The Complete Systems Design Checklist & Guide

Everything You Need to Review When Designing a Production System

Based on principles from Designing Data-Intensive Applications
by Martin Kleppmann and real-world engineering practice

Table of Contents

1. Problem Definition & Scope

Every good system design starts with a crisp understanding of the problem. This seems obvious, but a shocking number of failed projects trace back to a murky problem statement. The goal here is to draw a clear boundary around what you are building, who it is for, and what constraints you are working within.

What problem are we solving?

You should be able to state the core problem in one or two sentences. If you cannot, the scope is too vague. For example: "We need a system that lets users upload, store, and share photos with their followers, handling up to 50 million daily active users." That single sentence immediately tells you about storage, access patterns, social graph, and scale.

Who are the users?

Understanding your users shapes almost every decision. How many users do you have today, and how many do you expect in a year? Where are they geographically? A system serving users in a single country is fundamentally different from one serving users globally. Are there different user types with different needs (consumers vs. merchants, free vs. paid)?

Core use cases

Identify the 3 to 5 things the system absolutely must do. Resist the temptation to list 30 features. Focus on the critical path. Everything else is secondary. For a messaging app, the core use cases might be: send a message, receive a message in real time, retrieve message history, and show online status.

What is out of scope?

Equally important is defining what you are NOT building. This prevents scope creep and keeps the design focused. If you are building a messaging system, explicitly state whether you are handling media uploads, group chats, end-to-end encryption, or read receipts in this phase or not.

Constraints

Every system operates under constraints. Budget limits how much infrastructure you can provision. Timeline affects whether you build or buy. Team expertise determines which technologies are realistic. Regulatory requirements like GDPR, HIPAA, or PCI-DSS can dictate entire architectural decisions around data storage, encryption, and retention. Existing infrastructure means you may need to integrate with legacy systems.

2. Functional Requirements

Functional requirements describe what the system does from the perspective of its users. This is the contract between you and your stakeholders. Getting this wrong means building the wrong thing.

User-facing operations

List every operation a user can perform. For each, think about the input, the expected output, and the side effects. A social media platform might have: create a post (input: text/images, output: post ID, side effect: appears in followers' feeds), like a post (input: post ID, output: updated like count, side effect: notification to author), search for users (input: query string, output: ranked list of user profiles).

Access patterns

This is one of the most critical inputs to your design. Is the system read-heavy (like a news site where millions read but few write), write-heavy (like a logging system ingesting millions of events per second), or mixed? What is the read/write ratio? 100:1? 1000:1? 1:1? This directly determines your storage engine choice, caching strategy, and replication topology.

Real-time vs. batch

Some operations need to happen in real time (chat messages, live notifications, stock price updates), while others can be processed in the background (generating daily reports, training recommendation models, sending digest emails). Identifying which is which prevents you from over-engineering things that do not need millisecond latency and under-engineering things that do.

Search and filtering

If users need to search or filter data, this has major architectural implications. Simple lookups by ID are trivial. Full-text search requires a dedicated search engine like Elasticsearch. Faceted search, autocomplete, and fuzzy matching each add complexity. Define what is searchable early, because retrofitting search into an existing system is painful.

Multi-tenancy

If your system serves multiple customers (tenants), you need to decide: shared database with tenant column, separate schemas per tenant, or separate databases per tenant. Each has trade-offs in isolation, complexity, cost, and performance. This decision is very hard to change later.

3. Non-Functional Requirements

Non-functional requirements define how well the system performs its functions. They are often more important than functional requirements because they determine whether users actually enjoy using the system and whether it survives in production.

3a. Performance

Performance means different things in different contexts. Latency is how long a single request takes. Throughput is how many requests the system handles per unit time. They are related but distinct: a system can have high throughput but poor latency for individual requests.

Always think in percentiles, not averages. Average latency hides the suffering of your worst-off users. If your p50 latency is 100ms but your p99 is 5 seconds, one in a hundred users is having a terrible experience. Amazon found that every 100ms of additional latency cost them 1% in sales. Define targets for p50, p95, and p99.

Set different latency budgets for different operations. A user login can tolerate 500ms. A search autocomplete must respond in under 50ms. A report generation might take 30 seconds and that is fine because the user expects it.

3b. Availability & Reliability

Availability is the percentage of time the system is operational. The "number of nines" is the standard measure. 99.9% (three nines) allows about 8.7 hours of downtime per year. 99.99% (four nines) allows about 52 minutes. 99.999% (five nines) allows about 5 minutes. Each additional nine is exponentially harder and more expensive to achieve.

