

System Design Questions with Detailed Answers for Engineering Managers

Distributed Systems & Infrastructure

Q: Design a social media platform like Facebook that can handle 2 billion users

Functional Requirements:

- User registration and profile management
- Friend connections and social graph
- News feed with posts, photos, videos
- Real-time messaging and notifications
- Content sharing and interactions (likes, comments, shares)

Non-Functional Requirements:

- Support 2 billion users with 1 billion daily active users
- 99.99% availability
- Sub-200ms response time for feed loading
- Handle 100K posts per second
- Global distribution with low latency

High-Level Architecture:

```
[Load Balancer] → [API Gateway] → [Microservices]
                        ↓
[User Service] [Post Service] [Feed Service] [Notification Service]
                        ↓
[Database Cluster] [Cache Layer] [Message Queue] [CDN]
```

Detailed Design:

1. User Service:

- Handles authentication, profile management
- Uses sharded MySQL for user data
- Redis cache for session management
- Implements OAuth 2.0 for third-party integrations

2. Social Graph Service:

- Manages friend relationships and connections
- Uses graph database (Neo4j) for relationship queries
- Implements friend suggestion algorithms
- Caches popular connections in Redis

3. Post Service:

- Handles content creation and storage
- Uses MongoDB for post metadata
- Object storage (S3) for media files
- Elasticsearch for content search

4. Feed Generation Service:

- Implements both push and pull models
- Pre-computes feeds for active users (push)
- On-demand generation for less active users (pull)
- Uses machine learning for content ranking

5. Notification Service:

- Real-time notifications using WebSockets
- Push notifications for mobile devices
- Email notifications for important events
- Message queue (Kafka) for reliable delivery

Scalability Considerations:

- Horizontal sharding of databases by user ID
- CDN for global content distribution
- Auto-scaling based on traffic patterns
- Read replicas for database scaling
- Caching at multiple layers (browser, CDN, application, database)

Technology Stack:

- **Frontend:** React, React Native for mobile
- **Backend:** Java/Spring Boot, Node.js for real-time features
- **Databases:** MySQL (sharded), MongoDB, Neo4j, Redis
- **Message Queue:** Apache Kafka
- **Search:** Elasticsearch
- **Storage:** Amazon S3, CloudFront CDN
- **Infrastructure:** Kubernetes, Docker, AWS/GCP

Key Design Decisions:

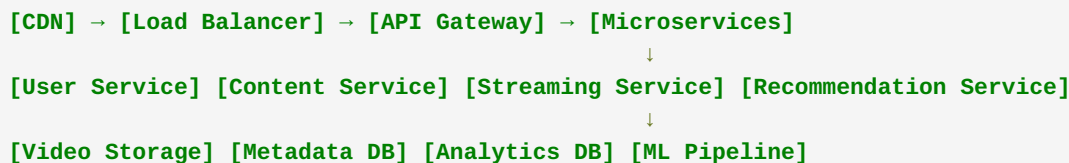
- Hybrid push/pull model for feed generation balances performance and resource usage
- Microservices architecture enables independent scaling and development
- Graph database optimizes social relationship queries
- Multi-layer caching reduces database load and improves response times

Q: Design a video streaming service like Netflix for global scale**Functional Requirements:**

- Video upload, encoding, and storage
- Content catalog and search
- User authentication and profiles
- Video streaming with adaptive bitrate
- Recommendation engine
- Offline viewing capabilities

Non-Functional Requirements:

- Support 200 million subscribers globally
- 99.9% availability
- Handle 1 million concurrent streams
- Support 4K video quality
- Global content delivery with <100ms startup time

High-Level Architecture:**Detailed Design:****1. Content Ingestion Pipeline:**

- Video upload service with chunked upload support
- Transcoding service using FFmpeg for multiple formats
- Quality control and content validation
- Metadata extraction and storage

2. Video Storage and CDN:

- Object storage (S3) for master video files
- Multiple CDN providers for global distribution
- Edge caching with TTL-based invalidation
- Adaptive bitrate streaming (HLS/DASH)

