# Software Design and Architecture

## Building Scalable and Maintainable Systems

Understanding the foundations of great software systems

Instructor: Mehdi Lotfinejad

# 🎯 Learning Objectives

By the end of this lesson, you will be able to:

1. **Distinguish** between software design and architecture
2. **Explain** key principles: separation of concerns, modularity, abstraction
3. **Identify** major architectural patterns (Layered, MVC, Microservices, Event-Driven)
4. **Apply** SOLID principles at scale
5. **Evaluate** architectural trade-offs
6. **Design** system architectures for specific requirements

# The House vs. City Analogy

## Building a House (Design)

- Room layouts
- Plumbing connections
- Electrical wiring
- Interior details
- Furniture placement

**Detailed implementation**

## Planning a City (Architecture)

- Neighborhoods
- Transportation systems
- Utility distribution
- District interactions
- Zone planning

**High-level structure**

**Both essential, different scales, different concerns**

# Why Architecture Matters

**The Cost of Bad Architecture:**

- System unable to scale from 1K to 1M users
- Years paying off technical debt
- Expensive to change fundamental decisions
- Team productivity grinds to halt

**The Benefit of Good Architecture:**

- System adapts to changing needs
- Easy to maintain and enhance
- Scales with business growth
- Team moves fast with confidence

# Architecture vs. Design: The Distinction

## Architecture (System Level)

**Big decisions:**

- Monolith vs. microservices
- Communication protocols
- Data storage strategy
- Security boundaries
- Deployment strategy

**Expensive to change**

## Design (Component Level)

**Implementation details:**

- Class structures
- Algorithm choices
- Data structures
- Implementation patterns
- Code organization

**Easier to change**

# Example: E-Commerce System

```
SYSTEM LEVEL (Architecture):
+------------------------------------------------------------+
|                  E-Commerce System                         |
+-------------+-------------+-------------+-----------------+
|   Frontend  |   Backend   |   Database  |    Payment      |
|   (React)   |   (Python)  | (PostgreSQL)|    Service      |
+-------------+-------------+-------------+-----------------+
      |             |              |              |
      +----HTTP-----+-----SQL------+----HTTPS-----+

COMPONENT LEVEL (Design):
Backend Component Structure:
+-- controllers/
|   +-- ProductController (HTTP requests)
|   +-- OrderController (order processing)
|   +-- UserController (user management)
+-- services/
|   +-- ProductService (business logic)
|   +-- OrderService (order processing)
+-- repositories/
|   +-- ProductRepository (data access)
|   +-- OrderRepository (persistence)
+-- models/
    +-- Product (data structures)
    +-- Order (domain entities)
```

# Architecture Determines Communication

**Architecture Level:**

- How components communicate: HTTP, gRPC, message queues
- Where data is stored: SQL, NoSQL, cache
- Security boundaries: authentication, authorization
- Deployment strategy: containers, VMs, serverless

**Design Level:**

- How classes interact within backend
- Which algorithms to use
- How to structure data
- Implementation patterns

**Key Point:** Architecture = System blueprint, Design = Component details

# 🏗️ Fundamental Architectural Principles

**Timeless principles that guide great architectures:**

1. Separation of Concerns
2. Modularity and Encapsulation
3. Abstraction and Information Hiding
4. Loose Coupling and High Cohesion

**These apply regardless of technology choices or programming languages**

# Principle 1: Separation of Concerns (SoC)

**Core Idea:**

Different aspects of the system should be isolated into distinct sections

**A "concern" is a specific aspect:**

- User interface
- Business logic
- Data access
- Logging
- Security
- Validation

**Benefit:** Change one aspect without touching others

# SoC: Bad vs. Good Example

## ❌ Everything Mixed

```python
def process_order(request):
    # Validation
    if not request.POST.get('id'):
        return "<html>Error</html>"

    # Database
    conn = psycopg2.connect(...)
    price = conn.execute(...)

    # Business logic
    total = price * 1.1

    # Persistence
    conn.execute("INSERT...")

    # Presentation
    return f"<html>{total}</html>"
```

## ✅ Clear Separation

```python
# Presentation
class OrderController:
    def create(self, request):
        data = self._validate(request)
        order = self.service.create(data)
        return self._respond(order)

# Business Logic
class OrderService:
    def create(self, data):
        product = self.repo.find(data['id'])
        order = Order(product, data['qty'])
        return self.repo.save(order)

# Data Access
class OrderRepository:
    def save(self, order):
        # SQL operations
```

# Benefits of Separation of Concerns

**Change Database:**

- Modify only Repository layer
- Business logic untouched
- UI remains unchanged

**Change UI:**

- Switch HTML to JSON
- Business logic unaffected
- Database queries unchanged

**Change Business Rules:**

- Update Service layer only
- UI and database unaffected

**Result:** Isolated, maintainable changes

# Principle 2: Modularity and Encapsulation

**Modularity:** Divide system into discrete, self-contained modules

**Encapsulation:** Hide internal details, expose only necessary interfaces

**Think of kitchen appliances:**

- Refrigerator has simple interface (door, controls)
- Internal complexity hidden (compressor, defrost)
- Can replace without rewiring kitchen

# Modularity Example

```python
class PaymentProcessor:
    """Public interface - stable contract"""

    def process_payment(self, amount, method, customer_id):
        """Client code depends on this stable interface"""
        self._validate_payment_request(amount, method)
        handler = self._get_payment_handler(method)
        result = self._process_with_retry(handler, amount, customer_id)
        self._log_transaction(result)
        return result

    # Private methods - hidden implementation
    def _validate_payment_request(self, amount, method): ...
    def _get_payment_handler(self, method): ...
    def _process_with_retry(self, handler, amount, customer_id): ...
    def _log_transaction(self, result): ...

# Client only sees public interface
processor = PaymentProcessor()
result = processor.process_payment(99.99, 'credit_card', 'CUST123')
```