Recovery Time Objective (RTO) is how quickly you must restore service after a failure. Recovery Point Objective (RPO) is how much data you can afford to lose. A banking system might need an RPO of zero (no data loss ever) while a social media platform might tolerate losing the last few seconds of likes.

Think about graceful degradation. Not every part of the system is equally critical. If your recommendation engine goes down, you can show a default list. If your payment processor goes down, you cannot process orders. Identify which components are critical (must never fail) versus which can degrade (show cached data, reduced functionality).

3c. Consistency

Consistency is one of the most nuanced topics in systems design. Strong consistency means every read sees the most recent write. Eventual consistency means reads might return stale data for a while, but will eventually converge.

The key insight from Kleppmann is that consistency requirements are per-operation, not per-system. Your bank balance needs strong consistency. Your Instagram follower count can be eventually consistent. Your shopping cart probably needs read-your-writes consistency (you should see the items you just added, but it is fine if other users see the old count briefly).

Identify your system's invariants: rules that must never be broken. You must never sell more airline seats than exist on the plane. You must never withdraw more money than is in the account. You must never assign the same username to two people. These invariants determine where you need strong consistency and transactions.

3d. Durability

Durability is about data survival. Once a user is told their write succeeded, the data must not disappear. Different data has different durability requirements. Financial transactions require extreme durability (replicated across multiple datacenters, backed up, with point-in-time recovery). Cached session data might not need any durability at all because it can be recomputed.

Think about the durability of your backups too. A backup that has never been tested is not a backup. It is a hope.

4. Back-of-the-Envelope Estimates

Before choosing any technology or architecture, do the math. Rough calculations prevent you from choosing a design that cannot possibly meet your requirements, and they prevent you from over-engineering a system that does not need it.

Traffic estimates

Start with users and work down to requests per second. If you have 10 million daily active users and each user makes 20 requests per day on average, that is 200 million requests per day, or about 2,300 requests per second on average. But peak traffic might be 3 to 5 times the average, so design for 7,000 to 12,000 requests per second.

Storage estimates

Calculate the size of each type of data object and multiply by the expected count. A tweet is about 300 bytes of text. If Twitter gets 500 million tweets per day, that is 150 GB per day of tweet text alone, or about 55 TB per year. Add metadata, indexes, images, and replication and you are looking at petabytes.

Bandwidth estimates

Multiply your request rate by the average response size. If each API response is 10 KB and you serve 10,000 requests per second, that is 100 MB/s or about 800 Mbps of outgoing bandwidth. This matters for network capacity planning, CDN costs, and data transfer pricing.

Memory and cache estimates

The 80/20 rule often applies: 80% of requests access 20% of data. If your total dataset is 1 TB but only 200 GB is hot data, you might need 200 GB of cache memory. At roughly 64 GB per cache server, that is 3 to 4 Redis instances. Double it for redundancy.

Growth projections

Project your estimates at 1 year, 3 years, and 5 years. Is growth linear (adding a fixed number of users per month) or exponential (doubling every quarter)? Exponential growth means your estimates for year 3 might be 10x or 100x year 1. This matters because some architectures scale linearly while others hit walls.

5. API Design

The API is the contract between your system and the outside world. Once published, it is very hard to change without breaking clients. Invest time here.

Protocol choice

REST is the default for public-facing APIs. It is well-understood, cacheable, and works with any HTTP client. GraphQL is excellent when clients need flexible queries and you want to avoid over-fetching or under-fetching. gRPC is ideal for internal service-to-service communication where performance matters (binary protocol, HTTP/2, streaming support). WebSockets are for real-time bidirectional communication (chat, live updates, collaborative editing).

Authentication and authorization

Authentication (who are you?) and authorization (what can you do?) are separate concerns. OAuth2 with JWT tokens is the standard for most web APIs. For machine-to-machine, API keys or mutual TLS work well. Authorization models range from simple role-based access control (admin, editor, viewer) to fine-grained attribute-based access control (user X can edit document Y only if they are in department Z).

Rate limiting

Every public API needs rate limiting. Without it, a single misbehaving client can take down your system. Common strategies include fixed window (100 requests per minute), sliding window (smoother), and token bucket (allows bursts). Different endpoints may need different limits: a search endpoint might allow 10 requests per second while a login endpoint allows 5 per minute.

Idempotency

Network failures mean clients sometimes retry requests. If a payment request is retried, you do not want to charge the user twice. Idempotency keys solve this: the client sends a unique key with each request, and the server deduplicates. Any state-changing operation that could be retried should support idempotency.

