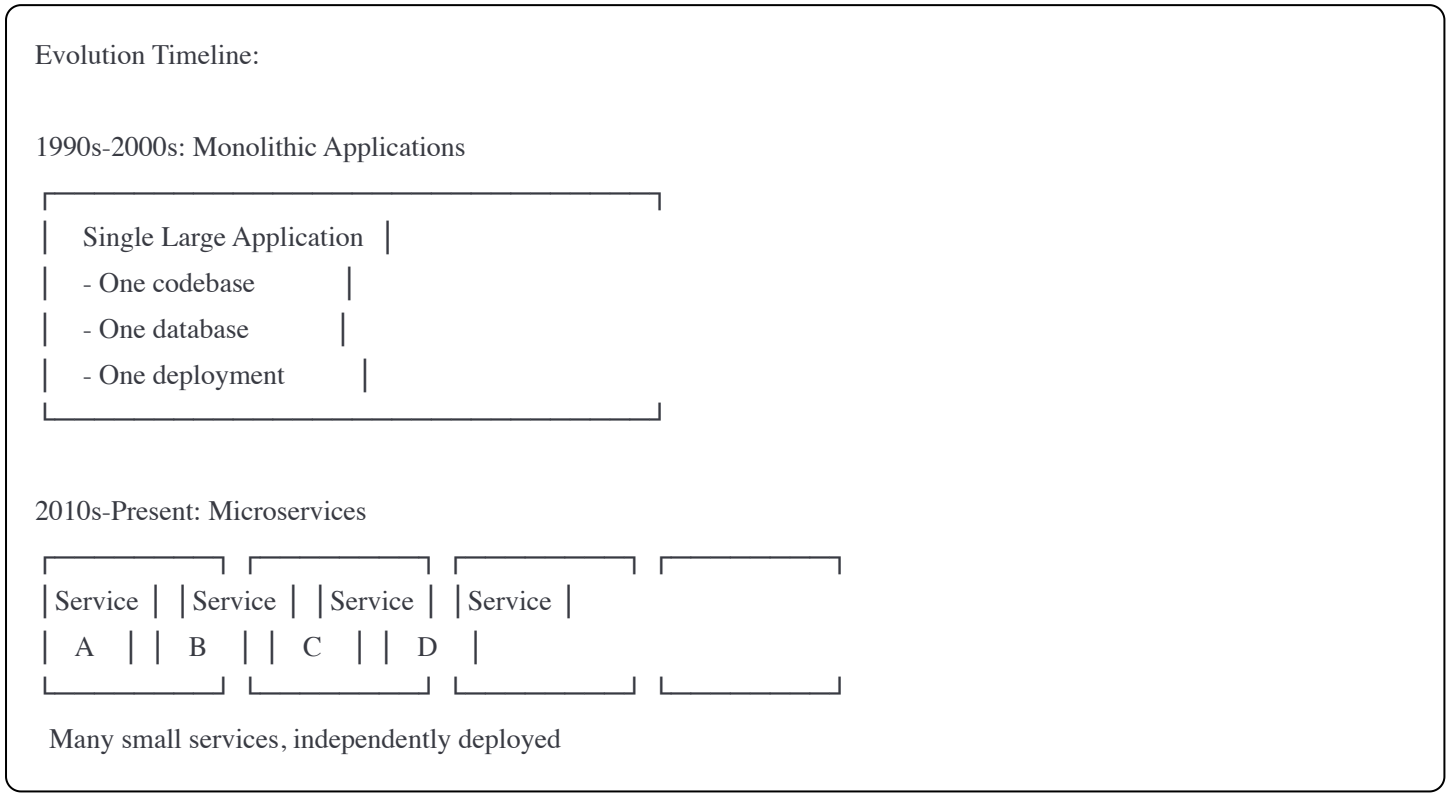
Chapter 15: Microservices Architecture

Introduction: From Monolith to Microservices

The evolution of application architecture.

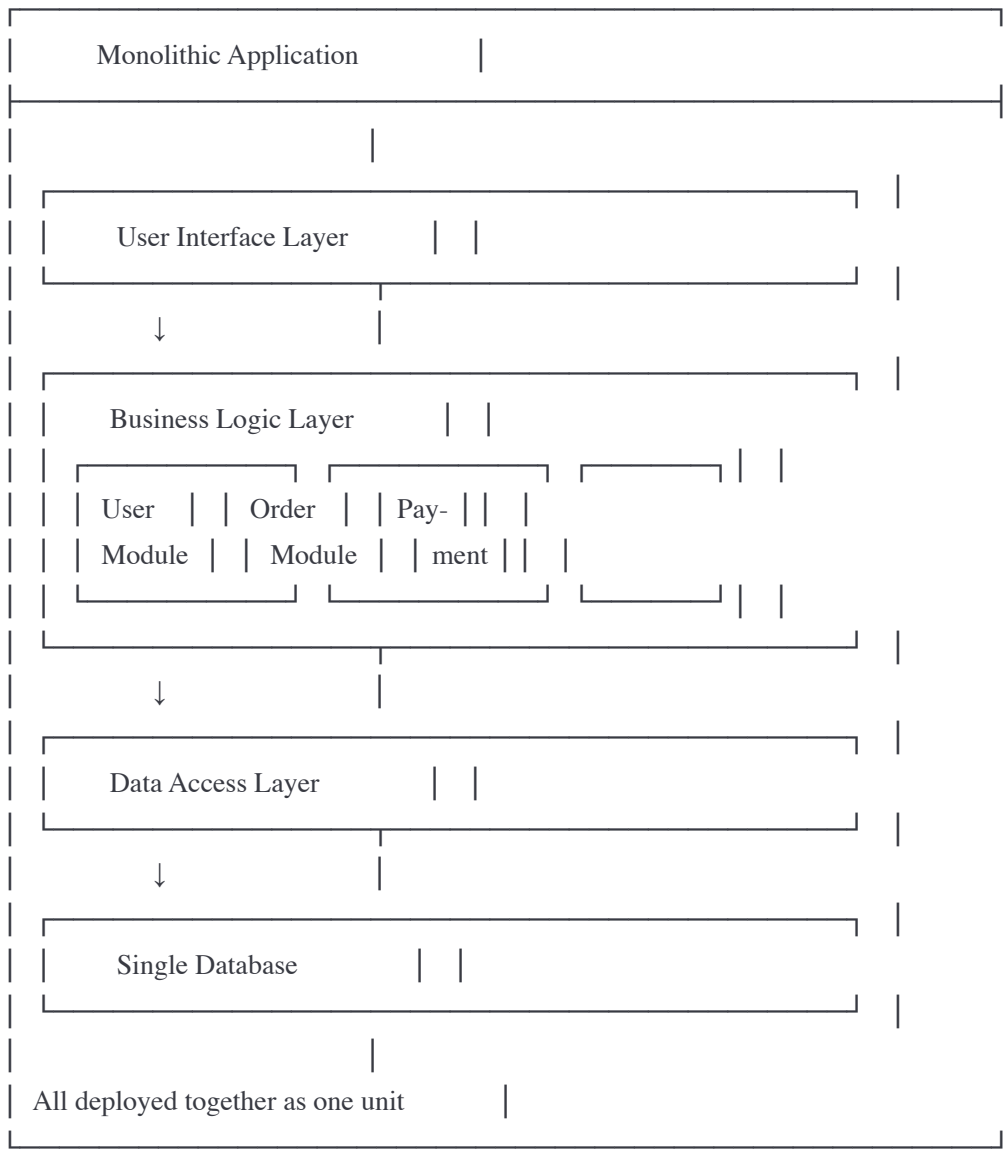


---

1. Monolith vs Microservices

Monolithic Architecture

Structure: All functionality in one application.



Characteristics:

- Single codebase
- Single deployment unit
- Shared database
- In-process communication

Monolith Code Example:

javascript

```
// Monolithic E-commerce Application
const express = require('express');
const app = express();

// All modules in same application
const userModule = require('./modules/user');
const orderModule = require('./modules/order');
const paymentModule = require('./modules/payment');
const inventoryModule = require('./modules/inventory');

// Shared database connection
const db = require('./database');

// User endpoints
app.post('/users', async (req, res) => {
  const user = await userModule.createUser(req.body);
  res.json(user);
});

// Order endpoints
app.post('/orders', async (req, res) => {
  try {
    // All in same process - direct function calls
    const user = await userModule.getUser(req.body.userId);
    const inventory = await inventoryModule.checkInventory(req.body.items);

    if (!inventory.available) {
      return res.status(400).json({ error: 'Out of stock' });
    }

    const order = await orderModule.createOrder(req.body);
    const payment = await paymentModule.processPayment(order.total);

    await inventoryModule.reduceStock(req.body.items);

    res.json(order);

  } catch (error) {
    res.status(500).json({ error: error.message });
  }
});

app.listen(3000);

// All code in same repository, deployed together
```

Monolith Pros:

