Non-Functional Requirements Guide for System Design Interviews

Latency, Throughput, Availability, Scalability, and More

Table of Contents

Introduction

Non-Functional Requirements (NFRs) define HOW a system should behave, as opposed to functional requirements which define WHAT a system should do.

Why NFRs Matter

NFRs are critical in system design interviews because they:

- **Drive Architecture Decisions**: Availability needs affect redundancy
- **Define Success Criteria**: What does "fast" mean?
- **Create Trade-offs**: Performance vs cost, consistency vs availability
- **Show Experience**: Production-minded thinking

Functional vs Non-Functional

```
Functional Requirement (WHAT):
"Users should be able to send messages to each other"

Non-Functional Requirements (HOW):
- Messages delivered in < 100ms (Latency)
- System handles 10K messages/sec (Throughput)
- System available 99.99% of time (Availability)
```

```
- Scales to 1M concurrent users (Scalability)
- Messages never lost (Reliability)
- End-to-end encrypted (Security)
```

Main Categories of NFRs

```
1. PERFORMANCE
   - Latency
   - Throughput
   - Response Time

2. SCALABILITY
   - Horizontal scaling
   - Vertical scaling
   - Elasticity

3. RELIABILITY
   - Availability
   - Fault Tolerance
   - Durability

4. CONSISTENCY
   - Strong consistency
   - Eventual consistency
   - Causal consistency

5. SECURITY
   - Authentication
   - Authorization
   - Encryption

6. OPERABILITY
   - Maintainability
   - Observability
   - Deployability

7. COST
   - Infrastructure costs
   - Operational costs
   - Development costs
```

Performance Requirements

1. Latency

Definition: Time between request initiation and response receipt

Types of Latency

```
Network Latency:
- Same datacenter: 1-2ms
- Cross-datacenter (same region): 5-10ms
- Cross-region (US East to West): 60-80ms
- Cross-continent (US to Europe): 100-150ms
- Cross-continent (US to Asia): 150-300ms

Application Latency:
- Cache hit (Redis): <1ms
- Cache hit (Memcached): 1-5ms
- Database query (indexed): 5-10ms
- Database query (unindexed): 100-1000ms
- API call (internal): 10-50ms
- API call (external): 100-500ms

End-to-End Latency Components:
Total = Network + Processing + Queue + Database

Example (Web Request):
- DNS lookup: 20ms
- TCP handshake: 50ms
- TLS handshake: 100ms
- Request send: 10ms
- Server processing: 50ms
- Database query: 10ms
- Response send: 10ms
Total: 250ms (first request)
Subsequent: 70ms (connection reused)
```

Latency Requirements by Use Case

Use Case	Acceptable Latency	Excellent Latency
Real-time Chat	< 1s	< 100ms
Social Media Feed	< 2s	< 500ms
Search Results	< 1s	< 200ms
E-commerce Checkout	< 3s	< 1s
Video Streaming	< 5s start	< 2s start
Gaming (action)	< 100ms	< 50ms
Trading Platform	< 50ms	< 10ms

```
| File Upload          | < 10s          | < 5s             |
|----------------------|----------------|------------------|
| Dashboard Load       | < 3s           | < 1s             |
|----------------------|----------------|------------------|
| Email Delivery       | < 5min         | < 1min           |
```

Optimizing Latency

```
Strategies:

1. Caching
   - In-memory cache (Redis): <1ms
   - CDN edge cache: 10-50ms
   - Browser cache: 0ms (no request)

2. Geographic Distribution
   - Multiple regions: Reduces network latency
   - CDN edge locations: Serve from nearest
   - Database replicas: Read from nearest

3. Connection Optimization
   - Keep-alive connections
   - HTTP/2 multiplexing
   - Connection pooling
   - Reduce handshakes

4. Database Optimization
   - Indexes on query columns
   - Query optimization
   - Read replicas
   - Database caching

5. Async Processing
   - Return immediately
   - Process in background
   - Notify when complete

Example:
Requirement: Page load < 1s

Without Optimization:
- Server: 500ms
- Database: 300ms
- Rendering: 200ms
Total: 1000ms (at limit!)

With Optimization:
- Cache hit: 5ms (instead of 300ms DB)
- CDN for static: 20ms (instead of 100ms)
```

```
- Async non-critical: 0ms wait
Total: 225ms (4x better!)
```

2. Throughput

Definition: Amount of work performed per unit time (requests/sec, transactions/sec, MB/sec)

Throughput Benchmarks

```
Database Systems:
- MySQL: 1K-10K queries/sec (single instance)
- PostgreSQL: 5K-15K queries/sec
- MongoDB: 10K-50K operations/sec
- Cassandra: 100K-1M writes/sec (cluster)
- Redis: 100K-1M operations/sec

Message Queues:
- RabbitMQ: 20K-50K msgs/sec (single node)
- Kafka: 1M+ msgs/sec (cluster)
- AWS SQS: 300K msgs/sec (standard)

Web Servers:
- Node.js: 5K-10K req/sec (single core)
- Nginx: 50K-100K req/sec (reverse proxy)
- Java/Spring: 10K-20K req/sec

CDN:
- Cloudflare: Millions of req/sec (global)
- CloudFront: Millions of req/sec
```