# Benefits of Modularity

**Change Implementation:**

- Add new payment methods
- Modify retry strategies
- Switch logging systems
- Update validation rules

**Without breaking clients:**

- Interface remains stable
- Client code unchanged
- Implementation evolves independently

**Result:** Flexibility and maintainability

# Principle 3: Abstraction and Information Hiding

**Abstraction:** Focus on essential characteristics, hide unnecessary details

**Information Hiding:** Conceal implementation behind interfaces

**Like driving a car:**

- Interface: steering wheel, pedals, gear shift
- Hidden: engine timing, fuel injection, transmission ratios
- You don't need to understand internals to drive

# Abstraction Example

```python
from abc import ABC, abstractmethod

# Abstract interface - defines WHAT, not HOW
class DataStorage(ABC):
    @abstractmethod
    def save(self, key: str, data: dict) -> bool: pass

    @abstractmethod
    def retrieve(self, key: str) -> dict: pass

# Concrete implementations - HOW is hidden
class SQLStorage(DataStorage):
    def save(self, key, data):
        # SQL-specific implementation hidden
        self.conn.execute("INSERT INTO...", (key, json.dumps(data)))
        return True

class MongoStorage(DataStorage):
    def save(self, key, data):
        # MongoDB-specific implementation hidden
        self.collection.insert_one({'_id': key, 'data': data})
        return True
```

# Using Abstraction

```python
# Client depends on abstraction, not concrete class
class UserService:
    def __init__(self, storage: DataStorage):  # Interface
        self.storage = storage

    def create_user(self, user_id, user_data):
        return self.storage.save(user_id, user_data)

# Can switch implementations without changing UserService
sql_storage = SQLStorage("postgresql://prod-db")
user_service = UserService(sql_storage)

# Later, switch to MongoDB - UserService unchanged
mongo_storage = MongoStorage("mongodb://new-db", "users")
user_service = UserService(mongo_storage)
```

**Benefit:** Change storage technology without touching business logic

# Principle 4: Loose Coupling & High Cohesion

## Loose Coupling

### Minimal interdependence

Like separate appliances:

- Each plugged into outlet
- Independent operation
- Break one, others work
- Share only interface

## High Cohesion

### Elements strongly related

Like organized toolbox:

- Screwdriver drawer
- Hammer drawer
- Each clear purpose
- Related items together

# Coupling: Bad vs. Good

## ❌ Tight Coupling

```
class OrderManager:
    def create_order(self, data):
        # Direct dependencies
        db = PostgreSQLDatabase()
        email = SMTPEmailer()
        stripe = stripe.Client('key')

        # Locked to concrete classes
        # Hard to test
        # Hard to change
```

## ✅ Loose Coupling

```
class OrderService:
    def __init__(self,
                 repo,       # Interface
                 notifier,   # Interface
                 payment):   # Interface
        self.repo = repo
        self.notifier = notifier
        self.payment = payment

    def create_order(self, data):
        # Depends on abstractions
        # Easy to test (mock deps)
        # Easy to change (swap impl)
```

# Benefits of Loose Coupling & High Cohesion

**Loose Coupling:**

- Change one part without breaking others
- Easy to test (mock dependencies)
- Reusable components
- Independent deployment

**High Cohesion:**

- Clear module purpose
- Easy to understand
- Changes localized
- Natural organization

**Together:** Maintainable, flexible systems

# 🏛️ Major Architectural Patterns

**Proven solutions to common structural problems**

1. Layered (N-Tier) Architecture
2. Model-View-Controller (MVC)
3. Microservices Architecture
4. Event-Driven Architecture

**Each pattern solves different problems at different scales**

# Pattern 1: Layered (N-Tier) Architecture

## Organize code into horizontal layers

```
+--------------------------+
|  Presentation Layer      | <- HTTP, JSON responses
|  (Controllers/Views)     |
+--------------------------+
           |
           v
+--------------------------+
|  Business Logic Layer    | <- Rules, validation
|  (Services)              |
+--------------------------+
           |
           v
+--------------------------+
|  Data Access Layer       | <- SQL queries, ORM
|  (Repositories)          |
+--------------------------+
           |
           v
+--------------------------+
|  Database Layer          | <- PostgreSQL, MySQL
+--------------------------+
```

# Layered Architecture: Code Example

```python
# PRESENTATION LAYER
class ProductController:
    def __init__(self, product_service):
        self.product_service = product_service

    def get_products(self):
        category = request.args.get('category')
        products = self.product_service.find_products(category)
        return jsonify({'products': [self._format(p) for p in products]})

# BUSINESS LOGIC LAYER
class ProductService:
    def __init__(self, product_repo, inventory_repo):
        self.product_repo = product_repo
        self.inventory_repo = inventory_repo

    def find_products(self, category=None):
        products = self.product_repo.find_all()
        if category:
            products = [p for p in products if p.category == category]
        return [p for p in products if self.inventory_repo.is_in_stock(p.id)]
```

# Layered Architecture: Data Layer

```python
# DATA ACCESS LAYER
class ProductRepository:
    def __init__(self, db_connection):
        self.conn = db_connection

    def find_all(self):
        query = "SELECT id, name, price, category FROM products"
        cursor = self.conn.execute(query)
        return [self._row_to_product(row) for row in cursor.fetchall()]

    def _row_to_product(self, row):
        return Product(
            id=row[0],
            name=row[1],
            price=row[2],
            category=row[3]
        )
```