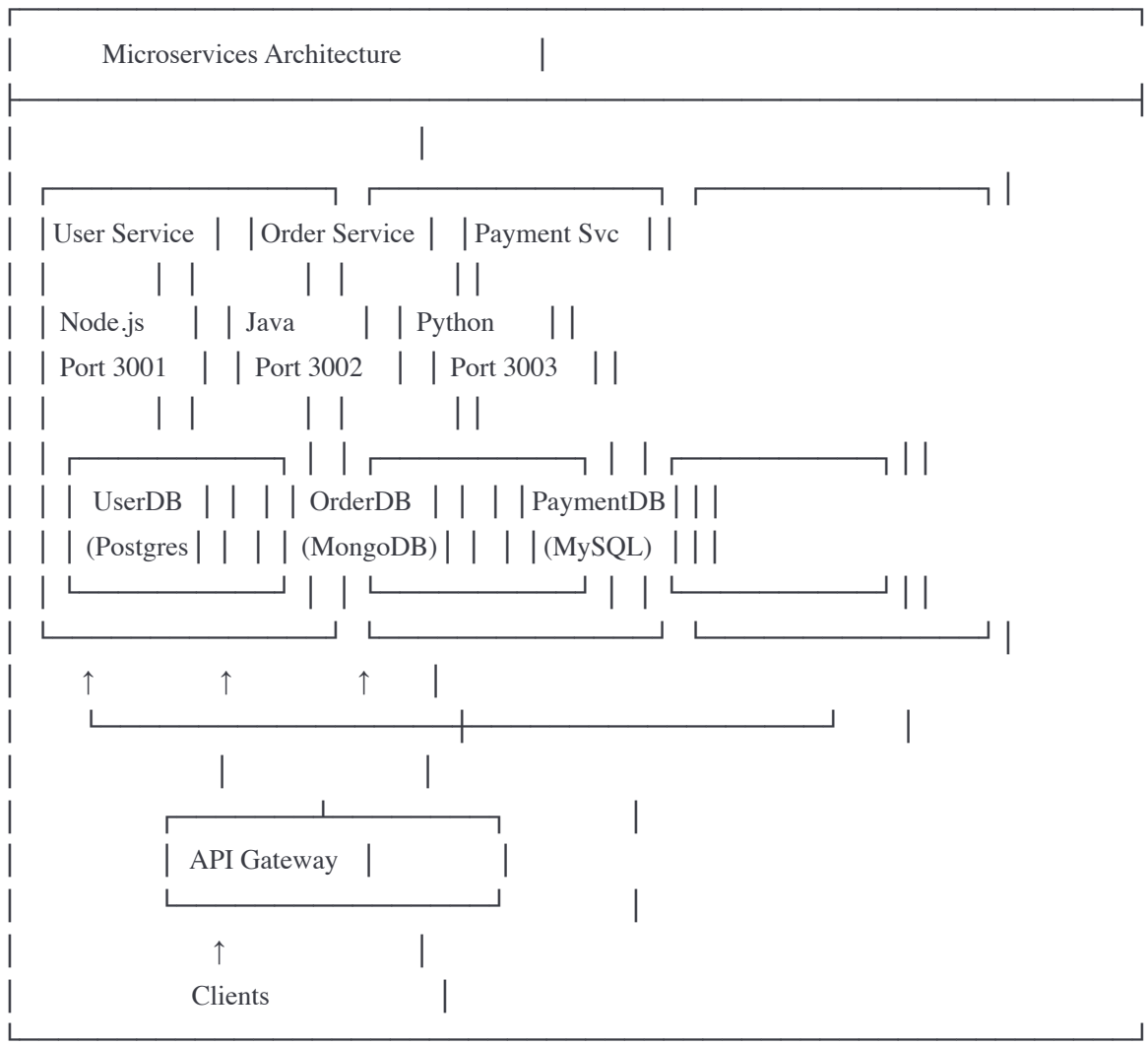
✓ Simple to develop (everything in one place)
✓ Easy to debug (single process)
✓ Fast inter-module communication (in-memory)
✓ Single deployment (simple)
✓ Strong consistency (single database, ACID transactions)
✓ Easy to test (integrated)

Monolith Cons:

✗ Tight coupling (changes affect entire app)
✗ Hard to scale (must scale entire app)
✗ Slow deployment (entire app must redeploy)
✗ Technology lock-in (one language, framework)
✗ Large codebase (hard to understand)
✗ Single point of failure (if it crashes, everything down)

Microservices Architecture

Structure: Many small, independent services.



Characteristics:

- Multiple codebases
- Independent deployments
- Separate databases (database per service)
- Network communication
- Different technologies possible

Microservices Code Example:

javascript

```
// USER SERVICE (Node.js on port 3001)
const express = require('express');
const app = express();

let users = [
  { id: 1, name: 'John Doe', email: 'john@example.com' }
];

app.get('/users/:id', (req, res) => {
  const user = users.find(u => u.id === parseInt(req.params.id));
  res.json(user || { error: 'Not found' });
});

app.post('/users', (req, res) => {
  const user = { id: users.length + 1, ...req.body };
  users.push(user);
  res.json(user);
});

app.listen(3001, () => console.log('User service on port 3001'));

// ORDER SERVICE (Java on port 3002)
// Separate codebase, different language!
@RestController
@RequestMapping("/orders")
public class OrderController {
    @Autowired
    private OrderRepository orderRepository;

    @PostMapping
    public Order createOrder(@RequestBody OrderRequest request) {
        // Call User Service via HTTP
        User user = restTemplate.getForObject(
            "http://user-service:3001/users/" + request.getUserId(),
            User.class
        );

        if (user == null) {
            throw new UserNotFoundException();
        }

        // Call Payment Service via HTTP
        Payment payment = restTemplate.postForObject(
            "http://payment-service:3003/charge",
            new PaymentRequest(request.getTotal()),
            Payment.class
```

```
        );

        // Create order
        Order order = new Order();
        order.setUserId(request.getUserId());
        order.setTotal(request.getTotal());
        order.setPaymentId(payment.getId());

        return orderRepository.save(order);
    }
}

// PAYMENT SERVICE (Python on port 3003)
# Separate codebase, different language again!
from flask import Flask, request, jsonify

app = Flask(__name__)

@app.route('/charge', methods=['POST'])
def charge():
    data = request.json
    amount = data['amount']

    # Process payment
    payment_id = process_payment(amount)

    return jsonify({
        'id': payment_id,
        'amount': amount,
        'status': 'completed'
    })

if __name__ == '__main__':
    app.run(port=3003)
```

Microservices Pros:

✓ Independent deployment (deploy one service)
✓ Technology diversity (best tool for each job)
✓ Scalability (scale only what needs scaling)
✓ Team autonomy (different teams own different services)
✓ Fault isolation (one service down ≠ entire system down)
✓ Easier to understand (smaller codebases)

Microservices Cons:

✗ Distributed complexity (network calls, latency)
✗ Data consistency (across services)
✗ Testing complexity (integration testing hard)
✗ Operational overhead (deploy/monitor many services)
✗ Debugging harder (trace across services)
✗ Network overhead (vs in-process calls)

Comparison Table

Aspect	Monolith	Microservices
Codebase	Single	Multiple
Deployment	All at once	Independent
Scaling	Entire app	Per service
Technology	One stack	Polyglot
Database	Shared	Per service
Team Structure	One team	Multiple teams
Development	Fast (start)	Slow (setup)
Testing	Easy	Complex
Debugging	Easy	Hard
Deployment	Simple	Complex
Monitoring	Simple	Complex
Consistency	Strong (ACID)	Eventual
Latency	Low (in-proc)	Higher (network)
Fault Isolation	None	Good
Team Autonomy	Low	High
Best For	Small teams	Large orgs
	Simple domains	Complex domains
	Startups	Mature products

Rule of Thumb:
- Start with monolith
- Split into microservices when:
  • Team > 10 people
  • Clear domain boundaries
  • Need independent scaling
  • Can handle complexity

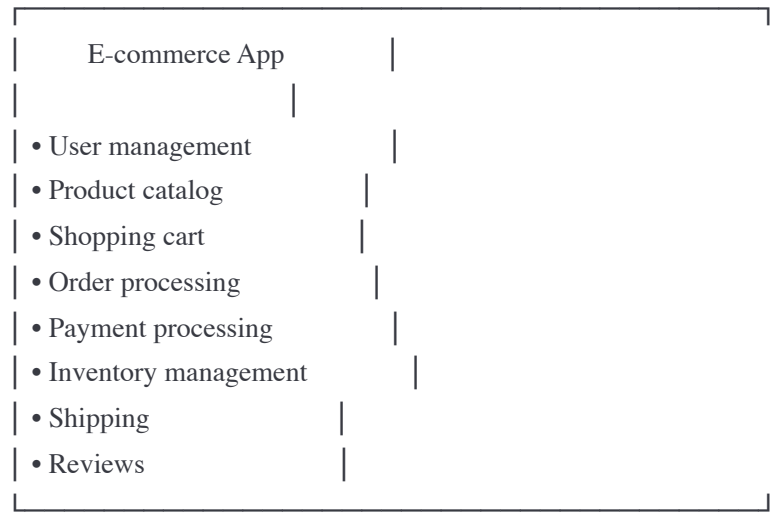
---

2. Service Decomposition Strategies

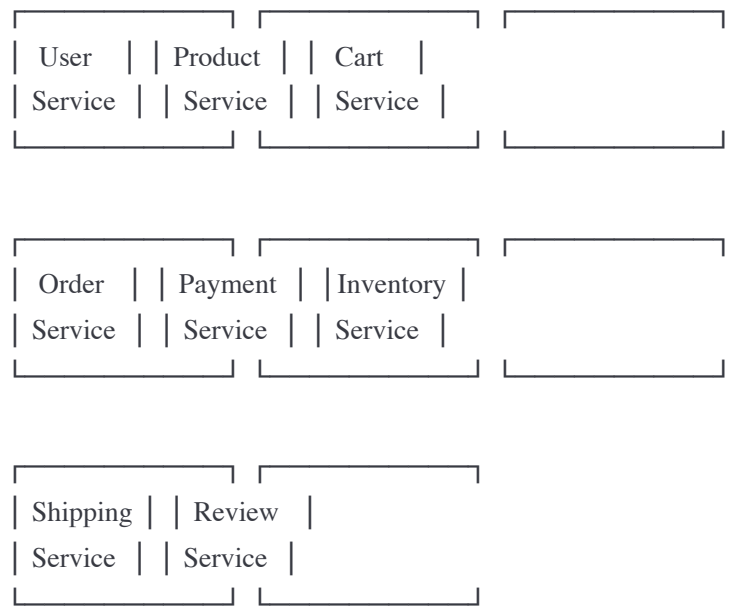
How to Break Down a Monolith

Strategy 1: Decompose by Business Capability

E-commerce Monolith:



Decompose by Business Capability:



Each service owns a business capability

Strategy 2: Decompose by Subdomain (Domain-Driven Design)

Using DDD (Domain-Driven Design):

Core Domains (Competitive advantage):

```
┌──────────────────────────────────────┐
│ • Recommendation Engine              │
│ • Pricing Algorithm                  │
│ • Fraud Detection                    │
└──────────────────────────────────────┘
```

→ Invest heavily, optimize, innovate

Supporting Domains (Necessary but standard):

```
┌──────────────────────────────────────┐
│ • Order Management                   │
│ • Inventory                          │
│ • Shipping                           │
└──────────────────────────────────────┘
```

