Architectural Patterns in Python

Table of Contents

Bîcu Andrei Ovidiu

What Are Architectural Patterns?

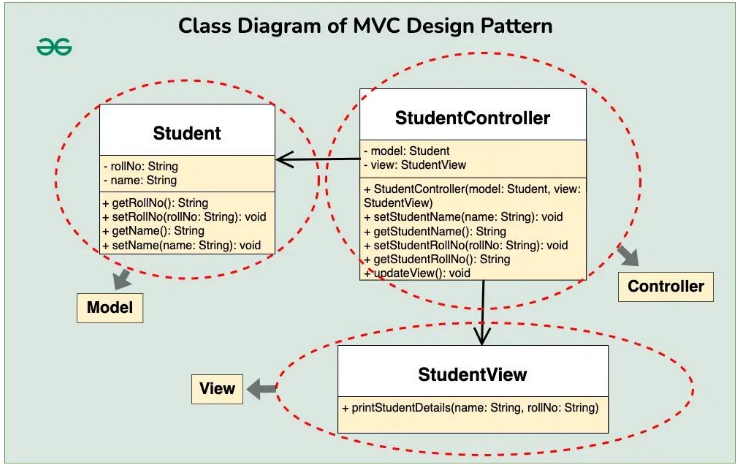
*Architectural patterns are **reusable solutions** to common high-level problems in software architecture. They guide the **organization of software systems**, defining responsibilities, communication, and relationships between components. Unlike design patterns (which address smaller-scale code structure), architectural patterns deal with the **blueprint** of an application. How it's structured, scaled, and maintained.*

Architectural patterns help you:

- Define how different parts of a system **interact and communicate**
- Support **scalability, maintainability, and testability.**
- Offer guidance on structuring both **monoliths** and **distributed systems**
- Make decisions early in the development lifecycle that affect **deployment, performance, and modularity**

Used across different contexts:

- **Web applications** (e.g., MVC, Microservices)
- **Enterprise systems** (e.g., Repository, Service Layer)
- **Data pipelines** (e.g., CQRS, Publish-Subscribe)



Pattern Categories Overview

*Architectural and design patterns are commonly grouped by the **type of problem they solve**. In this course, we focus on four main categories:*

1. ***Creational***
2. ***Structural***
3. ***Behavioral***
4. ***Modern architectural patterns*** *(used in large-scale and enterprise systems)*

Usage

- Grouping patterns helps developers **choose the right approach** depending on the phase or concern in their application.
- These categories **overlap in real-world projects**, but knowing the intent of each helps maintain clean, testable, and extensible code.

```
----------------------------
|   Pattern Categories     |
----------------------------
|  Creational | Singleton, Factory, Abstract Factory, Builder   |
|  Structural | Adapter, Composite                              |
|  Behavioral | Observer                                        |
|  Modern     | MVC, Repository, CQRS, Microservices, etc.      |
-----------------------------

# Tree Diagram for Visual Learners

Patterns
├── Creational
│   ├── Singleton
│   ├── Factory
│   ├── Abstract Factory
│   └── Builder
├── Structural
│   ├── Adapter
│   └── Composite
├── Behavioral
│   └── Observer
└── Modern
    ├── MVC
    ├── Repository
    ├── CQRS
    └── ...
```

Creational - Singleton Pattern

*The **singleton pattern** ensures that a class has **only one instance** and provides a global point of access to it. It’s used when exactly **one object** is needed to coordinate actions across a system (e.g., configuration, logging, database connection).*

Usage:

Use the Singleton pattern when:

- You need a **shared resource** or service (e.g., logger or a config loader)
- You want to **restrict instantiation** to one object
- You want to **avoid global variables**, but still allow centralized access.

Be cautious:

- Can lead to **hidden dependencies**
- Often **makes testing harder** if not used carefully
- Can be overused as a quick fix for state sharing.

```python
# Define a metaclass that controls the instantiation process
class SingletonMeta(type):
    _instance = None  # This will store the single instance of the class

    def __call__(cls, *args, **kwargs):
        # __call__ is triggered when you instantiate a class
        if cls._instance is None:
            # If no instance exists, create one
            cls._instance = super().__call__(*args, **kwargs)
        # Return the same instance every time
        return cls._instance

# Apply the SingletonMeta metaclass to your class
class Config(metaclass=SingletonMeta):
    def __init__(self):
        # You can initialize configuration values or any shared state here
        self.settings = {
            "db": "localhost",
            "port": 5432
        }

# Instantiate the class multiple times
c1 = Config()
c2 = Config()

# Check if both instances are actually the same object
print(c1 is c2)  # Output: True — proves Singleton behavior

# You can access the same shared state from both variables
print(c1.settings)  # {'db': 'localhost', 'port': 5432'}
print(c2.settings)  # {'db': 'localhost', 'port': 5432'}
```

Creational - Factory Pattern

*The **Factory Pattern** is a creational design pattern that provides an **interface for creating objects**, allowing subclasses or methods to decide which class to instantiate. It helps in **decoupling object creation logic** from the business logic, making the code more flexible and easier to maintain or extend.*

