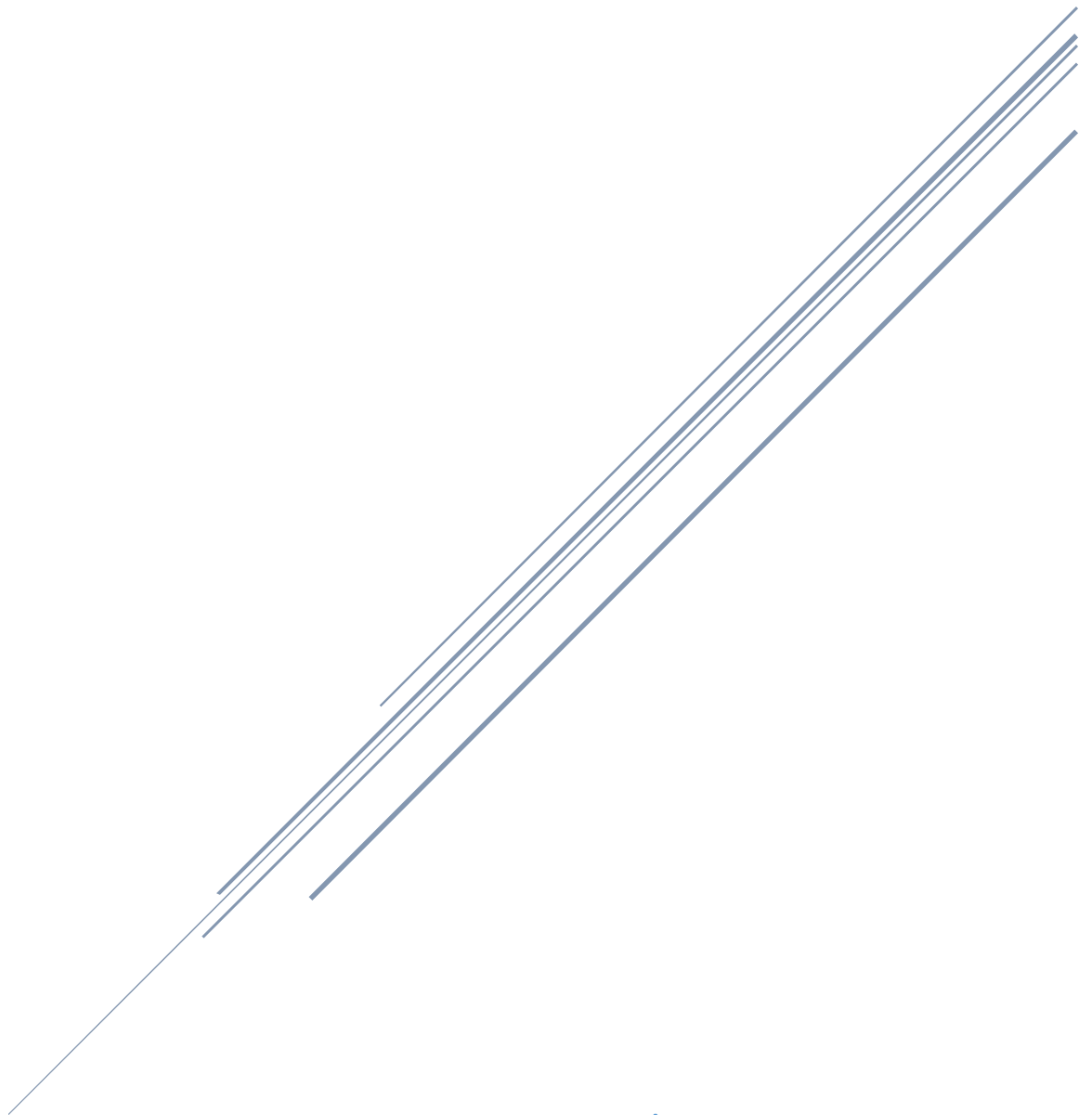


# CODE AND SUMMARY OF CLASS 5

Sahir Ahmed Sheikh

Saturday (2 – 5)



Teachers:

**Muhammad Bilal And Ali Aftab Sheikh**

## Code And Summary Of Class 5 – Saturday (2 – 5) | Quarter 3

**Assalamu Alaikum!**

Hope you all are doing well. Today's class was taken by **Sir Ameen Alam** and **Sir Ali Aftab Sheikh**. In today's session, Sir Ameen Alam provided a comprehensive revision of previously covered content from the official Panaversity GitHub repository.

