# 🧠 Generative AI Bootcamp

## AI System Design Patterns

**Week 0, Day 2, Session 1**

*November 14, 2025*

# 🎯 Learning Objectives

- Understand modular AI system architecture
- Learn key software design patterns for AI applications
- Recognize scalability, maintainability, and testability principles

# 🧩 What Is a Design Pattern?

> A reusable solution to a common software design problem.

- Encourages consistency and clarity
- Improves collaboration across teams
- Helps manage complex AI codebases

# 🏗️ AI System Layers

1. **Data layer** – Vector DB, prompt/context store, logs
2. **Model layer** – LLMs, embeddings, fine-tunes
3. **Orchestration layer** – AI workflows, routing, policies
4. **Interface layer** – API endpoints, UI, integrations

# 🏗️ AI System Layers

Compare with non-AI application

flowchart LR subgraph T[Traditional App] T1["Frontend (UI)"] T2["Backend (business logic, APIs)"] T3["Data Layer (DB, cache)"] T1 --> T2 --> T3 end subgraph A[AI System Layers]
A1["Interface Layer (API, UI, integrations)"]
A2["Orchestration Layer (AI workflows, routing, policies)"]
A3["Model Layer (LLMs, embeddings, fine-tunes)"]
A4["Data Layer (vector DB, prompt/context store, logs)"]
A1 --> A2 --> A3
A2 --> A4
end

T ~~~ A

# 🏗️ AI System Layers

Enterprise AI application

flowchart LR FE["Frontend"] --> BE["Backend"] BE -->|Business path| SVC[Traditional Backend Services] subgraph O[Orchestration Layer] ETL[Ingestion / Preprocessiong] AI[AI Services] end subgraph Data[Data Layer] DB[(System of Record DB)] VDB[(Vector DB / Artifacts)] LOGS[(Prompt / Result Logs)] end BE -->| AI path| O SVC --> DB O --> VDB O --> LOGS O --> MODEL["Model Layer (LLM / Embeddings)"]

# 🔁 Why Patterns Matter in AI

- Prevent “spaghetti AI pipelines”
- Support reusability and testing
- Enable easier debugging and monitoring

# ⚙️ Core Patterns for AI Systems

- **Factory Pattern** – dynamic model selection
- **Strategy Pattern** – switch between algorithms or prompts
- **Adapter Pattern** – standardize APIs
- **Observer Pattern** – monitor outputs or feedback

# 🧱 Example: Factory Pattern

```
class ClientFactory:
    def get_client(self, name):
        if name == 'openai':
            from openai import OpenAI
            return OpenAI()
        elif name == 'gemini':
            from google import genai
            return genai.Client()
        else:
            raise ValueError('Unknown client')

factory = ClientFactory()
client = factory.get_client('openai')
```

## 🧭 Example: Strategy Pattern

```
def summarize_basic(text):
    return text.split('.')[0] + '.'

def summarize_keywords(text):
    return ', '.join(text.split()[:5]) + '...'
```

Want to use different summarization strategies without changing main code.

# 🧭 Example: Strategy Pattern

Can be done via dependency injection:

```
class Summarizer:
    def __init__(self, strategy):
        self.strategy = strategy
    def summarize(self, text):
        return self.strategy(text)

text = 'Generative AI is revolutionizing industry.'
summarizer = Summarizer(summarize_keywords)
print('Keywords:', summarizer.summarize(text))
```

# 🔌 Example: Adapter Pattern

```
from openai import OpenAI

class OpenAIAdapter:
    def __init__(self, client):
        self.client = client
    def generate(self, prompt):
        return self.client.chat.completions.create(
            model="gpt-4o-mini",
            messages=[{"role": "user", "content": prompt}]
        ).choices[0].message.content

adapter = OpenAIAdapter(client)
print(adapter.generate('Hello world!'))
```

# 👀 Example: Observer Pattern

- Log model outputs in real time
- Capture user feedback for retraining

```
class LoggerObserver:
    def update(self, data):
        print("Logging:", data)

observer = LoggerObserver()
observer.update({'event': 'inference', 'result': 'Success'})
```

# 📐 Composition Example

Combine multiple patterns to build modular pipelines:

```
factory = ClientFactory()
client = factory.get_client("openai")
adapter = OpenAIAdapter(client)
observer = LoggerObserver()
output = adapter.generate('Describe AI design patterns.')
observer.update({'event': 'output', 'data': output})
```

# 🧠 AI Pipeline Design Principles

- Separate data, model, and service logic
- Use dependency injection where possible
- Implement logging and monitoring early

# ⚖️ Scalability Considerations

- Use async APIs
- Deploy microservices with Docker
- Centralize configuration files

# 🚧 Common Pitfalls

- Hard-coded model names
- Tight coupling between components
- Lack of observability

## ✅ Summary

- Patterns simplify complex AI system design
- Factory, Strategy, Adapter, and Observer are essential for modularity
- Build scalable, testable, maintainable pipelines

# 🚀 Next Session

**Session 2: API Exploration Lab**

Hands-on practice integrating multiple model SDKs.