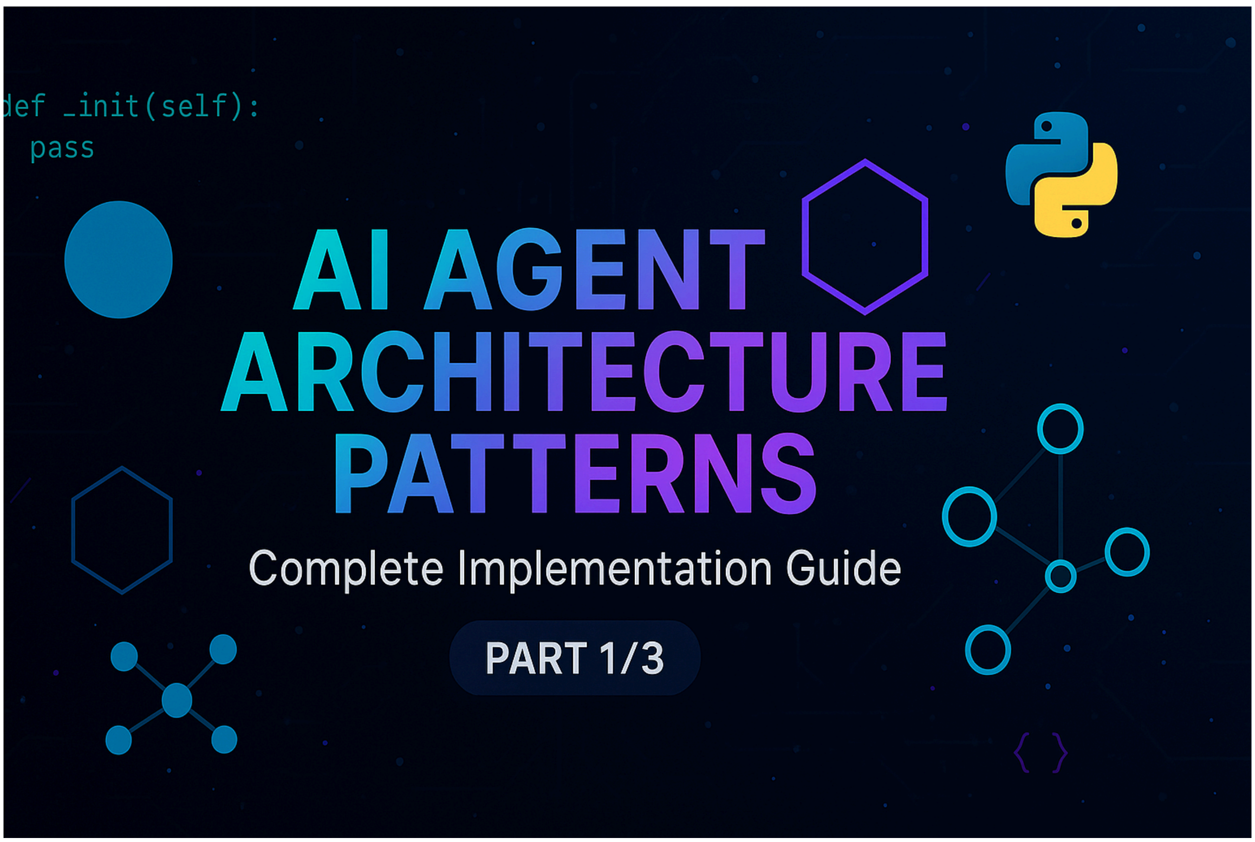
# AI Agent Architecture Patterns - Complete Implementation Guide - Part - 1/3

| Type | @datasciencebrain |
| --- | --- |




## Table of Contents

Part 1:

Part 2:

1. Multi-Agent Systems
2. Modular Agents
3. Learning Agents

Part 3:

1. LLM-Based Agents
2. Advanced Patterns

---

# Reactive Agents

**Definition**: Simple agents that respond directly to environmental stimuli using condition-action rules without complex reasoning or memory.