Calculating Required Throughput

```
Example: Social Media Platform

Daily Active Users (DAU): 10M
Average actions per user: 50/day
Peak factor: 3x average

Average RPS:
= (10M * 50) / (24 * 3600)
= 500M / 86400
= 5,787 requests/sec

Peak RPS:
= 5,787 * 3
= 17,361 requests/sec
```

```
Round up for safety: 20,000 RPS

Server Capacity Planning:
- Each server: 1000 RPS
- Servers needed: 20,000 / 1000 = 20 servers
- Add 50% buffer: 30 servers
- For 99.99% availability: Deploy across 3 AZs
- Per AZ: 10 servers

Database Capacity:
- 60% reads, 40% writes
- Reads: 12,000 RPS
- Writes: 8,000 RPS
- Read replicas: 12,000 / 5,000 = 3 replicas
- Primary for writes: 8,000 RPS (within limits)
```

Optimizing Throughput

```
Strategies:

1. Horizontal Scaling
   - Add more servers
   - Distribute load
   - Load balancer

2. Caching
   - Reduce database load
   - Increase cache hit ratio
   - Tiered caching (L1, L2, L3)

3. Async Processing
   - Queue heavy operations
   - Process in background
   - Increase request capacity

4. Connection Pooling
   - Reuse connections
   - Reduce overhead
   - Optimize pool size

5. Batch Operations
   - Combine multiple operations
   - Reduce round trips
   - Bulk inserts/updates

6. Database Optimization
   - Sharding
   - Read replicas
   - Query optimization
   - Denormalization
```

```
Throughput vs Latency Trade-off:
- Batching: ↑ Throughput, ↑ Latency
- Parallel: ↑ Throughput, ↓ Latency
- Caching: ↑ Throughput, ↓ Latency
```

3. Response Time

Definition: Time to complete an operation (includes processing + waiting)

Percentile-based SLA

```
Why Percentiles Matter:

Average can be misleading:
- 99 requests: 10ms
- 1 request: 1000ms
- Average: 19.9ms (looks good!)
- But 1% of users have terrible experience

Better: Use Percentiles
- P50 (Median): 10ms - half of requests
- P95: 15ms - 95% of requests
- P99: 50ms - 99% of requests
- P99.9: 200ms - 99.9% of requests

Example SLA:
"API response time:
- P50: < 100ms
- P95: < 500ms
- P99: < 1000ms"

Tail Latency Problem:
- P50: 50ms (great!)
- P99: 2000ms (terrible!)
- Means 1% of users wait 2 seconds
- At scale: 1% = 10,000 users!

Focus on: P95 and P99, not just average
```

Response Time Requirements

```
Interactive vs Batch:

Interactive (User waiting):
- Target: < 200ms (feels instant)
- Acceptable: < 1s (slight delay)
```

```
- Maximum: < 3s (noticeable delay)
- Beyond 3s: Show progress indicator

Batch Processing:
- Minutes to hours acceptable
- Show progress updates
- Allow cancellation
- Notify on completion

API Response Time by Type:
```

API Type	Target
GET (cached)	< 50ms
GET (uncached)	< 200ms
POST (simple)	< 500ms
POST (complex)	< 2s
Search query	< 500ms
Report generate	Async

Scalability Requirements

1. Types of Scalability

```
Horizontal Scaling (Scale Out):
Add more servers to handle load

┌────┐                ┌────┐ ┌────┐ ┌────┐
│ S1 │  Scale Out →   │ S1 │ │ S2 │ │ S3 │
└────┘                └────┘ └────┘ └────┘
100 RPS               300 RPS (3x servers)

Pros:
✅ Linear scaling (usually)
✅ Fault tolerance (redundancy)
✅ Cost-effective
✅ Cloud-friendly

Cons:
❌ Complexity (distributed state)
❌ Load balancing needed
❌ Data consistency challenges
```

```
Vertical Scaling (Scale Up):
Upgrade to more powerful server

┌──────────┐                ┌──────────────┐
│2 cores   │ Scale Up →     │  8 cores     │
│8GB RAM   │                │  32GB RAM    │
└──────────┘                └──────────────┘
100 RPS                     400 RPS (4x resources)

Pros:
✅ Simpler (no distribution)
✅ No data consistency issues
✅ Less operational overhead

Cons:
❌ Limited by hardware
❌ Single point of failure
❌ Expensive at scale
❌ Downtime for upgrades

Best Practice: Design for horizontal scaling
```