# Layered Architecture: When to Use

## ✅ Benefits

- Clear separation
- Easy to understand
- Testable layers
- Team organization
- Structured approach

## ✅ Best For

- Enterprise apps
- CRUD operations
- Traditional monoliths
- Small to medium apps

## ❌ Drawbacks

- Performance overhead
- Can become rigid
- Database-centric
- Cross-layer changes

## ❌ Not Ideal For

- High-performance needs
- Rapidly changing requirements
- Complex business logic
- Microservices

# Pattern 2: Model-View-Controller (MVC)

## Separate concerns: data, presentation, and control

```
        User Input
            |
            v
     +---------------+
     |  Controller   | <- Receives user actions
     +---------------+
       |         |
       v         v
+---------+  +---------+
| Model   |  |  View   |
| (Data)  |->| (UI)    | <- Model notifies View
+---------+  +---------+
```

**Controller** handles input → updates **Model** → **View** displays

# MVC: Model (Business Logic)

```python
# MODEL: Business logic and data
class BlogPost:
    def __init__(self, title, content, author):
        self.title = title
        self.content = content
        self.author = author
        self.created_at = datetime.now()
        self.status = 'draft'

    def publish(self):
        if not self.title or not self.content:
            raise ValueError("Title and content required")
        self.status = 'published'
        self.published_at = datetime.now()

    def can_be_edited_by(self, user):
        return user.is_admin or user.username == self.author
```

# MVC: View (Presentation)

```python
# VIEW: Presentation logic
class BlogView:
    def render_post_list(self, posts):
        html = "<html><body><h1>Blog Posts</h1><ul>"
        for post in posts:
            html += f'<li><a href="/posts/{post.id}">{post.title}</a></li>'
        html += "</ul></body></html>"
        return html

    def render_post_detail(self, post):
        html = f"""
        <html><body>
            <h1>{post.title}</h1>
            <p>By {post.author} on {post.created_at}</p>
            <div>{post.content}</div>
        </body></html>
        """
        return html

    def render_error(self, message):
        return f'<html><body><h1>Error</h1><p>{message}</p></body></html>'
```

# MVC: Controller (Orchestration)

```python
# CONTROLLER: Handles user input
class BlogController:
    def __init__(self, repository, view):
        self.repository = repository  # Model
        self.view = view             # View

    def list_posts(self, request):
        posts = self.repository.find_all_published()
        html = self.view.render_post_list(posts)
        return Response(html, status=200)

    def update_post(self, request, post_id):
        post = self.repository.find_by_id(post_id)
        post.title = request.form['title']
        post.content = request.form['content']
        post.publish()
        self.repository.save(post)
        return Response(status=302, headers={'Location': f'/posts/{post.id}'})
```

# MVC: When to Use

## ✅ Benefits

- Clear UI separation
- Parallel development
- Multiple views possible
- Testable business logic
- Framework support

## ✅ Best For

- Web applications
- Significant UI
- Multiple interfaces
- Django, Rails, ASP.NET

## ❌ Drawbacks

- Can be complex
- Responsibility confusion
- View-Controller coupling
- Overkill for simple apps

## ❌ Avoid When

- Simple CRUD
- No UI complexity
- API-only services
- Very small projects

# Pattern 3: Microservices Architecture

## Application as collection of small, independent services

```
+------------------+     +------------------+     +------------------+
| Product Service  |     | Order Service    |     | Payment Service  |
| - PostgreSQL     |     | - MongoDB        |     | - Stripe API     |
| - Port 8001      |     | - Port 8002      |     | - Port 8003      |
+------------------+     +------------------+     +------------------+
         |                       |                        |
         +----------HTTP/REST----+--------Events----------+
```

## Each service:

- Runs independently
- Own database
- Separate deployment
- Lightweight communication

# Microservices: Product Service

```python
# PRODUCT SERVICE (independent)
from flask import Flask, jsonify

app_products = Flask(__name__)

@app_products.route('/api/products/<product_id>', methods=['GET'])
def get_product(product_id):
    product = db.query_one(
        "SELECT * FROM products WHERE id = %s",
        (product_id,)
    )
    return jsonify({
        'product': {
            'id': product.id,
            'name': product.name,
            'price': product.price
        }
    })

if __name__ == '__main__':
    app_products.run(port=8001)
```

# Microservices: Order Service

```python
# ORDER SERVICE (calls Product Service)
import requests

@app_orders.route('/api/orders', methods=['POST'])
def create_order():
    data = request.get_json()

    # Call Product Service via HTTP
    products = []
    for item in data['items']:
        response = requests.get(
            f"http://product-service:8001/api/products/{item['id']}"
        )
        products.append(response.json()['product'])

    # Create order in this service's database
    order = {
        'items': data['items'],
        'total': sum(p['price'] * item['qty'] for p, item in zip(products, data['items']))
    }
    order_id = db.orders.insert_one(order).inserted_id

    # Publish event for other services
    event_bus.publish('order.created', {'order_id': str(order_id)})
    return jsonify({'order_id': str(order_id)}), 201
```

# Microservices: Circuit Breaker Pattern

```python
# Prevent cascading failures
class CircuitBreaker:
    def __init__(self, failure_threshold=5, timeout=60):
        self.failure_count = 0
        self.failure_threshold = failure_threshold
        self.timeout = timeout
        self.state = 'CLOSED'  # CLOSED, OPEN, HALF_OPEN
        self.last_failure_time = None

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

        try:
            result = func(*args, **kwargs)
            if self.state == 'HALF_OPEN':
                self.state = 'CLOSED'
                self.failure_count = 0
            return result
        except Exception as e:
            self.failure_count += 1
            self.last_failure_time = time.time()
            if self.failure_count >= self.failure_threshold:
                self.state = 'OPEN'
            raise
```