**Key Characteristics**:

- No internal state or world model
- Fast, immediate responses
- Rule-based behavior
- Suitable for predictable environments

## Basic Implementation

```
from abc import ABC, abstractmethod
from typing import Dict, Any, List

class ReactiveAgent:
    """Basic reactive agent implementation"""

    def __init__(self, name: str):
        self.name = name
        self.rules = []

    def add_rule(self, condition_func, action_func):
```

```python
        """Add a condition-action rule"""
        self.rules.append((condition_func, action_func))

    def perceive(self, environment: Dict[str, Any]) -> Dict[str, Any]:
        """Perceive the current environment state"""
        return environment

    def act(self, perception: Dict[str, Any]) -> str:
        """Execute action based on perception"""
        for condition, action in self.rules:
            if condition(perception):
                return action(perception)
        return "no_action"

# Example: Simple chatbot
class ChatbotAgent(ReactiveAgent):
    def __init__(self):
        super().__init__("Chatbot")
        self._setup_rules()

    def _setup_rules(self):
        # Greeting rule
        self.add_rule(
            lambda p: any(word in p.get('message', '').lower()
                         for word in ['hello', 'hi', 'hey']),
            lambda p: "Hello! How can I help you today?"
        )

        # Help rule
        self.add_rule(
            lambda p: 'help' in p.get('message', '').lower(),
            lambda p: "I can assist you with basic questions. What do you need help with?"
        )

        # Default rule
```

```
        self.add_rule(
            lambda p: True,
            lambda p: "I'm sorry, I don't understand. Can you rephrase?"
        )

# Usage example
chatbot = ChatbotAgent()
response = chatbot.act({"message": "Hello there!"})
print(response)  # Output: "Hello! How can I help you today?"
```

## Advanced Reactive Agent with State Machines

```
from enum import Enum
from dataclasses import dataclass
from typing import Callable, Optional

class AgentState(Enum):
    IDLE = "idle"
    PROCESSING = "processing"
    RESPONDING = "responding"
    ERROR = "error"

@dataclass
class Transition:
    from_state: AgentState
    to_state: AgentState
    condition: Callable
    action: Callable

class StateMachineReactiveAgent:
    """Reactive agent with finite state machine"""

    def __init__(self, initial_state: AgentState = AgentState.IDLE):
        self.current_state = initial_state
        self.transitions: List[Transition] = []
```

```
        self.context = {}

    def add_transition(self, from_state: AgentState, to_state: AgentState,
                      condition: Callable, action: Callable):
        """Add state transition rule"""
        self.transitions.append(Transition(from_state, to_state, condition, action))

    def process(self, input_data: Dict[str, Any]) → Optional[str]:
        """Process input and potentially transition state"""
        for transition in self.transitions:
            if (transition.from_state == self.current_state and
                transition.condition(input_data, self.context)):

                result = transition.action(input_data, self.context)
                self.current_state = transition.to_state
                return result
        return None

# Example: Customer service agent
class CustomerServiceAgent(StateMachineReactiveAgent):
    def __init__(self):
        super().__init__(AgentState.IDLE)
        self._setup_transitions()

    def _setup_transitions(self):
        # IDLE → PROCESSING when message received
        self.add_transition(
            AgentState.IDLE, AgentState.PROCESSING,
            lambda inp, ctx: inp.get('message') is not None,
            self._process_message
        )

        # PROCESSING → RESPONDING when processed
        self.add_transition(
            AgentState.PROCESSING, AgentState.RESPONDING,
            lambda inp, ctx: ctx.get('processed') is True,
```

```
            self._generate_response
        )

        # RESPONDING → IDLE after response
        self.add_transition(
            AgentState.RESPONDING, AgentState.IDLE,
            lambda inp, ctx: ctx.get('responded') is True,
            self._reset_context
        )

    def _process_message(self, input_data, context):
        message = input_data['message'].lower()
        if 'complaint' in message:
            context['type'] = 'complaint'
        elif 'question' in message:
            context['type'] = 'question'
        else:
            context['type'] = 'general'
        context['processed'] = True
        return f"Processing {context['type']}..."

    def _generate_response(self, input_data, context):
        response_map = {
            'complaint': "I understand your concern. Let me help resolve this issu
e.",
            'question': "I'd be happy to answer your question.",
            'general': "Thank you for contacting us. How can I assist you?"
        }
        context['responded'] = True
        return response_map.get(context['type'], "How can I help you?")

    def _reset_context(self, input_data, context):
        context.clear()
        return "Ready for next interaction."
```

## Deliberative Agents

**Definition**: Agents that use internal models of the world to plan sequences of actions before execution. They employ symbolic reasoning and explicit goal management.

