

# System Design

## Handbook



LEVEL UP  
CODING



# LEVEL UP CODING

## Copyright Notice

© 2026 Level Up Coding. All rights reserved.

This ebook was created to help engineers sharpen their system design skills.

Copying, distribution, or repackaging of this ebook, in whole or in part, is not permitted without consent.

If you're writing about system design, keep it original, don't copy from here.

For educational use only.

If you'd like to share it, please share **[blog.levelupcoding.com](https://blog.levelupcoding.com)** so others can grab their own copy, rather than distributing this file.

Thanks for respecting our work.

Published by Level Up Coding

**[www.levelupcoding.com](https://www.levelupcoding.com)**



# Preface

System design is not a checklist of patterns. It is the discipline of making tradeoffs under constraints.

Modern systems are shaped by competing forces: latency and consistency, availability and correctness, performance and complexity. The goal is rarely to find a perfect solution. It is to choose deliberately.

Most engineers become familiar with tools and architectures through experience. They learn, for example, how replication works, how caching affects latency, and how asynchronous systems decouple services. But without structured mental models, tradeoffs often remain implicit, disconnected, or only partially understood.

Architectural strength depends on making those trade-offs explicit and reasoning from durable mental models.

The *System Design Handbook* was created to provide that clarity. It brings foundational principles, architectural patterns, and tradeoff analysis into a structured, reusable reference rather than a linear read. Use it to:

- Clarify architectural decisions
- Evaluate trade-offs explicitly
- Recognize when to apply patterns and when not to
- Strengthen reasoning in interviews and production environments

The goal is not memorization. It is sharper reasoning.

This handbook is intended for engineers at all stages of their careers. For those early in their system design journey, it provides a structured foundation. For experienced engineers, it offers a concise way to revisit ideas, deepen knowledge, and reassess assumptions.

This is Version 1.0 (February 2026).

Future editions will expand its scope and coverage while preserving its clarity and precision.

Nikki Siapno  
*Level Up Coding*



# Table of Contents

## Foundational Principles

|                      |    |
|----------------------|----|
| • Availability       | 2  |
| • Reliability        | 4  |
| • Latency            | 7  |
| • Throughput         | 9  |
| • Scalability        | 11 |
| • Vertical Scaling   | 13 |
| • Horizontal Scaling | 15 |
| • CAP Theorem        | 18 |
| • Observability      | 20 |
| • Maintainability    | 22 |

## Network & Traffic Management

|                                    |    |
|------------------------------------|----|
| • TCP                              | 25 |
| • UDP                              | 28 |
| • HTTPS                            | 31 |
| • WebSockets                       | 34 |
| • Forward Proxy                    | 37 |
| • Reverse Proxy                    | 40 |
| • Load Balancing                   | 43 |
| • Content Delivery Networks (CDNs) | 46 |
| • API Gateway                      | 49 |
| • Connection Pooling               | 52 |

## API & Interface Design

|                 |    |
|-----------------|----|
| • REST          | 56 |
| • GraphQL       | 59 |
| • gRPC          | 62 |
| • Idempotency   | 65 |
| • Rate Limiting | 68 |

## **Data Layer & Persistence**

|                               |     |
|-------------------------------|-----|
| • ACID .....                  | 72  |
| • BASE .....                  | 74  |
| • SQL .....                   | 77  |
| • NoSQL .....                 | 80  |
| • Key-Value Databases .....   | 82  |
| • Document Databases .....    | 84  |
| • Wide-Column Databases ..... | 86  |
| • Graph Databases .....       | 88  |
| • Time-Series Databases ..... | 90  |
| • Vector Databases .....      | 92  |
| • Partitioning .....          | 94  |
| • Sharding .....              | 97  |
| • Read Replicas .....         | 100 |
| • Caching .....               | 102 |

## **Asynchronous & Distributed Communication**

|                                     |     |
|-------------------------------------|-----|
| • Synchronous Communication .....   | 106 |
| • Asynchronous Communication .....  | 108 |
| • Publish/Subscribe (Pub/Sub) ..... | 111 |
| • Message Queues .....              | 114 |
| • Streaming .....                   | 117 |

## **Security & Compliance**

|                               |     |
|-------------------------------|-----|
| • Encryption .....            | 121 |
| • Tokenization .....          | 124 |
| • JSON Web Tokens (JWT) ..... | 127 |
| • OAuth 2.0 .....             | 130 |
| • OIDC .....                  | 133 |
| • SSO .....                   | 136 |
| • SAML .....                  | 139 |



# Chapter 1

# Foundational Principles



# Availability

- i** Availability is the measure of how often a system is operational and accessible when needed. It's typically expressed as a percentage of uptime; for instance, "four nines" (99.99%) means less than an hour of downtime per year.

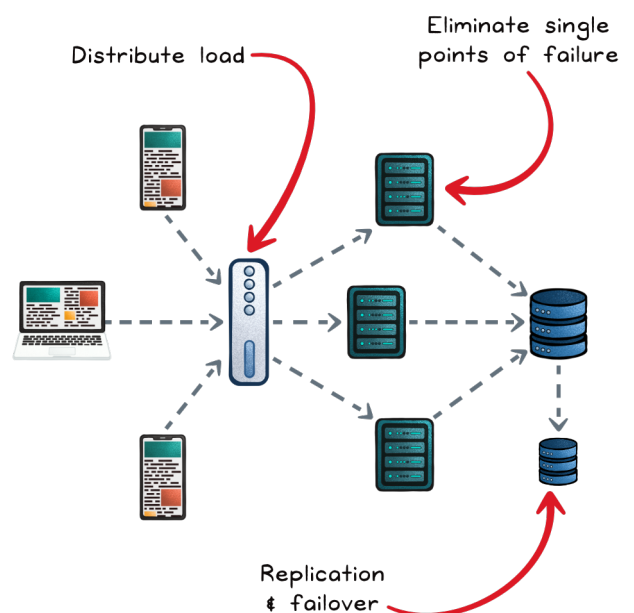
| Benefits                                                                                                                                                                                                                    | Tradeoffs                                                                                                                                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Fewer user-visible outages</li><li>• Smaller failure blast radius</li><li>• Safer deploys (rolling)</li><li>• Handles spikes better</li><li>• Faster detection + recovery</li></ul> | <ul style="list-style-type: none"><li>• Higher infrastructure spend</li><li>• Harder debugging/testing</li><li>• Consistency compromises</li><li>• Added latency/coordination</li><li>• Diminishing returns at "5 nines"</li></ul> |

## Why it matters

Downtime costs money, frustrates users, and erodes trust. In high-traffic or mission-critical systems like banking or healthcare, even brief outages can be catastrophic. Designing for availability ensures that services remain accessible despite hardware failures, network issues, or maintenance events.

## How it works

- **Eliminate single points of failure:** Use redundancy (extra servers, databases, and network paths) so no single component can bring down the system.
- **Use replication & failover:** Maintain backup instances (active-passive or active-active) that can instantly take over if a primary fails.
- **Distribute load:** Load balancers spread traffic across healthy servers and reroute requests away from failed ones.





- **Monitor continuously:** Health checks and metrics detect issues early and trigger automatic recovery or scaling.
  - **Design for graceful degradation:** Keep core features online even if non-critical parts fail.
- 

## Benefits

- Users experience **fewer interruptions**, which directly increases trust and repeat usage.
  - Redundancy shrinks the **blast radius** of failures so one bad node doesn't take the product down.
  - Teams can ship safer changes with **rolling deployments** and reduced maintenance downtime.
  - The system stays usable during traffic spikes because **load distribution** prevents single-node overload.
  - Better monitoring and automation enable **self-healing behavior**, reducing manual firefighting.
- 

## Tradeoffs

- You pay for **extra infrastructure** because redundancy means running more than you "need" on a good day.
  - More moving parts create **new failure modes**, especially around failover, networking, and coordination.
  - Distributed availability often forces **consistency compromises**, particularly under partitions (CAP tension).
  - Some approaches add **latency overhead**, like synchronous replication or consensus on writes.
  - Past a point you hit **diminishing returns**, where extra "nines" cost a lot and change little for users.
- 

## How to find the balance

Pushing for higher availability rapidly increases cost and complexity, with diminishing returns beyond "four nines." Ultra-high targets like "five nines" make sense only for mission-critical systems where downtime is unacceptable. For most products, aim for an availability level that balances reliability, cost, and real user impact.

# Reliability

**i** Reliability is a system's ability to **perform its intended function correctly and consistently over time**. It's not just about staying online; it's about doing the *right thing* every time, even under load or stress.

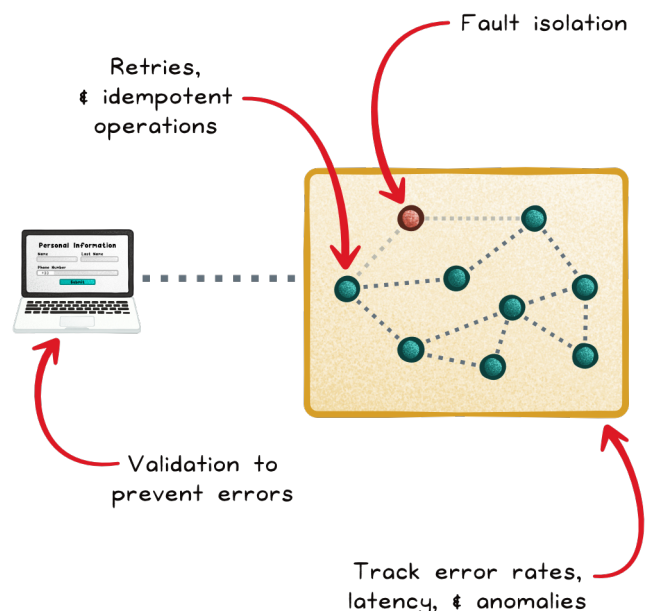
| Benefits                                                                                                                                                                       | Tradeoffs                                                                                                                                                                        |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Fewer failures</li><li>• User trust</li><li>• Predictable behavior</li><li>• Faster recovery</li><li>• Operational stability</li></ul> | <ul style="list-style-type: none"><li>• Higher complexity</li><li>• More infrastructure</li><li>• Latency overhead</li><li>• Higher cost</li><li>• Diminishing returns</li></ul> |

## Why it matters

Reliable systems minimize downtime, avoid data loss, and protect business continuity. Users equate reliability with trust: if a service repeatedly fails, they lose confidence fast. For businesses, high reliability reduces maintenance costs, safeguards revenue, and strengthens reputation.

## How it works

- **Error prevention:** Use validation, retries, and idempotent operations to ensure correctness under failure or retry conditions.
- **Fault isolation:** Design components so one failure doesn't cascade; contain issues within bounded contexts.
- **Replication integrity:** Replicate data safely and verify consistency across nodes, avoiding corruption or divergence.





- **Monitoring for correctness:** Track error rates, latency, and anomalies to detect reliability issues before they impact users.
  - **Testing & chaos engineering:** Continuously simulate faults to validate the system behaves correctly under failure.
  - **Recovery consistency:** When failures occur, recovery processes (backups, retries) must restore the *right* state, not just any state.
- 

## Benefits

- **Fewer outages and errors** mean users experience the system as dependable, not fragile.
  - Reliability builds **long-term user trust**, especially for systems people rely on daily.
  - Teams spend less time firefighting because failures happen **less often and recover faster**.
  - **Business continuity improves** as downtime-related revenue and reputation losses drop.
  - Predictable behavior enables **safer deployments and faster iteration** over time.
  - Strong reliability often leads to **higher availability** without constant manual intervention.
- 

## Tradeoffs

- Achieving high reliability adds **architectural complexity** through redundancy and failover logic.
- **Infrastructure costs increase** when running duplicate servers, replicas, or regions.
- Some reliability techniques introduce **latency overhead**, especially with synchronous replication.
- Systems become **harder to reason about** as failure modes multiply.
- Pushing for extreme reliability has **diminishing returns** beyond what users actually need.



## How to find the balance

Perfect reliability is impossible; every system will eventually fail or behave unexpectedly. The goal isn't to eliminate all errors but to **minimize their impact** and recover quickly.

Invest in higher reliability only where correctness truly matters (e.g. financial data, healthcare, safety systems), and design “good enough” reliability elsewhere to keep complexity manageable.

# Latency

**i** Latency is the time it takes for a system to complete a single operation or request, from start to finish. It's measured in milliseconds or microseconds and represents how "snappy" a system feels to a user.

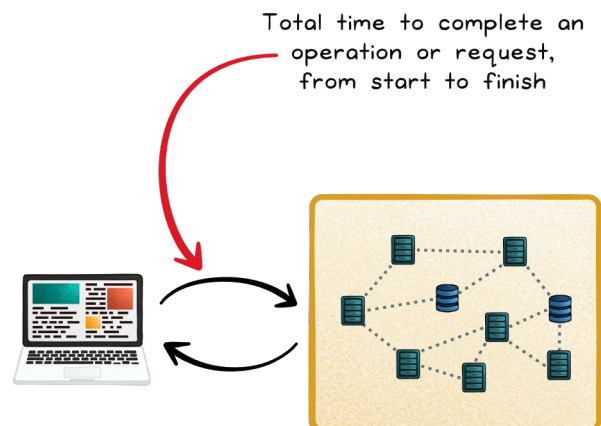
| Benefits                                                                                                                                                                         | Tradeoffs                                                                                                                                                                 | When to Use                                                                                                                                            |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Snappy user experience</li><li>• Real-time feedback</li><li>• Predictable response times</li><li>• Trust and perceived quality</li></ul> | <ul style="list-style-type: none"><li>• Higher infra cost</li><li>• Lower throughput efficiency</li><li>• Tail-latency tuning</li><li>• Added system complexity</li></ul> | <ul style="list-style-type: none"><li>• Interactive apps</li><li>• Strict SLAs</li><li>• User-facing APIs</li><li>• Time-sensitive workflows</li></ul> |

## Why it matters

Low latency is key to responsiveness. Even delays as small as 100 ms can impact engagement or conversion rates. In real-time systems like gaming, video calls, or trading, milliseconds can define success or failure. Many systems also have latency targets in SLAs that must be consistently met to maintain reliability and trust.

## How it works

- Every component in a system (network hops, databases, disk I/O, or code execution) adds delay.
- The total latency is the sum of these delays from the user action to the final response.
- Reducing latency often involves minimizing waiting time, cutting dependencies, and optimizing data paths.





## Benefits

- **Faster feedback loops** improve responsiveness and user experience.
  - **Real-time correctness** is possible for use cases like gaming, trading, or live collaboration.
  - Teams can enforce **clear SLAs** when response times are consistently bounded.
  - Low latency often simplifies debugging because delays are easier to localize and reason about.
- 

## Tradeoffs

- Optimizing for **ultra-low latency** often means over-provisioning resources to avoid queues.
  - Systems may sacrifice **throughput efficiency** when requests are handled immediately instead of batched.
  - Achieving low tail latency increases **architectural and operational complexity**.
  - Network calls, coordination, and consistency guarantees can introduce **hard latency floors**.
  - Returns diminish quickly once human-perceivable delays are already eliminated.
- 

## When to use it

- Real-time or interactive systems (games, live chat, streaming).
  - User-facing applications where speed impacts satisfaction or conversion.
  - APIs with strict response-time SLAs.
- 

## When not to use it

- Batch or background jobs where immediacy doesn't matter (ETL, data analytics).
- Workloads focused on volume, not per-task speed (e.g. report generation, backups).

# Throughput

**i** Throughput measures how much work a system can complete in a given amount of time; such as requests per second, transactions per second, or megabytes per hour. While latency is about *speed per task*, throughput is about *total capacity*.

| Benefits                                                                                                                                                                                                     | Tradeoffs                                                                                                                                                                                  | When to Use                                                                                                                                                                |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Higher system capacity</li><li>• Better hardware efficiency</li><li>• Handles traffic bursts</li><li>• Scales business output</li><li>• Predictable growth</li></ul> | <ul style="list-style-type: none"><li>• Added queueing latency</li><li>• More moving parts</li><li>• Harder debugging</li><li>• Weaker consistency</li><li>• Diminishing returns</li></ul> | <ul style="list-style-type: none"><li>• High-volume workloads</li><li>• Async or batch processing</li><li>• Streams and pipelines</li><li>• Scale &gt; immediacy</li></ul> |



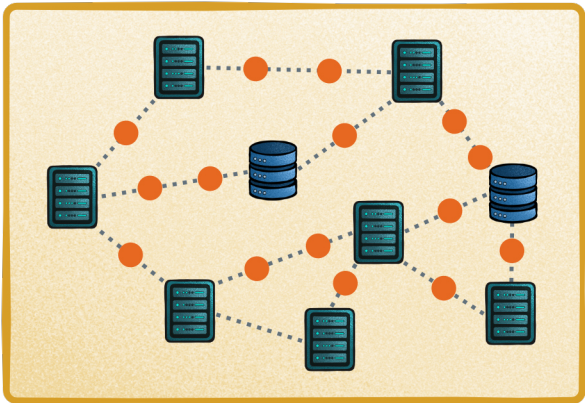
## Why it matters

High throughput means a system can handle large workloads efficiently, serving more users or processing more data without performance degradation. It's critical for scalability and cost efficiency, especially in systems that process streams, batches, or massive datasets.



## How it works

- Throughput is the rate at which operations complete across the system.
- It depends on factors like hardware capacity, concurrency, and resource utilization.
- Techniques like batching, parallelism, and load balancing are often used to raise throughput.



How much work a system can do in a given time

## Benefits

- **Higher capacity under load** allows the system to serve more users or process more data without degrading.
  - Throughput optimization improves hardware use, **reducing cost per request**.
  - **Burst tolerance increases** as queues and buffers absorb spikes.
  - High throughput enables **large-scale batch and streaming workloads** to complete in reasonable timeframes.
  - Teams gain **predictable scaling behavior**, making capacity planning and growth safer.
- 

## Tradeoffs

- Pushing for higher throughput often introduces **added latency** as work waits in queues or batches.
  - **Architectural complexity** increases with parallelism, coordination, and backpressure mechanisms.
  - Systems may require **relaxed consistency guarantees** to keep processing rates high.
  - Debugging bottlenecks becomes harder as **contention and saturation effects** emerge.
  - Beyond a point, **diminishing returns** set in as coordination overhead outweighs gains.
- 

## When to use it

- Data-intensive or bulk workloads (ETL, analytics, media processing).
  - Systems where total work done matters more than per-task speed.
  - Scenarios where scaling user volume or transaction rate is key to business value.
- 

## When not to use it

- Real-time or interactive apps where responsiveness defines the experience.
- User-facing APIs or UIs with strict SLA requirements.



# Scalability

**i** Scalability is a system's ability to handle increasing load (more users, data, or requests) by efficiently adding resources without sacrificing performance or reliability.

A scalable system grows gracefully as demand rises (and can often shrink when demand falls).

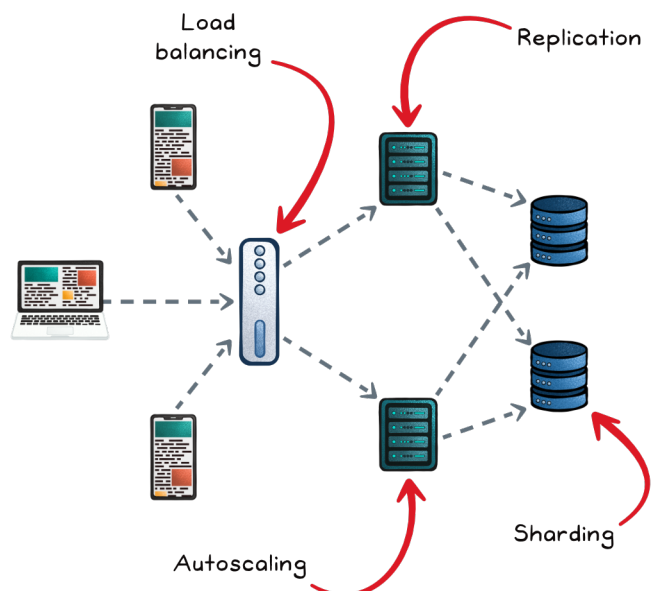
| Benefits                                                                                                                                                                                             | Tradeoffs                                                                                                                                                                                                               |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Handles growth smoothly</li><li>• Stable performance</li><li>• Higher availability</li><li>• Elastic cost scaling</li><li>• Faster business growth</li></ul> | <ul style="list-style-type: none"><li>• Added system complexity</li><li>• Network latency overhead</li><li>• Weaker consistency options</li><li>• Higher operations burden</li><li>• Risk of over-engineering</li></ul> |

## Why it matters

As applications grow, unscalable designs buckle under pressure: slow responses, downtime, and poor user experience. Scalable systems ensure smooth performance during viral spikes, seasonal surges, or long-term growth. They enable business agility, letting teams add features or expand globally without re-architecting from scratch.

## How it works

- **Load balancing:** Spread incoming requests across multiple servers so no single node becomes overloaded.
- **Replication:** Run duplicate services or database copies to share read load and increase fault tolerance.
- **Sharding:** Split large datasets across servers so storage and writes scale beyond one machine.



- **Autoscaling:** Automatically add or remove instances based on real-time metrics like CPU or request rate.
- 

## Benefits

- Handles **user and data growth** without degrading performance during peak demand.
  - Keeps **latency predictable at scale**, preserving a fast and consistent user experience.
  - Improves **availability and fault tolerance** by spreading load across multiple instances.
  - Enables **elastic cost control**, so infrastructure spend grows with usage, not ahead of it.
  - Supports **faster product evolution**, since capacity limits don't block feature launches.
  - Builds **organizational confidence**, allowing teams to scale the business safely.
- 

## Tradeoffs

- Introduces **architectural complexity**, especially with distributed components and coordination.
  - Scaling out often adds **network latency** compared to single-node designs.
  - Distributed systems may require **weaker consistency guarantees** to scale effectively.
  - Higher scale can mean **higher baseline infrastructure costs**, even when idle.
  - Requires **strong observability and ops maturity** to debug and operate reliably.
  - Over-scaling too early leads to **premature optimization and wasted effort**.
- 

## How to find the balance

Scale only as far as your current and near-future demand requires. Start simple, optimize your monolith and use caching or replicas before adding complex distributed systems. As growth or SLAs demand it, introduce horizontal scaling and elasticity, but always balance scalability's cost and complexity against its real business value.

# Vertical Scaling

**i** Vertical scaling (or “scaling up”) means boosting the power of a single machine by adding more CPU, memory, or storage. Instead of adding more servers, you make one server stronger.

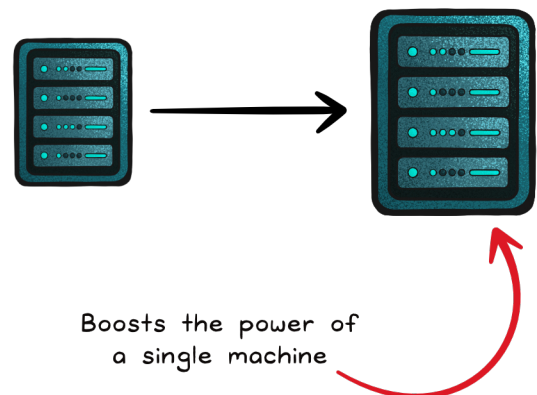
| Benefits                                                                                                                                                                                              | Tradeoffs                                                                                                                                                                                       | When to Use                                                                                                                                                                                                  |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Simple architecture</li><li>• Strong consistency</li><li>• Low initial complexity</li><li>• Low-latency local access</li><li>• Fast short-term wins</li></ul> | <ul style="list-style-type: none"><li>• Hard scaling ceiling</li><li>• Single point of failure</li><li>• Downtime to scale</li><li>• High-end cost curve</li><li>• Vendor constraints</li></ul> | <ul style="list-style-type: none"><li>• Small or early systems</li><li>• Legacy or monoliths</li><li>• Predictable traffic</li><li>• ACID-heavy workloads</li><li>• Acceptable maintenance windows</li></ul> |

## Why it matters

It's the simplest path to better performance; no architectural overhaul required. Many legacy, monolithic, or stateful systems can't easily run across multiple machines, so scaling up allows them to handle more users or data without a full redesign.

## How it works

- Increase machine specs (CPU, RAM, disk speed).
- In cloud environments, resize the instance type (e.g. AWS EC2 t3.medium → m5.2xlarge).
- The system is briefly taken offline to apply the upgrade.
- Once back online, the single server handles more load.



## Benefits

- **Simplicity of architecture** keeps systems easier to reason about, deploy, and debug.
  - **Strong consistency and shared memory** are easier to maintain on a single node.
  - Certain workloads see **lower latency** when everything runs in-process without network hops.
  - Many **legacy or monolithic systems** can scale without costly rewrites or re-architecture.
  - Small teams benefit from **lower operational overhead** early on.
- 

## Tradeoffs

- You eventually hit **hard limits on machine size**, even in the cloud.
  - Costs grow **non-linearly** as you move to high-end instances.
  - A single node creates a **single point of failure** unless extra redundancy is added.
  - Scaling often requires **planned downtime**, which limits elasticity.
  - Heavy reliance on large instances can increase **vendor lock-in**.
- 

## When to use it

- Early-stage systems with predictable, modest load.
  - Legacy or monolithic applications that can't run on multiple nodes.
  - Workloads that require strong consistency (e.g. relational databases).
  - Systems that can tolerate brief downtime during maintenance.
- 

## When not to use it

- Applications expecting rapid, unpredictable growth.
- Systems needing 24/7 uptime or high fault tolerance.
- Architectures designed for distributed or microservice operation.
- Environments where cost scaling and flexibility are key.



# Horizontal Scaling

- i** Horizontal scaling (or “scaling out”) means adding more machines or instances to share the workload instead of upgrading one. Each server handles part of the traffic, allowing the system to grow almost without limit by simply adding more nodes.

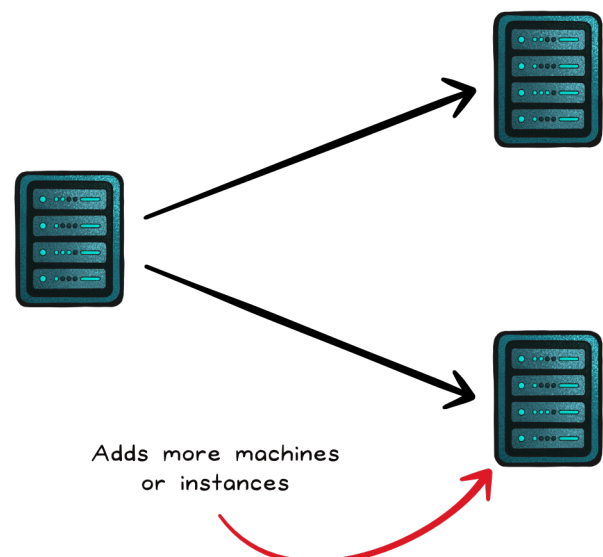
| Benefits                                                                                                                                                                                   | Tradeoffs                                                                                                                                                     | When to Use                                                                                                                                                                                       |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Near-infinite scale</li><li>• High availability</li><li>• Fault tolerance</li><li>• Elastic cost control</li><li>• Zero-downtime deploys</li></ul> | <ul style="list-style-type: none"><li>• Distributed complexity</li><li>• Eventual consistency</li><li>• Network latency</li><li>• Higher ops burden</li></ul> | <ul style="list-style-type: none"><li>• Rapid or spiky growth</li><li>• High uptime needs</li><li>• Parallelizable workloads</li><li>• Cloud-native systems</li><li>• Global user bases</li></ul> |

## Why it matters

It's how modern systems achieve massive scale and high reliability. By distributing load across multiple nodes, horizontal scaling avoids single points of failure and enables services to serve millions of users simultaneously.

## How it works

- Add more servers or containers behind a load balancer.
- Requests are distributed evenly across instances.
- Auto-scaling tools (e.g. AWS Auto Scaling Groups, Kubernetes) add or remove nodes as demand changes.
- Since new nodes can join live, scaling usually happens with zero downtime.



## Benefits

- **Near-unbounded scalability** makes it possible to keep growing by adding more nodes instead of redesigning the system.
  - Built-in **fault tolerance and redundancy** mean one server failing doesn't take the whole system down.
  - **Elastic capacity** lets you scale out during traffic spikes and scale back in to reduce costs.
  - Spreading nodes across regions improves **availability and user-perceived latency**.
  - Teams gain **deployment safety and velocity** through rolling updates and zero-downtime releases.
  - Using many standard machines often delivers better **cost efficiency at scale** than oversized hardware.
- 

## Tradeoffs

- Operating a distributed system introduces **architectural and operational complexity** that requires stronger tooling and expertise.
  - Applications must often be redesigned around **statelessness and partitioning**, which can slow delivery.
  - **Weaker consistency guarantees** are common, requiring teams to handle replication delays or eventual consistency.
  - Network hops between nodes add **latency and coordination overhead** compared to single-machine execution.
  - The initial setup can be **overkill at small scale**, increasing cost before traffic justifies it.
  - Ongoing systems demand **more monitoring, automation, and incident management** effort.
- 

## When to use it

- Applications expecting rapid or unpredictable growth.
- Cloud-native or microservices architectures.
- Services that require high uptime and resilience.
- Workloads that can run in parallel, like data processing or streaming.
- Systems with global users needing low-latency access across regions.