2. Scalability Metrics

```
Key Metrics:

1. User Scalability
   - Current: 100K users
   - Target: 10M users (100x)
   - Growth rate: 2x per year

2. Data Scalability
   - Current: 1TB
   - Growth: 100GB/month
   - 5-year projection: 7TB

3. Traffic Scalability
   - Current: 5K RPS
   - Peak: 15K RPS (3x)
   - Black Friday: 50K RPS (10x)

4. Geographic Scalability
   - Current: US only
   - Phase 2: Add Europe
   - Phase 3: Global

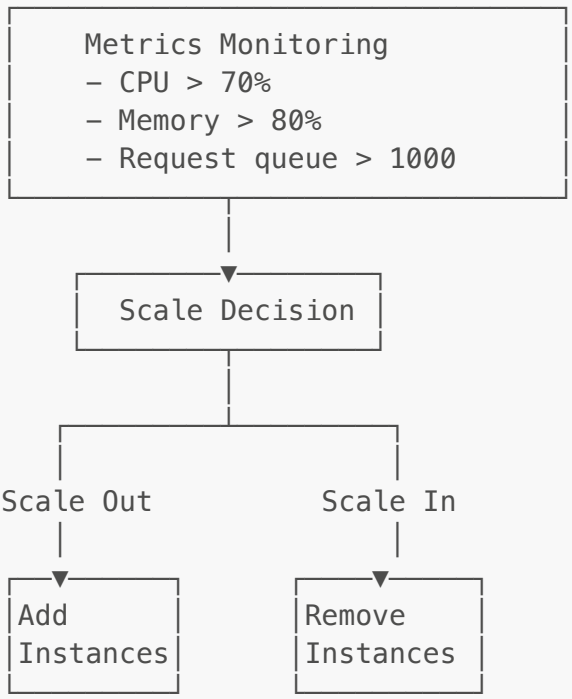
Scaling Patterns:
- Read-heavy: Add read replicas, caching
- Write-heavy: Sharding, partitioning
```

```
- Compute-heavy: Horizontal scaling, async processing
- Storage-heavy: Distributed storage, tiered storage
```

3. Elasticity

```
Definition: Ability to automatically scale up/down based on load

Auto-Scaling Strategy:




Scale-Out Triggers:
- CPU usage > 70% for 5 minutes
- Request queue depth > 1000
- Response time P95 > 500ms
- Active connections > 8000

Scale-In Triggers:
- CPU usage < 30% for 15 minutes
- Request queue depth < 100
- After scale-out, load normalized

Scaling Rules:
- Minimum instances: 3 (HA)
- Maximum instances: 100 (cost limit)
- Scale-out: Add 30% capacity
- Scale-in: Remove 10% capacity
- Cooldown: 5 minutes between actions

Benefits:
✅ Cost optimization
✅ Handle traffic spikes
```

```
✅ Maintain performance
✅ Automatic management
```

Availability & Reliability

1. Availability (Uptime)

Definition: Percentage of time system is operational and accessible

SLA Levels

SLA	Downtime/Year	Downtime/Mo	Common Use
90%	36.5 days	3 days	Internal tools
95%	18.25 days	1.5 days	Non-critical
99%	3.65 days	7.2 hours	Standard
99.9%	8.76 hours	43.2 min	High priority
99.95%	4.38 hours	21.6 min	Critical
99.99%	52.56 min	4.32 min	Mission-crit
99.999%	5.26 min	25.9 sec	Extreme HA

```
Cost of Nines:
Each additional "9" roughly doubles cost
- 99%: Baseline
- 99.9%: 2-3x cost
- 99.99%: 5-10x cost
- 99.999%: 20-50x cost
```

Achieving High Availability

```
Strategies:

1. Redundancy (Multiple Instances)
   ┌────┐ ┌────┐ ┌────┐
   │ S1 │ │ S2 │ │ S3 │  If S1 fails, S2 and S3 continue
   └────┘ └────┘ └────┘

2. Geographic Distribution (Multi-Region)
```

```
   ┌──────────┐  ┌──────────┐  ┌──────────┐
   │US-East   │  │US-West   │  │ Europe   │
   └──────────┘  └──────────┘  └──────────┘
   If US-East fails, traffic to US-West

3. Health Checks & Failover
   - Active monitoring
   - Automatic failover
   - Traffic rerouting
   - Self-healing

4. Fault Isolation
   - Bulkheads (isolate failures)
   - Circuit breakers
   - Graceful degradation
   - Fail independently

5. Zero-Downtime Deployments
   - Blue-green deployment
   - Rolling updates
   - Canary releases
   - Feature flags

Multi-AZ Deployment:
┌─────────────────────────────────────┐
│  Region: US-East-1                  │
│  ┌──────┐ ┌──────┐ ┌──────┐        │
│  │ AZ-1a│ │ AZ-1b│ │ AZ-1c│        │
│  │Server│ │Server│ │Server│        │
│  │  DB  │ │  DB  │ │  DB  │        │
│  └──────┘ └──────┘ └──────┘        │
└─────────────────────────────────────┘

Availability Calculation:
Single AZ: 99.5% availability
Multi-AZ (3): 1 - (0.005)³ = 99.9999875%
(Assumes independent failures)
```

2. Reliability (MTBF & MTTR)

```
Key Metrics:

MTBF (Mean Time Between Failures):
- Average time system runs before failure
- Higher is better
- Example: 720 hours (30 days)

MTTR (Mean Time To Repair):
- Average time to fix when failure occurs
- Lower is better
```

```
- Example: 1 hour

Availability = MTBF / (MTBF + MTTR)
Example: 720 / (720 + 1) = 99.86%

Reliability Patterns:

1. Redundancy
   - Active-Active: Both handle traffic
   - Active-Passive: Standby for failover
   - N+1: N needed, 1 spare

2. Replication
   - Synchronous: Consistent, slower
   - Asynchronous: Faster, may lag
   - Multi-master: Write anywhere

3. Backup & Recovery
   - Regular backups (hourly, daily)
   - Point-in-time recovery
   - Cross-region replication
   - Disaster recovery plan

4. Chaos Engineering
   - Intentionally inject failures
   - Test recovery procedures
   - Improve resilience
   - Netflix Chaos Monkey
```

