SOFTWARE ARCHITECTURE

Introduction to Microservices Architecture

From First Principles — Monoliths, Services, Gateways & Domain Design

Monolith vs MSA | Characteristics | API Gateway | DDD | Benefits & Challenges

SETTING THE STAGE

Before microservices — what problem are we actually solving?

The Core Problem

As software grows, teams struggle to build, deploy, and scale it. A single change can break everything. Deployments become terrifying. Teams block each other.

The Old Answer

Build everything together — one codebase, one database, one deployment. Simple at first, but complexity grows faster than the team can manage.

The New Answer

Break the system into small, independently deployable services — each owning its own data, logic, and lifecycle. Deploy them separately, scale them separately.

Architecture is the set of decisions that are hard to change later. Choose wisely.

WHAT IS A MONOLITH?

A monolith is ONE deployable unit containing ALL functionality:

MONOLITH

- UI Layer
- Business Logic
- Data Access Layer
- Single Shared Database

⚠️ *NOT BAD BY DEFAULT — monoliths are often the RIGHT choice for small teams and early products.*

Single Process

Everything runs in one process. One crash = total outage.

Shared Memory

All modules share the same memory space and data.

Deployed Together

Change one line? Redeploy the entire system.

One Codebase

UI, logic, and DB access all live in the same repo.

WHEN THE MONOLITH BREAKS DOWN

Monoliths don't fail overnight. They rot slowly. Watch for these warning signs:

Deployment Fear

Deploying even a small bug fix requires redeploying the entire application. Teams deploy less frequently, accumulating risk.

Team Coupling

50 engineers editing the same codebase means constant merge conflicts, blocked pipelines, and coordination overhead.

Database Bottleneck

One shared database becomes a performance chokepoint. A slow query in the reporting module slows down checkout.

Scaling Mismatch

You can't scale just the payment service — you must scale the entire monolith, wasting CPU on modules with zero load.

Tech Lock-in

The whole team is stuck on one language, one framework. Adopting new tech means a full rewrite.

Cognitive Load

No one understands the whole system anymore. A change to order logic might break the notification system — no one knows why.

CAN A TODO APP BE A MICROSERVICE?

THE HONEST ANSWER

Technically? Yes. Practically? Probably not worth it.

Microservices introduce massive overhead:

- Service discovery & registration
- Inter-service communication (HTTP/gRPC)
- Distributed tracing & monitoring
- Container orchestration (Kubernetes)
- CI/CD pipeline per service
- Network latency for every call

For a todo app with 1–2 devs: the operational cost > the benefit.

WHEN MSA MAKES SENSE

Multiple Independent Teams

5+ teams need to work without blocking each other.

Different Scaling Needs

Video encoding needs 10x more CPU than user auth.

Different Reliability SLAs

Payment must be 99.99% — search can tolerate downtime.

Different Tech Requirements

ML service in Python, API in Go, frontend in Node.

Regulatory Isolation

PCI/HIPAA compliance needs strict data boundary enforcement.

MICROSERVICES: DEFINED

"Microservices is an architectural style where a large application is built as a suite of small, independently deployable services, each running its own process."

Single Responsibility

Each service does ONE thing and does it well. The User Service handles users. The Order Service handles orders. No crossing boundaries.

Own Your Data

Each service owns its own database. Service A cannot directly query Service B's DB. Data flows through APIs — this is non-negotiable.

Communicate via API

Services talk to each other through well-defined interfaces — REST, gRPC, or message queues. Never shared memory, never shared DB tables.

Independent Deployment

Deploy the Payment Service without touching the Inventory Service. If it's not independently deployable, it's not truly a microservice.

Failure Isolation

If the Recommendation Service crashes, checkout still works. Services must be built to handle downstream failures gracefully.

