Complete System Design Cheatsheet - Beginner to Advanced

Table of Contents

Fundamentals

- **System Design** is the process of defining the architecture, components, modules, interfaces, and data for a system to satisfy specified requirements.
- Focus on scalability, reliability, maintainability, and performance.
- Key concepts: client-server model, stateless vs stateful, horizontal vs vertical scaling, CAP theorem, consistency models.

Key Metrics to Consider

Scalability

- **Vertical Scaling (Scale Up)**: Adding more power to existing machines
- **Horizontal Scaling (Scale Out)**: Adding more machines to the pool of resources

Reliability

- System continues to work correctly even when failures occur
- Measured by Mean Time Between Failures (MTBF)
- Target: 99.9% uptime = 8.76 hours downtime per year

Availability

- System remains operational over time
- Measured in "nines": 99.9% = 43.8 minutes downtime per month
- High Availability (HA): 99.99% or higher

Consistency

- All nodes see the same data simultaneously
- **Strong Consistency**: All reads receive the most recent write
- **Eventual Consistency**: System will become consistent over time
- **Weak Consistency**: No guarantees when all nodes will be consistent

Partition Tolerance

- System continues to operate despite network partitions
- Essential for distributed systems

CAP Theorem

You can only guarantee 2 out of 3:

- **Consistency**: All nodes see the same data simultaneously
- **Availability**: System remains operational
- **Partition Tolerance**: System continues despite network failures

Examples:

- **CP Systems**: Traditional RDBMS (MySQL, PostgreSQL)
- **AP Systems**: DNS, Web Caching
- **CA Systems**: Single-node systems (rare in distributed environments)

ACID Properties (Databases)

- **Atomicity**: Transactions are all-or-nothing
- **Consistency**: Database remains in valid state
- **Isolation**: Concurrent transactions don't interfere
- **Durability**: Committed transactions survive system failures

BASE Properties (NoSQL)

- **Basically Available**: System guarantees availability
- **Soft State**: State may change over time
- **Eventual Consistency**: System will become consistent over time

---

System Design Principles

- **Single Responsibility Principle**: Each component should have one responsibility.
- **Loose Coupling & High Cohesion**: Minimize dependencies, group related logic.
- **Separation of Concerns**: Divide system into distinct features.
- **Fail Fast & Graceful Degradation**: Detect failures early, degrade gracefully.
- **Idempotency**: Operations can be repeated safely.
- **Backpressure**: Prevent overload by controlling request flow.

1. Single Responsibility Principle

Each component should have one reason to change.

```python
# Bad: User class handling multiple responsibilities
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

    def save_to_database(self):
        # Database logic
        pass

    def send_email(self):
        # Email logic
        pass

# Good: Separate responsibilities
class User:
    def __init__(self, name, email):
        self.name = name
        self.email = email

class UserRepository:
    def save(self, user):
        # Database logic
        pass

class EmailService:
    def send(self, user, message):
        # Email logic
        pass
```

2. Loose Coupling

Components should be independent and interact through well-defined interfaces.

3. High Cohesion

Related functionality should be grouped together.

4. Separation of Concerns

Different aspects of functionality should be separated into distinct sections.

5. Don't Repeat Yourself (DRY)

Avoid code duplication by abstracting common functionality.

6. KISS (Keep It Simple, Stupid)

Prefer simple solutions over complex ones.

Scalability Concepts

- **Horizontal Scaling**: Add more machines/instances.
- **Vertical Scaling**: Add more resources (CPU, RAM) to a single machine.
- **Load Balancing**: Distribute traffic across servers.
- **Partitioning/Sharding**: Split data across nodes.
- **Replication**: Copy data for redundancy and availability.
- **Caching**: Store frequently accessed data in fast storage.
- **Eventual Consistency**: Data will become consistent over time.

Horizontal vs Vertical Scaling

Vertical Scaling (Scale Up)

Pros:

- Simpler to implement
- No need to change application architecture
- Better for ACID compliance

Cons:

- Hardware limits
- Single point of failure
- Expensive at high end

Example:

```
# Before scaling
Server: 4 CPU cores, 8GB RAM, 100GB storage

# After vertical scaling
Server: 16 CPU cores, 64GB RAM, 1TB storage
```

Horizontal Scaling (Scale Out)

Pros:

- No theoretical limit
- Better fault tolerance
- Cost-effective

Cons:

- Complex application architecture
- Data consistency challenges
- Network latency

Example:

```
# Before scaling
1 Server: 4 CPU cores, 8GB RAM

# After horizontal scaling
4 Servers: Each with 4 CPU cores, 8GB RAM
```

Load Distribution Strategies

Round Robin

```python
class RoundRobinBalancer:
    def __init__(self, servers):
        self.servers = servers
        self.current = 0

    def get_server(self):
        server = self.servers[self.current]
        self.current = (self.current + 1) % len(self.servers)
        return server
```

Weighted Round Robin

```python
class WeightedRoundRobinBalancer:
    def __init__(self, servers_weights):
        self.servers_weights = servers_weights
        self.current_weights = [0] * len(servers_weights)

    def get_server(self):
        # Select server with highest current weight
        max_weight_index = 0
        for i, weight in enumerate(self.current_weights):
            if weight > self.current_weights[max_weight_index]:
                max_weight_index = i

        # Update weights
        self.current_weights[max_weight_index] -= sum(w for _, w in 
self.servers_weights)
        for i, (_, weight) in enumerate(self.servers_weights):
            self.current_weights[i] += weight

        return self.servers_weights[max_weight_index][0]
```

Least Connections

Routes to server with fewest active connections.

Hash-based

Routes based on client IP or session ID hash.

Database Design

- **Normalization**: Reduce redundancy, improve integrity.
- **Denormalization**: Improve read performance by duplicating data.
- **SQL vs NoSQL**: Relational (MySQL, PostgreSQL) vs Non-relational (MongoDB, Cassandra).
- **Indexing**: Speed up queries.
- **Partitioning**: Range, hash, list, composite.
- **Replication**: Master-slave, master-master.
- **ACID**: Atomicity, Consistency, Isolation, Durability.
- **BASE**: Basically Available, Soft state, Eventual consistency.