3. Durability

```
Definition: Data is not lost once written

Durability Levels:
```

Storage Type	Durability	Use Case
Memory (Redis)	None/Low	Cache, temp
Single disk	95%	Development
RAID	99.9%	Production
Replicated (3x)	99.999%	Critical data
S3	99.999999999% (11 nines)	Archive

```
Achieving Durability:
```

```
1. Replication
   - Write to multiple copies
   - Acknowledge after N replicas
   - Quorum: W+R > N

   Example (3 replicas):
   - Write to 2 replicas before ack
   - Read from 2 replicas for consistency
   - W=2, R=2, N=3: W+R > N (consistent)

2. Write-Ahead Logging (WAL)
   - Log before applying
   - Replay on recovery
   - Ensures no data loss

3. Snapshots
   - Periodic full copies
   - Point-in-time recovery
   - Backup strategy

4. Cross-Region Replication
   - Protect against regional failure
   - Higher latency
   - Disaster recovery
```

Consistency Requirements

1. CAP Theorem

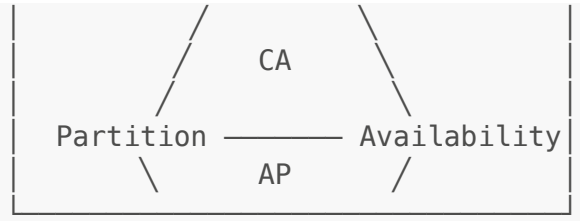
```
Pick 2 of 3:

Consistency (C):
All nodes see same data at same time

Availability (A):
Every request receives response (success/failure)

Partition Tolerance (P):
System continues despite network partition

┌──────────────────────────────────┐
│       CAP Theorem                │
│                                  │
│         Consistency              │
│              / \                 │
│             /   \                │
│            /     \               │
│           /  CP   \              │
│
```




```
In distributed systems: P is mandatory (network always fails)
Real choice: CP vs AP

CP (Consistency + Partition Tolerance):
- Strong consistency
- May refuse requests during partition
- Examples: Banking, inventory
- Systems: MongoDB, HBase, Zookeeper

AP (Availability + Partition Tolerance):
- Always available
- May return stale data during partition
- Examples: Social media, recommendations
- Systems: Cassandra, DynamoDB, Riak
```

2. Consistency Models

```
Strong Consistency:
- Read always returns latest write
- Linearizable
- Highest guarantee

Timeline:
Write(X=1) ──────────────────→ Complete
            Read(X) → Returns 1 ✓

Example: Bank account balance

Eventual Consistency:
- Reads may return stale data
- Eventually all reads return latest
- Highest availability

Timeline:
Write(X=1) ──────────────────→ Complete
  Read(X) → Returns 0 (stale)
  ...wait...
  Read(X) → Returns 1 ✓ (caught up)

Example: Social media likes count

Causal Consistency:
- Causally related operations ordered
```

```
- Concurrent operations may be unordered
- Middle ground

Timeline:
User A: Post → Like own post (causal order maintained)
User B: Like post (concurrent, may see before post!)

Read-Your-Own-Writes:
- User sees their own updates immediately
- Others may see stale data temporarily

Session Consistency:
- Within session: consistent
- Across sessions: eventual

Monotonic Reads:
- Once you read X=1, never read X=0
- Time doesn't go backward

Consistency Trade-offs:
Strong → High latency, lower availability
Eventual → Low latency, high availability
```

3. ACID vs BASE

```
ACID (Traditional Databases):

Atomicity: All or nothing
- Transaction succeeds completely or fails completely

Consistency: Valid state
- Data adheres to rules and constraints

Isolation: Transactions independent
- Concurrent transactions don't interfere

Durability: Permanent writes
- Committed data survives crashes

Use Case: Financial transactions, inventory

BASE (Modern Distributed Systems):

Basically Available:
- System remains available
- May return stale data

Soft State:
- State may change over time
- Even without input (due to eventual consistency)
```

```
Eventual Consistency:
- Will become consistent
- Given enough time

Use Case: Social media, recommendations, analytics

Comparison:
```

Property	ACID	BASE
Consistency	Immediate	Eventual
Availability	May block	Always
Scalability	Limited	Excellent
Complexity	Simple	Complex

Security Requirements

1. Authentication & Authorization

```
Authentication (Who are you?):
- Username/password
- OAuth 2.0
- JWT tokens
- Multi-factor (MFA)
- Biometric
- SSO (Single Sign-On)

Authorization (What can you do?):
- Role-Based Access Control (RBAC)
- Attribute-Based Access Control (ABAC)
- Fine-grained permissions
- Resource-level access

Security Levels:
```

Level	Requirements
Public	No auth, rate limiting
Authenticated	Login required
Sensitive	Auth + encryption
Financial	Auth + MFA + audit