# Microservices: When to Use

## ✅ Benefits

- Independent deployment
- Technology diversity
- Scalability per service
- Team autonomy
- Fault isolation

## ✅ Best For

- Large complex apps
- Multiple teams
- Different scaling needs
- Frequent updates
- DevOps maturity

## ❌ Drawbacks

- Distributed complexity
- Network overhead
- Data consistency hard
- Testing difficulty
- Operational overhead

## ❌ Avoid When

- Small team (< 10 devs)
- Simple application
- No scaling needs
- Limited DevOps skills
- Tight coupling required

# Pattern 4: Event-Driven Architecture

## Components communicate via events

```
Order Service                     Inventory Service
     |                                 |
     | 1. Create Order                 |
     |                                 |
     | 2. Publish "order.created"      |
     +--------------> Event Bus --------+
                         |            |
                         v            v
                  Payment Service  Notification
                                   Service
```

## Asynchronous, decoupled, scalable

# Event-Driven: Event Infrastructure

```python
# Event Infrastructure
class Event:
    def __init__(self, event_type, data):
        self.event_type = event_type
        self.data = data
        self.event_id = str(uuid.uuid4())
        self.timestamp = datetime.now()

class EventBus:
    def __init__(self):
        self.subscribers = {}

    def publish(self, event):
        """Publish event - fire and forget"""
        if event.event_type in self.subscribers:
            for handler in self.subscribers[event.event_type]:
                # Asynchronous processing
                threading.Thread(target=handler, args=(event,)).start()

    def subscribe(self, event_type, handler):
        """Subscribe to specific event type"""
        if event_type not in self.subscribers:
            self.subscribers[event_type] = []
        self.subscribers[event_type].append(handler)
```

# Event-Driven: Producer & Consumer

```python
# EVENT PRODUCER
class OrderService:
    def __init__(self, event_bus):
        self.event_bus = event_bus

    def create_order(self, customer_id, items):
        order = {'order_id': str(uuid.uuid4()), 'customer_id': customer_id, 'items': items}
        self.repository.save(order)

        # Publish event - don't wait for consumers
        event = Event('order.created', order)
        self.event_bus.publish(event)
        return order['order_id']

# EVENT CONSUMER
class InventoryService:
    def __init__(self, event_bus):
        event_bus.subscribe('order.created', self.handle_order_created)

    def handle_order_created(self, event):
        order = event.data
        for item in order['items']:
            self.reserve_stock(item['product_id'], item['quantity'])
```

# Event-Driven: When to Use

## ✅ Benefits

- Loose coupling
- Easy to scale
- Resilience
- Audit trail
- Add consumers easily

## ✅ Best For

- Complex workflows
- Multiple services
- Real-time updates
- Audit requirements
- Independent scaling

## ❌ Drawbacks

- Complexity
- Debugging difficulty
- Eventual consistency
- Schema management
- Learning curve

## ❌ Avoid When

- Simple workflows
- Immediate consistency needed
- Small team
- Synchronous requirements
- Tight deadlines

# 🎯 SOLID Principles at Scale

## SOLID scales from classes to architectures

**S** - Single Responsibility Principle

**O** - Open/Closed Principle

**L** - Liskov Substitution Principle

**I** - Interface Segregation Principle

**D** - Dependency Inversion Principle

# SOLID: Single Responsibility Principle

**Each service/module should have one reason to change**

## ❌ God Service

```python
class UserService:
    def register_user(self): ...
    def authenticate(self): ...
    def send_email(self): ...
    def process_payment(self): ...
    def generate_reports(self): ...
```

**Too many responsibilities!**

## ✅ Focused Services

```python
class UserRegistrationService:
    # Only registration

class AuthenticationService:
    # Only authentication

class NotificationService:
    # Only notifications

class BillingService:
    # Only billing
```

**One responsibility each**

# SOLID: Open/Closed Principle

**Open for extension, closed for modification**

```python
# Plugin architecture - extend without modifying core
class PaymentProcessor:
    def __init__(self):
        self.handlers = {}

    def register_handler(self, payment_type, handler):
        """Extend by adding new handlers"""
        self.handlers[payment_type] = handler

    def process(self, payment_type, amount):
        """Core logic unchanged"""
        if payment_type not in self.handlers:
            raise ValueError(f"Unknown payment type: {payment_type}")
        return self.handlers[payment_type].process(amount)

# Add new payment method WITHOUT modifying PaymentProcessor
processor.register_handler('crypto', CryptoPaymentHandler())
processor.register_handler('paypal', PayPalHandler())
```

# SOLID: Liskov Substitution Principle

**Services implementing interface should be substitutable**

```python
# Any implementation should work without breaking system
class MessageQueue(ABC):
    @abstractmethod
    def publish(self, message): pass

    @abstractmethod
    def consume(self): pass

# Can swap implementations
class RabbitMQQueue(MessageQueue): ...
class KafkaQueue(MessageQueue): ...
class SQSQueue(MessageQueue): ...

# System works with any implementation
def process_orders(queue: MessageQueue):
    message = queue.consume()
    # Works regardless of concrete queue type
```

# SOLID: Interface Segregation Principle

**Design focused, minimal APIs**

## ❌ Fat Interface

```
class StorageService:
    def read_file(self): ...
    def write_file(self): ...
    def delete_file(self): ...
    def backup_database(self): ...
    def replicate_data(self): ...
    def compress_archive(self): ...
```

