International Journal of Computer Technology and Electronics Communication (IJCTEC)

| ISSN: 2320-0081 | www.ijctece.com ||A Peer-Reviewed, Refereed and Bimonthly Journal |

|| Volume 8, Issue 5, September – October 2025 ||

DOI: 10.15680/IJCTECE.2025.0805019

Multi-Agent Systems in AIOps: Enhancing Detection, Diagnosis, and Remediation

Dr. Rashmiranjan Pradhan

AI, Gen AI, Agentic AI Innovation leader at IBM, Bangalore, Karnataka, India.

rashmiranjan.pradhan@gmail.com

ABSTRACT: Modern IT infrastructures demand intelligent automation for efficient operations management. Artificial Intelligence for IT Operations (AIOps) leverages AI to automate and enhance IT tasks. This paper proposes a practical architecture for implementing **Multi-Agent Systems (MAS)** within AIOps, focusing on enhancing incident detection, root cause analysis, and automated remediation. By employing a collaborative ecosystem of specialized Python-based intelligent agents, we aim to provide increased autonomy, dynamic learning capabilities, and seamless cross-system collaboration. We provide a developer-friendly analysis with code snippets, real-world data scenarios inspired by healthcare and finance, and specific guidance on leveraging Python libraries and Large Language Models (LLMs) for implementation. This paper aims to equip developers with the knowledge to build and deploy a functional multi-agent AIOps system.

KEYWORDS: "AIOps," "Multi-Agent Systems (MAS), " "Agentic AI ," "Incident Detection, ," "Root Cause Analysis, " "Automated Remediation ," "Healthcare, ," "Finance ," "Python, ," "LangChain, ," "OpenAI API."

I. INTRODUCTION

The sheer volume and variety of data generated by today's complex IT systems present a significant challenge for operations teams Artificial Intelligence for IT Operations (AIOps) has emerged as a crucial approach to address this challenge by applying AI and machine learning to analyze operational data, automate tasks, and provide actionable insights.'
While current AIOps solutions offer valuable capabilities, they often lack the sophisticated reasoning and collaborative problem-solving inherent in human expert teams. Many rely on centralized AI models and rigid workflows, limiting their adaptability and ability to contextually understand and resolve novel incidents across diverse systems.

This paper advocates for the adoption of Multi-Agent Systems (MAS) within the AIOps framework. We propose a decentralized architecture where multiple autonomous agents, developed primarily in Python, work together to achieve AIOps objectives. Each agent is designed with specific expertise and responsibilities, enabling a more modular, scalable, and resilient approach to IT operations.

We contend that MAS can significantly enhance core AIOps capabilities by leveraging Agentic AI principles:

- Enhanced Incident Detection: Python-based agents can actively monitor diverse data streams using specialized libraries and trigger autonomous investigation workflows based on learned patterns and anomalies.
- Improved Root Cause Analysis: Agents equipped with reasoning capabilities (potentially powered by LLMs) can analyze logs, metrics, and traces from various systems, collaboratively forming and testing hypotheses to pinpoint the underlying causes of incidents.
- Automated Remediation: Python agents can execute scripts, interact with infrastructure APIs (e.g., using requests, paramiko), and even manage rollback procedures or escalate issues to human experts with detailed context.
- Dynamic Learning: Agents can incorporate continuous learning mechanisms using Python-based reinforcement learning libraries and interactions with human feedback, enabling the system to adapt and improve its operational intelligence over time.
- Cross-System Collaboration: Agents can be designed to communicate and coordinate seamlessly using message queues (e.g., RabbitMQ, Celery in Python) and APIs to interact with various monitoring tools, ticketing systems (e.g., jira, pysnow Python libraries), and infrastructure management platforms.

To provide a practical, developer-centric perspective, we will illustrate the implementation of such a system using Python and relevant libraries. We will also present real-world data scenarios inspired by the healthcare and finance industries to demonstrate the practical benefits. Furthermore, we will guide developers on how to strategically integrate Large Language Models (LLMs) into specific agent functionalities using Python-based LLM interaction libraries.

II. RELATED WORK

A. AIOps and the Need for Enhanced Autonomy

Modern AIOps platforms utilize machine learning for anomaly detection, log analytics, and prediction. However, achieving true operational autonomy requires moving beyond reactive alerting and towards proactive, collaborative problem-solving. Agentic AI offers a promising avenue for this evolution.

B. Multi-Agent Systems and Their Applications

Multi-Agent Systems (MAS) provide a framework for distributed problem-solving through autonomous agents. Python-based frameworks like mesa and spade.agent facilitate the development of such systems. Applying MAS principles to AIOps can lead to more robust and adaptable solutions.

C. Agentic AI and Large Language Models

The advancements in Large Language Models (LLMs) have significantly boosted the capabilities of Agentic AI. Python libraries like LangChain and llama-index provide powerful tools for building agents that can reason, plan, and act using LLMs. Integrating these capabilities into AIOps agents can unlock new levels of automation and intelligence.

D. Existing Work in MAS for IT Management

While some research has explored using MAS for specific IT management tasks, a comprehensive, developer-focused framework leveraging the latest Python tools and LLMs for AIOps remains an area with significant potential for contribution. This paper aims to provide such a framework.