Example: User Table (SQL)

```sql
CREATE TABLE users (
    id SERIAL PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);
```

SQL vs NoSQL

SQL Databases (RDBMS)

Best for:

- Complex queries
- ACID compliance
- Structured data
- Financial systems

Examples: MySQL, PostgreSQL, Oracle, SQL Server

Scaling: Primarily vertical, read replicas for horizontal read scaling

NoSQL Databases

Document Stores

Best for: Content management, catalogs, user profiles **Examples:** MongoDB, CouchDB

```
// MongoDB Document Example
{
  "_id": "507f1f77bcf86cd799439011",
  "name": "John Doe",
  "email": "john@example.com",
  "address": {
    "street": "123 Main St",
    "city": "New York",
    "zip": "10001"
  },
  "orders": [
    {"id": 1, "total": 99.99},
    {"id": 2, "total": 149.99}
  ]
}
```

Key-Value Stores

Best for: Caching, session management, shopping carts **Examples:** Redis, DynamoDB, Riak

```python
# Redis Example
import redis
r = redis.Redis()

# Set values
r.set("user:1000:name", "John Doe")
r.set("user:1000:email", "john@example.com")

# Get values
name = r.get("user:1000:name")
```

Column-Family

Best for: Time-series data, IoT data, logging **Examples:** Cassandra, HBase

```sql
-- Cassandra Example
CREATE TABLE user_activity (
    user_id UUID,
    timestamp TIMESTAMP,
    activity TEXT,
    PRIMARY KEY (user_id, timestamp)
) WITH CLUSTERING ORDER BY (timestamp DESC);
```

Graph Databases

Best for: Social networks, recommendation engines, fraud detection **Examples:** Neo4j, Amazon Neptune

```
// Neo4j Cypher Query
MATCH (user:User {id: 123})-[:FRIEND]->(friend:User)
RETURN friend.name
```

Database Sharding

What is Sharding?

Horizontal partitioning where data is split across multiple database instances.

Sharding Strategies

Range-based Sharding

```python
def get_shard(user_id):
    if user_id <= 1000000:
        return "shard_1"
    elif user_id <= 2000000:
        return "shard_2"
    else:
        return "shard_3"
```

Hash-based Sharding

```python
def get_shard(user_id):
    shard_count = 3
    return f"shard_{hash(user_id) % shard_count + 1}"
```

Directory-based Sharding

Lookup service that knows which shard contains which data.

Challenges with Sharding

- **Rebalancing:** Moving data when adding/removing shards
- **Joins:** Complex queries across shards
- **Transactions:** Maintaining ACID across shards

Database Replication

Master-Slave Replication

```
Master (Write) → Slave 1 (Read)
               → Slave 2 (Read)
               → Slave 3 (Read)
```

Master-Master Replication

```
Master 1 ↔ Master 2
```

Advantages and Disadvantages

Pros:

- High availability
- Read scalability
- Data backup

Cons:

- Replication lag
- Complexity
- Consistency issues

Caching Strategies

- **Cache Aside (Lazy Loading)**: Application loads data into cache on demand.
- **Read Through**: Cache sits in front of database, loads data automatically.
- **Write Through**: Writes go to cache and database simultaneously.
- **Write Back (Write Behind)**: Writes go to cache, then asynchronously to database.
- **Eviction Policies**: LRU, LFU, FIFO.

Example: Cache Aside (Python)

```python
cache = {}
def get_user(user_id):
    if user_id in cache:
        return cache[user_id]
    user = db.get_user(user_id)
    cache[user_id] = user
    return user
```

Cache Levels

Browser Cache

Client-side caching for static resources.

CDN (Content Delivery Network)

Geographically distributed cache servers.

Reverse Proxy Cache

```
# Nginx caching configuration
location / {
    proxy_cache my_cache;
    proxy_cache_valid 200 302 10m;
    proxy_cache_valid 404 1m;
    proxy_pass http://backend;
}
```

Application-Level Cache

In-memory caching within application servers.

Database Cache

Query result caching at database level.

Cache Eviction Policies

LRU (Least Recently Used)

```python
from collections import OrderedDict

class LRUCache:
    def __init__(self, capacity):
        self.capacity = capacity
        self.cache = OrderedDict()

    def get(self, key):
        if key in self.cache:
            # Move to end (most recently used)
            self.cache.move_to_end(key)
            return self.cache[key]
        return None

    def put(self, key, value):
        if key in self.cache:
            # Update existing key
            self.cache.move_to_end(key)
        else:
            # Add new key
            if len(self.cache) >= self.capacity:
```

```
                # Remove least recently used
                self.cache.popitem(last=False)
        self.cache[key] = value
```

LFU (Least Frequently Used)

Evicts items used least frequently.

FIFO (First In, First Out)

Evicts oldest items first.

TTL (Time To Live)

Items expire after specified time.

Load Balancing

- **Round Robin**: Requests distributed in order.
- **Least Connections**: Send to server with fewest active connections.
- **IP Hash**: Route based on client IP.
- **Health Checks**: Remove unhealthy servers.
- **Sticky Sessions**: Keep user on same server.

Example: Simple Round Robin (Python)

```python
class LoadBalancer:
    def __init__(self, servers):
        self.servers = servers
        self.index = 0
    def get_server(self):
        server = self.servers[self.index]
        self.index = (self.index + 1) % len(self.servers)
        return server
```

Types of Load Balancers

Layer 4 (Transport Layer)

Routes based on IP and port information.

```
Client → Load Balancer → Server
         (IP:Port)
```

Layer 7 (Application Layer)

Routes based on application data (HTTP headers, URLs).

```python
# Example routing rules
def route_request(request):
    if request.path.startswith('/api/'):
        return api_servers
    elif request.path.startswith('/static/'):
        return static_servers
    else:
        return web_servers
```

Load Balancing Algorithms

Health Checks

```python
class HealthChecker:
    def __init__(self, servers):
        self.servers = servers
        self.healthy_servers = set(servers)

    def check_health(self):
        for server in self.servers:
            try:
                response = requests.get(f"http://{server}/health", timeout=5)
                if response.status_code == 200:
                    self.healthy_servers.add(server)
                else:
                    self.healthy_servers.discard(server)
            except:
                self.healthy_servers.discard(server)

    def get_healthy_servers(self):
        return list(self.healthy_servers)
```

Session Affinity (Sticky Sessions)

Ensures user requests go to same server.

```python
def get_server_with_affinity(session_id, servers):
    server_index = hash(session_id) % len(servers)
    return servers[server_index]
```

Message Queues & Communication

- **Message Queue**: Buffer for asynchronous communication (RabbitMQ, Kafka, SQS).

