



Designing Scalable Systems and Cloud Integration

A comprehensive technical guide for software engineers and architects covering design patterns, resilience strategies, security fundamentals, and practical implementation through real-world case studies.

Session Overview

Design Patterns for Scale

Horizontal scaling, partitioning, replication, and caching strategies for high-performance systems

Resilience & Failure Handling

Circuit breakers, retries, rate limiting, and multi-region design patterns

Security & Compliance

IAM, RBAC, secure design practices, and regulatory compliance fundamentals

Hands-On Case Studies

Real-world scenarios including e-commerce flash sales and collaborative document editing systems

Learning Objectives

1 Master Scalability Patterns

Apply horizontal scaling, partitioning, and replication strategies to handle millions of concurrent users

3 Secure System Design

Integrate security best practices and compliance requirements into architectural decisions

2 Implement Resilient Architectures

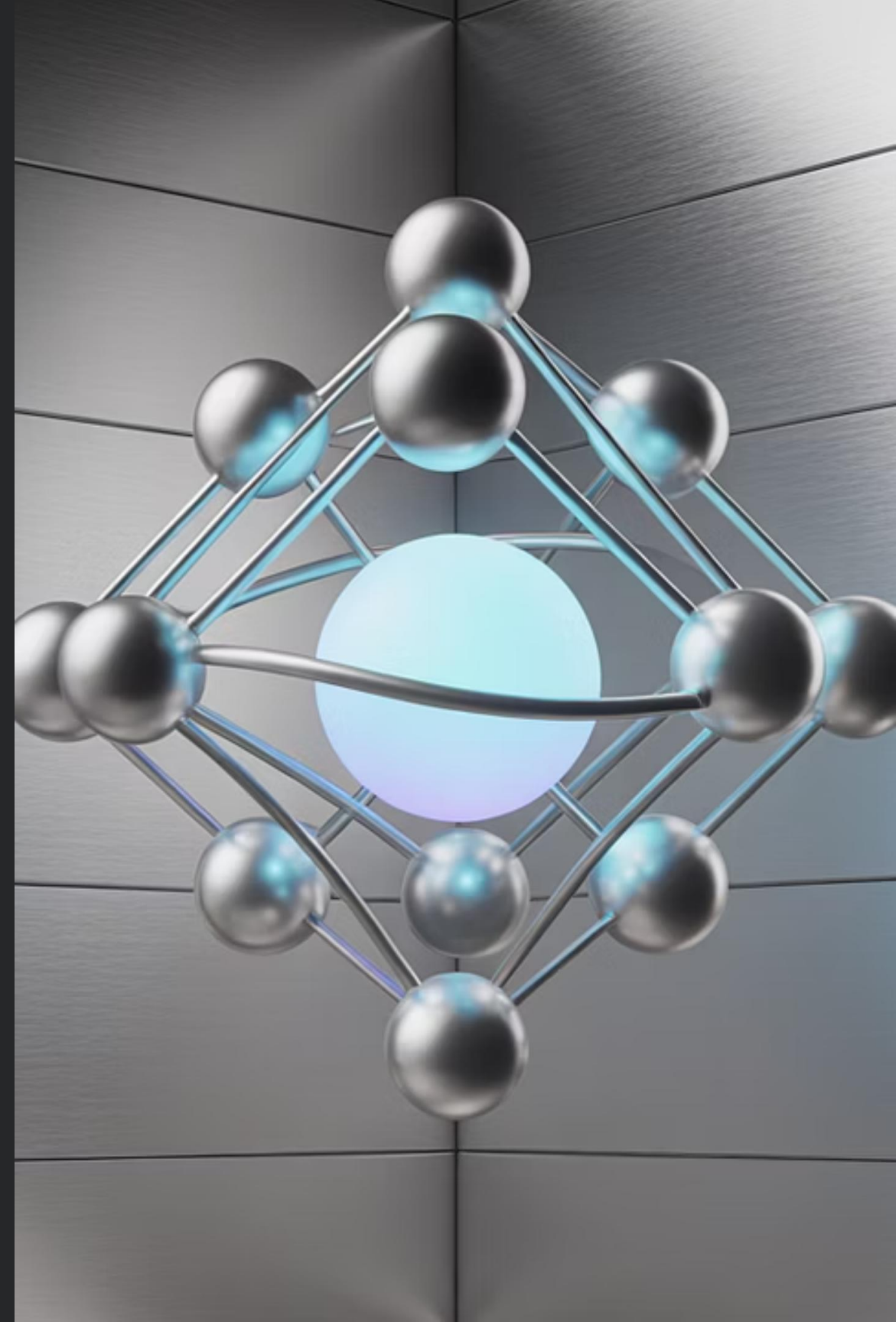
Design fault-tolerant systems using circuit breakers, retries, and multi-region failover mechanisms

4 Real-World Application

Solve complex scaling challenges through hands-on case studies and practical implementation

Design Patterns for Scale

Building Systems That Handle Growth



Horizontal Scaling Fundamentals

Horizontal scaling distributes load across multiple instances rather than upgrading single machines. This approach provides better fault tolerance and cost efficiency compared to vertical scaling.