3. Streaming Service:

- Video player API with seek and quality controls
- Bandwidth detection and adaptive streaming
- DRM integration for content protection
- Analytics collection for viewing patterns

4. Recommendation Engine:

- Collaborative filtering algorithms
- Content-based recommendations
- Real-time ML model serving
- A/B testing framework for algorithm optimization

5. User Management:

- Multi-profile support per account
- Viewing history and watchlist
- Parental controls and content ratings
- Subscription and billing management

Scalability Solutions:

- Geographic content distribution via CDNs
- Database sharding by user regions
- Microservices with independent scaling
- Caching at multiple levels (CDN, application, database)
- Asynchronous processing for non-critical operations

Technology Stack:

- **Frontend:** React, iOS/Android native apps
- **Backend:** Java/Spring Boot, Python for ML
- **Databases:** PostgreSQL (sharded), Cassandra for analytics
- **Streaming:** HLS/DASH protocols, Wowza streaming engine

- **ML:** TensorFlow, Apache Spark for data processing
- **Infrastructure:** AWS/GCP, Kubernetes, Docker

Key Challenges and Solutions:

- **Global Latency:** Multi-region CDN deployment with edge caching
- **Peak Traffic:** Auto-scaling and load balancing across regions
- **Content Protection:** DRM integration and secure streaming protocols
- **Recommendation Accuracy:** Real-time ML pipeline with continuous learning

Data & AI/ML Systems

Q: Design a search engine like Google with ranking algorithms

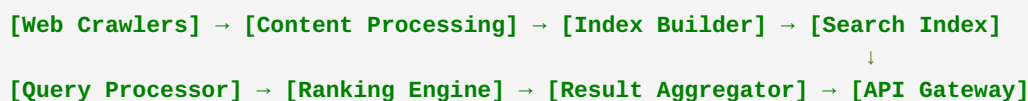
Functional Requirements:

- Web crawling and indexing
- Query processing and search
- Ranking algorithm implementation
- Auto-complete and spell correction
- Image and video search capabilities

Non-Functional Requirements:

- Index 50 billion web pages
- Handle 8 billion searches per day
- Sub-100ms query response time
- 99.99% availability
- Support multiple languages and regions

High-Level Architecture:



Detailed Design:

1. Web Crawling System:

- Distributed crawlers with politeness policies
- URL frontier management with priority queues
- Duplicate detection using bloom filters
- Robots.txt compliance and rate limiting
- Content freshness tracking and re-crawling

2. Content Processing Pipeline:

- HTML parsing and content extraction
- Language detection and text normalization
- Link extraction and anchor text processing
- Image and video metadata extraction
- Spam and low-quality content filtering

3. Indexing System:

- Inverted index construction with term frequencies

- Distributed index storage across multiple shards
- Index compression for storage efficiency
- Real-time index updates for fresh content
- Backup and recovery mechanisms

4. Query Processing:

- Query parsing and normalization
- Spell correction using edit distance algorithms
- Query expansion with synonyms and related terms
- Auto-complete using trie data structures
- Query intent classification

5. Ranking Algorithm:

- PageRank calculation for authority scoring
- TF-IDF scoring for relevance
- Machine learning models for ranking optimization
- Personalization based on user history
- Real-time signals (click-through rates, dwell time)

Scalability Architecture:

- Horizontal sharding of search index by terms
- Distributed query processing across multiple data centers
- Caching of popular queries and results
- Load balancing with geographic routing
- Asynchronous index updates

Technology Stack:

- **Crawling:** Python/Scrapy, distributed task queues
- **Processing:** Apache Spark, Hadoop for batch processing
- **Storage:** Distributed file systems (HDFS), NoSQL databases
- **Search:** Elasticsearch/Solr, custom inverted index
- **ML:** TensorFlow, scikit-learn for ranking models
- **Infrastructure:** Kubernetes, Docker, multi-cloud deployment

Key Design Decisions:

- Distributed architecture enables horizontal scaling
- Inverted index structure optimizes query performance
- Machine learning integration improves ranking quality
- Caching strategies reduce latency for popular queries
- Real-time updates balance freshness with system performance

