**1. Automated Configuration: A Quick Prototype to Show How AI Agents Can Automate the Configuration of NTP Settings, Ensuring Consistent Time Across All Devices**

This prototype demonstrates an AI agent automating NTP configuration by selecting the optimal server and poll interval to minimize offset. It uses Q-learning, a reinforcement learning technique, where the agent learns from simulated offset changes (e.g., random noise representing network variability). The environment simulates NTP behavior, and the agent chooses actions (server-interval combinations) to maximize rewards (negative offset). This ensures consistent time across devices by dynamically optimizing settings, reducing manual intervention. Assumptions: 2 servers and 2 intervals (expandable); offset starts at 5 ms baseline. Limitations: Simulation-based; real NTP requires integration with system commands. Extension: Add more parameters (e.g, minpoll, maxpoll) or use real network data.

**Code**:

python

import numpy as np

class NTPEnv:

def \_\_init\_\_(self):

"""Initialize NTP environment with initial offset and configuration options."""

self.offset = 5.0 *# Initial offset in ms*

self.step\_count = 0 *# Track simulation steps*

self.servers = ["192.168.1.1", "192.168.1.2"] *# Sample NTP servers*

self.intervals = [64, 128] *# Poll intervals in seconds*

def step(self, action):

"""Simulate a step: update offset based on action (server and interval choice)."""

self.step\_count += 1

server\_idx, interval\_idx = divmod(action, len(self.intervals)) *# Decode action*

server = self.servers[server\_idx]

interval = self.intervals[interval\_idx]

*# Simulate offset change: random noise minus reduction (better for server 0)*

self.offset += np.random.normal(0, 0.1) - (0.05 if server\_idx == 0 else 0.02)

reward = -abs(self.offset) *# Negative offset as reward (minimize offset)*

done = self.step\_count >= 100 *# End after 100 steps*

return self.offset, reward, done, {"server": server, "interval": interval}

def q\_learning\_agent(env, episodes=100):

"""Train Q-learning agent to find optimal NTP configuration."""

q\_table = np.zeros((len(env.servers) \* len(env.intervals), 100)) *# Q-table for actions and states*

alpha, gamma, epsilon = 0.1, 0.9, 0.1 *# Learning parameters*

for \_ in range(episodes):

state = 0

env = NTPEnv() *# Reset environment*

while not env.step\_count >= 100:

if np.random.random() < epsilon:

action = np.random.randint(0, len(q\_table)) *# Exploration*

else:

action = np.argmax(q\_table[:, state]) *# Exploitation*

offset, reward, done, info = env.step(action)

new\_state = min(int(abs(offset) \* 10), 99) *# State based on offset*

q\_table[action, state] += alpha \* (reward + gamma \* np.max(q\_table[:, new\_state]) - q\_table[action, state])

state = new\_state

best\_action = np.argmax(q\_table[:, 0])

server\_idx, interval\_idx = divmod(best\_action, len(env.intervals))

return env.servers[server\_idx], env.intervals[interval\_idx]

*# Run and configure*

best\_server, best\_interval = q\_learning\_agent(NTPEnv())

print(f"Recommended NTP Configuration: Server={best\_server}, Poll Interval={best\_interval}s")

**How to Execute**:

1. Save the code as python3 ntp\_auto\_config.py in /home/lab\_adm/.
2. Install dependencies if needed: pip3 install numpy.
3. Run: python3 /home/lab\_adm/ntp\_auto\_config.py.
4. Expected Output: "Recommended NTP Configuration: Server=192.168.1.1, Poll Interval=64s" (may vary due to randomness; run multiple times for stability).
5. Potential Issues: If output varies, increase episodes to 500 for better convergence. For real NTP, integrate with paramiko to apply settings via SSH (requires server access).

**2. Real-time Anomaly Detection: Agentic Can AI Identify Deviations from Expected Time Synchronization Patterns, Alerting Administrators to Potential Issues Before They Impact Services?**

This prototype detects anomalies in time synchronization patterns using a Z-score calculation in a sliding window. It identifies deviations (e.g, from 5 ms offset) and alerts with z-scores, allowing proactive intervention. The detector processes real-time data, flagging potential issues like spikes. Assumptions: Window of 10 steps, threshold of 2.0 z-score. Limitations: Sensitive to window size; false positives in noisy data. Extension: Add email alerts (using smtplib) or integrate with Prometheus for monitoring.

**Code**:

python

import numpy as np

class AnomalyDetector:

def \_\_init\_\_(self, window\_size=10, threshold=2.0):

"""Initialize anomaly detector with window size and threshold."""

self.window\_size = window\_size

self.threshold = threshold

self.offsets = [] *# List to store offsets*

def update(self, offset):

"""Update with new offset and check for anomaly."""

self.offsets.append(offset)

if len(self.offsets) > self.window\_size:

self.offsets.pop(0)

if len(self.offsets) == self.window\_size:

mean = np.mean(self.offsets)

std = np.std(self.offsets)

z\_score = abs(offset - mean) / std if std > 0 else 0

if z\_score > self.threshold:

print(f"Alert: Anomaly detected! Offset={offset:.2f}ms, Z-score={z\_score:.2f}")

else:

print(f"Normal: Offset={offset:.2f}ms, Z-score={z\_score:.2f}")

detector = AnomalyDetector(window\_size=10, threshold=2.0)

for i in range(20):

offset = 5.72 + np.random.normal(0, 0.1) if i < 15 else 15.0

detector.update(offset)

**How to Execute**:

1. Save the code as python3 ntp\_anomaly\_detection.py in /home/lab\_adm/.
2. Install dependencies if needed: pip3 install numpy.
3. Run: python3 /home/lab\_adm/ntp\_anomaly\_detection.py.
4. Expected Output: "Normal" for most steps, then "Alert: Anomaly detected! Offset=15.00ms, Z-score=3.00" when the spike occurs, followed by "Normal" as the window adjusts.
5. Potential Issues: If std = 0 (all offsets identical), z-score = 0; add a small epsilon (e.g., std = np.std(self.offsets) + 1e-6) if needed. For real data, replace offset with Falcon-RX logs.