Use the Factory pattern when:

- You need to **create objects dynamically** without knowing their concrete classes ahead of time
- You want to **abstract away the instantiation process**
- You aim to **follow the Open/Close Principle** (open for extension, closed for modification)

Common in:

- GUI toolkits (buttons, menus)
- Parsing libraries (different document types)
- Data processing pipelines

```python
from abc import ABC, abstractmethod

# Define an abstract product
class Notification(ABC):
    @abstractmethod
    def notify(self, message: str) -> None:
        """All notification types must implement this method."""
        pass


# Create concrete products
class EmailNotification(Notification):
    def notify(self, message):
        # Implementation specific to email notification
        print(f"Sending Email: {message}")

class SMSNotification(Notification):
    def notify(self, message):
        # Implementation specific to SMS notifiation
        print(f"Sending SMS: {message}")


# Now we define the Factory function

def notification_factory(type_: str) -> Notification:
    """
    Factory method to create instances of Notification subclasses
    based on the input type.
    """
    if type_ == "email":
        return EmailNotification()
    elif type_ == "sms":
        return SMSNotification()
    else:
        raise ValueError(f"Unknown notification type: {type_}")

# Use the factory to create instances without knowing their class
notifier = notification_factory("email") # Creates EmailNotification
notifier.notify("Hello, you've got mail!")

notifier2 = notification_factory("sms") # Creates SMSNotification
notifier2.notify("Your OTP is 123456")
```

Creational - Abstract Factory Pattern

*The **Abstract Factory Pattern** is a creational pattern that provides an **interface for creating families of related or dependent objects** without specifying their concrete classes. Unlike the Factory Pattern (which returns a single product), Abstract Factory **groups multiple factories** under a unified interface, promoting consistency among related objects.*

Use Abstract Factory when:

- You need to create **related objects** (e.g., UI components for Windows vs macOS)
- You want to enforce **product compatibility**
- You need to support **multiple themes, strategies, or platforms**.

Typical use cases:

- Cross-platform GUI toolkits
- Plugin systems
- Game engines (e.g., different worlds or unit families)

```python
# Abstract products
class Button(ABC):
    @abstractmethod
    def render(self) -> None:
        """Render the button on the screen."""
        pass

class Checkbox(ABC):
    @abstractmethod
    def check(self) -> None:
        """Check or uncheck the checkbox."""
        pass

# Concrete products - Windows style
class WindowsButton(Button):
    def render(self) -> None:
        print("Render a Windows-syle button.")

class WindowsCheckbox(Checkbox):
    def check(self) -> None:
        print("Check a Windows-style Checkbox.")

# Concrete products - macOS style
class MacButton(Button):
    def render(self) -> None:
        print("Render a Mac-style button.")

class MacCheckbox(Checkbox):
    def check(self) -> None:
        print("Check a Mac-style Checkbox")

# Abstract Factory
class GUIFactory(ABC):
    @abstractmethod
    def create_button(self) -> Button:
        pass

    @abstractmethod
    def create_checkbox(self) -> Checkbox:
        pass

# Concrete factories
class WindowsFactory(GUIFactory):
    def create_button(self) -> Button:
        return WindowsButton()
    def create_checkbox(self) -> Checkbox:
        return WindowsCheckbox()

class MacFactory(GUIFactory):
    def create_button(self) -> Button:
        return MacButton()
    def create_checkbox(self) -> Checkbox:
        return MacCheckbox()

# Client code - works with any factory
def create_ui(factory: GUIFactory) -> None:
    button = factory.create_button()
    checkbox = factory.create_checkbox()

    button.render() # Renders platform specific button
    checkbox.check() # Checks platform-specific checkbox

# Usage
factory =  WindowsFactory()
create_ui(factory)
factory = MacFactory()
create_ui(factory)
```

Creational - Builder Pattern

*The **Builder Pattern** is a creational design pattern that allows you to **construct complex objects step by step.** It separates the construction of an object from its representation, so the same construction process can create different representations. Unlike telescoping constructors or long parameter lists, the Builder pattern makes the construction **explicit and readable**.*

Use the Builder Pattern when:

- You're dealing with an object that requires **many optional parameters**
- You want to **build complex objects progressively**
- You need to **reuse the same building process** for different configurations

Common examples:

- Creating configurations or reports
- Constructing data transfer objects (DTOs)
- Fluent interfaces for object creation

```python
from typing import Optional

# A product class representing a complex object
class User:
    def __init__(self, username: str, email: str, age: Optional[int] = None, address: Optional[str] = None):
        self.username = username
        self.email = email
        self.age = age
        self.address = address

    def __str__(self):
        return f"User(username={self.username}, email={self.email}, age={self.age}, address={self.address})"

# The Builder class
class UserBuilder:
    def __init__(self, username: str, email: str):
        # Requred parameters
        self._username = username
        self._email = email
        # Optional parameters with default None
        self._age = None
        self._address = None

        # Fluent setter methods
    def with_age(self, age: int) -> "UserBuilder":
        self._age = age
        return self # Retun self to allow method chaining

    def with_address(self, address: str) -> "UserBuilder":
        self._address = address
        return self

    def build(self) -> User:
        # Final step: construct the User object using all gathered data
        return User(
                username = self._username,
                email = self._email,
                age = self._age,
                address = self._address
            )

# Example usage
builder = UserBuilder("johndoe", "john@example.com")
user = builder.with_age(30).with_address("123 Python Street").build()

print(user) # Output: User(username=johndoe, email=john@example.com, age=30, address=123 Python Street)
```