Healthcare	HIPAA compliance + encryption

2. Data Protection

```
Encryption:

In Transit:
- TLS/SSL (HTTPS)
- Minimum TLS 1.2
- Certificate management
- Perfect forward secrecy

At Rest:
- Database encryption
- Disk encryption
- Key management (KMS)
- Field-level encryption

End-to-End:
- Only sender and recipient can read
- Server cannot decrypt
- Use case: WhatsApp, Signal

Data Classification:
```

Level	Protection Requirements
Public	None
Internal	Access control
Confidential	Encryption + access control
Restricted	Encryption + MFA + audit

```
PII (Personally Identifiable Information):
- Name, email, phone: Encrypted
- SSN, credit cards: Tokenized
- Passwords: Hashed (bcrypt, Argon2)
- Audit logs: Track all access
```

3. Security Best Practices

```
Defense in Depth (Multiple Layers):
```

```
Layer 1: Network Security
- Firewall
- DDoS protection
- VPC/Private network
- Security groups

Layer 2: Application Security
- Input validation
- SQL injection prevention
- XSS prevention
- CSRF tokens
- Rate limiting

Layer 3: Data Security
- Encryption
- Access control
- Data masking
- Audit logging

Layer 4: Monitoring
- Intrusion detection
- Anomaly detection
- Security alerts
- Incident response

Common Security Requirements:
- OWASP Top 10 compliance
- Regular security audits
- Penetration testing
- Vulnerability scanning
- Security patches
- Incident response plan
```

Maintainability & Operability

1. Observability

```
Three Pillars of Observability:

1. Logging
   - Application logs
   - Access logs
   - Error logs
   - Audit logs

   Log Levels:
   - ERROR: System failures
   - WARN: Potential issues
   - INFO: Important events
```

```
   - DEBUG: Detailed information

2. Metrics
   - System metrics (CPU, memory, disk)
   - Application metrics (RPS, latency)
   - Business metrics (orders, revenue)
   - Custom metrics

   Key Metrics:
   - Request rate
   - Error rate
   - Duration (P50, P95, P99)
   - Saturation (resource usage)

3. Tracing
   - Distributed tracing
   - Request flow
   - Performance bottlenecks
   - Dependency mapping

Observability Stack:
Logs → ELK (Elasticsearch, Logstash, Kibana)
Metrics → Prometheus + Grafana
Tracing → Jaeger, Zipkin
All-in-one → Datadog, New Relic
```

2. Maintainability

```
Code Maintainability:
- Clean code principles
- Documentation
- Code reviews
- Automated testing
- CI/CD pipelines

System Maintainability:
- Modular architecture
- Loose coupling
- Clear interfaces
- Version control
- Configuration management

Deployment Practices:
- Infrastructure as Code (Terraform)
- Automated deployments
- Rollback capability
- Feature flags
- Canary releases

Documentation Requirements:
```

```
- Architecture diagrams
- API documentation
- Runbooks
- Troubleshooting guides
- Post-mortems
```

Cost & Resource Constraints

1. Infrastructure Costs

```
Cloud Cost Components:

Compute:
- EC2/VM instances
- Container orchestration
- Serverless functions
- Auto-scaling costs

Storage:
- Database storage
- Object storage (S3)
- Block storage
- Backup storage

Network:
- Data transfer out
- Cross-region transfer
- CDN bandwidth
- Load balancer

Cost Optimization Strategies:

1. Right-sizing
   - Monitor actual usage
   - Scale down over-provisioned
   - Use appropriate instance types

2. Reserved Instances
   - 1-year: 30-40% savings
   - 3-year: 50-60% savings
   - Trade flexibility for cost

3. Spot Instances
   - 70-90% savings
   - For fault-tolerant workloads
   - Batch processing
   - Can be interrupted

4. Caching
```

```
   - Reduce database queries
   - Lower data transfer
   - Cheaper cache vs compute

Example Cost Analysis:
Requirement: 100K RPS, 99.9% availability

Option 1: On-Demand
- 50 servers @ $100/mo = $5,000/mo
- Database: $2,000/mo
- Total: $7,000/mo

Option 2: Reserved + Caching
- 30 reserved servers @ $60/mo = $1,800/mo
- 20 spot instances @ $30/mo = $600/mo
- Redis cache: $500/mo
- Database: $1,500/mo
- Total: $4,400/mo (37% savings)
```

Compliance & Legal

1. Data Privacy Regulations

```
GDPR (Europe):
- User consent for data collection
- Right to be forgotten
- Data portability
- Data breach notification (72 hours)
- Privacy by design

CCPA (California):
- Right to know what data collected
- Right to delete data
- Right to opt-out of data sale
- Non-discrimination

HIPAA (Healthcare - US):
- Protected Health Information (PHI)
- Access controls
- Encryption
- Audit trails
- Business Associate Agreements

PCI DSS (Payment Cards):
- Never store CVV
- Tokenization
- Encryption
- Network segmentation
- Regular audits
```

```
Data Residency Requirements:
- EU data must stay in EU
- China data must stay in China
- Russia localization law
- Regional compliance
```