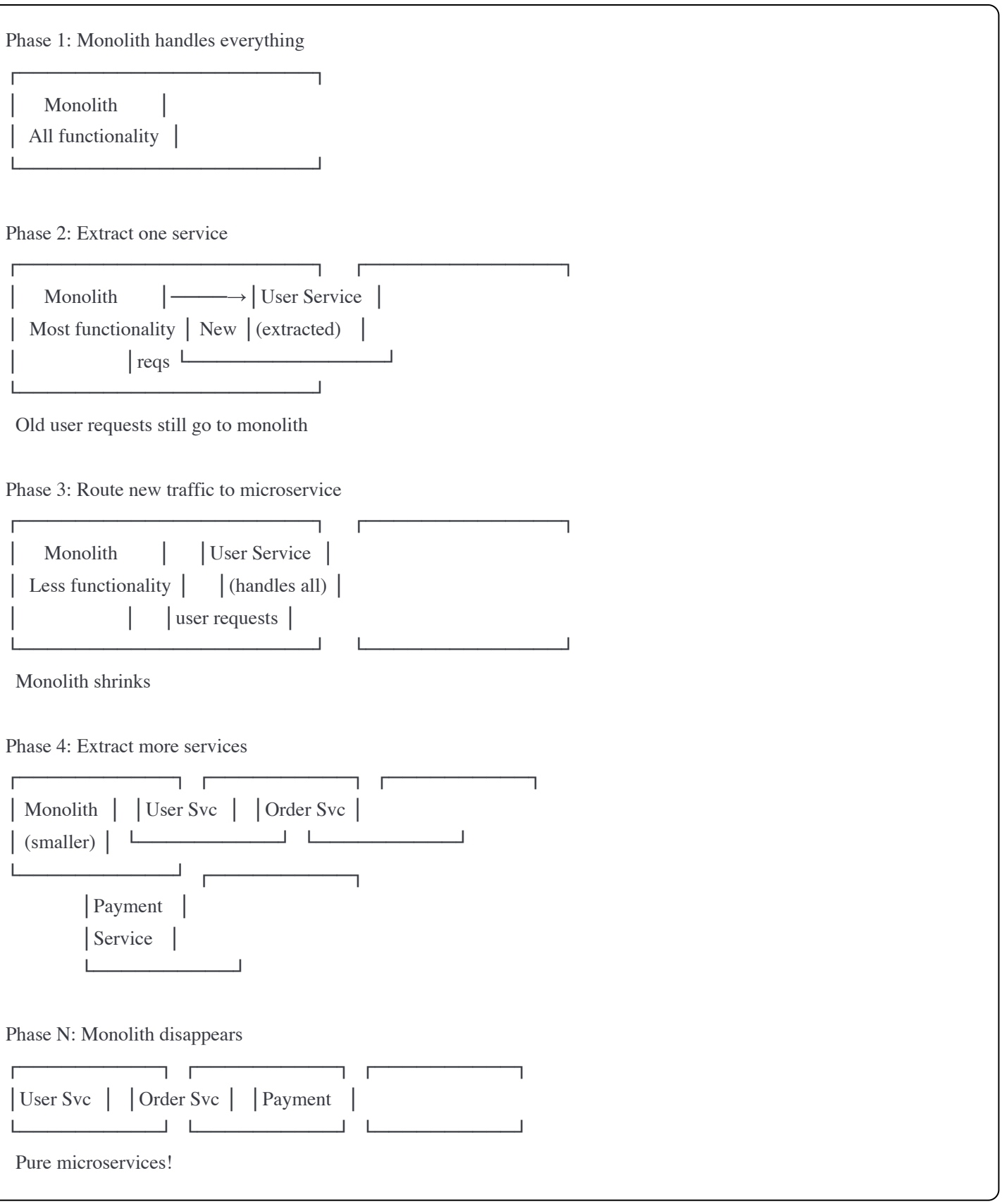
→ Build in-house, standard approach

Generic Domains (Buy, don't build):

```
┌──────────────────────────────────────┐
│ • Authentication (use Auth0)         │
│ • Email (use SendGrid)               │
│ • Payment (use Stripe)               │
└──────────────────────────────────────┘
```

→ Use third-party services

Strategy 3: Strangler Fig Pattern (Gradual Migration)



Implementation:

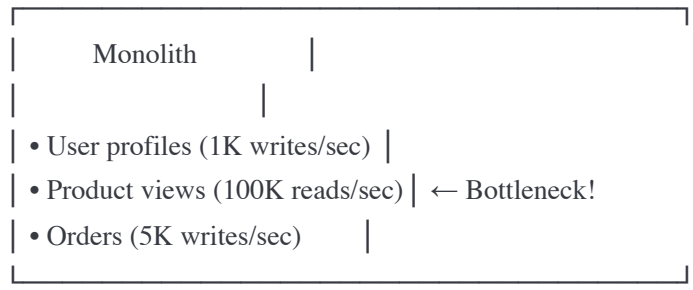
javascript

```
// API Gateway routes traffic
const express = require('express');
const proxy = require('express-http-proxy');

const app = express();

// Phase 1: Route user requests to new microservice
app.use('/api/users', proxy('http://user-service:3001'));

// Phase 2: Route orders to new microservice
app.use('/api/orders', proxy('http://order-service:3002'));

// Everything else still goes to monolith
app.use('/', proxy('http://monolith:8080'));

app.listen(80);

// Gradually migrate functionality service by service
// No big-bang rewrite!
```

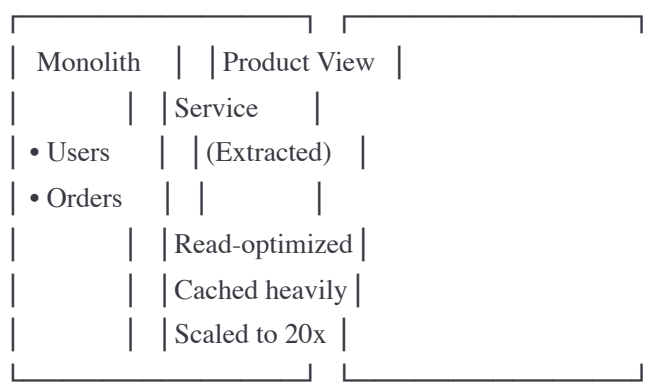
Strategy 4: By Data/Transaction Volume

Split high-volume parts first:

Before:



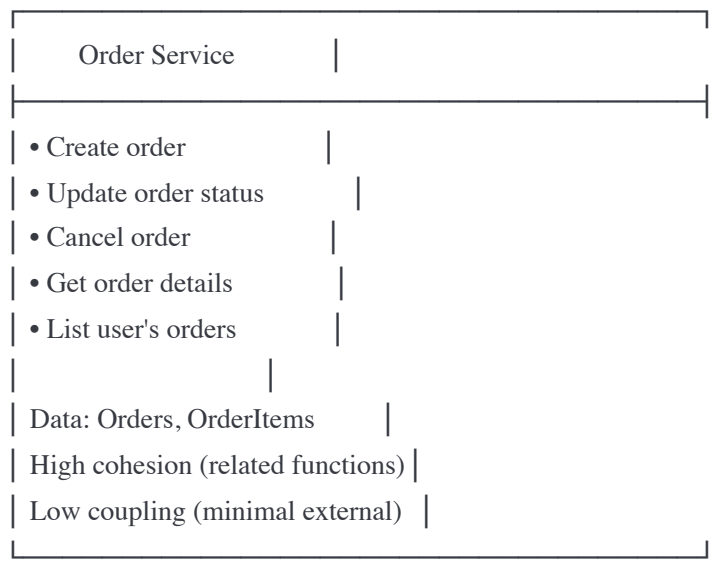
After:



Now can scale product views independently!

Determining Service Boundaries

Good Service Boundary:



Bad Service Boundary:

```
├──────────────────────────────────────────┤
│ • Get user data                          │
│ • Get product data                       │
│ • Get order data                         │
│ • Get payment data                       │
│                                          │
│ Too generic!                             │
│ No clear business capability             │
│ Becomes a database gateway               │
└──────────────────────────────────────────┘
```

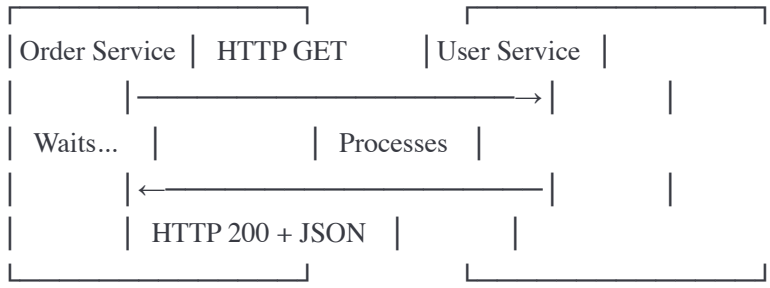
---

3. Inter-Service Communication

Synchronous Communication (Request/Response)

REST over HTTP:

Order Service needs User Service:



Pros:
✓ Simple to understand
✓ Immediate response
✓ Easy to debug

Cons:
✗ Tight coupling (order waits for user)
✗ Cascading failures (if user down, order fails)
✗ Higher latency (network calls)

gRPC (Binary, faster):

protobuf

```
// user.proto
service UserService {
  rpc GetUser(GetUserRequest) returns (GetUserResponse);
}

message GetUserRequest {
  int32 user_id = 1;
}

message GetUserResponse {
  int32 id = 1;
  string name = 2;
  string email = 3;
}
```

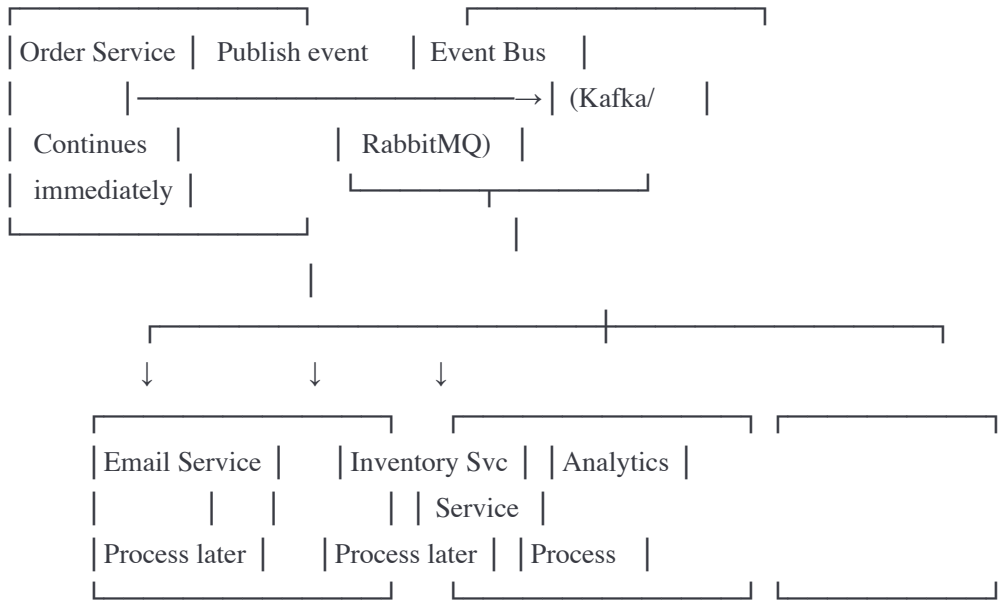
javascript

```
// Order service calls User service via gRPC
const grpc = require('@grpc/grpc-js');
const protoLoader = require('@grpc/proto-loader');

const packageDefinition = protoLoader.loadSync('user.proto');
const userProto = grpc.loadPackageDefinition(packageDefinition);

const userClient = new userProto.UserService(
  'user-service:50051',
  grpc.credentials.createInsecure()
);

// Call user service
function getUser(userId) {
  return new Promise((resolve, reject) => {
    userClient.GetUser({ user_id: userId }, (error, response) => {
      if (error) {
        reject(error);
      } else {
        resolve(response);
      }
    });
  });
}

// In order processing
const user = await getUser(orderId);
```

