Vasavi College of Engineering (A)

(Affiliated to Osmania University) Hyderabad-500031

Department Name: CSE

Name Of the Laboratory: AAI

Name: Roll Number: Page no:

WEEK-1

1. Simple Reactive Agent: Implement a reactive cleaning agent that exists in a two-room environment. The agent should perceive its current location and the status of that room ('Dirty' or 'Clean'). Its action should be based on these perceptions: if the current room is dirty, clean it; otherwise, move to the other room.

```
class TwoRoomEnvironment:
    def __init__(self, initial_state=None):
        # State: { 'A': 'Clean' or 'Dirty', 'B': 'Clean' or 'Dirty' }
        if initial_state:
            self.state = initial_state
        else:
            self.state = {'A': 'Dirty', 'B': 'Dirty'}
        self.agent_location = 'A'  # Agent starts in room A
    def perceive(self):
        return self.agent_location, self.state[self.agent_location]
    def execute_action(self, action):
        if action == 'Clean':
            self.state[self.agent_location] = 'Clean'
        elif action == 'Move':
            self.agent_location = 'B' if self.agent_location == 'A' else 'A'
    def is_done(self):
        return self.state['A'] == 'Clean' and self.state['B'] == 'Clean'
class ReactiveAgent:
    def act(self, percept):
        location, status = percept
        if status == 'Dirty':
            return 'Clean'
        else:
            return 'Move

#simulation
env = TwoRoomEnvironment({'A': 'Dirty', 'B': 'Dirty'}) agent = ReactiveAgent()
steps = 0
while not env.is_done():
    percept = env.perceive()
    action = agent.act(percept)
    print(f"Step {steps}: Location={percept[0]} | Status={percept[1]} | Action={action}")
    env.execute_action(action)
    steps += 1
print("Both rooms are clean!")
```

Output:

```
Step 0: Location=A | Status=Dirty | Action=Clean
Step 1: Location=A | Status=Clean | Action=Move
Step 2: Location=B | Status=Dirty | Action=Clean
Both rooms are clean!
```

2. Simple Deliberative Agent: Create a deliberative agent for a warehouse robot. The agent's goal is to pick up an item and deliver it. It must first generate a complete, sequential plan (create_plan) and then execute it step-by-step (execute_plan).

```
class WarehouseEnvironment:

  def __init__(self):
      self.robot_location = 'Entrance'
      self.item_location = 'Storage'
      self.delivery_location = 'Shipping'
      self.has_item = False
      self.item_delivered = False

class DeliberativeAgent:

   def create_plan(self, env):
      plan = []
      # Move to item location
      if env.robot_location != env.item_location:
         plan.append(f"Move to {env.item_location}")
      # Pick up item

      plan.append("Pick up item")
      # Move to delivery location
      if env.item_location != env.delivery_location:
         plan.append(f"Move to {env.delivery_location}")

      # Deliver item
      plan.append("Deliver item")
      return plan

   def execute_plan(self, env, plan):

      for step in plan:
         print(f"Executing: {step}")
         if step.startswith("Move to"):
            location = step.replace("Move to ", "")
            env.robot_location = location

         elif step == "Pick up item":
            if env.robot_location == env.item_location:
               env.has_item = True
               print("Item picked up.")
         else:
               print("Cannot pick up: Not at item location.")

         elif step == "Deliver item":

            if env.robot_location == env.delivery_location and env.has_item:
               env.item_delivered = True
               env.has_item = False
               print("Item delivered.")
            else:

                print("Cannot deliver: Item not picked or wrong location.")

        print(f"Current location: {env.robot_location}, Has item: {env.has_item}, Delivered: {env.item_delivered}")
# Example usage

env = WarehouseEnvironment()
agent = DeliberativeAgent()
plan = agent.create_plan(env)
print("Plan generated:", plan)
agent.execute_plan(env, plan)
```

Output:

```
Plan generated: ['Move to Storage', 'Pick up item', 'Move to Shipping', 'Deliver item']

Executing: Move to Storage

Current location: Storage, Has item: False, Delivered: False
Executing: Pick up item

Item picked up.

Current location: Storage, Has item: True, Delivered: False
Executing: Move to Shipping

Current location: Shipping, Has item: True, Delivered: False
Executing: Deliver item

Item delivered.

Current location: Shipping, Has item: False, Delivered: True
```

WEEK-2

1. Traditional AI vs. Agentic AI Approach: Demonstrate the difference between a traditional function and an agentic approach for recommending a movie. The agent should consider more context (e.g., user's mood) in its think phase compared to a simple function.

```
class MovieAgent:

    def __init__(self, user_profile):

        self.user_profile = user_profile
    def think(self):
        # Reason over multiple pieces of context
        mood = self.user_profile.get('mood')
        liked_genres = self.user_profile.get('liked_genres')
        watch_history = self.user_profile.get('watch_history')
        # Prioritize mood-driven recommendation
         if mood == 'happy':

            return 'Comedy'
        elif mood == 'sad':
             return 'Feel Good'

        elif 'Action' in liked_genres and 'Action' not in watch_history:
            return 'Action'
        else:

            return 'Drama'
    def act(self, genre):
        # Recommend movie based on chosen genre
        genre_map = {
            'Comedy': 'The Grand Budapest Hotel',
            'Feel Good': 'Amélie',
            'Action': 'Mad Max: Fury Road',

            Drama': 'The Shawshank Redemption'

        }

        return genre_map.get(genre, 'Inception')
    def recommend_movie(self):
        genre = self.think()
        return self.act(genre)
# Example use:

user_profile = {
    'mood': 'sad',
    'liked_genres': ['Action', 'Drama'],
    'watch_history': []
}

agent = MovieAgent(user_profile)
print(agent.recommend_movie()) # Output: Amélie
```