**Key Characteristics**:

- Internal world model
- Goal-oriented planning
- Symbolic reasoning
- Explicit knowledge representation

### Basic Planning Agent

```
from typing import List, Set, Dict, Tuple
from dataclasses import dataclass
from queue import PriorityQueue
import heapq

@dataclass
class State:
    """Represents a world state"""
    variables: Dict[str, Any]

    def __eq__(self, other):
        return self.variables == other.variables

    def __hash__(self):
        return hash(tuple(sorted(self.variables.items())))

@dataclass
class Action:
    """Represents an action with preconditions and effects"""
    name: str
    preconditions: Dict[str, Any]
```

```
    effects: Dict[str, Any]
    cost: float = 1.0

    def is_applicable(self, state: State) → bool:
        """Check if action can be executed in given state"""
        for var, value in self.preconditions.items():
            if state.variables.get(var) != value:
                return False
        return True

    def apply(self, state: State) → State:
        """Apply action to state, returning new state"""
        new_vars = state.variables.copy()
        new_vars.update(self.effects)
        return State(new_vars)

class DeliberativeAgent:
    """Agent that plans before acting"""

    def __init__(self, name: str):
        self.name = name
        self.actions: List[Action] = []
        self.current_state = State({})
        self.goals: Dict[str, Any] = {}

    def add_action(self, action: Action):
        """Add available action to agent's repertoire"""
        self.actions.append(action)

    def set_goal(self, goals: Dict[str, Any]):
        """Set the agent's goals"""
        self.goals = goals

    def is_goal_satisfied(self, state: State) → bool:
        """Check if current goals are satisfied"""
        for var, value in self.goals.items():
```

```
            if state.variables.get(var) != value:
                return False
        return True

    def plan(self, initial_state: State, goal_state: Dict[str, Any]) → List[Action]:
        """A* search algorithm for planning"""
        frontier = PriorityQueue()
        frontier.put((0, 0, initial_state, []))  # (f_score, g_score, state, path)
        visited = set()

        while not frontier.empty():
            f_score, g_score, current_state, path = frontier.get()

            if hash(current_state) in visited:
                continue
            visited.add(hash(current_state))

            # Check if goal is reached
            goal_satisfied = True
            for var, value in goal_state.items():
                if current_state.variables.get(var) != value:
                    goal_satisfied = False
                    break

            if goal_satisfied:
                return path

            # Expand successors
            for action in self.actions:
                if action.is_applicable(current_state):
                    new_state = action.apply(current_state)
                    new_path = path + [action]
                    new_g_score = g_score + action.cost
                    h_score = self._heuristic(new_state, goal_state)
                    new_f_score = new_g_score + h_score
```

```
                frontier.put((new_f_score, new_g_score, new_state, new_path))

        return []  # No plan found

    def _heuristic(self, state: State, goal: Dict[str, Any]) -> float:
        """Heuristic function for A* search"""
        unsatisfied = sum(1 for var, value in goal.items()
                         if state.variables.get(var) != value)
        return unsatisfied

    def execute_plan(self, plan: List[Action]) -> List[str]:
        """Execute a sequence of actions"""
        results = []
        for action in plan:
            if action.is_applicable(self.current_state):
                self.current_state = action.apply(self.current_state)
                results.append(f"Executed: {action.name}")
            else:
                results.append(f"Failed to execute: {action.name}")
                break
        return results

# Example: Blocks World Problem
class BlocksWorldAgent(DeliberativeAgent):
    def __init__(self):
        super().__init__("BlocksWorld")
        self._setup_actions()

    def _setup_actions(self):
        # Move block A from B to table
        self.add_action(Action(
            name="unstack_A_from_B",
            preconditions={"on_A": "B", "clear_A": True, "holding": None},
            effects={"on_A": "table", "clear_B": True, "holding": "A"},
            cost=1.0
        ))
```

```
        # Put block A on table
        self.add_action(Action(
            name="put_A_on_table",
            preconditions={"holding": "A"},
            effects={"on_A": "table", "holding": None},
            cost=1.0
        ))

        # Stack A on B
        self.add_action(Action(
            name="stack_A_on_B",
            preconditions={"holding": "A", "clear_B": True},
            effects={"on_A": "B", "clear_B": False, "holding": None},
            cost=1.0
        ))

# Usage example
agent = BlocksWorldAgent()
initial = State({
    "on_A": "B", "on_B": "table", "clear_A": True,
    "clear_B": False, "holding": None
})
goal = {"on_A": "table", "on_B": "table"}

plan = agent.plan(initial, goal)
for action in plan:
    print(f"Action: {action.name}")
```