Technology Freedom

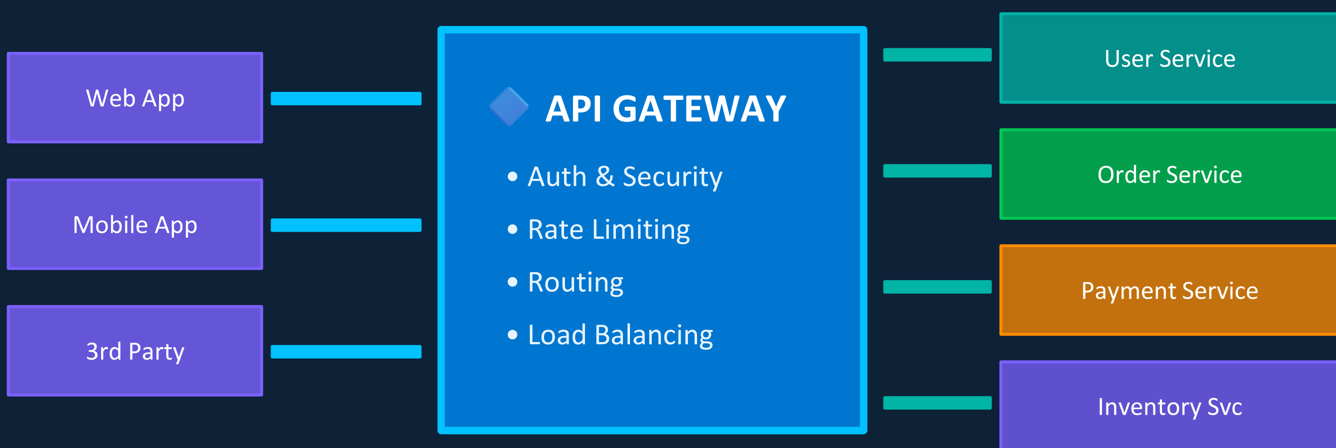
Each team picks the best tool for their job. Python for ML, Go for high-throughput APIs, Node.js for real-time — all within one system.

MONOLITH vs MICROSERVICES

MONOLITH	VS	MICROSERVICES
Single repo, grows without bound	**Codebase**	One repo (or monorepo) per service
Full redeploy for every change	**Deployment**	Deploy only the changed service
Scale the entire app	**Scaling**	Scale individual services as needed
One shared DB for everything	**Database**	Each service has its own DB
One bug can take down everything	**Failure**	Failures are isolated per service
Best for 1–5 developers	**Team Size**	Scales to 50+ developers
Simple to start, hard to maintain	**Complexity**	Complex to start, better at scale

THE API GATEWAY

Problem: With 20 services, clients would need to know ALL their addresses, ports, auth methods, and APIs.



Popular Gateways: Kong, AWS API Gateway, Spring Cloud Gateway, NGINX, Traefik, Envoy, Istio

Single Entry Point

Clients call ONE URL. The gateway knows how to route requests to the right service internally.

Authentication

Validates JWT/OAuth tokens ONCE at the gateway. Internal services trust the gateway — no duplicate auth code in every service.