Pagination

Any endpoint that returns a list needs pagination. Offset-based pagination (page=3&limit=20) is simple but breaks when data is inserted or deleted between pages. Cursor-based pagination (after=abc123&limit=20) is more robust and performs better on large datasets because the database can seek directly to the cursor instead of counting rows to skip.

Versioning

APIs evolve. You need a versioning strategy so old clients keep working when you deploy new versions. URL versioning (/v1/users, /v2/users) is explicit and easy to understand. Header versioning (Accept: application/vnd.api+json;version=2) is cleaner but less visible. Whatever you choose, be consistent and publish a deprecation policy.

6. Data Model & Storage

This is the foundation of your entire system. Bad data modeling creates problems that propagate everywhere. Good data modeling makes everything downstream easier.

6a. Choosing a data model

The relational model (tables, rows, columns, joins) is the right default for most applications. It has decades of battle-tested tooling, strong consistency guarantees, and handles complex queries well. Use it unless you have a specific reason not to.

The document model (JSON-like nested objects) works well when your data has a natural tree structure and you typically load the whole document at once. A user profile with nested addresses, preferences, and history fits well in a document. But if you frequently need to query across documents or join them, a document database becomes painful.

Graph databases shine when relationships between entities are the primary concern. Social networks, recommendation engines, fraud detection networks, and knowledge graphs are natural fits. If you find yourself doing recursive SQL queries with 6 levels of self-joins, consider a graph database.

Wide-column stores like Cassandra are designed for massive write throughput and horizontal scaling. They are excellent for time-series data, event logging, and any workload where you write far more than you read and can tolerate eventual consistency.

6b. Schema design

Normalization (eliminating data duplication) reduces inconsistency but requires joins at read time. Denormalization (duplicating data) eliminates joins and speeds up reads but means you must update data in multiple places. The right choice depends on your read/write ratio. Read-heavy systems benefit from denormalization. Write-heavy systems benefit from normalization.

Design your schema around your query patterns, not around some abstract notion of correctness. If your most common query is "get all posts by user X with their comment counts," your schema should make that query fast, even if it means denormalizing the comment count into the posts table.

6c. Indexing strategy

Indexes speed up reads but slow down writes (because the index must be updated on every write). Every index also consumes disk space and memory. Think carefully about which columns to index based on your actual query patterns. Composite indexes (on multiple columns) are powerful but order matters: an index on (user_id, created_at) is useful for queries filtering on user_id or on both user_id and created_at, but useless for queries filtering only on created_at.

Watch out for hot keys: if one key gets vastly more traffic than others, it can bottleneck a single partition or index node. A celebrity's user_id in a social media system is a classic hot key.

6d. Schema evolution

Your schema will change. New features require new columns, new tables, new relationships. Plan for zero-downtime schema migrations from the start. Common strategies include: adding new columns with defaults (safe), renaming columns through a two-phase process (add new, backfill, migrate code, drop old), and using expand-and-contract migrations for breaking changes.

---

7. Caching

Caching is the single most effective tool for improving read performance and reducing database load. But caching introduces complexity: stale data, cache invalidation, cold starts, and consistency issues. Use it deliberately, not reflexively.

What to cache

Cache data that is read frequently, changes infrequently, and is expensive to compute or fetch. A user's profile that is read on every page load but updated once a month is an excellent cache candidate. A real-time stock price that changes every millisecond is a terrible one.

Where to cache

Client-side caching (browser cache, mobile app cache) eliminates the network round trip entirely. CDN caching (CloudFront, Cloudflare) caches static assets and API responses at edge locations near users. Application-layer caching (Redis, Memcached) sits in front of your database and serves hot data from memory. Database query caching is built into most databases but is less flexible.

Cache invalidation strategies

Cache-aside (lazy loading): the application checks the cache first; on a miss, it queries the database, writes the result to cache, and returns it. Simple and flexible, but the first request after a miss is slow. Write-through: every write goes to both the cache and the database. Ensures cache is always fresh but adds latency to writes. Write-behind: writes go to cache first and are asynchronously flushed to the database. Fast writes but risks data loss if the cache crashes. Time-to-live (TTL): data automatically expires after a set time. Simple but means data can be stale for up to the TTL duration.

Cache stampede

When a popular cache key expires, hundreds or thousands of simultaneous requests hit the database to regenerate it. This can overload the database. Solutions include: request coalescing (only one request recomputes, others wait), probabilistic early expiration (randomly refresh before TTL expires), and locking (only one process can recompute a given key).

Cache as a single point of failure

If your system relies on cache to function (as opposed to just being faster with cache), you have made cache a critical dependency. Ask: can the system survive a total cache failure? If not, you need replicated cache clusters, fallback strategies, and cache warming procedures.