Asynchronous Communication (Events)

Message Queue/Event Bus:

Order Service publishes event (doesn't wait):



Pros:

✓ Loose coupling (services independent)

✓ Better fault tolerance (if email down, order still succeeds)

✓ Higher throughput (async processing)

✓ Event replay possible

Cons:

✗ Eventual consistency

✗ Complex debugging (events flow asynchronously)

✗ Message ordering challenges

Implementation:

javascript

```
// Order Service publishes events
const { Kafka } = require('kafkajs');

const kafka = new Kafka({
  brokers: ['kafka:9092']
});

const producer = kafka.producer();

class OrderService {
  async createOrder(orderData) {
    // Create order in database
    const order = await db.createOrder(orderData);

    // Publish event (async, doesn't wait)
    await producer.send({
      topic: 'order-events',
      messages: [{
        key: order.id.toString(),
        value: JSON.stringify({
          type: 'ORDER_CREATED',
          orderId: order.id,
          userId: orderData.userId,
          items: orderData.items,
          total: order.total,
          timestamp: new Date().toISOString()
        })
      }]
    });

    // Return immediately
    return order;
  }
}

// Email Service subscribes to events
const consumer = kafka.consumer({ groupId: 'email-service' });

await consumer.subscribe({ topic: 'order-events' });

await consumer.run({
  eachMessage: async ({ message }) => {
    const event = JSON.parse(message.value.toString());

    if (event.type === 'ORDER_CREATED') {
      // Send confirmation email
```

```
    await sendOrderConfirmation(event.userId, event.orderId);
    console.log(`Sent confirmation for order ${event.orderId}`);
   }
  }
});

// Inventory Service subscribes to same events
const inventoryConsumer = kafka.consumer({ groupId: 'inventory-service' });

await inventoryConsumer.subscribe({ topic: 'order-events' });

await inventoryConsumer.run({
  eachMessage: async ({ message }) => {
   const event = JSON.parse(message.value.toString());

   if (event.type === 'ORDER_CREATED') {
    // Reserve inventory
    await reserveItems(event.items);
    console.log(`Reserved inventory for order ${event.orderId}`);
   }
  }
});
```

Service Communication Patterns

```
┌──────────────────────┬──────────────┬──────────────────┐
│ Pattern              │ When         │ Trade-offs       │
├──────────────────────┼──────────────┼──────────────────┤
│ Synchronous HTTP     │ Need response│ Tight coupling   │
│ (REST/gRPC)          │ immediately  │ Cascading fails  │
├──────────────────────┼──────────────┼──────────────────┤
│ Async Messaging      │ Fire & forget│ Eventual         │
│ (Kafka/RabbitMQ)     │ Events       │ consistency      │
├──────────────────────┼──────────────┼──────────────────┤
│ Request/Reply        │ Async but    │ Complex routing  │
│ (RabbitMQ)           │ need response│                  │
└──────────────────────┴──────────────┴──────────────────┘
```

Recommendation:
- Sync (gRPC): When must have response (user login, checkout)
- Async (events): When can process later (email, analytics)
- Hybrid: Use both in same system

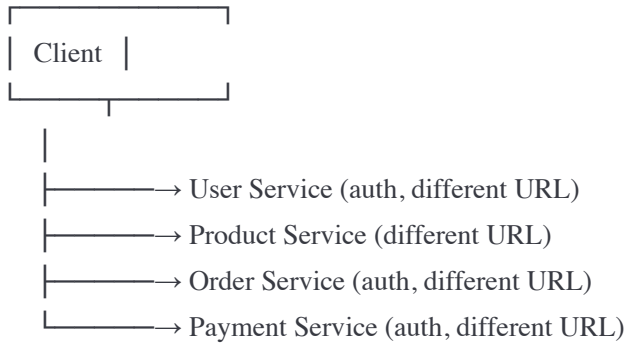
---

4. API Gateway Pattern

What is an API Gateway?

Single entry point for all client requests.

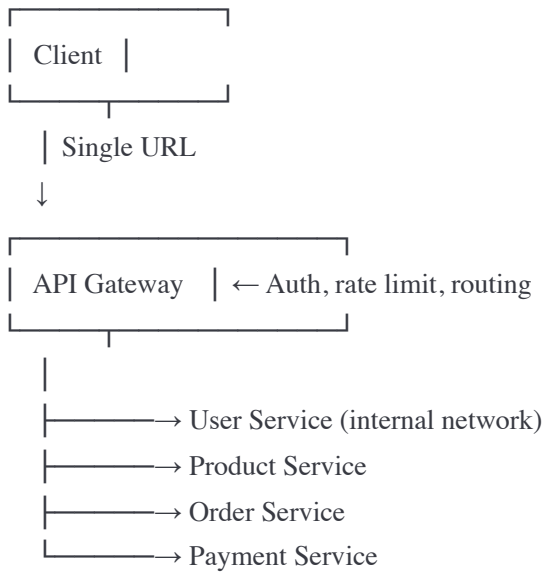
Without API Gateway (Clients call services directly):



Problems:

- Client needs to know all service URLs
- Each service must handle auth
- CORS on every service
- No central rate limiting

With API Gateway:



Benefits:

- Single entry point
- Centralized auth
- Rate limiting
- Request/response transformation
- Protocol translation (HTTP → gRPC)

API Gateway Responsibilities

1. ROUTING
   /api/users/* → User Service
   /api/products/* → Product Service
   /api/orders/* → Order Service

2. AUTHENTICATION
   Verify JWT token
   Extract user info
   Pass to downstream services

3. RATE LIMITING
   Limit requests per user/IP
   Prevent abuse

4. REQUEST/RESPONSE TRANSFORMATION
   Aggregate multiple service calls
   Transform formats

5. PROTOCOL TRANSLATION
   HTTP → gRPC
   REST → GraphQL

6. CACHING
   Cache frequent requests
   Reduce backend load

7. LOAD BALANCING
   Distribute across service instances

8. CIRCUIT BREAKING
   Protect services from cascading failures

9. LOGGING & MONITORING
   Central point for observability

10. SSL TERMINATION
    Handle HTTPS
    Backends use HTTP

API Gateway Implementation

javascript

```
const express = require('express');
const httpProxy = require('http-proxy');
const jwt = require('jsonwebtoken');
const rateLimit = require('express-rate-limit');

const app = express();
const proxy = httpProxy.createProxyServer();

// Service registry
const services = {
  users: ['http://user-service-1:3001', 'http://user-service-2:3001'],
  products: ['http://product-service-1:3002'],
  orders: ['http://order-service-1:3003', 'http://order-service-2:3003'],
  payments: ['http://payment-service-1:3004']
};

// Load balancer (round-robin)
const serviceIndex = {};

function getServiceUrl(serviceName) {
  const instances = services[serviceName];

  if (!serviceIndex[serviceName]) {
    serviceIndex[serviceName] = 0;
  }

  const url = instances[serviceIndex[serviceName]];
  serviceIndex[serviceName] = (serviceIndex[serviceName] + 1) % instances.length;

  return url;
}

// Authentication middleware
function authenticate(req, res, next) {
  const token = req.headers.authorization?.replace('Bearer ', '');

  if (!token) {
    return res.status(401).json({ error: 'No token provided' });
  }

  try {
    const decoded = jwt.verify(token, process.env.JWT_SECRET);
    req.user = decoded;
    next();
  } catch (error) {
    res.status(401).json({ error: 'Invalid token' });
```

```
  }
}

// Rate limiting
const limiter = rateLimit({
  windowMs: 15 * 60 * 1000, // 15 minutes
  max: 100, // 100 requests per window
  message: 'Too many requests'
});

app.use(limiter);

// Route to User Service
app.use('/api/users', authenticate, (req, res) => {
  const target = getServiceUrl('users');

  // Add user info to headers
  req.headers['X-User-ID'] = req.user.id;
  req.headers['X-User-Role'] = req.user.role;

  proxy.web(req, res, { target });
});

// Route to Product Service
app.use('/api/products', (req, res) => {
  // Public endpoint, no auth required
  const target = getServiceUrl('products');
  proxy.web(req, res, { target });
});

// Route to Order Service
app.use('/api/orders', authenticate, (req, res) => {
  const target = getServiceUrl('orders');
  req.headers['X-User-ID'] = req.user.id;
  proxy.web(req, res, { target });
});

// Aggregation endpoint (calls multiple services)
app.get('/api/dashboard', authenticate, async (req, res) => {
  try {
    // Call multiple services in parallel
    const [user, orders, recommendations] = await Promise.all([
      fetch(`${getServiceUrl('users')}/users/${req.user.id}`),
      fetch(`${getServiceUrl('orders')}/orders?userId=${req.user.id}`),
      fetch(`${getServiceUrl('products')}/recommendations?userId=${req.user.id}`)
    ]);
```

```
    // Aggregate responses
    res.json({
      user: await user.json(),
      orders: await orders.json(),
      recommendations: await recommendations.json()
    });

  } catch (error) {
    res.status(500).json({ error: 'Failed to load dashboard' });
  }
});

// Error handling
proxy.on('error', (err, req, res) => {
  console.error('Proxy error:', err);
  res.status(500).json({ error: 'Service unavailable' });
});

app.listen(80, () => {
  console.log('API Gateway running on port 80');
});
```

API Gateway with Circuit Breaker

javascript

```
const CircuitBreaker = require('./circuit-breaker');

// Circuit breakers for each service
const breakers = {
  users: new CircuitBreaker({ failureThreshold: 5, timeout: 30000 }),
  products: new CircuitBreaker({ failureThreshold: 5, timeout: 30000 }),
  orders: new CircuitBreaker({ failureThreshold: 5, timeout: 30000 })
};

// Route with circuit breaker protection
app.use('/api/users', authenticate, async (req, res) => {
  try {
    await breakers.users.execute(async () => {
      const target = getServiceUrl('users');
      proxy.web(req, res, { target });
    });

  } catch (error) {
    if (error.message.includes('Circuit breaker is OPEN')) {
      // Service is down, return cached data or error
      res.status(503).json({
        error: 'User service temporarily unavailable',
        cached: true,
        data: await cache.get(`user:${req.params.id}`)
      });
    } else {
      throw error;
    }
  }
});
```

---

5. Service Mesh (Istio, Linkerd)

What is a Service Mesh?