- **Pub/Sub**: Publishers send messages to topics, subscribers receive them.
- **At-Least-Once, At-Most-Once, Exactly-Once Delivery**: Guarantees for message delivery.
- **Backpressure**: Prevent queue overload.

Example: Simple Queue (Python)

```python
from queue import Queue
q = Queue()
q.put('message1')
msg = q.get()
```

Synchronous vs Asynchronous Communication

Synchronous (Request-Response)

```python
# HTTP API call
def get_user_data(user_id):
    response = requests.get(f"/api/users/{user_id}")
    return response.json()  # Blocks until response received
```

Asynchronous (Message-based)

```python
# Message queue
def process_user_signup(user_data):
    # Send welcome email (async)
    email_queue.send({
        'type': 'welcome_email',
        'user_data': user_data
    })

    # Update analytics (async)
    analytics_queue.send({
        'type': 'user_signup',
        'user_id': user_data['id']
    })
```

Message Queue Patterns

Point-to-Point Queue

One producer, one consumer per message.

```
Producer → Queue → Consumer
```

Publish-Subscribe

One producer, multiple consumers.

```
Publisher → Topic → Subscriber 1
                 → Subscriber 2
                 → Subscriber 3
```

Popular Message Queue Systems

Redis Pub/Sub

```python
import redis

# Publisher
r = redis.Redis()
r.publish('user_events', json.dumps({'user_id': 123, 'action': 'login'}))

# Subscriber
pubsub = r.pubsub()
pubsub.subscribe('user_events')

for message in pubsub.listen():
    if message['type'] == 'message':
        data = json.loads(message['data'])
        process_user_event(data)
```

Apache Kafka

```python
from kafka import KafkaProducer, KafkaConsumer

# Producer
producer = KafkaProducer(
    bootstrap_servers=['localhost:9092'],
    value_serializer=lambda x: json.dumps(x).encode('utf-8')
)

producer.send('user_events', {'user_id': 123, 'action': 'login'})

# Consumer
consumer = KafkaConsumer(
    'user_events',
    bootstrap_servers=['localhost:9092'],
    value_deserializer=lambda m: json.loads(m.decode('utf-8'))
)
```

```
for message in consumer:
    process_user_event(message.value)
```

RabbitMQ

```
import pika

# Producer
connection = pika.BlockingConnection(pika.ConnectionParameters('localhost'))
channel = connection.channel()

channel.queue_declare(queue='user_events')
channel.basic_publish(
    exchange='',
    routing_key='user_events',
    body=json.dumps({'user_id': 123, 'action': 'login'})
)

# Consumer
def callback(ch, method, properties, body):
    data = json.loads(body)
    process_user_event(data)

channel.basic_consume(queue='user_events', on_message_callback=callback,
auto_ack=True)
channel.start_consuming()
```

Event-Driven Architecture

Event Sourcing

Store all changes as sequence of events.

```
class EventStore:
    def __init__(self):
        self.events = []

    def append(self, event):
        self.events.append(event)

    def get_events(self, aggregate_id):
        return [e for e in self.events if e.aggregate_id == aggregate_id]

class UserAggregate:
    def __init__(self, user_id):
        self.user_id = user_id
        self.events = []
        self.name = None
```

```
        self.email = None

    def create_user(self, name, email):
        event = UserCreatedEvent(self.user_id, name, email)
        self.apply(event)
        self.events.append(event)

    def apply(self, event):
        if isinstance(event, UserCreatedEvent):
            self.name = event.name
            self.email = event.email
```

CQRS (Command Query Responsibility Segregation)

Separate read and write models.

```python
# Write Model (Commands)
class CreateUserCommand:
    def __init__(self, name, email):
        self.name = name
        self.email = email

class UserCommandHandler:
    def handle(self, command):
        user = User(command.name, command.email)
        user_repository.save(user)
        event_bus.publish(UserCreatedEvent(user.id, user.name, user.email))

# Read Model (Queries)
class UserReadModel:
    def __init__(self, user_id, name, email, created_at):
        self.user_id = user_id
        self.name = name
        self.email = email
        self.created_at = created_at

class UserQueryHandler:
    def get_user(self, user_id):
        return user_read_repository.get_by_id(user_id)
```

Microservices Architecture

- **Microservices**: Small, independent services communicating over network.
- **API Gateway**: Entry point for clients, routes requests to services.
- **Service Discovery**: Find service locations dynamically.
- **Circuit Breaker**: Prevent cascading failures.
- **Data Consistency**: Sagas, 2PC, eventual consistency.

Example: Service Communication (Python requests)

```python
import requests
response = requests.get('http://user-service/users/123')
```

Monolith vs Microservices

Monolithic Architecture

```
┌──────────────────────────┐
│     Single Application   │
│  ┌─────────────────────┐ │
│  │  User Service       │ │
│  │  Order Service      │ │
│  │  Payment Service    │ │
│  │  Notification Svc   │ │
│  └─────────────────────┘ │
│     Single Database      │
└──────────────────────────┘
```

Microservices Architecture

```
┌──────────────┐ ┌──────────────┐ ┌──────────────┐
│ User Service │ │Order Service │ │Payment Svc   │
│      ┌──────┐│ │    ┌──────┐  │ │    ┌──────┐  │
│      │ DB   ││ │    │ DB   │  │ │    │ DB   │  │
│      └──────┘│ │    └──────┘  │ │    └──────┘  │
└──────────────┘ └──────────────┘ └──────────────┘
        │                 │                 │
        └─────────────────┼─────────────────┘
                          │
                     API Gateway
```

Service Communication

REST APIs

```python
# User Service
@app.route('/users/<user_id>', methods=['GET'])
def get_user(user_id):
    user = user_repository.get_by_id(user_id)
    return jsonify(user.to_dict())

# Order Service calling User Service
def create_order(user_id, items):
```

```
    # Validate user exists
    user_response = requests.get(f"http://user-service/users/{user_id}")
    if user_response.status_code != 200:
        raise ValueError("User not found")

    order = Order(user_id, items)
    order_repository.save(order)
    return order
```

gRPC

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

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

```
# gRPC Client
import grpc
import user_pb2_grpc

channel = grpc.insecure_channel('user-service:50051')
stub = user_pb2_grpc.UserServiceStub(channel)

response = stub.GetUser(user_pb2.GetUserRequest(id='123'))
```

Service Discovery

Client-Side Discovery

```
class ServiceRegistry:
    def __init__(self):
        self.services = {}

    def register(self, service_name, host, port):
        if service_name not in self.services:
            self.services[service_name] = []
        self.services[service_name].append(f"{host}:{port}")

    def discover(self, service_name):
```

```
        return self.services.get(service_name, [])

# Service client
def call_user_service(user_id):
    instances = service_registry.discover('user-service')
    if not instances:
        raise Exception("No user-service instances available")

    # Load balance between instances
    instance = random.choice(instances)
    response = requests.get(f"http://{instance}/users/{user_id}")
    return response.json()
```

Server-Side Discovery (with Load Balancer)

Services register with a load balancer that handles discovery.