## When not to use it

- Small systems with stable, predictable load.
- Legacy or stateful applications that can't easily be distributed.
- Teams without the expertise or resources to manage distributed complexity.

# CAP Theorem

- i** CAP Theorem (Consistency, Availability, Partition Tolerance) states that a distributed system cannot guarantee all three properties at the same time. When a network partition occurs, the system must choose between being *consistent* (all nodes show the same data) or *available* (it continues responding to requests).

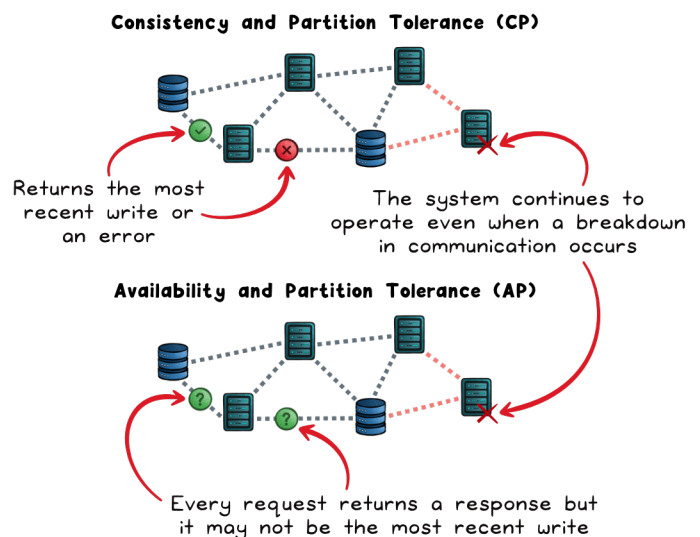
| Benefits                                                                                                                                                                                    | Tradeoffs                                                                                                                                                                                  |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Predictable failure behavior</li><li>• Clear design constraints</li><li>• Better database choices</li><li>• Shared architectural language</li></ul> | <ul style="list-style-type: none"><li>• Reduced uptime or stale data</li><li>• Added recovery complexity</li><li>• Harder operations and testing</li><li>• No “perfect” solution</li></ul> |

## Why it matters

CAP Theorem explains why distributed systems behave differently under failure. It forces architects to make conscious trade-offs: should the system return accurate but possibly delayed data, or keep serving users even if data may be stale? This theorem underpins the design of databases, caches, and microservices that operate across unreliable networks.

## How it works

- **Consistency (C):** Every read returns the most recent write, or an error.
- **Availability (A):** Every request gets a (non-error) response, even if it's outdated.
- **Partition Tolerance (P):** The system continues working despite lost or delayed messages between nodes.
- During a network partition, a system can't have both C and A. It either delays operations (CP) or continues with possible inconsistencies (AP).





## Benefits

- **Clear failure-mode thinking** helps teams design for outages.
  - Systems built with CAP in mind degrade in **controlled and intentional ways**.
  - It improves **data trust** by making correctness vs uptime an explicit choice.
  - CAP provides a shared language for **architectural decisions** across engineering and product teams.
  - It enables better **tool and database selection** based on real workload needs.
- 

## Tradeoffs

- Preserving consistency often means **reduced availability** during partitions.
  - Favoring availability introduces **temporary inconsistency** that must be resolved later.
  - Misunderstanding CAP can lead to **over-engineered or brittle architectures**.
  - AP systems require **conflict resolution logic**, which increases engineering effort and risk.
  - CAP decisions add **operational complexity**, especially around failover and recovery.
- 

## When to use it

- **CP:** Where correctness outweighs uptime; like financial transactions.
  - **AP:** Where responsiveness is more important than immediate consistency; like social feeds.
  - **Hybrid:** Most modern systems mix both; CP for critical paths, AP for non-critical ones.
- 

## When not to use it

- Don't apply CAP rigidly; it matters **only during partitions**.
- Don't assume you can "turn off" partition tolerance; real distributed systems can't avoid it.
- Avoid labeling entire systems CP or AP; different components can make different trade-offs.

# Observability

**i** Observability is the ability to understand a system’s internal state by examining its external outputs. It’s how engineers “see inside” distributed systems without manually probing or guessing what’s happening.

| Benefits                                                                                                                                                                                        | Tradeoffs                                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Faster debugging</li><li>• Clear root causes</li><li>• Lower downtime</li><li>• Better performance insight</li><li>• Shared system visibility</li></ul> | <ul style="list-style-type: none"><li>• Data volume &amp; cost</li><li>• Tooling complexity</li><li>• Performance overhead</li><li>• Alert noise risk</li><li>• Ongoing operations effort</li></ul> |



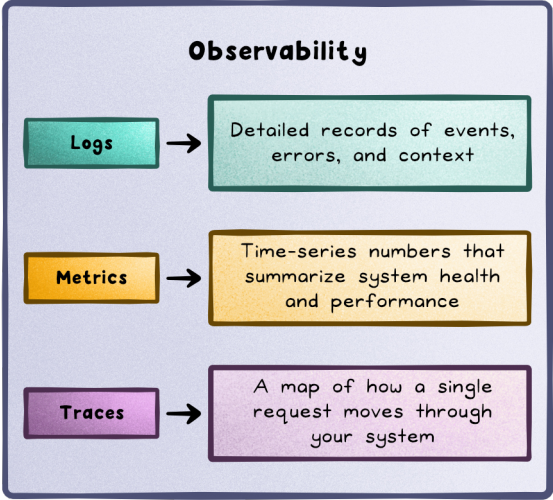
## Why it matters

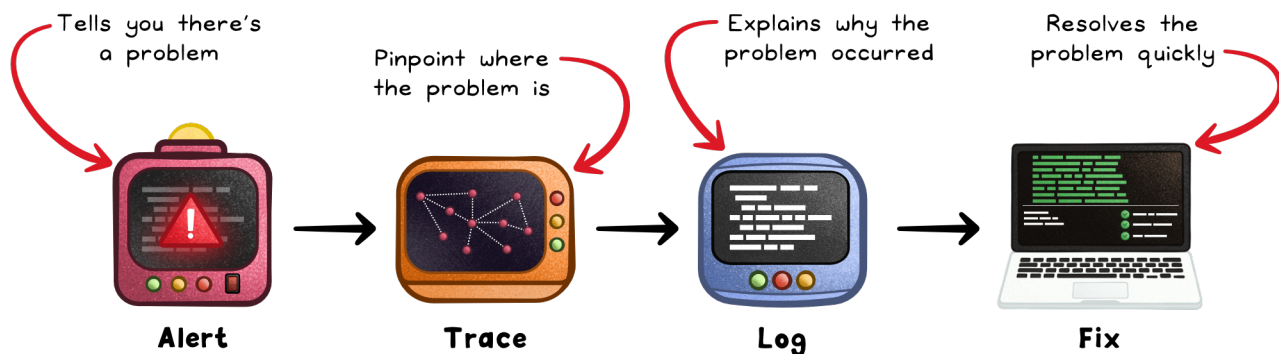
Modern software runs across dozens of services and environments, making failures hard to diagnose. Observability turns that complexity into clarity, enabling teams to detect, explain, and resolve issues faster. It transforms unknowns (“Why is checkout slow?”) into actionable insights and keeps systems reliable, performant, and transparent.



## How it works

- **Logs** record detailed events and errors for context.
- **Metrics** quantify health and performance over time (e.g. latency, CPU).
- **Traces** follow a request end-to-end across services, revealing bottlenecks.
- Together, these telemetry pillars give a complete view: metrics alert you to problems, traces show where, and logs explain why.





## ★ Benefits

- **Faster root cause analysis** because teams can move from symptoms to causes without guesswork.
- **Better performance optimization** by exposing slow paths, bottlenecks, and inefficient dependencies.
- **Reduced MTTR and downtime**, as issues are diagnosed directly in production.
- Safer **rapid deployments**, because regressions surface immediately after release.
- Stronger **system reliability and user trust**.

## ⚖ Tradeoffs

- **High data volume and cost**, especially from logs and high-cardinality metrics at scale.
- Instrumentation adds **runtime overhead**, which must be managed through sampling and tuning.
- Increased **architectural and tooling complexity**, particularly in distributed systems.
- Ongoing **operational burden** to maintain schemas, dashboards, alerts, and retention policies.
- Poorly designed setups can create **noise and alert fatigue**, hiding real problems.

## 🎯 How to find the balance

Go far enough with observability to see and solve real problems; but not so far that you drown in data. Collect telemetry tied to your reliability goals, focusing on critical paths and actionable signals instead of everything that moves. The sweet spot is visibility that scales with your system's complexity, giving clarity without clutter.

# Maintainability

**i** Maintainability measures how easily a system can be understood, modified, or extended over time without breaking other parts. It's about designing systems that evolve smoothly as requirements change and teams grow.

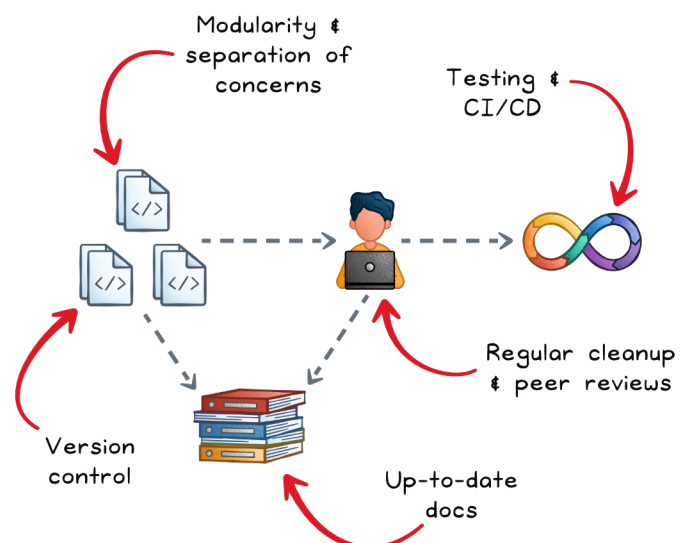
| Benefits                                                                                                                                                                      | Tradeoffs                                                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Safer changes</li><li>• Faster iteration</li><li>• Easier debugging</li><li>• Faster onboarding</li><li>• Long-term agility</li></ul> | <ul style="list-style-type: none"><li>• Upfront design cost</li><li>• Some performance overhead</li><li>• Operational complexity</li><li>• Risk of over-engineering</li><li>• Ongoing discipline required</li></ul> |

## Why it matters

Software rarely stays still: features evolve, bugs appear, and teams change. A maintainable system reduces long-term costs and risk by making updates fast, safe, and predictable. It keeps technical debt low, accelerates iteration, and ensures the product can grow without collapsing under its own complexity.

## How it works

- **Modularity & separation of concerns:** Organize code into clear layers or services so changes stay localized.
- **Simplicity & consistency:** Write clear, readable code and use standard patterns over cleverness.
- **Testing & CI/CD:** Automated tests catch regressions early; CI pipelines enforce code quality.



- **Refactoring & reviews:** Regular cleanup and peer reviews prevent decay and spread knowledge.
  - **Documentation & tooling:** Up-to-date docs, version control, and monitoring make debugging and updates straightforward.
- 

## Benefits

- **Changes stay localized**, so adding a feature or fixing a bug doesn't ripple unpredictably through the system.
  - Engineers move faster because **the system is easier to reason about**, not a maze of hidden dependencies.
  - **Reliability improves over time** since issues are easier to diagnose, fix, and verify with tests.
  - Teams onboard faster when **structure and intent are obvious** from the code and docs.
  - Long-lived systems avoid rewrites because **evolution is cheaper than replacement**.
  - **Developer morale** improves when working in a codebase that feels safe instead of fragile.
- 

## Tradeoffs

- Strong maintainability often adds **upfront design and testing cost**.
  - Abstractions and layering may introduce **small performance overheads**.
  - Maintainability requires **continuous effort**, not a one-time architectural decision.
  - Distributed designs that improve local maintainability can increase **operational and debugging complexity**.
  - Over-investing in future flexibility risks **over-engineering**, making systems harder to understand today.
- 

## How to find the balance

Invest in maintainability early, but match the effort to the system's lifespan and complexity. For long-lived, mission-critical, or frequently changing systems, make it a top priority with strong architecture, testing, and documentation. For short-lived prototypes or throwaway experiments, lighter practices are fine, but be cautious, since "temporary" code often becomes permanent.



## Chapter 2

# Network & Traffic Management

# TCP

- i** TCP is a **connection-oriented, reliable, ordered transport protocol** that sits on top of IP. It provides applications with the abstraction of a continuous, error-free byte stream; even though the underlying network is unreliable.

| Benefits                                                                                                                                                | Tradeoffs                                                                                                                                                      | When to Use                                                                                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Reliable delivery</li><li>• Ordered data stream</li><li>• Built-in retries</li><li>• Network-friendly</li></ul> | <ul style="list-style-type: none"><li>• Setup latency</li><li>• Head-of-line blocking</li><li>• Congestion throttling</li><li>• Per-connection state</li></ul> | <ul style="list-style-type: none"><li>• Web and APIs</li><li>• Databases</li><li>• File transfer</li><li>• Client-server defaults</li></ul> |

## Why it matters

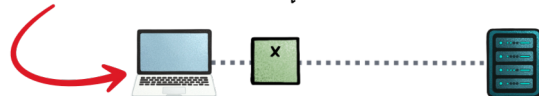
Distributed systems depend on predictable, correct data transfer. TCP ensures **no data is lost, duplicated, or delivered out of order**, which simplifies application logic dramatically. It forms the backbone of core internet traffic such as web browsing, email, file transfers, and APIs.

## How it works

- TCP establishes a connection using a **three-way handshake** before any data is sent.
- Once connected, data is split into segments with sequence numbers.
- The receiver sends acknowledgements, and any missing data is retransmitted.
- TCP continuously adjusts sending speed using **flow control and congestion control** so fast senders don't overwhelm slow receivers or congested networks.

### 1. SYN (Synchronize)

The client sends a TCP packet with the SYN flag set and an initial sequence number to the server



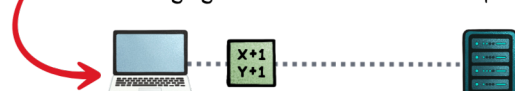
### 2. SYN-ACK (Synchronize-Acknowledge)

Replies with both the SYN and ACK flags set, it's an initial sequence number, and acknowledges the client's SYN packet



### 3. ACK (Acknowledge)

The client sends back an ACK packet to the server, acknowledging the server's SYN-ACK packet



## Benefits

- **Reliable, ordered delivery** means applications receive exactly the bytes that were sent, in the correct sequence.
  - Built-in **retransmission and error detection** remove the need for custom retry logic in most applications.
  - **Congestion control** protects the network and prevents one service from overwhelming others.
  - The **connection abstraction** simplifies application development by exposing a clean, stream-based interface.
  - TCP's ubiquity makes it **firewall-friendly and widely supported** across platforms, networks, and tools.
- 

## Tradeoffs

- **Connection setup latency** adds at least one round trip before data can flow.
  - Retransmissions introduce **head-of-line blocking**, where one lost packet stalls everything behind it.
  - **Congestion control throttling** can reduce throughput on high-latency or lossy networks.
  - Maintaining per-connection state adds **memory and CPU overhead** on busy servers.
  - TCP can be a poor fit for **real-time workloads** that prefer freshness over completeness.
- 

## When to use it

- You must deliver **all data exactly once and in order** (e.g. web pages, APIs, DB traffic).
- The application prefers correctness over latency (file transfer, email, configuration syncing).
- You want a **simple programming model**; just read/write a stream.
- Firewalls and enterprise networks must be traversed reliably (TCP/443 is universally allowed).



## When not to use it

- Real-time workloads where late data is worse than missing data (gaming, VoIP, live video).
- High-loss or mobile networks where TCP's HOL blocking hurts performance.
- Large numbers of short-lived messages where handshake cost dominates.
- When you're building on top of a protocol that already handles loss gracefully.

# UDP

- i** UDP is a lightweight, connectionless transport protocol that sends discrete messages (“datagrams”) without guaranteeing delivery, ordering, or retries. Think of it as **fire-and-forget networking**; fast, minimal, and hands-off.

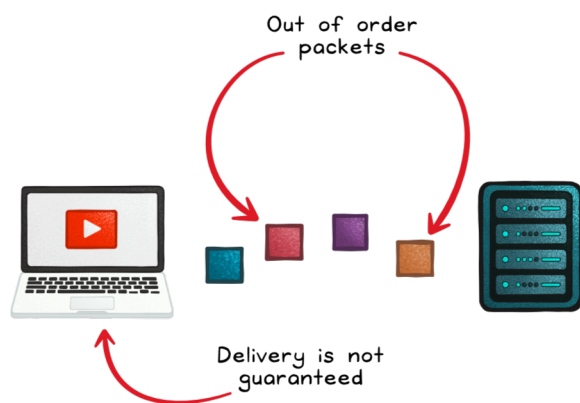
| Benefits                                                                                                                                                                                                   | Tradeoffs                                                                                                                                                                                             | When to Use                                                                                                                                                                                                     |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Ultra-low latency</li><li>• Minimal protocol overhead</li><li>• Scales to high message rates</li><li>• Enables multicast</li><li>• App-level flexibility</li></ul> | <ul style="list-style-type: none"><li>• No delivery guarantees</li><li>• Out-of-order packets</li><li>• No congestion control</li><li>• Network blocking risk</li><li>• More app complexity</li></ul> | <ul style="list-style-type: none"><li>• Real-time audio/video</li><li>• Online multiplayer games</li><li>• DNS and small queries</li><li>• Telemetry and streaming</li><li>• Custom transports (QUIC)</li></ul> |

## Why it matters

Some systems value **latency over reliability**. In real-time apps, waiting for a retransmission is worse than dropping a packet. UDP enables experiences like smooth video calls, online gaming, and rapid DNS lookups by removing the overhead inherent in TCP’s reliability mechanisms.

## How it works

- Sends standalone datagrams to an IP + port—no connection setup.
- Minimal 8-byte header (ports, length, checksum).
- No retransmission, reordering, congestion control, or handshakes; apps must handle these if needed.
- Supports multicast/broadcast so one message can reach many receivers efficiently.





## Benefits

- **Extremely low latency** because there's no connection setup or recovery logic slowing things down.
  - Systems can **keep moving under packet loss**, which is ideal for real-time audio, video, and games.
  - The protocol has **minimal overhead**, making it efficient for small, frequent messages.
  - UDP enables **one-to-many communication** through broadcast and multicast patterns.
  - It gives teams **full control at the application layer** to design custom reliability or ordering only where needed.
  - Infrastructure can handle **very high request rates** without tracking connection state.
- 

## Tradeoffs

- There are **no delivery guarantees**, so lost data is simply gone unless the application compensates.
  - **Out-of-order packets** can complicate application logic and data handling.
  - UDP provides **no built-in congestion control**, which can overload networks if misused.
  - Debugging issues is harder because **failures are silent** rather than explicit.
  - Some environments **block or throttle UDP traffic**, reducing reliability across networks.
  - Building reliability on top of UDP can **recreate TCP-like complexity** if you're not careful.
- 

## When to use it

- **Real-time media:** VoIP, video conferencing, live streaming.
- **Fast state updates:** multiplayer games, telemetry.
- **One-shot queries:** DNS lookups.
- **Multicast/broadcast distribution:** stock feeds, IPTV within a LAN.
- Custom protocols that implement their own reliability or ordering.



## When not to use it

- When data **must** arrive intact and in order (files, payments, API responses).
- When network environments block or throttle UDP.
- When implementing reliability yourself would effectively re-create TCP → use TCP instead.
- For large payloads that require fragmentation.

# HTTPS

**i** HTTPS (Hypertext Transfer Protocol Secure) is the secure version of HTTP. It runs HTTP traffic over a TLS (Transport Layer Security) connection to encrypt data in transit. This ensures that communication between client and server remains private, authentic, and tamper-proof.

| Benefits                                                                                                                                                                                            | Tradeoffs                                                                                                                                                                                    | When to Use                                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Encrypted data in transit</li><li>• Tamper detection</li><li>• Server identity verification</li><li>• User trust signals</li><li>• Safer defaults</li></ul> | <ul style="list-style-type: none"><li>• Handshake latency</li><li>• Certificate management</li><li>• CPU overhead</li><li>• Harder traffic inspection</li><li>• TLS misconfig risk</li></ul> | <ul style="list-style-type: none"><li>• Public websites</li><li>• APIs with auth</li><li>• Zero-trust networks</li><li>• Compliance-driven systems</li><li>• Internet-facing services</li></ul> |

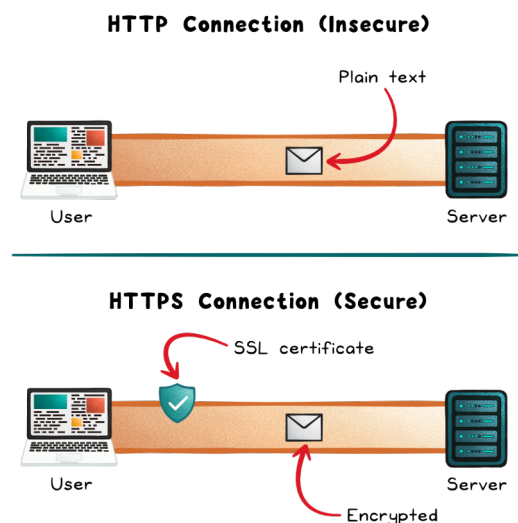
## Why it matters

Without HTTPS, anyone on the network can intercept or alter requests and responses.

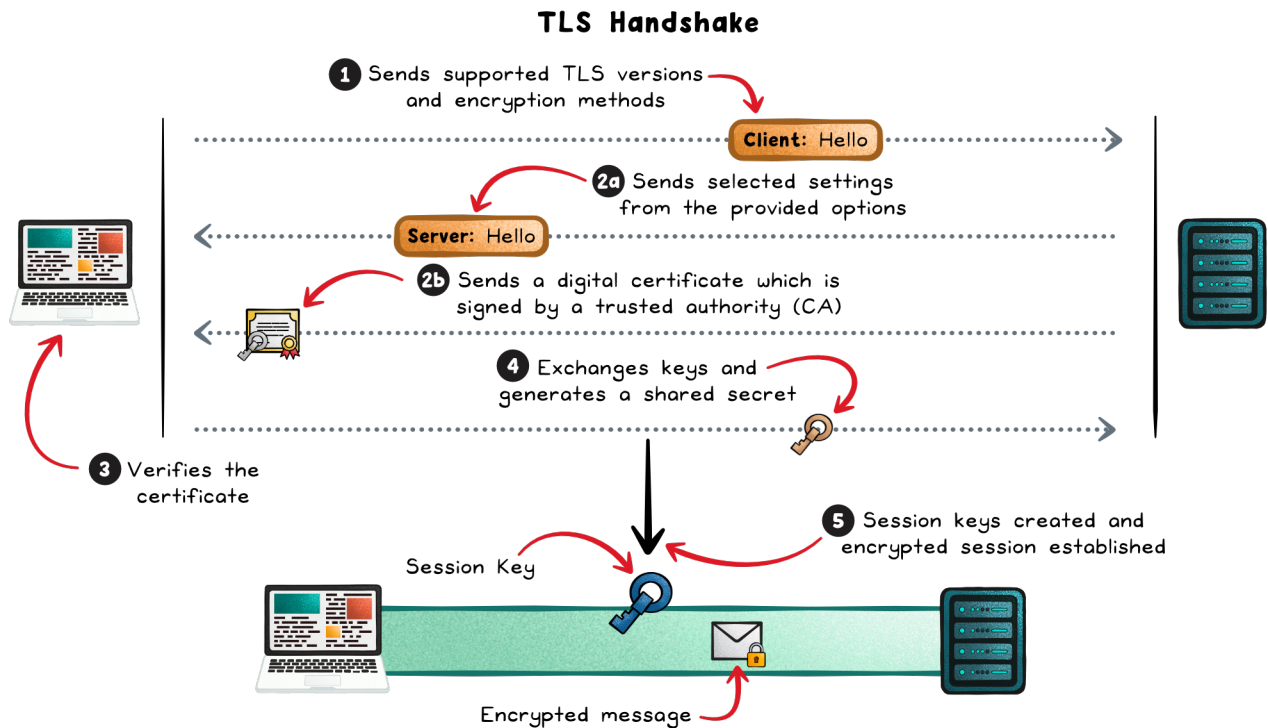
HTTPS prevents eavesdropping, data tampering, and impersonation by encrypting every exchange and verifying the server's identity through digital certificates. It's what makes modern web transactions safe over public networks.

## How it works

- A client connects to the server via TCP (port 443).
- The TLS handshake begins: client and server exchange supported ciphers, verify the server's certificate, and establish a shared secret key.
- Using symmetric encryption, all HTTP data is then encrypted and integrity-checked before transmission.



→ TLS 1.3 streamlines this handshake and enforces forward secrecy (old private keys can't decrypt past sessions).



## ★ Benefits

- Traffic is protected by **encryption in transit**, preventing attackers from reading sensitive data.
- Built-in **integrity checks** ensure responses can't be silently altered on the wire.
- Server identity verification via **certificates and certificate authorities** reduces phishing and impersonation risks.
- Users gain visible trust signals, improving **confidence and conversion rates**.
- Security guarantees are handled at the protocol level, reducing **application-level security burden**.

## ⚖ Tradeoffs

- The TLS handshake introduces **connection setup overhead**, especially for short-lived requests.
- Managing certificates adds **operational complexity**, including renewals and misconfiguration risk.

- Encryption consumes CPU, which can increase **infrastructure cost** at very high scale.
  - Debugging encrypted traffic requires tooling, reducing **observability by default**.
  - Misconfigured TLS can cause outages that are **hard to diagnose quickly**.
- 



## When to use it

- Anytime sensitive data (credentials, personal info, payment details) is transmitted.
  - In internal services where confidentiality or integrity matter (e.g. APIs, admin dashboards).
  - For any public-facing web service, browsers now flag HTTP as “Not Secure.”
- 



## When not to use it

You’d almost always use HTTPS. The only exception is in completely isolated internal networks that already have their own trusted encryption layer and no external exposure.



# WebSockets

- i** WebSockets is a protocol that enables *persistent, full-duplex* communication between a client and a server over a single TCP connection. Unlike HTTP (which is request/response and half-duplex), WebSockets allow both sides to send data anytime once connected.

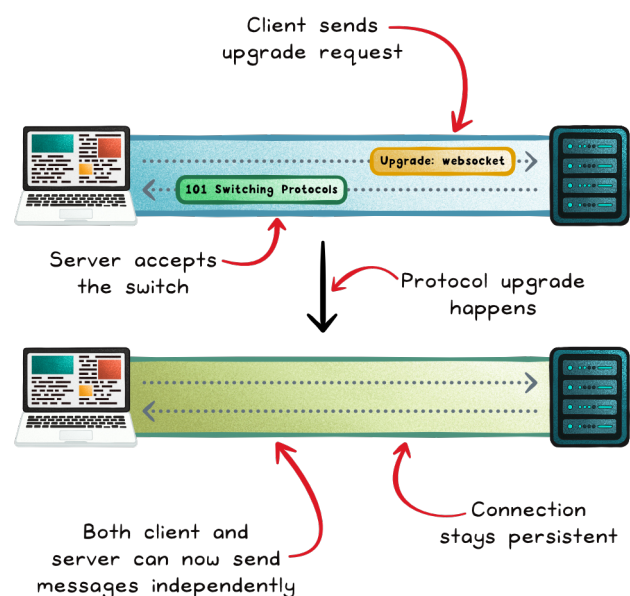
| Benefits                                                                                                                                                                                          | Tradeoffs                                                                                                                                                                                                              | When to Use                                                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Instant two-way updates</li><li>• Low latency</li><li>• Minimal per-message overhead</li><li>• Text + binary support</li><li>• Widely supported</li></ul> | <ul style="list-style-type: none"><li>• Stateful connections</li><li>• Harder horizontal scaling</li><li>• Manual reconnect logic</li><li>• Infra compatibility risks</li><li>• Higher operations complexity</li></ul> | <ul style="list-style-type: none"><li>• Real-time interaction</li><li>• High-frequency updates</li><li>• Collaborative features</li><li>• Live dashboards</li><li>• Multiplayer systems</li></ul> |

## Why it matters

Traditional HTTP requires clients to repeatedly poll for updates, adding latency and overhead. WebSockets solve this by keeping one connection open, allowing real-time data to flow both ways instantly. This makes them essential for experiences like live chat, collaborative editing, or streaming dashboards where every millisecond matters.

## How it works

- Starts as a HTTP handshake using the **Upgrade: websocket** header.
- The server replies with a **101 Switching Protocols** response, upgrading the connection.
- After the upgrade, HTTP is replaced with a lightweight framed protocol that stays open.