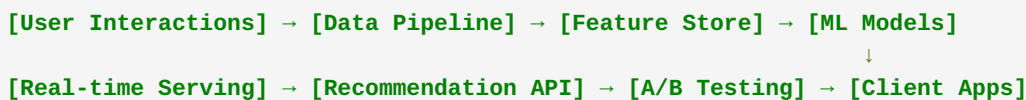
Q: Design a recommendation engine for e-commerce products

Functional Requirements:

- Product recommendations based on user behavior
- Real-time personalization
- Cold start handling for new users/products
- A/B testing for recommendation algorithms
- Recommendation explanations and transparency

Non-Functional Requirements:

- Support 100 million users and 10 million products
- Generate recommendations in <50ms
- Handle 1 million recommendation requests per second
- 99.9% availability
- Support for multiple recommendation types

High-Level Architecture:**Detailed Design:****1. Data Collection and Processing:**

- User interaction tracking (views, clicks, purchases, ratings)
- Product catalog and metadata management
- Real-time event streaming using Kafka
- Batch processing for historical data analysis
- Feature engineering pipeline

2. Recommendation Algorithms:

- **Collaborative Filtering:** User-based and item-based recommendations
- **Content-Based Filtering:** Product similarity using features
- **Matrix Factorization:** Latent factor models (SVD, NMF)
- **Deep Learning:** Neural collaborative filtering, autoencoders
- **Hybrid Approaches:** Combining multiple algorithms

3. Feature Store:

- User features (demographics, behavior patterns, preferences)
- Product features (category, price, ratings, descriptions)
- Contextual features (time, location, device, season)
- Real-time feature computation and serving
- Feature versioning and lineage tracking

4. Model Training and Serving:

- Offline model training with historical data
- Online learning for real-time adaptation
- Model versioning and A/B testing framework
- Real-time inference serving with low latency
- Model performance monitoring and alerting

5. Cold Start Solutions:

- Content-based recommendations for new products
- Popularity-based recommendations for new users
- Demographic-based initial recommendations
- Active learning to quickly gather user preferences
- Transfer learning from similar users/products

Scalability Solutions:

- Distributed model training using Apache Spark

- Real-time serving with Redis for fast lookups
- Horizontal scaling of recommendation services
- Caching of pre-computed recommendations
- Asynchronous model updates and deployment

Technology Stack:

- **Data Pipeline:** Apache Kafka, Apache Spark, Apache Airflow
- **Storage:** Cassandra for user data, Redis for caching
- **ML Framework:** TensorFlow, PyTorch, scikit-learn
- **Feature Store:** Feast, Tecton, or custom solution
- **Serving:** TensorFlow Serving, MLflow, custom APIs
- **Infrastructure:** Kubernetes, Docker, cloud platforms

Key Metrics and Evaluation:

- Click-through rate (CTR) and conversion rate
- Diversity and novelty of recommendations
- Coverage of product catalog
- User engagement and session duration
- Revenue impact and business metrics

A/B Testing Framework:

- Multi-armed bandit algorithms for exploration
- Statistical significance testing
- Gradual rollout and canary deployments
- Real-time monitoring of key metrics
- Automated rollback mechanisms

Real-time & Communication Systems

Q: Design a chat application like Slack with channels and direct messages

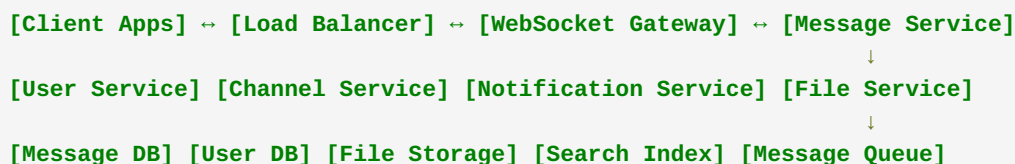
Functional Requirements:

- User authentication and workspace management
- Real-time messaging in channels and direct messages
- File sharing and media attachments
- Message search and history
- Notifications and presence indicators
- Integration with third-party services

Non-Functional Requirements:

