

# Cloud Architecture Patterns: Scale, Resilience, Security

A comprehensive guide for Technical Program Managers, Solution Architects, and Cloud Engineers




# Agenda

**1** Horizontal Scaling & Data Management  
Partitioning, replication, and scaling strategies for modern systems

**2** Performance Optimization  
Caching strategies and event-driven architectures

**3** Resilience Patterns  
Circuit breakers, retries, rate limiting, and high availability

**4** Security & Compliance  
IAM, RBAC, secure design practices, and compliance frameworks

 This training provides both theoretical concepts and real-world applications. We'll examine case studies from companies like Netflix, Amazon, and Shopify to see these patterns in action.

# Horizontal Scaling, Partitioning & Replication

Building systems that scale without limits

# Horizontal Scaling Fundamentals

Adding more machines instead of upgrading existing ones

- Distributes load across multiple nodes
- Enables near-linear scaling (with proper design)
- Provides resiliency against node failures
- Supports geographic distribution

**i** **TPM Insight:** Coordinate scaling events with downstream dependencies to prevent cascading failures



# Data Partitioning Strategies

## Horizontal (Sharding)

Splitting data across multiple databases based on a partition key

- Amazon DynamoDB: Partition by user ID
- YouTube: Video shards by popularity/access patterns

## Vertical

Splitting different tables or columns across databases

- Shopify: Separating product data from customer data
- Netflix: Content metadata vs. user profiles

## Functional

Splitting by business domain or function

- Microservice architectures
- Domain-driven design boundaries

# Replication: Reading at Scale



## Key Benefits

- Distributes read load
- Improves read performance

## Challenges

- Replication lag (eventual consistency)
- Conflict resolution



# Case Study: YouTube Traffic Spikes

## Challenge

- Viral videos creating sudden traffic spikes
- 10-100x normal traffic in minutes
- Regional viewing patterns

## Solution Components

- CDN caching at edge locations
- Dynamic re-sharding of popular content
- Read replicas with regional distribution
- Request buffering during peak loads



**TPM Insight:** Implement automated scaling triggers with clear thresholds and monitoring

# Partitioning Trade-offs

Strategy	Advantages	Disadvantages
Hash-based	Even distribution, scalable	No range queries, repartitioning complex
Range-based	Efficient range queries	Potential hotspots, uneven distribution
Geography-based	Locality, compliance, performance	Complex replication, consistency challenges
Dynamic	Adapts to changing patterns	Complex implementation, monitoring overhead

**TPM Guidance:**

- Align partition strategies with access patterns and business requirements
- Implement partition observability to detect hotspots early
- Balance cost vs. performance gains when evaluating partition schemes