Structural - Adapter Pattern

*The **Adapter Pattern** is a structural design pattern that allows objects with **incompatible interfaces to work together.** It acts as a **wrapper**, translating one interface into another. This is especially useful when **integrating third-party code,** legacy systems, or APIs that you can't modify.*

Use the Adapter Pattern when:

- You want to **reuse existing classes** with incompatible interfaces
- You're **integrating external libraries or legacy systems**
- You want to follow the **Open/Closed Principle** - adapt without modifying the original code

Examples:

- Wrapping a REST API response to match your internal model
- Adapting old classes to a new interface
- Creating database or API clients that conform to your service layer's interface

```python
# Existing class with an incompatible interface
class OldPrinter:
    def old_print(self, text: str) -> None:
        print(f"[OldPrinter] {text}")

# New interface expected by the system
class Printer:
    def prnt(self, text: str) -> None:
        raise NotImplementedError

# Adapter that makes OldPrinter compatible with Printer
class PrinterAdapter(Printer):
    def __init__(self, adaptee: OldPrinter):
        self._adaptee = adaptee

    def prnt(self, text: str) -> None:
        # Translate the call from the new interface to the old one
        self._adaptee.old_print(text)

# Client code that works with the new interface
def client_code(printer: Printer):
    printer.prnt("Hello from the new system!")

# Example usage
legacy_printer = OldPrinter()
adapter = PrinterAdapter(legacy_printer)
client_code(adapter) # Internally calls old_print()
```

Structural - Composite Pattern

*The **Composite Pattern** is a structural design pattern that lets you **compose objects into tree structures** and work with them as if they were individual objects. It treats both **individual objects** and **groups of objects** uniformly, making it easier to deal with **hierarchical data** like UI elements, file systems, or organizational charts.*

Use the Composite Pattern when:

- You want to represent **part-whole hierarchies**
- You need to treat **individual objects and groups uniformly**
- You work with tree-like structures (e.g., nested menus, folders/files)

Common use cases:

- UI toolkits (e.g., containers and widgets)
- File systems (folder containing files and other folders)
- Company structure (employees and departments)

```python
from abc import ABC, abstractmethod

# Base component class
class FileSystemItem(ABC):
    @abstractmethod
    def show(self, indent: int = 0) -> None:
        """Display the item with indentation base on hierarchy level."""
        pass

# Leaf class - cannot contain other items
class File(FileSystemItem):
    def __init__(self, name: str):
        self.name = name

    def show(self, indent: int = 0) -> None:
        print("  " * indent + f"File: {self.name}")

# Composit class - can contain children
class Folder(FileSystemItem):
    def __init__(self, name: str):
        self.name = name
        self._children: list[FileSystemItem] = []

    def add(self, item: FileSystemItem) -> None:
        """Add a file or folder to this folder."""
        self._children.append(item)

    def show(self, indent: int = 0) -> None:
        print("  " * indent + f"Folder: {self.name}")
        for child in self._children:
            child.show(indent + 1) # Recursively show children with more indentation

# Usage
root = Folder("root")
root.add(File("file1.txt"))
root.add(File("file2.txt"))

sub_folder = Folder("subfolder")
sub_folder.add(File("nested_file.txt"))

root.add(sub_folder)

# Display the whole file structure
root.show()
```

Behavioral - Observer Pattern

*The **Observer Pattern** is a behavioral design pattern that defines a **one-to-many dependency** between objects so that when one object changes state, all its dependents (observers) are **automatically notified.** It promotes **loose coupling** between subjects and observers, making systems more extensible and reactive.*

Use the Observer Pattern when:

- One object (the subject) needs to **notify multiple others** (observers) about changes
- You want to build **event-driven systems** (e.g., GUI events, real-time updates)
- You need to **decouple publishers from subscribers**

Common scenarios:

- GUI frameworks (button click events)
- Event buses / messaging systems
- Real-time apps (e.g., stock tickers, chat, monitoring)

```python
from abc import ABC, abstractmethod

# Observer interface
class Observer(ABC):
    @abstractmethod
    def update(self, data: str) -> None:
        """Receive update from the subjects."""
        pass

# Concrete observer
class EmailSubscriber(Observer):
    def __init__(self, email: str):
        self.email = email

    def update(self, data: str) -> None:
        print(f"Email sent to {self.email}: {data}")

# Another concrete observer
class SMSSubscriber(Observer):
    def __init__(self, phone: str):
        self.phone = phone

    def update(self, data: str) -> None:
        print(f"SMS sent to {self.phone}: {data}")

# Subject class that notifies observers
class NewsPublisher:
    def __init__(self):
        self._observers: list[Observer] = []

    def subscribe(self, observer: Observer) -> None:
        self._observers.append(observer)

    def unsuscribe(self, observer: Observer) -> None:
        self._observers.remove(observer)

    def notify(self, news: str) -> None:
        for observer in self._observers:
            observer.update(news)

# Example usage
publisher = NewsPublisher()

email_sub = EmailSubscriber("user@example.com")
sms_sub = SMSSubscriber("+123456789")

publisher.subscribe(email_sub)
publisher.subscribe(sms_sub)

# Notify all subscribers
publisher.notify("New article: Observer Pattern in Python")
```

