: {(successful_requests/num_requests)*100:.2f}%")

# Example usage:
# load_test('http://localhost:8000/api/test', 1000, 50)

# Example output:
# Total Requests: 1000
# Successful: 980
```

```
# Failed: 20
# Time: 12.34s
# Throughput: 79.42 RPS
# Success Rate: 98.00%
```

---

3. Availability (Uptime)

Definition: Percentage of time a system is operational

The Nines:

Availability	Uptime/Year	Downtime/Year	Downtime/Month	Use Case
90%	328.5 days	36.5 days	3 days	Unacceptable
95%	346.6 days	18.3 days	1.5 days	Poor
99%	360.4 days	3.65 days	7.2 hours	Acceptable
99.9%	364.6 days	8.76 hours	43.2 minutes	Good
99.99%	365.1 days	52.6 minutes	4.32 minutes	Great
99.999%	365.2 days	5.26 minutes	26 seconds	Excellent
99.9999%	365.24 days	31.5 seconds	2.6 seconds	Extreme

Note: Each additional nine gets exponentially harder and more expensive!

Calculating Availability:

```
Formula:
Availability = (Total Time - Downtime) / Total Time × 100%

Example 1: Website down 1 hour in a month
Availability = (720 hours - 1 hour) / 720 hours × 100%
           = 99.86%

Example 2: Service down 5 minutes in a year
Availability = (525,600 min - 5 min) / 525,600 min × 100%
           = 99.999% (Five nines!)
```

Serial vs Parallel Components:

```
Serial System (both must work):
┌─────────────┐      ┌─────────────┐
│ Service A │────→│ Service B │
│   99.9%    │      │   99.9%    │
└─────────────┘      └─────────────┘

Overall Availability = 99.9% × 99.9% = 99.8%
(Worse than individual components!)

Parallel System (failover):
     ┌─────────────┐
  ┌──→│ Service A │──┐
  │  │   99.9%   │ │
  │  └─────────────┘  │
Input │            │→ Output
  │  ┌─────────────┐  │
  └──→│ Service B │──┘
     │  99.9%   │
     └─────────────┘

Overall Availability = 1 - (0.001 × 0.001) = 99.9999%
(Much better! Redundancy improves availability)
```

Real-World Example: E-commerce Platform

```
Architecture:
┌─────────────┐
│Load Balancer│ 99.99%
└──────┬──────┘
       │
   ┌───┴───┬───────┐
   ↓       ↓       ↓
┌──────┐ ┌──────┐ ┌──────┐
│Web 1 │ │Web 2 │ │Web 3 │ 99.9% each
└──┬───┘ └──┬───┘ └──┬───┘
   │        │        │
   └────────┼────────┘
            ↓
   ┌─────────────┐
   │  Database   │ 99.95%
   │  Cluster    │
   └─────────────┘

Calculation:
- Web Servers (parallel): 1 - (0.001)³ = 99.9999%
- Load Balancer: 99.99%
- Database: 99.95%

Overall = 99.99% × 99.9999% × 99.95% = 99.94%

Downtime per year = 365 days × (1 - 0.9994)
                  = 0.219 days
                  = 5.25 hours/year
```

Performance Monitoring Dashboard Example

```python
```

```python
class PerformanceMonitor:
    def __init__(self):
        self.request_times = []
        self.successful_requests = 0
        self.failed_requests = 0
        self.start_time = time.time()

    def record_request(self, duration_ms, success=True):
        self.request_times.append(duration_ms)
        if success:
            self.successful_requests += 1
        else:
            self.failed_requests += 1

    def get_metrics(self):
        if not self.request_times:
            return None

        sorted_times = sorted(self.request_times)
        total_requests = len(self.request_times)
        elapsed_time = time.time() - self.start_time

        return {
            # Latency metrics
            'avg_latency': statistics.mean(sorted_times),
            'p50_latency': sorted_times[int(total_requests * 0.50)],
            'p95_latency': sorted_times[int(total_requests * 0.95)],
            'p99_latency': sorted_times[int(total_requests * 0.99)],