**Too much in one interface**

## ✅ Segregated

```
class FileStorage:
    def read(self): ...
    def write(self): ...

class DatabaseBackup:
    def backup(self): ...
    def restore(self): ...

class DataReplication:
    def replicate(self): ...
```

**Focused interfaces**

# SOLID: Dependency Inversion Principle

## Depend on abstractions, not concretions

```python
# HIGH-LEVEL module depends on ABSTRACTION
class OrderService:
    def __init__(self, repository: OrderRepository):  # Interface, not class
        self.repository = repository

    def create_order(self, data):
        order = Order(data)
        return self.repository.save(order)

# Can swap implementations without changing OrderService
sql_repo = SQLOrderRepository()
mongo_repo = MongoOrderRepository()
redis_repo = RedisOrderRepository()

# All work because they implement OrderRepository interface
order_service = OrderService(sql_repo)    # Works
order_service = OrderService(mongo_repo)  # Works
order_service = OrderService(redis_repo)  # Works
```

# ⚖️ Quality Attributes & Trade-offs

**Architecture is fundamentally about trade-offs**

## Key Quality Attributes:

- **Scalability**: Handle increased load
- **Performance**: Response time, throughput
- **Availability**: Uptime percentage
- **Maintainability**: Ease of changes
- **Security**: Protection from threats
- **Testability**: Ease of testing
- **Deployability**: Ease of deployment

**Every decision optimizes some while compromising others**

# Common Trade-offs: CAP Theorem

## Consistency vs. Availability

### You can't have both in distributed systems

### Choose Consistency:

- Banking systems
- Financial transactions
- Inventory management

### Choose Availability:

- Social media feeds
- Content delivery
- Shopping catalogs

## CAP Triangle

```
          Consistency
              /\
             /  \
            /    \
           /      \
          /        \
         /__________\
   Partition     Availability
   Tolerance

Pick 2 of 3:
- CA: Not realistic (networks fail)
- CP: Consistent, might be unavailable
- AP: Available, eventually consistent
```

# Common Trade-offs: Performance vs. Security

**Performance vs. Security:**

- Encryption adds latency
- Authentication adds overhead
- Authorization checks slow responses
- Security auditing impacts performance

**Decision depends on data sensitivity:**

- Medical records → Security wins
- Public blog → Performance wins
- Financial data → Security wins
- Marketing site → Performance wins

# Common Trade-offs: Flexibility vs. Simplicity

## More Flexible

- Plugin architecture
- Configuration-driven
- Multiple databases
- Extensible APIs

**Cost:**

- More complexity
- Harder to understand
- More code to maintain

## More Simple

- Hardcoded choices
- Single database
- Fixed workflow
- Simple APIs

**Cost:**

- Less flexibility
- Changes require code
- Harder to extend

**Guideline:** Start simple, add flexibility when needed (YAGNI)

# Common Trade-offs: Scalability vs. Consistency

```python
# Trade-off example: Caching
class ProductService:
    def get_product(self, product_id):
        # Check cache first (fast but might be stale)
        cached = redis.get(f'product:{product_id}')
        if cached:
            return cached  # FAST but potentially INCONSISTENT

        # Cache miss - hit database (slow but consistent)
        product = db.query("SELECT * FROM products WHERE id = %s", product_id)

        # Cache for 5 minutes
        redis.setex(f'product:{product_id}', 300, product)
        return product

# Trade-off: Performance + Scalability vs. Real-time Consistency
# Acceptable for: Product catalogs (prices don't change every second)
# Not acceptable for: Stock availability (need real-time inventory)
```

# Documenting Decisions: ADRs

## Architectural Decision Records capture the "why"

```
Title: Use Microservices for Order Processing
Status: Accepted
Date: 2024-01-15

Context:
- Monolith difficult to scale
- Order processing has 10x traffic of other components
- Different teams need independence

Decision:
Split order processing into separate microservice

Consequences:
+ Independent scaling of order component
+ Team can deploy without coordinating
- Added network latency for inter-service calls
- Need service discovery and monitoring
- Data consistency becomes harder

Alternatives Considered:
1. Keep monolith, optimize database
2. Use vertical scaling
3. Modular monolith approach
```

# 🏗️ Practical Example: Social Media Platform

**Let's design a scalable system from scratch**

## Business Requirements:

- 100M users
- 500M posts daily
- Real-time feed updates
- Follow/unfollow users
- Like, comment, share posts
- 99.9% availability required

**How do we architect this?**

# Social Media: Service Decomposition

```
MICROSERVICES BY BUSINESS CAPABILITY:

+-- User Service
|   +-- Authentication
|   +-- User profiles
|   +-- Account management
|
+-- Post Service
|   +-- Create, edit, delete posts
|   +-- Post metadata
|
+-- Feed Service
|   +-- Generate personalized feeds
|   +-- Feed ranking algorithm
|
+-- Social Graph Service
|   +-- Follows, connections
|   +-- Relationship queries
|
+-- Engagement Service
|   +-- Likes, comments, shares
|   +-- Engagement counting
|
+-- Media Service
|   +-- Image/video upload
|   +-- Storage management
|
+-- Notification Service
    +-- Real-time alerts
    +-- Push notifications
```

# Social Media: Communication Patterns

## Synchronous (REST)

### For immediate needs:

```
# Get user profile
user = user_service.get_user(
    user_id
)

# Upload image
url = media_service.upload(
    image_data
)
```

**When:** Need immediate response

## Asynchronous (Events)

### For scalability:

```
# Post created
event_bus.publish(
    "post.created",
    post_data
)

# Feed service consumes
# Notification service consumes
# Analytics service consumes
```