API Gateway Pattern

```
class APIGateway:
    def __init__(self):
        self.routes = {
            '/api/users': 'user-service',
            '/api/orders': 'order-service',
            '/api/payments': 'payment-service'
        }

    def route_request(self, path, request):
        service = self.find_service(path)
        if not service:
            return {'error': 'Service not found'}, 404

        # Authentication
        if not self.authenticate(request):
            return {'error': 'Unauthorized'}, 401

        # Rate limiting
        if not self.check_rate_limit(request):
            return {'error': 'Rate limit exceeded'}, 429

        # Forward request to service
        return self.forward_request(service, request)
```

Circuit Breaker Pattern

```
import time
from enum import Enum

class CircuitState(Enum):
```

```python
    CLOSED = 1
    OPEN = 2
    HALF_OPEN = 3

class CircuitBreaker:
    def __init__(self, failure_threshold=5, timeout=60):
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.failure_count = 0
        self.state = CircuitState.CLOSED
        self.last_failure_time = None

    def call(self, func, *args, **kwargs):
        if self.state == CircuitState.OPEN:
            if time.time() - self.last_failure_time > self.timeout:
                self.state = CircuitState.HALF_OPEN
            else:
                raise Exception("Circuit breaker is OPEN")

        try:
            result = func(*args, **kwargs)
            self.on_success()
            return result
        except Exception as e:
            self.on_failure()
            raise e

    def on_success(self):
        self.failure_count = 0
        self.state = CircuitState.CLOSED

    def on_failure(self):
        self.failure_count += 1
        self.last_failure_time = time.time()

        if self.failure_count >= self.failure_threshold:
            self.state = CircuitState.OPEN

# Usage
user_service_breaker = CircuitBreaker()

def get_user_with_circuit_breaker(user_id):
    return user_service_breaker.call(
        lambda: requests.get(f"http://user-service/users/{user_id}")
    )
```

Security & Authentication

- **Authentication**: Verify user identity (JWT, OAuth, SSO).
- **Authorization**: Check user permissions (RBAC, ABAC).
- **Encryption**: TLS/SSL for data in transit, AES for data at rest.

- **Input Validation**: Prevent injection attacks.
- **Rate Limiting**: Prevent abuse.
- **Audit Logging**: Track access and changes.

Example: JWT Authentication (Python)

```
import jwt
token = jwt.encode({'user_id': 123}, 'secret', algorithm='HS256')
data = jwt.decode(token, 'secret', algorithms=['HS256'])
```

Authentication vs Authorization

Authentication

Verifying identity ("Who are you?")

Authorization

Verifying permissions ("What can you do?")

Authentication Methods

Session-Based Authentication

```
from flask import Flask, session, request
import uuid

app = Flask(__name__)
app.secret_key = 'your-secret-key'

# In-memory session store (use Redis in production)
sessions = {}

@app.route('/login', methods=['POST'])
def login():
    username = request.json['username']
    password = request.json['password']

    if authenticate_user(username, password):
        session_id = str(uuid.uuid4())
        sessions[session_id] = {'user_id': username}
        session['session_id'] = session_id
        return {'success': True}
    else:
        return {'error': 'Invalid credentials'}, 401

@app.route('/protected')
def protected():
    session_id = session.get('session_id')
```

```
    if session_id and session_id in sessions:
        return {'message': 'Access granted'}
    else:
        return {'error': 'Unauthorized'}, 401
```

Token-Based Authentication (JWT)

```
import jwt
import datetime

SECRET_KEY = 'your-secret-key'

def generate_token(user_id):
    payload = {
        'user_id': user_id,
        'exp': datetime.datetime.utcnow() + datetime.timedelta(hours=24)
    }
    token = jwt.encode(payload, SECRET_KEY, algorithm='HS256')
    return token

def verify_token(token):
    try:
        payload = jwt.decode(token, SECRET_KEY, algorithms=['HS256'])
        return payload['user_id']
    except jwt.ExpiredSignatureError:
        return None
    except jwt.InvalidTokenError:
        return None

@app.route('/login', methods=['POST'])
def login():
    username = request.json['username']
    password = request.json['password']

    if authenticate_user(username, password):
        token = generate_token(username)
        return {'token': token}
    else:
        return {'error': 'Invalid credentials'}, 401

@app.route('/protected')
def protected():
    auth_header = request.headers.get('Authorization')
    if not auth_header or not auth_header.startswith('Bearer '):
        return {'error': 'No token provided'}, 401

    token = auth_header.split(' ')[1]
    user_id = verify_token(token)

    if user_id:
        return {'message': f'Access granted for user {user_id}'}
```

```
    else:
        return {'error': 'Invalid token'}, 401
```

OAuth 2.0

```python
# OAuth 2.0 Authorization Code Flow
@app.route('/auth/google')
def google_auth():
    google_auth_url = (
        "https://accounts.google.com/o/oauth2/auth?"
        "response_type=code&"
        f"client_id={GOOGLE_CLIENT_ID}&"
        f"redirect_uri={REDIRECT_URI}&"
        "scope=openid email profile"
    )
    return redirect(google_auth_url)

@app.route('/callback')
def callback():
    code = request.args.get('code')

    # Exchange code for access token
    token_response = requests.post('https://oauth2.googleapis.com/token', data={
        'client_id': GOOGLE_CLIENT_ID,
        'client_secret': GOOGLE_CLIENT_SECRET,
        'code': code,
        'grant_type': 'authorization_code',
        'redirect_uri': REDIRECT_URI
    })

    access_token = token_response.json()['access_token']

    # Get user info
    user_response = requests.get(
        'https://www.googleapis.com/oauth2/v2/userinfo',
        headers={'Authorization': f'Bearer {access_token}'}
    )

    user_info = user_response.json()
    # Create session or JWT token
    return {'user': user_info}
```