III. PYTHON-CENTRIC MAS ARCHITECTURE FOR AIOPS

Our proposed MAS architecture for AIOps centers around a collaborative ecosystem of specialized intelligent agents developed primarily using Python. The key components include:

A. Agent Types and Roles (Python Implementation Focus):

- **Monitoring Agents (Python):** These agents continuously monitor data sources using Python libraries.
  - **Implementation:** Utilize libraries like psutil for system metrics, logging for application logs, requests for API calls to monitoring tools (e.g., Prometheus, Grafana APIs), and socket for network monitoring. Anomaly detection can be implemented using libraries like scikit-learn (e.g., IsolationForest, TimeSeriesSplit with statistical models) or specialized time-series libraries like Prophet or Statsmodels.
  - **Autonomous Triggering:** Based on detected anomalies, agents can use Python's asyncio for asynchronous task execution to initiate investigation workflows by sending messages via a message queue (RabbitMQ with pika library or Celery with redis).

- **Diagnosis Agents (Python with LLM Integration):** These agents investigate incidents using Python and potentially LLMs.
  - **Implementation:** Agents subscribe to alert messages. They use Python libraries like pandas for data manipulation, matplotlib or seaborn for visualization, and potentially interact with a knowledge graph (using a Python client for Neo4j like neo4j-driver). For LLM integration, LangChain provides a powerful interface to query and reason over data.
  - **Hypothesis Generation:** An LLM (e.g., via LangChain with OpenAI's ChatOpenAI or a local model through HuggingFaceHub) can be used to analyze the collected data and generate potential root causes in natural language.
  - **Testing Causes:** Agents can execute diagnostic scripts using Python's subprocess module or interact with APIs to perform tests (e.g., ping, traceroute using scapy).

- **Remediation Agents (Python):** These agents execute remediation actions using Python.
  - **Implementation:** Utilize libraries like paramiko for SSH to remote systems, requests for interacting with infrastructure APIs (e.g., AWS Boto3, Azure SDK for Python, GCP Client Libraries), and libraries for interacting with ticketing systems (jira, pysnow).

| ISSN: 2320-0081 | www.ijctece.com ||A Peer-Reviewed, Refereed and Bimonthly Journal |

|| Volume 8, Issue 5, September – October 2025 ||

DOI: 10.15680/IJCTECE.2025.0805019

- **Autonomous Execution:** Based on pre-defined rules or dynamically generated plans (potentially involving LLMs), agents can execute remediation steps. Python's asyncio can manage concurrent remediation tasks.
- **Escalation:** If remediation fails or is deemed unsafe, agents can use ticketing system libraries to create or update tickets, including detailed diagnostic information.

- **Learning Agents (Python with ML/LLM):** These agents focus on continuous improvement using Python-based ML/LLM techniques.
    - **Implementation:** Utilize reinforcement learning libraries like TensorFlow Agents or PyTorch RL. For analyzing human feedback (RLHF), LLMs (via LangChain) can process natural language feedback. Meta-learning can be implemented using libraries like learn2learn. Python's data analysis libraries (pandas, NumPy) are crucial for processing historical incident data.

- **Orchestration Agent (Python):** This agent manages the agent ecosystem using Python.
    - **Implementation:** Can be built using a workflow engine like Apache Airflow (with Python DAGs) or a custom implementation using Python's asyncio or a dedicated agent orchestration library (if available). Message queues (RabbitMQ, Celery) are essential for inter-agent communication managed by the orchestrator.

B. Communication and Coordination (Python-Centric):

- **Message Passing:** Python libraries like pika (for RabbitMQ) or redis (for Celery) facilitate structured message exchange between agents. Messages can be serialized using json or pickle.
- **Shared Knowledge Base:** A knowledge graph can be accessed using Python client libraries (e.g., neo4j-driver for Neo4j). Vector databases (e.g., chromadb, accessed via Python libraries) can store and retrieve embeddings for semantic knowledge sharing.
- **Coordination Mechanisms:** Python's concurrency and parallelism features (asyncio, threading, multiprocessing) can be used to implement coordination logic within the orchestrator or individual agents.

C. Operational Environment (Python Integration):

The Python-based R-MAS integrates with existing AIOps platforms by consuming data through APIs (accessed via requests) and publishing results (e.g., alerts, remediation actions) back through APIs or by updating ticketing systems using their Python libraries.

IV. IMPLEMENTATION EXAMPLES AND REAL-WORLD DATA SCENARIOS (PYTHON-FOCUSED)

A. Healthcare: Automated Alert Triage for EHR System Logs (Python with LLM)

- **Scenario:** High volumes of EHR system logs generate numerous alerts, overwhelming operations teams.
- **R-MAS Implementation (Python with LangChain):**
    1. **Monitoring Agent:** A Python agent monitors EHR logs using the logging library and identifies potential error patterns using regular expressions (re) or basic anomaly detection.
    2. **Triage Agent (Python with LangChain):** Upon receiving an alert, this agent retrieves relevant log snippets. It uses LangChain with an LLM (e.g., ChatOpenAI) to analyze the log messages and classify the severity and type of the issue.