- Both client and server can send *text or binary* messages asynchronously without additional requests.
  - Heartbeat pings (*ping/pong*) keep the connection alive until one side closes it.
- 

## Benefits

- Enables **true two-way communication**, so servers can push updates instantly.
  - Reduces latency by avoiding repeated HTTP requests and header overhead.
  - Handles **high-frequency updates** efficiently over a single long-lived connection.
  - Supports both **text and binary messages**, making it flexible for many data types.
  - Improves user experience in interactive apps by making updates feel immediate and alive.
- 

## Tradeoffs

- **Persistent connections consume resources**, which makes large-scale concurrency harder to manage.
  - Scaling is more complex because connections are **stateful**, often requiring sticky sessions or routing layers.
  - There is no **built-in reconnection or message replay**, so reliability must be handled by the application or libraries.
  - WebSockets can introduce **infrastructure complexity**, as proxies and load balancers must explicitly support them.
  - Debugging and operating long-lived connections adds **ongoing operational burden** compared to stateless HTTP.
  - They are not ideal for **large media streaming**, where protocols like WebRTC perform better.
- 

## When to use it

- Real-time chat or messaging apps.
- Live dashboards or stock tickers.
- Collaborative editors (docs, whiteboards).
- Online multiplayer games.
- IoT control panels or live location tracking.



## When not to use it

- Simple one-way notifications (use Server-Sent Events).
- Infrequent updates (HTTP polling may suffice).
- Environments with limited WebSocket support or strict network constraints.

# Forward Proxy

**i** A forward proxy sits between clients and the internet, sending requests on the client's behalf to external servers.

| Benefits                                                                                                                                                                                               | Tradeoffs                                                                                                                                                                                                                   | When to Use                                                                                                                                                                                                                 |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Client IP masking</li><li>• Outbound traffic control</li><li>• Bandwidth savings via cache</li><li>• Centralized logging</li><li>• Simplified egress</li></ul> | <ul style="list-style-type: none"><li>• Central bottleneck risk</li><li>• Client configuration effort</li><li>• Limited threat protection</li><li>• Trust required in proxy</li><li>• Abuse and blacklisting risk</li></ul> | <ul style="list-style-type: none"><li>• Policy-controlled networks</li><li>• Privacy or anonymity needs</li><li>• Shared internet environments</li><li>• Repeated external access</li><li>• Geo-restricted access</li></ul> |



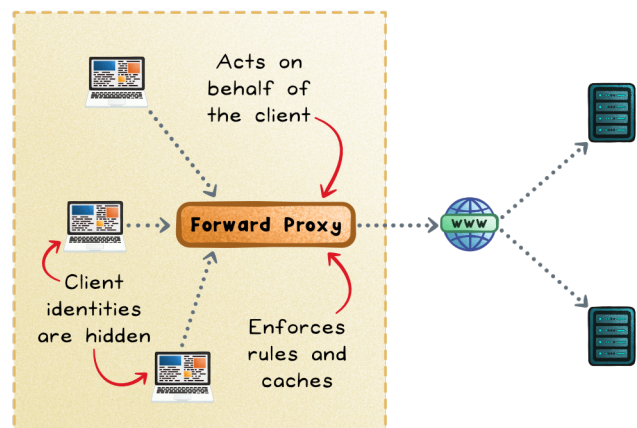
## Why it matters

Forward proxies give organizations and users control over **outbound traffic**. They help enforce policies, protect client identity, and reduce bandwidth usage when many clients access the same external resources.



## How it works

- Clients are explicitly configured to send requests to the proxy instead of directly to the internet.
- The proxy evaluates each request, applies rules (allow, block, log, cache), then forwards it to the destination server.
- Responses come back to the proxy first, which relays them to the client; while the external server only ever sees the proxy.



## Benefits

- Requests appear to come from the proxy, which improves **privacy and anonymity** for clients accessing external services.
  - Teams gain a single control point for **policy enforcement**, including content filtering, logging, and compliance tracking.
  - Frequently requested content can be served from cache, reducing **bandwidth costs and latency** for large networks.
  - Centralizing outbound traffic simplifies **network management** in offices, schools, and shared environments.
  - Proxies make it easier to access region-restricted services by routing traffic through **approved locations**.
- 

## Tradeoffs

- All traffic flows through one layer, so **proxy overload or failure** can degrade access for every client.
  - Clients must be configured correctly, which adds **setup and operational overhead** at scale.
  - A forward proxy does not automatically stop malware, creating **limited security protection** without additional tooling.
  - Users must trust the proxy operator, since **unencrypted traffic can be inspected or logged**.
  - Open or shared proxies are often abused, increasing the risk of **blacklisting by external sites**.
- 

## When to use it

- Enforcing internet usage policies in enterprise, education, or shared networks.
- Protecting client identity when accessing external services.
- Caching common external resources to reduce bandwidth usage.
- Accessing geo-restricted or censored content.



## When not to use it

- When clients cannot be reliably configured or managed.
- For inbound traffic protection or load balancing.
- When low latency is critical and an extra network hop is unacceptable.
- As a replacement for firewalls, malware scanning, or endpoint security.



# Reverse Proxy

**i** A reverse proxy sits in front of one or more servers and handles incoming client requests on their behalf. To the outside world, it looks like the real web server; clients never see or connect directly to the backend.

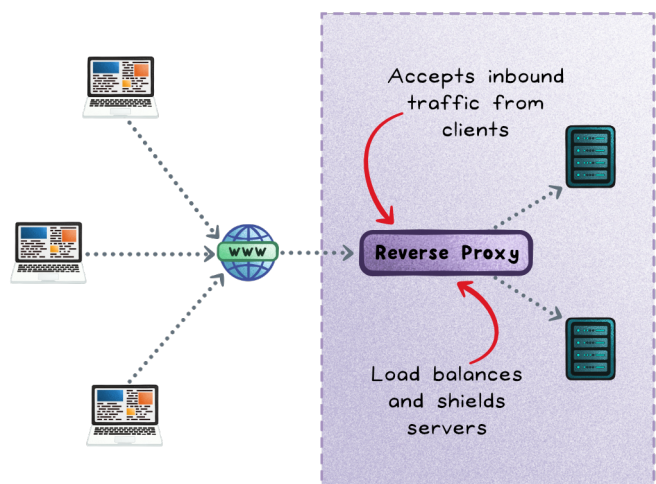
| Benefits                                                                                                                                                                                        | Tradeoffs                                                                                                                                                                                      | When to Use                                                                                                                                                                                              |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Backend isolation</li><li>• Horizontal scaling</li><li>• Central TLS &amp; routing</li><li>• Better performance</li><li>• Safer infra changes</li></ul> | <ul style="list-style-type: none"><li>• Single point of failure risk</li><li>• Extra network hop</li><li>• Operations complexity</li><li>• Proxy bottlenecks</li><li>• Attack target</li></ul> | <ul style="list-style-type: none"><li>• Multi-server systems</li><li>• Public-facing services</li><li>• Frequent deployments</li><li>• Security-sensitive apps</li><li>• Growing traffic loads</li></ul> |

## Why it matters

Reverse proxies protect and scale server infrastructure. They manage how users reach your services; balancing load, centralizing security, caching content, and hiding internal systems from direct exposure. This makes them essential for modern, high-traffic, and microservice-based architectures.

## How it works

- Clients send requests to the proxy (believing it's the actual server).
- The proxy decides which backend server should handle the request.
- It forwards the request internally, gets the response, and returns it to the client.
- Optional features include TLS termination, caching, compression, and authentication handling.



## Benefits

- Backend servers stay hidden behind the proxy, improving **security and attack isolation** without changing application code.
  - Traffic can be spread across multiple servers, enabling **horizontal scaling** without exposing internal topology.
  - Centralized TLS termination simplifies **certificate management** and reduces compute overhead on application servers.
  - Caching and compression at the edge improve **latency and perceived performance** for end users.
  - Teams can deploy, replace, or scale services safely because **clients remain decoupled** from server changes.
- 

## Tradeoffs

- The proxy can become a **single point of failure** if not deployed with redundancy and failover.
  - Every request adds an extra hop, introducing **additional latency**, even if usually small.
  - Operating and tuning the proxy introduces **ongoing operational complexity** and maintenance work.
  - If under-provisioned or misconfigured, the proxy itself can become a **performance bottleneck**.
  - Concentrating traffic means the proxy is a **high-value attack target** that must be carefully secured.
- 

## When to use it

- Scaling web apps or APIs across multiple backend servers.
- Protecting internal infrastructure from direct internet exposure.
- Serving static content efficiently (e.g. as part of a CDN).
- Managing certificates and routing logic for many services.
- Maintaining uptime during server updates or failovers.



## When not to use it

- For simple, single-server setups with low traffic; added complexity may not justify the benefit.
- Systems without redundancy. If the proxy fails, the entire service becomes unreachable.
- Don't use it as the sole security layer, reverse proxies don't fix insecure backend code.

# Load Balancing

**i** Load balancing distributes incoming requests across multiple servers or resources to keep any single one from being overwhelmed.

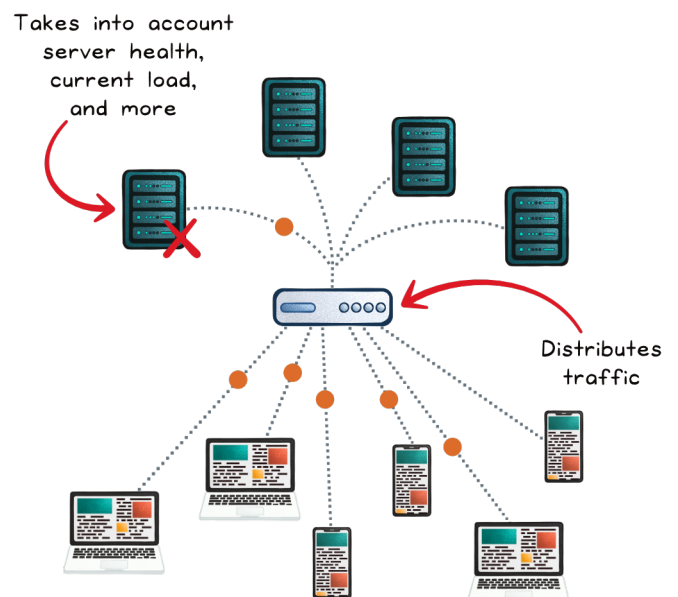
| Benefits                                                                                                                                                                                         | Tradeoffs                                                                                                                                                                                                        | When to Use                                                                                                                                                                       |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Even traffic distribution</li><li>• Higher availability</li><li>• Horizontal scaling</li><li>• Predictable latency</li><li>• Safer deployments</li></ul> | <ul style="list-style-type: none"><li>• Extra infrastructure</li><li>• Operational complexity</li><li>• Potential single point of failure</li><li>• Debugging difficulty</li><li>• DNS failover delays</li></ul> | <ul style="list-style-type: none"><li>• Multiple servers</li><li>• High uptime needs</li><li>• Traffic spikes</li><li>• Zero-downtime deploys</li><li>• Growing systems</li></ul> |

## Why it matters

Without load balancing, high-traffic systems would quickly bottleneck or fail when one server overloads. Load balancers keep systems fast, available, and scalable. They detect failures, reroute users to healthy servers, and let teams scale horizontally by simply adding more nodes.

## How it works

- Clients send requests to a single public endpoint (the load balancer).
- The load balancer selects a backend server using an algorithm (e.g. round robin or least connections).
- It monitors health and removes failed servers from rotation until recovery.
- Requests and responses usually pass entirely through the balancer (proxy mode) or the server responds directly to the client (a less common approach).



## Benefits

- It prevents **hot spots and overload**, keeping response times predictable even under traffic spikes.
  - Systems gain **higher availability** because failed servers are detected and bypassed automatically.
  - Teams can scale horizontally by **adding servers without changing clients**.
  - Users experience **more consistent latency**, since work is spread instead of queued on one node.
  - Deployments and maintenance are safer because **traffic can be shifted gradually**.
  - It improves infrastructure efficiency by **maximizing utilization** of all available capacity.
- 

## Tradeoffs

- Introducing a load balancer adds **another component to operate and monitor**.
  - Poorly designed setups can create **a single point of failure** if the balancer isn't redundant.
  - Advanced routing (Layer 7) introduces **extra latency and CPU overhead**.
  - Sticky sessions can cause **uneven load distribution** when some users are much heavier than others.
  - Debugging becomes harder because **requests no longer map cleanly to one server**.
  - At global scale, DNS-based balancing can lead to **slow failover due to caching**.
- 

## When to use it

- When scaling horizontally across multiple servers.
- For systems requiring high availability and seamless failover.
- When optimizing for latency, throughput, or user experience.
- In globally distributed or microservices-based architectures.



## When not to use it

- For single-server or low-traffic systems where routing overhead outweighs benefits.
- When workloads are already partitioned (e.g. by user ID) and don't need dynamic balancing.
- For real-time workloads requiring direct peer-to-peer connections without intermediaries.



# Content Delivery Networks (CDNs)

**i** A Content Delivery Network (CDN) is a **globally distributed network of edge servers** that cache and deliver content from locations close to users.

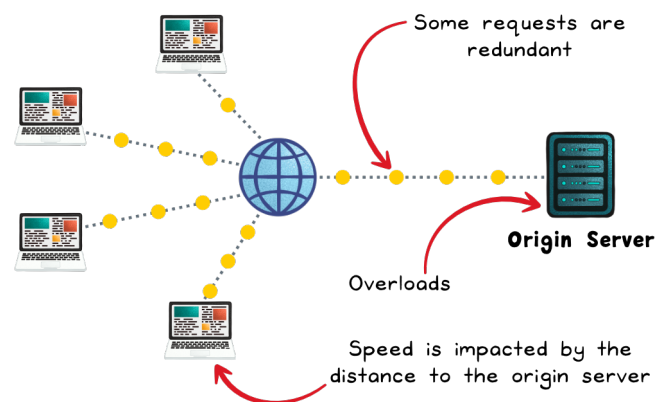
It sits between users and your origin servers to serve content faster and more reliably.

| Benefits                                                                                                                                                                                                     | Tradeoffs                                                                                                                                                                                   | When to Use                                                                                                                                                                                           |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Faster global delivery</li><li>• Origin load reduction</li><li>• Traffic spike absorption</li><li>• Built-in DDoS protection</li><li>• Higher availability</li></ul> | <ul style="list-style-type: none"><li>• Risk of stale content</li><li>• Extra dependency layer</li><li>• Harder debugging</li><li>• Cache miss overhead</li><li>• CDN usage costs</li></ul> | <ul style="list-style-type: none"><li>• Global user base</li><li>• Static-heavy workloads</li><li>• Viral or bursty traffic</li><li>• Public-facing apps</li><li>• Performance-sensitive UX</li></ul> |

## Why it matters

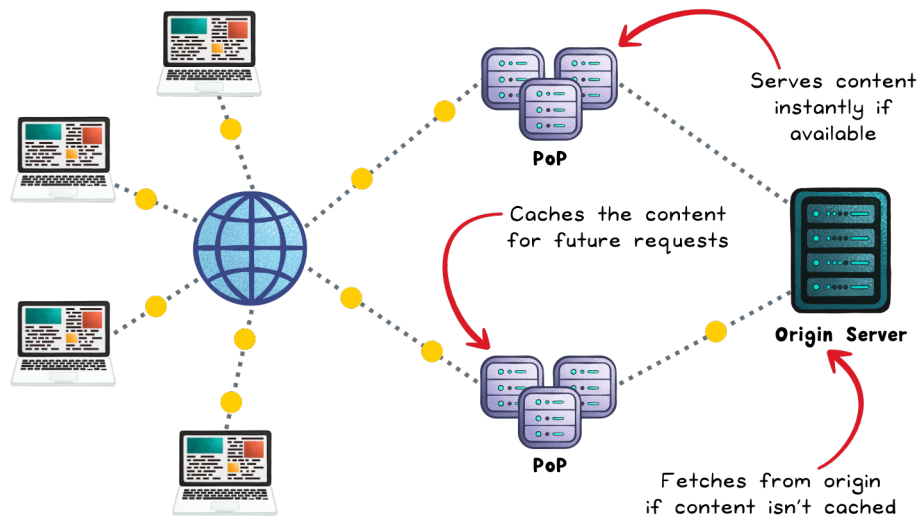
Distance is a major source of latency on the internet. Serving every request from a single region makes global performance slow and inconsistent.

CDNs reduce that distance, **offload work from origins**, and help systems survive traffic spikes, outages, and attacks.



## How it works

- When a user requests your website, DNS or anycast routing directs them to the nearest CDN Point of Presence (PoP).
- The PoP receives the request and checks if the requested file or response is already cached.
- If found in cache, the edge server serves it immediately.
- If not found, the edge retrieves the content from your origin server, stores a copy locally, and serves it to the user.



---

## ★ Benefits

- **Lower latency** because content is served from nearby edge locations instead of a distant origin.
- Origin servers stay healthier since **most requests never reach them**, even during traffic spikes.
- Global users get a more **consistent experience**, regardless of geography.
- Built-in **resilience and failover** means cached content can still be served during partial outages.
- CDNs absorb large volumes of traffic, improving **DDoS protection and security posture** by default.
- Teams avoid building global infrastructure themselves, which **reduces operational burden and time to scale**.

---

## ⚖ Tradeoffs

- Caching introduces **staleness risk** if TTLs and invalidation strategies are poorly designed.
- The CDN becomes an **additional dependency**, and outages can affect large portions of traffic.
- Debugging gets harder because failures may sit at the **edge, CDN config, or origin**, not just the app.
- Cache misses add a small **latency overhead** compared to direct origin requests.
- Costs can increase if **cache hit rates are low**, leading to double-paying for bandwidth.
- Dynamic or personalized content sees **diminishing returns** from CDN caching.



## When to use it

- Global audiences or users far from origin.
  - Ideal for images, CSS, JavaScript, fonts, videos, and downloadable files that don't change often.
  - Prevent origin overload during major events, product launches, or viral content surges.
  - When you want DDoS protection, rate limiting, or WAF policies closer to users to absorb attacks before they reach your infrastructure.
- 



## When not to use it

- Content that changes constantly (like stock tickers or chat messages).
- Apps requiring millisecond-level synchronization may be delayed by edge propagation.
- Intranets or region-locked apps with predictable traffic don't justify CDN complexity or cost.
- Sensitive, user-specific data should bypass public CDNs or use private edge configurations.

# API Gateway

**i** An API Gateway is a single entry point (a smart reverse proxy) that sits in front of your backend services and handles client requests on their behalf.

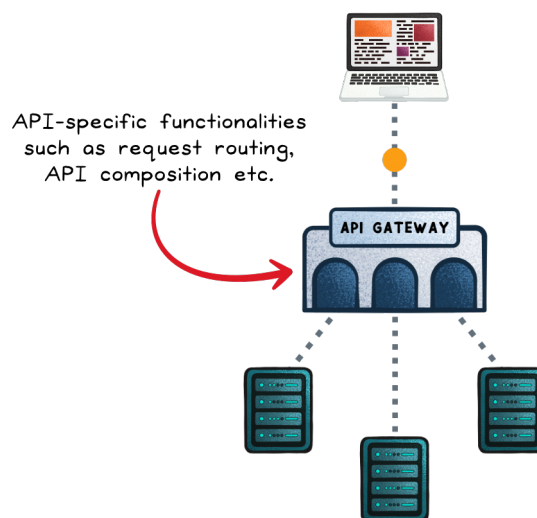
| Benefits                                                                                                                                                                                                                   | Tradeoffs                                                                                                                                                                                                                                       | When to Use                                                                                                                                                                                                                                                |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• One front door for APIs</li><li>• Central auth &amp; policy</li><li>• Backend protection</li><li>• Unified monitoring</li><li>• Safer rollouts</li><li>• Legacy bridging</li></ul> | <ul style="list-style-type: none"><li>• Potential single point of failure</li><li>• Extra latency hop</li><li>• Config sprawl risk</li><li>• Gateway “mini-monolith”</li><li>• Big security blast radius</li><li>• Added cost/on-call</li></ul> | <ul style="list-style-type: none"><li>• Many services + many clients</li><li>• Need consistent security rules</li><li>• Reduce client round-trips</li><li>• API product/partners</li><li>• Canary/version routing</li><li>• Legacy modernization</li></ul> |

## Why it matters

Without load balancing, high-traffic systems would quickly bottleneck or fail when one server overloads. Load balancers keep systems fast, available, and scalable. They detect failures, reroute users to healthy servers, and let teams scale horizontally by simply adding more nodes.

## How it works

- When a user requests your website, DNS or anycast routing directs them to the nearest CDN Point of Presence (PoP).
- The PoP receives the request and checks if the requested file or response is already cached.
- If found in cache, the edge server serves it immediately.



- If not found, the edge retrieves the content from your origin server, stores a copy locally, and serves it to the user.
- 

## Benefits

- Client apps stay simple because they talk to **one stable endpoint**.
  - Centralizing **authentication and authorization** reduces mistakes.
  - You can protect backends with **rate limiting and quotas** instead of hoping every service does it right.
  - Observability improves when **every call is measured** in one place (latency, errors, top consumers).
  - Safer rollouts become easier with **traffic splitting** at the edge.
  - You can modernize gradually via **protocol translation**.
- 

## Tradeoffs

- If you don't design it for high availability, the gateway becomes a **single point of failure** for your entire API surface.
  - Every request takes an extra hop, so there's **latency overhead** and capacity planning to do.
  - Configuration can grow into a brittle "rules jungle," creating **operational complexity** and misroute risk.
  - Stuffing workflows into the gateway turns it into a new monolith of **business logic bloat**.
  - Centralizing security also centralizes impact: a gateway bug can create a **large blast radius**.
  - Licensing/infrastructure and on-call effort add **real cost**, especially with heavy API management suites.
- 

## When to use it

- You have many microservices or public APIs and need a single access point.
- You want consistent authentication, throttling, and logging across APIs.
- You need to hide internal architecture changes from clients.
- You're modernizing legacy systems but want to expose them via REST/JSON.
- You plan to monetize or externally expose APIs through developer portals.



## When not to use it

- For small, static backends or prototypes where clients can safely talk to services directly, a gateway may slow iteration more than it helps.
- In systems like high-frequency trading or real-time gaming, even milliseconds of gateway processing can be too much.



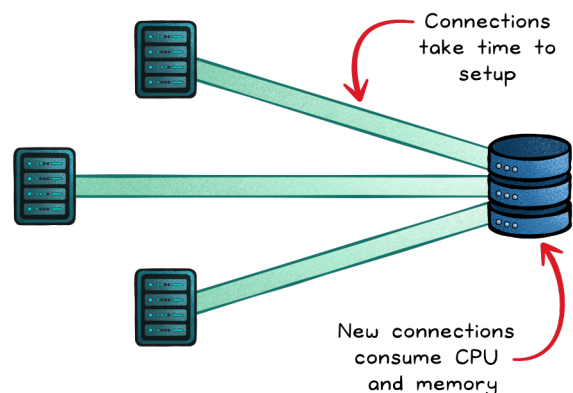
# Connection Pooling

**i** Connection pooling is a technique that reuses a fixed set of open connections instead of creating and closing a new connection for every request.

| Benefits                                                                                                                                                                                   | Tradeoffs                                                                                                                                                     | When to Use                                                                                                                                                                    |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Lower request latency</li><li>• Higher throughput</li><li>• Predictable load</li><li>• Backpressure control</li><li>• Stable performance</li></ul> | <ul style="list-style-type: none"><li>• Pool sizing complexity</li><li>• Idle resource overhead</li><li>• Risk of leaks</li><li>• Stale connections</li></ul> | <ul style="list-style-type: none"><li>• High concurrency</li><li>• Database access</li><li>• Microservices</li><li>• Connection limits</li><li>• Long-lived services</li></ul> |

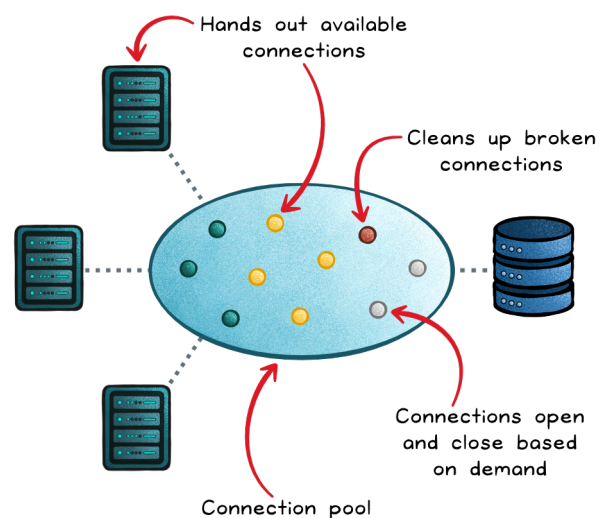
## Why it matters

Establishing and tearing down connections repeatedly is slow and CPU-intensive. Pooling eliminates that overhead, drastically improving latency and throughput in high-traffic systems. It also stabilizes backend resources by keeping connection counts predictable, preventing connection storms or server overload.



## How it works

- A **pool manager** maintains a set of open connections.
- When a request needs one, it's **checked out** from the pool.
- After use, the connection is **returned** instead of closed.
- Idle or stale connections are periodically **validated or replaced** to stay healthy.



## Benefits

- Requests complete faster because the system avoids paying the **connection setup cost** over and over again.
  - The system can handle more work with the same hardware since threads aren't stuck waiting on **handshakes and authentication**.
  - Downstream systems stay healthier because the pool enforces **predictable limits** instead of allowing connection floods.
  - During traffic spikes, pooling naturally applies **backpressure**, smoothing load instead of letting failures cascade.
  - Reusing warm connections often improves efficiency thanks to **cached session state** like prepared statements or open TCP flows.
  - Overall reliability improves because the system avoids **connection storms** that can destabilize databases and services.
- 

## Tradeoffs

- Finding the right pool size takes work, and **mis-sizing the pool** can either throttle throughput or overload dependencies.
  - Even unused connections consume memory and server resources, which means **idle overhead** is always present.
  - If application code forgets to return connections, **leaks can silently exhaust the pool** and cause widespread failures.
  - Long-lived pools must actively manage **stale or broken connections** caused by restarts, timeouts, or network drops.
  - Operating a pool adds ongoing responsibility around **timeouts, validation, and monitoring**.
  - At very low traffic levels, the extra machinery can introduce **complexity without meaningful gains**.
- 

## When to use it

- Applications with **frequent or concurrent database or API calls** (e.g. web apps, microservices).
- Systems with **limited connection capacity** on the backend.
- **Long-lived services** needing fast response to intermittent traffic.



## When not to use it

- **Low-throughput scripts** or batch jobs that run infrequently.
- **Serverless or short-lived** environments where connections can't persist between invocations.
- **Stateless or ultra-light connections** where setup cost is negligible.

# Chapter 3

# API & Interface Design

# REST

**i** REST (Representational State Transfer) is an architectural style for building web APIs using standard HTTP semantics and resource-based URIs. It relies on stateless, cache-friendly interactions and a uniform interface.

| Benefits                                                                                                                                                                        | Tradeoffs                                                                                                                                                                           | When to Use                                                                                                                                                         |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Simple HTTP model</li><li>• Stateless scaling</li><li>• Cache-friendly</li><li>• Broad compatibility</li><li>• Loose coupling</li></ul> | <ul style="list-style-type: none"><li>• Over/under-fetching</li><li>• Verbose payloads</li><li>• Versioning burden</li><li>• Weak typing</li><li>• Poor real-time support</li></ul> | <ul style="list-style-type: none"><li>• Public APIs</li><li>• CRUD domains</li><li>• Read-heavy traffic</li><li>• CDN leverage</li><li>• Default web APIs</li></ul> |

---

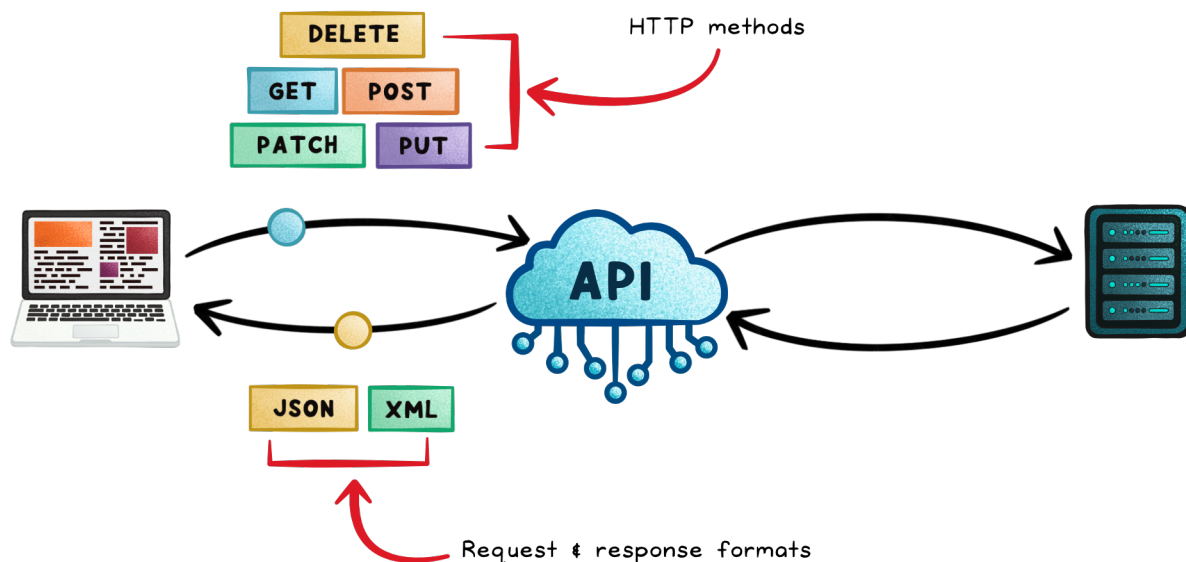
## Why it matters

REST became the default API style because it's simple, predictable, and universally compatible; any client that can make HTTP calls can use it. Its statelessness and caching model make horizontal scaling easy, while the resource-based approach keeps APIs intuitive for developers. REST is especially valuable when many different clients must integrate with your system.