Authorization Patterns

Role-Based Access Control (RBAC)

```python
class Role:
    def __init__(self, name, permissions):
```

```
        self.name = name
        self.permissions = permissions

class User:
    def __init__(self, username, roles):
        self.username = username
        self.roles = roles

    def has_permission(self, permission):
        for role in self.roles:
            if permission in role.permissions:
                return True
        return False

# Define roles
admin_role = Role('admin', ['read', 'write', 'delete'])
user_role = Role('user', ['read', 'write'])
guest_role = Role('guest', ['read'])

# Authorization decorator
def require_permission(permission):
    def decorator(func):
        def wrapper(*args, **kwargs):
            user = get_current_user()
            if user.has_permission(permission):
                return func(*args, **kwargs)
            else:
                return {'error': 'Insufficient permissions'}, 403
        return wrapper
    return decorator

@app.route('/admin/users')
@require_permission('delete')
def delete_user():
    return {'message': 'User deleted'}
```

Attribute-Based Access Control (ABAC)

More flexible, policy-based authorization.

```
class Policy:
    def __init__(self, condition):
        self.condition = condition

    def evaluate(self, subject, resource, action, environment):
        return self.condition(subject, resource, action, environment)

def owner_policy(subject, resource, action, environment):
    return resource.owner_id == subject.user_id

def time_based_policy(subject, resource, action, environment):
```

```
    current_hour = environment.get('current_hour')
    return 9 <= current_hour <= 17  # Business hours only

# Policy engine
class PolicyEngine:
    def __init__(self):
        self.policies = []

    def add_policy(self, policy):
        self.policies.append(policy)

    def authorize(self, subject, resource, action, environment):
        for policy in self.policies:
            if not policy.evaluate(subject, resource, action, environment):
                return False
        return True

# Usage
engine = PolicyEngine()
engine.add_policy(Policy(owner_policy))
engine.add_policy(Policy(time_based_policy))

def check_access(user, document, action):
    environment = {'current_hour': datetime.now().hour}
    return engine.authorize(user, document, action, environment)
```

Security Best Practices

Input Validation

```
import re
from html import escape

def validate_email(email):
    pattern = r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}
    return re.match(pattern, email) is not None

def sanitize_input(user_input):
    # Remove HTML tags and escape special characters
    sanitized = escape(user_input.strip())
    return sanitized

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

    # Validate input
    if not validate_email(data.get('email', '')):
        return {'error': 'Invalid email format'}, 400

    if len(data.get('name', '')) < 2:
```

```
        return {'error': 'Name too short'}, 400

    # Sanitize input
    name = sanitize_input(data['name'])
    email = sanitize_input(data['email'])

    user = create_user_record(name, email)
    return {'user': user}
```

SQL Injection Prevention

```python
# Bad - Vulnerable to SQL injection
def get_user_bad(user_id):
    query = f"SELECT * FROM users WHERE id = {user_id}"
    return db.execute(query)

# Good - Using parameterized queries
def get_user_good(user_id):
    query = "SELECT * FROM users WHERE id = %s"
    return db.execute(query, (user_id,))

# Using ORM (SQLAlchemy example)
def get_user_orm(user_id):
    return User.query.filter_by(id=user_id).first()
```

Password Security

```python
import bcrypt
import secrets

def hash_password(password):
    # Generate salt and hash password
    salt = bcrypt.gensalt()
    hashed = bcrypt.hashpw(password.encode('utf-8'), salt)
    return hashed.decode('utf-8')

def verify_password(password, hashed):
    return bcrypt.checkpw(password.encode('utf-8'), hashed.encode('utf-8'))

def generate_secure_token():
    return secrets.token_urlsafe(32)

# Password strength validation
def is_strong_password(password):
    if len(password) < 8:
        return False
    if not re.search(r'[A-Z]', password):
        return False
```

```
    if not re.search(r'[a-z]', password):
        return False
    if not re.search(r'\d', password):
        return False
    if not re.search(r'[!@#$%^&*(),.?":{}|<>]', password):
        return False
    return True
```

Monitoring & Observability

- **Metrics**: CPU, memory, latency, error rates.
- **Logging**: Centralized, structured logs.
- **Tracing**: Track requests across services (OpenTracing, Jaeger).
- **Alerting**: Notify on anomalies.
- **Dashboards**: Visualize system health (Grafana, Prometheus).

Example: Simple Logging (Python)

```
import logging
logging.basicConfig(level=logging.INFO)
logging.info('Service started')
```

The Three Pillars of Observability

1. Metrics

Numerical measurements of system behavior over time.

```
from prometheus_client import Counter, Histogram, Gauge, start_http_server
import time

# Define metrics
request_count = Counter('http_requests_total', 'Total HTTP requests', ['method', 'endpoint'])
request_duration = Histogram('http_request_duration_seconds', 'HTTP request duration')
active_connections = Gauge('active_connections', 'Number of active connections')

# Middleware to collect metrics
@app.before_request
def before_request():
    request.start_time = time.time()

@app.after_request
def after_request(response):
    request_duration.observe(time.time() - request.start_time)
    request_count.labels(method=request.method, endpoint=request.endpoint).inc()
```

```
    return response

# Start metrics server
start_http_server(8000)
```

2. Logs

Discrete events with timestamps.

```
import logging
import json
from datetime import datetime

# Structured logging
class JSONFormatter(logging.Formatter):
    def format(self, record):
        log_entry = {
            'timestamp': datetime.utcnow().isoformat(),
            'level': record.levelname,
            'message': record.getMessage(),
            'module': record.module,
            'function': record.funcName,
            'line': record.lineno
        }

        # Add extra fields if present
        if hasattr(record, 'user_id'):
            log_entry['user_id'] = record.user_id
        if hasattr(record, 'request_id'):
            log_entry['request_id'] = record.request_id

        return json.dumps(log_entry)

# Configure logger
logger = logging.getLogger(__name__)
handler = logging.StreamHandler()
handler.setFormatter(JSONFormatter())
logger.addHandler(handler)
logger.setLevel(logging.INFO)

# Usage
@app.route('/users/<user_id>')
def get_user(user_id):
    logger.info('Fetching user', extra={'user_id': user_id, 'request_id':
request.headers.get('X-Request-ID')})

    try:
        user = user_service.get_user(user_id)
        logger.info('User fetched successfully', extra={'user_id': user_id})
        return jsonify(user)
    except UserNotFoundError:
```

```
        logger.warning('User not found', extra={'user_id': user_id})
        return {'error': 'User not found'}, 404
    except Exception as e:
        logger.error('Error fetching user', extra={'user_id': user_id, 'error':
str(e)})
        return {'error': 'Internal server error'}, 500
```