- Support 10 million concurrent users
- Sub-100ms message delivery latency
- 99.99% availability
- Handle 1 million messages per second
- Support for mobile and web clients

High-Level Architecture:



Detailed Design:

1. Real-time Communication:

- WebSocket connections for real-time messaging
- Connection pooling and load balancing
- Heartbeat mechanism for connection health
- Fallback to HTTP polling for unreliable connections
- Message acknowledgment and delivery guarantees

2. Message Service:

- Message validation and sanitization
- Rate limiting to prevent spam
- Message persistence with timestamps
- Message threading and reply handling
- Emoji reactions and message formatting

3. Channel Management:

- Public and private channel creation
- Channel membership and permissions
- Channel archiving and deletion
- Channel discovery and search
- Integration with external services

4. User Presence and Status:

- Online/offline status tracking
- Typing indicators for active conversations
- Custom status messages and availability
- Last seen timestamps
- Do not disturb and notification preferences

5. File Sharing System:

- File upload with progress tracking
- Virus scanning and content validation
- Thumbnail generation for images
- File versioning and access control
- CDN distribution for fast downloads

Database Design:


```

-- Users table
CREATE TABLE users (
  id UUID PRIMARY KEY,
  username VARCHAR(50) UNIQUE,
  email VARCHAR(100) UNIQUE,
  workspace_id UUID,
  created_at TIMESTAMPTZ,
  last_active TIMESTAMPTZ
);

-- Channels table
CREATE TABLE channels (
  id UUID PRIMARY KEY,
  name VARCHAR(100),
  workspace_id UUID,
  is_private BOOLEAN,
  created_by UUID,
  created_at TIMESTAMPTZ
);

-- Messages table (partitioned by date)
CREATE TABLE messages (
  id UUID PRIMARY KEY,
  channel_id UUID,
  user_id UUID,
  content TEXT,
  message_type VARCHAR(20),
  created_at TIMESTAMPTZ,
  updated_at TIMESTAMPTZ
) PARTITION BY RANGE (created_at);

```

Scalability Solutions:

- Horizontal sharding of WebSocket connections
- Database partitioning by workspace or time
- Message queue for reliable delivery
- CDN for file distribution
- Caching of frequently accessed data

Technology Stack:

- **Frontend:** React, React Native, Electron
- **Backend:** Node.js, Go for WebSocket handling
- **Databases:** PostgreSQL (sharded), Redis for caching
- **Message Queue:** Apache Kafka, RabbitMQ
- **Storage:** Amazon S3, CloudFront CDN
- **Search:** Elasticsearch for message search
- **Infrastructure:** Kubernetes, Docker, AWS/GCP

Real-time Features Implementation:

- WebSocket connection management with sticky sessions
- Message broadcasting using pub/sub patterns
- Presence tracking with Redis sets
- Typing indicators with temporary state storage
- Push notifications for offline users

Security Considerations:

- End-to-end encryption for sensitive workspaces
- OAuth 2.0 and SSO integration
- Rate limiting and DDoS protection
- Content moderation and spam detection
- Audit logging for compliance requirements

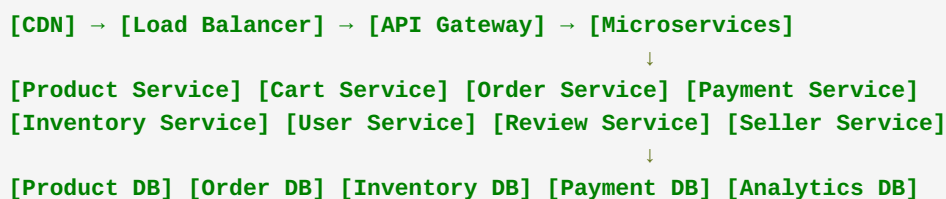
Product & Platform Systems

Q: Design an e-commerce platform like Amazon with inventory management**Functional Requirements:**

- Product catalog and search
- Shopping cart and checkout process
- Order management and fulfillment
- Inventory tracking and management
- Payment processing and fraud detection
- User reviews and ratings
- Seller marketplace functionality

Non-Functional Requirements:

- Support 300 million products and 100 million users
- Handle 10,000 orders per second during peak times
- 99.99% availability for critical services
- Sub-200ms page load times
- PCI DSS compliance for payments
- Global multi-region deployment

High-Level Architecture:**Detailed Design:****1. Product Catalog Service:**

- Product information management (PIM)
- Category hierarchy and attributes
- Product search with filters and facets
- Image and media management
- Price management and dynamic pricing

2. Inventory Management System:

- Real-time inventory tracking
- Multi-warehouse inventory allocation
- Reserved inventory for pending orders

- Automatic reorder point calculations
- Supplier integration for restocking

3. Shopping Cart Service:

- Session-based and persistent carts
- Cart abandonment tracking
- Price calculation with taxes and discounts
- Inventory validation before checkout
- Cart sharing and wishlist functionality

4. Order Management System:

- Order creation and validation
- Order status tracking and updates
- Fulfillment workflow orchestration
- Shipping integration and tracking
- Return and refund processing

5. Payment Processing:

- Multiple payment method support
- PCI DSS compliant payment handling
- Fraud detection and risk assessment
- Payment retry mechanisms
- Refund and chargeback management

Database Schema Design:

```

-- Products table
CREATE TABLE products (
  id UUID PRIMARY KEY,
  sku VARCHAR(50) UNIQUE,
  name VARCHAR(200),
  description TEXT,
  category_id UUID,
  price DECIMAL(10,2),
  seller_id UUID,
  created_at TIMESTAMP
);

-- Inventory table
CREATE TABLE inventory (
  product_id UUID PRIMARY KEY,
  warehouse_id UUID,
  available_quantity INTEGER,
  reserved_quantity INTEGER,
  reorder_point INTEGER,
  last_updated TIMESTAMP
);

-- Orders table
CREATE TABLE orders (
  id UUID PRIMARY KEY,
  user_id UUID,
  status VARCHAR(20),
  total_amount DECIMAL(10,2),
  shipping_address JSONB,
  created_at TIMESTAMP,
  updated_at TIMESTAMP
);

-- Order items table
CREATE TABLE order_items (
  id UUID PRIMARY KEY,
  order_id UUID,
  product_id UUID,
  quantity INTEGER,
  unit_price DECIMAL(10,2),
  total_price DECIMAL(10,2)
);

```

Inventory Management Strategy:

- **Real-time Updates:** Immediate inventory decrements on order placement
- **Reserved Inventory:** Temporary holds during checkout process
- **Distributed Inventory:** Multi-warehouse allocation algorithms
- **Predictive Restocking:** ML-based demand forecasting
- **Overselling Prevention:** Atomic inventory operations with locks

Scalability Solutions:

- Database sharding by product category or geographic region
- Read replicas for product catalog queries
- Caching layers for frequently accessed products
- Asynchronous order processing with message queues
- CDN for product images and static content

Technology Stack:

- **Frontend:** React, Next.js for server-side rendering
- **Backend:** Java/Spring Boot, Node.js for real-time features
- **Databases:** PostgreSQL (sharded), MongoDB for product catalog
- **Cache:** Redis for session and inventory data
- **Message Queue:** Apache Kafka for order processing
- **Search:** Elasticsearch for product search
- **Payment:** Stripe, PayPal integration
- **Infrastructure:** Kubernetes, Docker, multi-cloud deployment

Key Design Decisions:

- Microservices architecture enables independent scaling and development
 - Event-driven architecture ensures data consistency across services
 - Separate read and write paths optimize for different access patterns
 - Caching strategies reduce database load and improve response times
 - Asynchronous processing handles peak traffic without blocking user experience
-

System Design Fundamentals

Q: How would you design a system to handle 10x traffic growth?