Output:

```
Amélie
```

2. Cooperative Multi-Agent System (MAS): Simulate a cooperative system where a GeneratorAgent produces a list of data points (numbers), and a CheckerAgent analyzes the entire list to provide a statistical summary (count of even and odd numbers).

```
import random

class GeneratorAgent:

    def __init__(self, n):

        self.n = n

    def generate_data(self):

        # Generate a list of n random integers between 1 and 100
        data = [random.randint(1, 100) for _ in range(self.n)]


        print(f"GeneratorAgent produced data: {data}")
        return data
class CheckerAgent:

    def analyze_data(self, data):

        even_count = sum(1 for x in data if x % 2 == 0)
        odd_count = len(data) - even_count
        print(f"CheckerAgent analysis: Even numbers = {even_count}, Odd numbers = {odd_count}")
        return even_count, odd_count
# Simulation

generator = GeneratorAgent(n=20)
data_list = generator.generate_data()
checker = CheckerAgent()
checker.analyze_data(data_list)
```

Output:

```
GeneratorAgent produced data: [54, 73, 82, 30, 18, 27, 63, 79, 85, 97, 21, 18, 99, 98, 21, 32, 7, 98, 72, 89]

CheckerAgent analysis: Even numbers = 9, Odd numbers = 11
(9, 11)
```

WEEK-3

1. E-commerce Multi-Agent System (MAS): Model an e-commerce backend with an InventoryAgent that manages stock levels and an OrderAgent. The OrderAgent communicates with the inventory agent to process an order, which includes checking availability and updating the stock if the order is successful.

```
class InventoryAgent:

    def __init__(self, initial_stock):
        # initial_stock is a dict: {item_name: quantity}
        self.stock = initial_stock

    def check_availability(self, item, quantity):
        available = self.stock.get(item, 0) >= quantity
        print(f"InventoryAgent: Checking availability of {quantity} '{item}' - {'Available' if available else 'Not Available'}")
        return available

    def update_stock(self, item, quantity):
        if self.stock.get(item, 0) >= quantity:
            self.stock[item] -= quantity
            print(f"InventoryAgent: Stock updated. Remaining '{item}': {self.stock[item]}")
            return True
        print(f"InventoryAgent: Cannot update stock for '{item}', insufficient quantity.")
        return False

class OrderAgent:
    def __init__(self, inventory_agent):
        self.inventory_agent = inventory_agent
    def process_order(self, item, quantity):
         print(f"OrderAgent: Processing order for {quantity} '{item}'")
         if self.inventory_agent.check_availability(item, quantity):
             success = self.inventory_agent.update_stock(item, quantity)
            if success:
                print(f"OrderAgent: Order successful for {quantity} '{item}'")
                return True
            else:
                print(f"OrderAgent: Order failed during stock update.")
                return False
        else:
            print(f"OrderAgent: Order failed - Item not available.")
            return False

# Example use
initial_stock = {'laptop': 5, 'phone': 10}
inventory_agent = InventoryAgent(initial_stock)
order_agent = OrderAgent(inventory_agent)

# Process orders
order_agent.process_order('laptop', 3)  # Successful order
order_agent.process_order('laptop', 3) # Fails due to insufficient stock
order_agent.process_order('phone', 5)   # Successful order
```

2. Agent with Reflection: Demonstrate reflection in an agent that summarizes text. The SummaryAgent first performs an action (summarization) and then calls a reflect method to evaluate its output against multiple criteria (e.g., length and inclusion of keywords).

```
class SummaryAgent:
    def __init__(self, keywords):
        self.keywords = keywords
    def summarize(self, text):
        # Simple summarization: return first 2 sentences (for illustration)
        sentences = text.split('. ')
        summary = '. '.join(sentences[:2]) + ('.' if len(sentences) > 1 else '')
        print(f"SummaryAgent: Summary generated:\n{summary}")
        return summary
    def reflect(self, original_text, summary):

        results = {}
        # Criterion 1: Summary length (at least 50 characters)
        results['length_ok'] = len(summary) >= 50
        # Criterion 2: Check inclusion of keywords

        results['keywords_included'] = all(keyword.lower() in summary.lower() for keyword in self.keywords)
        # Display reflection results
        print("SummaryAgent Reflection:")
        for criterion, passed in results.items():
            print(f"- {criterion}: {'Passed' if passed else 'Failed'}")
        return results
# Example usage

text = ("Artificial intelligence (AI) is intelligence demonstrated by machines, in contrast to the natural intelligence displayed by humans and animals. "
        "Leading AI textbooks define the field as the study of \"intelligent agents\": any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals.")
keywords = ['intelligence', 'machines', 'environment']
agent = SummaryAgent(keywords)
summary = agent.summarize(text)
reflection_results = agent.reflect(text, summary)
```