MVC (Model-View-Controller)

Model-View-Controller (MVC) *is a software architectural pattern that separates an application into three main components:*

1. ***Model -*** *Manages the data and business logic*
2. ***View*** *- Handles the UI and presentation*
3. ***Controller*** *- Handles user input and coordinates between Model and View*

This separation improves ***modularity, testability, and scalability.***

Use MVC when:

- You want to **separate concerns** (UI, business logic, data)
- You're building **interactive applications** (like web UIs or desktop GUIs)
- You want to easily **test and maintain** different parts of the app

Commonly used in:

- Web frameworks (e.g., Django, Flask, ASP.NET)
- GUI frameworks (e.g., Tkinter, PyQt)
- Mobile and desktop applications

```python
# --- Model ---
class UserModel:
    def __init__(self, name: str):
        self.name = name
    def get_user_data(self) -> str:
        return f"User: {self.name}"

# --- View ---
class UserView:
    def display_user(self, user_info: str) -> None:
        print(f"[View] {user_info}")

# --- Controller ---
class UserController:
    def __init__(self, model: UserModel, view: UserView)
        self._model = model
        self._view = view

    def update_view(self) -> None:
        # Get data from model
        user_data = self._model.get_user_data()
        # Pass data to view
        self._view.display_user(user_data)

# Example usage
model = UserModel("Alice")
view = UserView()
controller = UserController(model, view)

# Controller acts as a bridge between Model and View
controller.update_view()
```

Repository Pattern

*The **Repository Pattern** is a structural pattern that acts as a **mediator between the domain and data mapping layers**, using a collection-like interface for accessing domain objects. It abstracts away **data persistence**, so your business logic doesn't need to know how data is stored (e.g., in a database, API, or in-memory).*

Use the Repository Pattern when:

- You want to **decouple business logic from data access logic**
- You want to **mock or fake your data layer** for testing
- You need to **encapsulate query logic** and **centralize access**

Commonly used in:

- Domain-driven design (DDD)
- Clean architecture
- Services that access database or external APIs

```python
from typing import List

# --- Domain Entity ---
class User:
    def __init__(self, user_id: int, name: str):
        self.user_id = user_id
        self.name = name

# --- Repository Interface ---
class UserRepository:
    def __init__(self):
        # In-memory database simulation
        self._users: dict[int, User] = {}

    def add(self, user: User) -> None:
        """Add a user to the repository."""
        self._users[user.user_id] = user

    def get_by_id(self, user_id: int) -> User:
        """Retrieve a user by ID. Raises ValueError if not found."""
        user = self._users.get(user_id)
        if user is None:
            raise ValueError(f"User with ID {user_id} not found")
        return user

    def list_all(self) -> List[User]:
        """Return all users"""
        return list(self._users.values())


# --- Business Logic / Service Layer ---
class UserService:
    def __init__(self, repository: UserRepository):
        self._repository = repository

    def register_user(self, user_id: int, name: str) -> None:
        user = User(user_id, name)
        self._repository.add(user)

    def show_users(self) -> None:
        for user in self._repository.list_all():
            print(f"{user.user_id}: {user.name}")

# Example usage
repo = UserRepository()
service = UserService(repo)

service.register_user(1, "Alice")
service.register_user(2, "Bob")

service.show_users() # Output: 1: Alice | 2: Bob
```

Service Layer Pattern

*The **Service Layer Pattern** defines a layer that **encapsulates business logic** and **coordinates domain operations**. It acts as an intermediary between controllers (or user interfaces) and the domain or persistence layers (like repositories). Its main purpose is to **organize complex business rules** in a reusable and testable way.*

Use the Service Layer pattern when:

- Your application has **non-trivial business logic**
- You want to **decouple controllers/UI from domain logic**
- You want to **orchestrate multiple repository operations**
- You want **thin controllers** and **testable business logic**

Common in:

- Domain-driven design (DDD)
- Layered or hexagonal architectures
- Web APIs, backends, microservices

```python
from typing import Optional

# --- Domain Entity ---
class User:
    def __init__(self, user_id: int, name: str):
        self.user_id = user_id
        self.name = name

# --- Repository Layer (Persistence) ---
class UserRepository:
    def __init__(self):
        self._users: dict[int, User] = {}

    def add(self, user: User) -> None:
        self._users[user.user_id] = user

    def get_by_id(self, user_id: int) -> Optional[User]:
        """Return user or None if not found."""
        return self._users.get(user_id)

# --- Service Layer (Business Logic) ---
class UserService:
    def __init__(self, repository: UserRepository):
        self._repository = repository

    def register_user(self, user_id: int, name: str) -> bool:
        """Register a new user if they don't already exist."""
        if self._repository.get_by_id(user_id):
            print(f"[Service] User ID {user_id} already exists.")
            return False

        user = User(user_id, name)
        self._repository.add(user)
        print(f"[Service] User '{name}' registered successfully.")
        return True

    def get_user_summary(self, user_id: int) -> str:
        """Fetch user info in a business-friendly format."""
        user = self._repository.get_by_id(user_id)
        if not user:
            return "User not found."
        return f"User: {user.name} (ID: {user.user_id})"

# --- Application Logic (e.g., Controller or CLI) ---
repo = UserRepository()
service = UserService(repo)

service.register_user(1, "Alice")
service.register_user(1, "Alice") # Should fail
print(service.get_user_summary(1))
print(service.get_user_summary(42))  # Should return not found.
```

CQRS (Command Query Responsibility Segregation)

CQRS *is a pattern that* ***separates reading (queries)*** *from* ***writing (commands)*** *in a system. Instead of using the same model for both, you split them into:*

1. ***Command model****: Handles mutations (create, update, delete)*
2. ***Query model****: Handles reads (fetching data)*

This enables better scalability, optimization, and clearer design.

Use CQRS when:

- Read and write workloads have **very different performance requirements**
- You want to **optimize read models** without impacting writes
- You want to apply **event sourcing** or **audit logging**
- You system has **complex domain logic** or **write validation**

Often used in:

- Event-driven architectures
- Microservices
- High-performance or high-scale systems

```python
from typing import Optional

# --- Domain Entity ---
class User:
    def __init__(self, user_id: int, name: str):
        self.user_id = user_id
        self.name = name

# --- Command Side (Write Model) ---
class UserCommandHandler:
    def __init__(self, store: dict[int, User]):
        self._store = store

    def create_user(self, user_id: int, name: str) -> bool:
        """Handles user creation command."""
        if user_id in self._store:
            print("[Command] User already exists.")

        self._store[user_id] = User(user_id, name)
        print(f"[Command] User '{name}' created.")
        return True

    def update_user(self, user_id: int, name: str) -> bool:
        """Handles user update command."""
        user = self._store.get(user_id)
        if not user:
            print("[Command] User not found.")
            return False

        user.name = name
        print(f"[Command] User {user_id} updated.")
        return True

# --- Query Side (Read Model) ---
class UserQueryHandler:
    def __init__(self, store: dict[int, User]):
        self._store = store

    def get_user_name(self, user_id: int) -> Optional[str]:
        """Handles read query."""
        user = self._store.get(user_id)
        return user.name if user else None

    def list_users(self) -> list[str]:
        """List all user names."""
        return [user.name for user in self._store.values()]

# Example usage
# Shared in-memory store (can be separated in real apps)
data_store: dict[int, User] = {}

# Instantiate command/query handlers
cmd = UserCommandHandler(data_store)
qry = UserQueryHandler(data_store)

cmd.create_user(1, "Alice")
cmd.create_user(2, "Bob")
cmd.create_user(3, "Alice Smith")

print("[Query] All users:", qry.list_users()) # Output: [Query] All users: ['Alice', 'Bob', 'Alice Smith']
print("[Query] User 1:", qry.get_user_name(1)) # Output: [Query] User 1: Alice
```

Dependency Injection (DI) Pattern

***Dependency Injection** is a design pattern where **an object receives its dependencies** (services, repositories, etc.) **from external sources** rather than creating them itself. This promotes:*

1. ***Loose coupling***
2. ***Better testability***
3. ***Easier maintenance***

Use Dependency Injection when:

- You want to **decouple components** from their concrete implementations
- You need to **swap implementations easily** (e.g., for testing or scaling)
- You want to **follow SOLID principles,** especially the *Dependency Inversion Principle*

Common in:

- Clean Architecture
- Unit testing
- Large, modular applications
- Frameworks like FastAPI (via function injection), Django, etc.

```python
# --- Service Interface ---
class NotificationService:
    def send(self, recipient: str, message: str) -> None:
        """"Send a message to a recipient."""
        raise NotImplementedError

# --- Concrete Implementation ---
class EmailNotificationService(NotificationService):
    def send(self, recipient: str, message: str) -> None:
        print(f"[Email] To: {recipient} | Message: {message}")

# --- Business Logic (dependent on NotificationService) ---
class UserRegistrationService:
    def __init__(self, notifier: NotificationService):
        """
        The notifier is injected as a dependency.
        We don't care *how* messages are sent.
        """
        self._notifier = notifier

    def register_user(self, name: str, email: str) -> None:
        print(f"[Service] Registering user: {name}")
        # Send a welcome notification via the injected notifier
        self._notifier.send(email, f"Welcome, {name}!")

# --- Application Setup (Injection) ---
notifier = EmailNotificationService() # could later swap this with SMS, mock, etc.
registration_service = UserRegistrationService(notifier)

registration_service.register_user("Alice", "alice@example.com")
```