Current System Assessment:

- Identify current bottlenecks and performance metrics
- Analyze traffic patterns and peak usage times
- Review database query performance and slow queries
- Assess infrastructure capacity and resource utilization
- Evaluate application architecture and scalability limitations

Scalability Strategy:**1. Horizontal Scaling:**

- **Application Servers:** Add more instances behind load balancers
- **Database Scaling:** Implement read replicas and database sharding
- **Microservices:** Break monolith into independently scalable services
- **Auto-scaling:** Implement dynamic scaling based on metrics
- **Load Distribution:** Use multiple load balancers and geographic distribution

2. Caching Strategy:

- **Application Cache:** Redis/Memcached for frequently accessed data
- **Database Query Cache:** Cache expensive query results
- **CDN:** Global content delivery for static assets
- **Browser Cache:** Optimize client-side caching headers
- **API Response Cache:** Cache API responses with appropriate TTL

3. Database Optimization:

- **Read Replicas:** Distribute read traffic across multiple replicas
- **Database Sharding:** Partition data across multiple database instances
- **Query Optimization:** Optimize slow queries and add proper indexes
- **Connection Pooling:** Efficient database connection management
- **NoSQL Integration:** Use NoSQL for specific use cases (caching, analytics)

4. Infrastructure Improvements:

- **Cloud Auto-scaling:** Leverage cloud provider auto-scaling features
- **Container Orchestration:** Use Kubernetes for efficient resource management
- **Multi-region Deployment:** Deploy across multiple geographic regions
- **Performance Monitoring:** Implement comprehensive monitoring and alerting
- **Capacity Planning:** Proactive capacity planning based on growth projections

Implementation Phases:

Phase 1: Quick Wins (1-2 months)

- Implement application-level caching
- Add database read replicas
- Optimize critical database queries
- Set up CDN for static content
- Implement basic auto-scaling

Phase 2: Architecture Changes (3-6 months)

- Break monolith into microservices
- Implement database sharding strategy
- Set up message queues for asynchronous processing
- Implement comprehensive monitoring
- Optimize API performance

Phase 3: Advanced Scaling (6-12 months)

- Multi-region deployment
- Advanced caching strategies
- Machine learning for predictive scaling
- Performance optimization based on real data
- Disaster recovery and high availability

Monitoring and Metrics:

- **Application Metrics:** Response time, throughput, error rates
- **Infrastructure Metrics:** CPU, memory, disk, network utilization
- **Database Metrics:** Query performance, connection counts, replication lag
- **Business Metrics:** User engagement, conversion rates, revenue impact
- **Alerting:** Proactive alerts for performance degradation

Technology Recommendations:

- **Load Balancers:** NGINX, HAProxy, cloud load balancers
- **Caching:** Redis, Memcached, Varnish
- **Databases:** PostgreSQL with read replicas, MongoDB for specific use cases
- **Message Queues:** Apache Kafka, RabbitMQ, cloud messaging services
- **Monitoring:** Prometheus, Grafana, ELK stack, cloud monitoring tools
- **Infrastructure:** Kubernetes, Docker, cloud platforms (AWS, GCP, Azure)

Cost Optimization:

- **Resource Right-sizing:** Match resources to actual usage patterns
 - **Reserved Instances:** Use cloud reserved instances for predictable workloads
 - **Spot Instances:** Leverage spot instances for non-critical workloads
 - **Auto-scaling Policies:** Implement intelligent scaling to avoid over-provisioning
 - **Performance Optimization:** Optimize code and queries to reduce resource usage
-

Q: Design a system with 99.99% uptime requirements

Uptime Calculation:

- 99.99% uptime = 52.56 minutes of downtime per year
- 4.38 minutes of downtime per month
- 8.64 seconds of downtime per day

High Availability Architecture:

1. Redundancy at Every Layer:

- **Multiple Data Centers:** Deploy across at least 3 availability zones
- **Load Balancer Redundancy:** Multiple load balancers with health checks
- **Application Server Redundancy:** N+1 redundancy for all services
- **Database Redundancy:** Master-slave replication with automatic failover
- **Network Redundancy:** Multiple network paths and ISP connections

2. Fault Tolerance Design:

- **Circuit Breakers:** Prevent cascade failures between services
- **Bulkhead Pattern:** Isolate critical resources from non-critical ones
- **Graceful Degradation:** Maintain core functionality during partial failures
- **Timeout and Retry Logic:** Handle transient failures gracefully
- **Health Checks:** Comprehensive health monitoring at all levels