---

## How it works

- Exposes **resources** (nouns) via clean URIs like `/users/123`.
- Uses **HTTP verbs** (GET, POST, PUT, PATCH, DELETE) to perform CRUD operations.
- Each request is **stateless**; authentication and context travel with every call.
- Servers return **representations** (typically JSON)
- Leverages **HTTP caching** (Cache-Control, ETags) to reduce load and latency.
- Uses standard **status codes** (200, 201, 404, 500) for predictable error handling.



---

## ★ Benefits

- **Simple, familiar semantics** make APIs easy to learn, test, and debug with standard HTTP tools.
- Stateless requests enable **horizontal scaling** without sticky sessions or shared server memory.
- Built-in HTTP caching improves **performance and cost efficiency** for read-heavy workloads.
- Loose coupling allows **clients and servers to evolve independently** as long as the contract stays stable.
- Broad compatibility supports **any language, platform, or device** that can speak HTTP.

---

## ⚖ Tradeoffs

- Fixed response shapes often lead to **over-fetching or under-fetching** data.
- Text-based JSON and HTTP headers add **payload and latency overhead** at high scale.
- API evolution requires careful **versioning strategies** to avoid breaking clients.
- Weakly enforced schemas can cause **runtime contract mismatches** if documentation drifts.
- Request-response semantics make **real-time updates** awkward without additional protocols.



## When to use it

- You're building **public or multi-client APIs** that must be easy to consume.
  - The domain maps naturally to **resources and CRUD**.
  - You benefit from **HTTP caching**, proxies, and CDNs.
  - Clients vary widely (web, mobile, backend), and **maximum interoperability** is needed.
  - Requirements are evolving and you need a **simple, extensible baseline**.
- 



## When not to use it

- Clients frequently need **complex, cross-resource queries** → consider GraphQL.
- You need **high-performance, low-latency, strongly typed** internal RPCs → consider gRPC.
- Workloads require **streaming or real-time bidirectional communication**.
- Payload sizes are extremely large or chatty interactions dominate.



# GraphQL

-  GraphQL is a strongly typed query language and runtime that lets clients request exactly the data they need through a single API endpoint. It replaces fixed REST responses with client-shaped JSON results.

| Benefits                                                                                                                                                                                                     | Tradeoffs                                                                                                                                                                                                        | When to Use                                                                                                                                                                   |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Precise data fetching</li><li>• Fewer network calls</li><li>• Faster frontend iteration</li><li>• Strong API contract</li><li>• Built-in real-time support</li></ul> | <ul style="list-style-type: none"><li>• Schema &amp; resolver complexity</li><li>• Harder caching</li><li>• Query abuse risk</li><li>• Performance tuning needed</li><li>• Tooling &amp; learning cost</li></ul> | <ul style="list-style-type: none"><li>• Complex, data-rich UIs</li><li>• Multiple client types</li><li>• Microservice aggregation</li><li>• Rapid product evolution</li></ul> |

---

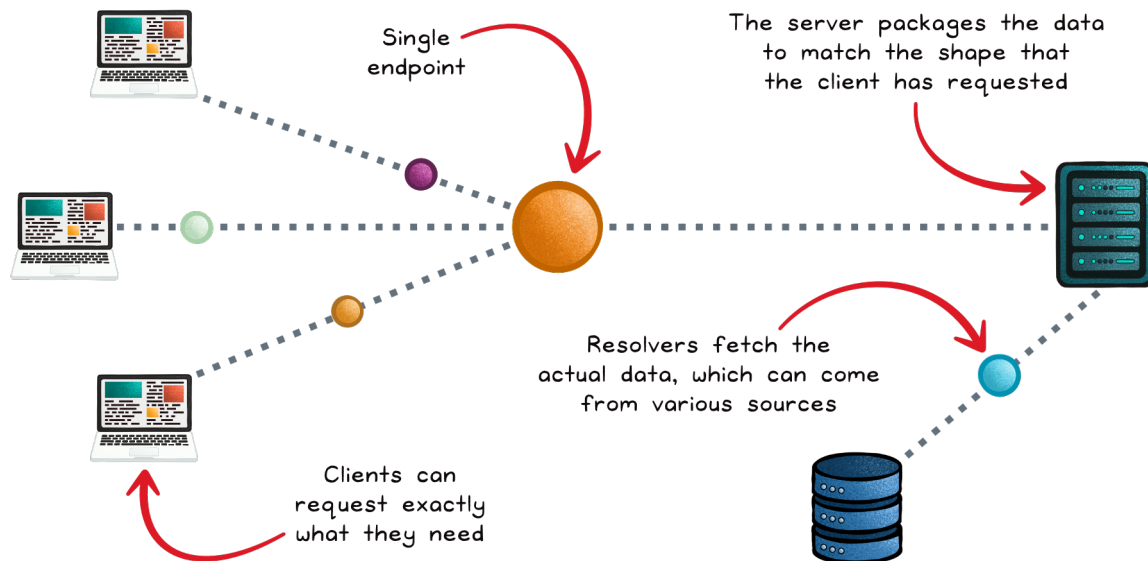
## Why it matters

Modern products pull data from many services, and REST often forces multiple endpoints or oversized responses. GraphQL solves this by giving clients precise control over fields and relationships, reducing round trips and payload size. It also accelerates frontend iteration; new UI needs rarely require new backend endpoints.

---

## How it works

- Clients send **queries** (reads), **mutations** (writes), or **subscriptions** (real-time streams).
- A **single /graphql endpoint** receives all requests.
- A **schema** defines types and relationships.
- **Resolvers** fetch data from databases, microservices, or third-party APIs and return a JSON response shaped exactly like the client query.



---

## ★ Benefits

- Clients avoid wasted bandwidth by **fetching exactly the fields they need**, no more and no less.
- Multiple resources can be retrieved in one request, which **reduces round trips and perceived latency**.
- Frontend teams move faster because **UI changes often require no new backend endpoints**.
- A **strongly typed schema** acts as a living contract and self-documenting API.
- GraphQL works well as an **aggregation layer over microservices**, simplifying client logic.
- Built-in subscriptions make **real-time features** easier to model consistently.

---

## ⚖ Tradeoffs

- Maintaining a global schema and resolvers adds **architectural and operational complexity**.
- Traditional HTTP and CDN caching is harder due to **dynamic queries and single endpoints**.
- Poorly designed resolvers can cause **N+1 query and performance issues**.
- APIs must defend against **expensive or deeply nested queries** to avoid overload.
- Teams face a **learning curve and tooling investment**, especially if new to GraphQL.
- For simple CRUD APIs, GraphQL can be **unnecessary overhead**.



## When to use it

- Screens or APIs that combine **multiple data sources** in one view.
  - Different clients (web, mobile) needing different data shapes.
  - High-latency environments where **fewer round trips** matter.
  - As a **BFF/gateway** over microservices.
  - Apps needing **real-time** updates via subscriptions.
- 



## When not to use it

- Simple CRUD services where REST is clearer and faster.
- Systems reliant on **heavy CDN caching**.
- Very high-throughput, low-latency operations where query parsing adds overhead.
- Teams without capacity to maintain schema discipline, caching strategies, and query-cost protections.

# gRPC

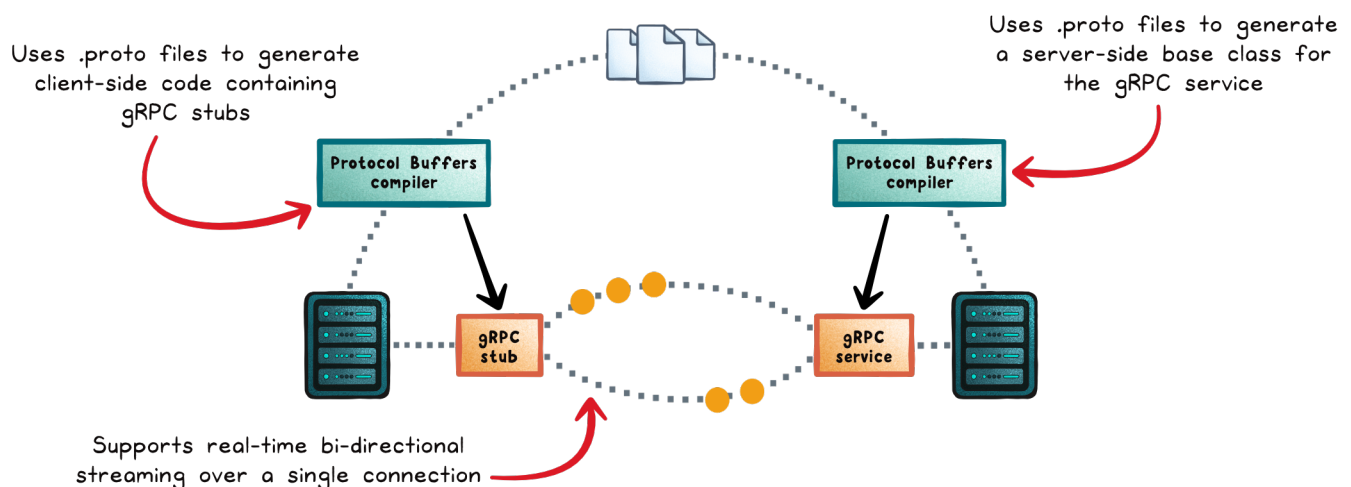
**i** A high-performance **Remote Procedure Call (RPC)** framework built on **HTTP/2** and **Protocol Buffers (Protobuf)**. It lets services call each other as if invoking local functions, using compact binary messages instead of JSON.

| Benefits                                                                                                                                                                                               | Tradeoffs                                                                                                                                                                                     | When to Use                                                                                                                                                                                          |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Low latency, small payloads</li><li>• Strong API contracts</li><li>• Streaming support</li><li>• Polyglot friendly</li><li>• Faster dev with codegen</li></ul> | <ul style="list-style-type: none"><li>• Browser incompatibility</li><li>• Harder debugging</li><li>• Tight client coupling</li><li>• Learning overhead</li><li>• Infra requirements</li></ul> | <ul style="list-style-type: none"><li>• Internal microservices</li><li>• Performance-critical paths</li><li>• Real-time systems</li><li>• Controlled environments</li><li>• Service meshes</li></ul> |

## Why it matters

Modern microservices make thousands of internal calls per user request. gRPC cuts latency, reduces payload size, and supports real-time bidirectional communication. It provides strong, typed API contracts which are crucial when many teams and languages need to interoperate reliably.

## How it works



- Services define APIs using **.proto files** that describe request/response messages and RPC methods.
  - From this contract, gRPC generates **client and server code** in multiple languages.
  - Calls are sent over **long-lived HTTP/2 connections**, allowing multiplexed requests and optional **unary or streaming** (server, client, or bidirectional) communication.
- 

## Benefits

- **High performance** comes from compact binary payloads and HTTP/2 multiplexing.
  - Strongly typed APIs create **clear contracts** that catch integration errors at compile time.
  - **Bi-directional streaming** enables real-time updates without polling or extra protocols.
  - Language-agnostic code generation supports **polyglot teams** without custom SDKs.
  - Built-in deadlines, retries, and status codes improve **system reliability** under load.
  - Less boilerplate speeds up teams and improves **developer velocity**.
- 

## Tradeoffs

- **Limited browser support** means frontends need gRPC-Web or REST gateways.
  - Binary payloads make **debugging harder** without specialized tools.
  - Introduces **tighter coupling** between client and server versions.
  - Teams face a **learning curve** with Protobuf schemas and code generation.
  - HTTP/2 requirements can add **infrastructure complexity** in legacy environments.
- 

## When to use it

- High-throughput, low-latency internal microservice communication.
- Systems needing **real-time streaming** (chat, telemetry, live dashboards).
- Polyglot environments that benefit from **type-safe shared APIs**.
- Backends where you fully control both client and server.

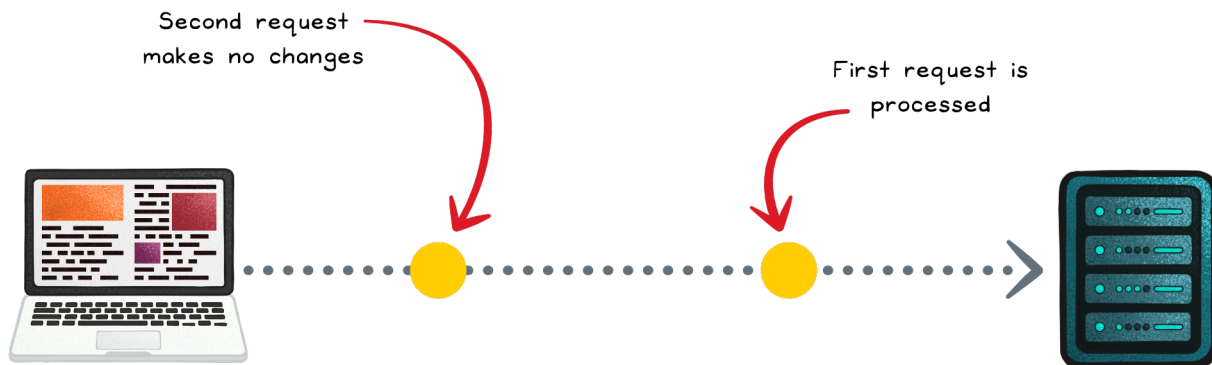


## When not to use it

- Public APIs or browser-facing services (requires gRPC-Web or REST proxy).
- Environments where debugging simplicity or human-readable payloads are priorities.
- Teams not ready for schema-first development or more complex tooling.
- Legacy networks lacking proper HTTP/2 support.

# Idempotency

**i** Idempotency ensures that performing the *same request* multiple times produces the *same effect* as doing it once. The system's state doesn't change after the first successful execution, even if retries or duplicates occur.



| Benefits                                                                                                                                                           | Tradeoffs                                                                                                                                                                            | When to Use                                                                                                                                                                                         |
|--------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Safe retries</li><li>• Data integrity</li><li>• Predictable UX</li><li>• Easier recovery</li><li>• Replay safety</li></ul> | <ul style="list-style-type: none"><li>• Added complexity</li><li>• Storage overhead</li><li>• Latency impact</li><li>• Distributed coordination</li><li>• Hard to retrofit</li></ul> | <ul style="list-style-type: none"><li>• Payments &amp; billing</li><li>• Unreliable networks</li><li>• Webhooks &amp; messaging</li><li>• User submissions</li><li>• High-cost duplicates</li></ul> |

## Why it matters

In distributed systems, retries are inevitable due to timeouts, network failures, and partial outages.

Without idempotency, retries can cause double charges, duplicate orders, or corrupted state.

Idempotency turns retries from a risk into a safe default.



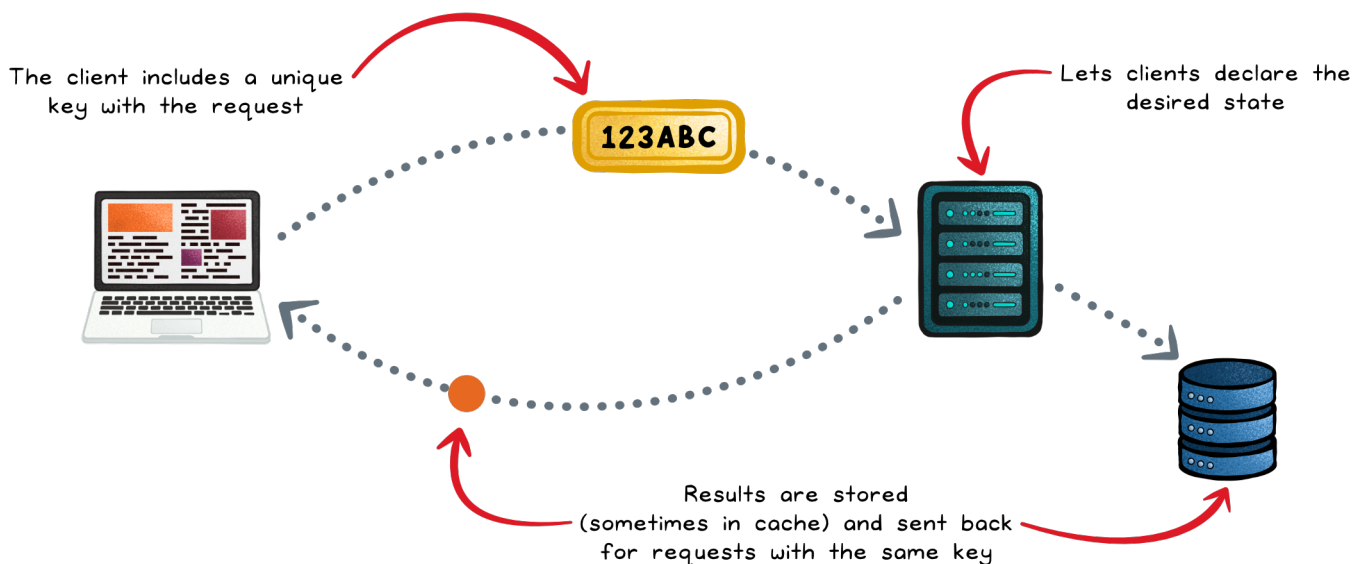


## How it works

Idempotency is usually enforced by making the server recognize and safely ignore duplicates.

Common approaches include:

- Clients send a **unique idempotency key** with the request.
- The server processes the request once and **stores the result** keyed by that identifier.
- If the same request arrives again, the server **returns the stored outcome** instead of re-executing.
- Some systems rely on **unique constraints or state checks** so repeated actions naturally have no effect.



## Benefits

- **Safe retries** let clients and services recover from timeouts without fear of duplicate side effects.
- Data stays correct because **duplicate writes are prevented**, even under heavy retry pressure.
- Users get a more predictable experience with **no double charges, orders, or notifications**.
- Systems become more resilient since **load balancers and retries don't require special handling**.
- Testing and debugging improve because **replaying requests doesn't corrupt state**.



## Tradeoffs

- Adding idempotency introduces **extra implementation complexity** around request tracking and state checks.
  - Storing request keys and responses creates **memory or database overhead** that must be managed.
  - Each request may incur **additional latency** due to duplicate detection.
  - Coordinating idempotency across instances adds **distributed consistency challenges**.
  - It's often **hard to retrofit later**, especially for operations with irreversible side effects.
- 



## When to use it

- Payment, billing, refunds, or any operation where **duplicates are unacceptable**.
  - APIs that will be **automatically retried** due to unreliable networks.
  - Webhooks or message consumers with **at-least-once delivery semantics**.
  - User actions like checkout, signup, or submissions that may be repeated.
- 



## When not to use it

- Low-risk operations where duplicates are harmless or easily cleaned up.
- Performance-critical paths where **duplicate detection cost outweighs the benefit**.
- Actions that rely on **irreversible external side effects** without redesign.
- Internal tools where retries are rare and failures are manually managed.

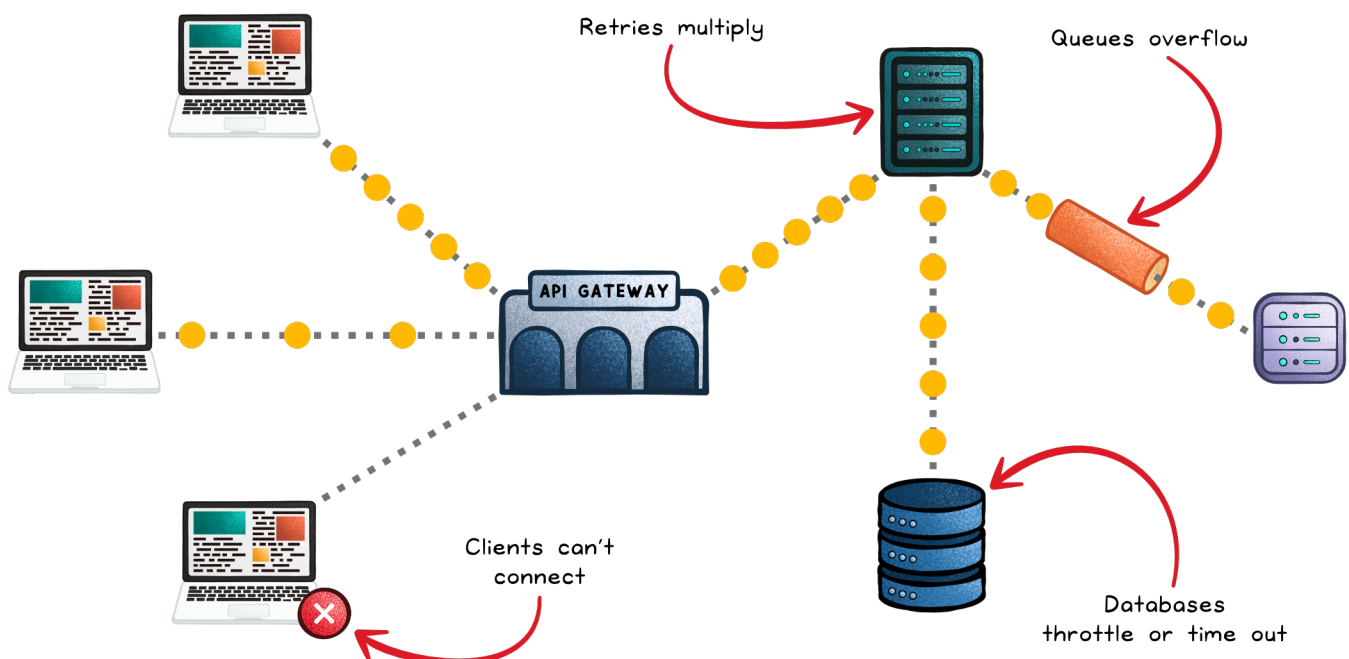
# Rate Limiting

**i** Rate limiting controls how often a client can perform an action (like calling an API) within a fixed time window. It protects shared resources by enforcing fair, predictable usage.

| Benefits                                                                                                                                                                          | Tradeoffs                                                                                                                                                                                | When to Use                                                                                                                                                                               |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Prevents overload</li><li>• Fair resource usage</li><li>• Predictable latency</li><li>• Abuse protection</li><li>• Cost control</li></ul> | <ul style="list-style-type: none"><li>• Added complexity</li><li>• Possible false rejections</li><li>• Coordination overhead</li><li>• Tuning required</li><li>• User friction</li></ul> | <ul style="list-style-type: none"><li>• Shared APIs</li><li>• Bursty traffic</li><li>• Multi-tenant systems</li><li>• Cost-sensitive services</li><li>• Public-facing endpoints</li></ul> |

## Why it matters

Modern systems fail most often due to overload; not bad code. Rate limiting prevents any client from overwhelming a service, improves reliability, and shields APIs from abuse and DDoS patterns. It's as essential to resilience as timeouts, retries, and circuit breakers.



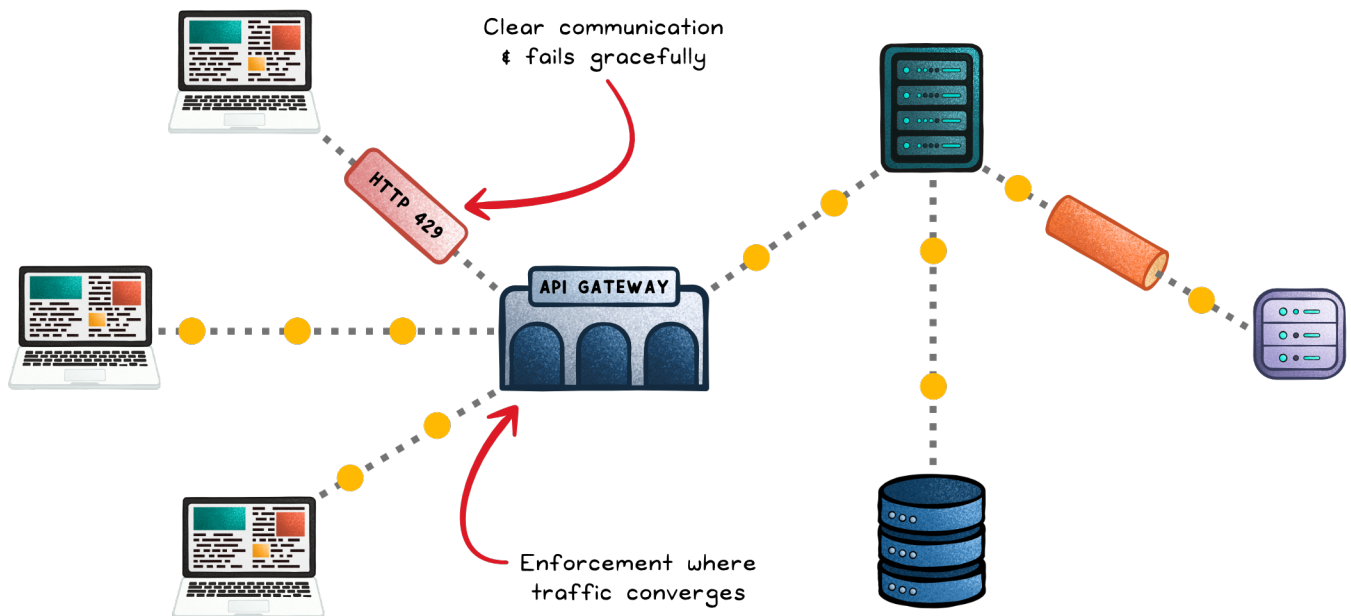
## How it works

Rate limiting tracks incoming requests and decides whether to allow, delay, or reject them based on recent activity.

Common approaches include:

- **Token Bucket** → tokens regenerate at a fixed rate; requests consume tokens. Allows short bursts while controlling long-term average.
- **Leaky Bucket** → all requests flow through a fixed-rate “leak.” Smooths bursts into a constant output.
- **Fixed Window** → count requests per time window (simple but can cause boundary spikes).
- **Sliding Window** → tracks requests in a moving window for more precise control.

These algorithms decide whether to allow, delay, or reject a request (often via HTTP 429 with [Retry-After](#)).



---

## Benefits

- **Protects system stability** by preventing traffic spikes from overwhelming downstream services.
- Improves **fairness across users**, so no single client can monopolize shared resources.

- Leads to **more predictable latency**, keeping response times consistent under load.
  - Acts as a lightweight **security control** against brute-force attacks and basic denial-of-service attempts.
  - Supports **cost control and tiered pricing**, making usage predictable and enforceable.
- 

## Tradeoffs

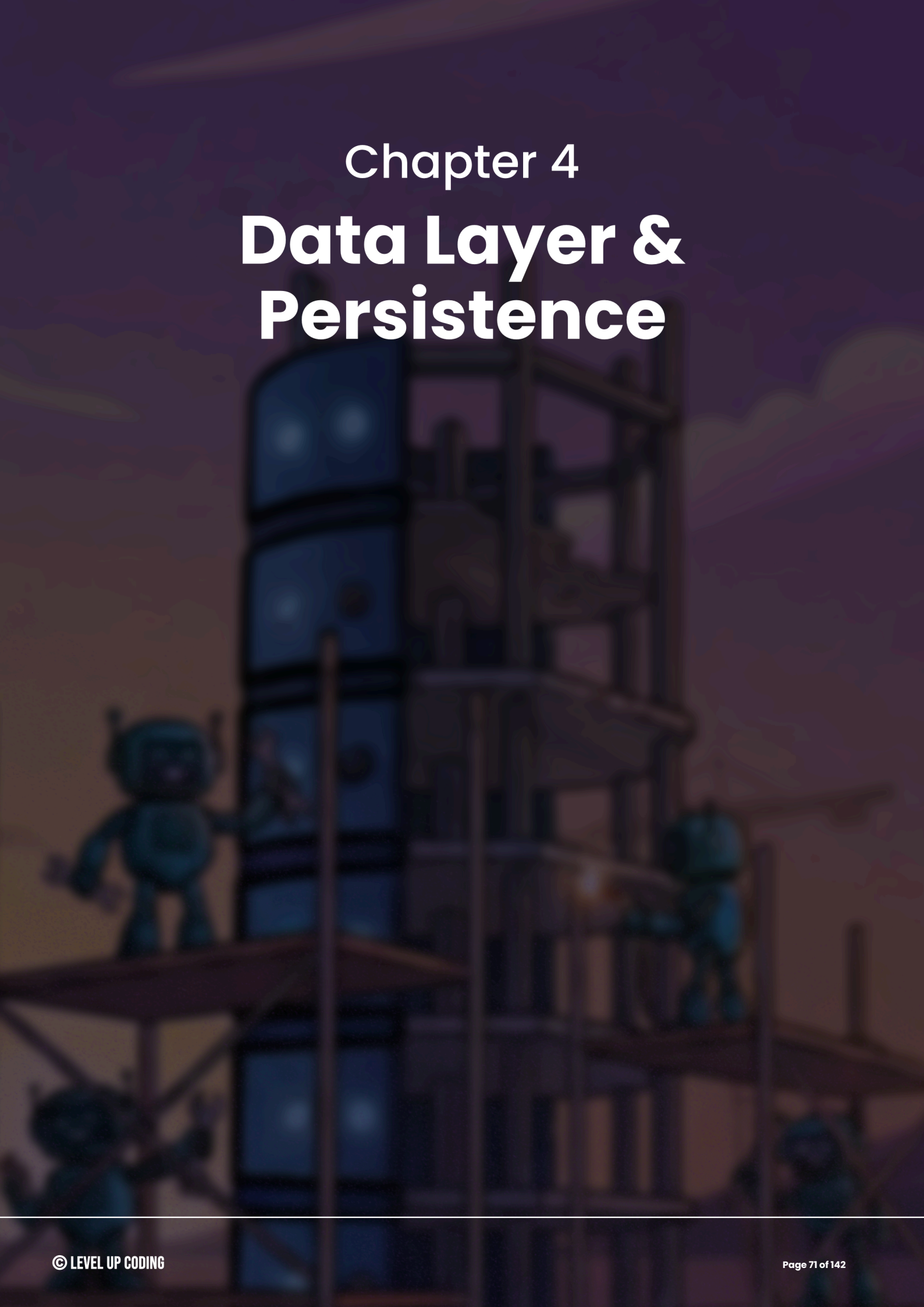
- Adds **architectural and operational complexity**, especially in distributed systems.
  - Can introduce **user-visible errors or delays** if limits are tuned too aggressively.
  - Requires **shared state or coordination**, which may become a bottleneck at scale.
  - Strict limits may **reject legitimate bursty traffic**, even when the system could handle it.
  - Needs **ongoing tuning and monitoring** as usage patterns evolve.
- 

## When to use it

- Protecting backend services from overload or cascading failures.
  - Public APIs where fairness and abuse prevention matter.
  - Expensive operations such as search, file uploads, or data-intensive queries.
  - Multi-tenant systems needing predictable per-client quotas.
  - Any system with bursty or unpredictable traffic.
- 

## When not to use it

- When internal traffic is already safely bounded by design.
- When actions must *never* be rejected (use queueing/shaping instead).
- When limits are unclear or hard to communicate, clients will misbehave without guidance.
- When rate limiting creates more overhead than the operations being protected.