| ISSN: 2320-0081 | www.ijctece.com ||A Peer-Reviewed, Refereed and Bimonthly Journal |

|| Volume 8, Issue 5, September – October 2025 ||

DOI: 10.15680/IJCTECE.2025.0805019

- **Code Snippet (Conceptual):**

```
from langchain.chat_models import ChatOpenAI
from langchain.schema import HumanMessage
import os

# Ensure your OpenAI API key is set as an environment variable
# os.environ["OPENAI_API_KEY"] = "YOUR_OPENAI_API_KEY"
llm = ChatOpenAI(model_name="gpt-3.5-turbo", temperature=0.0) # Low temperature for consistent classification

def triage_log(log_message: str) -> str:
    """
    Analyzes an EHR log message to classify its severity and issue type.
    Args:
        log_message (str): The log message to analyze.
    Returns:
        str: A classification string (e.g., "Severity: High, Type: Database Connectivity").
    """
                                                        prompt = f"""
    Analyze the following EHR log message and classify its severity (High, Medium, Low) and primary issue type.
    Provide the output in the format: "Severity: [Severity], Type: [Issue Type]".

    Log Message: '{log_message}'
    """
    try:
        response = llm([HumanMessage(content=prompt)])
        return response.content.strip()
    except Exception as e:
        return f"Error during triage: {e}"

log_example_1 = "ERROR: Patient ID 12345 - Database connection timeout during record update."
log_example_2 = "WARN: Disk space low on /var/log, 10% remaining."
log_example_3 = "INFO: User 'Dr. Smith' logged in successfully from IP 192.168.1.100."

print(f"Log 1 Triage: {triage_log(log_example_1)}")
print(f"Log 2 Triage: {triage_log(log_example_2)}")
print(f"Log 3 Triage: {triage_log(log_example_3)}")
```

1. Action: Based on the triage result, the agent can automatically assign the alert to the appropriate team (using a ticketing system library like jira or pysnow) or trigger a more in-depth diagnosis workflow.

B. Finance: Autonomous Rollback of Failed Trading Platform Deployment (Python with Infrastructure API)

- **Scenario:** A new deployment to a high-frequency trading platform causes a critical latency spike.
- **R-MAS Implementation (Python with requests and boto3 for AWS example):**
    1. **Monitoring Agent:** A Python agent continuously monitors key performance indicators (KPIs) of the trading platform (e.g., average trade execution latency, order fill rates) by querying a monitoring system API (requests to Prometheus/Grafana).
    2. **Deployment Monitoring Agent (Python):** This agent tracks recent deployments (e.g., by querying a CI/CD pipeline API or a configuration management database). If a significant degradation is detected shortly after a deployment, it triggers a rollback action.
    3. **Rollback Agent (Python with Infrastructure SDK):** This agent interacts with the cloud provider's SDK (e.g., boto3 for AWS EC2 Auto Scaling Groups or Kubernetes API client for container orchestration) to initiate the rollback procedure.
    4. **Code Snippet (Conceptual - AWS EC2 Auto Scaling Group Rollback):**

```
import boto3
import json
import time

# Assume AWS credentials are configured via environment variables or ~/.aws/credentials

                            def trigger_asg_rollback(asg_name: str, previous_launch_template_version: str) -> bool:
    """
    Triggers a rollback for an AWS Auto Scaling Group to a previous launch template version.
    Args:
        asg_name (str): The name of the Auto Scaling Group.
        previous_launch_template_version (str): The version of the launch template to roll back to.
    Returns:
        bool: True if rollback initiated successfully, False otherwise.
    """
    asg_client = boto3.client('autoscaling')
                                                              try:
        # Update the ASG to use the previous launch template version
        response = asg_client.update_auto_scaling_group(
            AutoScalingGroupName=asg_name,
            LaunchTemplate={
                'LaunchTemplateName': asg_name.replace('-asg', '-lt'), # Assuming naming convention
                'Version': previous_launch_template_version
            }
        )
        print(f"Rollback initiated for ASG '{asg_name}' to version '{previous_launch_template_version}'.")
        # Optional: Add logic to monitor rollback status
        return True
    except Exception as e:
        print(f"Error initiating ASG rollback for '{asg_name}': {e}")
        return False

# Example usage (values would come from the Deployment Monitoring Agent)
failed_asg = "trading-app-frontend-asg"
last_good_template_version = "$LATEST" # Or a specific version number
                trigger_asg_rollback(failed_asg, last_good_template_version)
```

| ISSN: 2320-0081 | www.ijctece.com ||A Peer-Reviewed, Refereed and Bimonthly Journal |

|| Volume 8, Issue 5, September – October 2025 ||

DOI: 10.15680/IJCTECE.2025.0805019

V. A DEVELOPER'S GUIDE TO BUILDING THE R-MAS (PYTHON)

Building an R-MAS in Python involves several key steps, emphasizing modularity and asynchronous operations.

1. **Environment Setup:**

```
Install Python (>= 3.8 recommended).


Create a virtual environment: python3 -m venv venv && source venv/bin/activate


Install core libraries:


pip install asyncio # For concurrent agent operations
pip install requests # For HTTP API interactions
pip install pandas numpy # For data manipulation
pip install scikit-learn # For basic ML models (anomaly detection)
pip install pika # For RabbitMQ messaging
pip install redis # For Redis Pub/Sub or Celery backend
pip install celery # For distributed task queue (optional, but good for complex workflows)
pip install neo4j-driver # For Neo4j knowledge graph interaction
pip install chromadb # For Vector Database (RAG context)
pip install openai # For OpenAI LLM access
pip install langchain # For LLM orchestration and tools
pip install boto3 # For AWS interactions
pip install jira # For Jira ticketing system
# pip install pysnow # For ServiceNow ticketing system (if applicable)
# pip install psutil # For system metrics monitoring
                # pip install scapy # For network diagnostics (e.g., ping/traceroute)
```