3. Traces

Request flows through distributed systems.

```
from opentelemetry import trace
from opentelemetry.exporter.jaeger.thrift import JaegerExporter
from opentelemetry.sdk.trace import TracerProvider
from opentelemetry.sdk.trace.export import BatchSpanProcessor

# Configure tracing
trace.set_tracer_provider(TracerProvider())
tracer = trace.get_tracer(__name__)

jaeger_exporter = JaegerExporter(
    agent_host_name="jaeger",
    agent_port=6831,
)

span_processor = BatchSpanProcessor(jaeger_exporter)
trace.get_tracer_provider().add_span_processor(span_processor)

# Instrument code with spans
@app.route('/orders', methods=['POST'])
def create_order():
    with tracer.start_as_current_span("create_order") as span:
        order_data = request.json
        span.set_attribute("order.items_count", len(order_data['items']))
        span.set_attribute("order.user_id", order_data['user_id'])

        # Validate user
        with tracer.start_as_current_span("validate_user") as user_span:
            user = validate_user(order_data['user_id'])
            user_span.set_attribute("user.validated", True)

        # Calculate total
        with tracer.start_as_current_span("calculate_total") as calc_span:
            total = calculate_order_total(order_data['items'])
            calc_span.set_attribute("order.total", total)

        # Save order
        with tracer.start_as_current_span("save_order") as save_span:
            order = save_order(order_data, total)
            save_span.set_attribute("order.id", order.id)
```

```
        return {'order_id': order.id}
```

Health Checks

Basic Health Check

```
@app.route('/health')
def health_check():
    return {'status': 'healthy', 'timestamp': datetime.utcnow().isoformat()}

# Detailed health check
@app.route('/health/detailed')
def detailed_health_check():
    health_status = {
        'status': 'healthy',
        'timestamp': datetime.utcnow().isoformat(),
        'checks': {}
    }

    # Database connectivity
    try:
        db.execute('SELECT 1')
        health_status['checks']['database'] = {'status': 'healthy'}
    except Exception as e:
        health_status['checks']['database'] = {'status': 'unhealthy', 'error':
str(e)}
        health_status['status'] = 'unhealthy'

    # External service connectivity
    try:
        response = requests.get('http://external-service/health', timeout=5)
        if response.status_code == 200:
            health_status['checks']['external_service'] = {'status': 'healthy'}
        else:
            health_status['checks']['external_service'] = {'status': 'degraded'}
    except Exception as e:
        health_status['checks']['external_service'] = {'status': 'unhealthy',
'error': str(e)}

    # Memory usage
    import psutil
    memory_percent = psutil.virtual_memory().percent
    health_status['checks']['memory'] = {
        'status': 'healthy' if memory_percent < 90 else 'warning',
        'usage_percent': memory_percent
    }

    return health_status
```

Alerting

```python
class AlertManager:
    def __init__(self):
        self.thresholds = {
            'cpu_usage': 80,
            'memory_usage': 85,
            'error_rate': 5,
            'response_time': 1000  # ms
        }
        self.alert_channels = []

    def add_channel(self, channel):
        self.alert_channels.append(channel)

    def check_metrics(self, metrics):
        alerts = []

        for metric, value in metrics.items():
            if metric in self.thresholds and value > self.thresholds[metric]:
                alert = {
                    'metric': metric,
                    'value': value,
                    'threshold': self.thresholds[metric],
                    'severity': self.get_severity(metric, value),
                    'timestamp': datetime.utcnow().isoformat()
                }
                alerts.append(alert)

        for alert in alerts:
            self.send_alert(alert)

    def send_alert(self, alert):
        for channel in self.alert_channels:
            channel.send(alert)

class SlackAlertChannel:
    def __init__(self, webhook_url):
        self.webhook_url = webhook_url

    def send(self, alert):
        message = {
            'text': f"🚨 Alert: {alert['metric']} is {alert['value']} (threshold: {alert['threshold']})"
        }
        requests.post(self.webhook_url, json=message)

# Usage
alert_manager = AlertManager()
alert_manager.add_channel(SlackAlertChannel('https://hooks.slack.com/...'))

# Check metrics periodically
```

```python
def monitor_system():
    while True:
        metrics = {
            'cpu_usage': psutil.cpu_percent(),
            'memory_usage': psutil.virtual_memory().percent,
            'error_rate': get_error_rate_last_5_minutes(),
            'response_time': get_avg_response_time_last_5_minutes()
        }

        alert_manager.check_metrics(metrics)
        time.sleep(60)  # Check every minute
```

Design Patterns

- **Singleton**: One instance per application.
- **Factory**: Create objects without specifying exact class.
- **Observer**: Notify subscribers of changes.
- **Circuit Breaker**: Stop calls to failing service.
- **Bulkhead**: Isolate failures.
- **Strangler Fig**: Gradually replace legacy system.

Example: Singleton (Python)

```python
class Singleton:
    _instance = None
    def __new__(cls, *args, **kwargs):
        if not cls._instance:
            cls._instance = super().__new__(cls)
        return cls._instance
```

Singleton Pattern

```python
class DatabaseConnection:
    _instance = None
    _lock = threading.Lock()

    def __new__(cls):
        if cls._instance is None:
            with cls._lock:
                if cls._instance is None:
                    cls._instance = super().__new__(cls)
                    cls._instance.connection = create_db_connection()
        return cls._instance

    def query(self, sql):
        return self.connection.execute(sql)
```

```python
# Usage
db1 = DatabaseConnection()
db2 = DatabaseConnection()
assert db1 is db2  # Same instance
```

Factory Pattern

```python
class DatabaseFactory:
    @staticmethod
    def create_database(db_type, config):
        if db_type == 'mysql':
            return MySQLDatabase(config)
        elif db_type == 'postgresql':
            return PostgreSQLDatabase(config)
        elif db_type == 'mongodb':
            return MongoDatabase(config)
        else:
            raise ValueError(f"Unknown database type: {db_type}")

class MySQLDatabase:
    def __init__(self, config):
        self.config = config

    def connect(self):
        # MySQL-specific connection logic
        pass

class PostgreSQLDatabase:
    def __init__(self, config):
        self.config = config

    def connect(self):
        # PostgreSQL-specific connection logic
        pass

# Usage
db = DatabaseFactory.create_database('mysql', {'host': 'localhost', 'port': 3306})
```

