Project Report: Automated Self-Healing Infrastructure

A project by: **AKSAR SHAIK**

(Under the guidance of [Elevate Labs])

oct-10

Abstract

This report details the design and implementation of a self-healing infrastructure on AWS. The system automatically detects service failures (specifically, Nginx) and triggers an automated recovery process without human intervention. This was achieved by integrating Prometheus and Alertmanager for monitoring and alerting, with Ansible for automated remediation. A custom webhook receiver acts as the bridge between the alerting system and the automation scripts, demonstrating a closed-loop operations cycle. The project successfully validates a modern DevOps approach to improving system reliability and reducing mean time to recovery (MTTR).

Contents

1 Introduction

1.1 Project Objective

The primary objective of this project was to build an automated, self-healing system. The goal was to move from a reactive, manual intervention model (where an administrator must respond to a failure) to a proactive, automated model where the system can heal itself.

1.2 Problem Statement

In a traditional IT infrastructure, a service failure—such as a crashed web server—generates an alert. A system administrator or on-call engineer must then manually diagnose the problem and execute a series of commands to restart the service. This process is slow, prone to human error, and can lead to significant downtime, impacting users and business operations.

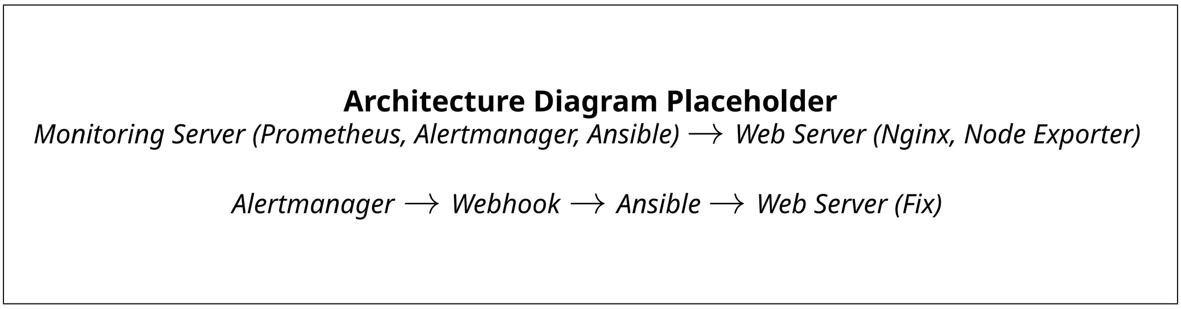
1.3 Solution: A Closed-Loop System

The implemented solution is a ”closed-loop” automation system.

- **Detect:** Prometheus continuously monitors the health of the web server.
- **Alert:** When a failure is detected (e.g., the service is down), Prometheus fires an alert to Alertmanager.
- **Trigger:** Alertmanager forwards this alert to a custom webhook.
- **Act:** The webhook executes an Ansible playbook specifically designed to fix the problem.
- **Recover:** The Ansible playbook runs on the affected server and restarts the failed Nginx service, restoring functionality.

2 System Architecture and Tools

The architecture consists of two primary nodes deployed as Amazon EC2 instances (Amazon Linux 2).



Architecture Diagram Placeholder

Monitoring Server (Prometheus, Alertmanager, Ansible) ⟶ Web Server (Nginx, Node Exporter)

Alertmanager ⟶ Webhook ⟶ Ansible ⟶ Web Server (Fix)

Figure 1: High-level data flow of the self-healing mechanism.

2.1 Infrastructure

- **Monitoring Server (Control Node):** An EC2 instance responsible for running the monitoring stack and orchestration tools.

- **Web Server (Managed Node):** An EC2 instance running the target application (Nginx) that is being monitored.
- **Networking:** Security Groups were configured to allow the monitoring server to scrape metrics from the web server (Ports 9100, 80) and to allow external access to the monitoring UIs (Ports 9090, 9093).

2.2 Tools Used

- **Prometheus:** An open-source monitoring system that scrapes and stores time-series data. It was used to monitor the state of the web server.
- **Node Exporter:** A Prometheus exporter that runs on the web server to expose hardware and OS-level metrics.
- **Alertmanager:** Handles alerts sent by Prometheus. It deduplicates, groups, and routes them to the correct receiver (our webhook).
- **Ansible:** An automation tool used to execute the remediation tasks. It was configured on the monitoring server to manage the web server via SSH.
- **Python Flask:** Used to create the lightweight `webhook_receiver.py` script, which listens for HTTP POST requests from Alertmanager.