8. Replication & Partitioning

Replication and partitioning are the two fundamental techniques for distributing data across multiple machines. Replication copies the same data to multiple nodes. Partitioning splits different data across different nodes. Most production systems use both.

8a. Replication

Single-leader replication is the most common approach. One node (the leader) accepts all writes and propagates them to followers. Followers serve reads. This is simple and provides good read scalability. The trade-off is that the leader is a bottleneck for writes and a potential single point of failure.

Multi-leader replication allows writes at multiple nodes. This is useful for multi-datacenter deployments where you want low write latency in each region. The price is handling write conflicts: if two leaders modify the same record simultaneously, you need a conflict resolution strategy (last-write-wins, merge, custom logic). This is much harder than it sounds.

Leaderless replication (Dynamo-style) allows any node to accept reads and writes. Clients send writes to multiple nodes and read from multiple nodes, using quorums ($W + R > N$) to ensure consistency. This provides the highest availability but with weaker consistency guarantees and complex conflict resolution (vector clocks, CRDTs).

Synchronous vs. asynchronous

Synchronous replication guarantees that a write is not confirmed until it is replicated. This gives you strong durability but adds latency (you wait for the slowest replica). Asynchronous replication confirms the write immediately and replicates in the background. This is faster but means recent writes can be lost if the leader crashes before replication completes. Most systems use a semi-synchronous approach: one synchronous follower for safety, the rest asynchronous for performance.

Replication lag problems

With asynchronous replication, followers may lag behind the leader. This creates situations where a user writes data and then cannot read it back (because they hit a stale follower). Solutions include read-your-writes consistency (route reads of your own data to the leader), monotonic reads (ensure a single user always reads from the same replica), and consistent prefix reads (ensure causally related writes appear in the right order).

8b. Partitioning (Sharding)

When a single machine cannot hold all your data or handle all your traffic, you split the data across multiple machines. The critical decision is choosing the partition key.

Key range partitioning sorts data by key and assigns contiguous ranges to different partitions. This enables efficient range queries (get all records from January) but can create hot spots if certain key ranges get more traffic.

Hash partitioning applies a hash function to the key and assigns partitions based on hash ranges. This distributes data evenly but destroys key ordering, making range queries expensive (you must query all partitions).

A common pattern is compound partitioning: hash on one part of the key (e.g., user_id) for even distribution, then sort within each partition by another key (e.g., timestamp) for efficient range queries within a partition.

Hot spots

Even with hash partitioning, hot spots can occur if a single key gets extreme traffic (a celebrity's social media profile, a viral post). Solutions include key splitting (append a random suffix to the hot key, spreading it across partitions) and dedicated handling (route hot keys to specially provisioned partitions).

Rebalancing

When you add or remove nodes, data needs to move. Strategies include fixed number of partitions (create many more partitions than nodes and reassign whole partitions), dynamic partitioning (split partitions that get too large), and consistent hashing (minimize data movement when nodes change). Automatic rebalancing is convenient but can be dangerous: if a node appears to be overloaded and the system automatically rebalances away from it, the rebalancing itself adds more load and can cascade.

9. Networking & Communication

How your services talk to each other determines the reliability, latency, and resilience of the whole system.

Sync vs. async communication

Synchronous communication (HTTP request-response, gRPC calls) is simple but creates temporal coupling: the caller blocks until the callee responds. If the callee is slow or down, the caller is affected. Asynchronous communication (message queues, event streams) decouples services in time: the producer sends a message and moves on, the consumer processes it later. Async is more resilient but harder to reason about (no immediate response, eventual consistency).

Service discovery

In a dynamic environment where services scale up and down, clients need to find service instances. DNS-based discovery is simple but slow to update. Client-side discovery (the client queries a registry like Consul or etcd) gives clients control but adds complexity. Server-side discovery (a load balancer queries the registry) keeps clients simple. Service meshes like Istio handle discovery, load balancing, and security transparently.

Load balancing

Round-robin distributes requests evenly but ignores server load. Least connections routes to the server with fewest active connections, which naturally adapts to varying request durations. Consistent hashing ensures the same client or key always reaches the same server, which is useful for cache locality. Weighted routing lets you send more traffic to beefier servers.

Resilience patterns

Circuit breakers prevent cascading failures: if a downstream service is failing, stop sending it requests and fail fast instead of waiting for timeouts. Retries with exponential backoff handle transient failures without overwhelming the failing service. Add jitter (randomness) to prevent thundering herds where all clients retry at the same time. Timeouts must be set on every external call. A missing timeout means a single slow dependency can tie up all your threads.