Observer Pattern

```python
class EventPublisher:
    def __init__(self):
        self.observers = []

    def subscribe(self, observer):
        self.observers.append(observer)

    def unsubscribe(self, observer):
        self.observers.remove(observer)
```

```python
    def notify(self, event):
        for observer in self.observers:
            observer.handle_event(event)

class EmailNotificationObserver:
    def handle_event(self, event):
        if event.type == 'user_registered':
            send_welcome_email(event.user_email)

class AnalyticsObserver:
    def handle_event(self, event):
        track_event(event.type, event.data)

# Usage
publisher = EventPublisher()
publisher.subscribe(EmailNotificationObserver())
publisher.subscribe(AnalyticsObserver())

# When user registers
publisher.notify(UserRegisteredEvent(user_email='user@example.com'))
```

Strategy Pattern

```python
class PaymentProcessor:
    def __init__(self, strategy):
        self.strategy = strategy

    def process_payment(self, amount, payment_details):
        return self.strategy.process(amount, payment_details)

class CreditCardStrategy:
    def process(self, amount, payment_details):
        # Credit card processing logic
        return {'status': 'success', 'transaction_id': 'cc_123'}

class PayPalStrategy:
    def process(self, amount, payment_details):
        # PayPal processing logic
        return {'status': 'success', 'transaction_id': 'pp_456'}

class CryptoStrategy:
    def process(self, amount, payment_details):
        # Cryptocurrency processing logic
        return {'status': 'success', 'transaction_id': 'crypto_789'}

# Usage
processor = PaymentProcessor(CreditCardStrategy())
result = processor.process_payment(100.0, {'card_number': '1234'})
```

Command Pattern

```python
class Command:
    def execute(self):
        pass

    def undo(self):
        pass

class CreateUserCommand(Command):
    def __init__(self, user_service, user_data):
        self.user_service = user_service
        self.user_data = user_data
        self.created_user_id = None

    def execute(self):
        self.created_user_id = self.user_service.create_user(self.user_data)
        return self.created_user_id

    def undo(self):
        if self.created_user_id:
            self.user_service.delete_user(self.created_user_id)

class CommandInvoker:
    def __init__(self):
        self.history = []

    def execute_command(self, command):
        result = command.execute()
        self.history.append(command)
        return result

    def undo_last_command(self):
        if self.history:
            command = self.history.pop()
            command.undo()

# Usage
invoker = CommandInvoker()
create_command = CreateUserCommand(user_service, {'name': 'John', 'email':
'john@example.com'})
user_id = invoker.execute_command(create_command)

# Later, undo the creation
invoker.undo_last_command()
```

Real-World System Examples

URL Shortener

- Use hash or counter for short URL.
- Store mapping in database.
- Cache popular URLs.
- Handle collisions.

Rate Limiter

- Token bucket or leaky bucket algorithm.
- Store counters in Redis.

Example: Token Bucket (Python)

```python
import time
class TokenBucket:
    def __init__(self, rate, capacity):
        self.rate = rate
        self.capacity = capacity
        self.tokens = capacity
        self.timestamp = time.time()
    def allow_request(self):
        now = time.time()
        elapsed = now - self.timestamp
        self.tokens = min(self.capacity, self.tokens + elapsed * self.rate)
        self.timestamp = now
        if self.tokens >= 1:
            self.tokens -= 1
            return True
        return False
```

Social Media Feed (Twitter/Instagram)

Requirements

- User posts/tweets
- Follow/unfollow users
- News feed generation
- Like/comment on posts
- Real-time notifications
- Media uploads

Database Design

```sql
-- Users table
CREATE TABLE users (
    id UUID PRIMARY KEY,
    username VARCHAR(50) UNIQUE NOT NULL,
    email VARCHAR(100) UNIQUE NOT NULL,
    display_name VARCHAR(100),
```

```sql
    bio TEXT,
    profile_image_url VARCHAR(500),
    follower_count INTEGER DEFAULT 0,
    following_count INTEGER DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Posts table
CREATE TABLE posts (
    id UUID PRIMARY KEY,
    user_id UUID REFERENCES users(id),
    content TEXT NOT NULL,
    media_urls JSON,
    like_count INTEGER DEFAULT 0,
    comment_count INTEGER DEFAULT 0,
    repost_count INTEGER DEFAULT 0,
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    updated_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP
);

-- Follows table
CREATE TABLE follows (
    follower_id UUID REFERENCES users(id),
    following_id UUID REFERENCES users(id),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (follower_id, following_id)
);

-- Likes table
CREATE TABLE likes (
    user_id UUID REFERENCES users(id),
    post_id UUID REFERENCES posts(id),
    created_at TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
    PRIMARY KEY (user_id, post_id)
);

-- Feed generation strategies
-- Option 1: Pull model (generate on request)
-- Option 2: Push model (pre-compute feeds)
-- Option 3: Hybrid model

CREATE TABLE user_feeds (
    user_id UUID REFERENCES users(id),
    post_id UUID REFERENCES posts(id),
    created_at TIMESTAMP,
    score FLOAT,  -- For ranking algorithm
    PRIMARY KEY (user_id, post_id)
);
```

Feed Generation

```python
class FeedService:
    def __init__(self):
        self.redis_client = redis.Redis()

    def generate_feed_pull_model(self, user_id, limit=20, offset=0):
        # 1. Get list of users this user follows
        following_ids = db.get_following_ids(user_id)
        # 2. Query recent posts from those users
        posts = db.query(
            "SELECT * FROM posts WHERE user_id IN %s ORDER BY created_at DESC
LIMIT %s OFFSET %s",
            (tuple(following_ids), limit, offset)
        )
        return posts
```

```python
def fanout_on_write(post_id, author_id):
    follower_ids = db.get_follower_ids(author_id)
    for follower_id in follower_ids:
        db.insert('user_feeds', {
        'user_id': follower_id,
        'post_id': post_id,
        'created_at': datetime.utcnow(),
        'score': compute_score(post_id, follower_id)
        })
```

Feed Ranking

```python
def compute_score(post, user):
    # Example: combine recency and engagement
    age_seconds = (datetime.utcnow() - post['created_at']).total_seconds()
    engagement = post['like_count'] + post['comment_count']
    return engagement * 2 - age_seconds / 3600  # Higher is better
```

Real-Time Notifications

- Use message queues (e.g., Redis, Kafka) to notify users of new posts, likes, comments, etc.
- WebSockets or push notifications for instant updates.

```python
def notify_followers(post_id, author_id):
    follower_ids = db.get_follower_ids(author_id)
    for follower_id in follower_ids:
        notification_service.send(follower_id, f"New post from {author_id}:
{post_id}")
```

Caching

- Cache hot feeds in Redis or Memcached for fast access.
- Cache user timelines, post details, and counts.