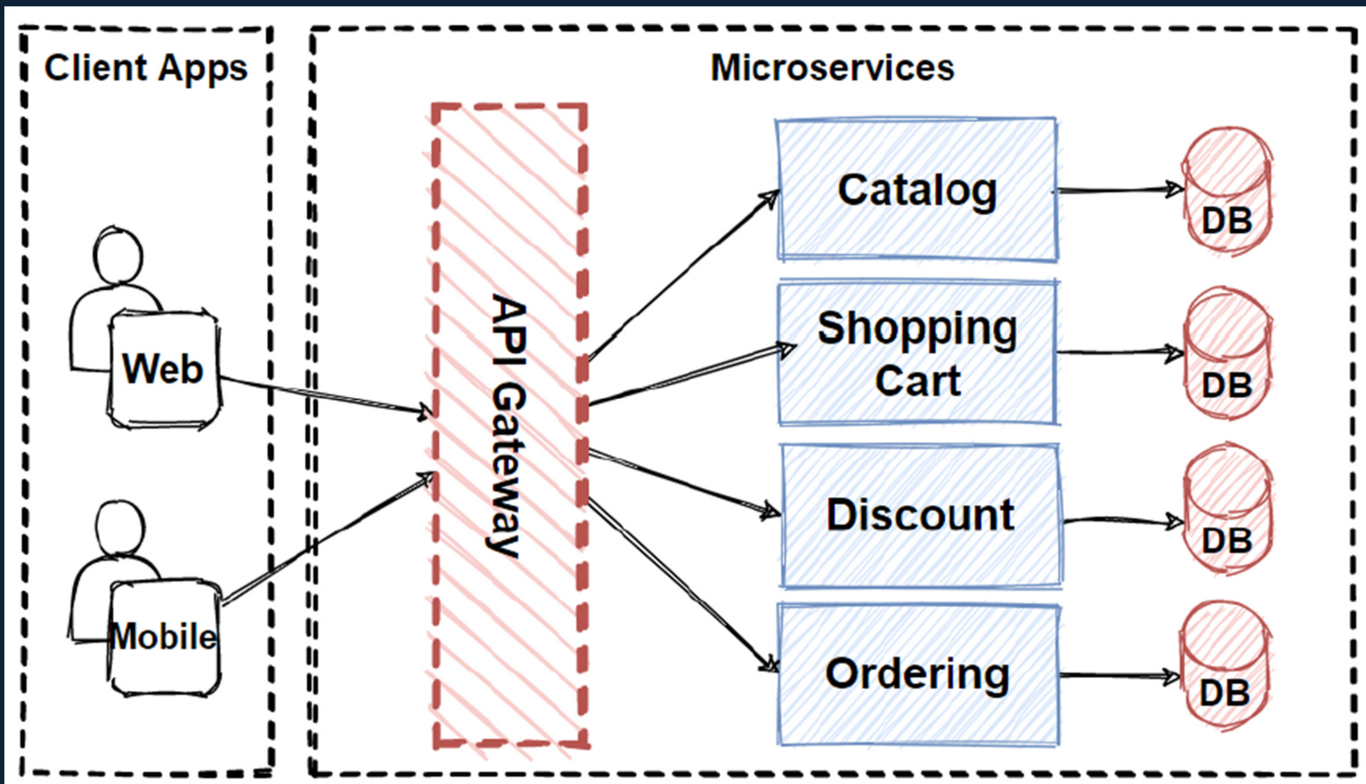
Rate Limiting

Throttle abusive clients at the edge before they hammer your services. Set per-user, per-IP, or per-endpoint limits.

Protocol Translation

Clients use REST. Internal services use gRPC. The gateway translates — clients never need to know.

THE API GATEWAY



Popular Gateways: Kong, AWS API Gateway, Spring Cloud Gateway, NGINX, Traefik, Envoy, Istio

Single Entry Point

Clients call ONE URL. The gateway knows how to route requests to the right service internally.

Authentication

Validates JWT/OAuth tokens ONCE at the gateway. Internal services trust the gateway — no duplicate auth code in every service.

Rate Limiting

Throttle abusive clients at the edge before they hammer your services. Set per-user, per-IP, or per-endpoint limits.

Protocol Translation

Clients use REST. Internal services use gRPC. The gateway translates — clients never need to know.

LEARN WITH NIKHIL

MICROSERVICES CHARACTERISTICS

1

Small & Focused

Can be rewritten in 2 weeks. If it takes longer, it's too big.

2

Loosely Coupled

Change Service A without touching B. True independence.

3

Highly Cohesive

Related logic stays together. Unrelated logic lives elsewhere.

4

Decentralized Data

No shared databases. Each service is the owner of its domain data.

5

Failure Tolerant

Designed to degrade gracefully when dependencies fail.

6

Independently Scalable

Scale out the bottleneck, not the whole system.

7

Observable

Metrics, logs, traces per service. If you can't see it, you can't fix it.

8

Owned by One Team

Conway's Law: your architecture mirrors your org chart. One service = one team.