## Knowledge-Based Deliberative Agent

```
from typing import Set, Union
import re

class KnowledgeBase:
```

```
    """Simple forward-chaining inference engine"""

    def __init__(self):
        self.facts: Set[str] = set()
        self.rules: List[Tuple[List[str], str]] = []

    def add_fact(self, fact: str):
        """Add a fact to the knowledge base"""
        self.facts.add(fact)

    def add_rule(self, conditions: List[str], conclusion: str):
        """Add an inference rule"""
        self.rules.append((conditions, conclusion))

    def infer(self) → Set[str]:
        """Forward chaining inference"""
        new_facts = set()
        changed = True

        while changed:
            changed = False
            for conditions, conclusion in self.rules:
                if conclusion not in self.facts and conclusion not in new_facts:
                    if all(cond in self.facts or cond in new_facts for cond in condition
s):
                        new_facts.add(conclusion)
                        changed = True

        self.facts.update(new_facts)
        return new_facts

    def query(self, fact: str) → bool:
        """Query if a fact can be derived"""
        return fact in self.facts

class ReasoningAgent(DeliberativeAgent):
```

```
    """Agent with knowledge-based reasoning"""

    def __init__(self, name: str):
        super().__init__(name)
        self.kb = KnowledgeBase()

    def add_knowledge(self, facts: List[str], rules: List[Tuple[List[str], str]]):
        """Add facts and rules to knowledge base"""
        for fact in facts:
            self.kb.add_fact(fact)
        for conditions, conclusion in rules:
            self.kb.add_rule(conditions, conclusion)

    def reason_and_plan(self, query: str) → Tuple[bool, List[str]]:
        """Reason about query and plan actions if needed"""
        # First, try inference
        new_facts = self.kb.infer()

        if self.kb.query(query):
            return True, [f"Inferred: {fact}" for fact in new_facts]

        # If inference fails, plan actions to achieve query
        return False, ["Need to gather more information or take action"]

# Example: Medical diagnosis agent
class MedicalDiagnosisAgent(ReasoningAgent):
    def __init__(self):
        super().__init__("MedicalDiagnosis")
        self._setup_medical_knowledge()

    def _setup_medical_knowledge(self):
        facts = [
            "patient_has_fever",
            "patient_has_cough",
            "patient_has_fatigue"
        ]
```

```
        rules = [
            (["patient_has_fever", "patient_has_cough"], "possible_flu"),
            (["possible_flu", "patient_has_fatigue"], "likely_flu"),
            (["patient_has_fever", "patient_has_fatigue"], "possible_infection"),
            (["likely_flu"], "recommend_rest_and_fluids"),
            (["possible_infection"], "recommend_blood_test")
        ]

        self.add_knowledge(facts, rules)

# Usage
med_agent = MedicalDiagnosisAgent()
result, reasoning = med_agent.reason_and_plan("recommend_rest_and_fluid
s")
print(f"Can recommend rest and fluids: {result}")
for step in reasoning:
    print(f"- {step}")
```

---

# Hybrid Agents

**Definition**: Agents that combine reactive and deliberative approaches in a layered architecture, balancing quick responses with thoughtful planning.

**Key Characteristics**:

- Layered architecture (reactive, tactical, strategic)
- Different time scales for different layers
- Interruption and override mechanisms
- Balance between speed and deliberation