2. Compliance Requirements

```
Audit Requirements:
- Log all access to sensitive data
- Retain logs for X years
- Tamper-proof logs
- Regular audits

Retention Policies:
- Financial records: 7 years
- Medical records: 6 years
- Customer data: Until deletion request
- Logs: 1-3 years

Data Deletion:
- Soft delete vs hard delete
- Cascading deletes
- Backup retention
- Right to be forgotten compliance
```

NFR Trade-offs

1. Performance vs Cost

```
Trade-off Scenario:

Requirement: P99 latency < 100ms

Option A: Over-provision (Safe)
- 100 servers
- 50% average utilization
- Cost: $10,000/mo
- P99: 50ms ✓
- Waste: 50% capacity

Option B: Right-sized (Efficient)
- 60 servers
- 80% average utilization
- Cost: $6,000/mo
- P99: 95ms ✓
```

```
- Risk: Spikes may breach SLA

Option C: Auto-scaling (Balanced)
- 40-80 servers (dynamic)
- 70% average utilization
- Cost: $7,000/mo
- P99: 80ms ✓
- Complexity: Higher

Decision Framework:
- Critical system → Option A
- Cost-sensitive → Option B
- Variable load → Option C
```

2. Consistency vs Availability

```
Trade-off Scenario:

E-commerce Inventory System

Strong Consistency (CP):
┌──────────────────────────────┐
│ Check inventory → Lock row   │
│ If available → Decrement     │
│ If not → Reject order        │
│ Other requests wait          │
└──────────────────────────────┘

Pros:
✅ Never oversell
✅ Accurate inventory
✅ Simple logic

Cons:
❌ Lower availability
❌ Higher latency
❌ Blocking

Eventual Consistency (AP):
┌──────────────────────────────┐
│ Check inventory (may be stale) │
│ Optimistically reserve       │
│ Async validation             │
│ Compensate if oversold       │
└──────────────────────────────┘

Pros:
✅ Higher availability
✅ Lower latency
✅ Better UX
```

```
Cons:
❌ May oversell
❌ Need compensation
❌ Complex logic

Best Practice: Hybrid
- Hot items: Strong consistency
- Regular items: Eventual consistency
- Based on stock level
```

3. Latency vs Throughput

```
Trade-off Example:

Message Processing System

Low Latency (Individual):
- Process each message immediately
- No batching
- Latency: 10ms per message
- Throughput: 100 msgs/sec

High Throughput (Batch):
- Batch 100 messages
- Process together
- Latency: 1000ms per message
- Throughput: 10,000 msgs/sec

Comparison:
```

Approach	Latency	Throughput
Individual	10ms	100/sec
Batch (10)	100ms	1,000/sec
Batch (100)	1000ms	10,000/sec

```
Decision Factors:
- Real-time needs → Low latency
- Analytics/ETL → High throughput
- Hybrid: Batch non-critical, immediate for critical
```

How to Discuss NFRs in Interviews

Interview Framework

```
STEP 1: Always Ask About NFRs (First 5 minutes)

Essential Questions:
"Before we dive into the design, let me clarify the
non-functional requirements:

Performance:
- What's our latency requirement?
- Expected throughput (requests/sec)?
- Any specific response time SLAs?

Scale:
- How many users (concurrent, daily active)?
- Data volume (current and projected)?
- Geographic distribution?

Reliability:
- Availability requirement (99.9%, 99.99%)?
- Can we tolerate data loss?
- Disaster recovery needs?

Consistency:
- Strong or eventual consistency acceptable?
- Read-your-own-writes needed?
- How stale can data be?

Other:
- Security/compliance requirements?
- Budget constraints?
- Any specific regulatory needs?"

This shows: Professional maturity, production thinking
```

How to Present NFRs

```
TEMPLATE:

"Based on the requirements, here are the key NFRs I'll
design for:

PERFORMANCE:
- Latency: P95 < 200ms, P99 < 500ms
- Throughput: 50K requests/sec peak
- Response time: < 1s for page loads

SCALABILITY:
- Support 10M DAU
- Scale to 100K concurrent users
- Handle 5x traffic spikes
```

```
AVAILABILITY:
- 99.99% uptime (52 min downtime/year)
- Multi-AZ deployment
- < 1 min failover time

CONSISTENCY:
- Eventual consistency acceptable
- Read-your-own-writes for user data
- Strong consistency for financial transactions

SECURITY:
- Authentication required
- Data encrypted in transit and at rest
- PCI-DSS compliant for payments

These NFRs will guide my architecture decisions..."

Then explain HOW each NFR affects design:
"For 99.99% availability, I'll deploy across 3 AZs with..."
"For P99 < 500ms, I'll implement caching with..."
```