The background of the page is a stylized illustration of a rocket launch. A large, blue, cylindrical rocket is being assembled on a complex metal scaffolding structure. Several small, blue, humanoid robots with antennae are positioned at different levels of the scaffolding, appearing to be working on the rocket. The scene is set against a dark, purple, and blue sky with some clouds. The overall tone is futuristic and technical.

## Chapter 4

# Data Layer & Persistence



# ACID

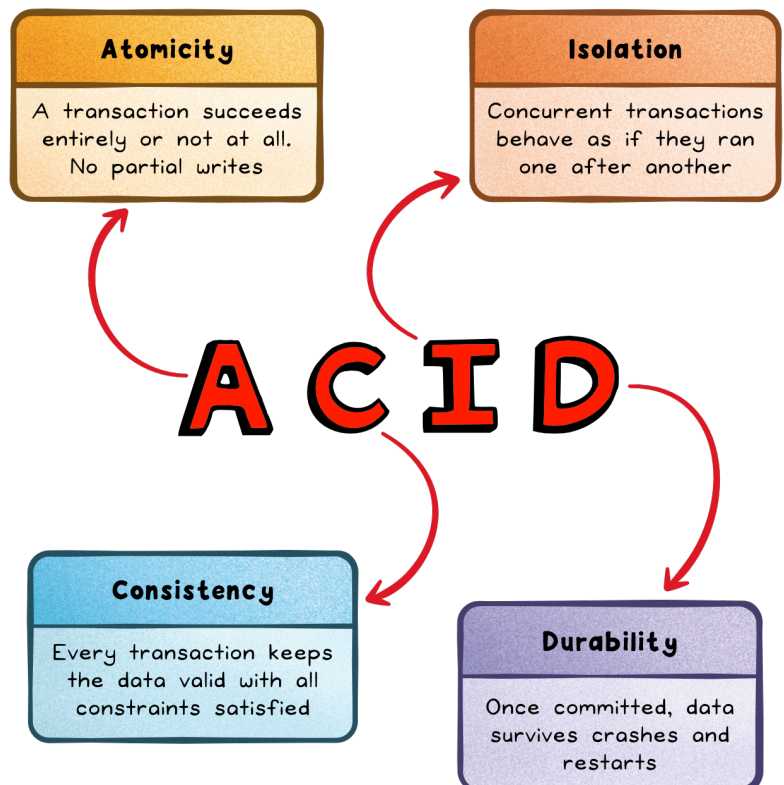
 ACID is a set of **transaction guarantees** (Atomicity, Consistency, Isolation, and Durability) that ensure database operations are processed safely and predictably as a single unit of work.

| Benefits                                                                                                                                                                                                               | Tradeoffs                                                                                                                                                                                | When to Use                                                                                                                                                                 |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Strong consistency</li><li>• Atomic all-or-nothing writes</li><li>• Simple concurrency model</li><li>• Durable, crash-safe commits</li><li>• Safer application logic</li></ul> | <ul style="list-style-type: none"><li>• Coordination overhead</li><li>• Harder to scale globally</li><li>• Reduced availability on failure</li><li>• Higher latency under load</li></ul> | <ul style="list-style-type: none"><li>• Financial transactions</li><li>• Multi-entity updates</li><li>• Regulated systems</li><li>• Correctness &gt; availability</li></ul> |



## Why it matters

ACID is what makes databases trustworthy for **critical data**. It prevents partial updates, race conditions, and lost writes, so the system is always in a valid state after a transaction completes. This is essential when correctness matters more than raw speed or global availability.



## How it works

ACID-compliant databases group multiple operations into a transaction that either fully succeeds or fully fails.



## Benefits

- **Strong consistency ensures** every read reflects correct, committed data.
  - **Atomic transactions** prevent partial writes and cleanup work.
  - **Isolation simplifies concurrency**, by focusing on one transaction at a time.
  - Once acknowledged, **durability protects committed data** even during crashes or power loss.
  - ACID systems enable **simpler application logic** because the database enforces correctness.
- 

## Tradeoffs

- Locking, logging, and coordination can **increase latency**.
  - Maintaining **global consistency limits horizontal scalability**, especially across regions.
  - Distributed ACID requires **complex coordination protocols**.
  - Systems may sacrifice **availability during failures** to avoid serving inconsistent data.
  - Strict schemas and guarantees can reduce **flexibility for fast-evolving data models**.
- 

## When to use it

- Financial, billing, or inventory systems where correctness is non-negotiable.
  - Workloads with **multi-step or multi-row transactions** that must succeed together.
  - Regulated domains requiring auditability and strict data integrity.
  - Applications that fit on a single node or small, tightly coordinated cluster.
- 

## When not to use it

- Internet-scale systems that must remain available during network partitions.
- Use cases tolerant of **slightly stale data** in exchange for speed and uptime.
- Write-heavy or globally distributed workloads where coordination cost dominates.

# BASE

-  BASE is a consistency model that prioritizes **availability and scalability** over immediate correctness by allowing data to be temporarily inconsistent. It stands for **Basically Available, Soft state, Eventually consistent**.

| Benefits                                                                                                                                                                                     | Tradeoffs                                                                                                                                                                          | When to Use                                                                                                                                                                                |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Always-on availability</li><li>• Horizontal scaling</li><li>• Low write latency</li><li>• Flexible schemas</li><li>• Resilient to failures</li></ul> | <ul style="list-style-type: none"><li>• Stale reads</li><li>• App-level complexity</li><li>• Weak invariants</li><li>• Harder debugging</li><li>• No atomic multi-writes</li></ul> | <ul style="list-style-type: none"><li>• Global systems</li><li>• User-facing features</li><li>• Analytics &amp; logging</li><li>• Derived data stores</li><li>• Rebuildable data</li></ul> |

---

## Why it matters

At large scale, strict consistency can make systems slow or unavailable during failures.

BASE accepts short-lived inconsistency so systems can stay responsive, scale horizontally, and survive network partitions.

This tradeoff underpins many modern, global systems where uptime matters more than perfectly fresh reads.

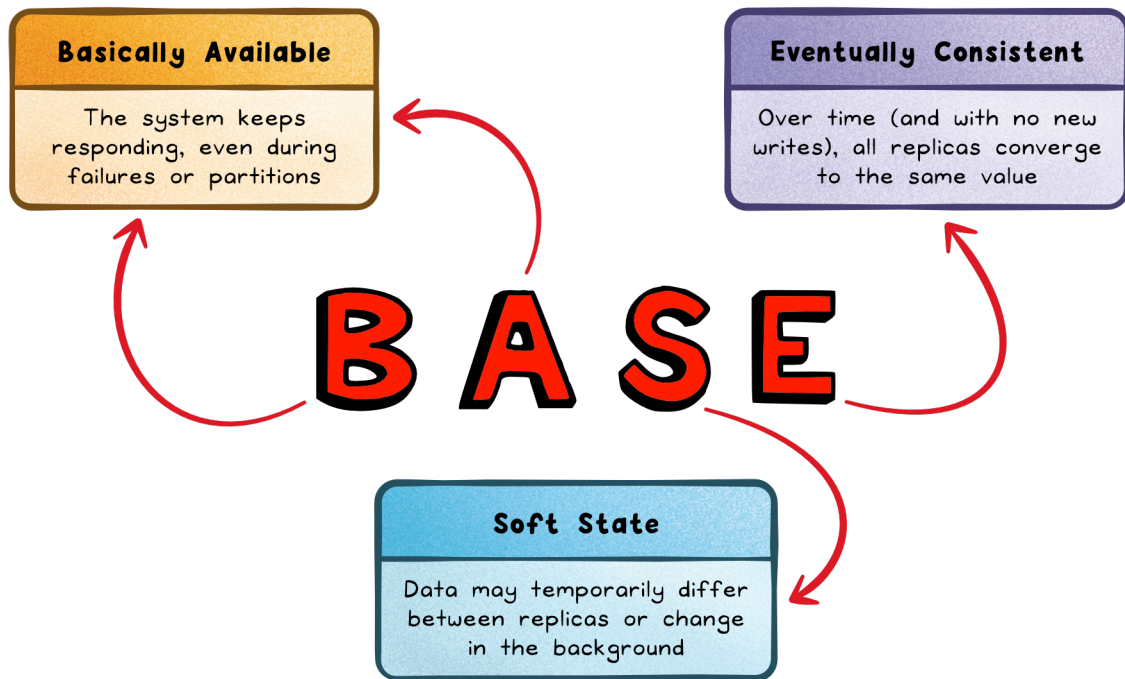
---

## How it works

BASE systems avoid global locks and synchronous coordination.

Writes are typically accepted by one node and **replicated asynchronously** to others. During this window, different replicas may return different values.

Background processes and conflict resolution strategies (like last-write-wins or merges) ensure replicas **eventually converge** to the same state.



## ★ Benefits

- **High availability** keeps systems responsive even during partial failures or network partitions.
- By avoiding synchronous coordination, BASE enables **horizontal scalability** across many nodes and regions.
- **Lower write latency** improves user-facing responsiveness under heavy load.
- Flexible schemas and relaxed guarantees support **rapid product iteration** and evolving data models.
- Systems remain usable during outages, increasing **user trust and perceived reliability**.

## ⚖ Tradeoffs

- **Eventual consistency** means reads may return stale or conflicting data for a period of time.
- The burden of correctness often shifts into **application-level logic** and reconciliation.
- Enforcing real-time invariants like uniqueness or balances becomes **architecturally complex**.
- Debugging is harder because system state depends on **timing, replication lag, and conflict resolution**.
- Multi-entity atomic updates are difficult or impossible without reintroducing ACID-like mechanisms.



## When to use it

- Global, internet-scale systems where uptime is more important than perfect freshness.
  - User-facing features like feeds, metrics, or profiles that tolerate slightly stale data.
  - High-throughput logging, analytics, or event ingestion pipelines.
  - Derived or secondary data stores that can be rebuilt or reconciled later.
- 



## When not to use it

- Financial, billing, or inventory systems where correctness is non-negotiable.
- Workflows requiring strict real-time constraints or atomic multi-record updates.
- Teams without the capacity to manage **eventual consistency complexity** safely.

# SQL

 SQL databases store data in structured tables with predefined schemas and relationships, queried using Structured Query Language (SQL).

| Benefits                                                                                                                                                                | Tradeoffs                                                                                                                                                                                           | When to Use                                                                                                                                                                                             |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Strong consistency</li><li>• ACID safety</li><li>• Powerful joins</li><li>• Mature tooling</li><li>• Predictable data</li></ul> | <ul style="list-style-type: none"><li>• Schema rigidity</li><li>• Hard sharding</li><li>• Vertical scaling limits</li><li>• Single-primary bottlenecks</li><li>• Global coordination cost</li></ul> | <ul style="list-style-type: none"><li>• Transactional systems</li><li>• Relational data</li><li>• Financial workflows</li><li>• Analytics &amp; reporting</li><li>• High data integrity needs</li></ul> |



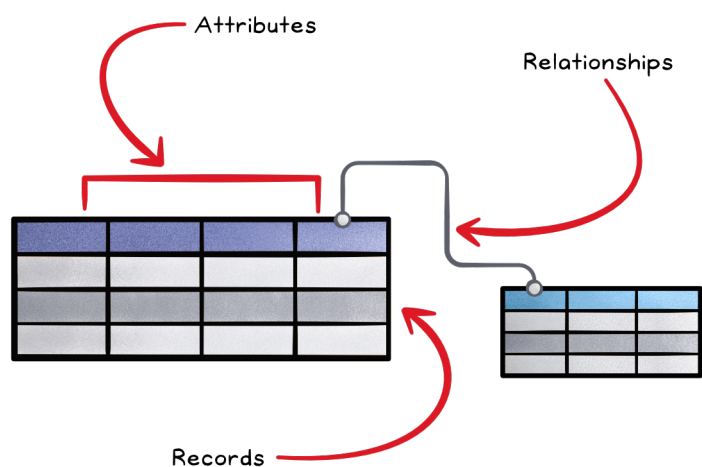
## Why it matters

Most business data is highly structured and relational. SQL provides strong correctness guarantees and powerful querying, making it the backbone of transactional systems where accuracy, consistency, and trust matter.



## How it works

- Data is modeled upfront into tables with rows and columns. Relationships are enforced using primary and foreign keys.
- Operations run inside **ACID transactions**, ensuring multi-step changes either fully succeed or fully fail.
- SQL queries declaratively describe *what* data you want, and the database optimizer figures out *how* to retrieve it efficiently.



## Benefits

- **Strong data integrity** is enforced by schemas, constraints, and transactions, making corruption and partial updates extremely unlikely.
  - Complex questions are easy to ask thanks to **powerful joins and aggregations**, even across many tables.
  - **Predictable data models** make systems easier to reason about, debug, and validate over time.
  - Teams benefit from a **mature ecosystem** of tooling, ORMs, monitoring, and operational knowledge.
  - **Transactional safety** simplifies application logic for multi-step business workflows like payments or inventory updates.
- 

## Tradeoffs

- Scaling beyond a single node introduces **significant architectural complexity**, especially with sharding.
  - **Rigid schemas** make rapid iteration or frequent structural changes slower and more expensive.
  - At very large scale, **vertical scaling costs** can grow quickly due to high-end hardware or licensing.
  - Strong consistency can introduce **latency tradeoffs** in distributed or high-availability setups.
  - Relational modeling requires **up-front design effort**, which can slow early experimentation.
- 

## When to use it

- You need **strict correctness** for transactions like payments, inventory, or accounts.
- Data has **clear relationships** that benefit from joins and constraints.
- You require **complex queries or reporting** across multiple entities.
- The dataset fits within **vertical scaling or modest replication**.



## When not to use it

- You expect **massive horizontal scale** across many nodes or regions.
- The data model is **highly dynamic or unstructured**.
- The system can tolerate **eventual consistency** for higher availability.
- Access patterns are **simple key-based lookups** with extreme throughput needs.



# NoSQL

- i** NoSQL refers to **non-relational databases** that use flexible, non-relational models. They prioritize scalability, availability, and adaptability over strict relational structure.

| Benefits                                                                                                                                                   | Tradeoffs                                                                                                                                                | When to Use                                                                                                                                                    |
|------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Horizontal scale-out</li><li>• Flexible schemas</li><li>• High availability</li><li>• Low-latency access</li></ul> | <ul style="list-style-type: none"><li>• Eventual consistency</li><li>• Fewer joins</li><li>• Operational complexity</li><li>• Data duplication</li></ul> | <ul style="list-style-type: none"><li>• Massive scale</li><li>• Rapidly changing data</li><li>• Global distribution</li><li>• Simple access patterns</li></ul> |



## Why it matters

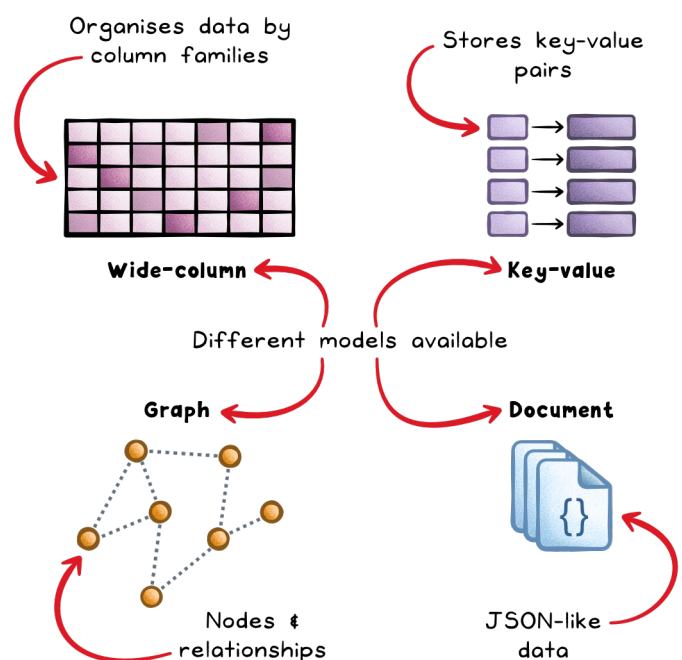
Modern systems often deal with **massive scale**, **rapidly changing data**, and **distributed traffic**. NoSQL databases make it practical to handle these workloads without constant schema migrations or complex sharding strategies.



## How it works

NoSQL databases avoid a single data model and instead use several specialized ones:

- **Key-value stores** map a key directly to a value for ultra-fast lookups.
- **Document stores** keep data as self-contained JSON-like documents.
- **Wide-column stores** group data by columns for large, sparse datasets.
- **Graph databases** model data as nodes and relationships for connected data.



Most NoSQL systems scale **horizontally**, partitioning and replicating data across nodes. They favor **availability and partition tolerance** over strict ACID, using eventual consistency.

## Benefits

- Systems **scale horizontally** by adding nodes rather than upgrading a single server.
  - Schema flexibility supports **rapid iteration**.
  - Built-in replication improves **availability and fault tolerance** in distributed environments.
  - Simple access patterns enable **low-latency reads and writes** at very high throughput.
  - Data models like documents or graphs often feel **more natural for developers** than relational tables.
- 

## Tradeoffs

- Relaxed guarantees can cause **temporary stale reads**.
  - **Limited join and query capabilities** make complex analytics harder than in SQL.
  - Denormalized data increases the risk of **data duplication and inconsistency**.
  - Operating distributed clusters adds **infrastructure and operational complexity**.
  - Each database has its own APIs, creating **vendor and tooling fragmentation**.
- 

## When to use it

- You need to handle **very large data volumes or high traffic**.
  - Data structures are **unpredictable or evolve frequently**.
  - Availability is more important than immediate consistency.
  - Access patterns are simple and well understood.
- 

## When not to use it

- You require **strong ACID transactions** across many records.
- Your workload depends on **complex joins and reporting**.
- Data integrity rules must be strictly enforced by the database.
- The system is small enough that a single relational database is sufficient.

# Key-Value Databases

 A key-value database stores data as simple **key** → **value** pairs, similar to a distributed hash map. Each key uniquely identifies a value, which the database treats as an opaque blob.

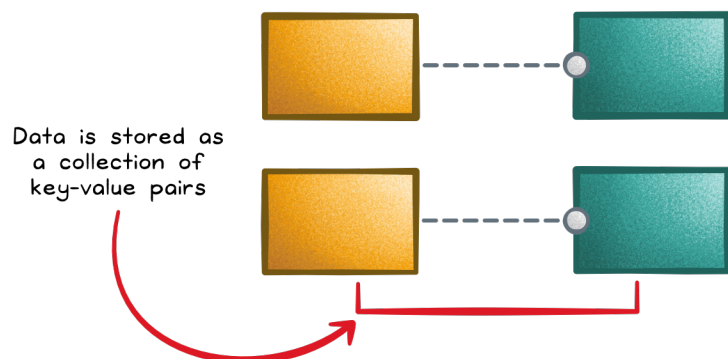
| Benefits                                                                                                                                                                              | Tradeoffs                                                                                                                                                           | When to Use                                                                                                                                          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Millisecond reads</li><li>• Simple mental model</li><li>• Easy horizontal scale</li><li>• High throughput</li><li>• Flexible values</li></ul> | <ul style="list-style-type: none"><li>• No rich queries</li><li>• Weak integrity guarantees</li><li>• Hard multi-key updates</li><li>• Blob rewrite costs</li></ul> | <ul style="list-style-type: none"><li>• ID-based access</li><li>• Caching layers</li><li>• Session storage</li><li>• Extreme traffic scale</li></ul> |

## Why it matters

Key-value stores are built for **speed and scale**. When systems need to read or write data in milliseconds under heavy traffic, the simplicity of key-based access avoids the overhead of schemas, joins, and query planners.

## How it works

- Data is written with a unique key and retrieved using that same key.
- Internally, keys are partitioned across nodes (often via consistent hashing), making it easy to scale horizontally.



Many systems keep data in memory or use append-only logs to optimize for extremely fast reads and writes, while replication provides fault tolerance.

## Benefits

- **Ultra-fast lookups** make key-value stores ideal for latency-sensitive paths.
  - **Simple data models** reduce cognitive load and ease reasoning.
  - **Horizontal scalability** comes naturally because each key is independent and easy to partition.
  - Teams gain **operational reliability** since failures affect only subsets of keys, not global queries.
  - **Flexible value storage** allows rapid iteration without schema migrations or coordination.
- 

## Tradeoffs

- **Limited queries** prevent efficient filtering, joins, and aggregation.
  - **Lack of relational integrity**, pushes consistency checks into application code.
  - **Multi-key transactions are hard**, making atomic updates across records error-prone.
  - Treating values as blobs creates **update inefficiencies** when small changes require rewriting large values.
  - Over time, **schema discipline erodes** unless teams enforce strong application-level contracts.
- 

## When to use it

- You need extremely fast reads and writes by ID.
  - Access patterns are simple and predictable.
  - The system must scale to very high request volumes.
  - Data can tolerate weaker transactional guarantees.
- 

## When not to use it

- You need complex queries, joins, or analytics.
- Strong consistency across multiple records is required.
- The data model has rich relationships.
- Long-term data integrity must be enforced centrally.

# Document Databases

- i** A document database stores data as **self-contained documents**, usually in JSON-like formats, instead of rows and tables. Each document can have its own structure, making the model flexible and schema-light by design.

| Benefits                                                                                                                                                      | Tradeoffs                                                                                                                                                                                                    | When to Use                                                                                                                                                               |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Flexible schema</li><li>• Fast aggregate reads</li><li>• Easy horizontal scaling</li><li>• Faster iteration</li></ul> | <ul style="list-style-type: none"><li>• No enforced relationships</li><li>• Data duplication risk</li><li>• Limited joins</li><li>• App-level integrity checks</li><li>• Non-standard query models</li></ul> | <ul style="list-style-type: none"><li>• Evolving data models</li><li>• Object-centric access</li><li>• Semi-structured data</li><li>• Whole objects are fetched</li></ul> |

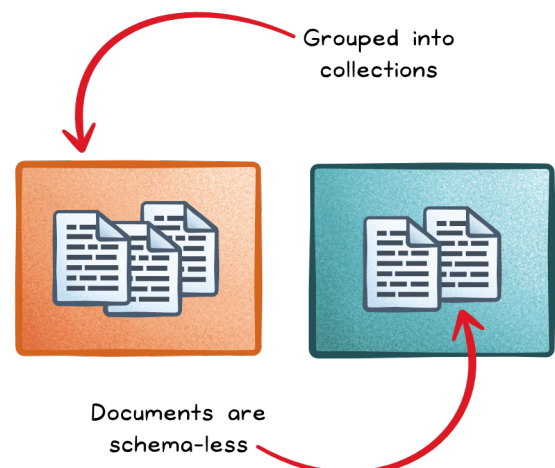
## Why it matters

Many modern applications deal with **evolving, semi-structured data** that doesn't fit neatly into rigid schemas.

Document databases let teams move faster by aligning storage with how data is already modeled in application code, while scaling easily for web workloads.

## How it works

- Documents are stored in collections (similar to tables, but without enforced schemas).
- Each document has a unique ID and often embeds related data directly, reducing the need for joins.
- Queries typically fetch entire documents by ID or filter on indexed fields within the document.



## Benefits

- **Schema flexibility** allows teams to add or change fields without migrations or downtime.
  - Grouping related data enables **faster reads** since one lookup often retrieves everything needed.
  - The document model matches application objects, **reducing ORM friction**.
  - **Horizontal scalability** is simpler because documents are independent and easy to shard by key.
  - Faster iteration supports **product experimentation and rapid feature evolution**.
- 

## Tradeoffs

- **Weak relational guarantees** mean no foreign keys or enforced relationships.
  - Data is often duplicated, increasing the **risk of inconsistent updates** across documents.
  - **Complex joins and cross-document queries** are inefficient or unsupported.
  - Query languages and tooling are less standardized than SQL, creating **vendor lock-in risk**.
  - Maintaining data quality requires more **application-level discipline and validation**.
- 

## When to use it

- Data is naturally **hierarchical or aggregate-oriented**.
  - Schemas change frequently or differ across records.
  - Reads usually fetch whole objects, not partial fields across entities.
  - You're building content-heavy or user-centric applications.
- 

## When not to use it

- You rely heavily on **many-to-many relationships and joins**.
- Strong consistency and referential integrity are critical.
- The data model is stable, highly structured, and relational.
- You need advanced analytics across normalized datasets.

# Wide-Column Databases

- i** Wide-column databases (also called column-family stores) are NoSQL databases that store data in tables with **flexible, sparse columns** that can vary per row, rather than a fixed schema.

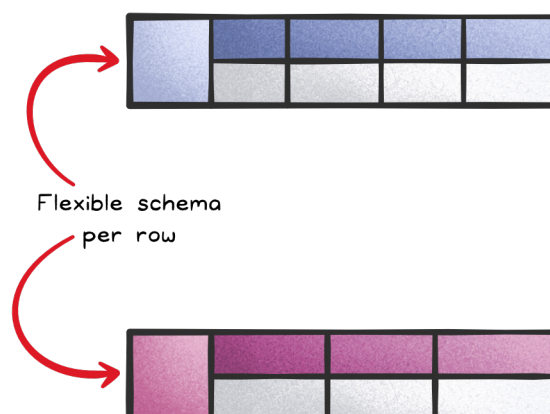
| Benefits                                                                                                                                                                                                            | Tradeoffs                                                                                                                                                                                            | When to Use                                                                                                                                                          |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Massive horizontal scale</li><li>• High write throughput</li><li>• Sparse storage efficiency</li><li>• Fault-tolerant by design</li><li>• Flexible per-row schema</li></ul> | <ul style="list-style-type: none"><li>• Complex data modeling</li><li>• Limited querying</li><li>• Eventual consistency</li><li>• No multi-row transactions</li><li>• Operational overhead</li></ul> | <ul style="list-style-type: none"><li>• Write-heavy systems</li><li>• Analytics pipelines</li><li>• Telemetry &amp; logs</li><li>• Petabyte-scale datasets</li></ul> |

## Why it matters

They are designed to handle **massive scale**, both in data volume and throughput, where traditional relational databases struggle. Wide-column databases power systems that ingest **billions of writes per day** while remaining highly available across clusters.

## How it works

- Data is organized by a **row key**, with columns grouped into **column families**.
- Each row can have a completely different set of columns, and missing columns simply don't exist.
- Rows are distributed across nodes by key, enabling horizontal scaling, and data is often replicated with **tunable consistency** to balance availability and correctness.