Advanced Topics

Event Sourcing

Instead of storing current state, you store every change as an event. Think of it like a bank account - rather than just keeping the current balance ($500), you store every transaction:

- Day 1: +$1000 (deposit)
- Day 2: -$300 (withdrawal)
- Day 3: -$200 (withdrawal)

To get current state, you replay all events: 1000 - 300 - 200 = $500

Benefits:

- Complete audit trail of all changes
- Can rebuild state at any point in time
- Enables time travel debugging
- Natural fit for collaborative systems

Drawbacks:

- Increased complexity
- Storage grows continuously
- Eventual consistency
- Event schema evolution challenges

When to Use: Financial systems, collaborative editing, systems requiring full audit trails

---

CQRS (Command Query Responsibility Segregation)

Separate your read operations from write operations using different models and potentially different databases.

Commands: Modify data (CreateOrder, UpdateInventory) **Queries:** Read data (GetProductCatalog, GetUserDashboard)

Example Architecture:

```
Write Side: Normalized PostgreSQL → Events → Message Bus
Read Side: Denormalized MongoDB/Redis ← Event Handlers
```

Benefits:

- Independent scaling (read-heavy vs write-heavy)
- Optimized data models for each use case
- Better performance for complex queries
- Can use different technologies for reads vs writes

Trade-offs:

- Added complexity
- Data synchronization between read/write models
- Eventual consistency

When to Use: High-scale applications with different read/write patterns, complex reporting requirements

---

Distributed Transactions

Two-Phase Commit (2PC)

Phase 1 (Prepare): Coordinator asks all participants "Can you commit?" **Phase 2 (Commit/Abort):** If all say yes, coordinator tells everyone to commit

Problems:

- Blocking protocol - if coordinator fails, participants wait indefinitely
- Not suitable for microservices across WAN
- Performance overhead

Sagas

Break large transactions into smaller compensatable steps. If any step fails, run compensating actions for completed steps.

Example - Order Processing Saga:

1. Reserve inventory → Compensate: Release inventory
2. Charge payment → Compensate: Refund payment
3. Create shipment → Compensate: Cancel shipment

If step 2 fails, automatically release the reserved inventory.

Types:

- **Choreography:** Each service knows what to do next
- **Orchestration:** Central coordinator manages the workflow

When to Use: Microservices architectures, long-running business processes

---

Geo-Replication

Distribute data across multiple geographic regions for better performance and disaster recovery.

Patterns:

- **Master-Slave:** One region handles writes, others serve reads
- **Multi-Master:** Multiple regions can handle writes (conflict resolution needed)
- **Sharding by Geography:** US users → US data center, EU users → EU data center

Challenges:

- Network latency between regions
- Data consistency across regions
- Conflict resolution for concurrent updates
- Compliance with data residency laws

Technologies: AWS RDS Cross-Region, Google Cloud Spanner, MongoDB Global Clusters

---

API Rate Limiting

Control the number of requests clients can make to prevent abuse and ensure fair usage.

Global Rate Limiting

Limit across entire system: "1M requests/hour for the entire API"

Per-User Rate Limiting

Individual limits: "1000 requests/hour per user"

Algorithms:

- **Token Bucket:** Refill tokens at fixed rate, consume tokens per request
- **Sliding Window:** Track requests in rolling time window
- **Fixed Window:** Reset counter every time period

Implementation:

```
Redis: INCR user:123:requests:2024-06-07-14
If count > limit: return 429 Too Many Requests
Set TTL to expire at end of window
```

Headers to Return:

- X-RateLimit-Limit: 1000
- X-RateLimit-Remaining: 999
- X-RateLimit-Reset: 1717776000

Chaos Engineering

Deliberately introduce failures to test system resilience and identify weaknesses before they cause outages.

Principles:

1. Hypothesize steady state behavior
2. Vary real-world events (server crashes, network delays)
3. Run experiments in production (carefully!)
4. Automate experiments

Common Experiments:

- **Latency:** Add random delays to service calls
- **Error Injection:** Return errors from dependencies
- **Resource Exhaustion:** Consume CPU/memory
- **Network Partitions:** Simulate network splits

Tools:

- Netflix Chaos Monkey (terminates instances)
- Gremlin (comprehensive chaos platform)
- Litmus (Kubernetes chaos engineering)

Example: "What happens if our payment service becomes 50% slower?" Run experiment and measure impact on user experience.

Blue-Green Deployments

Maintain two identical production environments and switch traffic between them for zero-downtime deployments.

Process:

1. **Blue Environment:** Currently serving production traffic
2. **Green Environment:** Deploy new version here
3. **Testing:** Verify green environment works correctly
4. **Switch:** Route traffic from blue to green
5. **Rollback:** If issues arise, instantly switch back to blue

Benefits:

- Zero downtime deployments
- Instant rollback capability
- Full production testing before switch
- Reduced deployment risk

Requirements:

- Load balancer that can switch traffic
- Identical infrastructure for both environments
- Database migration strategy
- Monitoring to detect issues quickly

Variations:

- **Canary Deployments:** Gradually shift percentage of traffic
- **A/B Testing:** Split traffic to test different versions
- **Rolling Updates:** Replace instances one by one

Tools: AWS CodeDeploy, Kubernetes rolling updates, HAProxy, NGINX

When to Use These Patterns

Event Sourcing + CQRS: Financial systems, collaborative platforms, audit-heavy domains **Distributed Transactions:** Microservices with cross-service business transactions **Geo-Replication:** Global applications, disaster recovery requirements **Rate Limiting:** Public APIs, preventing abuse, fair usage policies **Chaos Engineering:** Critical systems, microservices architectures **Blue-Green Deployments:** Applications requiring high availability, zero-downtime requirements