Key Benefits

- Linear performance improvement with added resources
- Natural fault isolation between instances
- Cost-effective scaling using commodity hardware
- Geographic distribution capabilities



Data Partitioning Strategies



1

Horizontal Partitioning (Sharding)

Split data across multiple databases based on key ranges or hash functions. Enables linear scalability but requires careful key distribution.

2

Vertical Partitioning

Separate tables or columns into different databases based on access patterns. Optimizes performance for specific query types.

3

Functional Partitioning

Distribute data by business domain or service boundaries. Aligns with microservices architecture and domain-driven design.

Replication Patterns

Master-Slave

Single write node with multiple read replicas. Provides read scalability but creates write bottlenecks.

- Simple consistency model
- Easy failover procedures
- Limited write throughput

Master-Master

Multiple write nodes with bidirectional replication. Enables write scalability but requires conflict resolution.

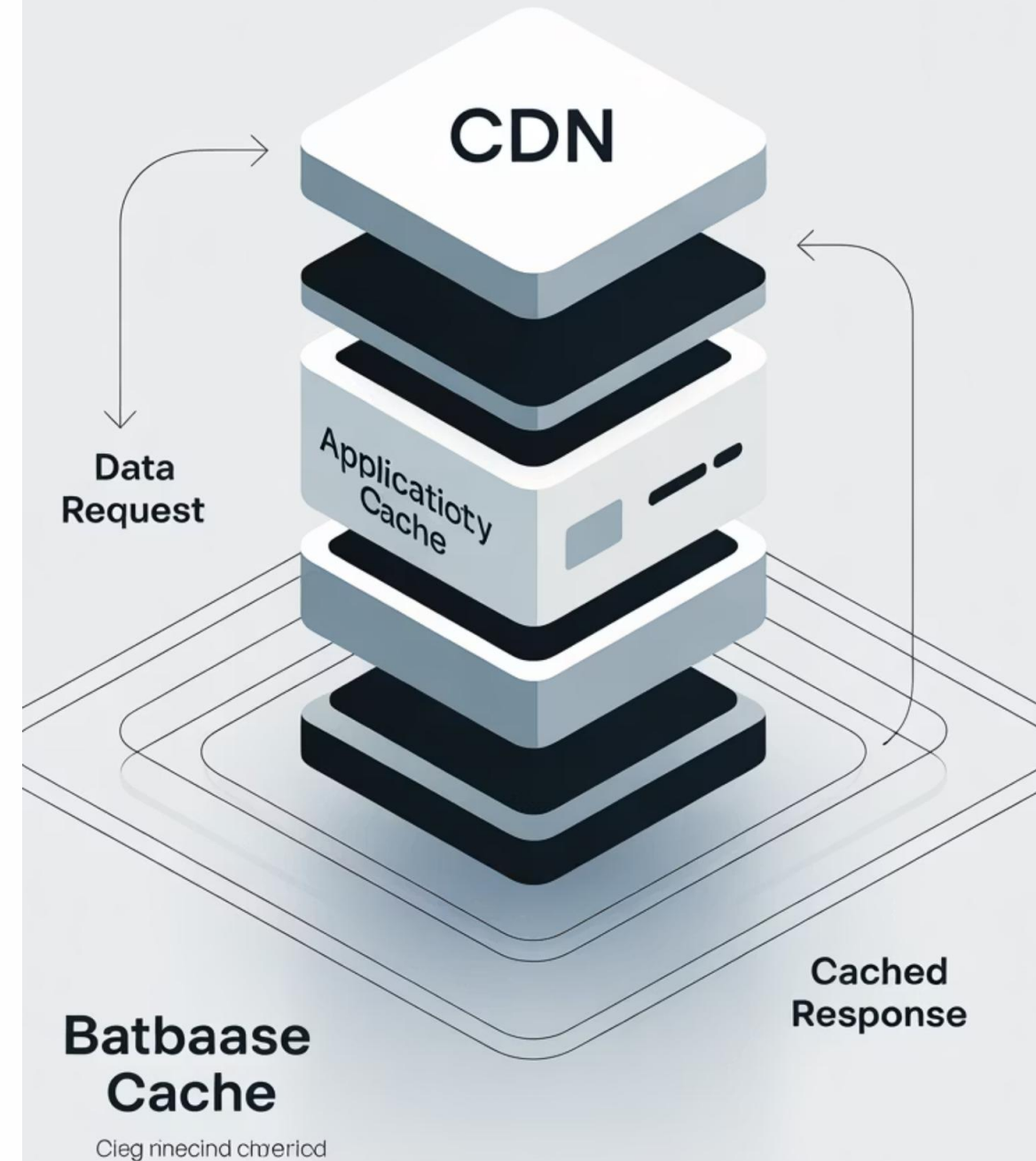
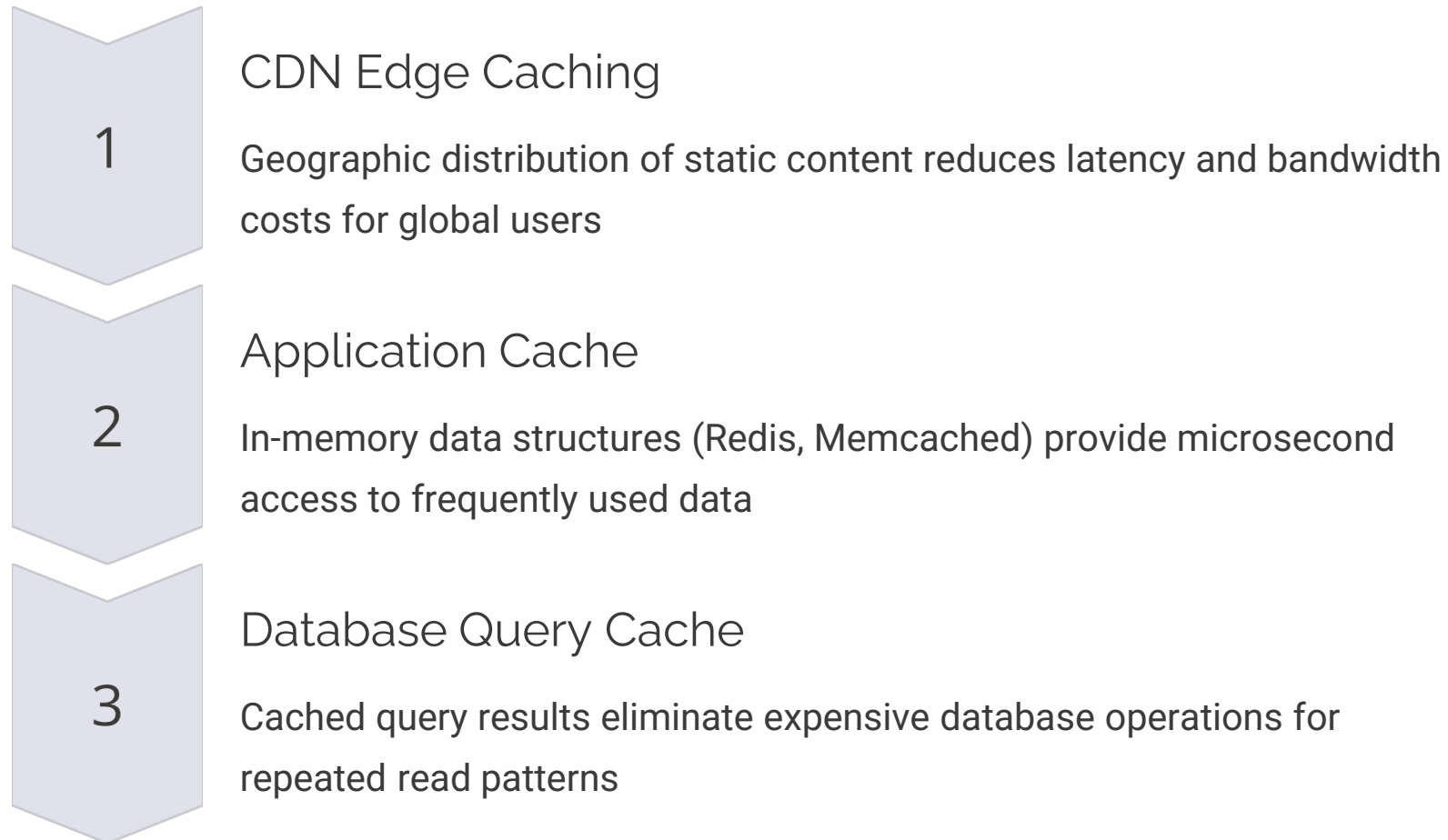
- High write availability
- Complex conflict handling
- Eventual consistency

Quorum-Based

Distributed consensus using majority votes. Balances consistency with availability in distributed systems.