Output:

```
SummaryAgent:  Summary  generated:

Artificial intelligence (AI) is intelligence demonstrated by machines, in contrast to the natural intelligence displayed by humans and animals. Leading AI textbooks define the field as the study of "intelligent agents": any device that perceives its environment and takes actions that maximize its chance of successfully achieving its goals..

SummaryAgent  Reflection:
- length_ok:  Passed
- keywords_included:  Passed
```

WEEK - 4

1. Meta-Reasoning Agent: Implement a MathSolverAgent that uses meta-reasoning to analyze a problem string. Based on the operator, it must select the correct internal strategy (_solve_addition, _solve_multiplication, etc.) before computing the result. It should also handle unknown strategies.

```
class MathSolverAgent:
   def solve(self, problem):
      # Meta-reasoning: analyze problem and delegate to internal strategies
      if '+' in problem:
         result = self._solve_addition(problem)
      elif '*' in problem:
         result = self._solve_multiplication(problem)
      elif '-' in problem:
         result = self._solve_subtraction(problem)
      elif '/' in problem:
         result = self._solve_division(problem)
      else:
         return "Unknown operation. Cannot solve."
      return result
   def _solve_addition(self, problem):
      # Parse and compute addition, e.g. "3 + 5"
      operands = problem.split('+')
      try:
         return sum(float(op.strip()) for op in operands)
      except ValueError:
         return "Invalid operands for addition."
   def _solve_multiplication(self, problem):
      # Parse and compute multiplication, e.g. "4 * 2
      operands = problem.split('*') try:
         product = 1
         for op in operands:
            product *= float(op.strip())
         return product
      except ValueError:
         return "Invalid operands for multiplication."

  def _solve_subtraction(self, problem):
      # Parse and compute subtraction, e.g. "10 - 4"
      operands = problem.split('-')
      try:
         values = [float(op.strip()) for op in operands]
         result = values[0]
         for v in values[1:]:
            result -= v
         return result
      except ValueError:
        return "Invalid operands for subtraction."

  def _solve_division(self, problem):
      # Parse and compute division, e.g. "20 / 5"
      operands = problem.split('/')
      Try:
            values = [float(op.strip()) for op in operands]
            result = values[0]
            for v in values[1:]:
                if v == 0:
                return "Error: Division by zero."
                result /= v
            return result
        except ValueError:
            return "Invalid operands for division."
# Example usage
agent = MathSolverAgent()
print(agent.solve("3 + 7"))
print(agent.solve("4 * 5"))
print(agent.solve("15 - 6"))
print(agent.solve("20 / 4"))
print(agent.solve("9 ^ 2"))  # Unknown operation
```

Output:

```
10.0
20.0
9.0
5.0
Unknown operation. Cannot solve.
```

2. Self-Explaining Agent: Create an EvaluationAgent for student grades. It must output a decision ('Pass', 'Fail', 'Distinction') and provide a clear, textual reason explaining *why* that decision was made based on its internal rules (e.g., passing score, distinction threshold).

```
class EvaluationAgent:
    def __init__(self, pass_threshold=40, distinction_threshold=85):
        self.pass_threshold = pass_threshold
        self.distinction_threshold = distinction_threshold
     def evaluate(self, score):
        if score >= self.distinction_threshold:
            decision = "Distinction"
            reason = (f"The score {score} is greater than or equal to the distinction threshold "
                    f"of {self.distinction_threshold}.")
        elif score >= self.pass_threshold:
            decision = "Pass"
            reason = (f"The score {score} is greater than or equal to the passing threshold "
                    f"of {self.pass_threshold} but less than the distinction threshold.")
        else:
            decision = "Fail"
            reason = (f"The score {score} is below the passing
        threshold of {self.pass_threshold}.") return decision,
        reason
agent = EvaluationAgent()
scores = [92, 76, 35]
for s in scores:
    decision, explanation = agent.evaluate(s)
    print(f"Score: {s} => Decision: {decision}\nReason:
    {explanation}\n")
```

Output:

```
Score: 92 => Decision: Distinction
Reason: The score 92 is greater than or equal to the
distinction threshold of 85.
Score: 76 => Decision: Pass
Reason: The score 76 is greater than or equal to the passing
threshold of 40 but less than the distinction threshold.
Score: 35 => Decision: Fail
Reason: The score 35 is below the passing threshold of 40.
```

WEEK - 5

1. Simple Learning Agent: Create a LearningChatbotAgent with a knowledge base. If it cannot answer a question, it should ask the user for the answer and add the new information to its knowledge base for future use. Implement a simple interactive loop.

```
class LearningChatbotAgent:

   def __init__(self):
       self.knowledge_base = {
          "What is AI?": "AI stands for Artificial Intelligence, the simulation of human intelligence in machines.",
          "What is Python?": "Python is a popular high-level programming language."
       }
   def answer_question(self, question):
      return self.knowledge_base.get(question, None)
   def learn(self, question, answer):
      self.knowledge_base[question] = answer
      print("Thank you for teaching me!")
# Interactive loop

agent = LearningChatbotAgent()
print("Welcome! Ask me a question or type 'exit' to stop.")
while True:
   user_question = input("You: ").strip()
   if user_question.lower() == 'exit':
      print("Goodbye!")
      break
   response = agent.answer_question(user_question)
   if response:
      print(f"Agent: {response}")
   else:

      print("Agent: I don't know the answer. Can you teach me?")
      user_answer = input("Your answer: ").strip()
      agent.learn(user_question, user_answer)
```