Problem: Each service needs:

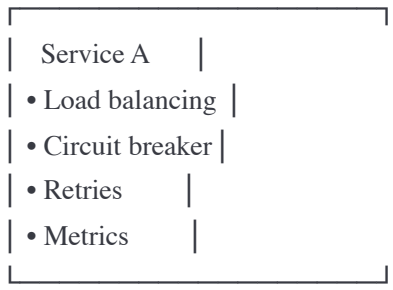
- Load balancing
- Circuit breaking
- Retries
- Timeouts
- Metrics

- Tracing
- mTLS encryption

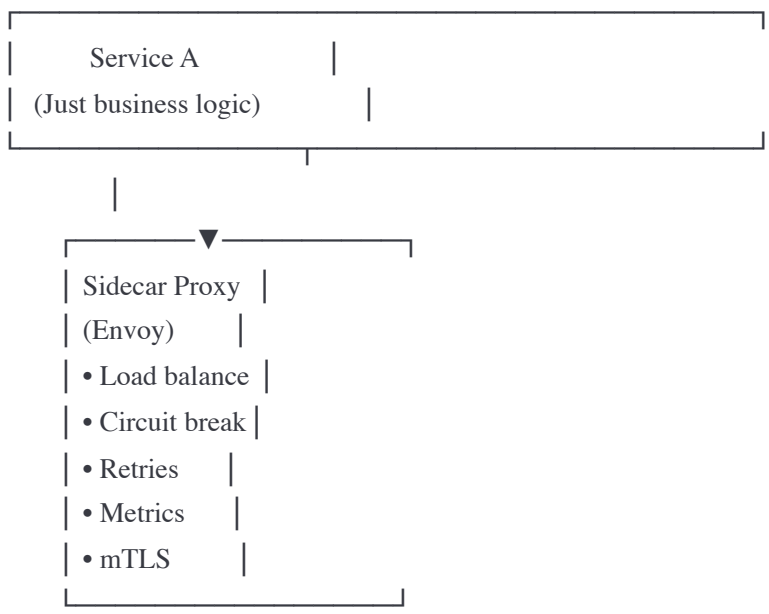
Solution: Service mesh handles this infrastructure layer.

Without Service Mesh:

Each service implements own:

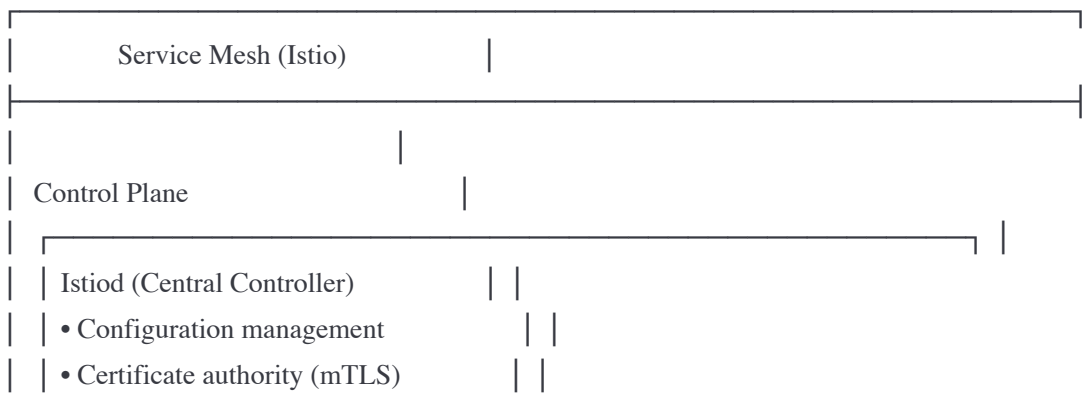


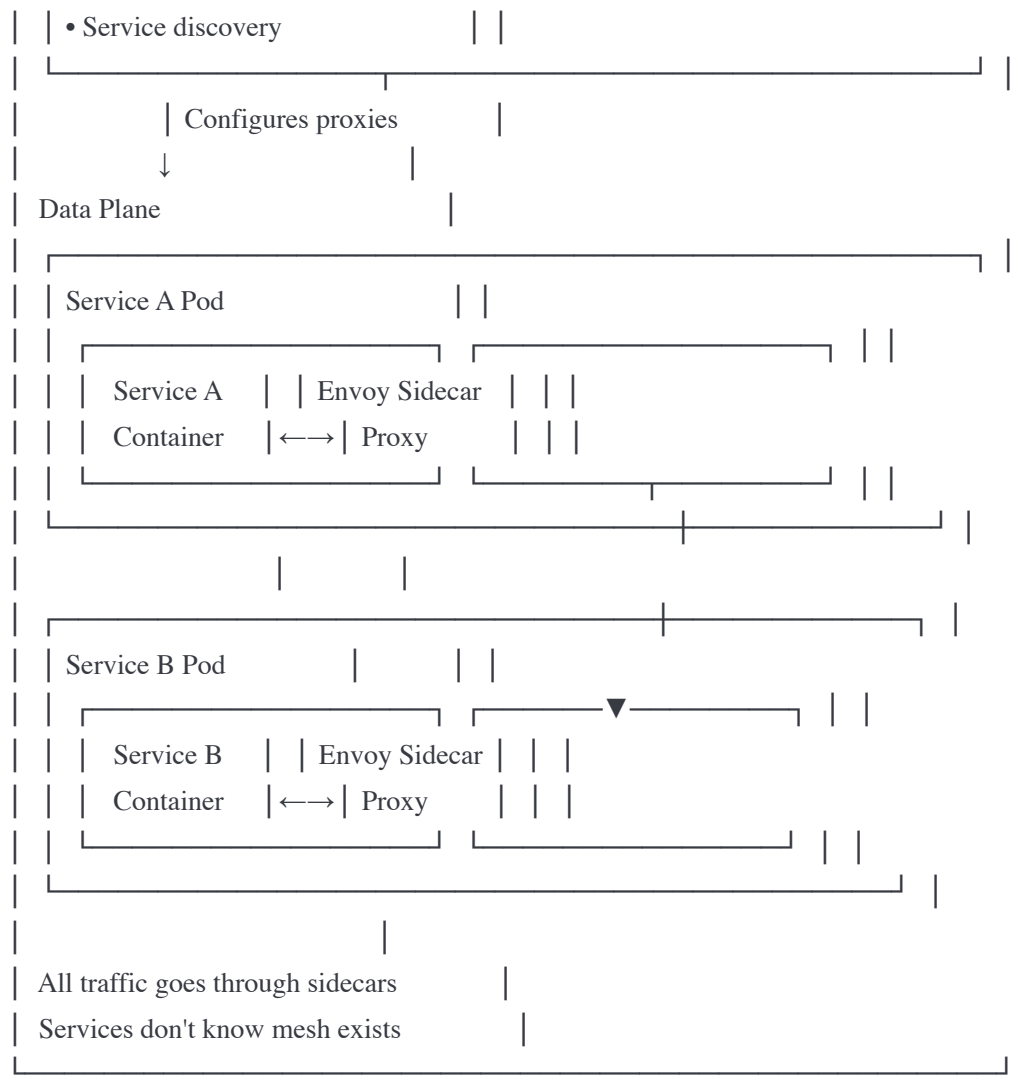
With Service Mesh:



All services get these features automatically!

Service Mesh Architecture





Istio Configuration Examples

Virtual Service (Traffic Routing):

yaml

```yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: order-service
spec:
  hosts:
  - order-service
  http:
  # Canary deployment (90% to v1, 10% to v2)
  - match:
    - headers:
        x-user-group:
          exact: beta
    route:
    - destination:
        host: order-service
        subset: v2
  - route:
    - destination:
        host: order-service
        subset: v1
      weight: 90
    - destination:
        host: order-service
        subset: v2
      weight: 10
```

Destination Rule (Load Balancing, Circuit Breaking):

```yaml
```

```yaml
apiVersion: networking.istio.io/v1beta1
kind: DestinationRule
metadata:
  name: order-service
spec:
  host: order-service
  trafficPolicy:
    # Load balancing
    loadBalancer:
      simple: LEAST_REQUEST

    # Connection pool
    connectionPool:
      tcp:
        maxConnections: 100
      http:
        http1MaxPendingRequests: 50
        http2MaxRequests: 100

    # Circuit breaker
    outlierDetection:
      consecutiveErrors: 5
      interval: 30s
      baseEjectionTime: 30s
      maxEjectionPercent: 50

  # Subsets for versioning
  subsets:
  - name: v1
    labels:
      version: v1
  - name: v2
    labels:
      version: v2
```

Retry Configuration:

```yaml
```

```yaml
apiVersion: networking.istio.io/v1beta1
kind: VirtualService
metadata:
  name: payment-service
spec:
  hosts:
  - payment-service
  http:
  - route:
    - destination:
        host: payment-service
    timeout: 10s
    retries:
      attempts: 3
      perTryTimeout: 3s
      retryOn: 5xx,reset,connect-failure,refused-stream
```

mTLS (Mutual TLS):

```yaml
apiVersion: security.istio.io/v1beta1
kind: PeerAuthentication
metadata:
  name: default
spec:
  mtls:
    mode: STRICT  # Require mTLS for all services

# All service-to-service communication now encrypted!
# Certificates automatically managed by Istio
```

Service Mesh Benefits

```
✓ AUTOMATIC FEATURES
  Every service gets:
  • Load balancing
  • Circuit breaking
  • Retries
  • Timeouts
  • Metrics
  • Tracing
  • mTLS encryption
```

Without writing code!

