EGR 4830 – Senior Project

CalPolyPomona

AIOPs-Driven Automation: Enhancing Kubernetes Management with AI and Ansible

Godinez, Ricardo
015670564

Dr. Valerio Formicola
Electrical and Computer Engineering
05/19/2025

AIOPs-Driven Automation: Enhancing Kubernetes Management with AI and Ansible

Introduction

Modern cloud-native applications rely on container orchestration platforms like Kubernetes (K8s) to ensure scalability, reliability, and efficient resource management. However, the complexity of these environments introduces significant challenges in real-time monitoring, fault detection, and automated remediation.

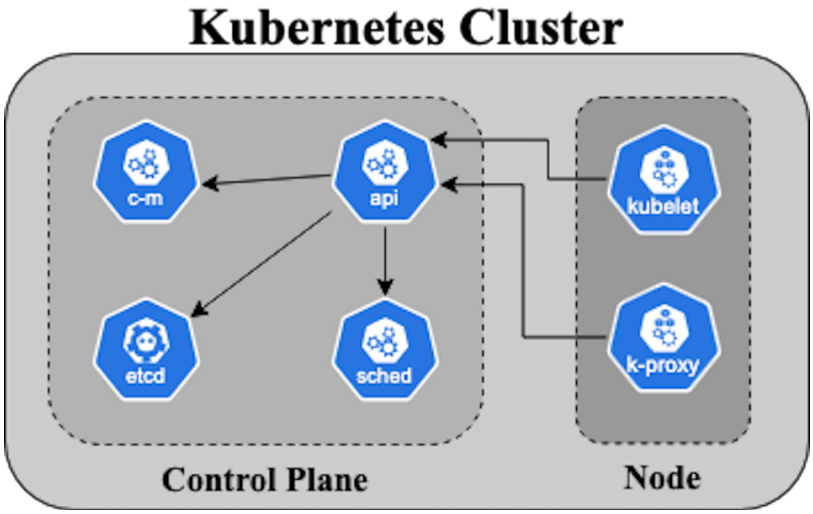
This senior design project aimed to develop a self-healing Kubernetes infrastructure that automatically detects service failures using Prometheus, visualizes system metrics with Grafana, and executes Ansible playbooks to restore functionality without manual intervention.

The system was implemented on CentOS-based virtual machines, chosen for their close alignment with Red Hat Enterprise Linux (RHEL) to realistically simulate enterprise-grade server environments. A host-only virtual network was configured using VMware Workstation to provide a controlled and isolated lab environment for simulation and testing.

The architecture was designed with extensibility in mind, incorporating a placeholder for a future AI-powered anomaly detection system that will leverage historical data to enable predictive remediation.

The project validated the feasibility of integrating these tools into a cohesive DevOps automation pipeline, demonstrating the potential for real-time, autonomous recovery workflows in production-like settings.

Kubernetes Control Plane and Node Interaction



In Kubernetes, the control plane is responsible for managing the overall state of the cluster, while the nodes (also known as worker nodes) are responsible for running application workloads in the form of containers. Kubernetes supports horizontal scalability, allowing users to dynamically scale the number of nodes in the cluster to meet varying resource demands.

Control Plane Components:

- API Server (api): Acts as the central communication hub between all components and external clients. All administrative commands pass through the API server.
- Controller Manager (c-m): Monitors the state of the cluster and ensures the desired state is maintained by managing controllers (e.g., node, replication).
- Scheduler (sched): Assigns new pods to appropriate nodes based on resource availability and scheduling policies.

- etcd: A distributed key-value store used to store the cluster's configuration and state data reliably.

Node Components:

- Kubelet: A node-level agent that communicates with the API server to receive pod specifications and ensure containers are running as instructed.
- Kube Proxy (k-proxy): Maintains network rules on each node to allow communication between services and pods.

Interaction Flow:

1. The API server receives instructions (e.g., deploy a pod) and communicates with the scheduler to place the pod on an appropriate node.
2. The controller manager ensures system stability by monitoring and acting on the current vs. desired cluster state.
3. Once scheduled, the kubelet on the assigned node manages the pod lifecycle based on instructions from the control plane.
4. The kube-proxy facilitates internal and external network communication for the pod.