Output:

```
Welcome! Ask me a question or type 'exit' to stop.
You: What is Python?
Agent: Python is a popular high-level programming language.
You: What is AI?
Agent: AI stands for Artificial Intelligence, the simulation of human intelligence in machines.
You: exit
Goodbye!
```

2. Coordinator-Worker-Delegator (CWD) Model: Simulate the CWD model where a CoordinatorAgent delegates a time-consuming task (e.g., "generating a report") to a WorkerAgent. The coordinator should show that it's waiting for the worker and prints the result upon completion.

```
import time
import threading
class WorkerAgent:
    def generate_report(self):
        print("WorkerAgent: Starting report generation...")
        time.sleep(3)  # Simulate time-consuming task
        report = "Report generated successfully!"
        print("WorkerAgent: Report generation completed.")
        return report
class CoordinatorAgent:
    def __init__(self):
        self.worker = WorkerAgent()
        self.report = None
    def delegate_task(self):
        print("CoordinatorAgent: Delegating report generation to worker and waiting for result...")
        def task():
            self.report = self.worker.generate_report()

        # Run worker task in a separate thread to simulate asynchronous work
        worker_thread = threading.Thread(target=task)
        worker_thread.start()

        # Simulate waiting while the worker finishes
        while worker_thread.is_alive():
            print("CoordinatorAgent: Waiting for worker...")
            time.sleep(1)
        worker_thread.join()

        print(f"CoordinatorAgent: Received report result -> {self.report}")
# Run simulation
coordinator = CoordinatorAgent()
coordinator.delegate_task()
```

Output:

```
CoordinatorAgent: Delegating report generation to worker and waiting for result...

WorkerAgent: Starting report generation...
CoordinatorAgent: Waiting for worker...
CoordinatorAgent: Waiting for worker...
CoordinatorAgent: Waiting for worker...
WorkerAgent: Report generation completed.
CoordinatorAgent: Received report result -> Report generated successfully!
```

WEEK- 6

1. Agent Role Assignment: Demonstrate agent role assignment in a content creation workflow. A JournalistAgent writes a raw news article (as a dictionary). An EditorAgent receives this dictionary, refines it according to its role (e.g., corrects grammar, formats title), and outputs the final version.

```python
class JournalistAgent:


    def write_article(self):
        # Simulated raw article with simple issues
        article = {
            'title': 'new discovery in space exploration',
            'content': 'scientists have found a new planet it is very exciting'
        }
        print("JournalistAgent: Raw article created.")
        return article

class EditorAgent:

    def edit_article(self, article):
        # Simple grammar correction and formatting simulation
        corrected_title = article['title'].title()
        corrected_content = article['content'].capitalize() + '.'
        refined_article = {
            'title': corrected_title,
            'content': corrected_content
        }
        print("EditorAgent: Article refined.")
        return refined_article

# Workflow simulation
journalist = JournalistAgent()
raw_article = journalist.write_article()


editor = EditorAgent()

final_article = editor.edit_article(raw_article)
print("\nFinal Article:")
print(final_article)
```

Output:

```
JournalistAgent: Raw article created.
EditorAgent: Article refined.
Final Article:
{'title': 'New Discovery In Space Exploration', 'content': 'Scientists have found a new planet it is very exciting.'}
```

2. Agent Communication for Collaboration: Show agent communication for collaboration. AgentA proposes its available meeting slots. AgentB receives the proposal, finds all common slots, and sends a confirmation response proposing the first available common slot.

```
class AgentA:

    def __init__(self, available_slots):
        self.available_slots = available_slots
    def propose_slots(self):

        print(f"AgentA: Proposing slots {self.available_slots}")
        return self.available_slots
    def  receive_confirmation(self,  confirmation):

        print(f"AgentA: Received confirmation for slot {confirmation}")
class AgentB:
    def __init__(self, available_slots):
        self.available_slots = available_slots
    def  receive_proposal(self,  slots_from_A):

        common_slots = list(set(slots_from_A) & set(self.available_slots))
        print(f"AgentB: Common available slots: {common_slots}")

        if common_slots:
            confirmation = sorted(common_slots)[0]
            print(f"AgentB: Confirming earliest slot {confirmation}")
            return confirmation
        else:
            print("AgentB: No common slots available.")
            return None
# Simulate communication

agentA = AgentA(available_slots=['10AM', '2PM', '4PM'])
agentB = AgentB(available_slots=['9AM', '2PM', '3PM'])
proposed_slots = agentA.propose_slots()
confirmation  = agentB.receive_proposal(proposed_slots)
if confirmation:
    agentA.receive_confirmation(confirmation)
```