# Caching Strategies

Accelerating performance through strategic data storage

# Caching: Principles & Patterns

Store frequently accessed data in fast-access storage to reduce load and latency

## Client-Side Cache

Browser cache, mobile app local storage

## CDN Cache

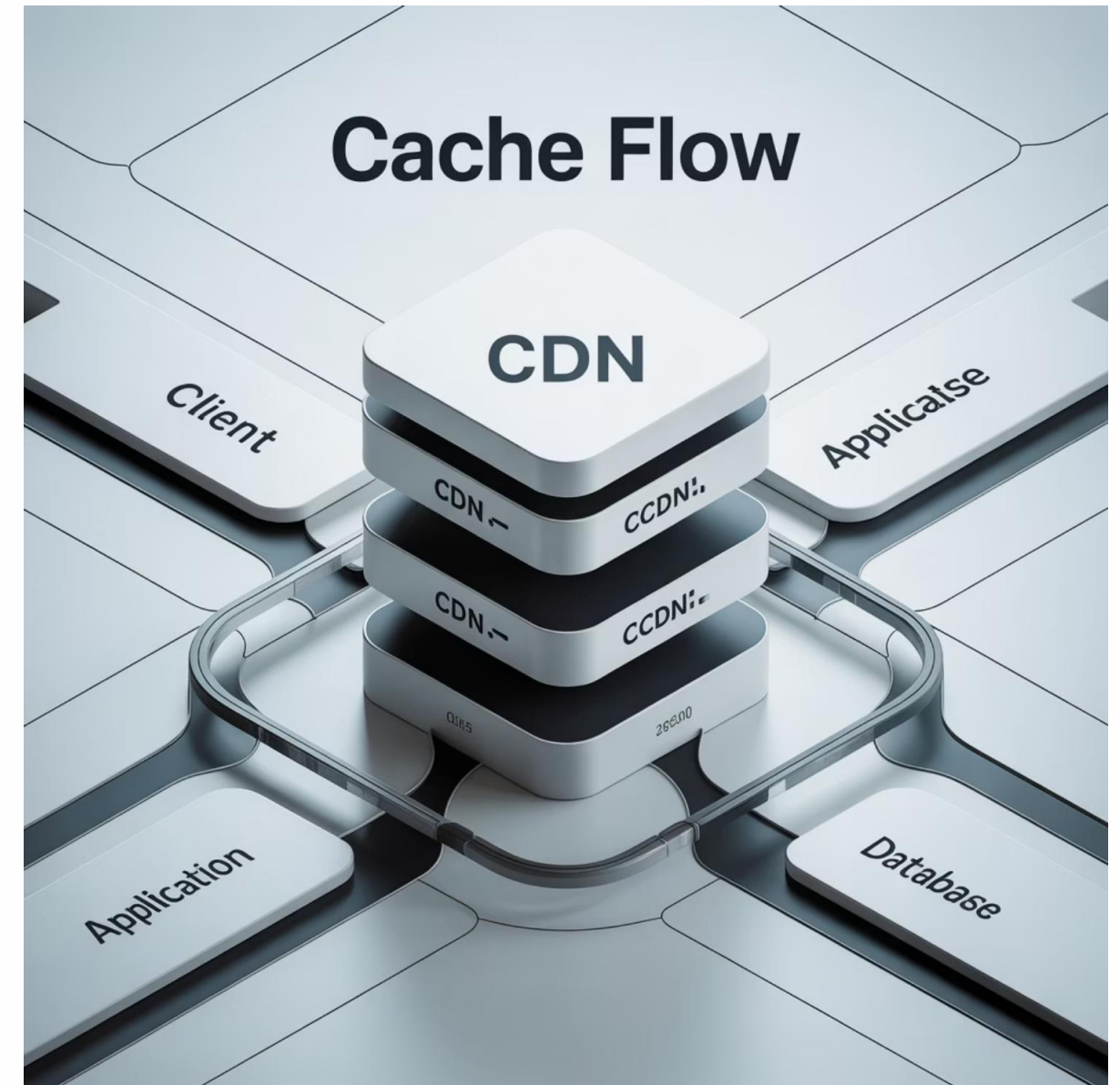
Akamai, Cloudflare, AWS CloudFront

## Application Cache

In-memory (Redis, Memcached)