## Three-Layer Architecture

```
from threading import Thread, Event, Lock
import time
```

```
from queue import Queue, Empty
from enum import Enum

class Priority(Enum):
    LOW = 1
    MEDIUM = 2
    HIGH = 3
    CRITICAL = 4

@dataclass
class Task:
    name: str
    priority: Priority
    data: Dict[str, Any]
    timestamp: float = time.time()

class ReactiveLayer:
    """Handles immediate, reflexive responses"""

    def __init__(self):
        self.reflexes = {}
        self.active = True

    def add_reflex(self, trigger, response):
        """Add stimulus-response reflex"""
        self.reflexes[trigger] = response

    def process(self, stimulus) -> Optional[str]:
        """Process stimulus and return immediate response if applicable"""
        for trigger, response in self.reflexes.items():
            if trigger(stimulus):
                return response(stimulus)
        return None

class PlanningLayer:
    """Handles tactical planning and goal management"""
```

```
    def __init__(self):
        self.goals = []
        self.current_plan = []
        self.planning_active = False

    def set_goals(self, goals: List[str]):
        self.goals = goals

    def plan(self, current_state: Dict[str, Any]) -> List[str]:
        """Generate plan to achieve goals"""
        self.planning_active = True
        # Simplified planning logic
        plan = []
        for goal in self.goals:
            if goal not in current_state.get('achieved', []):
                plan.append(f"work_towards_{goal}")
        self.current_plan = plan
        self.planning_active = False
        return plan

    def get_next_action(self) -> Optional[str]:
        """Get next planned action"""
        if self.current_plan:
            return self.current_plan.pop(0)
        return None

class StrategicLayer:
    """Handles long-term strategy and meta-level reasoning"""

    def __init__(self):
        self.strategies = []
        self.meta_goals = []

    def evaluate_performance(self, metrics: Dict[str, float]) -> Dict[str, Any]:
        """Evaluate agent performance and adjust strategies"""
```

```
        recommendations = {}

        if metrics.get('success_rate', 0) < 0.7:
            recommendations['action'] = 'revise_planning_strategy'

        if metrics.get('response_time', 0) > 5.0:
            recommendations['action'] = 'optimize_reactive_layer'

        return recommendations

    def adapt_strategy(self, feedback: Dict[str, Any]):
        """Adapt long-term strategy based on feedback"""
        # Strategy adaptation logic
        pass

class HybridAgent:
    """Three-layer hybrid architecture agent"""

    def __init__(self, name: str):
        self.name = name
        self.reactive_layer = ReactiveLayer()
        self.planning_layer = PlanningLayer()
        self.strategic_layer = StrategicLayer()

        self.input_queue = Queue()
        self.output_queue = Queue()
        self.state = {"achieved": [], "metrics": {}}
        self.running = False
        self.lock = Lock()

    def start(self):
        """Start the hybrid agent"""
        self.running = True
        self.reactive_thread = Thread(target=self._reactive_loop)
        self.planning_thread = Thread(target=self._planning_loop)
        self.strategic_thread = Thread(target=self._strategic_loop)
```

```
        self.reactive_thread.start()
        self.planning_thread.start()
        self.strategic_thread.start()

    def stop(self):
        """Stop the hybrid agent"""
        self.running = False

    def process_input(self, task: Task):
        """Add task to input queue"""
        self.input_queue.put(task)

    def get_output(self) → Optional[str]:
        """Get output from agent"""
        try:
            return self.output_queue.get_nowait()
        except Empty:
            return None

    def _reactive_loop(self):
        """Reactive layer processing loop"""
        while self.running:
            try:
                task = self.input_queue.get(timeout=0.1)

                # Check for immediate reflexive response
                response = self.reactive_layer.process(task)
                if response:
                    self.output_queue.put(f"REACTIVE: {response}")
                    self._update_metrics('reactive_response', 1)
                else:
                    # Pass to planning layer if no immediate response
                    self.input_queue.put(task)  # Put back for planning

            except Empty:
```

```
                continue

    def _planning_loop(self):
        """Planning layer processing loop"""
        while self.running:
            try:
                # Check for next planned action
                action = self.planning_layer.get_next_action()
                if action:
                    self.output_queue.put(f"PLANNED: {action}")
                    self._update_metrics('planned_action', 1)

                # Replan if needed
                if not self.planning_layer.current_plan and self.planning_layer.goals:
                    with self.lock:
                        self.planning_layer.plan(self.state)

                time.sleep(1)  # Planning operates on longer timescale

            except Exception as e:
                print(f"Planning error: {e}")

    def _strategic_loop(self):
        """Strategic layer processing loop"""
        while self.running:
            try:
                # Evaluate performance periodically
                with self.lock:
                    recommendations = self.strategic_layer.evaluate_performance(
                        self.state.get('metrics', {}))

                    if recommendations:
                        self.output_queue.put(f"STRATEGIC: {recommendations}")

                time.sleep(10)  # Strategic operates on even longer timescale
```

```
            except Exception as e:
                print(f"Strategic error: {e}")

    def _update_metrics(self, metric: str, value: float):
        """Update performance metrics"""
        with self.lock:
            if 'metrics' not in self.state:
                self.state['metrics'] = {}
            self.state['metrics'][metric] = self.state['metrics'].get(metric, 0) + valu
e

# Example: Autonomous Vehicle Agent
class AutonomousVehicleAgent(HybridAgent):
    def __init__(self):
        super().__init__("AutonomousVehicle")
        self._setup_layers()

    def _setup_layers(self):
        # Reactive layer - immediate safety responses
        self.reactive_layer.add_reflex(
            lambda stimulus: stimulus.data.get('obstacle_distance', 100) < 5,
            lambda stimulus: "EMERGENCY_BRAKE"
        )

        self.reactive_layer.add_reflex(
            lambda stimulus: stimulus.data.get('traffic_light') == 'red',
            lambda stimulus: "STOP"
        )

        # Planning layer - route planning and navigation
        self.planning_layer.set_goals(['reach_destination', 'follow_traffic_rules'])

# Usage example
vehicle = AutonomousVehicleAgent()
vehicle.start()
```

```
# Simulate inputs
emergency_task = Task("obstacle", Priority.CRITICAL, {"obstacle_distance":
3})
vehicle.process_input(emergency_task)

# Get response
response = vehicle.get_output()
print(response)  # Should be "REACTIVE: EMERGENCY_BRAKE"

time.sleep(1)
vehicle.stop()
```