Output:

```
AgentA: Proposing slots ['10AM', '2PM', '4PM']
AgentB: Common available slots: ['2PM']
AgentB: Confirming earliest slot 2PM
AgentA: Received confirmation for slot 2PM
```

WEEK - 7

1. Hybrid Agent Architecture: Implement a hybrid agent navigating a simple path. The deliberative layer has a plan to move forward. The reactive layer constantly checks the environment and can override the plan if it perceives an immediate obstacle, forcing a 'Dodge' action instead.

```
class Environment:


    def __init__(self, obstacles):

        self.obstacles = obstacles  # Positions with obstacles
    def is_obstacle_ahead(self, position):
        return position + 1 in self.obstacles
class HybridAgent:
    def __init__(self, env):

        self.env = env
        self.position = 0
        self.plan = ['Move Forward'] * 5  # Deliberative plan to move forward 5 steps
    def reactive_check(self):
        # Check for obstacle immediately in front

        if self.env.is_obstacle_ahead(self.position):

            return 'Dodge'
        return None
    def act(self):

        for action in self.plan:
            reactive_action = self.reactive_check()
            if reactive_action:
                print(f"Reactive override at position {self.position}: {reactive_action}")
                self.position += 1  # Dodge means skip forward safely
            else:
                print(f"Deliberative action at position {self.position}: {action}")

                elf.position += 1  # Move forward
        print(f"Final position: {self.position}")
# Example usage

env = Environment(obstacles={3})
agent = HybridAgent(env)
agent.act()
```

Output:

```
Deliberative action at position 0: Move Forward
Deliberative action at position 1: Move Forward
Reactive override at position 2: Dodge
Deliberative action at position 3: Move Forward
Deliberative action at position 4: Move Forward
Final position: 5
```

2. CWD with Reflection: Combine the CWD model with reflection. A CoordinatorAgent delegates a task (writing a draft) to a WorkerAgent. Upon receiving the draft, the coordinator uses a reflect method to check its quality (e.g., word count) and decides if it's 'Approved' or 'Needs Revision'.

```
class WorkerAgent:

    def write_draft(self):

        # Simulate draft writing

        draft = ("This is a draft article with sufficient length "
                 "to pass quality checks.")
        print("WorkerAgent: Draft completed.")
        return draft
class CoordinatorAgent:
    def __init__(self):
        self.worker = WorkerAgent()

     def reflect(self, draft):
        # Reflection criterion: draft must be at least 30 words
        word_count = len(draft.split())
         if word_count >= 30:

            quality = 'Approved'
        else:
            quality = 'Needs Revision'
        print(f"CoordinatorAgent Reflection: Word count = {word_count}, Quality = {quality}")
        return quality
    def delegate_and_reflect(self):
        print("CoordinatorAgent: Delegating draft writing...")
        draft = self.worker.write_draft()
        quality = self.reflect(draft)
        return quality, draft
# Simulation

coordinator = CoordinatorAgent()
quality_status, draft_text = coordinator.delegate_and_reflect()
print(f"Final decision: {quality_status}")
print(f"Draft text: {draft_text}")
```

Output:

```
CoordinatorAgent: Delegating draft writing...
WorkerAgent: Draft completed.
CoordinatorAgent Reflection: Word count = 12, Quality = Needs Revision
Final decision: Needs Revision
Draft text: This is a draft article with sufficient length to pass quality checks.
```

WEEK-8

1. CWD with Multiple Workers (Round-Robin): Write a Python program to simulate the CWD model with one coordinator and multiple workers. The CoordinatorAgent should have a list of tasks and two WorkerAgents. The coordinator must delegate the tasks one by one to the workers in a round-robin fashion (first task to Worker 1, second to Worker 2, third to Worker 1, etc.).

```
class WorkerAgent:

    def __init__(self, name):

        self.name = name
    def do_task(self, task):
        print(f"{self.name} is processing task: {task}")
class CoordinatorAgent:
    def __init__(self, tasks, workers):

        self.tasks = tasks
        self.workers = workers
    def delegate_tasks(self):

        print("=== CWD Round-Robin Task Delegation ===")
        worker_count = len(self.workers)
         for i, task in enumerate(self.tasks):
            worker = self.workers[i % worker_count]  # round-robin
            print(f"Coordinator delegating '{task}' to {worker.name}")
            worker.do_task(task)
# Demo
if __name___== "__main__":
    tasks = ["Task-1", "Task-2", "Task-3", "Task-4", "Task-5"]
    worker1 = WorkerAgent("Worker 1")
     worker2 = WorkerAgent("Worker 2")
    coordinator = CoordinatorAgent(tasks, [worker1, worker2])

    coordinator.delegate_tasks()
```

Output:

```
Coordinator delegating 'Task-1' to Worker 1
Worker 1 is processing task: Task-1
Coordinator delegating 'Task-2' to Worker 2
Worker 2 is processing task: Task-2
Coordinator delegating 'Task-3' to Worker 1
Worker 1 is processing task: Task-3
Coordinator delegating 'Task-4' to Worker 2
Worker 2 is processing task: Task-4
Coordinator delegating 'Task-5' to Worker 1
Worker 1 is processing task: Task-5
```

Vasavi College of Engineering (A)