            # Throughput metrics
            'total_requests': total_requests,
            'throughput_rps': total_requests / elapsed_time,

            # Availability metrics
            'success_rate': (self.successful_requests / total_requests) * 100,
            'failure_rate': (self.failed_requests / total_requests) * 100,
        }

    def print_dashboard(self):
        metrics = self.get_metrics()
        if not metrics:
            print("No data yet")
            return

        print("\n" + "="*50)
        print("      PERFORMANCE DASHBOARD")
```

```
print("="*50)
print(f"\nLATENCY:")
print(f"  Average:  {metrics['avg_latency']:.2f}ms")
print(f"  P50:      {metrics['p50_latency']:.2f}ms")
print(f"  P95:      {metrics['p95_latency']:.2f}ms")
print(f"  P99:      {metrics['p99_latency']:.2f}ms")
print(f"\nTHROUGHPUT:")
print(f"  Total:    {metrics['total_requests']} requests")
print(f"  RPS:      {metrics['throughput_rps']:.2f}")
print(f"\nAVAILABILITY:")
print(f"  Success:  {metrics['success_rate']:.2f}%")
print(f"  Failure:  {metrics['failure_rate']:.2f}%")
print("="*50 + "\n")
```

---

4. Back-of-the-Envelope Calculations

Essential skill for system design interviews and capacity planning!

Key Numbers to Memorize

```
┌─────────────────────────────────────────────────┐
│         LATENCY NUMBERS (APPROXIMATE)           │
├─────────────────────────────────────────────────┤
│ L1 cache reference            0.5 ns            │
│ L2 cache reference            7 ns              │
│ Main memory reference         100 ns            │
│ Read 1 MB from memory         250 μs            │
│ SSD random read               150 μs            │
│ Read 1 MB from SSD            1 ms              │
│ Disk seek                     10 ms             │
│ Read 1 MB from disk           20 ms             │
│ Send packet CA→Netherlands→CA  150 ms           │
└─────────────────────────────────────────────────┘

┌─────────────────────────────────────────────────┐
│         DATA SIZE CONVERSIONS                   │
├─────────────────────────────────────────────────┤
│ 1 Byte = 8 bits                                 │
│ 1 KB = 1,000 Bytes (or 1,024 in binary)         │
│ 1 MB = 1,000 KB                                 │
│ 1 GB = 1,000 MB                                 │
│ 1 TB = 1,000 GB                                 │
│ 1 PB = 1,000 TB                                 │
└─────────────────────────────────────────────────┘
```

```
┌──────────────────────────────────────────────────┐
│           TIME CONVERSIONS                       │
├──────────────────────────────────────────────────┤
│ 1 second = 1,000 milliseconds (ms)               │
│ 1 ms = 1,000 microseconds (μs)                   │
│ 1 μs = 1,000 nanoseconds (ns)                    │
│ 1 day = 86,400 seconds  (≈ 100,000 for quick calc) │
│ 1 month ≈ 2.5 million seconds                    │
│ 1 year ≈ 30 million seconds                      │
└──────────────────────────────────────────────────┘
```

Calculation Examples

Example 1: Twitter-like System

Question: Design Twitter. Estimate storage requirements.

Step 1: Define Assumptions

- 300 million daily active users (DAU)
- Each user tweets 2 times per day on average
- 10% of tweets contain an image
- Average tweet size: 300 bytes (text)
- Average image size: 2 MB

Step 2: Calculate Daily Data

Tweets per day:
  300M users × 2 tweets = 600M tweets/day

Text storage:
  600M tweets × 300 bytes = 180 GB/day

Images:
  600M tweets × 10% = 60M images/day
  60M images × 2 MB = 120 TB/day

Total daily storage:
  180 GB + 120 TB ≈ 120 TB/day

Step 3: Calculate 5-Year Storage

---

5 years = 5 × 365 days = 1,825 days
Total storage = 120 TB/day × 1,825 days
              = 219,000 TB
              = 219 PB

With replication (3 copies for reliability):
  219 PB × 3 = 657 PB

Step 4: Bandwidth Requirements

Upload bandwidth:
  120 TB per day / 86,400 seconds
  = 120×10^{12} bytes / 86,400 sec
  = 1.4×10^{9} bytes/sec
  = 11.1 Gbps

Read:write ratio is typically 10:1 for social media
Download bandwidth:
  11.1 Gbps × 10 = 111 Gbps

Step 5: Servers Needed