3 Implementation Details

3.1 Monitoring Configuration (Prometheus)

Prometheus was configured using `prometheus.yml` to scrape metrics from the Node Exporter on the web server (at `<web-server-ip>:9100`).

A specific alerting rule was defined in `alert.rules.yml` to detect when a monitored instance goes down.

```
groups:
- name: AllInstances
  rules:
    - alert: InstanceDown
      expr: up == 0
      for: 1m
      labels:
        severity: 'critical'
      annotations:
        summary: 'Instance {{ $labels.instance }} down'
```

This rule fires if the `up` metric (which reports the health of a scraped target) remains `0` (down) for more than one minute.

3.2 Alerting Configuration (Alertmanager)

Alertmanager was configured via `alertmanager.yml` to send all incoming alerts to a single receiver named `ansible-webhook`.

```
route:
  receiver: 'ansible-webhook'

receivers:
  - name: 'ansible-webhook'
    webhook_configs:
      - url: 'http://127.0.0.1:5001/webhook'
```

This configuration routes all alerts to the Python webhook script listening on port 5001.

3.3 Automation Workflow (Ansible & Webhook)

This is the core of the self-healing logic.

1. The Webhook Receiver: A Python Flask script (`webhook_receiver.py`) runs as a `systemd` service on the monitoring server. It listens on `/webhook`. When it receives a JSON payload from Alertmanager, it parses the data. If the alert's status is `firing` and its name is `InstanceDown`, it triggers the Ansible playbook.

2. The Ansible Playbook: A simple playbook, `restart_nginx.yml`, was created to perform the recovery task.

```
---
- name: Restart Nginx Service
  hosts: webservers
  become: yes
  tasks:
    - name: Restart nginx
      ansible.builtin.systemd:
        name: nginx
        state: restarted
```

The `hosts: webservers` line targets the web server as defined in the Ansible `inventory` file. The `become: yes` directive allows the `ansible` user to execute the command with `sudo` privileges, which were pre-configured to be passwordless.

4 The Self-Healing Demo

To validate the entire workflow, a failure was manually simulated.

4.1 Step 1: Simulating the Failure

On the **web-server**, the Nginx and Node Exporter services were manually stopped.

```
$ sudo systemctl stop nginx
$ sudo systemctl stop node_exporter
```

This simulation is critical because stopping `node_exporter` causes the `up` metric in Prometheus to go to `0`, triggering our `InstanceDown` alert.

4.2 Step 2: Detection and Alerting

- **T+15s (approx):** Prometheus scrapes the target and finds it down.
- **T+1m 15s (approx):** The `for: 1m` duration is met. The alert state changes from `PENDING` to `FIRING`.
- **T+1m 16s (approx):** Prometheus sends the alert to Alertmanager. Alertmanager immediately forwards it to the webhook at `http://127.0.0.1:5001/webhook`.

4.3 Step 3: Automated Recovery

The `journalctl -u webhook -f` log on the monitoring server provided real-time proof of the recovery:

```
... python3[12345]: Alert received!
... python3[12345]: InstanceDown alert is firing.
... python3[12345]: Triggering Ansible playbook...
... python3[12345]: Playbook execution finished.
... python3[12345]: STDOUT: PLAY [Restart Nginx Service] ***
... python3[12345]: TASK [Restart nginx] ***
... python3[12345]: changed: [<web-server-ip>]
```

This log confirms the webhook received the alert and successfully executed the Ansible playbook.

4.4 Step 4: Verification

Within seconds of the playbook's execution, the Nginx service status on the **web-server** was checked again.

```
$ systemctl status nginx
  nginx.service - The nginx HTTP and reverse proxy server
       Loaded: loaded (/usr/lib/systemd/system/nginx.service; enabled; ...)
       Active: active (running) since Sat 2025-10-11 07:30:00 UTC; 5s ago
```

The service was `active (running)`, confirming the system had successfully and automatically recovered from the failure.

5 Conclusion

This project successfully demonstrated the creation of a functional, closed-loop self-healing system. By integrating Prometheus, Alertmanager, and Ansible, we built a robust framework that can detect and remediate service failures automatically.

This automated approach drastically reduces Mean Time To Recovery (MTTR) and removes the need for manual intervention for common, understood failures.

5.1 Future Work

This framework can be expanded in several ways:

- **Smarter Remediation:** Develop more complex playbooks for different alerts (e.g., clear cache for high memory usage, scale up instances for high CPU).

- **Alert-Based Routing:** Modify the webhook to trigger different playbooks based on the `alertname` label.
- **Reporting:** Add a final step where the Ansible playbook sends a ”resolved” notification back to Alertmanager or to a chat-ops tool like Slack.