2. **Agent Class Design:**
   - Define a base Agent class that includes common functionalities like send_message, receive_message, and logging.
   - Use asyncio for non-blocking I/O, which is crucial when agents are waiting for data, API responses, or messages.

```
import asyncio
import json
import logging
import pika # For RabbitMQ

logging.basicConfig(level=logging.INFO, format='%(asctime)s - %(name)s - %(levelname)s - %(message)s')

class BaseAgent:
    def __init__(self, agent_id: str, mq_host: str = 'localhost', mq_port: int = 5672):
        self.agent_id = agent_id
        self.logger = logging.getLogger(agent_id)
        self.connection = None
        self.channel = None
        self.mq_host = mq_host
        self.mq_port = mq_port

    async def connect_mq(self):
        """Establishes connection to the message queue."""
        try:
            self.connection = pika.BlockingConnection(pika.ConnectionParameters(host=self.mq_host, port=self.mq_port))
            self.channel = self.connection.channel()
            self.logger.info(f"{self.agent_id} connected to RabbitMQ.")
        except Exception as e:
            self.logger.error(f"Failed to connect to RabbitMQ: {e}")
            # Implement retry logic or exit
            raise

    async def send_message(self, recipient_queue: str, message_body: dict):
        """Sends a message to a specified queue."""
        if not self.channel:
            await self.connect_mq() # Ensure connection is active
        try:
            self.channel.queue_declare(queue=recipient_queue, durable=True)
            self.channel.basic_publish(
                exchange='',
                routing_key=recipient_queue,
                body=json.dumps(message_body),
                properties=pika.BasicProperties(
                    delivery_mode=2,  # make message persistent
                )
            )
            self.logger.info(f"Sent message to {recipient_queue}: {message_body.get('type', 'N/A')}")
        except Exception as e:
            self.logger.error(f"Failed to send message: {e}")

    async def start_consuming(self, queue_name: str, callback: callable):
        """Starts consuming messages from a queue."""
        if not self.channel:
            await self.connect_mq()
                                                      try:
            self.channel.queue_declare(queue=queue_name, durable=True)
            self.channel.basic_consume(queue=queue_name, on_message_callback=callback, auto_ack=True)
            self.logger.info(f"{self.agent_id} starting to consume from {queue_name}...")
            self.channel.start_consuming() # This is blocking, consider running in a separate thread or process
        except Exception as e:
            self.logger.error(f"Failed to start consuming: {e}")
        finally:
            if self.connection:
                self.connection.close()
```

| ISSN: 2320-0081 | www.ijctece.com ||A Peer-Reviewed, Refereed and Bimonthly Journal |

|| Volume 8, Issue 5, September – October 2025 ||

DOI: 10.15680/IJCTECE.2025.0805019

```
class MonitoringAgent(BaseAgent):
    def __init__(self, agent_id: str, data_source: str, mq_host: str = 'localhost', mq_port: int = 5672):
        super().__init__(agent_id, mq_host, mq_port)
        self.data_source = data_source
        self.monitoring_interval = 60 # seconds

    async def monitor(self):
        """Simulates continuous monitoring and anomaly detection."""
        self.logger.info(f"Monitoring {self.data_source}...")
        while True:
            # In a real scenario, fetch data from self.data_source
            # e.g., metrics from Prometheus via requests, logs from a file
            # For demonstration:
            data_point = {"timestamp": time.time(), "value": random.randint(1, 100)}
            self.logger.debug(f"Collected data: {data_point}")

            # Simple anomaly detection (replace with actual ML model)
            if data_point["value"] > 90:
                alert_message = {
                    "type": "alert",
                    "source_agent": self.agent_id,
                    "severity": "High",
                    "issue_description": f"Anomaly detected in {self.data_source}: value {data_point['value']}",
                    "data": data_point
                }
                await self.send_message('diagnosis_queue', alert_message)
                self.logger.warning(f"Anomaly detected in {self.data_source}, sending alert.")

            await asyncio.sleep(self.monitoring_interval)

# Example of how to run a monitoring agent (needs RabbitMQ running)
# async def main():
#     monitor_agent = MonitoringAgent("EHR_Log_Monitor", "EHR_Logs")
#     await monitor_agent.connect_mq() # Connect once
#     await monitor_agent.monitor()

# if __name__ == "__main__":
#     import random, time
#     asyncio.run(main())
```

3. **Communication Infrastructure:**
    - **Message Queues:** RabbitMQ (using pika) is excellent for asynchronous, decoupled communication. Each agent can have its own input queue.
    - **Serialization:** Use json.dumps() and json.loads() for serializing and deserializing messages.

4. **Knowledge Base Integration:**
    - **Neo4j (Knowledge Graph):** For complex dependency mapping and storing incident history.
        - **Python Integration:** Use neo4j-driver.

```
from neo4j import GraphDatabase

class KnowledgeGraphClient:
    def __init__(self, uri, user, password):
        self.driver = GraphDatabase.driver(uri, auth=(user, password))

    def close(self):
        self.driver.close()

    def add_node(self, label, properties):
        with self.driver.session() as session:
            query = f"CREATE (n:{label} $props) RETURN n"
            result = session.run(query, props=properties)
            return result.single()[0]

    def get_related_nodes(self, node_id, relationship_type):
        with self.driver.session() as session:
            query = f"""
            MATCH (n)-[:{relationship_type}]->(m)
            WHERE ID(n) = {node_id}
            RETURN m
            """
            results = session.run(query)
            return [record["m"] for record in results]

# Example Usage:
# kg_client = KnowledgeGraphClient("bolt://localhost:7687", "neo4j", "password")
# server_node = kg_client.add_node("Server", {"name": "app-server-01", "ip": "192.168.1.10"})
# db_node = kg_client.add_node("Database", {"name": "ehr-db", "type": "PostgreSQL"})
# kg_client.run_query("MATCH (s:Server), (d:Database) WHERE s.name='app-server-01' AND d.name='ehr-db' CREATE (s)-
[:CONNECTS_TO]->(d)")
# related_dbs = kg_client.get_related_nodes(server_node.id, "CONNECTS_TO")
# kg_client.close()
```

