System Design Principles for TPMs and Engineers

A practical guide to architectural decisions, performance metrics, and real-world implementation strategies with retail industry examples

Agenda

1

System Design Basics

Architectural approaches, fundamental principles, and trade-offs in modern system design

2

Key Components

Essential building blocks that power scalable systems: load balancers, databases, caches, and more

3

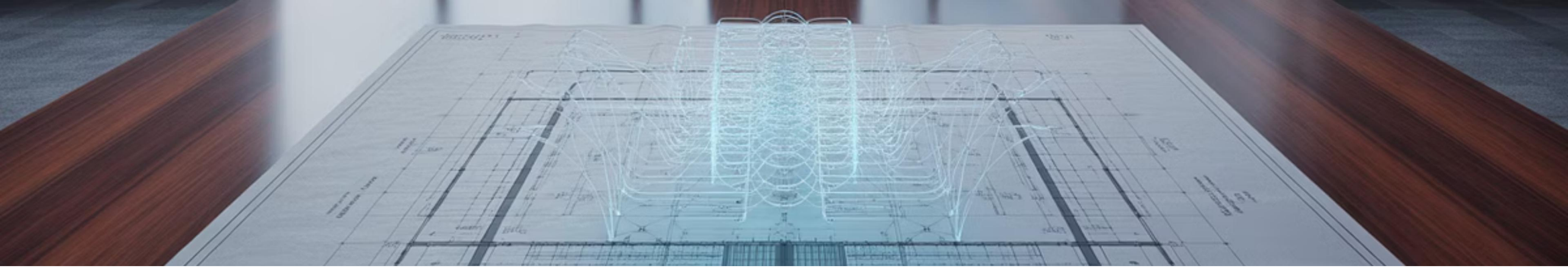
Metrics & SLAs

Critical performance indicators and service level agreements that TPMs must monitor and optimize

4

Workshop

Hands-on design exercise for an online file storage/sharing platform applying learned concepts



Part 1: System Design Basics

Understanding architectural paradigms, scalability principles, and reliability considerations

Monolith vs. Microservices

Monolithic Architecture

- Single codebase with all functionality
- Simpler development workflow
- Tightly coupled components
- Scaling requires replicating entire application
- Single point of failure risk

Microservices Architecture

- Distributed services with focused responsibilities
- Independent deployment capabilities
- Technology diversity across services
- Granular scaling of individual components
- Increased operational complexity

Retail Case Study: Evolution from Monolith to Microservices

1 Monolithic Beginning

Single codebase handling product catalog, user accounts, payments, and order processing

Challenges: Deployment bottlenecks, scaling inefficiencies during seasonal peaks

2 Initial Service Extraction

Payment processing extracted as first microservice due to compliance requirements

Benefits: Isolated security concerns, specialized scaling during checkout surges

3 Domain-Based Decomposition

Product catalog, inventory, and order management separated as distinct services

Benefits: Team autonomy, targeted performance optimizations

4 Fully Distributed Architecture

Complete transition with API gateway, service mesh, and event-driven communication

Benefits: 300% improvement in peak traffic handling, 99.99% availability

Fundamental Principles of System Design

Scalability

System's ability to handle growing workloads by adding resources

- Horizontal scaling: adding more machines
- Vertical scaling: adding more power to existing machines

Reliability

System's ability to perform its functions correctly under stated conditions

- Fault tolerance through redundancy
- Graceful degradation when components fail

Availability

Proportion of time a system is operational and accessible

- Measured in percentage (e.g., 99.9% = 8.76 hours downtime/year)
- Achieved through eliminating single points of failure

CAP Theorem and Its Impact

CAP Theorem States:

In a distributed system, you can only guarantee two of these three properties simultaneously:

- **Consistency:** All nodes see the same data at the same time
- **Availability:** Every request receives a response
- **Partition Tolerance:** System continues to operate despite network failures

Since network partitions are unavoidable in distributed systems, we must choose between consistency and availability when partitions occur.