✓ CENTRALIZED CONFIGURATION
Configure once, applies to all services

✓ OBSERVABILITY
Automatic metrics for all service communication
Distributed tracing built-in

✓ SECURITY
mTLS between all services
Fine-grained access control

✓ TRAFFIC MANAGEMENT
Canary deployments
A/B testing
Traffic splitting

✗ COMPLEXITY
Additional infrastructure
Learning curve
Resource overhead (sidecar per pod)

✗ PERFORMANCE
Small latency overhead (~1ms per hop)

✗ DEBUGGING
Another layer to understand

---

6. Microservices Challenges and Trade-offs

Challenge 1: Distributed Transactions

Problem: Transaction spans multiple services

Order Creation Flow:
1. Reserve inventory (Inventory Service)
2. Charge payment (Payment Service)
3. Create order (Order Service)

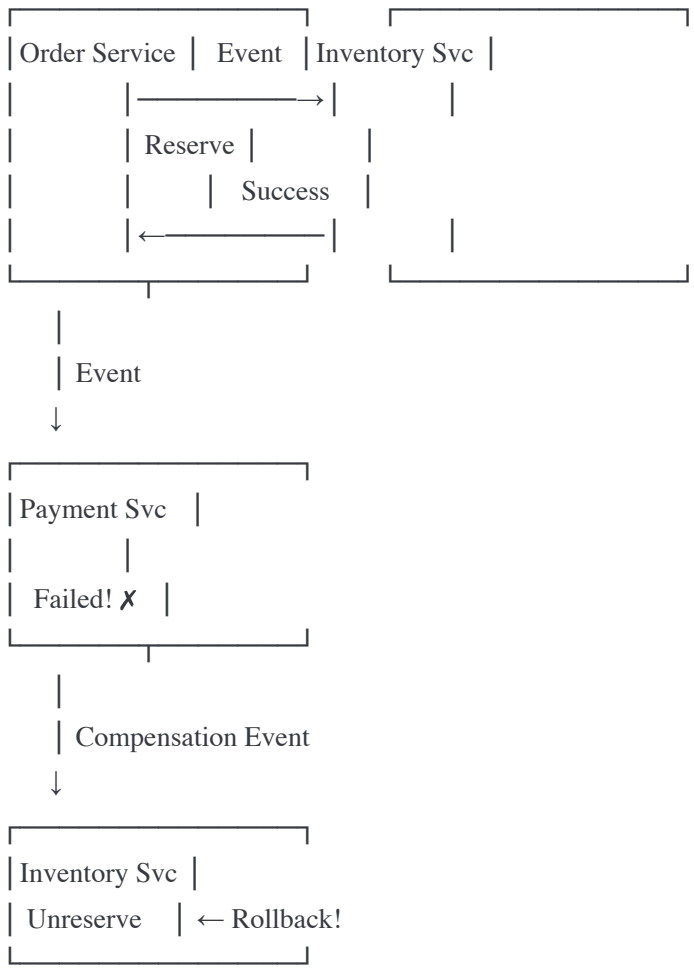
What if step 2 fails?
Must rollback step 1!

In monolith:
BEGIN TRANSACTION;
  -- All steps here
COMMIT; -- All or nothing!

In microservices:
No distributed transaction!

Solutions:

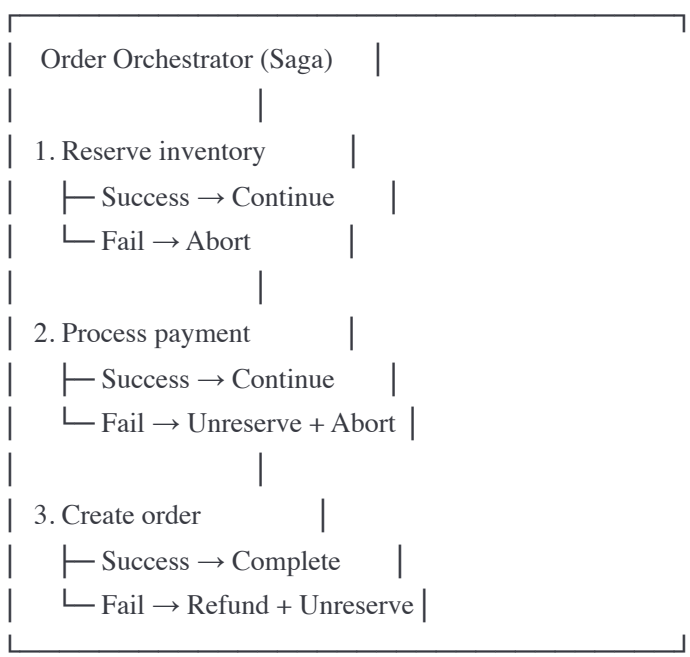
Solution 1: Saga Pattern (Choreography)



Each service listens and reacts to events

Compensating transactions for rollback

Solution 2: Saga Pattern (Orchestration)



Central coordinator manages workflow

Knows compensation logic

Saga Implementation (Orchestration):

javascript

```
class OrderSaga {
  constructor() {
    this.inventoryService = new InventoryClient();
    this.paymentService = new PaymentClient();
    this.orderService = new OrderClient();
  }

  async execute(orderData) {
    const context = {
      orderId: generateId(),
      reservationId: null,
      paymentId: null,
      completed: false
    };

    try {
      // Step 1: Reserve inventory
      console.log('[Saga] Step 1: Reserving inventory...');
      context.reservationId = await this.inventoryService.reserve(orderData.items);
      console.log('[Saga] Inventory reserved:', context.reservationId);

      // Step 2: Process payment
      console.log('[Saga] Step 2: Processing payment...');
      context.paymentId = await this.paymentService.charge({
        amount: orderData.total,
        userId: orderData.userId
      });
      console.log('[Saga] Payment processed:', context.paymentId);

      // Step 3: Create order
      console.log('[Saga] Step 3: Creating order...');
      const order = await this.orderService.create({
        ...orderData,
        reservationId: context.reservationId,
        paymentId: context.paymentId
      });
      console.log('[Saga] Order created:', order.id);

      context.completed = true;
      return { success: true, order };

    } catch (error) {
      console.error('[Saga] Failed:', error.message);

      // Compensate (rollback)
      await this.compensate(context, error);
```

```
      return { success: false, error: error.message };
    }
  }

  async compensate(context, error) {
    console.log('[Saga] Starting compensation...');

    // Rollback in reverse order

    // Step 3: Compensate order (if created)
    if (context.orderId) {
      console.log('[Saga] Compensate: Cancelling order...');
      try {
        await this.orderService.cancel(context.orderId);
      } catch (err) {
        console.error('Failed to cancel order:', err);
      }
    }

    // Step 2: Compensate payment (if charged)
    if (context.paymentId) {
      console.log('[Saga] Compensate: Refunding payment...');
      try {
        await this.paymentService.refund(context.paymentId);
      } catch (err) {
        console.error('Failed to refund payment:', err);
      }
    }

    // Step 1: Compensate inventory (if reserved)
    if (context.reservationId) {
      console.log('[Saga] Compensate: Unreserving inventory...');
      try {
        await this.inventoryService.unreserve(context.reservationId);
      } catch (err) {
        console.error('Failed to unreserve inventory:', err);
      }
    }

    console.log('[Saga] Compensation completed');
  }
}

// Usage
const saga = new OrderSaga();
```

```
const result = await saga.execute({
  userId: 123,
  items: [
    { productId: 1, quantity: 2 }
  ],
  total: 99.99
});

if (result.success) {
  console.log('Order created successfully');
} else {
  console.log('Order creation failed, all changes rolled back');
}
```

Challenge 2: Data Consistency

Problem: Each service has own database

```
Scenario: Update user email

In Monolith:
BEGIN TRANSACTION;
  UPDATE users SET email = 'new@example.com' WHERE id = 123;
  UPDATE orders SET customer_email = 'new@example.com' WHERE user_id = 123;
  UPDATE notifications SET email = 'new@example.com' WHERE user_id = 123;
COMMIT;

Guaranteed consistent!

In Microservices:
User Service: updates users table ✓
Order Service: has old email in orders ✗
Notification Service: has old email ✗

Eventual consistency!
```

Solutions:

Solution 1: Event-Driven Synchronization

User Service publishes event:

```
{
  type: 'USER_EMAIL_CHANGED',
  userId: 123,
  oldEmail: 'old@example.com',
  newEmail: 'new@example.com'
}
```

Order Service subscribes:
→ Updates customer_email in orders

Notification Service subscribes:
→ Updates email in notifications

Eventually all consistent!

Solution 2: Read from Source

Don't duplicate data!
When need user email, call User Service

Trade-off:
- More network calls
- But always consistent
- Slower but correct

Solution 3: Materialized View

User Service publishes changes
Other services build their own views

Each service has optimized copy
Updated via events

Challenge 3: Service Discovery

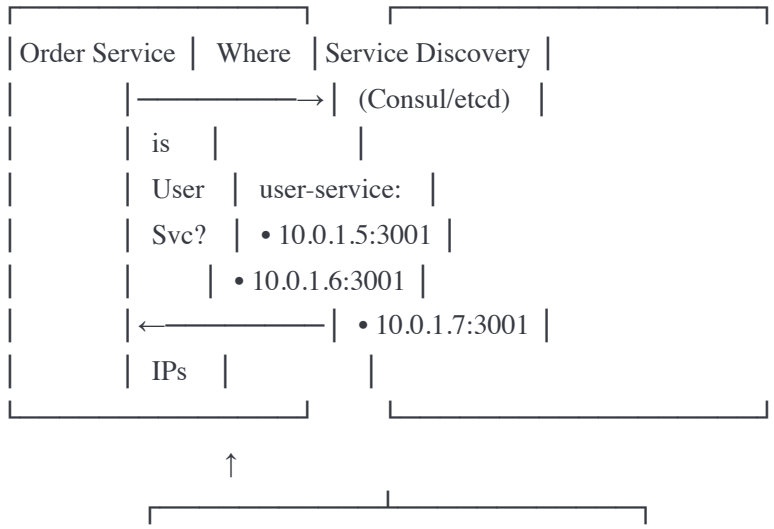
Problem: Service locations change dynamically

Order Service needs to call User Service
Where is User Service?