10. Message Queues & Event Streaming

Message queues and event streams are the backbone of asynchronous architectures. They decouple producers from consumers, buffer traffic spikes, and enable event-driven designs.

Queues vs. logs

Traditional message queues (RabbitMQ, SQS) deliver each message to one consumer and delete it after acknowledgment. Good for task distribution (process this image, send this email). Log-based message brokers (Kafka, Kinesis) append messages to a persistent, ordered log. Multiple consumers can read from the same log independently, and messages are retained for a configurable period. Good for event sourcing, change data capture, and any pattern where multiple systems need to react to the same events.

Delivery guarantees

At-most-once: the message might be lost but will never be delivered twice. At-least-once: the message will be delivered but might be duplicated (the consumer must be idempotent). Exactly-once: the message is delivered exactly once (extremely hard to achieve in distributed systems; Kafka achieves it within Kafka through idempotent producers and transactional consumers, but true end-to-end exactly-once requires careful design).

Event sourcing and CDC

Event sourcing means storing every state change as an immutable event ("user changed email to X") rather than storing the current state ("user email is X"). This gives you a complete audit trail, the ability to reconstruct any past state, and a natural event stream for other systems. Change data capture (CDC) watches the database transaction log and publishes changes as events. This lets you derive search indexes, caches, and analytics without dual-write inconsistencies.

Dead letter queues and backpressure

Some messages cannot be processed (bad format, missing dependencies, bugs). A dead letter queue catches these failed messages so they do not block the main queue. You can inspect and retry them later. Backpressure is what happens when producers generate messages faster than consumers can process them. Without handling, the queue grows unboundedly until you run out of memory or disk. Solutions include flow control (slow down producers), dropping messages (if acceptable), and elastic scaling of consumers.

11. Compute & Service Architecture

How you organize your code and deploy it determines how fast you can develop, how independently teams can work, and how the system scales.

Monolith vs. microservices

Start with a monolith unless you have a strong reason not to. A well-structured monolith is simpler to develop, deploy, debug, and reason about. Microservices add enormous operational complexity: distributed tracing, inter-service communication, data consistency across services, deployment coordination, and more.

Microservices make sense when you have large teams that need to deploy independently, when different components have very different scaling requirements, or when you need to use different technology stacks for different components. The right approach for most teams is a modular monolith that can be split into services later if needed.

Stateless vs. stateful services

Stateless services do not store any data between requests. Any instance can handle any request. This makes scaling trivial: just add more instances. Most web application servers should be stateless, with state pushed to databases, caches, and queues. Stateful services (databases, caches, stream processors) are harder to scale because each instance holds specific data. They require careful partitioning and replication.

Auto-scaling

Define scaling triggers carefully. CPU utilization is common but not always the best signal. Queue depth is often better for worker services. Request latency can signal that you need more capacity before CPU saturates. Always set both minimum and maximum instance counts. Scale out aggressively (to handle load) but scale in conservatively (to avoid flapping). Set a cooldown period between scaling actions.

Serverless

Functions-as-a-Service (Lambda, Cloud Functions) are excellent for bursty, event-driven workloads with unpredictable traffic. You pay only for what you use and scaling is automatic. The trade-offs are cold start latency, execution time limits, vendor lock-in, and difficulty with complex state management. Good for: webhook handlers, image processing, scheduled tasks. Bad for: long-running processes, latency-sensitive services, anything needing persistent connections.

12. Batch & Stream Processing

Most real-world systems need derived data: search indexes built from primary data, recommendation models trained on user behavior, aggregated metrics computed from raw events. Batch and stream processing are how you create and maintain this derived data.

Batch processing

Batch processing takes a large chunk of data (a day's worth of events, a full database dump) and processes it all at once. MapReduce was the original framework for this; modern tools like Spark and Flink are faster because they can keep intermediate data in memory instead of writing it to disk between stages. Batch processing is good for: building search indexes, training ML models, computing daily reports, backfilling data after schema changes.

Stream processing

Stream processing handles data as it arrives, one event at a time or in small micro-batches. This gives you near-real-time results. Tools include Kafka Streams, Apache Flink (which does both batch and stream), and custom consumers. Stream processing is good for: real-time analytics dashboards, fraud detection, monitoring and alerting, maintaining materialized views that update as data changes.

Lambda vs. Kappa architecture

Lambda architecture runs both a batch layer (for accuracy) and a stream layer (for timeliness) and merges their outputs. This gives you the best of both worlds but is complex to maintain because you have two codepaths for the same logic. Kappa architecture uses only stream processing for everything, treating batch as a special case of stream (replay the entire log). This is simpler but requires a log that retains all historical data.