CAP Theorem: Retail Applications

CP Systems (Consistency + Partition Tolerance)

Prioritize accurate data over constant availability

Retail Example: Payment processing systems that must maintain transaction integrity even if it means temporarily rejecting requests during network issues

AP Systems (Availability + Partition Tolerance)

Prioritize responsiveness over perfect data synchronization

Retail Example: Product catalog browsing that may show slightly outdated inventory information rather than failing to display products

CA Systems (Consistency + Availability)

Not realistic in distributed environments with network partitions

Retail Example: Single-node systems for non-critical functions like collecting customer feedback

Part 2: Key Components of System Design

Building blocks that power modern distributed systems and their critical interactions

Load Balancers

Core Functionality

- Distributes incoming traffic across multiple servers
- Prevents overloading of individual resources
- Performs health checks on backend services
- Provides SSL termination and connection persistence

Common Algorithms

- Round Robin: Sequential distribution
- Least Connections: Routes to least busy server
- IP Hash: Consistent routing based on client IP

Retail Application: During flash sales, load balancers direct traffic to geographically distributed checkout services based on current load and customer location, preventing regional bottlenecks

Databases: Choosing the Right Tool

Relational Databases

- Structured data with complex relationships
- ACID compliance for transactional integrity
- Examples: PostgreSQL, MySQL
- **Retail Use:** Order processing, inventory management

NoSQL Databases

- Unstructured or semi-structured data
- Horizontal scaling capabilities
- Examples: MongoDB, Cassandra, DynamoDB
- **Retail Use:** Product catalogs, customer profiles

In-Memory Databases

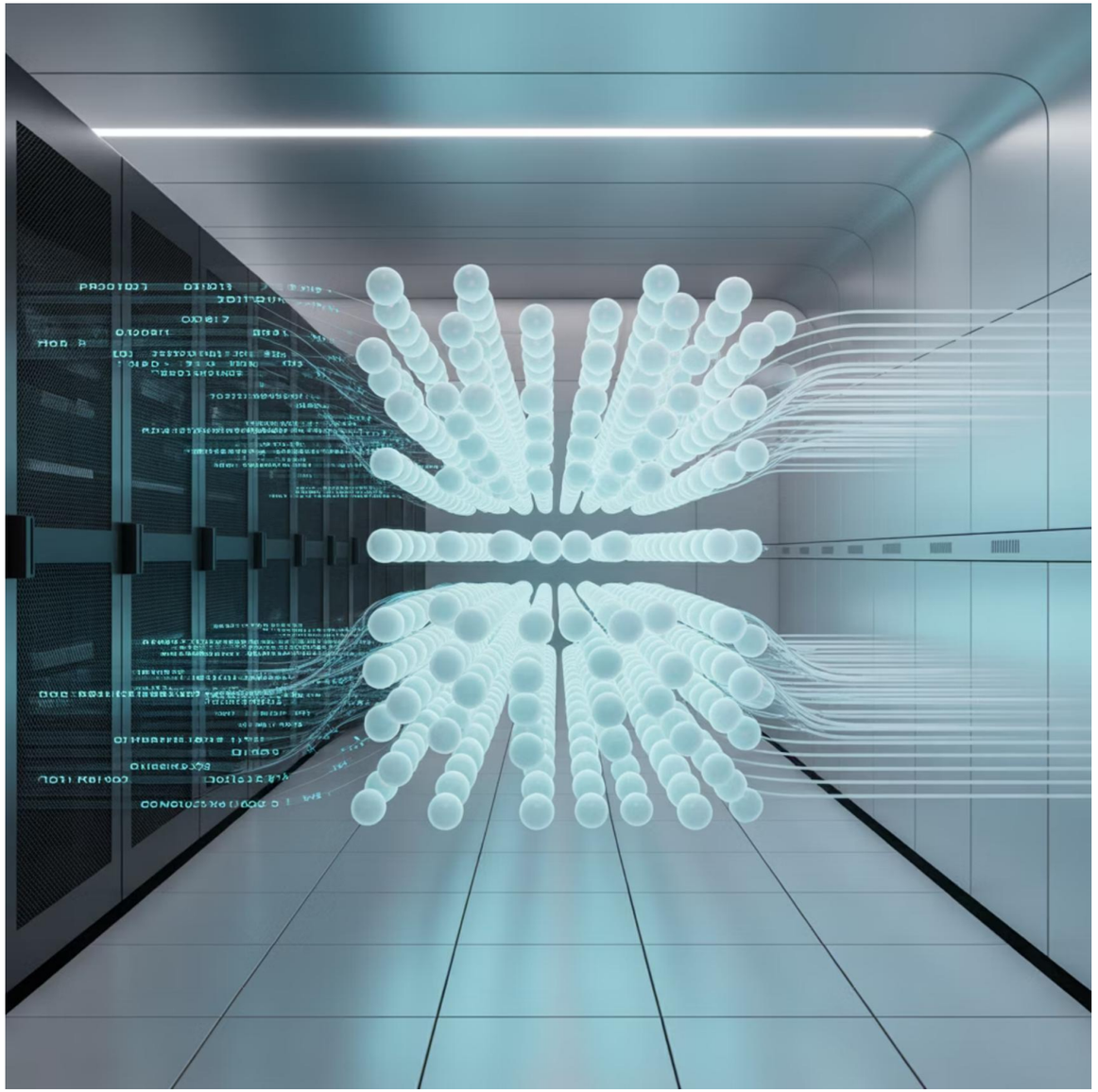
- Ultra-low latency requirements
- Volatile storage with persistence options
- Examples: Redis, Memcached
- **Retail Use:** Session management, real-time pricing