3. Database High Availability:

- **Master-Slave Replication:** Automatic failover to slave in case of master failure
- **Database Clustering:** Multi-master setup for write scalability
- **Backup and Recovery:** Regular backups with point-in-time recovery
- **Data Replication:** Synchronous replication for critical data
- **Connection Pooling:** Efficient connection management with failover

Disaster Recovery Strategy:

1. Backup Strategy:

- **Automated Backups:** Daily full backups and continuous incremental backups
- **Cross-Region Replication:** Replicate data to geographically distant regions
- **Backup Testing:** Regular testing of backup restoration procedures
- **Recovery Time Objective (RTO):** Target recovery time of <15 minutes
- **Recovery Point Objective (RPO):** Maximum data loss of <5 minutes

2. Failover Mechanisms:

- **Automatic Failover:** Automated detection and failover for database and services
- **DNS Failover:** Automatic DNS updates to redirect traffic
- **Application Failover:** Stateless applications with session replication
- **Manual Failover:** Well-documented procedures for manual intervention
- **Failback Procedures:** Safe procedures to return to primary systems

Monitoring and Alerting:

1. Comprehensive Monitoring:

- **Infrastructure Monitoring:** CPU, memory, disk, network metrics
- **Application Monitoring:** Response times, error rates, throughput
- **Database Monitoring:** Query performance, replication lag, connection counts
- **Business Metrics:** User experience, transaction success rates
- **External Monitoring:** Third-party monitoring for external perspective

2. Alerting Strategy:

- **Tiered Alerting:** Different alert levels based on severity
- **On-call Rotation:** 24/7 on-call coverage with escalation procedures
- **Alert Fatigue Prevention:** Intelligent alerting to reduce false positives
- **Incident Response:** Automated incident creation and tracking
- **Post-incident Reviews:** Regular reviews to improve system reliability

Deployment and Change Management:

1. Safe Deployment Practices:

- **Blue-Green Deployments:** Zero-downtime deployments with instant rollback
- **Canary Releases:** Gradual rollout to detect issues early
- **Feature Flags:** Control feature rollout without code deployments
- **Automated Testing:** Comprehensive testing before production deployment
- **Rollback Procedures:** Quick rollback mechanisms for failed deployments

2. Change Management:

- **Change Review Process:** Peer review for all production changes
- **Maintenance Windows:** Scheduled maintenance during low-traffic periods
- **Change Documentation:** Detailed documentation of all changes
- **Risk Assessment:** Evaluate risk for all production changes
- **Communication:** Clear communication of planned changes to stakeholders

Technology Stack for High Availability:

- **Load Balancers:** HAProxy, NGINX with health checks
- **Application Servers:** Kubernetes with pod auto-restart and rolling updates
- **Databases:** PostgreSQL with streaming replication, Redis Sentinel
- **Message Queues:** Apache Kafka with replication, RabbitMQ clustering
- **Monitoring:** Prometheus, Grafana, PagerDuty for alerting
- **Infrastructure:** Multi-cloud deployment, Terraform for infrastructure as code

SLA and Metrics:

- **Availability SLA:** 99.99% uptime with clear measurement methodology
- **Performance SLA:** Response time and throughput guarantees
- **Error Budget:** Allowable error rate based on availability target
- **Incident Response SLA:** Time to acknowledge and resolve incidents
- **Regular Reporting:** Monthly availability reports and trend analysis

Cost Considerations:

- **Redundancy Costs:** Balance redundancy with cost efficiency
- **Reserved Capacity:** Use reserved instances for predictable workloads
- **Auto-scaling:** Scale down during low-traffic periods
- **Monitoring Costs:** Optimize monitoring to avoid excessive costs
- **ROI Analysis:** Calculate return on investment for availability improvements

This comprehensive system design guide provides detailed answers for engineering manager interviews, covering distributed systems, data platforms, real-time systems, and fundamental scalability concepts. Each answer includes practical implementation details, technology recommendations, and real-world considerations that demonstrate both technical depth and leadership perspective.