Why API Gateway Came Into Existence ?

Stage 1: The Monolith Era (No Gateway Needed)

What was good ?

Single codebase
One endpoint
Simple auth
Easy deployment

Why no gateway?

Because:

One service
One URL
One security boundary

Gateway would add no value here

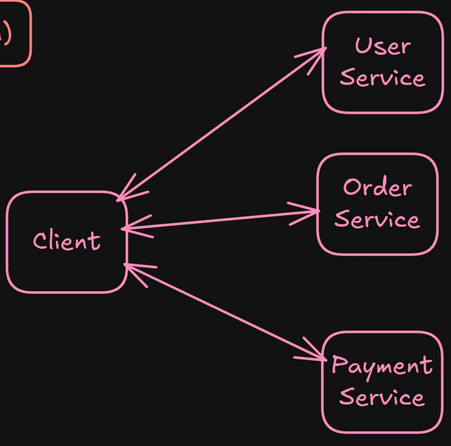
Stage 2: Services Started Splitting (Problems Begin)

Companies broke monoliths into multiple services.

Real-world example (E-commerce app)

Mobile app needs homepage:

User profile → User Service
Cart → Cart Service
Orders → Order Service
Offers → Offer Service



Problems that appeared

Problem 1: Too many network calls

Mobile app makes 4–5 API calls just to load one screen.

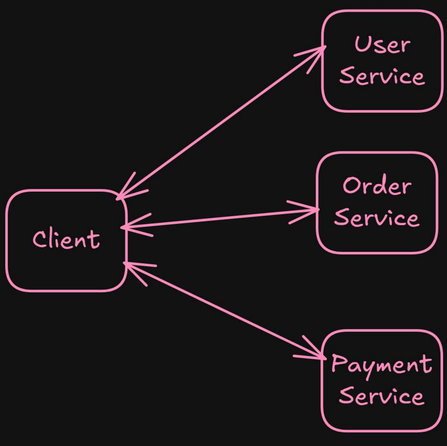
Problem 2: Client must know everything

All service URLs
All API formats
All auth rules

Problem 3: Security nightmare

Every service implements:
JWT validation
Rate limiting
Logging
IP filtering

-> Duplicate logic everywhere.



Problem 4: No central control

No global rate limit
No traffic shaping
No monitoring at one place

Problem 5: Changing backend breaks clients

Rename one API → app crashes.

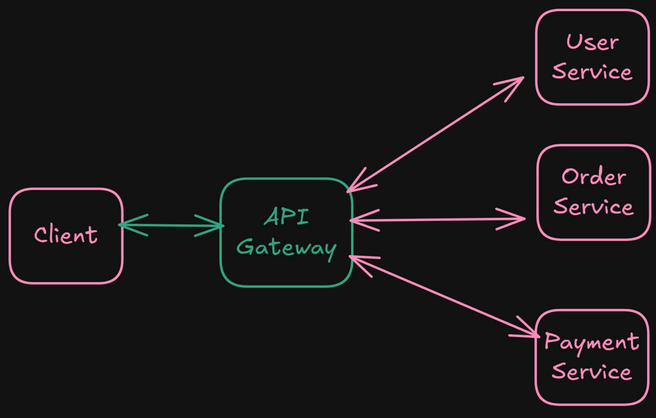
This Is Where API Gateway Was Born

Core idea: Clients should NOT talk to services directly

So we introduced a single entry point.