Together, these components ensure that Kubernetes clusters are scalable, self-healing, and efficiently managed.

Literature Review

Container orchestration has become a foundational element of modern infrastructure, with Kubernetes emerging as the industry standard due to its flexibility, scalability, and strong community support [1]. To support observability in such dynamic environments, Prometheus provides a robust open-source monitoring solution capable of collecting real-time metrics and generating alerts based on defined conditions [2]. Grafana, often paired with Prometheus, is widely adopted for its powerful and customizable visualization capabilities, allowing operators to quickly interpret system health and performance trends [3].

To reduce operational overhead and enforce repeatability in system configuration, tools like Ansible have gained popularity in infrastructure management. Ansible's agentless, declarative approach enables consistent deployment and automated remediation across distributed systems [4].

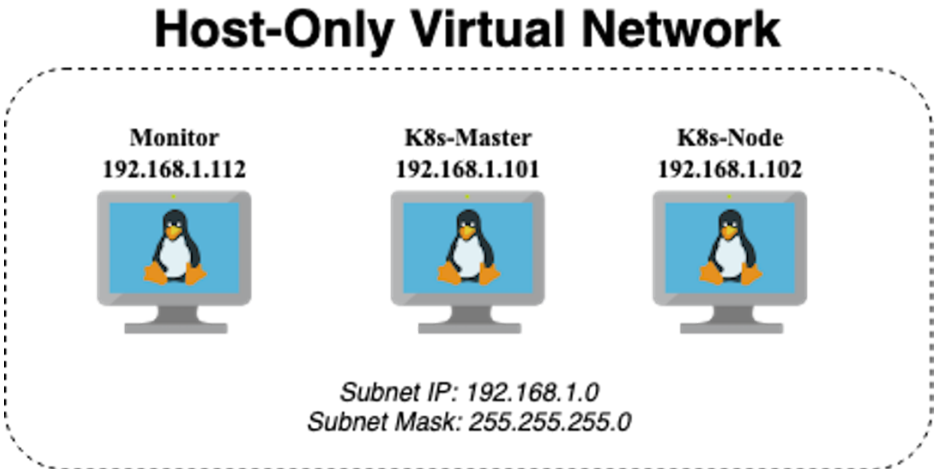
Recent advancements in AI-driven monitoring propose the use of machine learning models to detect anomalies and anticipate system failures. While promising, the full integration of AI into autonomous infrastructure management—from anomaly detection to automated response—remains limited in practice due to challenges in system complexity, data reliability, and trust in automated decisions [5].

This project addresses that gap by demonstrating a working framework that unifies monitoring, visualization, and automated remediation—while also laying the architectural groundwork for integrating predictive AI models into the decision-making process.

Results and Discussion

The system was deployed and tested on a virtualized testbed hosted on a Fedora Linux machine. The environment consisted of three CentOS-based virtual machines, each configured to simulate different roles within a Kubernetes ecosystem Both Fedora and CentOS are Red Hat Enterprise Linux distributions. A host-only virtual network was created using VMware Workstation to emulate an isolated infrastructure for research purposes. One virtual machine was configured as the Kubernetes master node, another as the worker node, and a third served as a dedicated monitoring machine. The monitoring VM hosted both Ansible and Grafana to manage automation and visualization independently from the Kubernetes cluster. This separation mirrors

best practices in production environments where observability and automation components are often isolated from core infrastructure.



Monitoring Stack Configuration

Prometheus was installed across the cluster to collect real-time system metrics including CPU usage, memory consumption, pod availability, and node health. Grafana, running on the monitoring VM, was configured to visualize these metrics using interactive dashboards that were accessed locally from the same monitoring virtual machine. Ansible was also installed on the monitoring VM and used to manage both the Kubernetes master and worker nodes remotely. It established secure connections to these nodes using the SSH protocol, allowing it to execute system-level commands such as checking service statuses and restarting failed services. This setup reflects a realistic deployment model where a dedicated operations or monitoring machine securely manages infrastructure components over the network without requiring physical access to each system.