Queues: Decoupling and Buffering

Queue Benefits

- Asynchronous processing
- Load leveling during traffic spikes
- Resilience to downstream failures
- Ordered message delivery
- Event-driven architecture enablement

Retail Example: During Black Friday, order submission is decoupled from order processing. Customers receive immediate confirmation while backend processing scales independently to handle the queue.



Caches: Speed and Efficiency

Identify Hot Data

Analyze access patterns to determine frequently requested, rarely changing data (product details, pricing information)

Implement Caching Layer

Place in-memory storage between application and data sources (Redis, CDN for static assets)

Manage Cache Invalidation

Implement TTL or event-based invalidation strategies to maintain data freshness

Monitor Hit Rates

Track cache effectiveness and adjust caching policies based on performance metrics

Retail Impact: Product catalog caching can reduce database load by 70%+ during peak shopping periods while improving page load times by 300ms+.

Stateless vs. Stateful Services

Stateless Services

- Don't store client session data between requests
- Any instance can handle any request
- Horizontally scalable by adding instances
- Resilient to instance failures
- **Examples:** API gateways, content rendering

Stateful Services

- Maintain client context between interactions
- Sessions often pinned to specific instances
- Require shared state or sticky sessions
- More complex scaling and failover
- **Examples:** Databases, authentication services

Best Practice: Design services to be stateless when possible, externalizing state to dedicated stores like Redis or databases.

API Gateways and Service Mesh

API Gateway

Single entry point for all client requests that handles:

- Request routing to appropriate microservices
- Authentication and authorization
- Rate limiting and throttling
- Response aggregation from multiple services

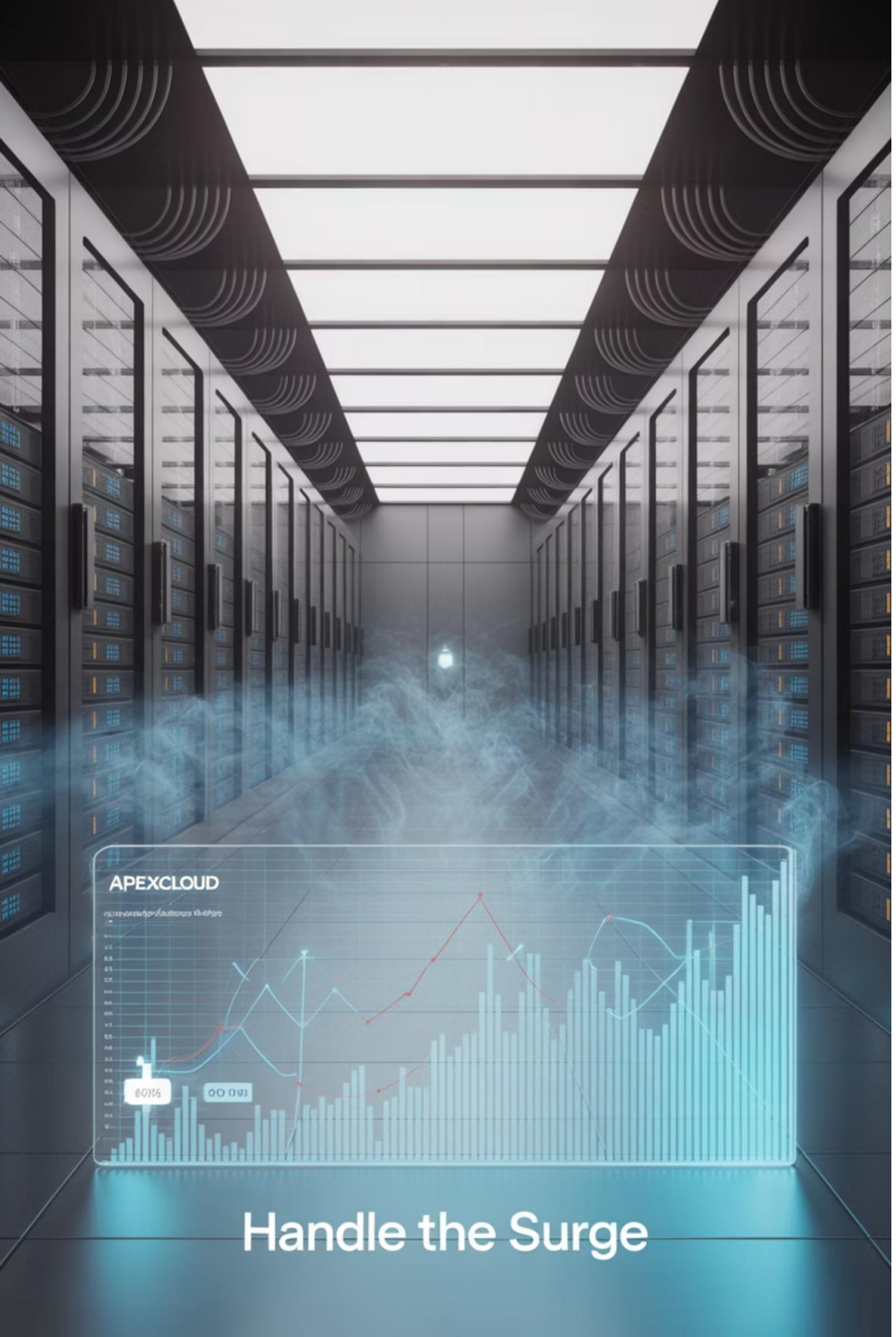
Retail Example: Mobile app requests flow through gateway that assembles product, pricing, and inventory data from separate services.