## Database Cache

Query cache, materialized views



# CDN Caching in Action



Reduced origin



Lower latency



Cost savings

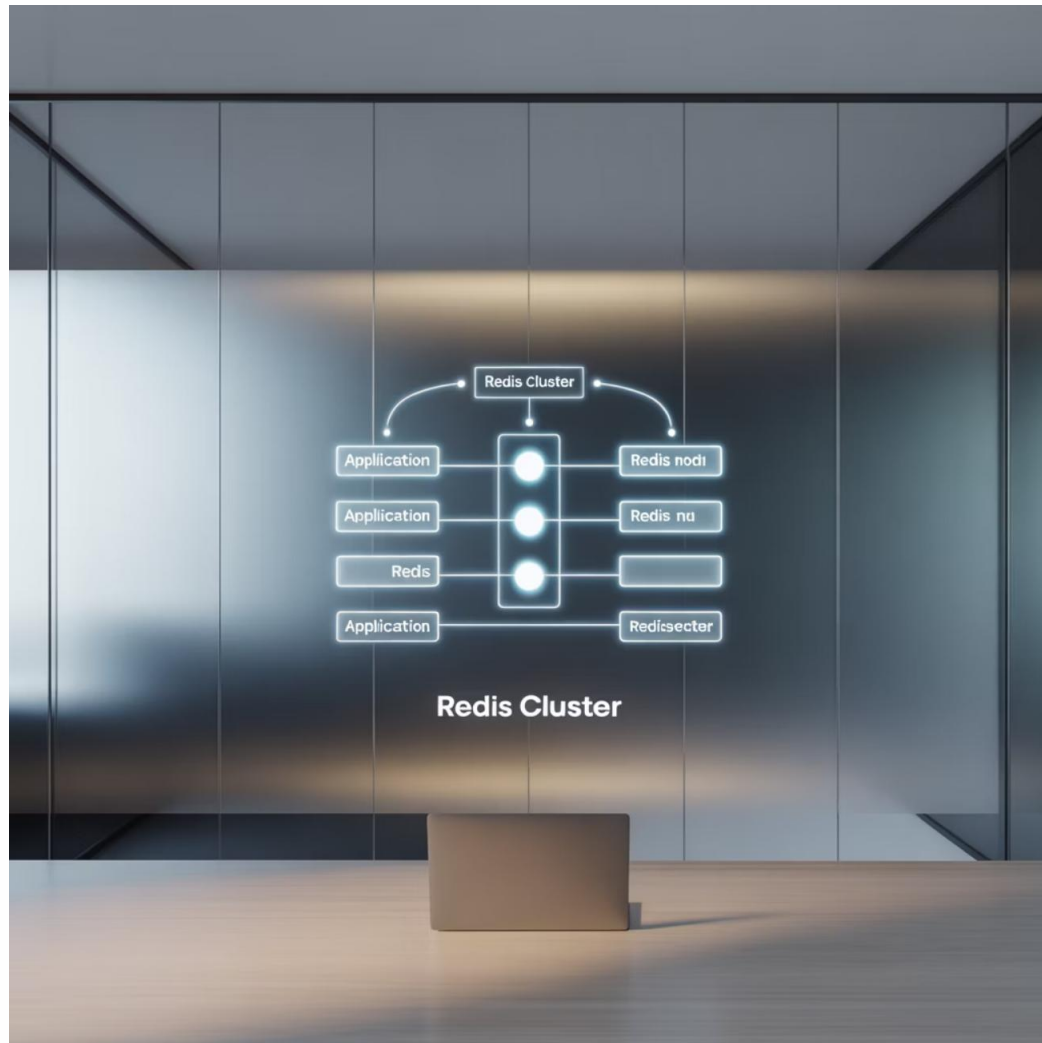


Improved conversion



DDoS protection

# In-Memory Caching with Redis



## Key Use Cases

- Session storage