Handling late data

In stream processing, events can arrive late due to network delays, client clock skew, or retry logic. You need a windowing strategy (how long do you keep a window open for late arrivals?) and a policy for updating previously emitted results when late data arrives. Watermarks (a heuristic for the oldest unprocessed event time) help, but no solution is perfect.

13. Security

Security is not a feature you add at the end. It is a property of the entire system that must be designed in from the beginning. A security breach can destroy user trust and your business.

Defense in depth

Never rely on a single layer of security. Use multiple overlapping layers: network security (firewalls, VPCs, private subnets), transport security (TLS everywhere), application security (input validation, parameterized queries), authentication (verify identity), authorization (verify permissions), and monitoring (detect breaches). If one layer fails, others still protect you.

Authentication

Use established standards. OAuth2 with OpenID Connect for user-facing applications. API keys or mutual TLS for service-to-service communication. Always enforce multi-factor authentication for admin accounts and sensitive operations. Never store passwords in plaintext; use bcrypt, scrypt, or Argon2 with proper salt.

Authorization

Role-based access control (RBAC) assigns permissions to roles and users to roles. Simple and works for most applications. Attribute-based access control (ABAC) makes decisions based on attributes of the user, resource, and environment ("user in engineering department can access engineering documents during business hours"). More flexible but more complex. Whatever model you choose, enforce it at the API layer, not just the frontend.

Encryption

Encrypt data in transit (TLS 1.3 for all connections, including internal service-to-service) and at rest (AES-256 for databases, file stores, backups). Manage encryption keys carefully: use a key management service (AWS KMS, HashiCorp Vault), rotate keys regularly, and separate the keys from the data they protect.

Input validation

Every input from outside the system is potentially malicious. Validate and sanitize everything: SQL injection (use parameterized queries, never string concatenation), cross-site scripting (escape output, use Content Security Policy), cross-site request forgery (use CSRF tokens), request size limits (prevent memory exhaustion), and file upload validation (check type, size, and scan for malware).

Audit logging

Record every security-relevant action: who logged in, who accessed what data, who changed what configuration, who was granted what permission. These logs must be tamper-proof (write-once storage or append-only logs) and retained for compliance periods. They are essential for incident investigation and regulatory compliance.

14. Privacy & Compliance

Privacy and compliance are not just legal checkboxes. They are fundamental architectural constraints that affect data storage, retention, access control, and even which cloud regions you can use.

Regulatory landscape

GDPR (European Union) gives users the right to access, correct, and delete their data, and requires explicit consent for data processing. CCPA (California) gives similar rights to California residents. HIPAA (US healthcare) requires specific safeguards for health information. PCI-DSS applies to any system that handles credit card data. SOC2 is an audit framework for service providers. Understand which regulations apply to your system and design compliance in from the start.

Right to deletion

GDPR's "right to be forgotten" means you must be able to completely delete a user's personal data when requested. This is architecturally challenging. You must find and delete data from every database, cache, search index, analytics system, backup, and log file. If you use event sourcing with an immutable log, you need a strategy for redacting events (crypto-shredding: encrypt personal data with a per-user key, and delete the key to make the data unreadable).

Data minimization

Collect only the data you actually need. Every piece of personal data you store is a liability: it can be breached, subpoenaed, or create compliance obligations. If you do not need a user's date of birth for your service, do not collect it. If you only need their age range, store that instead.

Data residency

Some regulations require that data about certain users stays within specific geographic boundaries. EU personal data might need to stay in EU data centers. Russian personal data might need to stay in Russia. This affects your cloud region choices, replication strategy, and CDN configuration.

15. Failure Modes & Resilience

Every component in your system will fail eventually. The question is not if, but when, and whether your system handles it gracefully or catastrophically.

Single points of failure

Walk through every component and ask: if this dies right now, what happens? If the answer is "the whole system goes down," you have a single point of failure. Eliminate it through redundancy (multiple instances), failover (hot standby), or graceful degradation (the system works with reduced functionality).

Cascading failures

A failure in one component can cascade through the system. Service A calls Service B, which calls Service C. If C slows down, B's threads fill up waiting for C's responses, and B stops responding too, causing A to also fill up. Solutions include circuit breakers (stop calling failing services), bulkheads (isolate resources so one failing path does not consume all resources), timeouts (fail fast rather than hang), and load shedding (reject excess load rather than slow down for everyone).