**When:** Don't need immediate response

# Social Media: Data Storage Strategy

```
POLYGLOT PERSISTENCE - Right database for right job:

User Service      -> PostgreSQL
                     (ACID for accounts, transactions)

Post Service      -> MongoDB
                     (Flexible schema, document storage)

Feed Service      -> Redis
                     (In-memory, ultra-fast reads)

Social Graph      -> Neo4j
                     (Graph database for relationships)

Media Service     -> Amazon S3 + CloudFront CDN
                     (Object storage for images/videos)

Analytics Service -> Apache Cassandra
                     (Time-series data, write-heavy)

Search Service    -> Elasticsearch
                     (Full-text search)
```

# Social Media: Feed Generation Strategy

## Fan-out on Write

### Push posts immediately

```
User posts ->
  Push to all followers' feeds

Pros:
+ Fast reads
+ Precomputed feeds

Cons:
- High write cost
- Celebrities problem
```

**For:** Regular users

## Fan-out on Read

### Compute on demand

```
User requests feed ->
  Query posts from followed users
  Rank and return

Pros:
+ Low storage
+ Works for celebrities

Cons:
- Slow reads
- High compute
```

**For:** Inactive users, celebrities

**Solution:** Hybrid approach based on user patterns

# Social Media: Scalability Strategy

```python
# Hybrid Feed Generation
class FeedService:
    def generate_feed(self, user_id):
        user = self.user_service.get_user(user_id)

        if user.is_celebrity():
            # Fan-out on read (too many followers)
            return self._compute_feed_on_demand(user_id)

        if user.is_active():
            # Fan-out on write (precomputed)
            return self._get_precomputed_feed(user_id)

        # Inactive users: compute on demand
        return self._compute_feed_on_demand(user_id)

    def _get_precomputed_feed(self, user_id):
        # Fast: just read from Redis
        return redis.lrange(f'feed:{user_id}', 0, 50)

    def _compute_feed_on_demand(self, user_id):
        # Slower: query posts from followed users
        following = self.social_graph.get_following(user_id)
        posts = self.post_service.get_recent_posts(following)
        return self._rank_posts(posts)
```

# Social Media: Availability & Resilience

```
RESILIENCE PATTERNS:

+-- Multi-Availability Zone Deployment
|   +-- Services across 3+ zones
|   +-- Load balancing
|
+-- Circuit Breakers
|   +-- Prevent cascading failures
|   +-- Fail fast, don't wait
|
+-- Graceful Degradation
|   +-- Show cached feeds if real-time unavailable
|   +-- Disable non-critical features
|
+-- Retry with Backoff
|   +-- Retry failed requests
|   +-- Exponential backoff
|
+-- Rate Limiting
|   +-- Prevent overload
|   +-- Per-user limits
|
+-- Database Replication
    +-- Master-slave replication
    +-- Read replicas for scaling
```

# Social Media: Security Architecture

```
SECURITY LAYERS:

+--------------------------+
|      API Gateway         |
| - JWT Authentication     |
| - Rate Limiting          |
| - DDoS Protection        |
| - Request Routing        |
| - SSL/TLS Termination    |
+--------------------------+
            |
            v
+--------------------------+
|   Microservices Layer    |
| - Service-to-service     |
|   authentication (mTLS)  |
| - Input validation       |
| - Authorization checks   |
| - TLS encryption         |
+--------------------------+
            |
            v
+--------------------------+
|      Data Layer          |
| - Encryption at rest     |
| - Access controls        |
| - Audit logging          |
| - Database firewalls     |
+--------------------------+
```

# Social Media: Evolution Path

```
PHASE 1: MVP (0-10K users)
- Monolithic application
- Single server
- PostgreSQL database
- Simple caching

PHASE 2: Growth (10K-1M users)
- Modular monolith
- Separate frontend/backend
- Read replicas
- Redis caching
- CDN for static assets

PHASE 3: Scale (1M-10M users)
- Microservices for high-traffic
- Message queue
- Multiple database types
- Container orchestration (Kubernetes)

PHASE 4: Hypergrowth (10M+ users)
- Full microservices
- Event-driven architecture
- Global CDN
- Multi-region deployment
- Advanced caching strategies
```

# 🚫 Common Pitfalls

## Pitfall #1: Premature Optimization

**Problem:**

Starting with microservices for 100 users

**Example:**

- 3 developers
- 100 users
- Simple CRUD app
- Choose microservices "because Netflix uses them"

**Result:**

- Operational nightmare
- Slow development
- Overwhelmed team

# Pitfall #2: Distributed Monolith

**Problem:**

Microservices that are tightly coupled

**Warning Signs:**

- Deploying one service requires deploying three others
- Shared database between services
- Services calling each other synchronously in chains
- No clear service boundaries

**Result:**

Worst of both worlds:

- Complexity of microservices
- Coupling of monolith

**Solution:**

Clear service boundaries, async communication, independent data stores

# Pitfall #3: Ignoring Conway's Law

**Conway's Law:**

"Organizations design systems that mirror their communication structure"

**Example:**

- Frontend team (10 people)
- Backend team (10 people)
- Database team (5 people)

**Result:** Frontend monolith + Backend monolith + Shared database

**Better:**

Organize teams around business capabilities:

- User Management team (owns full stack)
- Content team (owns full stack)



- Payments team (owns full stack)

# Pitfall #4: No Clear Boundaries

## Bad Service Design:

```
UserService:
- User registration
- Order processing
- Email sending
- Report generation
```

**Too many responsibilities!**

## Good Service Design:

```
UserService: User management only
OrderService: Order processing only
NotificationService: Email/SMS only
ReportService: Analytics only
```