Microservices Architecture

Microservices Architectures *is an architectural style where an application is composed of* ***small, independent services,*** *each focused on a* ***specific business capability****. Each service:*

1. *Has its own codebase and database*
2. *Communicates with others via* ***APIs*** *(typically HTTP or messaging)*
3. *Can be deployed, scaled, and updated* ***independently***

Use microservices when:

- Your system is **large or complex enough** to be split into business domains
- You want **independent scaling and deployment**
- Your teams work on **separate functionalities**
- You need **fault isolation** and **resilience**

Common in:

- Distributed systems
- Cloud-native architecture (Kubernetes, Docker)
- Scalable enterprise backends

```python
# --- User Service ---
class UserService:
    def get_user(self, user_id: int) -> dict:
        """Simulates fetching user data from the User microservice."""
        print(f"[UserService] Getting userr info...")
        return {"id": user_id, "name": "Alice"}

# --- Order Service ---
class OrderService:
    def create_order(self, user_id: int, product: str) -> str:
        """Simulates creating an order in the Order microservice."""
        print(f"[OrderService] Creating order for user {user_id} - Product: {product}")
        return f"Order for {product} created for user {user_id}"

# --- API Gateway or Orchestrator ---
class Orchestrator:
    def __init__(self, user_service: UserService, order_service: OrderService):
        self.user_service = user_service
        self.order_service = order_service

    def place_order(self, user_id: int, product: str) -> None:
        """Simulates API gateway or orchestrator coordinating services."""
        user = self.user_service.get_user(user_id)
        if user:
            result = self.order_service.create_order(user_id, product)
            print(f"[Gateway] Success: {result}")
        else:
            print("[Gateway] User not found.")

# --- Simulated Microservices Communication ---
user_service = UserService()
order_service = OrderService()
gateway = Orchestrator(user_service, order_service)

gateway.place_order(1, "Book")
'''Output:
[UserService] Getting userr info...
[OrderService] Creating order for user 1 - Product: Book
[Gateway] Success: Order for Book created for user 1'''
```

Publish-Subscribe (Pub/Sub) Pattern

*The **Publish-Subscribe** pattern is a messaging architecture where **senders** (**publishers**) broadcast messages without knowing who will receive them, and **receivers (subscribers)** react to events they are interested in. Key features:*

1. ***Loose coupling**: Publishers and subscribers are unaware of each other*
2. ***Asynchronous communication***
3. ***Decouples producers from consumers***

Use Pub/Sub when:

- You want to **reactive behavior** across services or modules
- Events need to **trigger multiple side-effects**
- You need **scalable, event-driven communication**
- You want to log, monitor, or audit changes without tightly coupling services

Common in:

- Microservices
- Real-time applications (chat, notifications)
- Distributed systems
- Logging/event sourcing pipelines

```python
# --- Event Bus ---
class EventBus:
    def __init__(self):
        """
        Initializes the EventBus.
        _subscribers is a dictionary where each key is an event type (e.g., "order_placed")
        and the value is a list of handler functions subscribed to that event.
        defaultdict(list) ensures new keys are initialized with empty lists automatically.
        """
        self._subscribers: dict[str, list[Callable[[Any], None]]] = defaultdict(list)

    def subscribe(self, event_type: str, handler: Callable[[Any], None]) -> None:
        """
        Subscribes a handler function to a specific event type.

        Parameters:
        - event_type (str): The name of the event (e.g., "order_placed").
        - handler (Callable): A function that will be called with event data when the event is published.

        This allows multiple subscribers to respond to the same event independently.
        """
        self._subscribers[event_type].append(handler)

    def publish(self, event_type: str, data: Any) -> None:
        """
        Publishes an event to all subscribed handler functions.

        Parameters:
        - event_type (str): The name of the event being triggered.
        - data (Any): The data payload associated with the event, passed to each subscriber.

        It loops through all handlers for the given event type and calls each with the event data.
        """
        print(f"[EventBus] Publishing event: {event_type}")
        for handler in self._subscribers[event_type]:
            handler(data)

# --- Subscribers (Event Handlers) ---
def log_event(data):
    """
    A subscriber that logs the entire event data.

    Example:
    Input: {"user": "Alice", "item": "Laptop"}
    Output: [Logger] Event received: {'user': 'Alice', 'item': 'Laptop'}
    """
    print(f"[Logger] Event received: {data}")

def notify_user(data):
    """
    A subscriber that simulates notifying a user.

    Example:
    Input: {"user": "Alice", "item": "Laptop"}
    Output: [Notification] Notifying user: Alice
    """
    print(f"[Notification] Notifying user: {data['user']}")

# --- Simulating Pub/Sub Usage ---
# Create an instance of the EventBus
event_bus = EventBus()

# Subscribe both logging and notification handlers to the "order_placed" event
event_bus.subscribe("order_placed", log_event)
event_bus.subscribe("order_placed", notify_user)

# Simulate publishing an event that represents a user placing an order
order_data = {"user": "Alice", "item": "Laptop"}
event_bus.publish("order_placed", order_data)

"""
Expected Output:
[EventBus] Publishing event: order_placed
[Logger] Event received: {'user': 'Alice', 'item': 'Laptop'}
[Notification] Notifying user: Alice

Explanation:
- First, the event is published.
- Then, log_event() prints the entire payload.
- Then, notify_user() prints a message specific to the user.
"""
```