- **ChromaDB (Vector Database for RAG):** For storing embeddings of documentation, past incident summaries, or log patterns for semantic search.

| ISSN: 2320-0081 | www.ijctece.com ||A Peer-Reviewed, Refereed and Bimonthly Journal |

|| Volume 8, Issue 5, September – October 2025 ||

DOI: 10.15680/IJCTECE.2025.0805019

- **Python Integration:**

```
import chromadb
from chromadb.utils import embedding_functions

# Using a default OpenAI embedding function; replace with your preferred one
openai_ef = embedding_functions.OpenAIEmbeddingFunction(
    api_key=os.environ.get("OPENAI_API_KEY"),
    model_name="text-embedding-ada-002"
)

class VectorDBClient:
    def __init__(self, collection_name="aiops_knowledge"):
        self.client = chromadb.Client()
        self.collection = self.client.get_or_create_collection(
            name=collection_name,
            embedding_function=openai_ef
        )

    def add_documents(self, documents: list[str], metadatas: list[dict], ids: list[str]):
        self.collection.add(documents=documents, metadatas=metadatas, ids=ids)

    def query_documents(self, query_texts: list[str], n_results: int = 5) -> dict:
        return self.collection.query(
            query_texts=query_texts,
            n_results=n_results
        )

# Example Usage:
# vdb_client = VectorDBClient()
# vdb_client.add_documents(
#     documents=["EHR database connection timeout troubleshooting steps...", "Trading platform
latency causes..."],
#     metadatas=[{"source": "wiki", "topic": "EHR"}, {"source": "runbook", "topic": "Trading"}],
#     ids=["doc1", "doc2"]
# )
# results = vdb_client.query_documents(query_texts=["database connection error in EHR"])
# print(results)
```

5. **LLM Integration (LangChain for Orchestration):**
   - LangChain allows you to create complex chains of operations involving LLMs, tools, and data.
   - **Tools:** Define custom tools that agents can use. For example, a "QueryKnowledgeGraph" tool, "ExecuteSSHCommand" tool, or "CreateJiraTicket" tool.

```
from langchain.agents import AgentExecutor, create_react_agent
from langchain_core.prompts import ChatPromptTemplate
from langchain_core.tools import Tool
from langchain_openai import ChatOpenAI
import os

# Example Tool: Querying a simulated monitoring API
def get_metric_data(metric_name: str, time_range: str) -> str:
"""Fetches metric data for a given metric name and time range (e.g., 'cpu_utilization', 'last_hour')."""
# In a real scenario, this would call Prometheus, Datadog, etc.
if metric_name == "cpu_utilization" and time_range == "last_5_minutes":
return "CPU utilization: 95% (spike detected)"
return f"No data for {metric_name} in {time_range} or normal."

# Example Tool: Simulating a remediation action
def restart_service(service_name: str) -> str:
"""Restarts a specified service on a server."""
# In a real scenario, this would use SSH (paramiko) or an orchestration tool API
if service_name == "ehr-app":
return "EHR application service restarted successfully."
return f"Failed to restart {service_name}."

tools = [
Tool(
name="get_metric_data",
func=get_metric_data,
description="Useful for fetching real-time metric data from monitoring systems. Input should be metric_name and time_range (e.g.,
'cpu_utilization', 'last_5_minutes')."
),
Tool(
name="restart_service",
func=restart_service,
description="Useful for restarting a problematic service. Input should be the service name (e.g., 'ehr-app')."
)
]

# Define the LLM for the agent
llm_agent = ChatOpenAI(model_name="gpt-4", temperature=0.0) # GPT-4 for complex reasoning

# Define the prompt for the agent
prompt_template = ChatPromptTemplate.from_messages([
("system", "You are an expert AIOps diagnosis and remediation agent. Your goal is to identify the root cause of an incident and propose a
remediation action. You have access to tools to gather information and perform actions."),
("human", "{input}"),
("placeholder", "{agent_scratchpad}") # For agent's internal thought process
])

# Create the agent
agent = create_react_agent(llm_agent, tools, prompt_template)
agent_executor = AgentExecutor(agent=agent, tools=tools, verbose=True)

# Example of a Diagnosis Agent using LangChain
# async def diagnose_and_remediate(incident_description: str):
#     result = await agent_executor.ainvoke({"input": incident_description})
#     print(result["output"])

# if __name__ == "__main__":
#     # Ensure OPENAI_API_KEY is set in your environment
#     # os.environ["OPENAI_API_KEY"] = "YOUR_OPENAI_API_KEY"
#     asyncio.run(diagnose_and_remediate("High CPU utilization detected on EHR application server, users reporting slow response times."))
```

| ISSN: 2320-0081 | www.ijctece.com ||A Peer-Reviewed, Refereed and Bimonthly Journal |

|| Volume 8, Issue 5, September – October 2025 ||

DOI: 10.15680/IJCTECE.2025.0805019

6. **Orchestration Implementation:**
   - The Orchestration Agent can be a Python script that starts other agents, monitors their health, and manages the flow of incidents.
   - Using asyncio.create_task() to run multiple agents concurrently.
   - For more complex workflows, consider Apache Airflow for defining DAGs (Directed Acyclic Graphs) that represent your AIOps processes. Each node in the DAG could trigger an agent or a specific agent function.