(Affiliated to Osmania University) Hyderabad-500031

Department Name: CSE

Name Of the Laboratory: AAI

Name: Roll Number: Page no:

2. Agent Role Assignment (Client-Server): Write a Python program to demonstrate agent role assignment using a client-server model. Create a ClientAgent with a send_request method and a ServerAgent with a process_request method. The client's request should

```
class ServerAgent:

    def process_request(self, request):
        print(f"ServerAgent: received request -> {request}")
        # Simple "processing": convert to upper case
        response = f"Processed: {request.upper()}"
        print("ServerAgent: sending response back to client")
        return response
class ClientAgent:

    def __init__(self, server):
        self.server = server
    def send_request(self, request):

        print("=== Client-Server Interaction ===")
        print(f"ClientAgent: sending request -> {request}")
        response = self.server.process_request(request)
        print(f"ClientAgent: got response -> {response}")
# Demo

if __name___== "__main__":

    server = ServerAgent()
    client = ClientAgent(server)
    client.send_request("get current status of system")
```

Output:

```
ClientAgent: sending request -> get current status of system
ServerAgent: received request -> get current status of system
ServerAgent: sending response back to client
ClientAgent: got response -> Processed: GET CURRENT STATUS OF SYSTEM
```

WEEK-9

1. MAS Communication (Broadcast): Write a Python program to simulate a one-to-many broadcast communication pattern. Create one NotifierAgent and three ReceiverAgent objects. The NotifierAgent must have a list of all receivers and an announce method that, when called, sends a message to every receiver in its list.

```
class ReceiverAgent:

    def __init__(self, name):

        self.name = name

    def receive(self, message):

        print(f"{self.name} received message: {message}")
class NotifierAgent:
    def __init__(self, receivers):

        self.receivers = receivers  # list of ReceiverAgent objects
    def announce(self, message):
        print("=== Broadcast Announcement ===")
        print(f"Notifier: broadcasting -> {message}")
        for receiver in self.receivers:
            receiver.receive(message)
# Demo
if __name___== "__main__":

    r1 = ReceiverAgent("Receiver 1")
    r2 = ReceiverAgent("Receiver 2")
    r3 = ReceiverAgent("Receiver 3")
    notifier = NotifierAgent([r1, r2, r3])
    notifier.announce("System will go down for maintenance at 10 PM.")
```

Output:

```
Notifier: broadcasting -> System will go down for maintenance at 10 PM.
Receiver 1 received message: System will go down for maintenance at 10 PM.
Receiver 2 received message: System will go down for maintenance at 10 PM.
Receiver 3 received message: System will go down for maintenance at 10 PM
```

2. CWD for a Generative Task (Simulation): Write a Python program that simulates a CWD approach for a generative task (like building a sentence). Create a CoordinatorAgent, a NounAgent (Worker), and a VerbAgent (Worker). The Coordinator must first delegate to the NounAgent to get a noun, then to the VerbAgent to get a verb, and finally, the Coordinator itself will assemble these parts into a complete sentence.

```
class NounAgent:

    def get_noun(self):
        noun = "cat"
        print(f"NounAgent: generated noun -> {noun}")
        return noun
class VerbAgent:

    def get_verb(self):
        verb = "jumps"
        print(f"VerbAgent: generated verb -> {verb}")
        return verb
class CoordinatorAgent:

    def __init__(self, noun_agent, verb_agent):
        self.noun_agent = noun_agent
        self.verb_agent = verb_agent
    def build_sentence(self):

        print("=== CWD Generative Task (Sentence Building) ===")
        noun = self.noun_agent.get_noun()

        verb = self.verb_agent.get_verb()
        sentence = f"The {noun} {verb}."
        print(f"CoordinatorAgent: assembled sentence -> {sentence}")
        return sentence
# Demo
if __name__== "__main__":
    noun_agent = NounAgent()
    verb_agent = VerbAgent()
    coordinator = CoordinatorAgent(noun_agent, verb_agent)
    final_sentence = coordinator.build_sentence()
```

Output:

```
NounAgent: generated noun -> cat
VerbAgent: generated verb -> jumps
CoordinatorAgent: assembled sentence -> The cat jumps.
```

WEEK-10

1. Agent with a Tool: Write a Python program.

- **Create a class PriceAgent.**
- **Inside its __init__, give it a "tool" which is just a dictionary: self.price_tool = {"apple": 2, "banana": 1, "orange": 3}**
- **Create a method get_price(self, item). This method uses the self.price_tool to look up the item and print the price.**
- **Test it by creating an agent and asking for the price of "apple".**

```
class PriceAgent:

    def __init__(self):

        # "Tool" is just a dictionary of prices
        self.price_tool = {
            "apple": 2,
            "banana": 1,
            "orange": 3
        }
    def get_price(self, item):

        print("=== Price Lookup ===")
        item_lower = item.lower()
        if item_lower in self.price_tool:
            price = self.price_tool[item_lower]
            print(f"PriceAgent: The price of '{item_lower}' is {price} units.")
        else:
            print(f"PriceAgent: Sorry, '{item}' is not available.")
# Demo
if __name__== "__main__":
    agent = PriceAgent()

    agent.get_price("apple")
```