- Real-time leaderboards
- Rate limiting
- Page rendering acceleration

## Cache Invalidation Strategies

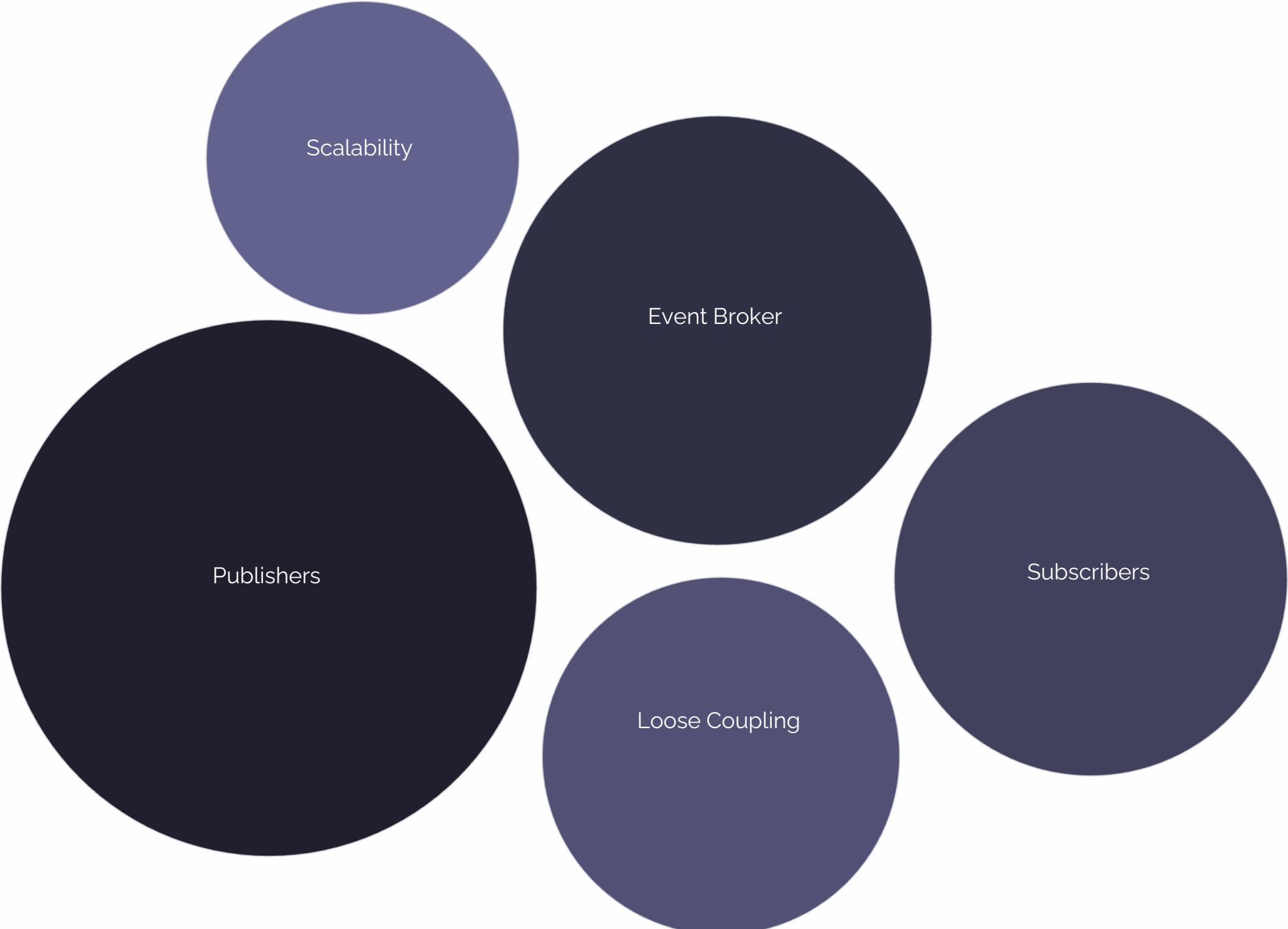
- Time-based (TTL)
- Write-through
- Write-behind
- Cache-aside (lazy loading)