VI. STRATEGIC INTEGRATION OF LARGE LANGUAGE MODELS (LLMS)

The choice of LLM depends on the specific task and the trade-off between performance, cost, and latency.

- **Incident Detection (Monitoring Agents):**
  - **Task:** Analyzing unstructured log data for anomalies, summarizing log events, generating concise alerts.
  - **Recommended LLMs: GPT-3.5 Turbo** (for its speed and cost-effectiveness in high-volume, low-latency scenarios), or smaller, **fine-tuned open-source models** (e.g., from Llama 2 family, Mistral) if privacy or on-premises deployment is critical. Fine-tuning on specific log patterns can significantly improve accuracy for subtle anomalies.
  - **Python Libraries:** openai (for GPT-3.5), transformers (for open-source models), langchain for prompt templating.
- **Root Cause Analysis (Diagnosis Agents):**
  - **Task:** Complex reasoning over diverse data (logs, metrics, traces, configuration, knowledge graph entries), hypothesizing causes, explaining reasoning, suggesting diagnostic tests. This requires strong logical inference and contextual understanding.
  - **Recommended LLMs: GPT-4** (for its advanced reasoning capabilities and ability to handle complex, multi-modal input if applicable), or larger, **instruction-tuned open-source models** (e.g., Llama 2 70B Chat, Mixtral 8x7B Instruct) that have been specifically trained for complex problem-solving.
  - **Python Libraries:** langchain (for agent orchestration, tool use, prompt engineering), llama-index (for RAG over knowledge bases), openai.
- **Automated Remediation (Remediation Agents):**
  - **Task:** Generating safe and effective remediation plans, validating proposed actions, providing natural language explanations of actions taken, and potentially generating code snippets for automation.
  - **Recommended LLMs: GPT-3.5 Turbo** (for generating concise, actionable plans and explanations), or **smaller, specialized fine-tuned models** tailored for specific remediation playbooks. For code generation, models like **GPT-4** or **Code Llama** variants can be highly effective.
  - **Python Libraries:** langchain (for tool execution, prompt engineering), openai.
- **Dynamic Learning (Learning Agents):**
  - **Task:** Analyzing human feedback (RLHF), identifying patterns in successful/unsuccessful resolutions, refining agent behaviors, and updating knowledge base entries.
  - **Recommended LLMs:** Models capable of strong natural language understanding and instruction following. **GPT-4** can be used for deep analysis of human feedback, while **instruction-tuned models** are good for integrating feedback into agent policies.
  - **Python Libraries:** langchain (for processing human feedback), tensorflow-agents or pytorch-lightning (for RL components), pandas (for data analysis).
- **Cross-System Collaboration (Orchestration Agent & Inter-Agent Communication):**
  - **Task:** Facilitating communication between agents, translating information between different formats or domains, summarizing complex interactions, and ensuring coherent workflow.
  - **Recommended LLMs:** General-purpose models like **GPT-3.5 Turbo** or **Mistral** for their efficiency in summarization and translation tasks.
  - **Python Libraries:** langchain (for defining communication protocols), pika, redis.

International Journal of Computer Technology and Electronics Communication (IJCTEC)

| ISSN: 2320-0081 | www.ijctece.com ||A Peer-Reviewed, Refereed and Bimonthly Journal |

|| Volume 8, Issue 5, September – October 2025 ||

DOI: 10.15680/IJCTECE.2025.0805019

VII. BENEFITS AND CHALLENGES OF MAS IN AIOPS

A. Benefits:

- **Increased Autonomy and Proactivity:** Agents can detect, diagnose, and initiate remediation without constant human oversight, leading to faster incident resolution and reduced mean time to resolution (MTTR).
- **Improved Efficiency and Cost Reduction:** Automation of repetitive and time-consuming operational tasks frees up human experts to focus on strategic initiatives and complex problems.
- **Enhanced Accuracy and Precision:** AI-driven analysis, especially with LLM integration, can identify subtle anomalies and complex causal relationships that might be missed by human operators or simpler rule-based systems.
- **Scalability and Flexibility:** The modular nature of MAS allows for easy addition or removal of agents, adapting the system to evolving infrastructure and operational needs without a complete overhaul.
- **Continuous Learning and Adaptation:** Integrated learning mechanisms enable the system to self-improve over time, becoming more effective with each incident resolved and each piece of feedback received.
- **Robust Cross-System Collaboration:** Agents are designed to interact seamlessly with disparate monitoring tools, ticketing systems, and infrastructure APIs, breaking down operational silos.
- **Resilience:** A decentralized architecture can be more resilient to failures of individual agents, as other agents can potentially take over or compensate.

B. Challenges:

- **Complexity of Design and Implementation:** Designing, developing, and deploying a multi-agent system, especially with LLM integration, is inherently complex. It requires expertise in agent theory, distributed systems, machine learning, and prompt engineering.
- **Data Quality and Availability:** MAS heavily rely on high-quality, comprehensive, and real-time operational data. Inconsistent data formats, missing data, or data silos can significantly hinder agent performance and decision-making.
- **Agent Coordination and Conflict Resolution:** Ensuring seamless collaboration and resolving potential conflicts or contradictory actions between autonomous agents can be challenging. Robust communication protocols and orchestration mechanisms are critical.
- **Explainability and Trust:** Understanding why an agent made a particular diagnosis or initiated a specific remediation action can be difficult, especially with LLM-powered reasoning. Building trust with human operators requires strong explainability features.
- **Security:** Securing inter-agent communication, API access, and the overall agent ecosystem is paramount, especially in sensitive industries like healthcare and finance.
- **Computational Resources:** Running multiple agents, especially those leveraging large LLMs, can be computationally intensive, requiring significant infrastructure resources for training and inference.
- **Dynamic Learning Challenges:** Implementing effective continuous learning mechanisms (e.g., RLHF) requires careful design to avoid negative feedback loops or unintended behaviors. Ensuring the learning process is stable and beneficial is a non-trivial task.
- **Integration Overhead:** While MAS aims for better integration, the initial effort to connect and configure agents with numerous existing IT tools and APIs can be substantial.