Failure Injection and Recovery Test

To validate the system's self-healing capabilities, a failure injection test was conducted in which the node_exporter service—responsible for exposing system metrics to Prometheus—was manually stopped on the Kubernetes master node, simulating a critical service fault (see Figure 2). This intentional interruption halted the transmission of metrics, which was immediately reflected on the Grafana dashboard as missing data.

The Ansible playbook was executed automatically using a Bash script configured to run every minute via a cron job. (see Figure 3–5) This script checked whether the node_exporter service was active on the master and worker nodes. If the service was inactive, the script triggered the playbook to restart it. Additionally, the script logged the service status on each run, creating a time-stamped history of successful health checks and recovery actions. This design allowed for continuous monitoring, ensuring resilience in the automation process.

Logs captured on the monitoring and master node VM confirmed the full event sequence—from failure detection to successful service restoration—and a follow-up system check verified that node_exporter was again transmitting metrics to Prometheus (see Figure 6–7). Metric flow resumed on Grafana, restoring full system visibility.

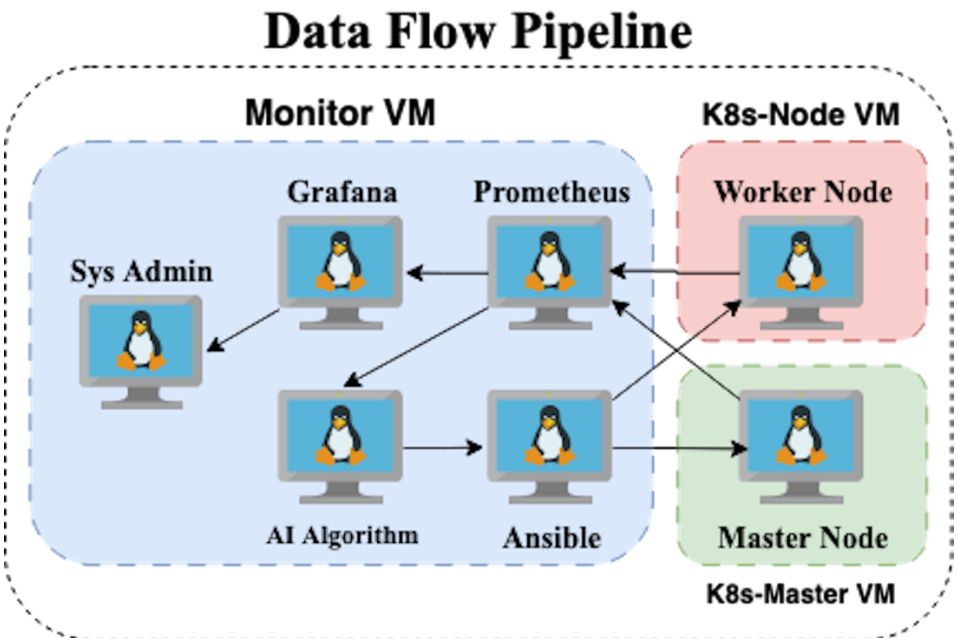


Figure 1. Self-Healing Kubernetes Monitoring Pipeline

```
master@k8s-master:~$ sudo systemctl stop node_exporter.service
master@k8s-master:~$ sudo systemctl status node_exporter.service
○ node_exporter.service - Node Exporter
     Loaded: loaded (/etc/systemd/system/node_exporter.service; enabled; preset: disabled)
     Active: inactive (dead) since Mon 2025-05-12 19:27:14 PDT; 18s ago
   Duration: 15min 42.743s
 Invocation: fdbaa69a19284da2bb8846c0a50a82e2
    Process: 978 ExecStart=/opt/node_exporter/node_exporter (code=killed, signal=TERM)
   Main PID: 978 (code=killed, signal=TERM)
   Mem peak: 29.6M
        CPU: 3.875s
May 12 19:11:32 k8s-master node_exporter[978]: time=2025-05-13T02:11:32.273Z level=INFO source=node_exporter.go:141 msg=vmstat
May 12 19:11:32 k8s-master node_exporter[978]: time=2025-05-13T02:11:32.273Z level=INFO source=node_exporter.go:141 msg=watchd>
May 12 19:11:32 k8s-master node_exporter[978]: time=2025-05-13T02:11:32.273Z level=INFO source=node_exporter.go:141 msg=xfs
May 12 19:11:32 k8s-master node_exporter[978]: time=2025-05-13T02:11:32.273Z level=INFO source=node_exporter.go:141 msg=zfs
May 12 19:11:32 k8s-master node_exporter[978]: time=2025-05-13T02:11:32.283Z level=INFO source=tls_config.go:347 msg="Listenin>
May 12 19:11:32 k8s-master node_exporter[978]: time=2025-05-13T02:11:32.284Z level=INFO source=tls_config.go:350 msg="TLS is d>
May 12 19:27:14 k8s-master systemd[1]: Stopping node_exporter.service - Node Exporter...
May 12 19:27:14 k8s-master systemd[1]: node_exporter.service: Deactivated successfully.
May 12 19:27:14 k8s-master systemd[1]: Stopped node_exporter.service - Node Exporter.
May 12 19:27:14 k8s-master systemd[1]: node_exporter.service: Consumed 3.875s CPU time, 29.6M memory peak.
lines 1-20/20 (END)
```