📄 **TPM Discussion:** When is stale cache data acceptable? Consider business context and user expectations.

# Event-Driven Systems

Building resilient, decoupled architectures

# Event-Driven Architecture Fundamentals





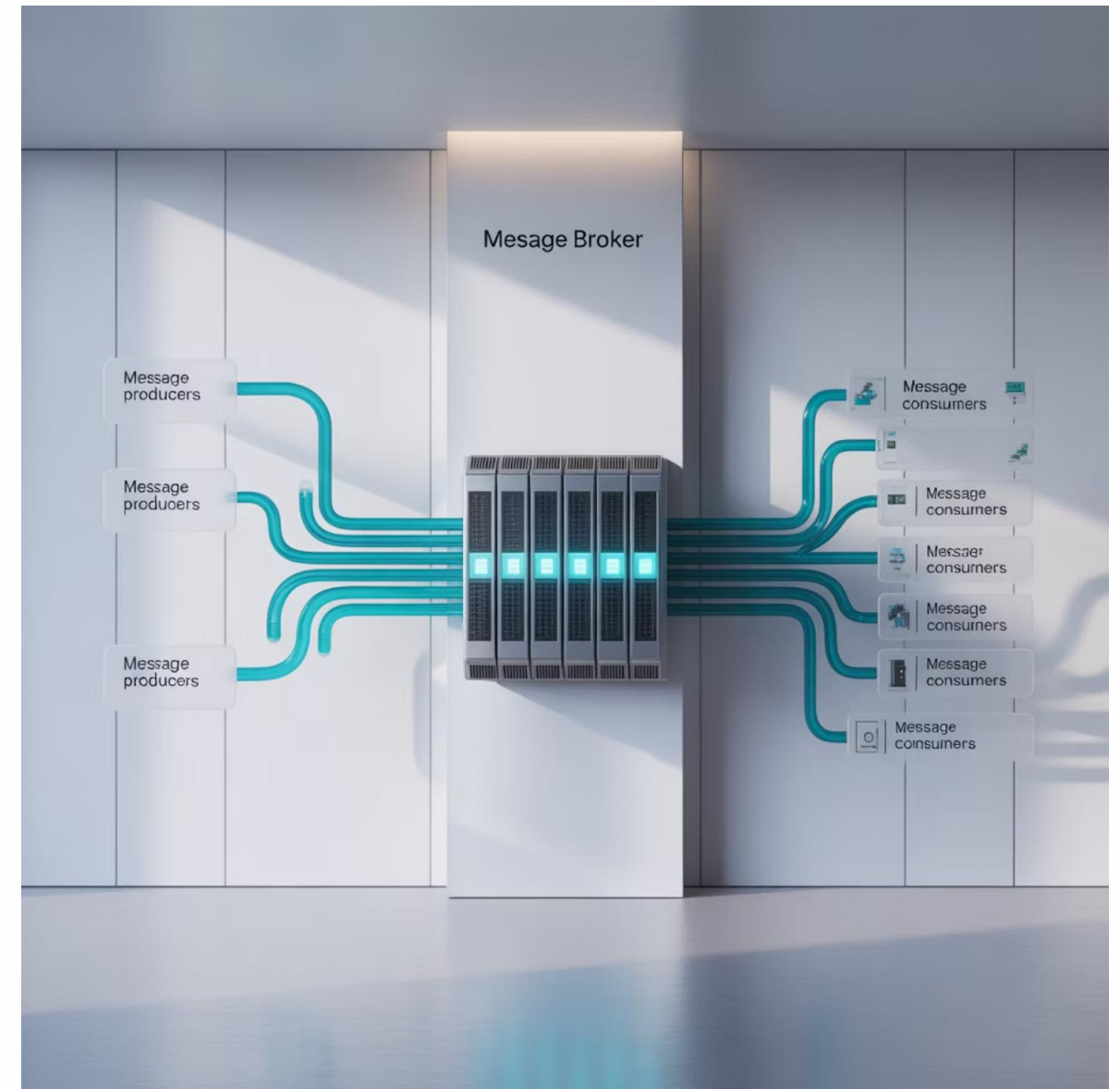
# Pub/Sub Implementation Patterns

Common implementation platforms:

- AWS SNS/SQS
- Google Cloud Pub/Sub
- Azure Event Grid/Service Bus
- RabbitMQ
- Apache Kafka

## Real-World Examples

- Zalando: Order processing pipeline
- Flipkart: Inventory updates
- Uber: Driver location updates



# Kafka for High-Scale Event Processing



## Producers

Write events to topics

- User service
- Order service
- Inventory service



## Kafka Cluster

Distributed event store

- Partitioned topics
- Replicated logs
- Retention policies



## Consumers

Process events

- Analytics
- Notifications
- Data pipeline

📄 **TPM Discussion:** How to handle consumer downtime without data loss? Consider consumer group offsets, dead-letter queues, and replay capabilities.