Unit of Work Pattern

*The **Unit of Work (UoW)** pattern is used to group a set of operations (usually database commands) into a **single transactional unit**. Either **all operations succeed** or **none do**, ensuring **data consistency**.*

Use UoW when:

- You want to **track changes across multiple objects** and save them in a single transaction
- You're working with **repositories or domain models** and want to avoid partial updates.
- You need **transactional safety** in service layers.

Common in:

- ORMs (like SQLAlchemy, Django ORM)
- Domain-Driven Design (DDD)
- Financial and order-processing systems

```python
from typing import List, Protocol

# Define a protocol (interface) for operations that can be committed/rolled back
class Command(Protocol):
    def execute(self) -> None:
        ...
    def rollback(self) -> None:
        ...

# --- Unit of Work ---
class UnitOfWork:
    def __init__(self):
        """ Initializes the Unit of Work with empty command and rollback stacks."""
        self._commands: List[Command] = []

    def register(self, command: Command) -> None:
        """Adds a new command (operation) to the current unit of work."""
        self._commands.append(command)

    def commit(self) -> None:
        """Executs all registered commands. If any command fails,
        rolls back all previously executed commands"""
        executed = []
        try:
            for cmd in self._commands:
                cmd.execute()
                executed.append(cmd)
        except Exception as e:
            print(f"[UnitOfWork] Error: {e}. Rolling back...")
            for cmd in reversed(executed):
                cmd.rollback()

# --- Example Commands ---
class CreateUserCommand:
    def __init__(self, username: str):
        self.username = username
        self.created = False

    def execute(self) -> None:
        # Simulate DB insert
        print(f"[DB] Creating user: {self.username}")
        self.created = True
        if self.username == "fail":
            raise Exception("Simulated failure")

    def rollback(self) -> None:
        if self.created:
            print(f"[DB] Rolling back user: {self.username}")

class SendWelcomeEmailCommand:
    def __init__(self, username: str):
        self.username = username
        self.sent = False

    def execute(self) -> None:
        print(f"[Email] Sending welcome email to: {self.username}")
        self.sent = True

    def rollback(self) -> None:
        if self.sent:
            print(f"[Email] Recalling welcome email for: {self.username}")

# Usage
uow = UnitOfWork()
uow.register(CreateUserCommand('Alice'))
uow.register(SendWelcomeEmailCommand("Alice"))
uow.commit()
'''Expected Output:
[DB] Creating user: Alice
[Email] Sending welcome email to: Alice'''
```

Active Record Pattern

*The **Active Record** pattern is an architectural pattern where an object wraps a database row and includes both the **data** and the **behavior** (CRUD operations) to manipulate it. Each object represents a row in the database and contains methods to **save, update, delete,** and **query** itself.*

Use Active Record when:

- You want to simplify persistence logic by **embedding it inside models**
- Your application is **CRUD-heavy**
- You prefer **convention over configuration**

Common in:

- Django ORM (models.Model)
- Ruby on Rails
- Flask with SQLAlchemy (via Base declarative models)

Limitations:

- Not ideal for **complex business logic**
- Violates **Single Responsibility Principle** in large systems

```python
from typing import ClassVar

class User:
    # Class-level storage acting as a mock database
    _db: ClassVar[dict[int, "User"]] = {}
    _id_counter: ClassVar[int] = 1

    def __init__(self, name: str):
        self.id = User._id_counter
        self.name = name
        User._id_counter += 1

    def save(self) -> None:
        """Saves the current user instance to the in-memory 'database'."""
        User._db[self.id] = self
        print(f"[DB] Saved : {self}")

    def delete(self) -> None:
        """Deletes the current user instance from the 'database'."""
        if self.id in User._db:
            del User._db[self.id]
            print(f"[DB] Deleted: {self}")

    @classmethod
    def find_by_id(cls, user_id: int) -> "User | None":
        """Retrieves a user from the 'database' by ID."""
        return cls._db.get(user_id)

    def __repr__(self) -> str:
        return f"User(id={self.id}, name='{self.name}')"

# --- Usage ---
# Create and save a user
alice = User("Alice")
alice.save()

# Retrieve a user
found = User.find_by_id(alice.id)
print(f"[Query] Found: {found}")

# Delete the user
alice.delete()

'''Expected Output:
[DB] Saved: User(id=1, name='Alice')
[Query] Found: User(id=1, name='Alice')
[DB] Deleted: User(id=1, name='Alice')'''
```