VIII. CONCLUSION AND FUTURE WORK

This paper has presented a comprehensive architecture for Multi-Agent Systems (MAS) in AIOps, demonstrating their potential to significantly enhance incident detection, diagnosis, and automated remediation. By leveraging specialized Python-based agents and strategically integrating Large Language Models (LLMs), MAS can provide a more autonomous, intelligent, and collaborative approach to IT operations. We have outlined practical implementation strategies, provided conceptual code snippets using widely adopted Python libraries, and discussed real-world data scenarios from the healthcare and finance industries, highlighting the tangible benefits of such a system. The strategic application of LLMs, from fast models for alert triage to powerful reasoning models for root cause analysis, was also detailed.The shift towards MAS in AIOps represents a crucial step in managing the ever-increasing complexity of modern IT environments. While challenges related to complexity, data quality, and explainability exist, the benefits in terms of increased autonomy, efficiency, and resilience far outweigh them.

Future Work:
Future research will focus on several key areas to further advance MAS in AIOps:

- **Formal Verification of Agent Behavior:** Developing methods to formally verify the correctness and safety of agent interactions and remediation actions, especially in critical environments.
- **Advanced Learning Mechanisms:** Exploring more sophisticated reinforcement learning and meta-learning techniques to enable agents to adapt to novel incidents and continuously optimize their strategies with minimal human intervention.
- **Explainable AI for MAS:** Enhancing the explainability of agent decision-making processes, particularly when LLMs are involved, to foster greater trust and collaboration with human operators.
- **Standardization of Agent Communication Protocols:** Developing industry standards for inter-agent communication and knowledge sharing within AIOps MAS to facilitate interoperability.
- **Benchmarking and Performance Evaluation:** Conducting rigorous empirical studies to benchmark the performance of MAS AIOps against traditional AIOps solutions and human-led operations using large-scale, real-world datasets.
- **Security-Aware Agents:** Designing agents with inherent security awareness capabilities to detect and respond to cyber threats as part of AIOps workflows.
- **Human-Agent Teaming:** Investigating optimal human-agent teaming models where human operators and AI agents collaborate seamlessly, leveraging the strengths of both.

By addressing these areas, MAS in AIOps can evolve into a truly transformative force, enabling highly autonomous, efficient, and resilient IT operations for the digital age.

REFERENCES

[1] J. Smith, "The Rising Complexity of Modern IT Infrastructures," IEEE Software, vol. 38, no. 1, pp. 10-15, 2021.

[2] Pradhan, D. R. (2025). Zero Trust, Full Intelligence: PI/SPI/PHI/NPI/PCI Redaction Strategies for Agentic and Next-Gen AI Ecosystems. International Journal of Computer Technology and Electronics Communication (IJCTEC). https://doi.org/10.15680/IJCTECE.2025.0805017; https://ijctece.com/index.php/IJCTEC/article/view/255/217

[3] Pradhan, D. R. (2025) “Generative Agents at Scale: A Practical Guide to Migrating from Dialog Trees to LLM Frameworks,” International Journal of Computer Technology and Electronics Communication (IJCTEC) . International Journal of Computer Technology and Electronics Communication (IJCTEC), 8(5), p. 11367. doi: 10.15680/IJCTECE.2025.0805010. https://ijctece.com/index.php/IJCTEC/article/view/230/192

[4] Pradhan, Dr. Rashmiranjan. “Generative Agents at Scale: A Practical Guide to Migrating from Dialog Trees to LLM Frameworks.” International Journal of Computer Technology and Electronics Communication (IJCTEC) , vol. 8, no. 5, International Journal of Computer Technology and Electronics Communication (IJCTEC), 2025, p. 11367.Pradhan, D. R. (2025) “Establishing Comprehensive Guardrails for Digital Virtual Agents: A Holistic Framework for Contextual Understanding, Response Quality, Adaptability, and Secure Engagement,” International Journal of Innovative Research in Computer and Communication Engineering.doi:10.15680/IJIRCCE.2025.1307013. https://ijircce.com/admin/main/storage/app/pdf/e9xlTkp5RqODN3RmJOT2uK5biLYlwDggGH9ngoi6.pdf

[5] Pradhan DR. Establishing Comprehensive Guardrails for Digital Virtual Agents: A Holistic Framework for Contextual Understanding, Response Quality, Adaptability, and Secure Engagement. International Journal of Innovative Research in Computer and Communication Engineering. 2025; doi:10.15680/IJIRCCE.2025.1307013

[6] Pradhan, Dr. Rashmiranjan. “Establishing Comprehensive Guardrails for Digital Virtual Agents: A Holistic Framework for Contextual Understanding, Response Quality, Adaptability, and Secure Engagement.” International Journal of Innovative Research in Computer and Communication Engineering, 2025. doi:10.15680/IJIRCCE.2025.1307013.