# Case Study

Retail Flash Sale: Scaling Under Pressure

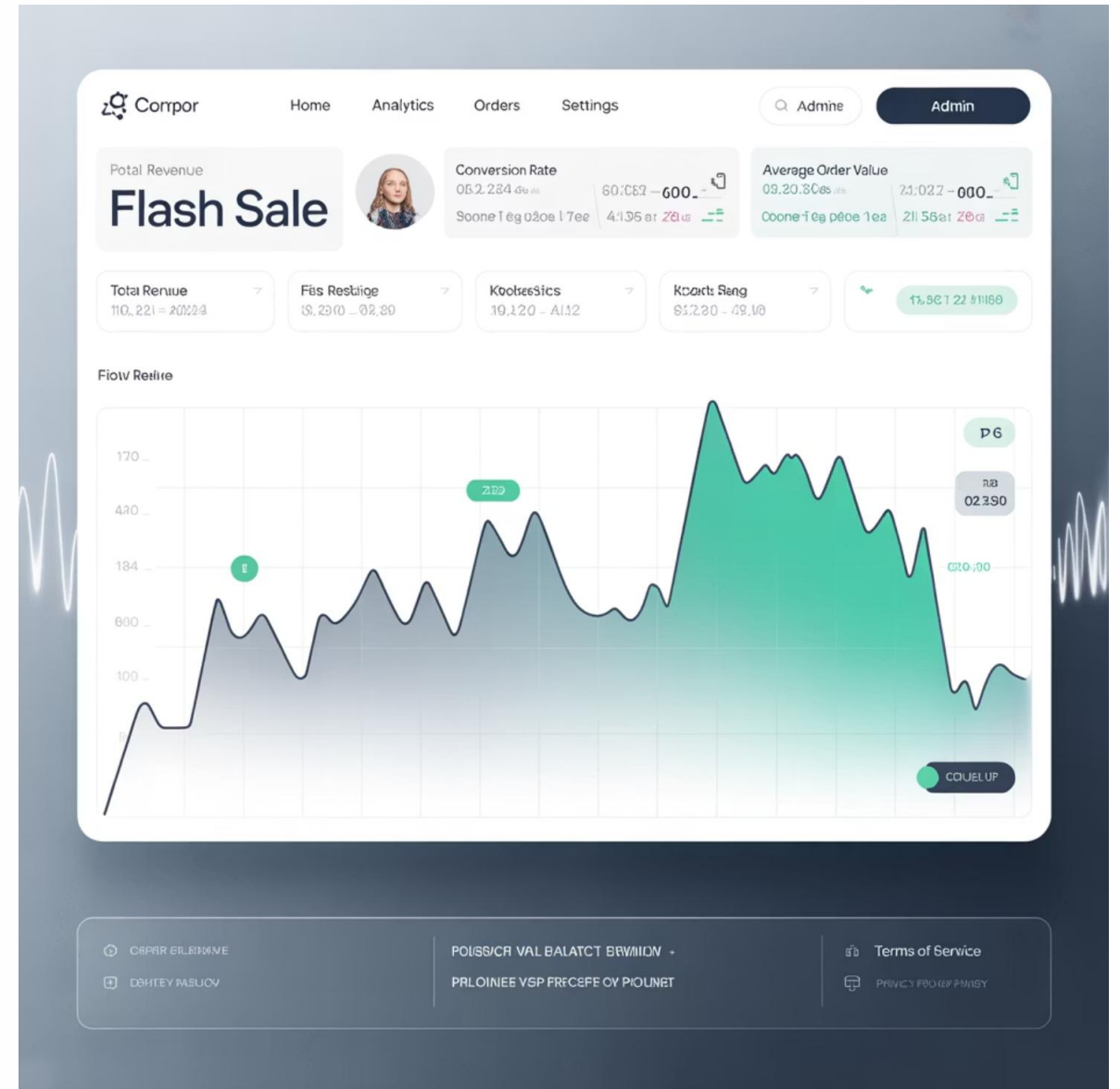
# Retail Flash Sale: The Challenge

## Scenario

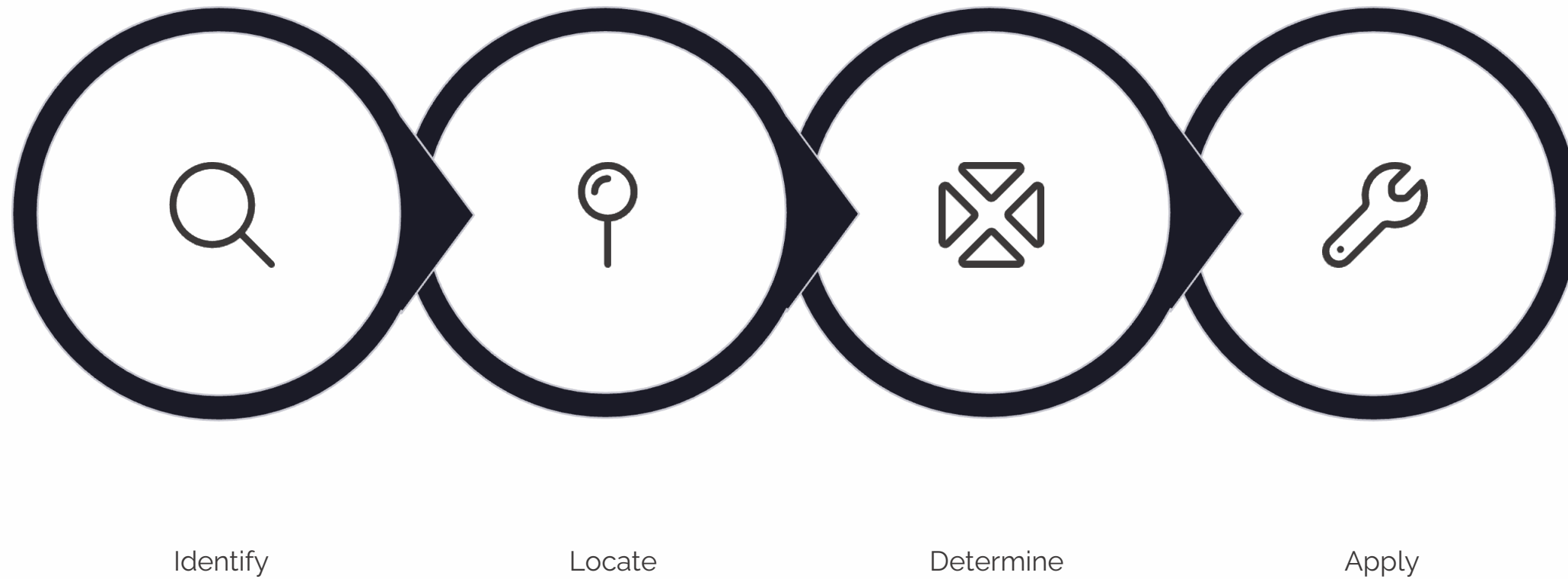
- E-commerce platform launching limited-time sale
- 50x normal traffic expected in first 30 minutes
- Limited inventory, high-demand products
- Previous sales resulted in site crashes and overselling