- Tunable consistency levels
- Partition tolerance
- Network overhead

Caching Layer Architecture



Redis vs Memcached: Technical Comparison

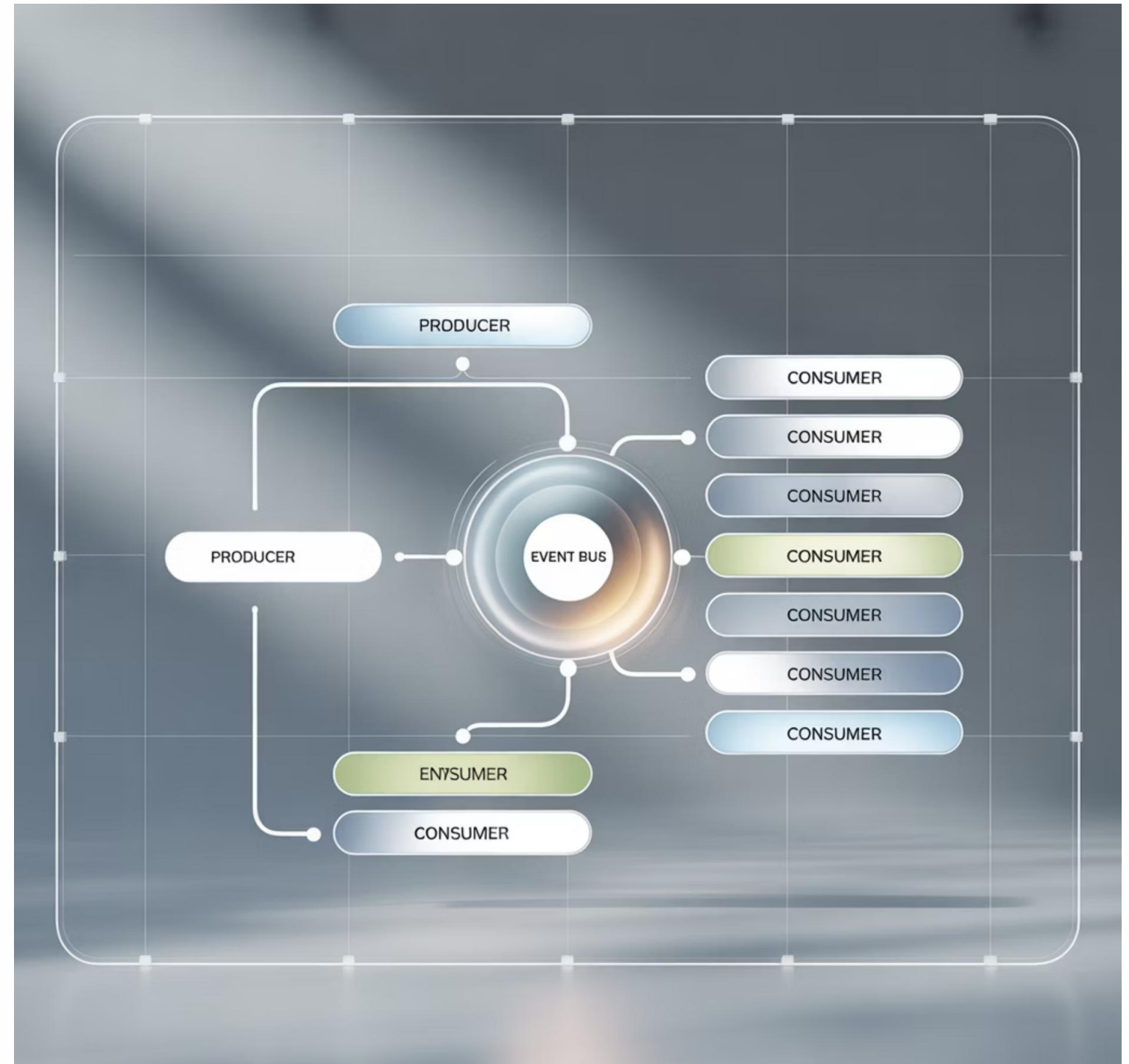
Feature	Redis	Memcached
Data Structures	Strings, hashes, lists, sets, sorted sets, bitmaps, HyperLogLog	Simple key-value strings only
Persistence	RDB snapshots, AOF logging	Memory-only, no persistence
Replication	Master-slave with automatic failover	No built-in replication
Threading	Single-threaded with event loop	Multi-threaded
Memory Usage	Higher overhead per key	Lower memory footprint
Use Cases	Complex data operations, pub/sub, sessions	Simple caching, high throughput