Repo Link: [learn-modern-ai-python](https://github.com/panaversity/learn-modern-ai-python)

This repository contains structured lessons from Step 00 to Step 18, with Python programming as its primary focus. While all materials are valuable for comprehensive learning, students should note that the upcoming quiz will specifically cover content from Step 00 through Step 11. The goal of this session is to strengthen these foundational concepts in preparation for the assessment. Students are strongly encouraged to thoroughly review these designated steps (00-11), as they contain all key materials essential for both quiz success and real-world application readiness.

### Core Focus – Python in Real-World Applications

The heart of the session revolved around the **practical application of Python programming** in real-world scenarios. Sir Ameen emphasized that mastering Python syntax is just the starting point. Students must learn **how Python integrates with systems**, processes data, and solves business problems. The session encouraged a mindset shift—from writing code in isolation to thinking about how that code functions **within a broader application ecosystem**.

### Learning Objectives – Becoming Proficient in Python

Students were reminded that the path to becoming an efficient Python developer involves **clear understanding of logic, syntax, and problem-solving methodologies**. However, more important is the ability to apply these skills practically. Sir Ameen highlighted that proficiency is judged not just by writing correct syntax but by understanding its impact in software systems, interviews, and real-world development environments.

## Key Concepts Discussed

### 1. Python Data Types and Error Checking

An essential discussion focused on how Python handles data types and errors at runtime. A comparison was made with TypeScript's Just-In-Time (JIT) error checking. The class explored whether Python enforces strict type checking in real-time or runs code with potential data type mismatches. This led to a research-based assignment, prompting students to explore Python's behavior regarding runtime errors and type validations using tools like VS Code and MyPy (a tool that adds stricter type checking to Python).

### 2. Python Operators, Casting, and Control Flow

The session progressed into a comprehensive exploration of Python operators, with special emphasis on type casting mechanisms—both explicit (manually declared conversions, e.g., `str(42)`) and implicit (automatically handled by Python, e.g., `3 + 7.0`). **This naturally led to the foundational question: What is string casting?** In Python, converting data to string type can be either explicit (`str(123) → "123"`) or implicit (automatic concatenation with strings, e.g., `"Age: " + 25` raises `TypeError` unless explicitly cast).

To make it easy to understand

- Type Casting (Explicit vs Implicit)

→ Explicit: When YOU manually convert types

Example: `str(25) →` converts number 25 to string "25"

→ Implicit: When PYTHON automatically converts types

Example: `3 + 2.5 →` converts integer 3 to float 3.0

#### What is Control Flow?

The discussion then transitioned to control flow structures—the programming paradigm that interrupts linear execution (characteristic of scripting languages like Python) to make dynamic decisions. At its core, control flow uses conditional logic (`if/else`) to branch execution paths.

### Simple Definition:

Control flow is how your Python program decides which code to run next. Normally, Python runs code line-by-line from top to bottom. Control flow lets you break this order based on conditions. For instance:

```
age = 18
if age >= 18:
    print("You can vote") # ← This runs only if condition is True
else:
    print("Can't vote")   # ← Otherwise this runs
```

Through hands-on examples, students strengthened their ability to architect decision-driven Python code, solidifying their understanding of both type manipulation and program flow control—essential skills for writing robust, adaptable applications.

### 3. Modularity and Code Organization

Sir Ameen emphasized how modular programming transforms code into reusable components, demonstrated through a custom calc module. The critical Python construct:

```
if __name__ == "__main__":
```

was explained as a gatekeeper that executes code only when the file runs directly (not when imported). For instance:

- Running calc.py directly → `__name__` becomes `"__main__"` → Triggers `add()`
- Importing calc into app.py → `__name__` reflects `"calc"` → Code remains dormant.

#### **This enables:**

- ✓ Clean imports (no unintended executions)
- ✓ Self-testable modules
- ✓ Professional-grade code separation

## Why Learn Python?

Python unlocks global tech opportunities:

- Versatility: Backs web apps (Django), AI (TensorFlow), cloud (AWS), and robotics
- Job Market: 70% of machine learning jobs demand Python (Indeed 2023)
- Future-Proof: Powers automation (e.g., Boston Dynamics' robots use Python scripts).