## ★ Benefits

- **Extreme horizontal scalability** allows systems to grow to petabytes large.
  - High write throughput makes them ideal for **write-heavy workloads** like logs, metrics, and event streams.
  - The **sparse column model** avoids wasted storage when many attributes are optional or rarely present.
  - Data replication across nodes improves **availability and fault tolerance** under failures.
  - Query performance can be very fast when access patterns align with **row keys and column families**.
  - **Per-row schema flexibility** enables teams to evolve data models safely.
- 

## ⚖ Tradeoffs

- **Data modeling is complex**, and schemas must carefully follow query patterns.
  - Limited querying means **no joins and weak ad-hoc queries**, pushing complexity into application logic.
  - Many systems favor **eventual consistency**, which can surprise teams expecting immediate correctness.
  - **No multi-row ACID transactions** make cross-entity updates harder to reason about.
  - Operating clusters introduces **significant operational overhead** around tuning, repairs, and monitoring.
  - Poor fit for workloads that require **frequent filtering by non-key attributes**.
- 

## 🤖 When to use it

- You need to store **very large datasets** with predictable access patterns.
  - The workload is **write-heavy** and latency sensitive.
  - Data can be naturally partitioned by a **primary key or key range**.
  - Time-series, telemetry, analytics, or event data is core to the system.
- 

## 🤖 When not to use it

- You need **rich relational queries or joins**.
- Strong, immediate consistency across entities is required.
- The dataset is small or moderate.
- Query patterns are **unpredictable or exploratory**.

# Graph Databases

**i** A graph database stores data as **nodes** (entities) and **edges** (relationships), with both able to hold properties. It's designed for data where **relationships are first-class**.

| Benefits                                                                                                                                                                                                   | Tradeoffs                                                                                                                                                                                          | When to Use                                                                                                                                                                               |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Fast relationship traversal</li><li>• Natural data modeling</li><li>• No joins needed</li><li>• Flexible connections</li><li>• Built-in graph algorithms</li></ul> | <ul style="list-style-type: none"><li>• Harder to scale horizontally</li><li>• Specialized query languages</li><li>• Weaker analytics performance</li><li>• Added operational complexity</li></ul> | <ul style="list-style-type: none"><li>• Relationship-centric domains</li><li>• Social or recommendation systems</li><li>• Fraud and network analysis</li><li>• Knowledge graphs</li></ul> |



## Why it matters

Many real systems are fundamentally about connections: users to users, users to content, entities to events.

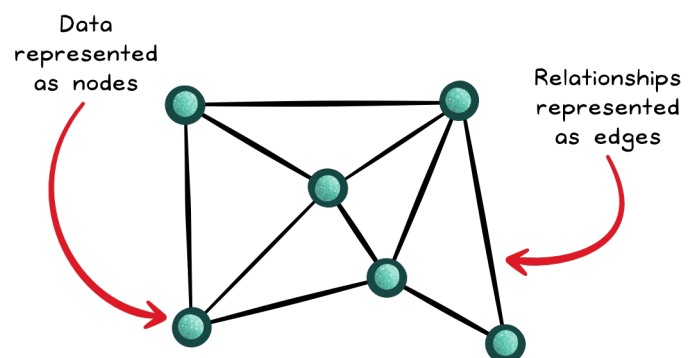
Graph databases make these connections **fast to traverse and easy to reason about**, avoiding complex joins or recursive queries.

They shine when questions are about **how things are related**, not just what they are.



## How it works

- Data is modeled as a graph: nodes represent things, edges represent how they're connected.
- Queries start from one or more nodes and **traverse edges directly**, often across multiple hops.
- Instead of joining tables, the database follows pointers between nodes, making deep relationship queries efficient.



## ★ Benefits

- **Relationship-heavy queries stay fast** even as the graph grows, because traversals don't degrade like joins do.
  - The **data model mirrors real-world relationships**, making systems easier to design, explain, and evolve.
  - Adding new connections is simple since **no global schema changes** are required.
  - **Multi-hop queries feel natural**, such as friends-of-friends or dependency chains.
  - Built-in **graph algorithms** enable recommendations, influence scoring, and path analysis.
  - Complex relationship logic moves **out of application code** and into the database.
- 

## ⚖ Tradeoffs

- **Poor fit for simple lookups or tabular data**, where simpler databases are faster and cheaper.
  - There's a **learning curve with graph query languages** like Cypher or Gremlin.
  - **Horizontal scaling is harder** because traversals may span partitions.
  - Operational tooling is **less mature than SQL ecosystems**.
  - Large analytical scans can be slower since graphs aren't optimized for aggregation.
  - Overusing graphs can add **unnecessary architectural complexity**.
- 

## 💡 When to use it

- Social networks, follows, friendships, or interactions.
  - Recommendation engines based on shared behavior.
  - Fraud detection and relationship analysis.
  - Knowledge graphs and semantic data.
  - Dependency graphs or impact analysis.
- 

## 🤖 When not to use it

- Simple CRUD apps with minimal relationships.
- Analytics-heavy workloads with large aggregations.
- Key-based access patterns.
- Systems needing extreme horizontal write scalability.
- Teams without capacity to support specialized tooling.

# Time-Series Databases

**i** A specialized database designed to store and query **time-stamped data points**, where time is the primary axis. It is optimized for continuously appended data.

| Benefits                                                                                                                                                                                                 | Tradeoffs                                                                                                                                                                             | When to Use                                                                                                                                                                                           |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• High-throughput ingestion</li><li>• Time-optimized queries</li><li>• Efficient compression</li><li>• Automatic retention</li><li>• Real-time analytics</li></ul> | <ul style="list-style-type: none"><li>• Not general-purpose</li><li>• Weaker transactions</li><li>• Costly long scans</li><li>• New query languages</li><li>• Limited joins</li></ul> | <ul style="list-style-type: none"><li>• Metrics and telemetry</li><li>• Monitoring and alerting</li><li>• IoT sensor data</li><li>• Financial time-series</li><li>• Trend and rate analysis</li></ul> |



## Why it matters

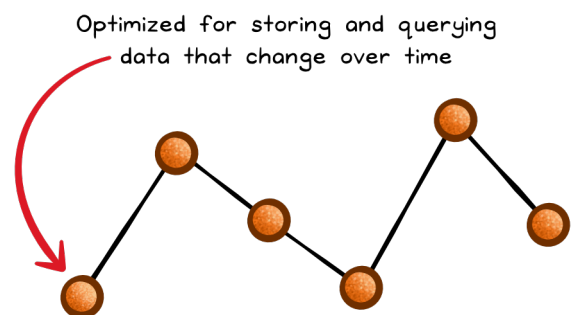
Many systems produce data that only makes sense **in relation to time**.

Using a general-purpose database for this often leads to poor write performance, high storage costs, and slow time-range queries. TSDBs solve this by treating time as a first-class concern.



## How it works

- Data is written in an **append-only** pattern, usually ordered by timestamp.
- Storage is **partitioned by time windows** (e.g. hours or days) to keep recent data fast and older data compact.
- TSDBs apply **compression techniques** that exploit similarity between consecutive values.
- Many systems support **retention policies** and **automatic downsampling**, keeping raw data briefly and summaries long-term.
- Query engines are optimized for **time-range scans** and common aggregations like averages, rates, and percentiles.



## Benefits

- **High write throughput** makes it easy to ingest millions of data points per second without backpressure.
  - Storage stays efficient thanks to **compression and delta encoding** across sequential values.
  - Queries like “last 5 minutes” or “hourly average over a week” are **fast and predictable**.
  - Built-in **retention and downsampling** reduce operational burden and storage cost automatically.
  - Systems support **real-time monitoring and alerting**, improving reliability and incident response.
  - Purpose-built analytics functions accelerate dashboards, trends, and anomaly detection.
- 

## Tradeoffs

- **Complex joins and relational queries** are **limited** or unsupported.
  - Many designs relax **strong transactional guarantees** for speed and availability.
  - Without downsampling, high-resolution queries can cause **expensive scans**.
  - Some systems use **custom query languages**, increasing learning and tooling friction.
  - Correlating time-series with rich metadata often requires **external databases or pipelines**.
- 

## When to use it

- Write volume is high and mostly append-only.
  - Queries are dominated by **time windows, trends, and aggregates**.
  - You need **monitoring, alerting, or real-time dashboards**.
  - Data naturally becomes less valuable as it ages.
- 

## When not to use it

- Your data requires **complex relationships or frequent joins**.
- You need **strict ACID transactions** across multiple entities.
- Access patterns are mostly **point lookups unrelated to time**.
- Time is just an attribute, not the core query dimension.

# Vector Databases

**i** A vector database stores **high-dimensional vectors (embeddings)** generated by ML models and is optimized for **similarity search** rather than exact matches.

| Benefits                                                                                                                                                          | Tradeoffs                                                                                                                                                                                             | When to Use                                                                                                                                                                       |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Semantic relevance</li><li>• Fast similarity lookup</li><li>• AI-native workflows</li><li>• Real-time retrieval</li></ul> | <ul style="list-style-type: none"><li>• Approximate results</li><li>• ML tuning required</li><li>• High memory/CPU</li><li>• Added system complexity</li><li>• Limited query expressiveness</li></ul> | <ul style="list-style-type: none"><li>• Semantic search</li><li>• Recommendations</li><li>• RAG pipelines</li><li>• Image/audio similarity</li><li>• AI-driven products</li></ul> |



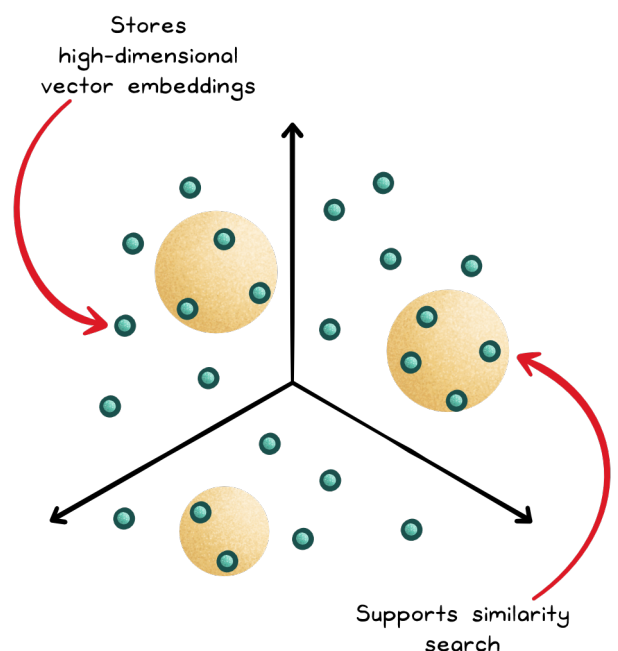
## Why it matters

Modern AI systems rely on meaning, not keywords. Vector databases make it practical to retrieve **semantically similar** content at scale, which is foundational for search, recommendations, and retrieval-augmented generation (RAG) systems.



## How it works

- Data (text, images, audio, code) is converted into numeric embeddings using ML models.
- These embeddings are indexed using **approximate nearest-neighbor (ANN)** structures (e.g. HNSW, IVF) that quickly narrow down likely matches instead of scanning everything.
- Queries embed the input, compare vectors using distance metrics (like cosine similarity), and return the closest results, often combined with metadata filters.



## Benefits

- Enables **semantic search**, where results match intent even when keywords don't overlap.
  - Delivers **fast similarity queries at scale**.
  - Unlocks **AI-native features** like recommendations and RAG pipelines.
  - Improves **user experience and trust** by returning more relevant, context-aware results.
  - Keeps AI systems **updatable in real time**, as new embeddings can be added continuously.
- 

## Tradeoffs

- Results are **approximate**, trading perfect accuracy for speed and scalability.
  - Requires **ML model quality and tuning**, since embeddings define what "similar" means.
  - Comes with **higher memory and CPU costs**, especially for large vector indexes.
  - Adds **operational complexity**, including index tuning and rebalancing over time.
  - Offers **limited general querying**, so it's often paired with a traditional database.
- 

## When to use it

- You need **semantic or similarity search** over text, images, audio, or code.
  - You're building **AI-powered recommendations or discovery features**.
  - You're implementing **RAG for chatbots or assistants**.
  - Keyword or SQL search no longer returns relevant results.
- 

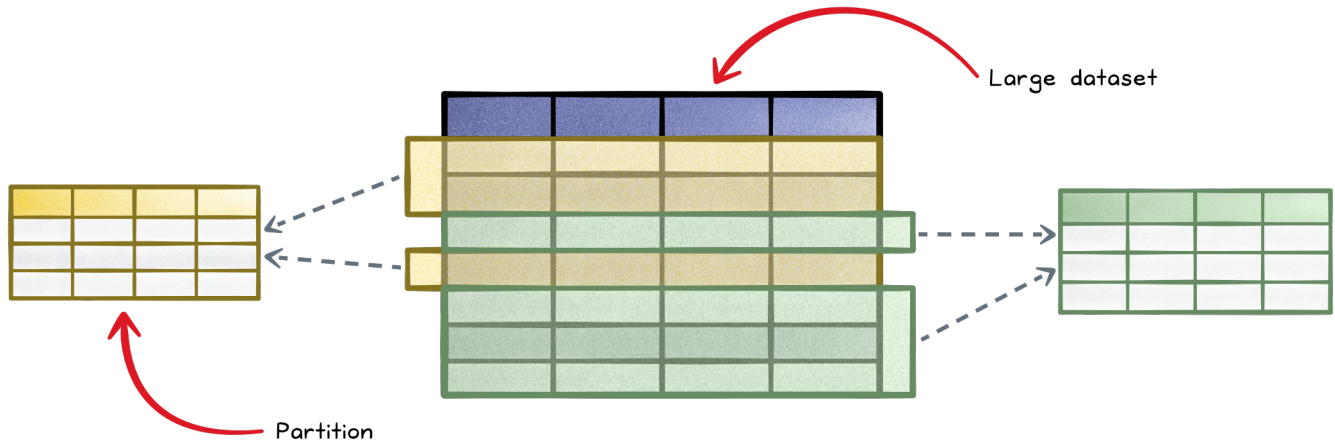
## When not to use it

- Your queries require **exact matches, joins, or transactions**.
- You don't have or need **ML embeddings**.
- Data size or cost constraints make **vector indexing impractical**.
- Simple filters or keyword search already solve the problem.



# Partitioning

**i** Database partitioning is the practice of **splitting a large dataset into smaller, independent pieces** called partitions, where each record belongs to exactly one partition. Each partition behaves like a smaller database, while together they form the full dataset.



| Benefits                                                                                                                                                                                          | Tradeoffs                                                                                                                                                                                                        | When to Use                                                                                                                                                                                                                                              |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Faster targeted queries</li><li>• Horizontal scale-out</li><li>• Fault isolation</li><li>• Easier archiving</li><li>• Better workload isolation</li></ul> | <ul style="list-style-type: none"><li>• Added system complexity</li><li>• Harder cross-partition joins</li><li>• Risk of hotspots</li><li>• Costly repartitioning</li><li>• Ongoing operational burden</li></ul> | <ul style="list-style-type: none"><li>• Large or fast-growing datasets</li><li>• Time-based or tenant-based access</li><li>• High write or read throughput</li><li>• Regular retention or purge needs</li><li>• Multi-tenant or global systems</li></ul> |

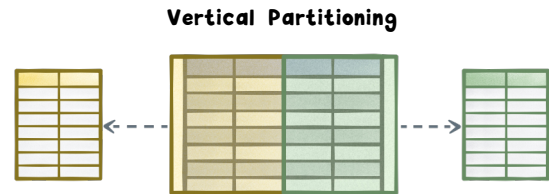
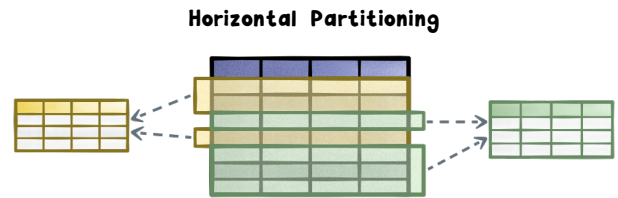
## Why it matters

As data grows, single tables become slow, hard to manage, and expensive to scale. Partitioning improves **performance, scalability, and operational safety** by letting systems work on smaller slices of data instead of one massive whole.



## How it works

- Data is split using a **partition key**, which determines where each record is stored.
- With **horizontal partitioning**, rows are divided across partitions, so each partition holds the same schema but different records (for example, users by ID range or orders by date).
- With **vertical partitioning**, columns are split instead, keeping frequently accessed fields separate from large or rarely used ones while sharing a common primary key.
- Queries that include the partition key can be routed to **only the relevant partitions**, reducing the amount of data scanned.
- Partitions may live on the same machine or be **distributed across nodes**, enabling parallel reads, writes, and maintenance.



## Benefits

- Queries get noticeably faster because they only touch **a small, relevant slice of data** instead of scanning everything.
- As traffic and data grow, systems can scale smoothly by **spreading partitions across machines** rather than upgrading one giant box.
- When a node or disk fails, the blast radius is limited since **only a single partition is affected**, not the entire database.
- Day-to-day operations become safer and simpler because backups, deletes, and archives happen **one partition at a time**.
- In multi-tenant systems, noisy customers are easier to contain since **each tenant's workload stays isolated** from others.
- Compliance becomes more manageable when **sensitive or regional data lives in clearly separated partitions** with tighter controls.



## Tradeoffs

- Query logic becomes harder to reason about once **data is split across partitions**, especially for aggregations and joins.
- Strong guarantees get trickier when updates span partitions, often requiring **distributed transactions or relaxed consistency**.

- If the partition key is chosen poorly, **hotspots can form** and cancel out most of the performance gains.
  - Running a partitioned system adds ongoing overhead, since **more tables, indexes, and monitoring** need to be maintained.
  - Over time, partition layouts can become constraints because **changing strategies usually means large data migrations**.
- 



## When to use it

- Datasets are **too large for a single machine** to handle efficiently.
  - Queries naturally focus on **subsets of data** (by time, tenant, region, or ID).
  - You need **scale-out growth**, not just bigger servers.
  - Old data must be **archived or purged regularly**.
- 



## When not to use it

- The dataset is **small or moderate** and performs well without partitioning.
- Most queries **scan all data anyway**, regardless of filters.
- The team cannot support the **operational complexity** partitioning introduces.

# Sharding

**i** Database sharding is a scaling technique that **splits a large database into smaller, independent databases (shards)**, each holding a subset of the data and running on separate servers. Together, all shards represent the full dataset.

| Benefits                                                                                                                                                                                                | Tradeoffs                                                                                                                                                                                                      | When to Use                                                                                                                                                                                       |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Horizontal scale-out</li><li>• Higher write throughput</li><li>• Failure isolation</li><li>• Better resource usage</li><li>• Cost efficiency at scale</li></ul> | <ul style="list-style-type: none"><li>• High operational complexity</li><li>• Costly resharding</li><li>• Cross-shard query overhead</li><li>• Hard consistency guarantees</li><li>• Hard to reverse</li></ul> | <ul style="list-style-type: none"><li>• Massive data volume</li><li>• Write-heavy workloads</li><li>• Clear shard key</li><li>• Multi-tenant scale</li><li>• Single DB no longer viable</li></ul> |

## Why it matters

A single database server eventually hits hard limits on storage, CPU, and write throughput.

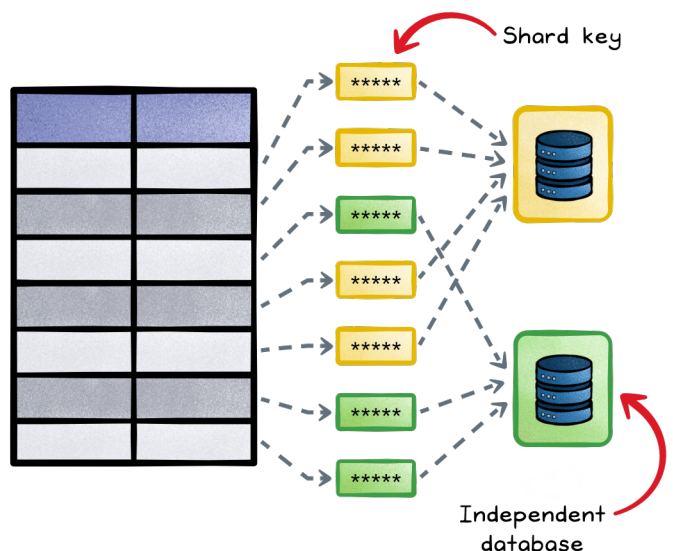
Sharding removes that ceiling by **scaling data and traffic horizontally**, allowing systems to grow far beyond what one machine can handle. It's often the only viable path at true web scale.

## How it works

Sharding partitions data using a **shard key** that determines where each record lives.

Common approaches:

- **Horizontal sharding:** Each shard stores a subset of rows but shares the same schema.



- **Vertical sharding:** Different shards store different tables or columns (often by feature or domain).
- **Hash-based sharding:** A hash of the shard key evenly distributes data across shards.
- **Range-based sharding:** Shards own contiguous key ranges (e.g. user IDs or dates)
- **Directory-based sharding:** A lookup table maps keys to shards for flexible placement.

Applications or routing layers use the shard key to send reads and writes to the correct shard.

---

## Benefits

- **Horizontal scalability** allows you to keep adding capacity by introducing new shards instead of upgrading one massive server.
  - Smaller datasets per shard often lead to **faster queries and higher throughput**, especially under heavy write load.
  - Failures are isolated, so **outages affect only a subset of users**, not the entire system.
  - Spreading load across machines improves **resource utilization** and reduces hot spots on CPU, memory, and disk.
  - At scale, using multiple commodity servers can be **more cost-effective** than a single high-end database machine.
- 

## Tradeoffs

- The biggest cost is **architectural and operational complexity**, since routing, monitoring, and backups now span many databases.
- **Resharding and rebalancing** are risky and expensive when data distribution becomes uneven over time.
- Queries that span shards introduce **cross-shard coordination and higher latency**, especially for joins and aggregates.
- Maintaining **strong consistency across shards** is difficult without heavy distributed transaction mechanisms.
- Once adopted, sharding is hard to undo, creating **long-term lock-in to the chosen shard strategy**.



## When to use it

- Your dataset or write traffic has outgrown what a single database can handle.
  - You need linear write scaling and storage growth.
  - You can shard cleanly by user, tenant, or another high-cardinality key.
- 



## When not to use it

- Vertical scaling, indexing, caching, or read replicas are still sufficient.
- Your queries rely heavily on cross-entity joins.
- The team cannot absorb significant operational and on-call complexity.

# Read Replicas

- i** Read replicas are **read-only copies of a primary database** that stay synchronized with it and are used to serve read queries. All writes still go to the primary, while reads are distributed across replicas.

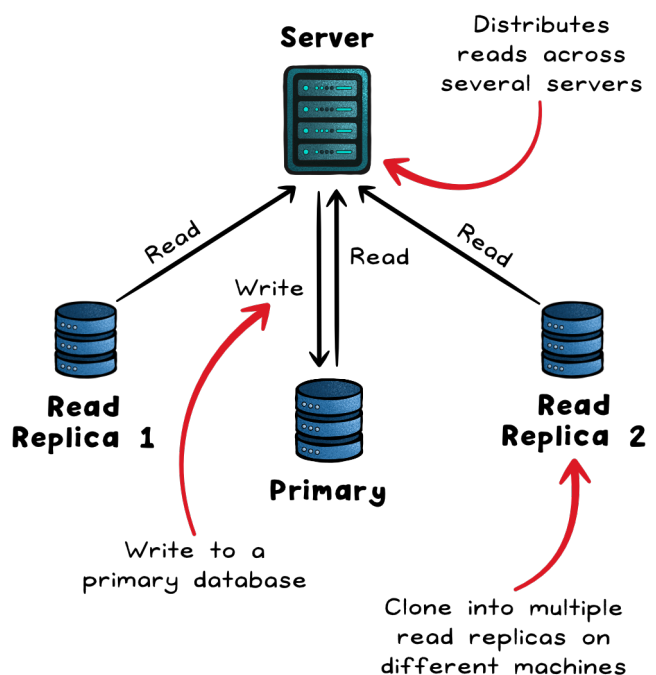
| Benefits                                                                                                                                                  | Tradeoffs                                                                                                                                                    | When to Use                                                                                                                                                          |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• More read capacity</li><li>• Primary stays fast</li><li>• Better availability</li><li>• Geo-local reads</li></ul> | <ul style="list-style-type: none"><li>• Eventual consistency</li><li>• Added ops complexity</li><li>• Higher infra cost</li><li>• No write scaling</li></ul> | <ul style="list-style-type: none"><li>• Read-heavy workloads</li><li>• Stale reads acceptable</li><li>• Need easy failover</li><li>• Reporting &amp; feeds</li></ul> |

## Why it matters

Many systems are **read-heavy**, and letting every query hit the primary quickly becomes a bottleneck. Read replicas increase read capacity and protect write performance, at the cost of sometimes serving slightly stale data.

## How it works

- The primary database accepts all writes and records changes in a replication log.
- Replicas continuously consume that log and apply changes in order, usually via **asynchronous replication**, which keeps writes fast but introduces replication lag.
- Applications route **writes to the primary** and **reads to replicas**, sometimes falling back to the primary for flows that require fresh data (like read-after-write).





## Benefits

- **Higher read throughput** comes from spreading queries across multiple replicas instead of one overloaded primary.
  - Offloading read workloads **protect the primary database's performance**.
  - Replicas provide **built-in redundancy**, enabling faster failover if the primary goes down.
  - Placing replicas closer to users enables **geo-distributed reads** with reduced network latency.
- 

## Tradeoffs

- **Eventual consistency** means replicas may briefly return stale data after recent writes.
  - Routing logic and monitoring introduce **operational complexity** compared to a single database.
  - Every replica increases **infrastructure cost**.
  - Replication traffic adds **overhead on the primary**, especially as replica count grows.
  - Read replicas **do not improve write scalability**, so they won't help if writes are the bottleneck.
- 

## When to use it

- Your workload is **read-heavy**, and the primary is strained by read queries.
  - Slightly stale data is acceptable for most user flows.
  - You want higher availability and easier failover without changing your data model.
  - You need to isolate heavy read or reporting queries from transactional writes.
- 

## When not to use it

- Users must always see **strongly consistent, up-to-date data** (e.g. payments, balances).
- Your main bottleneck is **write throughput**, not reads.
- A simple fix like indexing or caching would solve the performance issue.
- You want to avoid the operational overhead of distributed database management.

# Caching

 Caching stores copies of data or computation results in a faster, closer layer so future requests can be served quickly instead of recomputing or reloading from the source of truth.

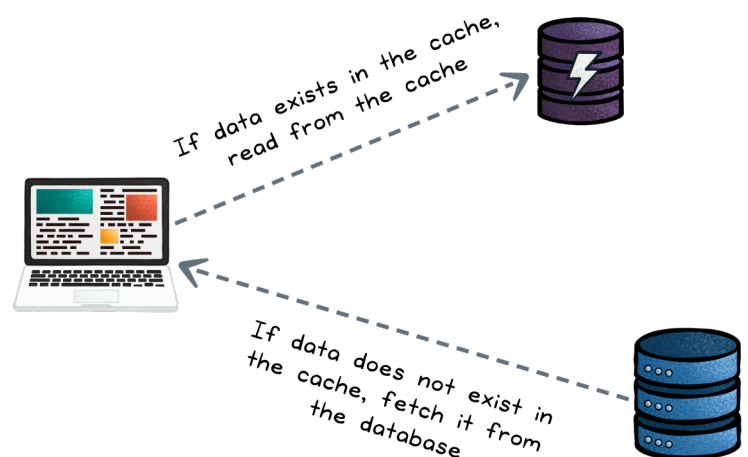
| Benefits                                                                                                                                                                              | Tradeoffs                                                                                                                                                                                | When to Use                                                                                                                                                                 |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Low latency</li><li>• Backend offload</li><li>• Cost reduction</li><li>• Traffic spike protection</li><li>• Predictable performance</li></ul> | <ul style="list-style-type: none"><li>• Stale data risk</li><li>• Invalidation complexity</li><li>• Memory overhead</li><li>• Stampedes on misses</li><li>• Operational burden</li></ul> | <ul style="list-style-type: none"><li>• Read-heavy traffic</li><li>• Expensive data fetches</li><li>• Hot data patterns</li><li>• Eventual consistency acceptable</li></ul> |

## Why it matters

Most systems repeatedly access the same small subset of data, and fetching it from databases or remote services is slow and expensive. Caching exploits this reuse to dramatically reduce latency, protect backend systems, and scale read traffic without linear cost increases.

## How it works

- A cache sits in front of a slower data source.
- On a **cache hit**, the request is served directly from memory or a nearby node.
- On a **cache miss**, data is fetched from the source, returned to the caller, and usually stored in the cache for future requests.



Because cache space is limited, eviction policies (like Least Recently Used or Least Frequently Used) decide what to remove, while TTLs (Time To Live) or invalidation rules control freshness.

## Benefits

- **Lower latency** makes user-facing pages and APIs feel instant by avoiding slow disk or network calls.
  - Offloading reads improves **system scalability**, allowing databases to handle far more traffic with fewer resources.
  - Caching reduces **infrastructure costs** by serving high request volumes from inexpensive memory instead of scaling primary stores.
  - Protecting databases from hot keys increases **reliability under traffic spikes** and prevents cascading failures.
  - Faster, more predictable responses build **user trust and engagement**, especially during peak load.
  - Cached data can act as a short-term buffer, improving **resilience during partial outages**.
- 

## Tradeoffs

- Serving cached responses risks **stale data**, which can violate correctness if freshness matters.
  - **Cache invalidation complexity** makes correctness hard to reason about and easy to get wrong.
  - Large caches introduce **memory and infrastructure cost**, especially at scale.
  - Cold starts and expirations can cause **cache stampedes**, overwhelming backends if not mitigated.
  - Write-back strategies add **data loss risk** if caches fail before persisting changes.
  - Distributed caching increases **operational complexity**, including coherence and monitoring overhead.
- 

## When to use it

- Read-heavy workloads with repeated access patterns.
- Expensive or slow data sources (databases, external APIs, heavy computations).
- Hot data with strong temporal locality.
- Systems that tolerate eventual consistency.