Event-Driven Architecture Fundamentals

Core Principles

Event-driven systems decouple producers from consumers, enabling asynchronous processing and improved scalability.

- Loose coupling between components
- Asynchronous message processing
- Event sourcing and replay capabilities
- Natural integration with microservices



Pub/Sub vs Message Queues

Publish/Subscribe

One-to-many broadcast pattern where publishers send messages to topics, and multiple subscribers receive copies.

- Decoupled fan-out messaging
- Multiple consumers per message
- Topic-based routing

Message Queues

One-to-one delivery pattern where messages are consumed by single workers from queues.

- Load distribution across workers
- Guaranteed delivery semantics
- Queue-based routing

Apache Kafka Architecture

Kafka provides distributed streaming with high throughput, fault tolerance, and horizontal scalability through partitioned topics and consumer groups.



Brokers

Distributed cluster nodes that store and serve topic partitions with automatic replication and failover



Topics & Partitions

Logical message categories split into ordered partitions for parallel processing and scalability



Consumer Groups

Coordinated consumers that automatically balance partition assignments for fault-tolerant processing





Swiftship Logistics

Case Study

E-Commerce Flash Sale Performance Crisis

Diagnosing and resolving critical bottlenecks when traffic spikes 50x during a limited-time promotion

Flash Sale Scenario: The Problem

❌ **Crisis Alert:** Your e-commerce platform crashes within 5 minutes of announcing a 90% off flash sale on popular items

Observed Symptoms

- Database connection pool exhaustion (max 100 connections hit immediately)
- Application servers showing 15-second response times
- CDN cache miss rate jumps to 85% for product pages
- Payment processing failures at 40% rate
- Users reporting infinite loading screens and timeouts

Traffic analytics show 50,000 concurrent users attempting to access 500 available items, with normal traffic being 1,000 concurrent users.

Bottleneck Analysis Framework

Database Layer Investigation

Profile slow queries, connection pool utilization, and lock contention during peak load periods

Network and CDN Analysis

Examine bandwidth utilization, cache hit ratios, and geographic distribution of failed requests

Application Performance Monitoring

Analyze response times, memory usage, CPU utilization, and garbage collection patterns across services

External Dependencies Review

Evaluate payment processor response times, third-party API limits, and downstream service capacity

Immediate Mitigation Solutions

Emergency Response (0-30 minutes)

- Enable aggressive caching for product pages (TTL: 1 minute)
- Implement connection pooling with circuit breakers
- Deploy read replicas for product catalog queries
- Activate rate limiting (100 requests/minute per user)

Short-term Fixes (30 minutes - 2 hours)

- Scale application servers horizontally (3x capacity)
- Pre-warm CDN with anticipated popular product pages
- Implement inventory checking via Redis cache
- Queue payment processing for async handling

Long-term Scalability Improvements

1

Database Optimization

Implement read replicas, query optimization, and connection pooling with proper sizing (500+ connections)

2

Caching Strategy

Multi-layer caching with CDN, Redis for sessions/cart, and database query caching with smart invalidation

3

Event-Driven Processing

Asynchronous order processing, inventory updates, and notification systems using Kafka message streams

4

Auto-scaling Infrastructure

Kubernetes HPA based on CPU/memory metrics and custom metrics like queue depth and response times



Designing for Failures and Resilience

Building Fault-Tolerant Distributed
Systems

Circuit Breaker Pattern Implementation



Circuit breakers prevent cascading failures by monitoring downstream service health and failing fast when errors exceed thresholds.

State Transitions

- **Closed:** Normal operation, requests pass through
- **Open:** Failure threshold exceeded, requests fail immediately
- **Half-Open:** Limited requests test service recovery

Configure failure thresholds (e.g., 50% error rate over 10 requests) and timeout periods (30-60 seconds) based on service characteristics.