Figure 2. Manual Shutdown of node_exporter on Master Node VM

```yaml
---
- name: Recover Node Exporter Service
  hosts: all
  become: true
  tasks:
    - name: Check if node_exporter is running
      shell: systemctl is-active node_exporter
      register: exporter_status
      ignore_errors: true

    - name: Log status to /var/log/recovery.log
      lineinfile:
        path: /var/log/recovery.log
        create: yes
        line: "Checked node_exporter at {{ ansible_date_time.iso8601 }} - Status: {{ exporter_status.stdout }}"

    - name: Restart node_exporter if not running
      when: exporter_status.stdout != "active"
      systemd:
        name: node_exporter
        state: restarted
        enabled: true

    - name: Log restart action
      when: exporter_status.stdout != "active"
      lineinfile:
        path: /var/log/recovery.log
        create: yes
        line: "Restarted node_exporter at {{ ansible_date_time.iso8601 }}"
```

Figure 3. Ansible Playbook

```bash
#!/bin/bash

export PATH=/usr/local/sbin:/usr/local/bin:/usr/sbin:/usr/bin:/sbin:/bin

SSHPASS="/usr/bin/sshpass"
ANSIBLE="/home/monitor/.local/bin/ansible-playbook"

PASSWORD="ece"

INVENTORY="/home/monitor/ansible/hosts"
PLAYBOOK="/home/monitor/ansible/recover-node-exporter.yaml"
LOGFILE="/home/monitor/ansible/ansible.log"
DEBUGFILE="/home/monitor/ansible/debug.log"

# Debug
echo "Start: $(date)" >> "$DEBUGFILE"
echo "PATH: $PATH" >> "$DEBUGFILE"

# Log start
echo "Running Ansible Playbook at $(date)" >> "$LOGFILE"

# Run Ansible with sshpass
$SSHPASS -p "$PASSWORD" $ANSIBLE -i "$INVENTORY" "$PLAYBOOK" \
    --user=monitor \
    --ask-become-pass \
    --extra-vars "ansible_ssh_pass=$PASSWORD ansible_become_pass=$PASSWORD" >> "$LOGFILE" 2>&1

# Log end
echo "Finished at $(date)" >> "$LOGFILE"
```

