Cloud Computing
Assignment # 1: (BSE – 6A)

Design a Cloud-Based Solution for a Smart Campus Management System (No Deployment Required)

You are hired as a cloud solutions architect to design a smart campus management system for a university. The system must manage online attendance, learning content delivery, notifications, and virtual classrooms. Your task is to design the cloud architecture, select appropriate cloud service models, and justify the use of specific services while addressing performance, scalability, cost, and security.

Functional Modules (Design Only)

- 1. User Management Module (Student/Faculty/Admin roles)
- 2. Lecture & Content Management System
- 3. Attendance & Scheduling
- 4. Notification System (SMS/Email)
- 5. Virtual Classroom Integration (e.g., Zoom/MS Teams APIs or any other free service for classroom)
- 6. Analytics Dashboard for Faculty/Admins

What you Must Do

- **Design System Architecture**: Create a complete system architecture diagram using cloud components and identify which modules use IaaS, PaaS, or SaaS.
- **Select Cloud Services**: Choose cloud platform (AWS, Azure, GCP) and justify selections for compute, storage, messaging, and identity services.
- **Analyze Non-Functional Requirements**: Include analysis for scalability, fault tolerance, security/privacy, and cost.
- **Service Model Mapping**: Identify and explain which parts of the system use IaaS, PaaS, or SaaS.
- **Prepare a Risk & Challenge Matrix**: List potential risks and their mitigation strategies.

Required outcomes:

- **System Architecture Diagram:** Visual design showing services/modules and their connections
- **Service Selection Rationale:** Justify every chosen cloud service
- **Service Model Mapping:** Table showing IaaS, PaaS, SaaS assignments
- **Risk & Challenge Matrix:** Table with risks, impacts, and mitigation plans
- **Cost Estimation**: Use online calculators or documentation estimates
- **Final Report:** Professional write-up covering all of the above

Washington Accord Attributes Mapping

WK – Knowledge Profile

Code	Description	Mapping
WK1	Understanding of computing systems	Cloud service models, virtualization concepts
WK4	Engineering fundamentals	Designing scalable and secure architectures
WK6	Knowledge of tools and technologies	Evaluating AWS/Azure/GCP services

WP – Problem Solving Profile

Code	Description	Mapping
WP1	Problem identification	Identify real-world needs of campus systems
WP2	Problem analysis	Evaluate performance, cost, and availability constraints
WP3	Design and development	Cloud-native system architecture planning
WP6	Engineering judgment	Choose services based on constraints and justify decisions

Evaluation Rubric (Out of 100)

Criteria	Description	Marks
Architecture Design	Clear, logical system architecture using cloud components; includes communication and modularity	20
Cloud Service Selection & Justification	Appropriateness and justification of cloud platform and service choices (e.g., compute, storage, identity)	20
Service Model Mapping	Accurate identification and explanation of IaaS, PaaS, SaaS usage for each module	15
Risk & Challenge Analysis	Includes potential risks and clear mitigation strategies (e.g., downtime, cost overruns, lock-in)	10
Cost Estimation	Reasonable and well-researched use of free tier/pricing calculators for cost breakdown	10
Report Quality & Presentation	Well-structured report with professional formatting, clarity, and coherence	15
Innovation / Additional Insights	Creative enhancements, futuristic thinking, or alternative architecture ideas	10

Lab 7: Implementing A Search Algorithm*

Learning Objective

To implement the A* (A-Star) Search Algorithm and understand its working mechanism using Python.

A* Search

- A* Search is an **informed search** technique (also known as **best-first search**).
- It combines features of **Uniform Cost Search** and **Greedy Best-First Search**.
- It evaluates nodes using the function:

$$f(n) = g(n) + h(n)$$

  - **g(n)**: Actual cost from the start node to the current node.
  - **h(n)**: Heuristic estimate of the cost from the current node to the goal.
- A* finds the lowest cost path to the goal if **h(n)** is **admissible** (never overestimates) and **consistent**.

Why A* is Important?

- Finds the **optimal path** using both costs so far and estimated future cost.
- Commonly used in **navigation systems**, **game AI**, **robot path planning**, and more.

Key Properties of A*

- Complete
- Optimal (with admissible heuristic)
- Uses both path cost and heuristic to guide search
- Uses priority queue (open set) to store frontier nodes

Description of A* Components

1. openSet

- A collection of discovered nodes that are yet to be evaluated.
- Initially contains only the start node.
- Nodes are picked based on the lowest fScore.

2. cameFrom

- A map tracking the most efficient previous node for each explored node.
- Helps reconstruct the final path once the goal is reached.

3. gScore

- A map storing the cost of the shortest path from the start node to every other node.
- Initialized with ∞ for all nodes except the start node (set to 0).

4. fScore

- Estimated total cost from start to goal through a node: f(n) = g(n) + h(n).
- Combines known cost and estimated cost to guide the search.

5. h(n)

- The heuristic function that estimates the cost from a node to the goal.
- Must be admissible and consistent for A* to guarantee the shortest path.

6. reconstruct_path()

- Builds the final path from goal to start using cameFrom.

A* Pseudocode

```
function A_Star(start, goal, h):
   openSet = {start}                  // Initialize the open set with the start node
   cameFrom = empty map          // Tracks the best previous step to each node
    gScore = map with default ∞       // Cost from start to each node
   gScore[start] = 0
   fScore = map with default ∞       // Estimated total cost from start to goal through each node
fScore[start] = g(start) + h(start) = 0 + h(start)
fScore[start] = h(start)
   while openSet is not empty:
      current = node in openSet with lowest fScore[]     // Select node with the lowest estimated cost
```

```
if current == goal:
    return reconstruct_path(cameFrom, current)      // Goal reached, reconstruct the path
remove current
```

Lab Task

Write a Python program to:

- Implement a graph using dictionaries.
- Define a heuristic function for each node.
- Use the A* algorithm to find the shortest path between two nodes.
- Print the final path and cost.

Note: The efficiency of A* highly depends on the heuristic function used.