Common Interview Scenarios

Scenario 1: URL Shortener

```
FUNCTIONAL:
- Shorten long URLs
- Redirect to original URL
- Track click analytics

NON-FUNCTIONAL REQUIREMENTS TO DISCUSS:

Q: "What's the expected throughput?"
A: "100M new URLs/month, 10B redirects/month"

Calculation:
- Writes: 100M / (30 * 86400) = ~40 writes/sec
- Reads: 10B / (30 * 86400) = ~4K reads/sec
- Read:Write ratio = 100:1 (read-heavy!)

Architecture Implications:
✅ Use caching aggressively (Redis)
✅ CDN for redirect endpoint
✅ Read replicas for database
✅ Cache hit ratio target: 95%

Q: "What about latency?"
A: "Redirects should be < 100ms"
```

```
Architecture Implications:
✅ Cache popular URLs (80/20 rule)
✅ CDN reduces network latency
✅ Database indexes on short_url
✅ Monitor cache hit ratio

Q: "Availability requirements?"
A: "99.9% available (43 min downtime/month)"

Architecture Implications:
✅ Multi-AZ deployment
✅ Database replication
✅ Health checks and auto-failover
✅ No single points of failure

Q: "Consistency needs?"
A: "Eventual consistency OK for analytics,
    Strong consistency for URL creation"

Architecture Implications:
✅ Async analytics processing
✅ ACID database for URL storage
✅ Separate read/write paths
```

Scenario 2: Twitter-like Feed

```
NON-FUNCTIONAL REQUIREMENTS:

SCALE:
- 500M DAU
- 100M concurrent users
- 500M tweets/day
- 5B timeline reads/day

Calculations:
- Writes: 500M / 86400 = 5,787 tweets/sec
- Reads: 5B / 86400 = 57,870 reads/sec
- Peak: 3x = 17K writes/sec, 174K reads/sec

LATENCY:
- Feed load: < 1s
- Tweet post: < 500ms
- Notifications: < 2s

AVAILABILITY:
- 99.99% uptime
- No data loss for tweets
- Graceful degradation for non-critical features
```

```
CONSISTENCY:
- Eventual consistency for feed (stale < 5s OK)
- Strong consistency for tweet storage
- Eventually consistent follower counts

Architecture Decisions Based on NFRs:

For High Read Throughput:
✅ Redis cache for timelines (hot data)
✅ CDN for media assets
✅ Read replicas (10+)
✅ Fanout-on-write for active users
✅ Fanout-on-read for inactive users

For Low Latency:
✅ Pre-computed timelines in cache
✅ Geographic distribution
✅ Async processing for analytics
✅ CDN for static assets

For High Availability:
✅ Multi-region active-active
✅ Database replication across regions
✅ Cache replication
✅ Graceful degradation (show cached data if DB down)

For Eventual Consistency:
✅ Async fanout to followers
✅ Timeline updates propagate within 5s
✅ Follower counts updated async
✅ Strong consistency for tweet storage only
```

Measuring & Monitoring

Key Metrics to Track

```
1. Golden Signals (Google SRE):

Latency:
- Request duration
- P50, P95, P99, P99.9
- By endpoint/service
- Historical trends

Traffic:
- Requests per second
- Bandwidth usage
- Active connections
- Request patterns
```

```
Errors:
- Error rate (%)
- Error types
- Failed requests
- 4xx vs 5xx errors

Saturation:
- CPU usage
- Memory usage
- Disk I/O
- Network bandwidth
- Queue depth

2. Business Metrics:

User Engagement:
- Daily/Monthly active users
- Session duration
- Feature usage
- Conversion rates

System Health:
- Availability (uptime %)
- Success rate
- Cache hit ratio
- Database connection pool

Financial:
- Infrastructure costs
- Cost per request
- Revenue per user
- ROI metrics
```

Monitoring Strategy

```
Levels of Monitoring:

Level 1: Infrastructure
- Server health
- Network connectivity
- Disk space
- Resource utilization

Level 2: Application
- Request/response metrics
- Error rates
- API performance
- Queue depths
```

```
Level 3: Business
- User actions
- Revenue metrics
- Feature adoption
- Customer satisfaction

Alert Priorities:

P0 (Critical):
- System down
- Data loss
- Security breach
Response: Immediate (page on-call)

P1 (High):
- Degraded performance
- Error rate spike
- SLA breach
Response: Within 15 minutes

P2 (Medium):
- Resource warnings
- Slow queries
- Cache misses
Response: Within 1 hour

P3 (Low):
- Optimization opportunities
- Long-term trends
- Capacity planning
Response: Next business day
```

Interview Tips

How to Ace NFR Discussion

```
DEMONSTRATE PRODUCTION EXPERIENCE:

1. Start with NFRs
   "Before designing, I need to understand:
   - What's our latency requirement?
   - Expected scale?
   - Availability needs?"

2. Use Specific Numbers
   ❌ "System should be fast"
   ✅ "P99 latency < 200ms"

   ❌ "Handle lots of users"
```

```
   ✅ "Scale to 1M concurrent, 10M DAU"

   ❌ "Highly available"
   ✅ "99.99% uptime, multi-AZ deployment"

3. Explain Trade-offs
   "For 99.99% availability, I'll deploy across 3 AZs,
   which increases cost by 50% but provides redundancy.
   Alternatively, 99.9% with 2 AZs would be cheaper but
   allows more downtime."

4. Calculate Capacity
   "With 10M DAU doing 50 actions each:
   - Average: 5,787 RPS
   - Peak (3x): 17,361 RPS
   - Servers needed (1K RPS each): 20
   - With buffer (1.5x): 30 servers"

5. Discuss Monitoring
   "I'll monitor:
   - Latency (P50, P95, P99)
   - Error rate
   - Throughput
   - Resource utilization
   Alert if P99 > 500ms or error rate > 1%"
```