## Subsumption Architecture

```
class Behavior:
    """Base class for behaviors in subsumption architecture"""

    def __init__(self, name: str, priority: int):
        self.name = name
        self.priority = priority
        self.active = True
        self.suppressed = False

    def should_activate(self, sensors: Dict[str, Any]) -> bool:
        """Check if behavior should be active"""
        return True

    def generate_output(self, sensors: Dict[str, Any]) -> Optional[Dict[str, Any]]:
        """Generate behavior output"""
        raise NotImplementedError

    def suppress(self):
        """Suppress this behavior"""
        self.suppressed = True
```

```
    def release(self):
        """Release suppression"""
        self.suppressed = False

class SubsumptionAgent:
    """Agent using subsumption architecture"""

    def __init__(self, name: str):
        self.name = name
        self.behaviors: List[Behavior] = []
        self.sensors = {}

    def add_behavior(self, behavior: Behavior):
        """Add behavior, maintaining priority order"""
        self.behaviors.append(behavior)
        self.behaviors.sort(key=lambda b: b.priority, reverse=True)

    def update_sensors(self, sensor_data: Dict[str, Any]):
        """Update sensor readings"""
        self.sensors.update(sensor_data)

    def step(self) -> Dict[str, Any]:
        """Execute one step of the subsumption architecture"""
        outputs = {}

        # Process behaviors in priority order
        for behavior in self.behaviors:
            if (behavior.active and not behavior.suppressed and
                behavior.should_activate(self.sensors)):

                output = behavior.generate_output(self.sensors)
                if output:
                    # Higher priority behaviors can suppress lower ones
                    for lower_behavior in self.behaviors:
                        if lower_behavior.priority < behavior.priority:
                            lower_behavior.suppress()
```

```
                outputs[behavior.name] = output
                break  # Take action from highest priority active behavior

        # Release suppression for next iteration
        for behavior in self.behaviors:
            behavior.release()

        return outputs

# Example behaviors for a robot
class AvoidObstacle(Behavior):
    def __init__(self):
        super().__init__("avoid_obstacle", priority=100)

    def should_activate(self, sensors):
        return sensors.get('obstacle_distance', 100) < 20

    def generate_output(self, sensors):
        return {"action": "turn_away", "speed": 0}

class FollowWall(Behavior):
    def __init__(self):
        super().__init__("follow_wall", priority=50)

    def should_activate(self, sensors):
        return 10 < sensors.get('wall_distance', 100) < 30

    def generate_output(self, sensors):
        return {"action": "follow_wall", "speed": 0.5}

class Wander(Behavior):
    def __init__(self):
        super().__init__("wander", priority=10)

    def should_activate(self, sensors):
```

```python
        return True  # Always active as lowest priority

    def generate_output(self, sensors):
        return {"action": "move_forward", "speed": 1.0}

# Usage
robot = SubsumptionAgent("Explorer")
robot.add_behavior(AvoidObstacle())
robot.add_behavior(FollowWall())
robot.add_behavior(Wander())

# Simulate robot operation
robot.update_sensors({"obstacle_distance": 15, "wall_distance": 25})
action = robot.step()
print(action)  # Will show obstacle avoidance behavior
```