**Clear, focused responsibilities**

# ✅ Best Practices

## Best Practice #1: Start Simple, Evolve

```
EVOLUTION PATH:

Monolith -> Modular Monolith -> Microservices

WHEN TO SPLIT:
+ Performance bottleneck in specific component
+ Team too large for single codebase
+ Different scaling needs for components
+ Clear boundaries established

DON'T SPLIT WHEN:
- Team is small (< 10 developers)
- Requirements are unclear
- User base is small
- No operational complexity needed
```

**Remember:** Premature distribution = premature optimization

# Best Practice #2: Design for Failure

## In distributed systems, failures are normal

```python
# Timeouts
response = requests.get(url, timeout=5)

# Retries with exponential backoff
@retry(tries=3, delay=1, backoff=2)
def call_external_service():
    return requests.get(url)

# Circuit breakers
breaker = CircuitBreaker(failure_threshold=5, timeout=60)
result = breaker.call(risky_operation)

# Graceful degradation
try:
    recommendations = recommendation_service.get_recommendations()
except ServiceUnavailable:
    recommendations = get_cached_recommendations()
    # System still works, just with stale data
```

# Best Practice #3: Observability First

**You can't fix what you can't see**

## Three Pillars of Observability:

**1. Logging:**

- Structured logs (JSON)
- Correlation IDs for tracing
- Centralized aggregation (ELK, Splunk)

**2. Metrics:**

- Response times, error rates
- Resource usage (CPU, memory)
- Business metrics (orders/min)

**3. Tracing:**

- Distributed tracing (Jaeger, Zipkin)

# Best Practice #4: API Versioning

## Services will evolve. Plan for it.

```python
# URL versioning
@app.route('/api/v1/users')
def get_users_v1():
    return jsonify(users)

@app.route('/api/v2/users')
def get_users_v2():
    # Enhanced with pagination
    return jsonify({
        'users': users,
        'page': page,
        'total': total
    })

# Deprecation strategy:
# v1: Current (supported 12 months)
# v2: New (encouraged)
# v3: Beta (testing)
# After 6 months: Announce v1 deprecation
# After 12 months: Remove v1
```

# Best Practice #5: Document Architecture

## Use C4 Model for clarity:

```
LEVEL 1: System Context
+--------------------------+
|   Social Media System    |
+--------------------------+
    |           |
  Users    Mobile Apps

LEVEL 2: Containers
+--------+  +--------+  +---------+
|  Web   |  |  API   |  | Mobile |
|  App   |  |Gateway|  |  App    |
+--------+  +--------+  +---------+

LEVEL 3: Components
API Gateway:
+-- Authentication
+-- Rate Limiting
+-- Routing
+-- Monitoring

LEVEL 4: Code
(Class diagrams, detailed design)
```

# 📝 Practice Quiz #1

## Healthcare Scenario:

Medical device software with FDA regulations. Requirements defined by regulatory standards. Extensive documentation needed for approval. Safety-critical with thorough testing before release.

## Question:

Which architectural approach is MOST appropriate?

A) Microservices with continuous deployment
B) Event-driven architecture with message queues
C) Layered monolithic with comprehensive documentation
D) Serverless architecture with AWS Lambda

**Think before next slide...**

# Quiz #1: Answer

**Answer: C) Layered monolithic with comprehensive documentation**

## Why Layered Monolith:

✅ **Stable requirements** (defined by FDA)
✅ **Extensive documentation** (regulatory requirement)
✅ **Fixed compliance standards**
✅ **Safety-critical** (thorough testing before release)
✅ **Full audit trail**

## Why Not Others:

❌ **Microservices:** Insufficient documentation, incremental releases not allowed
❌ **Event-driven:** Eventual consistency not acceptable for medical devices
❌ **Serverless:** Execution limits, cold starts, lack of control

# 📝 Practice Quiz #2

## E-Commerce Scenario:

Product catalog shows prices from inventory database. Every page load queries database directly (2-3 seconds during peak). Considering Redis cache with 5-minute TTL.

## Question:

What trade-off are you making?

A) Trading security for performance
B) Trading consistency for performance
C) Trading availability for scalability
D) Trading maintainability for flexibility

**Think before next slide...**

# Quiz #2: Answer

**Answer: B) Trading consistency for performance**

## Explanation:

**Consistency Sacrifice:**

- Prices might be up to 5 minutes stale
- Database shows $89, cache shows old $99
- Users see inconsistent data temporarily

**Performance Gain:**

- Page loads: 2-3 seconds → milliseconds
- Database load dramatically reduced
- Can handle much more traffic

## When Acceptable:



✅ Product catalog prices (change infrequently)

# 📝 Practice Quiz #3

## Scaling Scenario:

Which provides STRONGEST justification for microservices over monolith?

A) Team has 3 developers, needs to launch quickly

B) Different parts have vastly different scaling needs (1000x traffic difference)

C) Want to use multiple programming languages

D) System needs to be highly available

**Think before next slide...**

# Quiz #3: Answer

**Answer: B) Different parts have vastly different scaling needs**

## Why This Justifies Microservices:

**Example:** E-commerce platform

- **Search Service:** 100,000 requests/min (users browsing)
- **Checkout Service:** 100 requests/min (only 0.1% purchase)

**With Monolith:**

- Must scale entire app for search traffic
- Waste resources on rarely-used checkout

**With Microservices:**

- Deploy 50 search instances
- Deploy 2 checkout instances
- Massive cost savings

# 📝 Practice Quiz #4

## Code Structure Scenario:

Which BEST demonstrates separation of concerns?