## Critical Functions

- Product catalog and search
- Inventory status
- Cart operations
- Checkout process
- Order confirmation



# Diagnosis Flow: Symptoms to Solutions



## Real-World Examples:

- Myntra End of Reason Sale: 15x traffic increase managed through partitioning and CDN caching
- Amazon Prime Day: Dynamic scaling across service tiers with circuit breakers
- Shopify Black Friday: Gradual rollout of flash sales across geographic regions

# Flash Sale: Architecture Solutions

## Product Catalog & Search

- Read-only replicas
- Aggressive CDN caching
- Search service scaling

## Inventory Management

- Distributed counters
- Optimistic locking
- Inventory service isolation

## Cart & Checkout

- Redis-backed carts
- Checkout queuing
- Asynchronous order processing

## Infrastructure

- Auto-scaling groups
- Rate limiting at API gateway
- Circuit breakers for non-critical services

📄 **TPM Discussion:** What service degradations are acceptable during peak load? How do we communicate these to stakeholders?



# Flash Sale: Operational Readiness

## Pre-Sale Testing

- Load testing with realistic traffic patterns
- Failover drills
- Chaos engineering experiments

## Coordination

- Cross-team war room
- Service-level dashboards
- Escalation paths defined

## Monitoring

- Real-time traffic visualization
- Custom flash sale metrics
- Anomaly detection

## TPM Coordination Points:

- Align marketing, inventory, and technical teams on timing and expectations
- Establish clear SLAs for each system component during the event
- Define acceptable degradation paths if systems approach limits

# Resilience Patterns

Circuit Breakers, Retries, Rate Limiting

# Circuit Breaker Pattern

Prevents cascade failures by temporarily "breaking the circuit" to failing services



Closed State

Normal operation: requests flow through



Open State

Circuit broken: requests fail fast without attempting downstream call



Half-Open State

Testing recovery: limited requests to test if system has recovered



# Circuit Breaker Implementation

## Key Parameters

- Failure threshold (e.g., 50% of 20 requests)
- Timeout duration (e.g., 5 seconds)
- Reset timeout (e.g., 30 seconds in open state)
- Fallback behavior

## Real-World Examples

- Netflix Hystrix (now Resilience4j)
- Microsoft Azure Circuit Breaker
- Istio service mesh resilience

## Implementation Best Practices

- Implement per service/endpoint
- Configure meaningful thresholds
- Provide graceful fallbacks
- Log state transitions
- Alert on circuit open events
- Test circuit behavior regularly

⊗ **TPM Risk:** Circuit breakers must be carefully tuned—too sensitive and services become unavailable unnecessarily; too lax and cascading failures occur.