Service Mesh

Infrastructure layer for service-to-service communication:

- Traffic management between internal services
- Observability of inter-service communication
- Security through mutual TLS
- Resilience patterns (circuit breaking, retries)

Retail Example: Istio mesh enabling canary deployments of pricing algorithm updates during holiday season.



Retail Case Study: Flash Sale Architecture

1

Pre-Sale Preparation

Product catalog pre-cached in CDN and Redis

Inventory snapshot loaded into in-memory database

Auto-scaling groups configured with pre-warming

2

Traffic Surge Management

Global load balancers route based on geographic proximity

Rate limiting applied at API gateway to prevent abuse

Static content served entirely from edge locations

3

Order Processing

Checkout requests validated then placed in queue

Order processing services scale independently

Inventory locks implemented with distributed locking service

4

Graceful Degradation

Non-critical features disabled during peak load

Personalization reduced to increase throughput

Circuit breakers prevent cascading failures

Part 4: Workshop

Applying concepts to design an online file storage and sharing platform

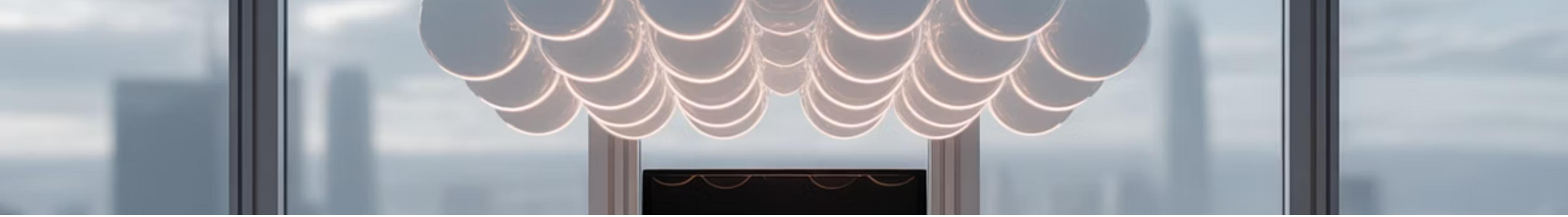
File Storage Platform: Requirements

Functional Requirements

- Upload, download, and share files up to 10GB
- Organize files in folders with metadata
- Control access permissions (view, edit, comment)
- Support file versioning and restoration
- Real-time collaboration on documents
- Search across files and content

Non-Functional Requirements

- Support 10 million daily active users
- 99.99% availability for read operations
- 99.9% availability for write operations
- Maximum file upload time of 1 second per MB
- Global access with low latency
- Compliance with data protection regulations



File Storage Platform: System Components

Frontend Components

- Web application (React)
- Mobile applications (iOS/Android)
- Desktop sync clients
- CDN for static assets

Backend Services

- Authentication & authorization
- File metadata service
- Storage orchestration service
- Sharing & permissions service
- Search & indexing service
- Notification service