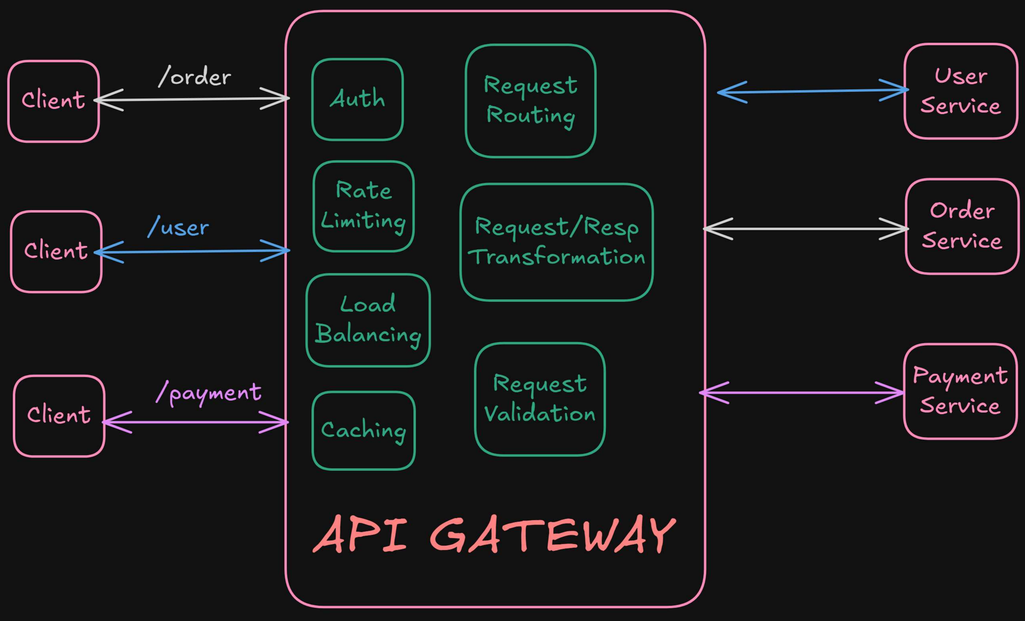
API Gateway (The Turning Point)

An API Gateway is a single entry point for all client requests going to backend services (usually microservices).



Problem	Solution
Multiple endpoints	Single endpoint
Repeated auth	Centralized auth
No rate limit	Global throttling
Chatty APIs	Request aggregation
Versioning pain	Central versioning
Security leaks	One security wall

Core Responsibilities of an API Gateway



Request Routing

Routes incoming requests to correct backend service.
Routing is not just URL-based forwarding.

Gateway can route based on:

Path (/users)
HTTP method (GET, POST)
Headers (x-version, x-platform)
Query params
User identity
Region
Canary / A-B testing rules

Example:

/v1/orders → Order Service v1
/v2/orders → Order Service v2
Header: x-beta=true → Canary service

Authentication & Authorization

Handled before request reaches services.
Common mechanisms:

JWT validation
OAuth 2.0
API Keys
mTLS (Mutual TLS)

Backend services trust the gateway

Request Validation

Gateway validates:

- Required headers
- Query params
- JSON schema
- Data types
- Max length / ranges
- Content-Type
- API contract

Example :

```
POST /orders
{
  "userId": "string" ❌ (should be number)
}
```

Gateway validation ≠ business validation
(no DB checks here)

API Versioning & Compatibility

Gateway:

- Routes versions
- Supports multiple versions simultaneously
- Helps gradual migrations

Example:
/v1 → legacy service
/v2 → new service

Rate Limiting & Throttling

Why at gateway?
- Central point
- Per-user, per-IP, per-token control

Types
- Hard limit (reject after limit)
- Soft throttling (slow down)

Algorithms
- Token bucket
- Leaky bucket
- Fixed window
- Sliding window

Example:

100 req/min per user
10 req/sec per IP

Caching

Gateway-level caching
- GET responses
- Frequently accessed data

Cache strategies
- TTL-based
- Header-based (Cache-Control)
- Key-based

Gateway cache ≠ business cache (Redis)

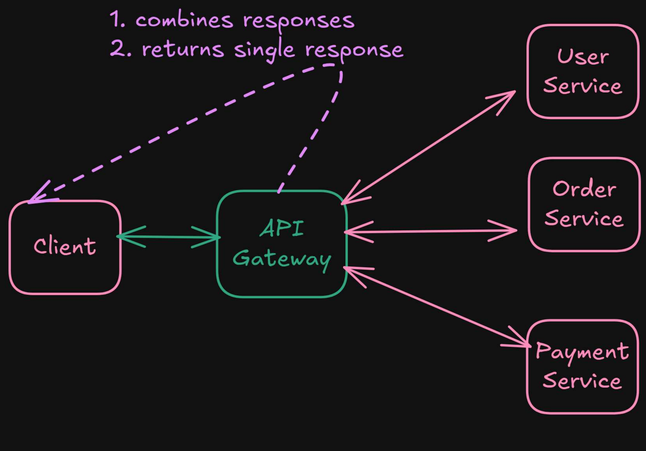
API Aggregation (BFF Pattern)