# Retry Strategies



## Fixed Interval

Retry every X seconds

Simple but can amplify load spikes



## Exponential Backoff

Progressively longer delays

Prevents thundering herd



## Jitter

Random variation added to delay

Prevents synchronization



## Retry with Timeout

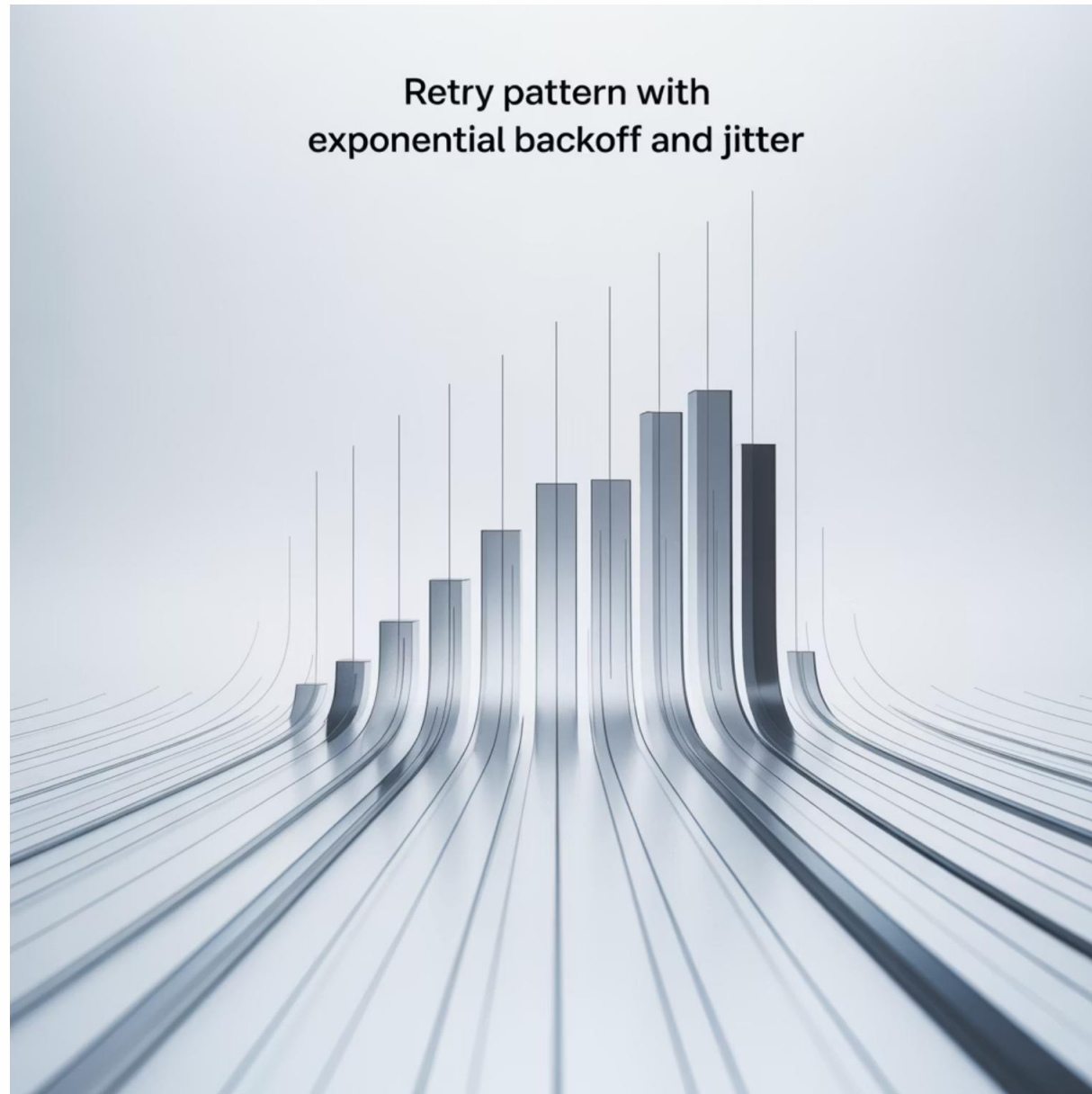
Maximum total retry duration

Prevents indefinite retries

### Implementation Examples:

- AWS SDK built-in retry mechanisms
- Spring Retry module
- Polly (.NET resilience library)

# Retry Pattern Implementation



## When to Use Retries

- Transient failures (network glitches)
- Resource contention (database locks)
- Service temporarily unavailable
- Rate limit exceeded

## When NOT to Retry

- Authentication failures
- Validation errors
- Resource not found
- Permission denied



# Rate Limiting Patterns



## Token Bucket

Tokens refill at fixed rate; requests consume tokens

Allows bursts up to bucket size

Used by AWS API Gateway, GitHub API



## Leaky Bucket

Requests processed at constant rate

Excess requests queued or rejected

Smoother traffic flow, no bursting



## Fixed Window

X requests per time window

Simple but edge effects at window boundaries

Common in simple API rate limiting



## User-Based

Different limits per user tier

Prioritizes premium customers

Used by Stripe, Twilio APIs

# Rate Limiting Implementation

## Implementation Locations

- API Gateway (AWS, Azure, Kong, Apigee)
- Load Balancer (Nginx, HAProxy)
- Application Code (Redis-backed)
- Service Mesh (Istio, Linkerd)

## Best Practices

- Return 429 (Too Many Requests) status code
- Include Retry-After header
- Expose limit information in headers
- Log and monitor rate limit events
- Implement graceful degradation



# High Availability & Multi-Region Design

Building systems that never go down

# High Availability Fundamentals

Eliminate single points of failure at every layer

## Infrastructure Redundancy

- Multiple AZs/data centers
- N+1 or N+2 capacity
- Network path redundancy

## Data Redundancy

- Multi-AZ databases
- Data replication
- Backup and recovery

## Application Redundancy

- Stateless design
- Load balancing
- Auto-scaling