## Key Differentiator:

While beginners write basic functions like:

```
def add(a, b):  
    return a + b
```

Experts leverage Python's depth – like understanding `__call__` enables creating callable objects:

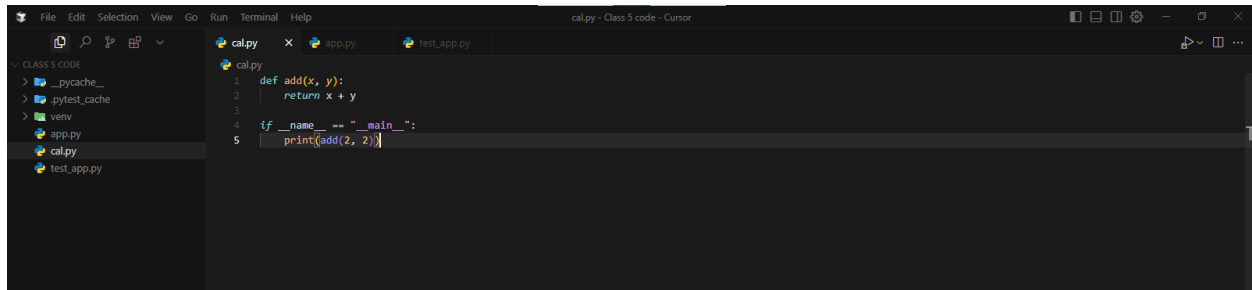
```
class Adder:  
    def __call__(self, a, b):  
        return a + b  
add = Adder() # Now add() is an object masquerading as a function!
```

## Real-World Impact:

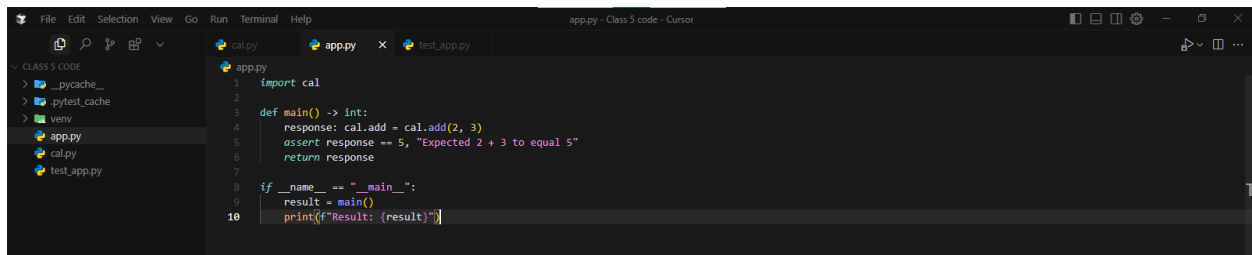
A candidate who explains `__name__ == "__main__"` and callable objects demonstrates:

- ◆ System-level Python mastery
- ◆ Ability to write import-safe libraries
- ◆ Readiness for senior engineering roles

This holistic approach – combining modular design with language internals – is what makes Python skills interview-worthy at FAANG companies.



```
cal.py
1 def add(x, y):
2     return x + y
3
4 if __name__ == "__main__":
5     print(add(2, 2))
```



```
app.py
1 import cal
2
3 def main() -> int:
4     response = cal.add(2, 3)
5     assert response == 5, "Expected 2 + 3 to equal 5"
6     return response
7
8 if __name__ == "__main__":
9     result = main()
10    print(f"Result: {result}")
```

## 4. Introduction to Unit Testing with Pytest

The session provided a comprehensive introduction to unit testing using pytest, emphasizing its role in test-driven development (TDD). Here's a detailed breakdown of key concepts covered:

### ◆ Pytest Testing & Naming Conventions

Pytest simplifies testing by following smart naming conventions:

#### 1. File Naming:

- Files should start or end with test (e.g., test\_calc.py or calc\_test.py).

#### 2. Function Naming:

- Test functions must begin with test\_ (e.g., test\_addition()).