Common NFR Questions

```
Q: "How would you ensure 99.99% availability?"

Strong Answer:
"99.99% allows 52 minutes downtime per year. I'll achieve this through:
1. Multi-AZ deployment (3 availability zones)
2. Redundant components (no single point of failure)
3. Health checks with automatic failover (<1 min)
4. Zero-downtime deployments (rolling updates)
5. Chaos testing to verify resilience
This gives theoretical availability of 99.9995%,
providing buffer for unexpected issues."

Q: "How do you handle 10x traffic spike?"

Strong Answer:
"Prepare for spikes through:
1. Auto-scaling (scale out to 10x capacity in 5 min)
2. Aggressive caching (reduces backend load 80%)
3. Rate limiting (protect system from overload)
4. Queue-based processing (smooth load)
5. Graceful degradation (disable non-critical features)
6. Pre-warming during known events (Black Friday)"
```

```
Q: "Strong consistency vs eventual consistency?"

Strong Answer:
"Depends on use case. For financial transactions like payments,
strong consistency is mandatory - can't have duplicate charges.
For social media likes, eventual consistency is fine - few
seconds delay acceptable. I'd use:
- Strong: Financial, inventory, user auth
- Eventual: Feeds, likes, view counts, recommendations
Trade-off: Strong adds latency but prevents inconsistency."
```

Quick Reference

NFR Checklist for Interviews

```
ALWAYS ASK ABOUT:
□ Latency requirements (ms, seconds?)
□ Throughput needs (requests/sec, data/sec)
□ Scale (users, data volume, geographic)
□ Availability SLA (99.9%, 99.99%?)
□ Consistency needs (strong, eventual?)
□ Data durability (can we lose data?)
□ Security requirements (auth, encryption?)
□ Compliance (GDPR, HIPAA, PCI?)
□ Budget constraints
□ Growth projections

DURING DESIGN, ADDRESS:
□ How you'll achieve latency targets
□ Capacity calculations for throughput
□ Scaling strategy (horizontal/vertical)
□ High availability architecture
□ Consistency model choice
□ Security measures
□ Monitoring and alerting
□ Cost optimization

SHOW DEPTH WITH:
□ Specific numbers (not just "fast")
□ Percentiles (P95, P99, not just average)
□ Calculations (capacity planning)
□ Trade-off discussions
□ Alternative approaches
□ Real-world constraints
```

Common NFR Values

```
Typical Interview Numbers:

Users:
- Small: 10K-100K users
- Medium: 1M-10M users
- Large: 100M+ users

Throughput:
- Low: 100-1K RPS
- Medium: 10K-100K RPS
- High: 1M+ RPS

Latency:
- Real-time: < 50ms
- Interactive: < 200ms
- Standard: < 1s
- Batch: Minutes-hours

Availability:
- Standard: 99.9%
- High: 99.99%
- Critical: 99.999%

Data:
- Small: GB range
- Medium: TB range
- Large: PB range
```

Conclusion

Non-functional requirements are as important as functional requirements in system design. They drive architectural decisions and demonstrate your production experience.

Key Principles

1. **Always Clarify NFRs First**
    - Don't assume
    - Ask specific questions
    - Get concrete numbers
2. **Use NFRs to Drive Design**
    - High availability → Multi-AZ
    - Low latency → Caching, CDN
    - High throughput → Horizontal scaling
    - Strong consistency → ACID database
3. **Discuss Trade-offs**

- Performance vs Cost
- Consistency vs Availability
- Latency vs Throughput
- Simplicity vs Features

4. **Be Specific**
   - Use numbers (99.99%, <100ms, 10K RPS)
   - Use percentiles (P95, P99)
   - Calculate capacity
   - Estimate costs

5. **Show Production Thinking**
   - Monitoring and alerting
   - Failure scenarios
   - Operational concerns
   - Cost awareness

Interview Success Pattern

```
1. Requirements: "What are the NFRs?"
2. Calculations: "With 10M users, that's X RPS..."
3. Architecture: "To achieve 99.99% availability, I'll..."
4. Trade-offs: "We could also do Y, but the trade-off is..."
5. Monitoring: "I'll track these metrics..."

This demonstrates: Systematic thinking, production experience,
depth of knowledge, ability to make engineering trade-offs
```

Remember

NFRs are not afterthoughts - they're fundamental to system design. Strong candidates:

- Ask about NFRs upfront
- Use them to justify decisions
- Discuss trade-offs explicitly
- Show production awareness
- Demonstrate depth with numbers

Good luck with your interviews!

Additional Resources

Books

- "Site Reliability Engineering" by Google

- "Designing Data-Intensive Applications" by Martin Kleppmann
- "The Art of Scalability" by Martin L. Abbott

Online Resources

- Google SRE Book (free online)
- AWS Well-Architected Framework
- Azure Architecture Best Practices
- High Scalability Blog

Key Concepts to Study

- CAP Theorem
- PACELC Theorem
- SLA/SLO/SLI
- The Four Golden Signals
- Percentile-based monitoring
- Cost optimization strategies