Figure 4. Bash Script used in Cron Job

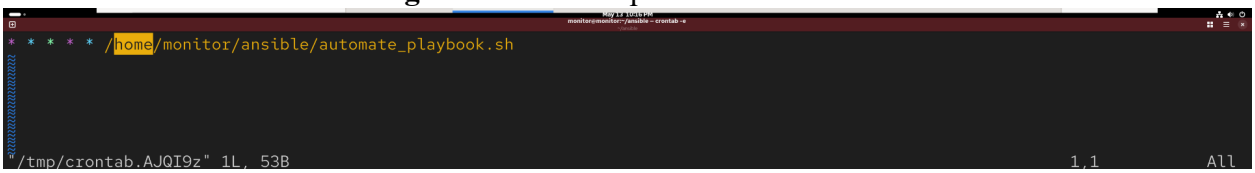


Figure 5. Cron Job Calling Script

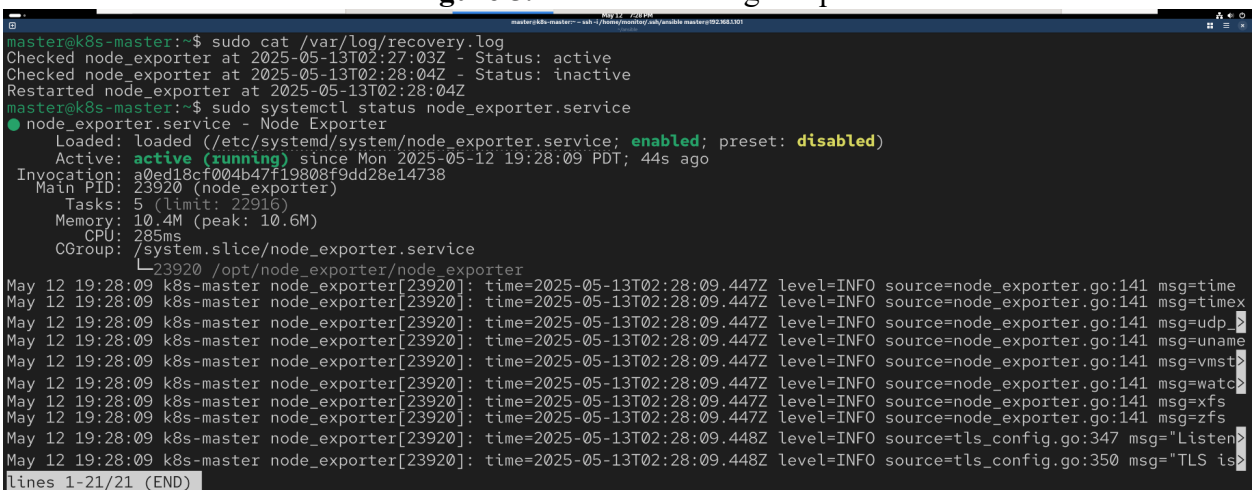


Figure 6. Recovery Log Showing Detection and Restart

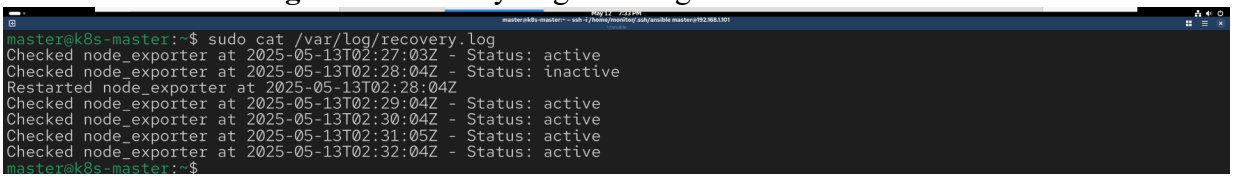


Figure 7. Verified node_exporter Status After Recovery

Simulated Experiment and Results

The experiment demonstrated that the system could autonomously detect and resolve service-level failures with high responsiveness. In practice, the time from failure detection to successful recovery was approximately 9 seconds, with an average recovery time of under 10 seconds across repeated tests. No manual intervention was required once the failure occurred. A cron job was configured to run every minute, executing a Bash script that triggered the Ansible playbook and created logs. This playbook checked whether the node_exporter service was running and restarted it if necessary.

These results confirm that the Prometheus-Ansible integration worked as intended and that the architecture is robust, responsive, and well-suited for environments where high availability is essential. Furthermore, this successful demonstration lays the groundwork for extending the system with AI-driven anomaly detection and predictive remediation in future iterations. This experiment demonstrated that the Prometheus-Ansible integration reliably restored failed services within 9 seconds of failure detection. The average recovery time was less than 10 seconds, making the system viable for deployment in environments requiring high uptime.