### How Pytest Automates Testing?

Simply run:



- Pytest **automatically** scans for:

- ✓ Files matching **test\_\*.py** or **\*\_test.py**
- ✓ Functions prefixed with **test\_**

- **No manual file/function specification needed!**

**Example Test Case:**

```
# test_calc.py
def test_addition():
    assert add(2, 3) == 5 # Passes if add() returns 5
```

### ◆ Handling Errors & Debugging

A **deliberate error** was introduced to demonstrate pytest's **failure detection**:

```
def add(a, b):
    return a / b # Logical error (should be +)
```

**Test Result:**

```
- assert add(2, 3) == 5
+ Actual output: 0.666... (due to division)
```

**Key Takeaway:**

- ✓ Tests fail if actual  $\neq$  expected output.
- ✓ Forces developers to fix logic errors early.

### ◆ Best Practices for Developers

#### 1. TDD Mindset:

- Write tests **before** coding.

- Ensures functionality matches requirements.

## 2. Pre-Share Checklist:

- ✓ Run tests locally before sharing code.
- ✓ Review test results when receiving others' code.

## 3. Interview/Industry Standards:

- Writing **tested code** impresses interviewers.
- Companies enforce testing in **CI/CD pipelines**.

### ◆ Real-World Application

#### Why This Matters?

**Bug Prevention:** Catch errors before deployment.

**Code Maintainability:** Tests act as documentation.

**Team Collaboration:** Shared tests = fewer integration issues.

**Pro Tip:** Use **pytest -v** for verbose output, showing passed/failed tests clearly.

#### Final Thought

Adopting pytest and TDD transforms coding from "**Does it work?**" to "**How do we prove it works?**"—a critical skill for **FAANG-level interviews** and professional software development.



## 5. Virtual Environments for Project Isolation

The session emphasized **virtual environments** as a critical tool for **project isolation**, ensuring dependency conflicts are avoided. Here's a detailed breakdown with actionable steps:

### ◆ What is a Virtual Environment?

A **self-contained directory** that isolates a project's dependencies (libraries/packages) from other projects and the global Python installation.

### Why Use It?

- ✓ Prevents **version clashes** (e.g., Project A needs Django 3.0, Project B needs Django 4.0)
- ✓ Enables **clean project management** (no global pollution)
- ✓ Simplifies **collaboration** (share **requirements.txt** instead of entire systems)

### ◆ How to Create & Activate a Virtual Environment?

#### 1. Creation

Run in terminal (project directory):

```
python -m venv myenv # Replace 'myenv' with your environment name (e.g., 'amin')
```

- Creates a folder (myenv/) containing Python binaries and package storage.

#### 2. Activation

- Windows:

```
.\myenv\Scripts\activate
```

- macOS/Linux:

```
source myenv/bin/activate
```

✓ **Success:** Terminal prompt shows **(myenv)**, confirming activation.

### 3. Deactivation

```
deactivate
```

#### ◆ Pro Tips

- **VS Code Integration:** Automatically detects and activates virtual environments.
- **.gitignore:** Exclude virtual env folders (add **myenv/**).
- **Reusability:** Reactivate environments anytime **with source myenv/bin/activate**.

**Interview Insight:** FAANG interviews often ask about dependency management—virtual environments demonstrate **professional-grade practices**.

## 6. Using External Libraries – API Interaction with requests

The session provided a hands-on introduction to third-party libraries in Python, with a focus on the powerful requests library for HTTP API interactions. Here's a comprehensive breakdown of key concepts covered:

#### ◆ Installing and Using External Libraries

##### 1. Installation via pip:

```
pip install requests # Installs the latest version
```

## 2. Basic API Call:

```
import requests
response = requests.get("https://api.github.com")
```

### ◆ Understanding API Responses

- Status Codes:

```
print(response.status_code) # 200 = Success, 404 = Not Found, etc.
```

- Response Data:

```
print(response.text)      # Raw text response
print(response.json())    # Parsed JSON (for APIs)
```

### Example: Fetch Google's Homepage

```
response = requests.get("https://google.com")
if response.status_code == 200:
    print("Success! Website is reachable.")
else:
    print("Failed with status:", response.status_code)
```

### ◆ Why This Matters in Real Development

1. **Microservices:** Modern apps rely on API communication
2. **Data Pipelines:** Fetching data from sources like Twitter/Weather APIs
3. **Automation:** Scripting interactions with web services

## 7. Exception Handling In Python

### ⚠ What is an Error in Programming?

Errors are unexpected problems that stop program execution.