Why needed
Clients (especially mobile) should not:

Call 5 services
Handle failures individually

Gateway:

Calls multiple services
Combines responses
Returns a single payload



Trade-off

✔ Fewer network calls
❌ Gateway complexity increases

Request / Response Transformation

What gateway can transform:

- Headers
- Query params
- Payload fields

Examples

REST → gRPC

XML → JSON

Internal field names hidden from client

This helps:

Hide internal schema

Decouple client from service changes

Popular API Gateway Implementations

Cloud Managed

- AWS API Gateway
- Azure API Management
- GCP API Gateway

Open Source

- Kong
- NGINX
- Traefik
- Spring Cloud Gateway
- Envoy

Request Flow (Step-by-Step)

Client sends request

Gateway authenticates

Rate limit check

Request validation

Route to service

Retry / timeout handling

Response transformation

Caching (optional)

Response to client

Failure Handling in API Gateway

Timeouts
- Protects clients from slow services

Retries
- Safe for idempotent requests (GET)

Fallbacks
- Return cached or default responses

Security at API Gateway

Handled at gateway:

- SSL termination
- JWT validation
- API keys
- IP whitelisting
- WAF (SQLi, XSS)

Services stay clean & simple

Performance Considerations

Pros

- Reduced client complexity
- Caching
- Aggregation

Cons

- Extra network hop
- Potential bottleneck

Mitigations

- Horizontal scaling
- Stateless gateways
- CDN integration

Common Anti-Patterns ❌

❌ Putting business logic in gateway

❌ Over-aggregating responses

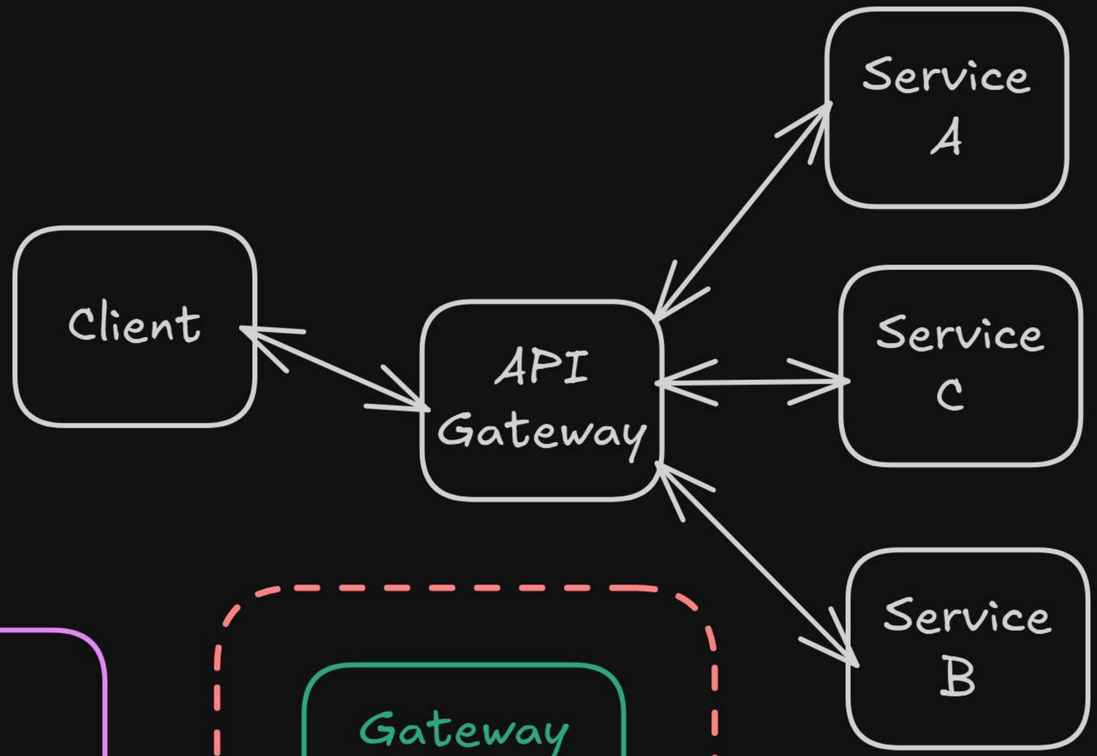
❌ Single gateway without scaling

❌ No timeout / retry limits

Is API Gateway a Single Point of Failure (SPOF)?