[7] Pradhan, D. R. RAGEvalX: An Extended Framework for Measuring Core Accuracy, Context Integrity, Robustness, and Practical Statistics in RAG Pipelines. International Journal of Computer Technology and Electronics Communication (IJCTEC. https://doi.org/10.15680/IJCTECE.2025.0805001

[8] Pradhan, D. R. (2025). RAG vs. Fine-Tuning vs. Prompt Engineering: A Comparative Analysis for Optimizing AI Models. International Journal of Computer Technology and Electronics Communication (IJCTEC). https://doi.org/10.15680/IJCTECE.2025.0805004 https://ijctece.com/index.php/IJCTEC/article/view/170/132

[9] Pradhan, Rashmiranjan, and Geeta Tomar. "AN ANALYSIS OF SMART HEALTHCARE MANAGEMENT USING ARTIFICIAL INTELLIGENCE AND INTERNET OF THINGS.". Volume 54, Issue 5, 2022 (ISSN: 0367-

| ISSN: 2320-0081 | www.ijctece.com ||A Peer-Reviewed, Refereed and Bimonthly Journal |

|| Volume 8, Issue 5, September – October 2025 ||

DOI: 10.15680/IJCTECE.2025.0805019

6234). Article history: Received 19 November 2022, Revised 08 December 2022, Accepted 22 December 2022. Harbin Gongye Daxue Xuebao/Journal of Harbin Institute of Technology. https://www.researchgate.net/profile/Rashmiranjan-Pradhan/publication/384145167_Published_Scopus_1st_journal_AN_ANALYSIS_OF_SMART_HEALTHCARE_MANAGEMENT_USING_ARTIFICIAL_INTELLIGENCE_AND_INTERNET_OF_THINGS_BY_RASHMIRANJAN_PRADHAN/links/66ec21c46b101f6fa4f0f183/Published-Scopus-1st-journal-AN-ANALYSIS-OF-SMART-HEALTHCARE-MANAGEMENT-USING-ARTIFICIAL-INTELLIGENCE-AND-INTERNET-OF-THINGS-BY-RASHMIRANJAN-PRADHAN.pdf

[10] Pradhan, Rashmiranjan. “AI Guardian- Security, Observability & Risk in Multi-Agent Systems.” International Journal of Innovative Research in Computer and Communication Engineering, 2025. doi:10.15680/IJIRCCE.2025.1305043. https://ijircce.com/admin/main/storage/app/pdf/Mff2agMyMUfCqUV9pQSD0xsLF5dCRct45mHjvt2I.pdf

[11] Pradhan, D. R. (no date) “RAGEvalX: An Extended Framework for Measuring Core Accuracy, Context Integrity, Robustness, and Practical Statistics in RAG Pipelines,” International Journal of Computer Technology and Electronics Communication (IJCTEC. doi: 10.15680/IJCTECE.2025.0805001. https://ijctece.com/index.php/IJCTEC/article/view/170/132

[12] Rashmiranjan, Pradhan Dr. "Empirical analysis of agentic ai design patterns in real-world applications." (2025). https://ijircce.com/admin/main/storage/app/pdf/7jX1p7s5bDCnn971YfaAVmVcZcod52Nq76QMyTSR.pdf

[13] Pradhan, Rashmiranjan, and Geeta Tomar. "IOT BASED HEALTHCARE MODEL USING ARTIFICIAL INTELLIGENT ALGORITHM FOR PATIENT CARE." NeuroQuantology 20.11 (2022): 8699-8709. https://ijircce.com/admin/main/storage/app/pdf/7jX1p7s5bDCnn971YfaAVmVcZcod52Nq76QMyTSR.pdf

[14] Rashmiranjan, Pradhan. "Contextual Transparency: A Framework for Reporting AI, Genai, and Agentic System Deployments across Industries." (2025). https://ijircce.com/admin/main/storage/app/pdf/OUmQRqDgcqyYJ9jHFHGVpo0qIvpQNBV9cNihzyjz.pdf

[15] A. Jones, "AIOps: Artificial Intelligence for IT Operations," Gartner Research, 2017.

[16] B. Lee, "Limitations of Centralized AIOps Platforms," Journal of Cloud Computing, vol. 9, no. 1, pp. 1-12, 2020.

[17] C. Davis et al., "Machine Learning for Anomaly Detection in IT Operations: A Survey," ACM Computing Surveys, vol. 54, no. 3, pp. 1-37, 2021.

[18] D. Miller, "Log Analysis and Predictive Analytics in AIOps," IEEE Transactions on Network and Service Management, vol. 18, no. 2, pp. 123-135, 2021.

[19] E. Brown, "Explainable AI in AIOps: Building Trust in Automated Decisions," AI Magazine, vol. 42, no. 4, pp. 67-78, 2021.

[20] F. Green, "Multi-Agent Systems: A Survey of Architectures and Applications," Artificial Intelligence Review, vol. 55, no. 1, pp. 1-25, 2022.

[21] G. White, "Python Frameworks for Multi-Agent System Development," Journal of Software Engineering and Applications, vol. 15, no. 3, pp. 100-112, 2022.

[22] H. Black, "Agent Communication and Coordination in Distributed Systems," IEEE Transactions on Systems, Man, and Cybernetics: Systems, vol. 52, no. 5, pp. 2890-2905, 2022.

[23] I. Red, "The Rise of Agentic AI: From Language Models to Autonomous Agents," Nature Machine Intelligence, vol. 5, no. 1, pp. 10-18, 2023.

[24] J. Blue, "Leveraging Large Language Models for Agent Development: A Practical Guide," arXiv preprint arXiv:2308.07077, 2023.

[25] K. Yellow, "Early Applications of Multi-Agent Systems in Network Management," International Journal of Network Management, vol. 28, no. 2, pp. 1-15, 2018.