### ✗ Example:

```
print(10 / 0)
```

**Output:** ZeroDivisionError: division by zero

- The program stops here and doesn't continue.
- This is where **Exception Handling** comes in — to catch such errors and continue the program.



## 💡 What is Exception Handling?

Exception handling allows a program to gracefully recover from errors without crashing.

### 🔧 Syntax:

```
try:
    # Code that might raise an error
except ErrorType:
    # Code that runs if error occurs
```

### 🔍 Example:

```
try:
    print(10 / 0)
except ZeroDivisionError:
    print("You cannot divide by zero.")
```

✅ Output: You cannot divide by zero.

🚫 The program doesn't crash — it handles the error.

## 👤 Why Specific Exception Handling Matters

Using specific error types (like **ZeroDivisionError**, **TypeError**, **ValueError**) helps in providing meaningful messages and better debugging.

### 💡 Example: Multiple Error Types

```
try:
    a = int(input("Enter a number: "))
    b = int(input("Enter another number: "))
    result = a / b
except ZeroDivisionError:
    print("Number cannot be divided by zero.")
except ValueError:
    print("Only numeric input is allowed.")
except TypeError:
    print("Invalid type provided.")
else:
    print("Division successful:", result)
```

### 🚩 Explanation:

- If  $b = 0 \rightarrow \text{ZeroDivisionError}$
- If `input = "apple" \rightarrow \text{ValueError}`
- If wrong data type used  $\rightarrow \text{TypeError}$
- If no error  $\rightarrow$  else runs

### 🍕 Analogy for Better Understanding

Sir uses an example:

If you're making pizza at home and gas runs out or pizza burns  $\rightarrow$  It's like an **exception**.

If everything works fine  $\rightarrow$  the else block runs — **Pizza successful!**

### 👁 Practical Try-Except-Else Structure

The else block only runs if the try block has no errors.

### ✅ Example:

```
try:
    print(10 / 2)
except:
    print("An error occurred.")
else:
    print("Successfully run.")
```

Output:

5.0

Successfully run.

### ⬇ User Input Handling with Exception

Python's `input()` returns a string. If you expect a number, you must convert it.

### 🌟 Example:

```
try:
    x = int(input("Enter a number: "))
    y = int(input("Enter another number: "))
    print("Result:", x / y)
except ValueError:
    print("Invalid number entered.")
except ZeroDivisionError:
    print("Cannot divide by zero.")
else:
    print("Operation successful.")
```

### 🌸 The finally Block

The finally block is always executed, no matter whether there was an error or not. It's like a cup of tea — whether it's winter or summer, you always drink it. No matter what happens in the try or except blocks, the finally block will run. This is where you put things that must happen no matter what, like closing a database connection, cleaning up resources, etc.

Imagine this: you're working with a database. You've opened a connection. Even if an error happens during the transaction, or everything runs perfectly, you still need to close the connection. That's where finally comes in. It ensures your program behaves properly in all scenarios.

### 👤 Error Types and Practical Code Examples

So far, you've seen these three error types:

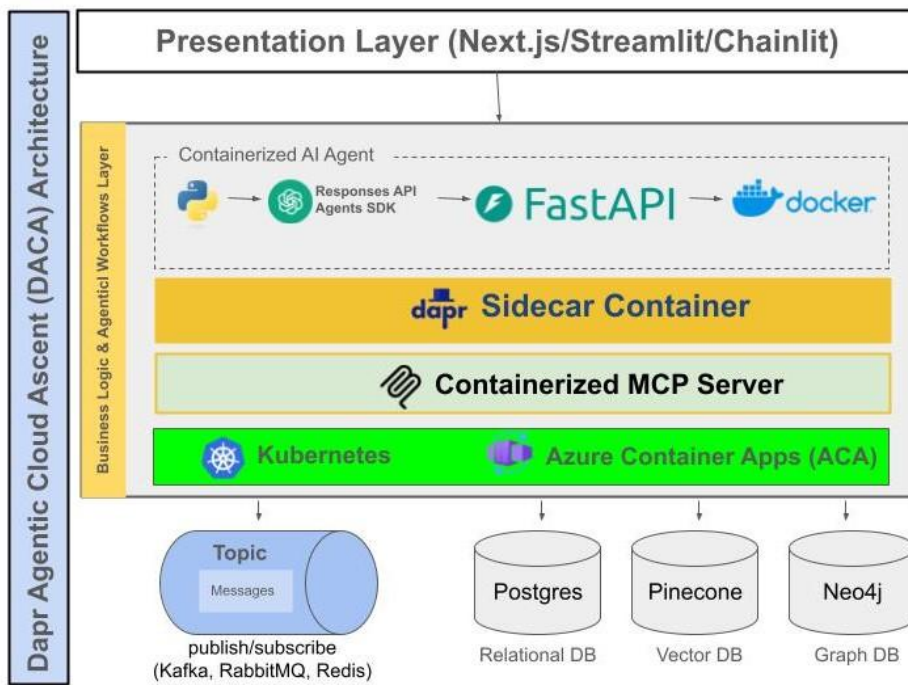
- **ZeroDivisionError:** Dividing by zero.
- **TypeError:** Performing an invalid operation on incompatible types (e.g., dividing a string by an integer).