Data Stores

- Object storage for file content
- Relational DB for metadata
- Cache for frequent access patterns
- Search index for content discovery

Infrastructure

- Global load balancers
- API gateway with rate limiting
- Message queues for async operations
- Multi-region deployment

File Storage Platform: Key Design Decisions

1 Separation of Content and Metadata

Store actual file content in object storage (S3, Google Cloud Storage)

Keep metadata (filename, permissions, version history) in relational database

Benefit: Optimized storage costs and query performance

2 Chunked File Transfer

Split large files into 5MB chunks for parallel upload/download

Implement resumable transfers for network interruptions

Benefit: Improved reliability and performance for large files

3 Multi-Region Architecture

Replicate metadata globally for low-latency access

Store file content in region nearest to primary user

Benefit: Balance between performance and storage costs

4 Tiered Storage Strategy

Hot storage for frequently accessed files

Cold storage for archived or rarely accessed content

Benefit: Cost optimization while maintaining performance

Key Takeaways

Architectural Foundations

Choose between monolithic and microservices architectures based on scale, team structure, and deployment needs

Component Selection

Select the right tools (databases, caches, queues) for specific requirements rather than following trends

Measurement Discipline

Define and track SLIs and SLOs that directly correlate with user experience and business outcomes

Balance Trade-offs

Recognize that system design involves trade-offs between consistency, availability, cost, performance, and development velocity

User-Centered Design

Always design with end-user experience as the primary consideration, making technical decisions that support business requirements

Part 3: Metrics and SLAs for TPMs

Measuring, monitoring, and maintaining system performance in production environments

Key Performance Metrics

Latency

Time required to complete a request

- Measured in milliseconds (ms)
- Usually tracked as percentiles (p50, p90, p99)
- Critical for user experience

Throughput

Number of requests processed per time unit

- Measured in requests per second (RPS)
- Indicates system capacity
- Often correlates with business volume

Error Rate

Proportion of failed requests

- Measured as percentage of total requests
- Categorized by error type (4xx vs 5xx)
- Direct indicator of system health

Saturation

How "full" your service is

- Resource utilization (CPU, memory, disk, network)
- Queue depths and processing backlogs
- Helps predict impending issues

Understanding Service Level Terminology

Service Level Agreement (SLA)

A formal contract between service provider and customer defining expected service quality

- Often includes financial penalties for violations
- Typically more conservative than internal targets
- Example: "99.9% monthly uptime or partial refund"

Service Level Objective (SLO)

Internal target for service performance and reliability

- More aggressive than customer-facing SLAs
- Provides buffer for unexpected issues
- Example: "99.95% API availability on 30-day rolling window"

Service Level Indicator (SLI)

Actual measured performance metric that tracks against SLOs

- Quantitative measure of service quality
- Collected and monitored continuously
- Example: "Percentage of successful HTTP requests"

Error Budget

Allowable unreliability derived from SLO

- If SLO is 99.9%, error budget is 0.1%
- Used to balance reliability vs. feature velocity
- When depleted, focus shifts to stability over features

Retail Case Study: Holiday Season Uptime

99.9%	99.95%	< 500ms	5K
Checkout Uptime SLA	Checkout Uptime SLO	Checkout Latency SLO	Orders/Minute
Maximum allowed downtime of 43.8 minutes/month	Internal target allowing just 21.9 minutes/month	For 95% of requests during peak periods	Minimum throughput SLO during flash sales

TPMs track these metrics in real-time dashboards and receive alerts when values approach thresholds. Cross-functional incident response teams are on standby during peak shopping periods.

What TPMs Should Monitor

Business Metrics

- Conversion rates
- Cart abandonment
- Revenue per minute
- Customer satisfaction scores

Technical Metrics

- Service availability
- Response times
- Error rates by type
- Infrastructure utilization

Operational Metrics

- Deployment frequency
- Change failure rate
- Time to detect incidents
- Time to recover service

Effective TPMs correlate technical metrics with business outcomes to prioritize improvements and communicate value to stakeholders.

Creating Effective SLIs and SLOs

Characteristics of Good SLIs

- Directly measure user experience
- Simple to understand and explain
- Consistently measurable over time
- Responsive to system changes
- Resistant to gaming or manipulation

Principles for Setting SLOs

- Base on customer expectations and business needs
- Focus on what matters most to users
- Set realistic targets based on historical performance
- Include buffer below SLA commitments
- Review and adjust quarterly based on learnings

Remember: It's better to have a few meaningful SLOs than many metrics that nobody acts on. Focus on the vital few rather than the trivial many.