Logically single ≠ physically single

An API Gateway is a logical single entry point,
but never deployed as a single machine.

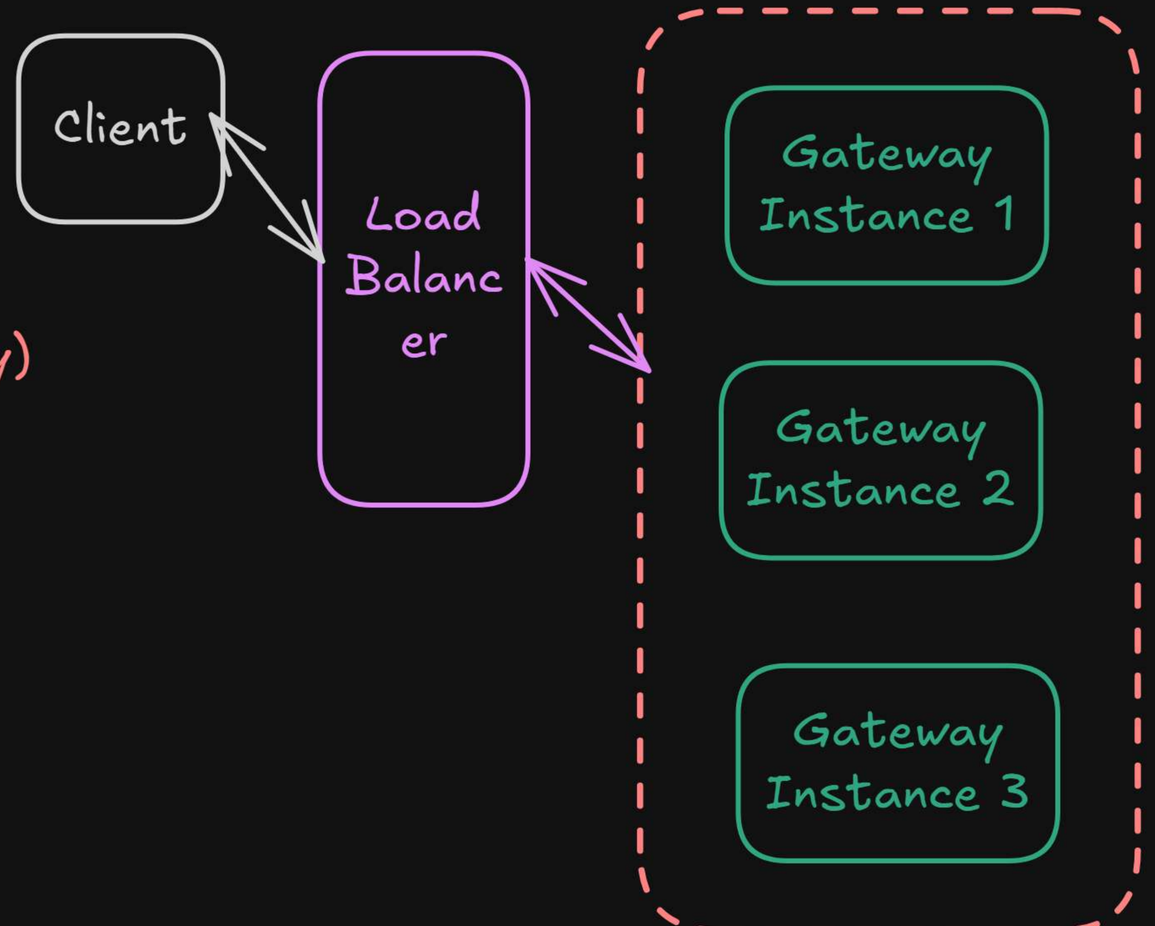


How API Gateway Is Designed NOT to Be a SPOF

1. Horizontal Scaling (Stateless)

API Gateways are:

- Stateless
- Horizontally scalable



2. Managed Gateways (AWS API Gateway)

AWS API Gateway:

- Runs across multiple AZs
- Auto-scales
- No single server

3. Redundancy Across Availability Zones