A) Single function: validates input, queries database, performs calculations, renders HTML

B) Controller handles HTTP, Service contains business logic, Repository handles data

C) All database queries in controller, business logic spread across layers

D) Business logic in view templates, controllers only route requests

**Think before next slide...**

# Quiz #4: Answer

**Answer: B) Controller handles HTTP, Service contains business logic, Repository handles data**

## Why This Is Correct:

**Controller Layer:**

- HTTP concerns only
- Parse requests, format responses
- No business logic or database queries

**Service Layer:**

- Business logic only
- Calculations, validations, workflows
- No HTTP or SQL knowledge

**Repository Layer:**

- Data access only

# 📝 Practice Quiz #5

## Event-Driven Scenario:

Order Service publishes "order.created" event. Three subscribers: Inventory (reserves stock), Payment (charges customer), Notification (sends email). Payment service fails due to outage.

## Question:

What happens?

A) Entire order creation fails and rolls back
B) Inventory and Notification succeed; Payment processes when recovered
C) All three services fail (tightly coupled)
D) System enters unrecoverable inconsistent state

**Think before next slide...**

# Quiz #5: Answer

**Answer: B) Inventory and Notification succeed; Payment processes when recovered**

## How Event-Driven Works:

1. **Order Service publishes** event to message broker
2. **Message broker stores** event persistently
3. **Inventory subscribes**, receives immediately, reserves stock ✅
4. **Notification subscribes**, receives immediately, sends email ✅
5. **Payment is down**, broker keeps event in Payment's queue
6. **When Payment recovers**, processes all queued events ✅

## Key Benefits:

✅ **Resilience:** One service failure doesn't cascade
✅ **Decoupling:** Services don't know about each other
✅ **Eventual Consistency:** System becomes consistent eventually



## This Is NOT A Bug:

# 🎯 Key Takeaways

**Remember Forever:**

1. **Architecture ≠ Design**: Architecture = system structure, Design = component implementation
2. **Principles guide decisions**: SoC, modularity, abstraction, loose coupling
3. **Patterns solve specific problems**: Layered, MVC, Microservices, Event-Driven
4. **Trade-offs are inevitable**: Optimize some attributes, compromise others
5. **SOLID scales to systems**: Not just classes, but entire architectures
6. **Start simple, evolve**: Monolith → Modular Monolith → Microservices
7. **Document decisions**: Use ADRs to capture context and rationale

# Universal Success Factors

**Regardless of architecture chosen:**

✓ **Clear communication** between team and stakeholders
✓ **Skilled team** competent in chosen approach
✓ **Committed stakeholders** engaged appropriately
✓ **Quality focus** on testing and code quality
✓ **Realistic planning** with honest estimates
✓ **Risk management** identifying and mitigating risks
✓ **Continuous learning** from retrospectives
✓ **Leadership support** with executive sponsorship

**Bottom Line:** Well-executed architecture > poorly-executed "best practice"

# Action Items

## This Week:

1. Assess your current project's architecture
2. Identify one architectural principle being violated
3. Propose one specific improvement
4. Share learning with your team

## This Month:

- Study one architectural pattern in depth
- Read one book from references
- Review architectural decisions in your codebase
- Hold architecture retrospective with team

# Resources for Continued Learning

**Essential Reading:**

- **Domain-Driven Design** by Eric Evans
- **Building Microservices** by Sam Newman
- **Clean Architecture** by Robert Martin
- **Software Architecture in Practice** by Bass, Clements, Kazman
- **Fundamentals of Software Architecture** by Richards and Ford

**Online Resources:**

- Martin Fowler's Blog (martinfowler.com)
- Microsoft Architecture Center
- AWS Well-Architected Framework
- C4 Model (c4model.com)

# Architecture Decision Framework

```
WHEN CHOOSING ARCHITECTURE:

1. Assess Requirements
   - Stability (stable → Waterfall, evolving → flexible)
   - Scale (small → monolith, large → distributed)
   - Team size (small → simple, large → microservices)

2. Identify Constraints
   - Budget, timeline, team skills
   - Regulatory, compliance requirements
   - Technology limitations

3. Evaluate Trade-offs
   - What are you optimizing for?
   - What are you willing to sacrifice?
   - Document in ADRs

4. Start Simple
   - Don't prematurely distribute
   - Establish clear boundaries
   - Split when scaling demands

5. Measure and Adapt
   - Monitor quality attributes
   - Gather feedback
   - Evolve architecture based on data
```

# Final Thoughts

### Software Architecture Is:

- About trade-offs, not perfection
- Context-dependent, not one-size-fits-all
- Evolutionary, not set in stone
- Balancing multiple concerns

### Success Comes From:

- Understanding principles deeply
- Matching architecture to context
- Documenting decisions and rationale
- Evolving based on feedback
- Prioritizing quality attributes
- Building for today, designing for tomorrow

# Thank You! 🚀

## Questions?

**Key Message:**

Architecture is about making informed trade-offs that balance competing quality attributes while keeping your system maintainable, scalable, and aligned with business goals.

**Choose wisely, document thoroughly, evolve thoughtfully.**

# Lesson Complete

## You Are Now Equipped To:

✓ Distinguish architecture from design
✓ Apply fundamental principles (SoC, modularity, abstraction)
✓ Choose appropriate architectural patterns
✓ Evaluate trade-offs between quality attributes
✓ Apply SOLID principles at system level
✓ Design for failure and resilience
✓ Document architectural decisions
✓ Evolve architecture thoughtfully

**Now go build amazing systems!**

**Keep learning, keep building, keep improving!**