```
Assume each server can handle:
- 10,000 requests/second
- 1 Gbps network

Peak requests (assume 2x daily average):
  600M tweets/day / 86,400 sec × 2 = 14,000 requests/sec

Servers for request handling:
  14,000 / 10,000 = 2 servers (minimum)
  Add redundancy: 4-6 servers

Servers for bandwidth:
  111 Gbps / 1 Gbps = 111 servers

Therefore, we need ~120 servers for bandwidth alone!
```

Example 2: URL Shortener (like bit.ly)

Question: How many unique short URLs can we generate?

Step 1: Choose Character Set

Options:
- Numbers only (0-9): 10 characters
- Lowercase (a-z): 26 characters
- Alphanumeric (a-z, A-Z, 0-9): 62 characters

Let's use alphanumeric: 62 characters

Step 2: Choose URL Length

Length	Combinations	Readable?
3	62^3 = 238K	Too short!
4	62^4 = 14.7M	Too short
5	62^5 = 916M	Getting there
6	62^6 = 56.8B	Good!
7	62^7 = 3.5T	Plenty!

Step 3: Calculate Years of Service

Assumptions:
- 1 million new URLs per day
- 6-character URLs
- 56.8 billion possible combinations

Years of service = 56.8B / (1M per day × 365 days)
= 56.8B / 365M
= 155 years

Conclusion: 6 characters is sufficient!

Step 4: Storage Requirements

Per URL record:
- Short code: 6 bytes
- Original URL: 200 bytes (average)
- Created date: 8 bytes
- User ID: 8 bytes
- Metadata: 50 bytes

Total: ~272 bytes per record

For 56.8 billion URLs:
56.8B × 272 bytes = 15.4 TB

With indexing overhead (2x):

15.4 TB × 2 = 30.8 TB

Very manageable!

Example 3: YouTube Video Storage

Question: How much storage for 1 million videos?

Step 1: Video Size Assumptions

Average video:
- Length: 10 minutes
- Quality: 1080p
- Size: ~100 MB per minute
- Total: 10 min × 100 MB = 1 GB per video

Multiple quality versions:
- 4K: 10 GB
- 1080p: 1 GB
- 720p: 0.5 GB
- 480p: 0.2 GB
- 360p: 0.1 GB

Total per video: 11.8 GB

Step 2: Calculate for 1M Videos

Storage = 1M videos × 11.8 GB
        = 11.8 million GB
        = 11.8 PB

Step 3: Add Thumbnails and Metadata

Thumbnails: 1M × 100 KB = 100 GB
Metadata: 1M × 10 KB = 10 GB
Total extra: ~110 GB (negligible compared to videos)

Step 4: Factor in Redundancy

---

3 copies for reliability:
  11.8 PB × 3 = 35.4 PB

Step 5: Daily Upload Rate

Assume 10,000 videos uploaded daily:
  10,000 videos × 11.8 GB = 118 TB/day

Bandwidth needed:
  118 TB / 86,400 sec = 1.4 GB/sec = 11 Gbps upload

Step 6: Viewing Bandwidth

Assume:
- 100 million views per day
- Average video watched: 5 minutes

```
- Average quality: 720p (0.5 GB per 10 min)

Data served:
  100M views × (5 min / 10 min) × 0.5 GB
  = 100M × 0.5 × 0.5 GB
  = 25 million GB
  = 25 PB per day