INTER-SERVICE COMMUNICATION

How do services talk to each other? There are two fundamentally different patterns:

SYNCHRONOUS (Request/Response)

REST over HTTP/HTTPS

Most common. JSON payloads, standard HTTP verbs (GET, POST, PUT, DELETE). Simple, universal, browser-friendly.

gRPC (Protocol Buffers)

Binary protocol, 5–10× faster than REST. Strongly typed, auto-generated clients. Ideal for internal service calls.

GraphQL

Client specifies exactly what data it needs. Great for API gateways that aggregate multiple services.

⚠️ *Caller waits for response. Creates temporal coupling.*

ASYNCHRONOUS (Event-Driven)

Message Queues (RabbitMQ)

Service A publishes a message. Service B consumes it when ready. A and B never need to be online simultaneously.

Event Streaming (Kafka)

Services publish events to topics. Multiple consumers process the same event independently. Replay-able, durable, high-throughput.

Pub/Sub Pattern

One publisher, many subscribers. Order placed → triggers fulfillment, notification, analytics simultaneously.

✅ *Caller doesn't wait. Services are truly decoupled.*

DOMAIN-DRIVEN DESIGN (DDD)

DDD answers the hardest microservices question: How do you decide what each service should own?

Ubiquitous Language

Developers and domain experts use the SAME vocabulary. An 'Order' in the code means exactly what a 'Order' means to the business. No translation layer.

Bounded Context

A boundary within which a model is consistent and valid. 'Customer' means different things in Sales vs. Shipping — separate bounded contexts, separate models.

Aggregates

A cluster of domain objects treated as a single unit for data changes. An Order Aggregate includes OrderLines, but they cannot exist without the Order.

Domain Events

Something important that happened: OrderPlaced, PaymentFailed, ItemShipped. Events drive the system forward without tight coupling.

Example: E-commerce bounded contexts → Customer Management | Order Processing | Inventory | Payment | Shipping | Notifications

BOUNDED CONTEXTS IN PRACTICE

The same word means different things in different contexts. DDD makes this explicit.

What does 'Product' mean?

Product Catalog

```
Product {
  productId
  name
  description
  images
  SEO tags
  category
}
```

Inventory

```
Product {
  productId
  warehouseLocation
  stockCount
  reorderLevel
  supplier
}
```

Pricing

```
Product {
  productId
  basePrice
  discounts
  taxClass
  currency
}
```

Shipping

```
Product {
  productId
  weight
  dimensions
  hazmat?
  shippingClass
}
```

💡 Key Insight: Each service maintains its own model of 'Product'. They share only the productId. This prevents the 'god object' antipattern and keeps each service truly independent.

BENEFITS OF MICROSERVICES

Independent Scaling

Your payment service getting hammered on Black Friday? Scale it to 100 instances. Your blog service? Keep it at 1. Pay for what you use.

Faster Deployments

Ship the checkout service without touching search. Teams deploy multiple times per day without coordination theater.

Fault Isolation

When recommendations service crashes, users can still buy. Failures stay contained. No cascading system-wide outages.

Technology Flexibility

Use the right tool: Python for ML, Go for APIs, React for UI, Rust for performance-critical services. Best tool per job.

Team Autonomy

Each team owns, builds, deploys, and monitors their service end-to-end. No waiting for other teams. True DevOps culture.

Easier Modernization

Replace one service at a time. You don't need to rewrite the whole system — just the parts that need it. The strangler fig pattern.

CHALLENGES OF MICROSERVICES

Microservices are not free. Here's what you're signing up for:

Distributed System Complexity

Network calls fail. Services are unavailable. Data is eventually consistent. You've traded local function calls for distributed computing — with all the fallibility that entails.

Operational Overhead

Each service needs its own CI/CD pipeline, monitoring, alerting, deployment config, and runbooks. 20 services = 20x the ops burden.

Distributed Tracing

A single user request touches 8 services. When it fails, which one caused it? You need distributed tracing (Jaeger, Zipkin) to even answer that question.

Data Consistency

Two-phase commits don't work across services. You'll need Sagas, eventual consistency, and compensating transactions. Bugs here are subtle and catastrophic.