Table 1. Automated Recovery Log Summary

Event Description	Timestamp	Status
Ansible Playbook Executed	07:28:01 AM	Triggered
Service Failure Detected	07:28:04 PM	Inactive
Service Restarted	07:28:10 AM	Active
Metrics Restored in Grafana	07:28:10 AM	Normal

```
Running Ansible Playbook at Mon May 12 07:27:01 PM PDT 2025

PLAY [Recover Node Exporter Service] ***************************************

TASK [Gathering Facts] *****************************************************
ok: [192.168.1.102]
ok: [192.168.1.101]

TASK [Check if node_exporter is running] ***********************************
changed: [192.168.1.102]
changed: [192.168.1.101]

TASK [Log status to /var/log/recovery.log] *********************************
changed: [192.168.1.102]
changed: [192.168.1.101]

TASK [Restart node_exporter if not running] ********************************
skipping: [192.168.1.101]
skipping: [192.168.1.102]

TASK [Log restart action] **************************************************
skipping: [192.168.1.101]
skipping: [192.168.1.102]

PLAY RECAP *****************************************************************
192.168.1.101              : ok=3    changed=2    unreachable=0    failed=0
skipped=2    rescued=0    ignored=0
192.168.1.102              : ok=3    changed=2    unreachable=0    failed=0
skipped=2    rescued=0    ignored=0

Finished at Mon May 12 07:27:08 PM PDT 2025
```

Figure 8. Ansible Log Before Manual Failure

```
Running Ansible Playbook at Mon May 12 07:28:01 PM PDT 2025

PLAY [Recover Node Exporter Service] ***************************************

TASK [Gathering Facts] *****************************************************
ok: [192.168.1.102]
ok: [192.168.1.101]

TASK [Check if node_exporter is running] ***********************************
changed: [192.168.1.102]
fatal: [192.168.1.101]: FAILED! => {"changed": true, "cmd": "systemctl is-active
node_exporter", "delta": "0:00:00.018042", "end": "2025-05-12 19:28:06.258759",
"msg": "non-zero return code", "rc": 3, "start": "2025-05-12 19:28:06.240717",
"stderr": "", "stderr_lines": [], "stdout": "inactive", "stdout_lines":
["inactive"]}
...ignoring

TASK [Log status to /var/log/recovery.log] *********************************
changed: [192.168.1.102]
changed: [192.168.1.101]

TASK [Restart node_exporter if not running] ********************************
skipping: [192.168.1.102]
changed: [192.168.1.101]

TASK [Log restart action] **************************************************
skipping: [192.168.1.102]
changed: [192.168.1.101]

PLAY RECAP *****************************************************************
192.168.1.101              : ok=5    changed=4    unreachable=0    failed=0
skipped=0    rescued=0    ignored=1
192.168.1.102              : ok=3    changed=2    unreachable=0    failed=0
skipped=2    rescued=0    ignored=0

Finished at Mon May 12 07:28:10 PM PDT 2025
```