## When not to use it

- Strongly consistent, correctness-critical data (e.g. financial balances).
- Write-heavy workloads with little read reuse.
- Highly unique or one-off requests.
- Systems without capacity to manage cache invalidation safely.

# Chapter 5

# Asynchronous & Distributed Communication

# Synchronous Communication

- i** Synchronous communication is a **blocking request–response interaction** where a caller sends a request and waits until the receiver processes it and returns a result before continuing.

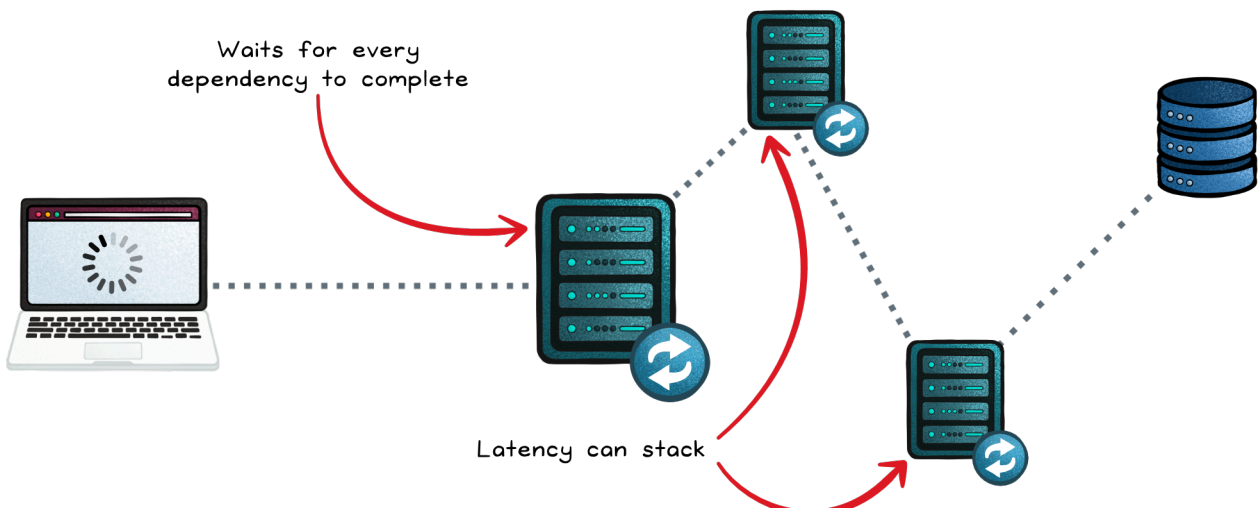
| Benefits                                                                                                                                                                            | Tradeoffs                                                                                                                                                                                | When to Use                                                                                                                                                                                 |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Immediate responses</li><li>• Simple control flow</li><li>• Easy error handling</li><li>• Clear outcomes</li><li>• Low setup cost</li></ul> | <ul style="list-style-type: none"><li>• Tight coupling</li><li>• Cascading failures</li><li>• Limited throughput</li><li>• Latency amplification</li><li>• Inefficient scaling</li></ul> | <ul style="list-style-type: none"><li>• User-facing reads</li><li>• Validation &amp; auth</li><li>• Low-scale systems</li><li>• Fast, reliable deps</li><li>• Early architectures</li></ul> |

## Why it matters

This model shapes how responsive, reliable, and simple a system feels.

It's the default for many user-facing workflows because it provides immediate feedback and a clear success or failure outcome. But it also tightly couples services in time, which can limit resilience and scale if overused.

## How it works



- A service makes a direct call to another service, usually over HTTP or gRPC.
  - The caller pauses execution while waiting for the response.
  - The interaction is typically one-to-one: one request maps to one response.
  - If the downstream service is slow or unavailable, the caller experiences that delay or failure immediately.
- 

## Benefits

- **Immediate feedback** makes it easy to power user-facing actions.
  - The **linear execution flow** is straightforward to reason about, debug, and test.
  - Error handling is simpler because **failures surface instantly** in the same call stack.
  - The caller finishes with a **known outcome** every time the request returns.
- 

## Tradeoffs

- **Tight temporal coupling** requires both services to be healthy simultaneously.
  - Slow or failing dependencies can cause **cascading failures** across request chains.
  - **Throughput is limited** by threads waiting on responses.
  - Chained calls increase **end-to-end latency**.
- 

## When to use it

- User-facing requests that need an immediate answer.
  - Simple read or validation operations.
  - Low-latency internal calls with reliable dependencies.
  - Early-stage systems where simplicity matters more than scale.
- 

## When not to use it

- Long-running or unpredictable operations.
- Systems that must remain available despite partial failures.
- High-throughput or bursty workloads.
- Event fan-out or one-to-many workflows.



# Asynchronous Communication

i Asynchronous communication is a **non-blocking** interaction style where a sender sends a message or event and continues execution without waiting for a response.

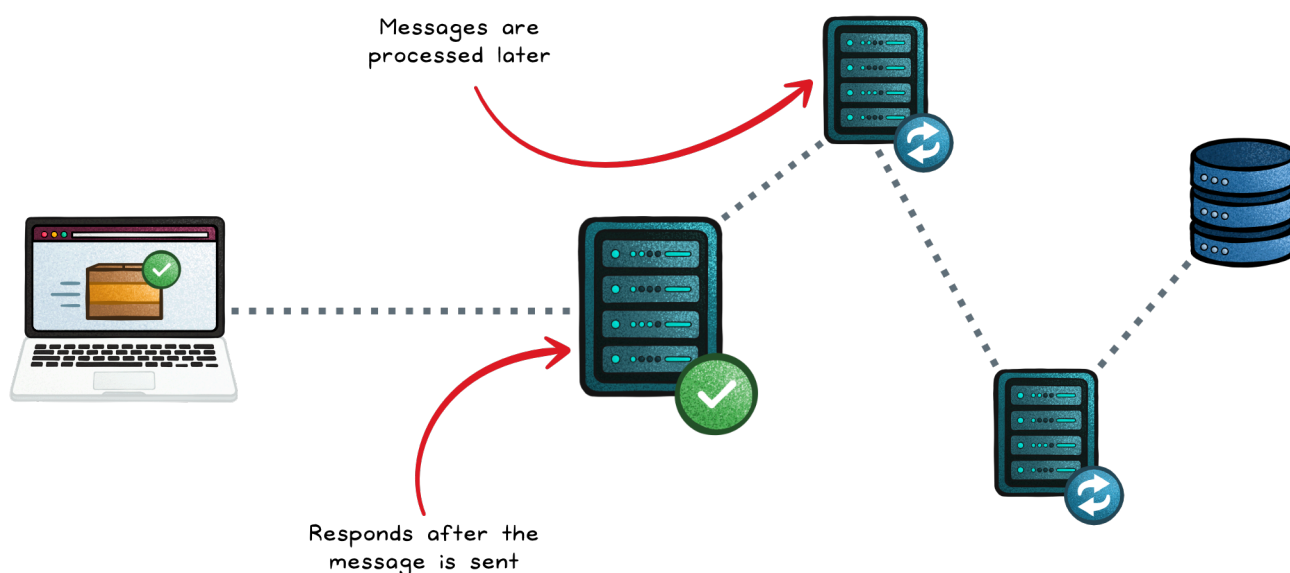
| Benefits                                                                                                                                      | Tradeoffs                                                                                                                                                                                  | When to Use                                                                                                                                                                     |
|-----------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Loose coupling</li><li>• Failure isolation</li><li>• High throughput</li><li>• Easy fan-out</li></ul> | <ul style="list-style-type: none"><li>• Higher complexity</li><li>• Harder debugging</li><li>• Eventual consistency</li><li>• Latency variability</li><li>• Operational overhead</li></ul> | <ul style="list-style-type: none"><li>• Background work</li><li>• Event-driven flows</li><li>• Traffic spikes</li><li>• Async microservices</li><li>• Non-blocking UX</li></ul> |

## Why it matters

As systems scale, waiting on every dependency becomes fragile and expensive.

Asynchronous communication reduces tight coupling, improves resilience under failure, and allows systems to absorb spikes in traffic without collapsing. It's a foundational pattern for building scalable, fault-tolerant distributed systems.

## How it works



- Instead of calling another service directly, a producer publishes a message to a **queue, topic, or event bus**.
  - Consumers process the message independently and on their own timeline.
  - Responses, if needed, happen via **separate messages**, callbacks, or state updates.
  - This enables **one-to-many fan-out**, background processing, and eventual completion rather than immediate results.
- 

## Benefits

- **Loose coupling** allows services to evolve, deploy, and fail independently without taking others down.
  - Systems become more **resilient to partial failures**, since messages can wait safely when consumers are slow or offline.
  - **Higher throughput** is possible because work is queued and processed in parallel instead of blocking threads.
  - User experience improves when **long-running work happens in the background**, with fast acknowledgements up front.
  - Teams gain **extensibility** by adding new consumers without changing existing producers.
- 

## Tradeoffs

- **Increased architectural complexity** from brokers, retries, idempotency, and message ordering concerns.
- Debugging becomes harder due to **non-linear execution paths** and delayed failures.
- You trade immediacy for **eventual consistency**, which can complicate product and UX design.
- **Latency variability** increases since processing time depends on queue depth and consumer speed.
- Strong observability is required to manage **hidden failures and dead-letter queues**.



## When to use it

- Background or **long-running tasks** like emails, media processing, or report generation.
  - **Event-driven workflows** where one action triggers multiple downstream reactions.
  - Systems with **spiky or unpredictable traffic**.
  - Microservices that prioritize **resilience and autonomy**.
- 



## When not to use it

- User flows that require an **immediate, deterministic response**.
- Simple systems where **synchronous calls are sufficient** and scale is limited.
- Workflows that demand **strong consistency guarantees**.
- Teams without the tooling or maturity to operate async infrastructure.

# Publish/Subscribe (Pub/Sub)

**i** Pub/Sub is a messaging pattern where producers publish events to a topic, and consumers subscribe to those topics to receive events asynchronously. Publishers and subscribers never talk to each other directly.

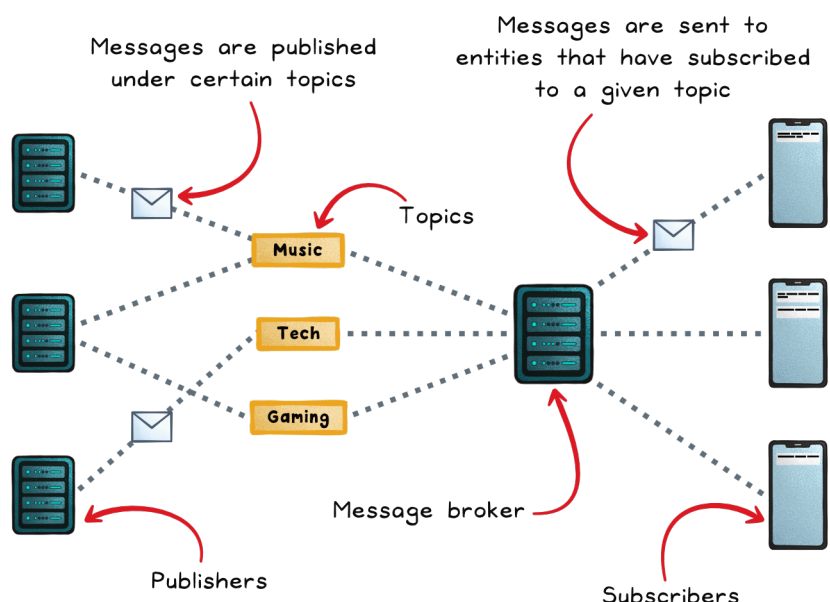
| Benefits                                                                                                                                                                           | Tradeoffs                                                                                                                                                                         | When to Use                                                                                                                                                                                            |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Loose coupling</li><li>• Async, non-blocking</li><li>• Fan-out built in</li><li>• Failure isolation</li><li>• Easy extensibility</li></ul> | <ul style="list-style-type: none"><li>• Harder debugging</li><li>• Duplicate handling</li><li>• Ordering limits</li><li>• Broker overhead</li><li>• Schema coordination</li></ul> | <ul style="list-style-type: none"><li>• One-to-many events</li><li>• Event-driven systems</li><li>• Parallel workflows</li><li>• Real-time updates</li><li>• Eventual consistency acceptable</li></ul> |

## Why it matters

Pub/Sub lets systems scale and evolve without tight coupling between components. It's a foundational pattern for event-driven architectures, where many parts of the system need to react to the same event independently.

## How it works

- A publisher sends a message to a **broker** (or event bus) under a specific topic.
- The broker tracks subscriptions and delivers a copy of that message to every subscriber interested in that topic.



- Publishers fire events and move on. Subscribers process messages on their own timeline, often in parallel.
  - Some systems persist messages for durability, while others only deliver to active subscribers.
- 

## Benefits

- **Loose coupling** allows publishers and subscribers to evolve independently without coordinated changes.
  - Systems stay responsive because **asynchronous communication** avoids blocking critical paths.
  - One event can trigger many workflows thanks to built-in **fan-out**.
  - Failures are isolated, since **slow or broken subscribers** don't impact publishers or other consumers.
  - Teams can move faster by adding new capabilities through **new subscribers**, not new integrations.
  - High throughput becomes achievable with **parallel processing** across subscriber instances.
- 

## Tradeoffs

- Debugging becomes harder due to **indirect execution paths** and hidden consumers.
  - Most systems offer **at-least-once delivery**, requiring idempotent consumers.
  - Message ordering is often **not guaranteed**, especially across partitions.
  - Operating brokers introduces **infrastructure and operational complexity**.
  - Publishers get no feedback, since **fire-and-forget messaging** lacks built-in responses.
  - Teams must manage **schema evolution carefully** to avoid breaking downstream consumers.
- 

## When to use it

- One event needs to trigger **multiple independent actions**.
- You want **event-driven or reactive architectures**.
- High throughput with **parallel consumers** is required.
- Loose coupling and **eventual consistency** are acceptable.



## When not to use it

- You need **strict request/response** semantics.
- Only **one consumer per message** is required.
- Global **ordering guarantees** are critical.
- The system is small and **simplicity matters more than flexibility**.

# Message Queues

**i** A message queue is an **asynchronous communication mechanism** that lets producers send messages to a buffer and consumers process them later.

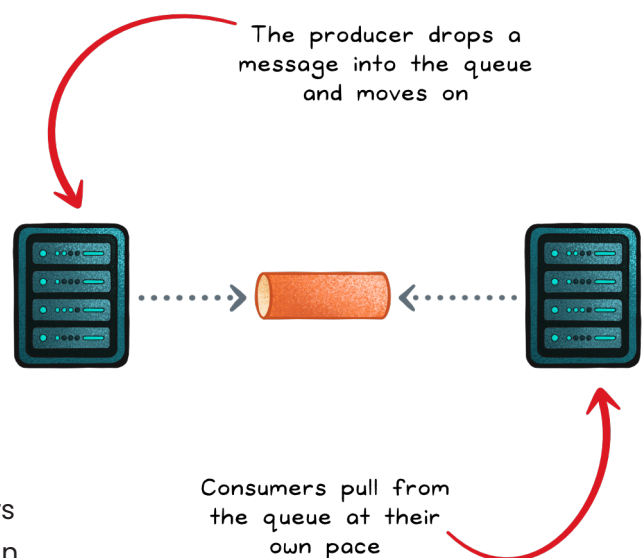
| Benefits                                                                                                                                                                                      | Tradeoffs                                                                                                                                                                                       | When to Use                                                                                                                                                                |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Loose service coupling</li><li>• Load buffering</li><li>• Failure isolation</li><li>• Easy horizontal scaling</li><li>• Reliable processing</li></ul> | <ul style="list-style-type: none"><li>• Added infrastructure</li><li>• Eventual consistency</li><li>• Duplicate handling</li><li>• Ordering complexity</li><li>• Debugging difficulty</li></ul> | <ul style="list-style-type: none"><li>• Background jobs</li><li>• Traffic spikes</li><li>• Async workflows</li><li>• Microservices</li><li>• Guaranteed delivery</li></ul> |

## Why it matters

Message queues decouple system components, which reduces cascading failures and improves resilience. They let systems absorb traffic spikes, offload slow work, and keep user-facing paths fast even when downstream services are slow or unavailable.

## How it works

- Producers enqueue messages into a broker-managed queue.
- Consumers pull messages when ready, process them, and acknowledge completion.
- If processing fails or the consumer crashes, the message can be retried or moved to a dead-letter queue.
- Most systems support competing consumers for horizontal scaling and FIFO ordering within a queue or partition.





## Benefits

- **Loose coupling between services** allows producers and consumers to evolve independently without breaking each other.
  - Systems stay responsive by **offloading slow or expensive work** to background processing.
  - Queues provide **natural load buffering**, smoothing traffic spikes instead of overwhelming downstream systems.
  - **Fault tolerance improves** because messages persist until successfully processed, even across failures.
  - Teams gain **simpler scaling mechanics**, since adding consumers increases throughput without changing producers.
  - Business-critical workflows benefit from **higher reliability and fewer dropped requests**, protecting user trust.
- 

## Tradeoffs

- Introducing a queue adds **operational and architectural complexity**, including brokers, monitoring, and tuning.
  - Asynchronous processing introduces **latency and eventual consistency**, which may be unacceptable for real-time flows.
  - Most systems use at-least-once delivery, so **duplicate messages** must be handled safely by consumers.
  - Maintaining strict ordering while scaling can be difficult, creating **ordering and concurrency challenges**.
  - Debugging becomes harder due to **reduced visibility across async boundaries** and delayed failures.
- 

## When to use it

- You need to **offload background work** like emails, media processing, or data pipelines.
- Traffic is **bursty or unpredictable**, and downstream systems need protection.
- Services must remain **loosely coupled and failure-isolated**.
- Guaranteed, eventually-processed work is more important than immediate results.



## When not to use it

- The flow requires **immediate, synchronous results**.
- Strong consistency is required across multiple steps.
- The system is small and simple, where a queue would be **over-engineering**.
- Ultra-low latency paths where queue overhead is unacceptable.

# Streaming

- i Streaming is a communication model where data flows continuously over **long-lived connections**, instead of being exchanged as one-off requests and responses.

| Benefits                                                                                                                                                                                                                    | Tradeoffs                                                                                                                                                                                                                        | When to Use                                                                                                                                                                                               |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Real-time data flow</li><li>• Decoupled producers/consumers</li><li>• Scales with event volume</li><li>• Durable, replayable events</li><li>• Live analytics &amp; alerts</li></ul> | <ul style="list-style-type: none"><li>• Higher system complexity</li><li>• Resource-intensive infrastructure</li><li>• Ordering &amp; consistency challenges</li><li>• Harder debugging</li><li>• Operational overhead</li></ul> | <ul style="list-style-type: none"><li>• Low-latency requirements</li><li>• Continuous data streams</li><li>• Fan-out to many services</li><li>• Event-driven systems</li><li>• Bursty workloads</li></ul> |



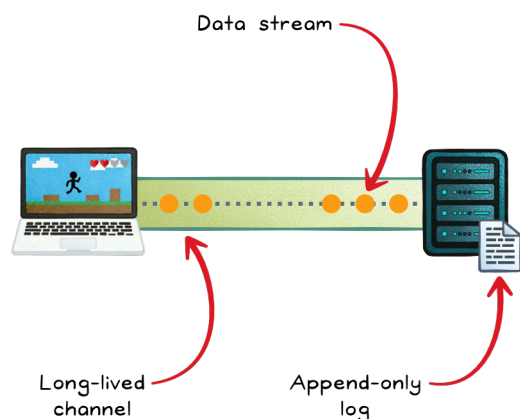
## Why it matters

Many systems generate data constantly and need to react immediately. Streaming enables **near-real-time updates**, supports event-driven architectures, and avoids the latency and overhead of batching or polling.



## How it works

- Producers write events to a stream or channel, backed by an append-only log that preserves order and durability.
- Consumers use persistent connections and process events as they arrive, often using a **pub-sub** model where consumers can react to events independently.
- Because the channel stays open, the system minimizes connection overhead and can apply flow control, retries, and replay when failures occur.



## Benefits

- **Low-latency, real-time delivery** lets systems react to events within milliseconds instead of minutes.
  - Streaming enables **loose coupling between services**, since producers don't need to know who consumes the data.
  - Systems scale naturally because **consumers can be added or removed** to match event volume.
  - Durable streams improve **fault tolerance**, allowing consumers to recover and resume without data loss.
  - Teams gain **timely insights** from live analytics, monitoring, and alerts.
  - Modeling behavior as events creates a **more natural fit for real-world processes** that evolve over time.
- 

## Tradeoffs

- Operating streaming platforms introduces **significant architectural and operational complexity**.
  - **Long-lived connections and buffering** increase memory, storage, and infrastructure costs.
  - Handling **ordering, duplication, and delivery guarantees** requires careful design and discipline.
  - Systems often accept **eventual consistency**, which can complicate reasoning about state.
  - Backpressure and flow control must be tuned to avoid **lag, overload, or dropped events**.
  - Debugging asynchronous flows is harder due to **non-deterministic timing and distributed behavior**.
- 

## When to use it

- You need **real-time or near-real-time updates**.
- Data arrives **continuously and at high volume**.
- Multiple services must react independently to the same event.
- You're building **event-driven or asynchronous microservices**.
- Workloads are **bursty or unpredictable** and must scale quickly.



## When not to use it

- Data changes are **infrequent or delay-tolerant**.
- Simple request/response APIs meet the requirements.
- The team cannot support the **operational overhead**.
- Strong, immediate consistency is mandatory everywhere.

# Chapter 6

# **Security & Compliance**

# Encryption

**i** Encryption is the process of transforming readable data into an unreadable form so only authorized parties can access it.

| Benefits                                                                                                                                                                                             | Tradeoffs                                                                                                                                                                                            | When to Use                                                                                                                                                                        |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Data stays unreadable</li><li>• Reduced breach impact</li><li>• Strong user trust</li><li>• Works across systems</li><li>• Enables secure defaults</li></ul> | <ul style="list-style-type: none"><li>• CPU and latency cost</li><li>• Key management burden</li><li>• Harder debugging</li><li>• Limited data querying</li><li>• Crypto upgrades required</li></ul> | <ul style="list-style-type: none"><li>• Sensitive data</li><li>• Public networks</li><li>• Compliance needs</li><li>• Zero-trust environments</li><li>• Defense in depth</li></ul> |



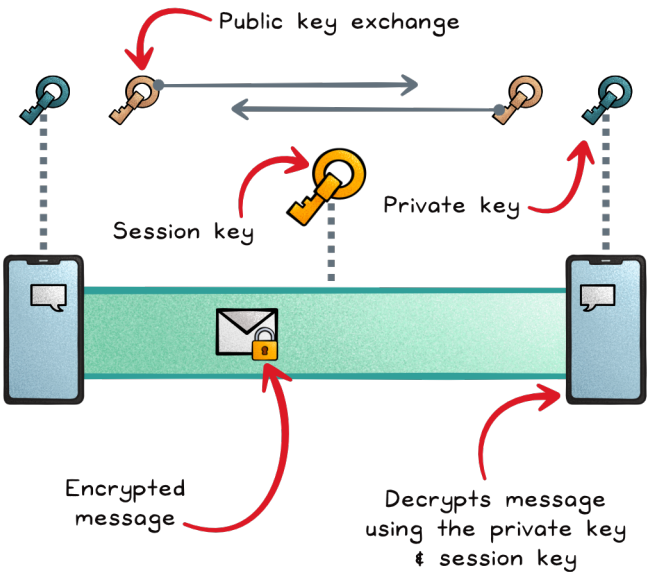
## Why it matters

Modern systems constantly move and store sensitive data. Encryption ensures that even if data is intercepted, leaked, or stolen, it cannot be understood or misused. It is a foundational control for privacy, trust, and regulatory compliance.



## How it works

- A system first **generates encryption keys**, either a public/private key pair or a shared secret key.
- The sender **retrieves and verifies the recipient's public key** or identity.
- A **temporary symmetric session key** is created for this connection or conversation.





- That session key is **securely exchanged or derived** using asymmetric encryption or a key-agreement protocol.
  - Once the key is shared, all data is **encrypted into ciphertext** using fast symmetric encryption.
  - Each message is **decrypted only at the destination**, never in between.
  - Session keys **expire or rotate**, limiting the impact of key compromise.
  - Long-term keys are stored and protected separately through **key management systems**.
- 

## Benefits

- It protects **data confidentiality** even when networks or the system are compromised.
  - Encryption strengthens **user trust** by ensuring private information stays private.
  - It reduces breach impact by turning stolen data into **unusable ciphertext**.
  - Teams can safely operate across **untrusted networks** like the public internet.
  - Enables secure-by-default systems that meet **compliance** needs.
  - Supports integrity and authenticity with **authenticated encryption or signatures**.
- 

## Tradeoffs

- There is **performance overhead** with frequent encryption and decryption.
  - Managing keys introduces **operational complexity** and new failure modes.
  - Lost or compromised keys can result in **permanent data loss or exposure**.
  - Encryption can limit **data usability**, such as searching on encrypted fields.
  - Systems must stay **cryptographically up to date** as algorithms age or weaken.
- 

## When to use it

- Protecting sensitive data at rest, in transit, or across system boundaries.
- Communicating over public or untrusted networks.
- Meeting security, privacy, or compliance requirements.
- Reducing blast radius from inevitable breaches.



## When not to use it

- Data that is intentionally public and low-risk.
- Extremely latency-sensitive paths where encryption is unjustifiable.
- Systems without a plan for key management and recovery.
- When simpler access controls fully address the threat model.

# Tokenization

- i** Tokenization replaces **sensitive data** with a non-sensitive placeholder called a token. The token has no inherent meaning and can only be mapped back to the original value through a secure tokenization system.

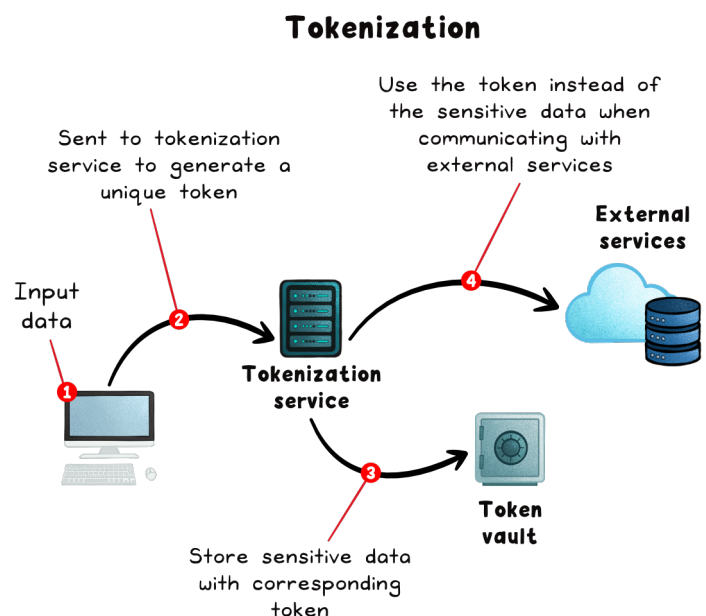
| Benefits                                                                                                                                                                                | Tradeoffs                                                                                                                                                                      | When to Use                                                                                                                                                                            |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Smaller breach blast radius</li><li>• Reduced compliance scope</li><li>• Legacy-friendly formats</li><li>• Least-privilege by default</li></ul> | <ul style="list-style-type: none"><li>• Added service and vault</li><li>• Vault is a prime target</li><li>• Extra latency at scale</li><li>• Ongoing governance cost</li></ul> | <ul style="list-style-type: none"><li>• Highly sensitive data</li><li>• Few systems need plaintext</li><li>• Heavy regulatory pressure</li><li>• Shared or untrusted systems</li></ul> |

## Why it matters

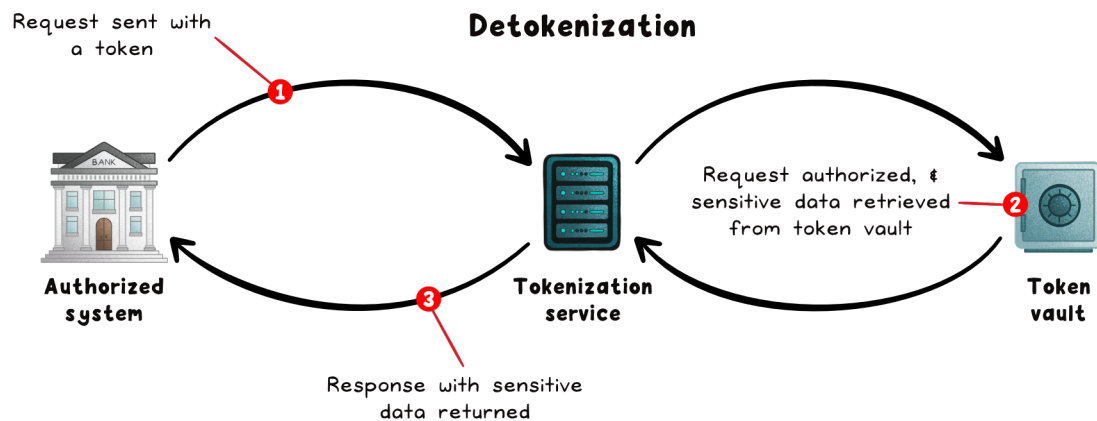
Sensitive data spreads quickly across databases, logs, services, and third-party tools. Tokenization reduces the damage of breaches by ensuring most systems never see real secrets. It also shrinks compliance scope and limits who can access raw data.