Output:

```
PriceAgent: The price of 'apple' is 2 units.
```

2. RAG Orchestrator Agent: Write a Python program.

- **Create a RetrievalAgent class with one method, retrieve(topic), that just returns a fake document string (e.g., return "Apples are a red fruit.").**
- **Create a GenerationAgent class with one method, generate(prompt, context), that just prints a formatted string like: f"PROMPT: {prompt} \nCONTEXT: {context}"**
- **Create a RAGAgent class. Give it an ask(question) method.**
- **Inside ask(question), the RAGAgent must (1) create a RetrievalAgent and call retrieve() to get context, then (2) create a GenerationAgent and call generate() using the question and context.**

```
class RetrievalAgent:
    def retrieve(self, topic):
        print(f"RetrievalAgent: retrieving context for topic -> {topic}")
        # Fake document / context
        context = "Apples are a red fruit rich in fiber and vitamins."
        return context
class GenerationAgent:

    def generate(self, prompt, context):
        print("=== Generation Step ===")
        print(f"PROMPT: {prompt}")
        print(f"CONTEXT: {context}")
class RAGAgent:

    def ask(self, question):

        print("=== RAG Orchestrator ===")
        # 1. Retrieval
        retrieval_agent = RetrievalAgent()

        context = retrieval_agent.retrieve(question)
        # 2. Generation
        generation_agent = GenerationAgent()
        generation_agent.generate(prompt=question, context=context)
# Demo
if __name___== "__main__":

    rag_agent = RAGAgent()

    rag_agent.ask("What are the health benefits of apples?")
```

Output:

```
RetrievalAgent: retrieving context for topic -> What are the health benefits of apples?
PROMPT: What are the health benefits of apples?
CONTEXT: Apples are a red fruit rich in fiber and vitamins.
```

WEEK-11

1. **Stateful Agent: Write a Python program.**
   - **Create a class ChatAgent.**
   - **Inside its __init__, give it a "state" variable: self.user_name = None.**
   - **Create a respond(self, message) method.**
   - **Inside this method, if self.user_name is None, check if the message is "My name is Bob". If it is, set self.user_name = "Bob" and print "Hello Bob!".**
   - **If self.user_name *is* set, just print f"Hello again, {self.user_name}!".**
   - **Test it by calling respond("Hi"), then respond("My name is Bob"), then respond("Hi") again.**

```
class ChatAgent:
    def _init_(self):
        self.user_name = None
        self.history = []
        self.greet_count = 0
    def _remember(self, message, actor="user"):
        self.history.append((actor, message))
    def _extract_name(self, message):
        m = message.strip()
        if m.lower().startswith("my name is "):
            name = m[len("my name is "):].strip()
            if name:
                return name
        return None
    def respond(self, message):
        self._remember(message, "user")
        if self.user_name is None:

            name = self._extract_name(message)
            if name:
                self.user_name = name
                self.greet_count = 1
                reply = f"Hello {self.user_name}! Nice to meet you."
            else:
                reply = "I don't know your name yet. Please tell me."
        else:
            self.greet_count += 1

            if message.strip().lower() in ("who am i?", "who am i"):
                reply = f"You are {self.user_name}."
            elif message.strip().lower() == "reset":

                old = self.user_name
                self.user_name = None
                self.greet_count = 0
                self.history = []
                reply = f"State reset. I have forgotten {old}."
            else:
                reply = f"Hello again, {self.user_name}! (visit #{self.greet_count})"
        self._remember(reply, "agent")
        print(reply)
    def show_history(self):

        for i, (actor, text) in enumerate(self.history, 1):
            print(f"{i:02d} {actor}: {text}")
# Test run for ChatAgent
if __name__== "__main__":
    agent = ChatAgent()
    agent.respond("Hi"
    agent.respond("My name is Bob")
    agent.respond("Who am I?")
    agent.respond("Hi")
    # agent.respond("Reset")
    # agent.respond("Hi")
    print("\nConversation history:")
    agent.show_history()
```

Output:

```
I don't know your name yet. Please tell me.
Hello Bob! Nice to meet you.
You are Bob.

Hello again, Bob! (visit #3)
Conversation history:
01 user: Hi

02 agent: I don't know your name yet. Please tell me.
03 user: My name is Bob
04 agent: Hello Bob! Nice to meet you.
05 user: Who am I?
06 agent: You are Bob.
07 user: Hi
08 agent: Hello again, Bob! (visit #3)
```