Data corruption

Hardware failures, software bugs, and human errors can corrupt data. Detect it with checksums and data integrity verification. Recover from it with backups and replication. Kleppmann emphasizes that end-to-end data integrity checks (from the original write all the way through every transformation to the final read) are essential because corruption can occur at any layer.

Chaos engineering

Resilience that has never been tested is theoretical. Chaos engineering (pioneered by Netflix's Chaos Monkey) involves deliberately injecting failures in production: killing random instances, introducing network latency, corrupting messages, filling disks. This reveals weaknesses in your resilience before real failures do. Start small: kill one instance and see what happens. Gradually increase the scope.

16. Monitoring, Observability & Alerting

You cannot fix what you cannot see. Observability is your window into the running system. It is the difference between spending 10 minutes diagnosing an issue and spending 10 hours.

The three pillars

Metrics are numerical measurements over time: request rate, error rate, latency percentiles, CPU usage, memory usage, queue depth. They tell you what is happening. Logs are discrete event records with details: "User X attempted to login from IP Y at time Z and failed because password was incorrect." They tell you why it happened. Traces follow a single request across multiple services: "This request hit the API gateway, then the auth service, then the user service, then the database, and the database query took 800ms." They tell you where the problem is.

The four golden signals

Google's SRE book identifies four signals to monitor: latency (how long requests take, broken down by success and error), traffic (how many requests you are serving), errors (how many requests are failing), and saturation (how full your system is approaching capacity). If you monitor nothing else, monitor these four.

SLIs, SLOs, and SLAs

Service Level Indicators (SLIs) are the metrics you measure (e.g., p99 latency, availability percentage). Service Level Objectives (SLOs) are the targets you aim for (e.g., p99 latency below 200ms, 99.95% availability). Service Level Agreements (SLAs) are contractual commitments to customers with consequences if violated. Set SLOs slightly tighter than SLAs so you have a buffer.

Error budgets

If your SLO is 99.95% availability, your error budget is 0.05%: about 22 minutes of downtime per month. As long as you are within budget, you can ship features and take risks. When the budget is exhausted, you stop shipping and focus on reliability. This gives product and engineering teams a shared framework for balancing velocity and stability.

Alerting

Alert on symptoms (high error rate, high latency) not causes (high CPU). Users do not care about your CPU usage; they care about whether the site is slow. Every alert should be actionable: if there is nothing the on-call engineer can do about it at 3 AM, it should not wake them up. Pages are for critical issues requiring immediate action. Tickets are for important issues that can wait until business hours. Logs are for informational data that might be useful in investigations.

17. Deployment & Operations

How you deploy and operate the system determines how fast you can iterate and how quickly you recover from problems.

Deployment strategies

Rolling updates gradually replace old instances with new ones. Simple but if the new version has a bug, some users are affected before you notice. Blue-green deployment runs two identical environments: deploy to the inactive one, test it, then switch traffic. Fast rollback (just switch back) but requires double the infrastructure. Canary deployment routes a small percentage of traffic to the new version first. If metrics look good, gradually increase. This is the safest approach for large-scale systems.

Feature flags

Feature flags decouple deployment from release. You can deploy new code without enabling it for users, then gradually enable it for internal testers, then a percentage of users, then everyone. If something goes wrong, flip the flag instead of rolling back the deployment. This is enormously valuable for reducing deployment risk.

Database migrations

Schema changes are one of the riskiest parts of deployment. Always make migrations backward-compatible: new code should work with both the old and new schema. The expand-and-contract pattern works well: expand (add new column), migrate (backfill data and update code), contract (drop old column). Never make a destructive change in a single step.

Infrastructure as Code

Define your entire infrastructure in code (Terraform, Pulumi, CloudFormation). This gives you version control, code review, reproducibility, and the ability to spin up identical environments. If you cannot recreate your production environment from a script, you have a ticking time bomb.

Incident response and post-mortems

Have a clear incident response process: who gets paged, who coordinates, how you communicate status to users. After every significant incident, write a blameless post-mortem: what happened, what was the impact, what was the timeline, what was the root cause, and what are the action items to prevent recurrence. A blameless culture means people report problems instead of hiding them.

18. Cost

Cloud infrastructure is expensive, and costs scale with your system. Designing for cost efficiency from the start prevents budget surprises later.

Understand your cost drivers

Compute, storage, network transfer, and managed services are the big four. Within each, understand what drives cost. For compute, it is instance type and utilization. For storage, it is volume size, IOPS, and storage class. For network, data transfer between regions and out to the internet is far more expensive than within a single availability zone. Data transfer costs are the most commonly underestimated line item.