Static Configuration (bad):
userServiceUrl = 'http://192.168.1.10:3001'

Problems:
- IP changes when server restarts
- Can't scale (only one instance)
- Failover difficult

Service Discovery (good):



Implementation with Consul:

javascript

```
const Consul = require('consul');

class ServiceRegistry {
  constructor() {
    this.consul = new Consul({
      host: 'consul-server',
      port: 8500
    });
  }

  // Register service
  async register(serviceName, serviceId, port) {
    await this.consul.agent.service.register({
      name: serviceName,
      id: serviceId,
      address: getLocalIP(),
      port: port,
      check: {
        http: `http://localhost:${port}/health`,
        interval: '10s',
        timeout: '5s'
      }
    });

    console.log(`Registered ${serviceName} with Consul`);
  }

  // Discover service instances
  async discover(serviceName) {
    const result = await this.consul.health.service({
      service: serviceName,
      passing: true  // Only healthy instances
    });

    return result.map(entry => ({
      address: entry.Service.Address,
      port: entry.Service.Port
    }));
  }

  // Get service URL with load balancing
  async getServiceUrl(serviceName) {
    const instances = await this.discover(serviceName);

    if (instances.length === 0) {
      throw new Error(`No healthy instances of ${serviceName}`);
```

```
    }

    // Round-robin load balancing
    const instance = instances[Math.floor(Math.random() * instances.length)];
    return `http://${instance.address}:${instance.port}`;
  }

  // Deregister on shutdown
  async deregister(serviceId) {
    await this.consul.agent.service.deregister(serviceId);
    console.log(`Deregistered ${serviceId}`);
  }
}

// Usage in User Service
const registry = new ServiceRegistry();
const serviceId = 'user-service-' + generateId();

// Register on startup
await registry.register('user-service', serviceId, 3001);

// Graceful shutdown
process.on('SIGTERM', async () => {
  await registry.deregister(serviceId);
  process.exit(0);
});

// Usage in Order Service (client)
const userServiceUrl = await registry.getServiceUrl('user-service');
const user = await fetch(`${userServiceUrl}/users/123`);
```

Challenge 4: Distributed Logging and Tracing

Problem: Request spans multiple services

User request goes through:
API Gateway → User Service → Order Service → Payment Service

Logs scattered:
API Gateway logs: "Request received"
User Service logs: "User fetched"
Order Service logs: "Order created"
Payment Service logs: "Payment processed"

How do you correlate?

Solution: Correlation IDs + Distributed Tracing
(Covered in Chapter 14)

With Trace ID: trace-abc-123
All services log with same trace ID
Can reconstruct entire request flow

Challenge 5: Testing

Monolith Testing:

```
┌──────────────────────────┐
│ Integration              │
│ Test                     │
│ • Start app              │
│ • Test flows             │
│ Simple!                  │
└──────────────────────────┘
```

Microservices Testing:

```
┌──────────────────────────────────────────────────┐
│ Integration Test                                 │
│ • Start User Service                             │
│ • Start Order Service                            │
│ • Start Payment Service                          │
│ • Start Inventory Service                        │
│ • Start Message Queue                            │
│ • Start Service Discovery                        │
│ • Test flows                                     │
│ Complex!                                         │
└──────────────────────────────────────────────────┘
```

Solutions:
1. Contract Testing (Pact)

- Services agree on API contracts
- Test against contracts, not actual services

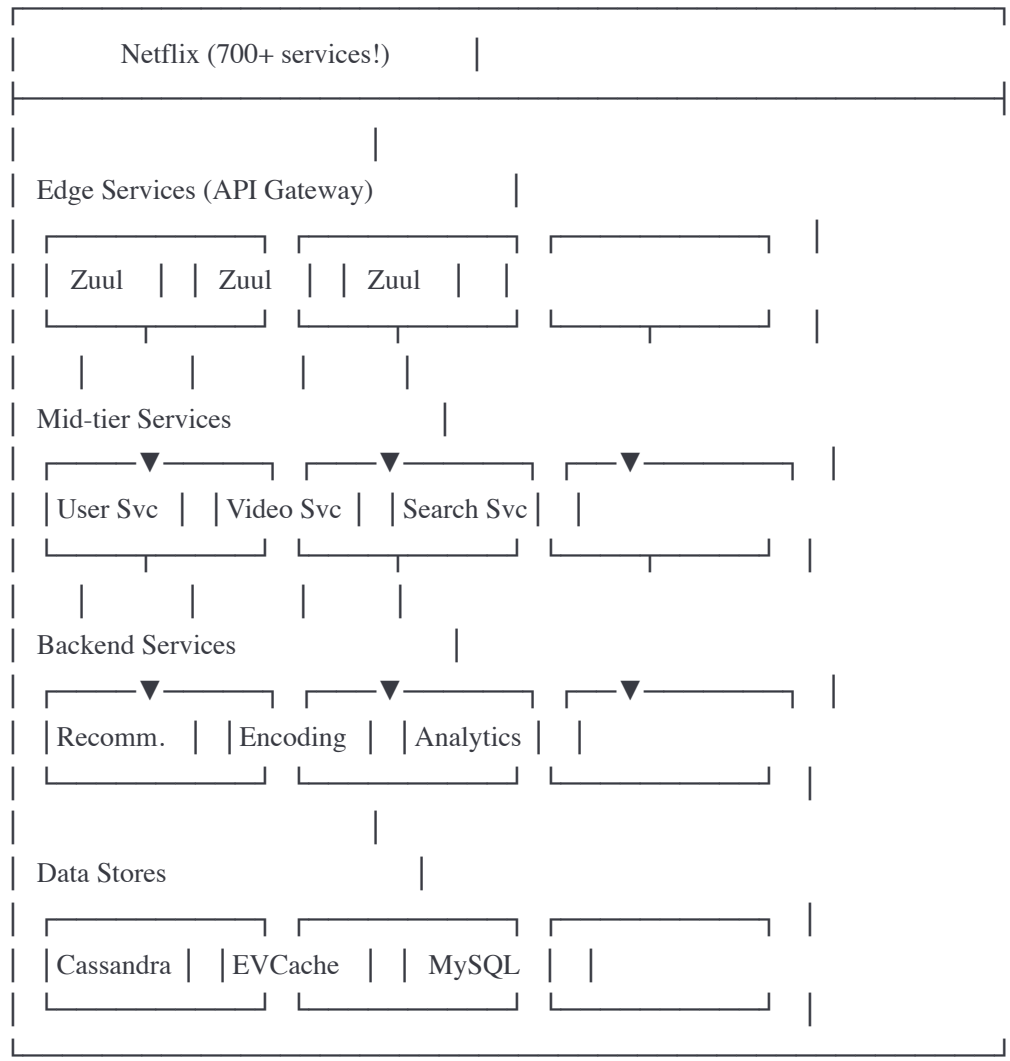
2. Service Virtualization
- Mock external services
- Test in isolation

3. End-to-end tests in staging
- Full environment
- Subset of tests (expensive)

Real-World Examples

Netflix Microservices

Architecture:

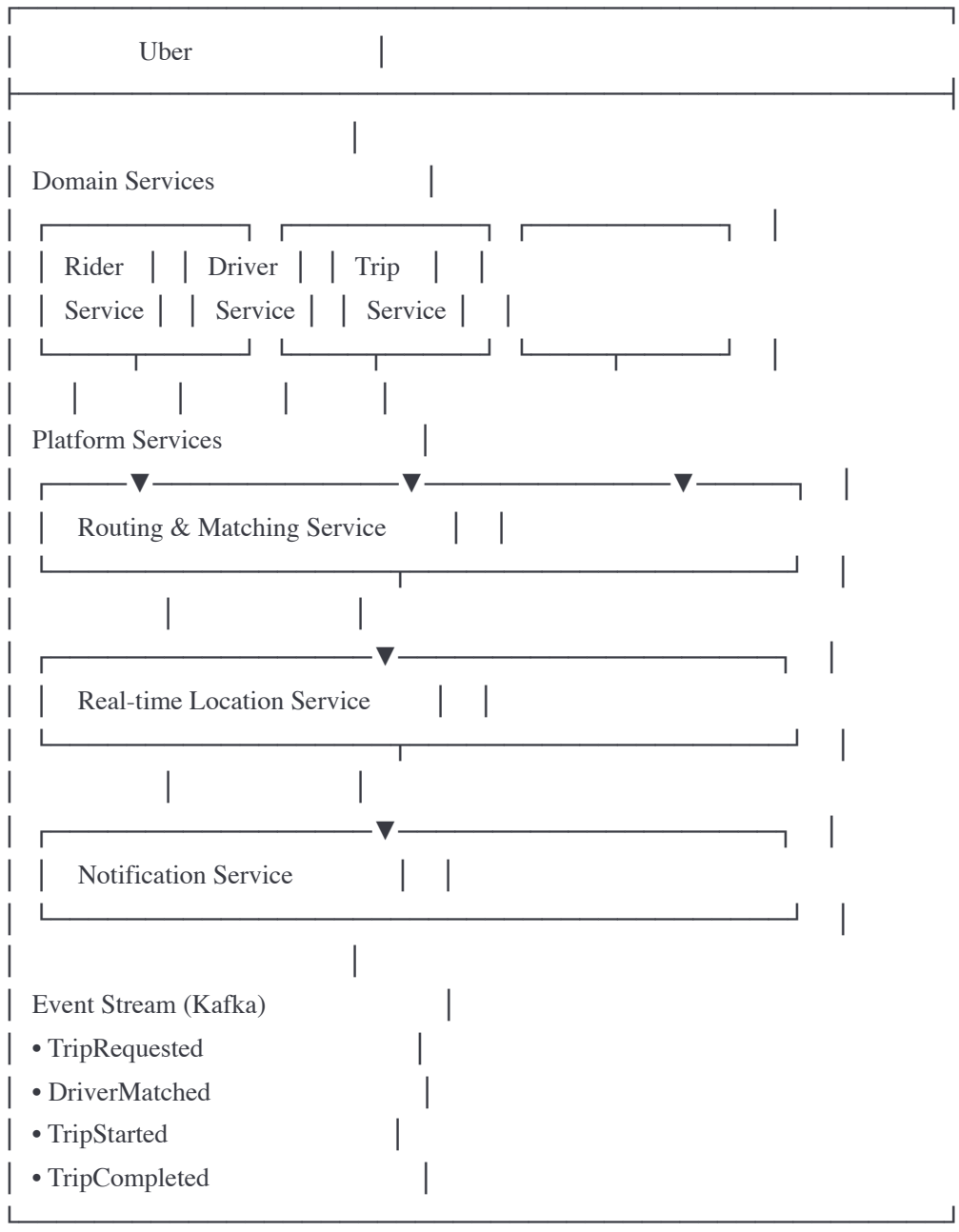


Key Practices:
- Each team owns services
- Independent deployment

- Chaos engineering daily
- Extensive monitoring

Uber's Microservices

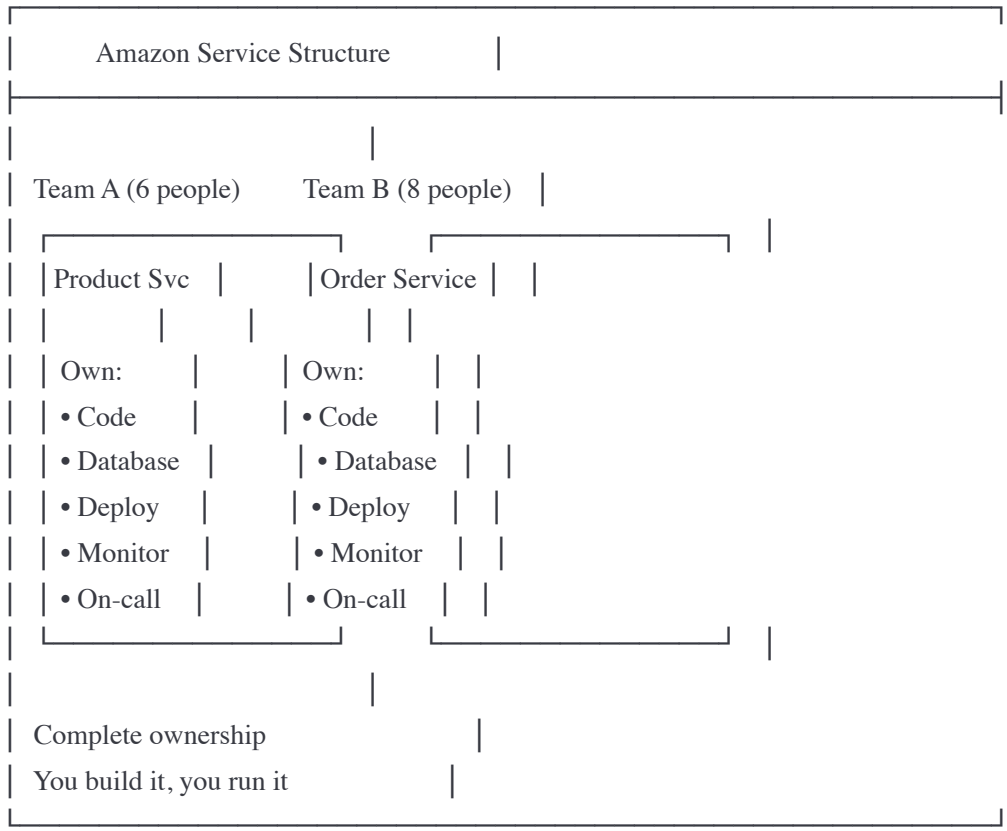
Architecture:



Communication:
- Synchronous: gRPC for low-latency calls
- Asynchronous: Kafka for events

Amazon's Two-Pizza Teams

Philosophy: Team should be small enough to feed with 2 pizzas

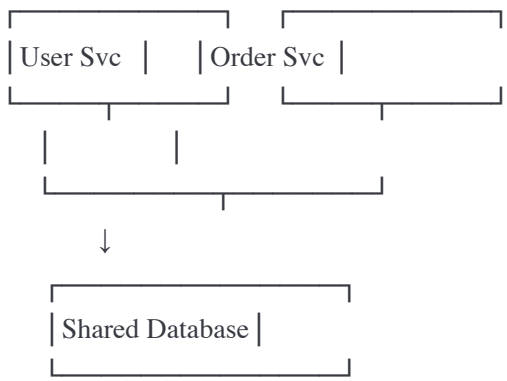


Benefits:
- Fast decision making
- Clear ownership
- Reduced coordination overhead

Microservices Best Practices

1. Database Per Service

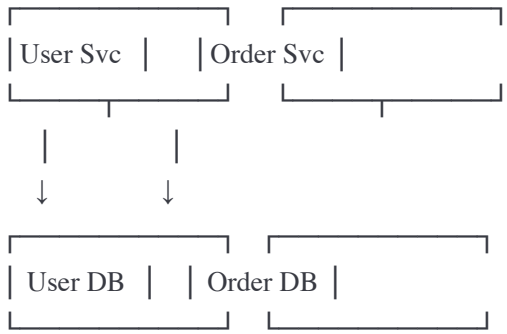
❌ Shared Database (bad):



Problems:
- Services coupled via schema
- Can't change schema freely
- Database becomes bottleneck

- No technology choice

✓ Database Per Service (good):



Benefits:

- Service independence
- Schema evolution
- Different DB types (Postgres, MongoDB, etc.)
- Scales independently

---

2. Design for Failure

javascript

```
// Every service call can fail!
// Design with resilience patterns

class ResilientOrderService {
  constructor() {
    // Circuit breakers
    this.userServiceBreaker = new CircuitBreaker();
    this.paymentServiceBreaker = new CircuitBreaker();

    // Timeouts
    this.defaultTimeout = 5000;

    // Retries
    this.retryConfig = { maxRetries: 3, backoff: 'exponential' };
  }

  async createOrder(orderData) {
    try {
      // 1. Get user (with all resilience patterns)
      const user = await this.userServiceBreaker.execute(async () => {
        return await withTimeout(
          retry(() => callUserService(orderData.userId), this.retryConfig),
          this.defaultTimeout
        );
      });

      // 2. Process payment (with fallback)
      let payment;
      try {
        payment = await this.paymentServiceBreaker.execute(async () => {
          return await callPaymentService(orderData.total);
        });
      } catch (error) {
        // Fallback: Queue payment for later
        console.log('Payment service down, queueing for later');
        payment = await queuePayment(orderData);
      }

      // 3. Create order
      const order = await db.createOrder({
        userId: user.id,
        paymentId: payment.id,
        status: payment.status === 'queued' ? 'pending_payment' : 'confirmed'
      });

      // 4. Publish event (fire and forget)
```

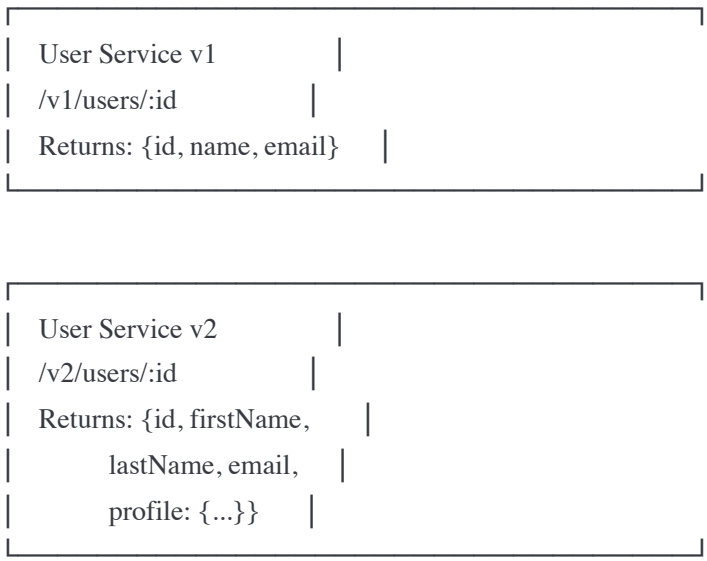
```
    await publishEvent('ORDER_CREATED', order).catch(err => {
      console.error('Failed to publish event:', err);
      // Don't fail order creation if event publish fails
    });

    return order;

  } catch (error) {
    console.error('Order creation failed:', error);
    throw error;
  }
 }
}
```

---

3. Versioning Services

Problem: Update service without breaking clients

Solution: Version your services



Both versions running simultaneously!
Clients migrate at their own pace

app.get('/v1/users/:id', v1Handler);
app.get('/v2/users/:id', v2Handler);

Or use header versioning:
Accept-Version: v1
Accept-Version: v2

Migration Strategy: Monolith to Microservices

Step-by-Step Migration

```
Phase 1: IDENTIFY BOUNDARIES
└─ Analyze monolith
└─ Find bounded contexts
└─ Draw service boundaries

Phase 2: EXTRACT VERTICAL SLICE
└─ Pick one service to extract
└─ Implement as new service
└─ Dual-write (monolith + service)
└─ Test thoroughly

Phase 3: ROUTE TRAFFIC
└─ Use API Gateway
└─ Route some traffic to new service (10%)
└─ Monitor closely
└─ Gradually increase (50%, 100%)

Phase 4: DECOMMISSION FROM MONOLITH
└─ Remove code from monolith
└─ Drop tables from monolith DB

Phase 5: REPEAT
└─ Extract next service
└─ Continue until monolith small or gone
```

Code Example (Dual Write Pattern):

```
javascript
```

```
// During migration: Write to both old and new
class UserService {
  async createUser(userData) {
    let user;

    try {
      // Write to new microservice (primary)
      user = await fetch('http://user-service/users', {
        method: 'POST',
        body: JSON.stringify(userData)
      }).then(r => r.json());

    } catch (error) {
      console.error('New service failed, using monolith');

      // Fallback to monolith
      user = await monolithDB.createUser(userData);
    }

    // Also write to monolith (for safety during migration)
    try {
      await monolithDB.createUser(userData);
    } catch (error) {
      // Log but don't fail (new service is primary)
      console.error('Monolith write failed:', error);
    }

    return user;
  }

  async getUser(userId) {
    try {
      // Read from new microservice (primary)
      return await fetch(`http://user-service/users/${userId}`)
        .then(r => r.json());

    } catch (error) {
      console.warn('New service failed, falling back to monolith');

      // Fallback to monolith
      return await monolithDB.getUser(userId);
    }
  }
}
```

```
// After migration stabilizes: Remove monolith code
```

Key Takeaways

1. **Monolith vs Microservices:**
   - Monolith: Simple, fast, good for small teams
   - Microservices: Complex, scalable, good for large orgs
   - Start monolith, evolve to microservices
2. **Service Decomposition:**
   - By business capability (recommended)
   - By subdomain (DDD)
   - Strangler fig (gradual migration)
   - Database per service
3. **Inter-Service Communication:**
   - Synchronous: gRPC/REST (immediate response)
   - Asynchronous: Events (loose coupling)
   - Use both appropriately
4. **API Gateway:**
   - Single entry point
   - Auth, rate limiting, routing
   - Request aggregation
   - Protocol translation
5. **Service Mesh:**
   - Infrastructure layer
   - Automatic resilience
   - mTLS, observability
   - Complex but powerful
6. **Challenges:**
   - Distributed transactions (Saga pattern)
   - Data consistency (eventual)

- Service discovery (Consul/etcd)
- Testing complexity
- Operational overhead

Practice Problems

1. Design a microservices architecture for an e-commerce platform. How would you decompose it?
2. How would you handle a transaction that spans 3 services (inventory, payment, shipping)?
3. When would you choose monolith over microservices? When microservices over monolith?
4. Design an API Gateway for a system with 10 microservices. What features would it have?

Next Chapter Preview

In Chapter 16, we'll explore **Distributed Systems Theory**:

- CAP theorem
- Consistency models
- Consensus algorithms (Paxos, Raft)
- Byzantine fault tolerance
- Vector clocks

Ready to continue?