## How it works

- An application sends sensitive data to a **tokenization service** instead of storing it directly.
- The service generates a **random or non-reversible token** to represent that value.
- The original data is stored in a **secure token vault** with strict access controls.
- The token is returned and used everywhere else in the system.



→ When needed, authorized systems can **detokenize** by requesting the original value from the vault.



## ★ Benefits

- **Breach impact is dramatically reduced**, because stolen tokens reveal nothing usable on their own.
- Teams gain **stronger compliance posture** by keeping regulated data isolated in one hardened system.
- Tokens can preserve **format and length**, allowing legacy systems and schemas to keep working unchanged.
- Most services operate on tokens, reinforcing **least-privilege access** across the architecture.
- Centralizing sensitive access improves **auditability and monitoring**, making misuse easier to detect.

## ⚖ Tradeoffs

- Introducing a token service and vault adds **architectural and operational complexity**.
- The token vault becomes a **high-value target** that must be extremely well secured.
- Tokenization and detokenization add **latency and throughput overhead** at scale.
- Poor token design can create **integration issues** with systems that expect real data formats.
- Ongoing governance is required to manage **access controls, logging, and data lifecycle**.



## When to use it

- You store **highly sensitive or regulated data** like payment details or personal identifiers.
  - Many systems reference the data, but **few need the real value**.
  - You want to reduce **compliance scope** without breaking existing workflows.
  - Sensitive data flows through **untrusted or third-party environments**.
- 



## When not to use it

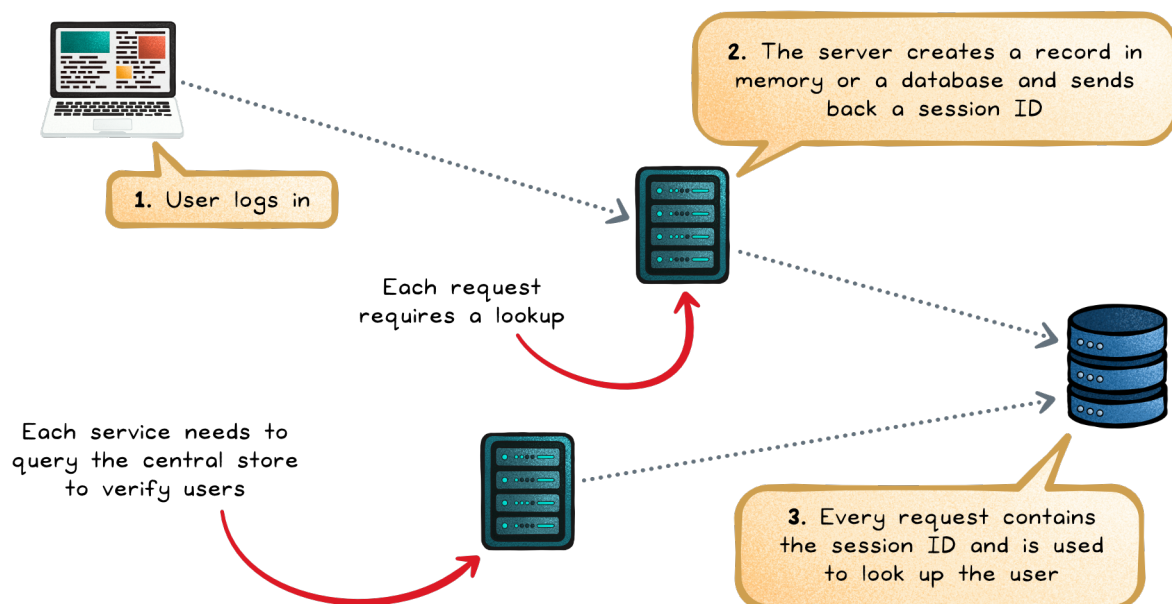
- Data must be **frequently accessed in plaintext** by many systems.
- The data is low risk and **does not justify added infrastructure**.
- You only need **one-way verification**, where hashing is simpler and sufficient.

# JSON Web Tokens (JWT)

**i** JWT is a **compact, self-contained token format** used to transmit identity and authorization data between parties. It is digitally signed so receivers can **verify trust** **without storing server-side session state**.

| Benefits                                                                                                                                                                    | Tradeoffs                                                                                                                                                                           | When to Use                                                                                                                                                              |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Stateless auth</li><li>• Fast verification</li><li>• Easy scaling</li><li>• Cross-domain trust</li><li>• Strong integrity</li></ul> | <ul style="list-style-type: none"><li>• Hard revocation</li><li>• Readable payloads</li><li>• Larger headers</li><li>• Added token operations</li><li>• Stale permissions</li></ul> | <ul style="list-style-type: none"><li>• Microservices</li><li>• APIs &amp; SPAs</li><li>• Mobile clients</li><li>• SSO scenarios</li><li>• Distributed systems</li></ul> |

## Why it matters



JWTs enable **stateless authentication**, which makes systems easier to scale horizontally and simpler to operate across multiple services. They are a foundational building block for **APIs, SPAs, mobile apps, and microservices**, where traditional server sessions become a bottleneck.



# How it works

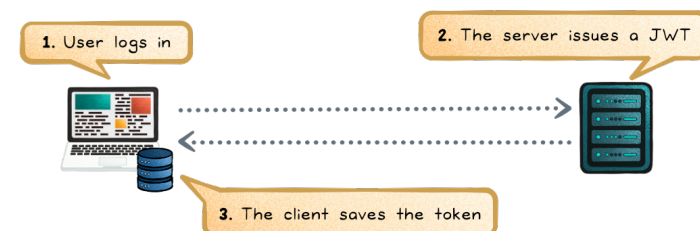
A JWT is a string with three parts:

`header.payload.signature`.

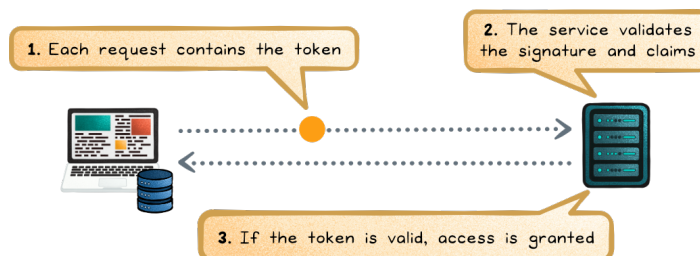
- The **header** declares the token type and signing algorithm.
- The **payload** contains claims like user ID, roles, audience, and expiration.
- The **signature** binds the data, allowing any service with the key to verify integrity and issuer trust.

After login, the client sends the JWT with each request. Services validate it locally; no database lookup required.

## Issuing the Token



## Using the Token



# Benefits

- You don't need to store server-side session state, making **stateless authentication** simpler as systems scale.
- **Requests are faster** because services can trust the token, avoiding extra lookups.
- SSO and microservices are easier to support since **trust can be shared across domains** without sticky sessions.
- The cryptographic signature gives teams confidence that tokens haven't been tampered with, providing **strong integrity guarantees** even in distributed systems.
- JWTs fit naturally into modern stacks thanks to **broad ecosystem support**.



# Tradeoffs

- Once issued, a token is valid until expiry, making **revocation and logout hard** without extra infrastructure.
- Anyone with the token can read its contents since **JWTs aren't encrypted by default**.
- Every request carries the full token, **adding overhead at scale or on slow networks**.
- Real-world setups often require refresh tokens, rotation, and key management, adding complexity.
- Role or permission changes don't apply immediately, causing **stale authorization** until the token expires.





## When to use it

- You need **stateless auth** across APIs, microservices, or serverless workloads.
  - Multiple services must **trust the same identity source** without central session storage.
  - You're building **SPAs, mobile apps, or third-party integrations**.
  - Single Sign-On across domains is a core requirement.
- 



## When not to use it

- You require **immediate logout or forced revocation** with strong guarantees.
- The system is **small, single-server**, and simpler cookie-based sessions are sufficient.
- Tokens would need to carry **highly sensitive or frequently changing data**.
- The team lacks experience operating secure token lifecycles.

# OAuth 2.0

**i** OAuth is an **authorization framework** that lets an application access resources on a user's behalf **without sharing the user's password**.

OAuth's job is **delegating API permissions** ("this app can read your calendar" or "post on your behalf") and **limiting that delegation** by scope, audience, and expiry.

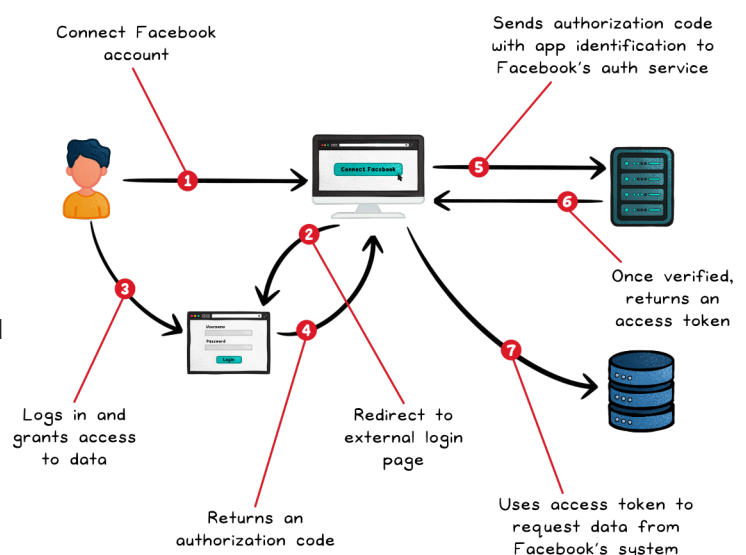
| Benefits                                                                                                                                                                          | Tradeoffs                                                                                                                                                                        | When to Use                                                                                                                                                                                 |
|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• No password sharing</li><li>• Scoped access</li><li>• Scales across services</li><li>• SSO-friendly</li><li>• Secure delegation</li></ul> | <ul style="list-style-type: none"><li>• Flow complexity</li><li>• Token management</li><li>• Harder revocation</li><li>• Misconfig risk</li><li>• Operational overhead</li></ul> | <ul style="list-style-type: none"><li>• API-driven systems</li><li>• User-delegated access</li><li>• Mobile &amp; SPAs</li><li>• Microservices</li><li>• Third-party integrations</li></ul> |

## Why it matters

Modern systems are full of APIs, mobile apps, and third-party integrations that need limited access to data. OAuth provides a safe way to delegate that access while keeping credentials centralized and protected. It's foundational for secure API access, single sign-on flows, and service-to-service communication in cloud-native systems.

## How it works

- A client app wants to call an API on a user's behalf, so it redirects the user to an **Authorization Server** asking for specific **scopes** (permissions).
- The user authenticates (if needed) and approves the requested access (often via a consent screen).
- The authorization server returns an **authorization code** to the client.



- The client exchanges the code for an **access token** (and optionally a refresh token).
  - The client presents the access token to the **resource server (API)**, which validates it and enforces scopes.
- 

## Benefits

- **Passwords stay with the identity provider**, reducing credential leakage risk.
  - Fine-grained **scopes limit blast radius**, so apps only get the access they truly need.
  - Stateless **token access scales well** across APIs, microservices, and global traffic.
  - **Single sign-on experiences** feel seamless across apps without repeated logins.
  - Teams can rotate, revoke, and expire access **without touching user credentials**.
  - OAuth enables secure **machine-to-machine access** with service identities.
- 

## Tradeoffs

- **Conceptual complexity** can trip up teams new to auth flows and token handling.
  - Poorly designed scopes lead to **over-permissioned access** that weakens security.
  - Misconfigured flows or skipped checks create **severe security vulnerabilities**.
  - **Token validation and lifecycle management** add ongoing operational responsibility.
  - Revoking access instantly is harder with **stateless JWTs** unless extra controls are added.
- 

## When to use it

- You need to grant **API access without sharing passwords**.
- Multiple apps or services must act **on behalf of a user**.
- You're building **mobile, SPA, or cloud-native systems**.
- You want fine-grained, revocable access control at scale.



## When not to use it

- Simple internal apps where **session-based auth is enough**.
- Systems that don't expose APIs or third-party access.
- Teams unwilling to manage **token security and validation correctly**.

# OIDC

 OpenID Connect (OIDC) is an **authentication** protocol built on top of **OAuth 2.0** that lets apps **verify user identity** and get basic profile info. It adds **ID tokens** and standard identity claims alongside OAuth **access tokens** for API authorization.

OIDC is one of the **protocols** you can use to implement SSO.

| Benefits                                                                                                                                                                                                         | Tradeoffs                                                                                                                                                                     | When to Use                                                                                                                                                                                |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Stateless JWT identity</li><li>• Centralized login policies</li><li>• Passwords never shared</li><li>• Strong MFA enforcement</li><li>• Faster dev integration</li></ul> | <ul style="list-style-type: none"><li>• Token revocation complexity</li><li>• IdP as failure point</li><li>• More moving parts</li><li>• Ongoing operational burden</li></ul> | <ul style="list-style-type: none"><li>• Web + mobile + APIs</li><li>• Enterprise SSO</li><li>• Cloud-native systems</li><li>• Scalable auth needs</li><li>• Multi-app ecosystems</li></ul> |

---

## Why it matters

Modern systems need login to work across **web apps, mobile apps, and APIs** without sharing passwords. OIDC solves this by turning authentication into a **trusted token exchange**, making single sign-on and identity federation far easier to implement and scale. It's now the default choice for cloud-native and enterprise applications.

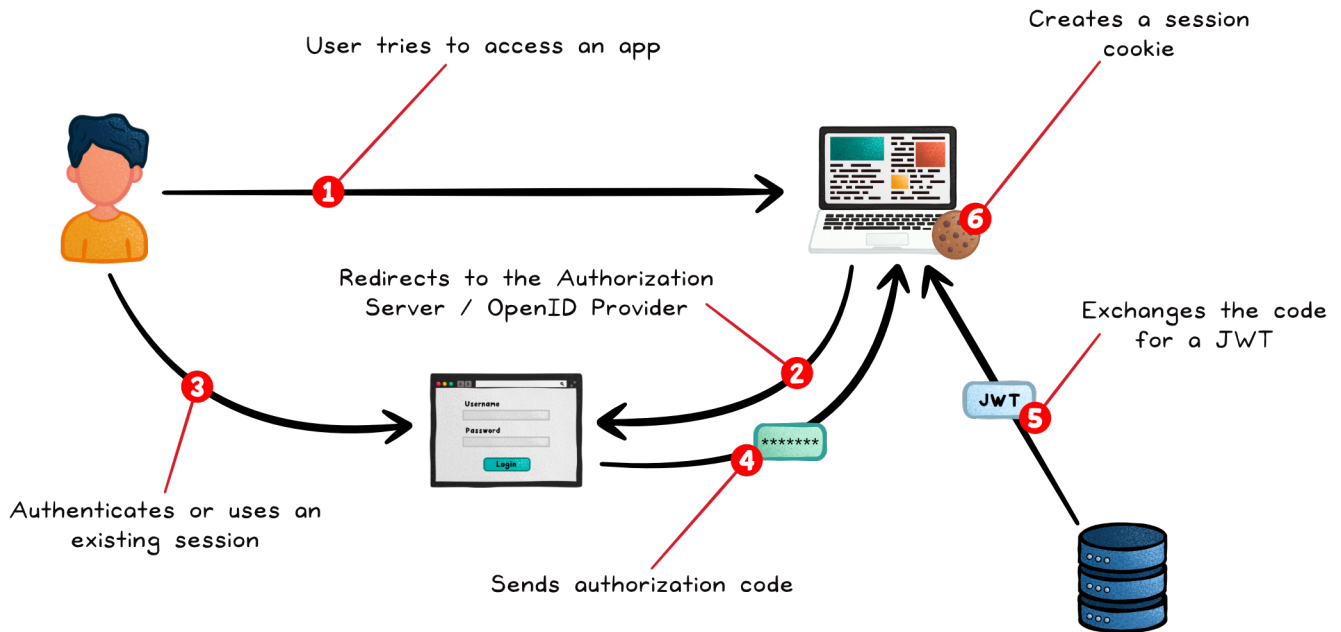
---

## How it works

OIDC extends OAuth with an identity flow:

- A user is redirected from the app to the **Authorization Server/OpenID Provider**.
- The user authenticates (or the provider reuses an existing provider session).
- The provider redirects back with an **authorization code**.
- The app exchanges that code on the back-channel at the token endpoint to get an **ID Token (JWT)** (and often an access token too).

- The app validates the ID Token to confirm user identity (issuer, audience, signature).
- The app creates its own **session cookie**, and optionally keeps tokens for calling APIs.



## ★ Benefits

- **Standard identity tokens** make auth consistent across web, mobile, and APIs.
- **Seamless SSO**, improving trust and reducing login friction.
- **JWT tokens** enable stateless, horizontally scalable systems.
- Centralized login enforces **MFA, device checks, and policies** in one place.
- Teams move faster because OIDC is **developer-friendly and widely supported**.
- Businesses benefit from **lower breach risk** and simpler compliance auditing.

## ⚖ Tradeoffs

- Misconfigured **token validation** can lead to serious security vulnerabilities.
- **JWT revocation is hard**, since tokens are often stateless and short-lived.
- OIDC adds **protocol and operational complexity** over basic sessions.
- Centralized IdPs can become a **single point of failure** without proper redundancy.
- Over-broad scopes or claims can cause **unintended access expansion** over time.



## When to use it

- You need authentication across **APIs, SPAs, and mobile apps.**
  - You want **enterprise SSO** with modern security controls.
  - You're building **cloud-native or microservice-based systems.**
  - You want to avoid handling passwords directly.
- 



## When not to use it

- Very small, internal tools where **simple session auth is sufficient.**
- Systems that cannot support **redirect-based login flows.**
- Legacy platforms tightly coupled to **SAML-only infrastructure.**



# SSO

- i** Single Sign-On (SSO) lets a user authenticate once and then access multiple applications without logging in again. Authentication is centralized and shared across trusted systems.

Single Sign-On (SSO) is a **user experience and system outcome**.

Unlike SAML and OIDC, which are protocols, SSO is the umbrella goal those protocols help implement.

| Benefits                                                                                                                                                                                                         | Tradeoffs                                                                                                                                                                                            | When to Use                                                                                                                                                                                                                   |
|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• One login across apps</li><li>• Consistent security rules</li><li>• Faster onboarding/offboarding</li><li>• Lower password risk</li><li>• Reduced support load</li></ul> | <ul style="list-style-type: none"><li>• Central IdP dependency</li><li>• Added system complexity</li><li>• Login-time latency</li><li>• Harder auth debugging</li><li>• Wider blast radius</li></ul> | <ul style="list-style-type: none"><li>• Many apps, shared users</li><li>• Need MFA &amp; policy control</li><li>• Centralized identity needed</li><li>• UX matters at scale</li><li>• Enterprise or platform setups</li></ul> |



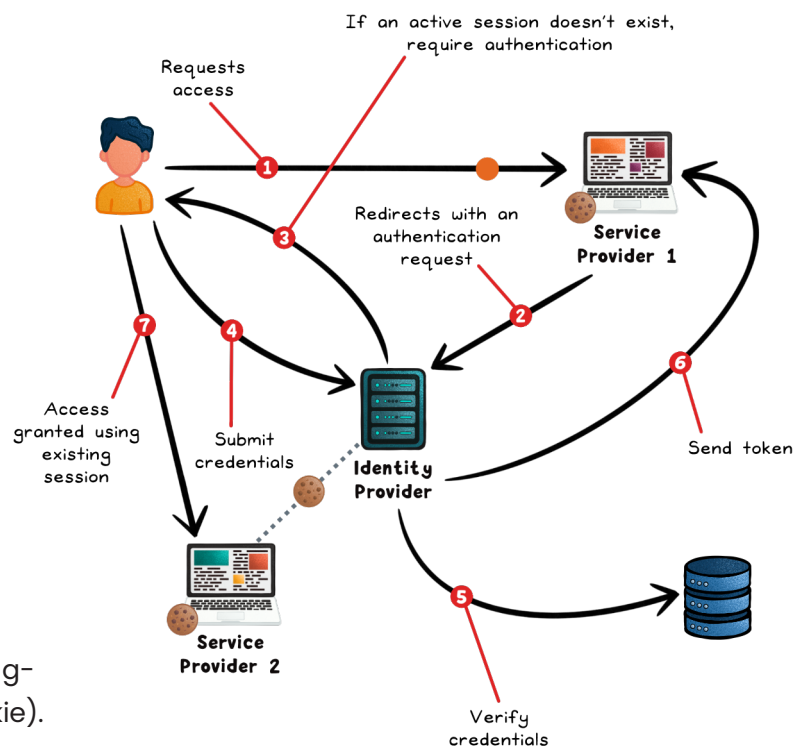
## Why it matters

SSO removes repeated logins, which improves user experience and reduces password fatigue. For teams, it centralizes identity control, making security, access management, and compliance easier at scale.



## How it works

- A user signs in once with a **central identity provider** (IdP) and gets a long-lived sign-in state there (often a cookie).



- When the user opens a second app, that app redirects the browser to the same IdP.
  - The IdP sees the existing **active login session** and doesn't prompt again.
  - The IdP sends a "yes, this user is signed in" result back to the app (via SAML or OIDC).
  - Each app still creates **its own local session** (usually a session cookie); SSO is the experience of not re-entering credentials, not a single shared app session.
- 

## Benefits

- With **frictionless access** across tools, users get less login fatigue and better flow.
  - Centralized authentication improves **security consistency**.
  - Fewer login issues and resets reduce **support overhead** and operational cost.
  - Teams avoid handling passwords directly, lowering the risk of **credential leaks** in individual apps.
  - Account provisioning and deprovisioning become faster, improving **onboarding and offboarding speed**.
- 

## Tradeoffs

- Adding an IdP adds **architectural complexity** over app-local authentication.
  - If the IdP is down, it can **impact many systems at once**.
  - Token validation, redirects, and cryptography add **latency overhead**.
  - Misconfigured trust relationships can lead to **overly broad access** across applications.
  - Debugging authentication issues becomes harder due to **cross-system coupling** and opaque failures.
- 

## When to use it

- Multiple internal or external applications share the same users.
- Security policies like MFA and device checks must be enforced consistently.
- You want centralized user lifecycle management.
- User experience matters across many tools or services.



## When not to use it

- A single, standalone application with simple auth needs.
- Systems that must operate fully offline or independently.
- Extremely latency-sensitive flows where redirects are unacceptable.
- Very small teams where operational overhead outweighs benefits.

# SAML

 SAML is a **federated authentication standard** that lets applications trust a central identity provider to authenticate users on their behalf.

SAML uses **XML-based assertions** and is most common in established enterprise IdP-to-SaaS integrations.

| Benefits                                                                                                                                                                            | Tradeoffs                                                                                                                                                                            | When to Use                                                                                                                                                                                           |
|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <ul style="list-style-type: none"><li>• Central SSO</li><li>• Strong trust model</li><li>• Wide vendor support</li><li>• Fewer passwords</li><li>• Central policy control</li></ul> | <ul style="list-style-type: none"><li>• XML complexity</li><li>• Certificate management</li><li>• Browser-only flows</li><li>• Harder debugging</li><li>• Slower iteration</li></ul> | <ul style="list-style-type: none"><li>• Enterprise SaaS</li><li>• Legacy web apps</li><li>• Existing IdP environments</li><li>• Compliance-driven orgs</li><li>• Stable, long-lived systems</li></ul> |

---

## Why it matters

Before SAML, every app managed its own usernames and passwords, creating security risk and user fatigue.

SAML centralizes authentication, so enterprises can enforce strong login policies once and reuse them everywhere.

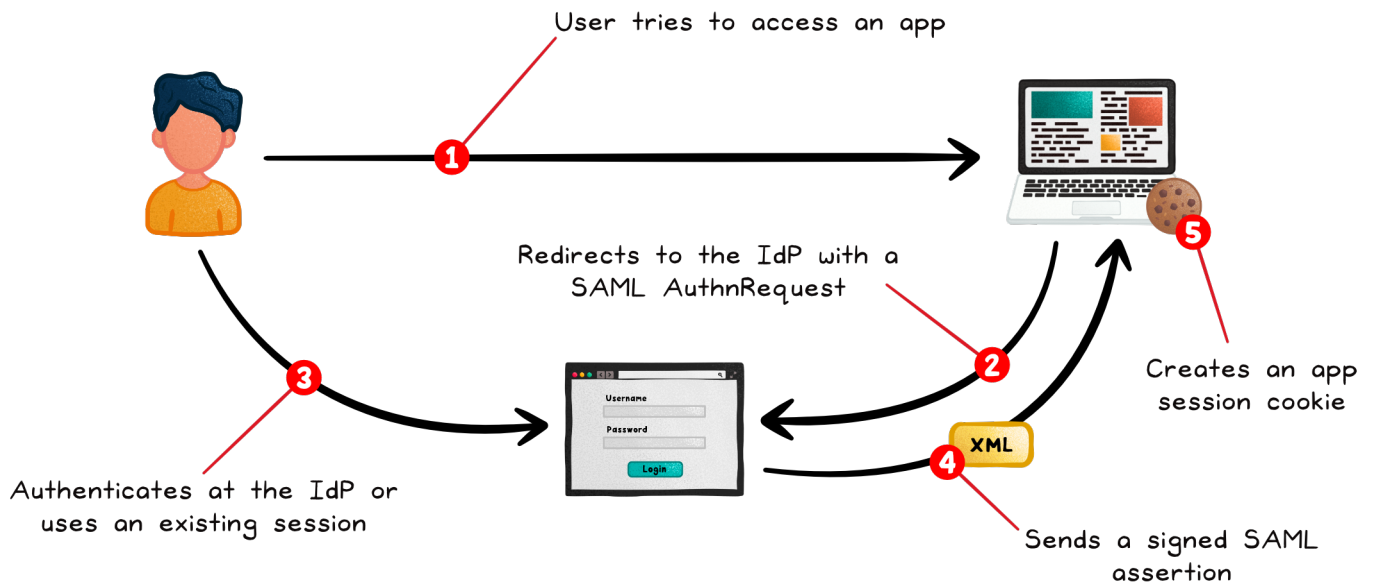
This makes large fleets of SaaS and internal tools **manageable, auditable, and safer**.

---

## How it works

- The user tries to access an app (the **Service Provider**).
- The app redirects the browser to the **Identity Provider** (IdP) with a SAML AuthnRequest.
- The user authenticates at the IdP (password/MFA), or the IdP reuses an existing IdP session.

- The IdP generates a **signed SAML Assertion (XML)** that says “this user authenticated” and may include attributes (email, groups, roles).
- The assertion is delivered through the browser back to the app (often as an HTML form POST).
- The app validates the XML signature, checks audience/time conditions, maps attributes, then creates its own **app session cookie**.



## ★ Benefits

- Centralized authentication enables **true single sign-on**, so users log in once and move across many tools seamlessly.
- Security teams get **one place to enforce MFA and access policies**, instead of configuring every app separately.
- Mature, battle-tested standards make SAML **widely supported by enterprise SaaS vendors**.
- Signed assertions provide **strong non-repudiation**, reducing impersonation risk.
- Fewer passwords improve **user trust and helpdesk load**, lowering reset and lockout costs.



## Tradeoffs

- The **XML-heavy protocol** increases implementation and debugging complexity.
  - Certificate rotation and metadata management add **ongoing operational overhead**.
  - Browser-only redirects make SAML a **poor fit for mobile apps and APIs**.
  - Misconfigurations can lead to **subtle security bugs** that are hard to detect.
  - Compared to OIDC, SAML flows feel **slower and less developer-friendly**.
- 



## When to use it

- You need SSO for **enterprise SaaS or legacy web applications**.
  - You're integrating with customers that already run a corporate IdP.
  - Browser-based access is the primary interaction model.
  - Stability and compatibility matter more than developer ergonomics.
- 



## When not to use it

- You're building **mobile apps, SPAs, or API-first systems**.
- You need lightweight, token-based auth across microservices.
- Your team lacks experience managing certificates and XML security.
- You want a future-forward identity layer with minimal protocol friction.



That's a wrap on the

# System Design Handbook

For more system design insights,  
join 400,000+ engineers already following us:

👉 Follow **Nikki Siapno** on **LinkedIn** (turn on notifications) 🔔

👉 Follow **Nikki Siapno** on **X/Twitter** (turn on notifications) 🔔

👉 **YouTube, Instagram, TikTok** coming soon... 🎬

Stay sharp. Keep leveling up! 🚀