Right-sizing

Oversized instances waste money. Undersized instances cause performance problems. Monitor actual CPU, memory, and I/O utilization and resize accordingly. Reserved instances or savings plans offer 30 to 60 percent discounts in exchange for a 1 or 3 year commitment. Use them for baseline workloads that you know will persist. Use spot or preemptible instances for batch processing and fault-tolerant workloads at 60 to 90 percent discounts.

Storage tiering

Not all data needs fast, expensive storage. Most data follows a hot/warm/cold pattern: recent data is accessed frequently (SSD, high-performance storage), older data is accessed occasionally (standard storage), and historical data is rarely accessed (archival storage like S3 Glacier at a fraction of the cost). Implement lifecycle policies that automatically move data to cheaper tiers as it ages.

Cost at scale

Project your costs at 10x your current scale. Some costs scale linearly (storage), some sub-linearly (reserved capacity gets cheaper per unit), and some super-linearly (some managed services have pricing tiers that jump). Identify cost cliffs before you hit them.

19. Testing Strategy

Testing in distributed systems goes far beyond unit tests. You need to test correctness, performance, resilience, and data integrity across multiple services and failure modes.

The testing pyramid

Unit tests (fast, isolated, many of them) form the base. Integration tests (verify that services work together correctly) form the middle. End-to-end tests (simulate real user workflows across the entire system) are at the top. The pyramid shape matters: mostly unit tests, fewer integration tests, even fewer end-to-end tests. Inverting this creates slow, flaky test suites.

Load and stress testing

Load testing verifies the system handles expected traffic. Stress testing pushes beyond expected traffic to find the breaking point. Both are essential. Use tools like k6, Locust, or Gatling to simulate realistic traffic patterns. Test with production-like data volumes. A system that works with 1,000 records might fail with 10 million. Run load tests regularly, not just before launch, because performance can regress with new code.

Chaos testing

As described in the resilience section, deliberately inject failures. Kill instances, introduce network latency, fill disks, corrupt messages, and simulate datacenter outages. This is the only way to verify that your failover, circuit breakers, and redundancy actually work.

Contract testing

In a microservices architecture, services depend on each other's APIs. Contract tests verify that a service's API matches what its consumers expect, without needing to deploy the full system. Tools like Pact let consumer and provider teams independently verify compatibility. This catches breaking API changes before they reach production.

Data migration testing

Before running a migration in production, test it with a copy of production data. Verify row counts, data integrity, and that all edge cases are handled. Measure how long the migration takes: a migration that takes 10 minutes on test data might take 10 hours on production data and block deployments.

20. Evolution & Future-Proofing

Systems are never done. Requirements change, traffic grows, technologies evolve, and teams turn over. Design for change.

Schema and API evolution

As Kleppmann emphasizes, forward and backward compatibility in data encoding is crucial. New code must read old data, and old code must read new data during rolling deployments. Choose encoding formats that support this (Protobuf, Avro) and establish a schema registry. For APIs, version explicitly and publish deprecation timelines.

Modular architecture

Whether you choose a monolith or microservices, keep components loosely coupled with well-defined interfaces. This means you can replace the database behind one module without affecting others, swap a message queue implementation without changing business logic, and extract a component into a separate service when it makes sense. The key discipline is: communicate through interfaces, not through shared databases or internal data structures.

Data migration paths

At some point, you will need to migrate from one database to another, or from one service to another. The dual-write pattern (writing to both old and new systems) is tempting but dangerous because it is hard to keep both systems consistent. A safer approach is change data capture: write to the old system, capture the changes, and feed them to the new system. Run both in parallel, verify consistency, then switch reads to the new system.

Technical debt management

Track technical debt explicitly. Not all debt is bad; sometimes a quick hack that ships a feature today is the right business decision. But untracked debt accumulates silently until it slows development to a crawl. Maintain a tech debt register, prioritize paydown alongside feature work, and establish coding standards and review processes to prevent unnecessary debt accumulation.

Architecture decision records

Document every significant architectural decision: what was decided, what alternatives were considered, what trade-offs were accepted, and what context led to the decision. When the original engineers have moved on, these records prevent new engineers from either repeating mistakes or unknowingly undoing deliberate decisions. A simple template of title, context, decision, and consequences is sufficient.

Final Thought

No system needs every item on this checklist. A weekend project and a global financial platform are very different. The value of a comprehensive checklist is not that you implement everything, but that you consciously decide which items matter for your specific system and which you are deliberately skipping. The most dangerous gaps are the ones you did not know existed.

Design is the art of making informed trade-offs. This checklist ensures you are at least aware of the trade-offs you are making.