- **ValueError:** Trying to convert an invalid string to a number (e.g., converting "Pizza" to int).

You're taught how to handle each using multiple except blocks. **For example:**

```
try:
    num1 = int(input("Enter first number: "))
    num2 = int(input("Enter second number: "))
    result = num1 / num2
except ZeroDivisionError:
    print("Error: Cannot divide by zero.")
except ValueError:
    print("Error: Please enter only numbers.")
except TypeError:
    print("Error: Incompatible types for division.")
else:
    print("Successfully Run:", result)
finally:
    print("This will always run (finally block).")
```

## 8. Kubernetes & Cloud-Native Architecture





## **1) Presentation Layer (Next.js/Streamlit/Chainlit)**

The Presentation Layer serves as the user-facing interface, built using frameworks like:

- Next.js: For dynamic web applications.
- Streamlit/Chainlit: For rapid AI app prototyping with Python.

## **2) Data Layer & Containerized AI Agent**

Containerized AI Agent: Deployed via Docker for consistency across environments.

APIs & SDKs:

- FastAPI: High-performance backend for handling AI responses.
- Responses API Agents SDK: Facilitates communication between services.

## **3) Kubernetes (K8s) & Cloud-Native Infrastructure**

Kubernetes is the backbone of modern cloud-native applications, offering:

- Orchestration: Automates deployment, scaling, and management of containers.
- Self-Healing: Restarts failed containers to ensure uptime.
- Load Balancing: Distributes traffic across multiple instances (e.g., scaling from 1 to 3 replicas).

Key Components:

- Azure Container Apps (ACA): Serverless Kubernetes service for simplified deployments.
- Message Brokers (Kafka/RabbitMQ/Redis): Handle pub/sub workflows for real-time data.

Databases:

- Postgres: Relational DB for structured data.
- Pinecone/Vector DB: For AI/ML embeddings.
- Neo4j: Graph DB for relationship-heavy data.

#### **4) Why Kubernetes?**

Industry Standard: Used by FAANG companies (e.g., Google runs billions of containers on K8s).

Scalability:

- Scale Up: Boost CPU/RAM of existing servers.
- Scale Out: Add more servers (e.g., `kubectl scale --replicas=3`).
- Career Growth: Cloud architects earn \$150K+ (Indeed 2023).

#### **5) Cloud-Native Principles**

- Hybrid Approach: Start on-premises, then migrate to cloud (e.g., Azure).
- K8s in Pakistan: Certified professionals prioritize Kubernetes training due to global demand.

#### **6) Key Takeaways**

- K8s = Traffic cop for containers, ensuring efficiency and reliability.
- Cloud-Native = Microservices + Scalability + DevOps automation.
- Future-Proof Skills: Learn K8s, FastAPI, and cloud services (Azure/ACA) to stay competitive.

Diagram Interpretation

The architecture flows as:

User → Presentation Layer (Next.js) → AI Agent (FastAPI/Docker) →  
Kubernetes (Orchestration) → Databases/Message Brokers → Cloud (Azure  
ACA).

## Class Resources

### Today's Class Code:

The complete code for today's class is available in this GitHub repository:

Repo Link: [Class Code](#)

Stay consistent and keep learning!

## Thank You for Reading!

Hope you understood Class 5 well.

"Success is no accident. It is hard work, perseverance, learning, studying, sacrifice and most of all, love of what you are doing. " – *Pellé*