Figure 9. Ansible Log During Manual Failure

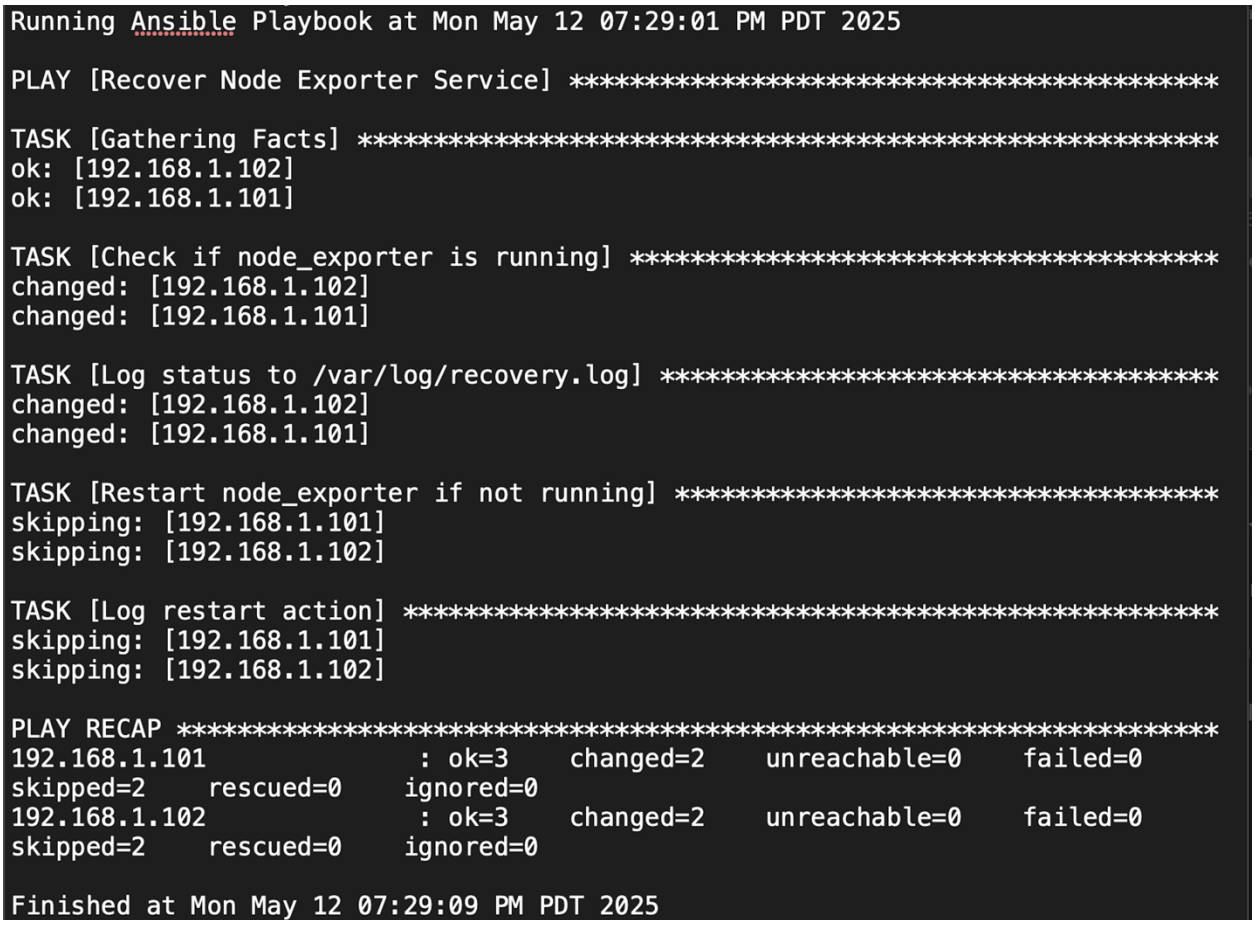


Figure 10. Ansible Log After Manual Failure

Conclusion

The integration of Prometheus, Grafana, and Ansible within a Kubernetes environment successfully demonstrated the feasibility of real-time monitoring and automated service recovery. The system exhibited high responsiveness, low-latency remediation, and a modular architecture conducive to scalability and future enhancements. Key benefits included minimized downtime, increased infrastructure reliability, and the elimination of manual intervention for routine service failures.

However, a notable limitation of the current implementation is the absence of an AI-driven anomaly detection component. Incorporating such functionality would enable the system to transition from reactive automation to proactive and predictive infrastructure management. Future work will involve developing a Python-based ETL pipeline to collect and preprocess historical performance metrics, training a machine learning model to classify anomalies, and integrating that model into the existing Ansible-driven automation workflow.

With these planned enhancements, the system has the potential to evolve into a fully autonomous infrastructure management solution—capable of intelligent, self-healing operations in alignment with modern DevOps methodologies and the increasing demands of cloud-native environments.

Work Cited

[1] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, and J. Wilkes, "Borg, Omega, and Kubernetes," *Communications of the ACM*, vol. 59, no. 5, pp. 50–57, May 2016.
[2] The Prometheus Authors, “Prometheus Documentation,” [Online]. Available: https://prometheus.io/docs/introduction/overview/
[3] Grafana Labs, “Grafana Documentation,” [Online]. Available: https://grafana.com/docs/
[4] Red Hat, “Ansible Documentation,” [Online]. Available: https://docs.ansible.com/
[5] M. Sculley, D. Holt, D. Golovin, et al., "Hidden Technical Debt in Machine Learning Systems," in *Proc. Advances in Neural Information Processing Systems (NIPS)*, 2015.