Testing Complexity

Unit tests are easy. Integration tests across services require full environments. Contract testing (Pact) helps but adds tooling overhead.

Service Discovery

Services are ephemeral — their IPs change. You need a service registry (Consul, etcd, Kubernetes DNS) so services can find each other dynamically.

OBSERVABILITY & SERVICE MESH

"You can't fix what you can't see." — The Three Pillars of Observability

LOGS

What happened?

Structured logs from every service. Centralize in Elasticsearch or Splunk. Search across ALL services in one query to trace an error.

ELK Stack · Splunk · Loki

METRICS

How is it behaving?

CPU, memory, request rate, error rate, p99 latency. Track over time. Alert when thresholds breach. Dashboards per service.

Prometheus · Grafana · Datadog

TRACES

Where did time go?

A trace follows ONE request across ALL services. See exactly which service added latency. Identify bottlenecks surgically.

Jaeger · Zipkin · AWS X-Ray

SERVICE MESH (Istio / Linkerd)

Istio · Linkerd · Consul Connect

A service mesh adds a sidecar proxy to each service. It handles: mTLS encryption between services, automatic retries & circuit breakers, load balancing, traffic shaping, and telemetry collection — all WITHOUT changing your application code.

MONOLITH → MICROSERVICES

Don't rewrite everything at once. Use the Strangler Fig Pattern — extract services one at a time:

1. **Identify Bounded Contexts**

   Map your monolith's domains using DDD. Find the seams — where data models don't overlap, where teams naturally divide, where scale requirements differ.

2. **Modularize the Monolith First**

   Before extracting, add internal module boundaries. Create clear interfaces between domains. This makes extraction far safer and reveals hidden dependencies.

3. **Extract High-Value Services**

   Start with the service with the most to gain: highest traffic, most independent team, or sharpest scaling needs. The strangler fig grows around the old tree.

4. **Add an API Gateway**

   Put an API Gateway in front that routes some traffic to the new service, the rest to the monolith. Users see no change. You control the cutover gradually.

5. **Migrate Data**

   Use the Database-per-Service pattern. Run dual writes during transition. Eventually, cut the monolith's access off entirely. This is the hardest step.

6. **Strangle & Repeat**

   Once the extracted service proves stable, deprecate that module in the monolith. Repeat for the next bounded context. The monolith shrinks over time.

WHO USES MICROSERVICES?

Netflix

700+

services

Runs 700+ microservices on AWS. Pioneered resilience patterns: Circuit Breaker, Chaos Engineering (Chaos Monkey). Handles 250M+ subscribers.

Amazon

~1000

services

Amazon.com runs thousands of microservices. Each product team owns their service entirely. Two-pizza rule: if 2 pizzas can't feed the team, it's too big.

Uber

2200+

services

Started as a monolith, migrated to 2200+ microservices. Separate services for: matching riders/drivers, payments, maps, pricing, notifications.

Spotify

800+

squads

Organized around 'Squads', 'Tribes', and 'Guilds'. Each squad owns their microservice end-to-end. Conway's Law in action — org structure IS the architecture.

These companies didn't start with microservices. They evolved to them — when the pain of the monolith exceeded the cost of migration.

KEY TAKEAWAYS

What You've Learned

- A monolith is not an anti-pattern. It's the right starting point for most products. Know when to evolve.
- Microservices = independently deployable services, each owning its own data and lifecycle.
- The API Gateway is the single front door — handling auth, routing, rate limiting, and protocol translation.
- DDD gives you the language and tools to draw service boundaries that match your business domains.
- Benefits: scaling, deployment speed, fault isolation, team autonomy, and technology freedom.
- Challenges: distributed complexity, observability, data consistency, and massive operational overhead.

The Golden Rule

"Don't distribute what doesn't need to be distributed. Complexity is a choice. Make it consciously."

Start Simple:

Monolith (validated idea)

→ Modular Monolith

→ Extract First Service

→ API Gateway

→ Full MSA