Architectural Patterns Cheat Sheet

Pattern	Type	Purpose	Typical Use Case	Usage Context
Singleton	Creational	Ensure a class has only one instance	Config managers, loggers	Shared global state
Factory	Creational	Create objects without specifying concrete class	Shape/connection factories	Object creation decoupled from client code
Abstract Factory	Creational	Produce families of related objects	GUI toolkits, themed widgets	Swappable object families
Builder	Creational	Build complex objects step-by-step	Document builders, query builders	Customization-heavy object creation
Adapter	Structural	Convert one interface into another	Legacy integration, interface compatibility	System upgrades, API transformations
Composite	Structural	Treat individual and groups uniformly	UI trees, file systems	Recursive, hierarchical structures
Observer	Behavioral	Notify dependents on state change	Event handling, UI models	Decoupled event propagation
MVC	Modern	Separate model, view, and controller	Web applications, GUIs	Clean UI logic separation
Repository	Modern	Abstract data access logic	Business apps, DDD	Persistence layer encapsulation
Service Layer	Modern	Encapsulate business logic	Enterprise apps	Between controllers and repositories
CQRS	Modern	Separate read and write models	High-performance systems	Read/write optimization, scalability
Dependency Injection	Modern	Inject dependencies instead of instantiating	Testable, modular apps	Loose coupling, testability
Microservices	Modern	Split app into independently deployable services	Large, scalable apps	Distributed teams, independent scaling
Publish-Subscribe	Modern	Event-driven decoupling between components	Messaging systems, logs	Asynchronous, reactive architectures
Unit of Work	Modern	Group operations into one transactional unit	DB transactions, order handling	Ensure data consistency
Active Record	Modern	Model includes both data and persistence logic	Django, Rails, CRUD models	Simple and rapid development

Glossary

Architectural Pattern - A high-level solution to common software design problems within a given context.

Design Pattern - A general repeatable solution to a commonly occurring problem in a software design at a lower level.

Creational Pattern - A category of patterns focused on how objects are created and instantiated.

Structural Pattern - Patterns that help compose classes or objects into larger structures while keeping them flexible.

Behavioral Pattern - Concerned with object communication and responsibility distribution.

Singleton - Ensures only one instance of a class exists and provides a global access point.

Factory - Delegates instantiation logic to subclasses or methods without exposing instantiation details.

Abstract Factory - Produces families of related objects without requiring their concrete classes.

Builder - Separates object construction from its representation, enabling step-by-step customization.

Adapter - Allows incompatible interfaces to work together by converting one interface to another..

Composite - Combines objects into tree-like structures to represent part-whole hierarchies.

Observer - A subject maintains a list of observers and notifies them automatically on state changes.

MVC (Model-View-Controller) - Divides application responsibilities into model (data), view (UI), and controller (input logic).

Repository - Acts as a collection-like interface for accessing domain objects from a data source.

Service Layer - Defines application logic in a separate layer to promote separation of concerns.

CQRS - Separates read (query) and write (command) responsibilities to improve scalability and flexibility.

Dependency Injection - A technique where dependencies are passed to a class instead of being created by the class itself.

Microservices - An approach where software is split into small, independently deployable services.

Publish-Subscribe - Messaging model where publishers emit messages without knowledge of subscribers.

Unit of Work - Maintains a list of operations to be performed and coordinates them into a single transaction.

Active Record - An object pattern where each instance wraps a database row and includes persistence methods.

Domain Model - Represents conceptual entities and logic that reflect the real-world business domain.

Inversion of Control (IoC) - A design principle where control flow is inverted to improve flexibility and decoupling.

Tight Coupling - A scenario where components are heavily dependent on each other’s internal workings.

Loose Coupling - Promotes designing components with minimal knowledge of each other to increase modularity.

Separation of Concerns - Dividing a software system into distinct sections with separate responsibilities.

Encapsulation - Bundling data and the methods that operate on it, restricting access to internal representation.

Testability - The degree to which software supports testing with minimal setup, typically improved by DI/SoC.

Transaction - A unit of work that must either be fully completed or fully rolled back to maintain consistency.

References & Further Reading

Core books

Design Patterns: Elements of Reusable Object-Oriented Software - Erich Gamma, Richard Helm, Ralph Johnson, John Vlissides

Patterns of Enterprise Application Architecture - Martin Fowler

Clean Architecture - Robert C. Martin

Domain-Driven Design **-** Eric Evans

Architecture Patterns with Python - Harry Percival, Bob Gregory

Online Resources

1. https://refactoring.guru/design-patterns/python - Visual, beginner-friendly explanations of common patterns in Python.
2. https://learn.microsoft.com/en-us/dotnet/architecture/microservices/ - Language-agnostic but great for understanding the microservices mindset.
3. https://martinfowler.com - Official source of many enterprise architecture and DDD concepts
4. https://github.com/faif/python-patterns - Collection of pattern implementations in Python.

Talks & Videos

https://www.youtube.com/watch?v=C7MRkqP5NRI - Clean Architectures in Python - presented by Leonardo Giordani.

https://www.youtube.com/watch?v=o1FZ_Bd4DSM - Ariel Ortiz - Design Patterns in Python for the Untrained Eye - PyCon 2019