Aim

The aim is to model a call center as an RL environment where states represent queue conditions (e.g., number of pending calls and agent availability), actions assign agents to specific call types or queues, and rewards are negative handling times. Through Monte Carlo simulations, the algorithm learns an optimal policy that minimizes average call handling time by balancing load and prioritizing urgent calls.

Algorithm

Monte Carlo policy control follows generalized policy iteration:

1. Generate episodes using the current policy π, recording state-action-reward trajectories until terminal states (all calls handled).
2. For each state-action pair, compute returns (negative sum of handling times) and update Q-values as averages:
3. $\mathrm{Q(s,a)} \leftarrow \mathrm{Q(s,a)} + 1\mathrm{N(s,a)}(\mathrm{Gt} - \mathrm{Q(s,a)})$
4. $Q(s,a) \leftarrow Q(s,a) +$
5. $N(s,a)$
6. 1
7. $(G$
8. t
9. $-Q(s,a))$, where
10. Gt
11. G
12. t
13. is the return and
14. $\mathrm{N(s,a)}$
15. $N(s,a)$ is visit count.
16. Improve policy greedily:
17. $\pi(\mathrm{s}) \leftarrow \mathrm{argmax}_a \mathrm{Q(s,a)}$
18. $\pi(s) \leftarrow \mathrm{argmax}$
19. a
20. $Q(s,a)$.
21. Repeat until policy stabilizes, using exploring starts or ε-soft policies for coverage.

Code

python

```
import numpy as np
import random
from collections import defaultdict, namedtuple

Episode = namedtuple('Episode', ['states', 'actions',
'rewards'])

class CallCenterEnv:
    def __init__(self, num_agents=5, max_queue=20,
call_types=2):
        self.num_agents = num_agents
        self.max_queue = max_queue
        self.call_types = call_types
        self.reset()

    def reset(self):
        self.queue = np.zeros(self.call_types)  # Pending calls
per type
        self.agents_busy = np.zeros(self.num_agents)  # Busy time
left
        self.state = (tuple(self.queue),
tuple(self.agents_busy))
        return self.state

    def step(self, action):  # action: agent_id to assign,
call_type
        agent_id, call_type = action
        if self.queue[call_type] > 0 and
self.agents_busy[agent_id] == 0:
            handle_time = np.random.exponential(5) + 3  # Avg 8
min handling
            self.queue[call_type] -= 1
```

```
            self.agents_busy[agent_id] = handle_time
        else:
            handle_time = 0

        # Update busy times
        self.agents_busy = np.maximum(self.agents_busy - 1, 0)

        # New arrivals
        for ct in range(self.call_types):
            if random.random() < 0.2:  # Arrival prob
                self.queue[ct] = min(self.queue[ct] + 1,
self.max_queue)

        reward = -handle_time  # Minimize time
        done = np.all(self.queue == 0) and
np.all(self.agents_busy == 0)
        self.state = (tuple(self.queue),
tuple(self.agents_busy))
        return self.state, reward, done

def generate_episode(env, policy):
    states, actions, rewards = [], [], []
    state = env.reset()
    while True:
        action = policy(state)
        next_state, reward, done = env.step(action)
        states.append(state)
        actions.append(action)
        rewards.append(reward)
        state = next_state
        if done:
            break
    return Episode(states, actions, rewards)

def mc_policy_control(env, num_episodes=10000, epsilon=0.1):
```

```
    Q = defaultdict(lambda: np.zeros((env.num_agents,
env.call_types)))
    N = defaultdict(lambda: np.zeros((env.num_agents,
env.call_types)))
    policy = lambda state: (random.randint(0, env.num_agents-1),
random.randint(0, env.call_types-1))

    for ep in range(num_episodes):
        env.reset()
        episode = generate_episode(env, policy)
        G = 0
        for t in reversed(range(len(episode.states))):
            state = episode.states[t]
            action = episode.actions[t]
            G = G + episode.rewards[t]
            N[state][action] += 1
            Q[state][action] += (G - Q[state][action]) /
N[state][action]
            A = np.argmax(Q[state])
            policy = lambda s: A if random.random() < (1 -
epsilon) else \
                               (random.randint(0,
env.num_agents-1), random.randint(0, env.call_types-1))

    optimal_policy = lambda state:
np.unravel_index(np.argmax(Q[state]), Q[state].shape)
    return optimal_policy, Q

# Usage
env = CallCenterEnv()
policy, Q = mc_policy_control(env)
print("Learned optimal policy ready for assignment
optimization.")[web:3][web:4]
```

This implementation simplifies the environment for demonstration; real call centers would incorporate actual arrival data and more states.[web:2]