Retry Strategies and Backoff Patterns

Exponential Backoff

Increases delay exponentially: 1s, 2s, 4s, 8s. Prevents thundering herd but may delay recovery.

```
delay = base_delay * (2 ^ attempt) + jitter
```

Linear Backoff

Fixed increment delays: 1s, 2s, 3s, 4s. Predictable timing but less efficient for temporary failures.

```
delay = base_delay * attempt + jitter
```

Fibonacci Backoff

Fibonacci sequence delays: 1s, 1s, 2s, 3s, 5s. Balances rapid initial retries with progressive backing off.

```
delay = fibonacci(attempt) + jitter
```


Rate Limiting Algorithms

Algorithm	Memory Usage	Accuracy	Use Case
Token Bucket	$O(1)$ per bucket	High	Burst handling with sustained rate
Leaky Bucket	$O(1)$ per bucket	High	Smooth output rate, queue buffering
Fixed Window	$O(1)$ per window	Medium	Simple implementation, periodic reset
Sliding Window	$O(\text{window size})$	Very High	Precise rate limiting, higher overhead

Token bucket is most common for API rate limiting, while sliding window provides the highest accuracy for critical services.

High Availability Design Principles



Eliminate Single Points of Failure

Redundant components, load balancers, and database replicas ensure no single component failure brings down the system



Health Checks and Monitoring

Continuous health monitoring with automatic failover triggers based on response times, error rates, and resource utilization



Graceful Degradation

Maintain core functionality when non-essential services fail by implementing feature toggles and fallback mechanisms

Multi-Region Architecture Patterns

Active-Passive

- Primary region handles all traffic
- Secondary region on standby
- RTO: 5-15 minutes
- Cost-effective for DR

Active-Active

- Traffic distributed across regions
- Both regions serve requests
- RTO: Near-zero failover
- Higher complexity and cost

Multi-Active

- 3+ regions with load balancing
- Geographic traffic routing
- Maximum availability
- Complex data consistency






Case Study

Checkout Resilience During Regional Outage

Maintaining payment processing and order completion when primary data center fails

Checkout Resilience Scenario

 **Incident:** Primary AWS us-east-1 region experiences complete outage during peak shopping hours, affecting payment processing and order management systems

Critical Systems at Risk

- Payment processing service (Stripe integration)
- Order management database (PostgreSQL primary)
- User session management (Redis cluster)
- Inventory tracking system
- Email notification service

Business impact: \$50,000 revenue per minute at risk, with 5,000 active shopping carts containing \$2.3M in potential orders.

Fault-Tolerant Checkout Design

1

Payment Processor Failover

Multiple payment providers (Stripe, PayPal, Square) with automatic routing based on availability and regional optimization

2

Database Cross-Region Replication

PostgreSQL read replicas in multiple regions with automatic promotion and connection string updates via service discovery

3

Stateless Session Management

JWT tokens with cart data to eliminate session store dependencies, encrypted and signed for security

Regional Failover Implementation

Automated Failover Triggers

- Health check failures >3 consecutive attempts
- Response time degradation >5 seconds
- Error rate exceeding 25% over 2 minutes
- Circuit breaker state changes to OPEN

DNS-Based Traffic Routing

Route 53 health checks with 30-second intervals automatically redirect traffic to healthy regions within 60-90 seconds.

```
# Terraform Route 53 Failover resource
"aws_route53_record" "primary" { zone_id = var.zone_id
name = "api.example.com" type = "A"
failover_routing_policy { type = "PRIMARY" }
health_check_id = aws_route53_health_check.primary.id
set_identifier = "primary-us-east-1" ttl = 60 records
= [aws_lb.primary.dns_name]}
```

Security and Compliance Basics

Building Secure and Compliant Systems



Identity and Access Management (IAM)



1

Authentication

Verifying user identity through credentials, multi-factor authentication, and OAuth/SAML protocols with proper session management

2

Authorization

Controlling resource access through role-based permissions, attribute-based policies, and principle of least privilege

3

Access Control

Implementing fine-grained permissions with regular access reviews, automated provisioning, and audit logging

Hands-On: Real-Time Collaborative Document System

Design Challenge

Build a Google Docs-like system supporting simultaneous editing, conflict resolution, and real-time synchronization across multiple clients.

10K

Concurrent Users

Per document editing session

50ms

Sync Latency

Maximum acceptable delay

99.9%

Availability

System uptime requirement

Key Technical Considerations

- Operational Transform vs. CRDT for conflict resolution
- WebSocket connection management and scaling
- Document versioning and history storage
- Real-time presence indicators and cursor tracking