2. Router Agent (Graph Logic): Write a Python program.

- **Create a WeatherAgent class with a get_weather() method that prints "It is sunny."**
- **Create a MathAgent class with a do_math() method that prints "2 + 2 = 4."**
- **Create a RouterAgent class with a handle_query(self, query) method.**
- **nside handle_query, if the query contains "weather", create and use the WeatherAgent. If the query contains "math", create and use the MathAgent. Otherwise, print "I can't help with that."**
- **Test it by calling handle_query("What is the weather?") and handle_query("Do some math.").**

```
class WeatherAgent:
    def get_weather(self, location="your area"):
        print(f"Weather report for {location}:")
        print("Condition: Sunny")
        print("Temperature: 28°C")
        print("Forecast: Clear skies for the next 3 days")
class MathAgent:
    def do_math(self, expression=None):
        if not expression:
            print("2 + 2 = 4.")
            return
        try:
            # A safe evaluation environment
            safe_env = {"__builtins__": None}
            result = eval(expression, safe_env, {})
            print(f"{expression} = {result}")
        except Exception as e:
            print("ERROR!")
            print("Could not compute expression. Use simple arithmetic like 2+3*4. Error:", e)
class RouterAgent:

    def _init_(self):

        self.routing_table = {

            “weather": WeatherAgent,
            "forecast": WeatherAgent,
            "math": MathAgent,
            "calculate": MathAgent,
            "compute": MathAgent
        }

    def _find_domain(self, query):

       q = query.lower()
        for key in self.routing_table:
            if key in q:
                return key
        return None

    def _extract_math_expression(self, query):
        # split after colon: "calculate: 12/3 + 5"
        if ":" in query:
            return query.split(":")[1].strip()
        words = query.lower().split()
        if "calculate" in words:
            idx = words.index("calculate")
            return " ".join(query.split()[idx + 1:])
        if "compute" in words:

            idx = words.index("compute")
            return " ".join(query.split()[idx + 1:])
        return None

    def handle_query(self, query):
        domain_key = self._find_domain(query)

        if domain_key is None:
            print("I can't help with that.")
            return
        AgentClass = self.routing_table[domain_key]
        agent = AgentClass()
        if AgentClass is WeatherAgent:
            location = "your area"
            tokens = query.split()
            if "in" in tokens:
                idx = tokens.index("in")
                if idx + 1 < len(tokens):
                    location = tokens[idx + 1]
            agent.get_weather(location)
        elif AgentClass is MathAgent:
            expr = self._extract_math_expression(query)
            agent.do_math(expr)
# Test run
if __name__== "__main__":
    router = RouterAgent()
    router.handle_query("What is the weather today?")
    print()
    router.handle_query("Do some math: 12/3 + 5")
    print()
    router.handle_query("Please calculate 2 ** 3")
    print()
    router.handle_query("Tell me a joke.")
```

Output:

```
Weather report for your area:
Condition: Sunny
Temperature: 28°C
Forecast: Clear skies for the next 3 days
12/3 + 5 = 9.0
2 ** 3 = 8
I can't help with that.
```

WEEK - 12

Build a simple code review loop using LangGraph where the system generates code, reviews it, improves it, and performs one extra review pass before stopping.
Steps to Follow:
1. Create three agents
Developer that writes simple Python code
Reviewer that gives feedback
Improver that updates the code
2. Set up a shared state, Store code, review text, and iteration count
3. Build a LangGraph workflow, Add the three agents as nodes
4. Connect the nodes, Developer to Reviewer, Reviewer to Improver
5. Add a loop, After the Improver runs, send it back to Reviewer once.

```
from typing import TypedDict
from langgraph.graph import StateGraph, END

# 1. Define the State
class GraphState(TypedDict):
    code: str
    review: str
    iterations: int

# 2. Define the Agents (Nodes)
def developer(state: GraphState):
    print("\n[Developer]: Writing initial code...")
    return {"code": "print('Hello World')", "iterations": 0}

def reviewer(state: GraphState):
    code = state["code"]
    print(f"\n[Reviewer]: Reviewing code: {code}")
    if "improved" in code:
        review = "Code looks great now. Approved."
    else:
        review = "Missing functionality. Needs improvement."
    return {"review": review}

def improver(state: GraphState):
    print("\n[Improver]: Improving code based on review...")
    new_code = state["code"].replace(")", " improved)")
    # Increment iteration count
    return {"code": new_code, "iterations": state["iterations"] + 1}

# 3. Define Conditional Logic
def should_continue(state: GraphState):
    # Loop back to reviewer only once (when iterations is 1)
    # Flow so far: Dev(0) -> Rev -> Imp(1).
    # We want one extra review pass.
    if state["iterations"] <= 1:
        print("-> Loop back to Reviewer")
        return "reviewer"
    return END
```

```
# 4. Build the Graph
workflow = StateGraph(GraphState)
# Add nodes
workflow.add_node("developer", developer)
workflow.add_node("reviewer", reviewer)
workflow.add_node("improver", improver)

# Connect nodes
workflow.set_entry_point("developer")
workflow.add_edge("developer", "reviewer")
workflow.add_edge("reviewer", "improver")

# Add conditional edge from Improver
workflow.add_conditional_edges(
    "improver",
    should_continue,
    {
        "reviewer": "reviewer",
        END: END
    }
)
# 5. Compile and Run
app = workflow.compile()

# Execute
print("--- Starting Graph ---")
result = app.invoke({"code": "", "review": "", "iterations": 0})
print("\n--- Final State ---")
print(result)
```

OUTPUT:

```
--- Starting Graph ---

[Developer]: Writing initial code...

[Reviewer]: Reviewing code: print('Hello World')

[Improver]: Improving code based on review...
-> Loop back to Reviewer

[Reviewer]: Reviewing code: print('Hello World improved')

[Improver]: Improving code based on review...

--- Final State ---
{'code': "print('Hello World improved improved')", 'review': 'Code looks great now. Approved.', 'iterations': 2}
```