Bandwidth:
  25 PB / 86,400 sec = 289 GB/sec = 2.3 Tbps

This is why YouTube uses CDNs worldwide!
```

Example 4: Real-Time Chat Application

Question: WhatsApp-like system for 1 billion users.

Step 1: Active Users

Total users: 1 billion
Daily active users (DAU): 500 million (50%)
Peak concurrent users: 50 million (10% of DAU)

Step 2: Message Volume

Average messages per user per day: 50
Total daily messages: 500M × 50 = 25 billion messages

Messages per second (average):
  25B / 86,400 sec = 289,000 messages/sec

Peak (2x average):
  578,000 messages/sec

Step 3: Message Size

Text message: 100 bytes (average)
Image: 2 MB (10% of messages)
Video: 10 MB (1% of messages)

Average message size:
  (0.89 × 100 bytes) + (0.10 × 2 MB) + (0.01 × 10 MB)
  = 89 bytes + 0.2 MB + 0.1 MB
  ≈ 0.3 MB

Step 4: Daily Storage

25B messages × 0.3 MB = 7.5 PB/day

Annual storage:
  7.5 PB × 365 = 2,737 PB ≈ 2.7 EB

With 5-year retention:
  2.7 EB × 5 = 13.5 EB

Step 5: Connection Requirements

Peak concurrent users: 50 million
Each maintains WebSocket connection

Assume each server handles 10,000 connections:
  50M / 10,000 = 5,000 servers

Step 6: Bandwidth
───────────────────────
Peak message rate: 578,000/sec
Average message: 0.3 MB

Bandwidth = 578,000 × 0.3 MB = 173 GB/sec = 1.4 Tbps

Step 7: Database Queries
─────────────────────────────────
Each message requires:
- 1 write (sender)
- 1 read (recipient)
- Plus metadata updates

Query rate: 578,000 × 3 = 1.7M queries/sec

If one DB handles 10,000 queries/sec:
  1.7M / 10,000 = 170 database servers

With read replicas (10:1 read:write ratio):
- Write DBs: 17 servers
- Read DBs: 153 servers

Quick Estimation Framework

For ANY system design problem, calculate:

1. USERS
   - Total users
   - Daily active users (DAU)
   - Peak concurrent users

2. TRAFFIC
   - Requests/actions per day
   - Requests per second (RPS)
   - Peak RPS (typically 2-3x average)

3. STORAGE
   - Data per item/record
   - Daily data generated
   - Total storage needed (5-10 years)
   - With replication (3x)

4. BANDWIDTH
   - Upload bandwidth

```
  - Download bandwidth
  - Read:write ratio

5. SERVERS
  - Based on RPS capacity
  - Based on storage needs
  - Based on connection limits

6. MEMORY/CACHE
  - Cache hit ratio assumption (80-90%)
  - Hot data size (20% of total)
  - RAM needed per server
```

Key Takeaways

1. **Scaling Strategies:**
   - Vertical: Quick but limited and expensive
   - Horizontal: Complex but unlimited and cost-effective
   - Use hybrid: Horizontal for stateless, vertical for stateful
2. **Bottlenecks:**
   - Always monitor: CPU, Memory, Disk, Network, Database
   - Optimize the slowest component first
   - Measure before and after optimization
3. **Performance Metrics:**
   - Latency: Use percentiles (P95, P99), not just averages
   - Throughput: Plan for 2-3x peak load
   - Availability: Each nine gets exponentially harder
4. **Calculations:**
   - Memorize key numbers
   - Break down into steps
   - Round for simplicity
   - State your assumptions clearly

Practice Problems

Problem 1: Instagram-like app with 500M DAU, users upload 2 photos/day. Calculate: Daily storage, bandwidth, servers needed.

Problem 2: Design a cache for a system getting 10,000 RPS. Calculate: Cache size if 80% requests should hit cache.

Problem 3: Video conferencing app, 1M concurrent calls. Calculate: Bandwidth requirements.

Next Chapter Preview

In Chapter 4, we'll dive into **Load Balancing**:

- How load balancers distribute traffic
- Different algorithms (Round Robin, Least Connections, etc.)
- Layer 4 vs Layer 7 load balancing
- Health checks and failover

Ready to continue? Let me know!