**3. Proactive Problem Resolution: A Follow-up on Whether AI Agents Could Automatically Initiate Corrective Actions, Such as Restarting NTP Services or Adjusting Clock Offsets.**

This prototype extends anomaly detection to an agent that automatically resolves issues by resetting the offset to a baseline (5.72 ms) upon detection, simulating an NTP restart. It uses the detector to flag anomalies and applies corrections, ensuring minimal service impact. Assumptions: Reset to baseline on alert. Limitations: Simulated reset; real NTP restarts require sudo access. Extension: Use os.system for real commands or integrate with Docker for safe testing.

**Code**:

python

import numpy as np

class NTPResolver:

def \_\_init\_\_(self, window\_size=10, threshold=2.0):

"""Initialize resolver with anomaly detector."""

self.detector = AnomalyDetector(window\_size, threshold)

self.offset = 5.72 *# Baseline offset in ms*

def step(self, new\_offset):

"""Process new offset and resolve if anomaly detected."""

self.detector.update(new\_offset)

if hasattr(self.detector, 'alert\_triggered') and self.detector.alert\_triggered:

print("Action: Restarting NTP service...")

self.offset = 5.72 *# Reset to baseline*

print(f"Offset reset to {self.offset:.2f}ms")

self.detector.alert\_triggered = False

else:

self.offset = new\_offset

return self.offset

class AnomalyDetector:

def \_\_init\_\_(self, window\_size=10, threshold=2.0):

"""Initialize anomaly detector."""

self.window\_size = window\_size

self.threshold = threshold

self.offsets = []

self.alert\_triggered = False

def update(self, offset):

"""Update offsets and check for anomaly."""

self.offsets.append(offset)

if len(self.offsets) > self.window\_size:

self.offsets.pop(0)

if len(self.offsets) == self.window\_size:

mean = np.mean(self.offsets)

std = np.std(self.offsets)

z\_score = abs(offset - mean) / std if std > 0 else 0

if z\_score > self.threshold:

print(f"Alert: Anomaly detected! Offset={offset:.2f}ms, Z-score={z\_score:.2f}")

self.alert\_triggered = True

else:

print(f"Normal: Offset={offset:.2f}ms, Z-score={z\_score:.2f}")

resolver = NTPResolver(window\_size=10, threshold=2.0)

for i in range(20):

offset = 5.72 + np.random.normal(0, 0.1) if i < 15 else 15.0

new\_offset = resolver.step(offset)

print(f"Current Offset: {new\_offset:.2f}ms")

**Execution**:

1. Save the code as python3 ntp\_proactive\_resolution.py in /home/lab\_adm/.
2. Install dependencies if needed: pip3 install numpy.
3. Run: python3 /home/lab\_adm/ntp\_proactive\_resolution.py.
4. Expected Output: "Normal" for most steps, then "Alert: Anomaly detected!" and "Action: Restarting NTP service. Offset reset to 5.72ms" when the anomaly occurs, followed by "Current Offset: 5.72ms".
5. Potential Issues: If std = 0, z-score = 0; add std = np.std(self.offsets) + 1e-6 if needed. For real NTP, replace the print statements with os.system("sudo systemctl restart ntp") and test on a VM.

**4. Scalable Management: A More Advanced Level of the Research is to Use Agentic AI to Manage Time Synchronization Across Large and Complex Networks, Adapting to Changes in Network Topology and Device Distribution.**

This prototype uses a multi-agent system to manage offsets across devices, adapting to topology changes by recalculating adjustments based on latency distribution. It simulates a network agent that updates topology and optimizes offsets, ensuring scalable synchronization.

**Code**:

python

import numpy as np

class NetworkAgent:

def \_\_init\_\_(self, device\_count):

self.device\_count = device\_count

self.latency\_dist = np.random.normal(20, 5, device\_count)

self.offsets = np.full(device\_count, 5.72)

def update\_topology(self, new\_count):

self.device\_count = new\_count

self.latency\_dist = np.random.normal(20, 5, new\_count)

self.offsets = np.full(new\_count, 5.72)

print(f"Topology updated: {new\_count} devices")

def optimize(self):

avg\_latency = np.mean(self.latency\_dist)

adjustment = -0.1 \* (avg\_latency - 20)

self.offsets += adjustment

print(f"Optimized offsets: Mean={np.mean(self.offsets):.2f}ms, Adjustment={adjustment:.2f}ms")

network = NetworkAgent(device\_count=5)

print("Initial offsets:", network.offsets)

network.optimize()

network.update\_topology(10)

network.optimize()

**Execution**:

1. Save the code as python3 ntp\_scalable\_management.py in /home/lab\_adm/.
2. Install dependencies if needed: pip3 install numpy.
3. Run: python3 /home/lab\_adm/ntp\_scalable\_management.py.
4. Expected Output: Prints initial offsets for 5 devices (e.g., [5.72, 5.72, ...]), optimized mean offset after adjustment, then updates to 10 devices and re-optimizes.
5. Potential Issues: Random seed not set, so outputs vary; add np.random.seed(42) for consistency. For real networks, replace np.random.normal with actual latency data from Falcon-RX logs.

**Notes**

* **Execution**: All codes run on CPU; no GPU required.
* **Real Data**: Replace sample values with Falcon-RX logs for accuracy.
* **Excursion Demo**: Save each as separate files and run sequentially, explaining outputs (e.g anomaly alerts, resets, topology adjustments).