Day 12 Internship Report

Contact Manager Application (Alerting)

Intern Name: Arsalan Sharief

Company: Signiance Technology

Role: DevOps Intern

Day: 12

Repository Link:
https://github.com/arsalan-signiance/internship-day10-11-12

Todays Goal

Implement a fully working monitoring and alerting pipeline connecting Amazon ECS, AWS Distro for OpenTelemetry (ADOT), and Amazon Managed Prometheus (AMP). The objective is to build and verify an end-to-end system.

ECS → ADOT → AMP → Alert Rule → AlertManager → SNS → Email

Today Tasks:

1. Backend Metrics Integration

- Added Prometheus metrics support to the Flask backend application.
- Utilized the `prometheus-flask-exporter` library for automatic instrumentation.
- Exposed the standard `/metrics` endpoint for scraping.
- Rebuilt the Docker image and pushed the updated revision to DockerHub.
- Updated the ECS task definition and forced a new deployment.

```
from prometheus_flask_exporter import PrometheusMetrics


metrics = PrometheusMetrics(app)
```

Result

Backend service now successfully exposes metrics at `/metrics`.

2. ADOT Collector Integration in ECS

- Modified ECS Task Definition to include a sidecar container.
- Used the official image: `public.ecr.aws/aws-observability/aws-otel-collector:latest`.
- Configured the collector via SSM Parameter Store to keep configuration decoupled.
- Enabled the Prometheus receiver to scrape local container metrics at `127.0.0.1:5000/metrics`.
- Configured the `remote_write` exporter to forward metrics to the AMP workspace.

Result

Metrics stream established flowing from ECS → AMP workspace.

3. Verified AMP Metric Ingestion

- Used AWS CloudShell to manually query the AMP workspace API.
- Executed PromQL query via `awscurl` to verify data presence.

```
awscurl --service aps ...
```

```
Query: up{job="contact-manager"}
```

Result

Received `value = 1`, confirming the job is healthy and ingestion is active.

4. Created SNS Topic for Alerting

- Provisioned a new SNS topic for critical alerts.
- Subscribed an operator email address to the topic.
- Confirmed subscription via the verification email link.
- Updated the SNS access policy to explicitly allow `aps.amazonaws.com` to publish messages.

Result

SNS topic is active and ready to bridge AMP alerts to email.

5. Configured AMP AlertManager

- Defined AlertManager configuration directly within the AMP workspace console.
- Set up the receiver to target the SNS topic ARN.
- Enabled `SigV4` authentication for secure AWS service-to-service communication.
- Configured `send_resolved: true` to ensure "All Clear" emails are sent.

Result

AlertManager is now configured to route firing alerts to SNS.

6. Created Alert Rule

- Created a recording rule group namespace in AMP.
- Defined a critical alert rule using PromQL to detect service downtime.
- Evaluation period set to trigger if the condition persists for 1 minute.

```
# Rule Definition
```

`up{job="contact-manager"} == 0`

Result

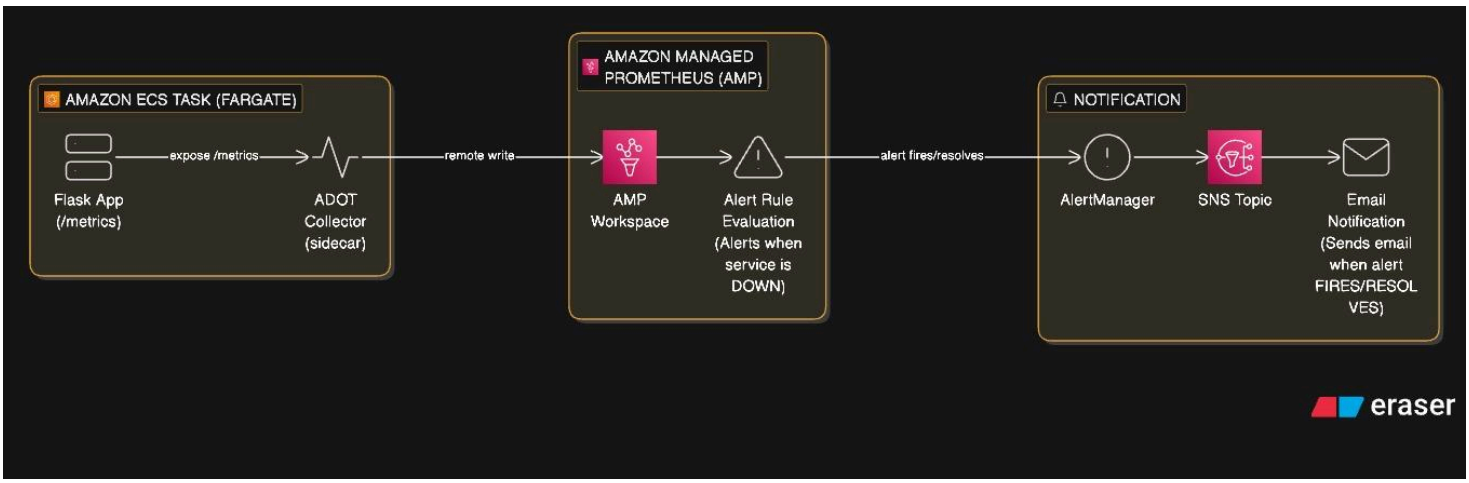
Namespace status is Active. The rule is currently evaluating metrics.

7. End-to-End Testing

- Conducted a manual trigger test to verify the full pipeline.
- Temporarily inverted the alert logic to `up{job="contact-manager"} == 1` to force a firing state on a healthy system.
- Observed the alert state transition to FIRING.
- Verified receipt of the email notification via SNS.
- Reverted the rule logic to the correct production setting to `== 0`.

Result: Email received successfully. The monitoring pipeline is fully operational.

Architecture Flow Implemented:



Result:

We have successfully implemented a production-grade monitoring and alerting system. The architecture leverages managed AWS services to minimize operational overhead while providing robust observability.

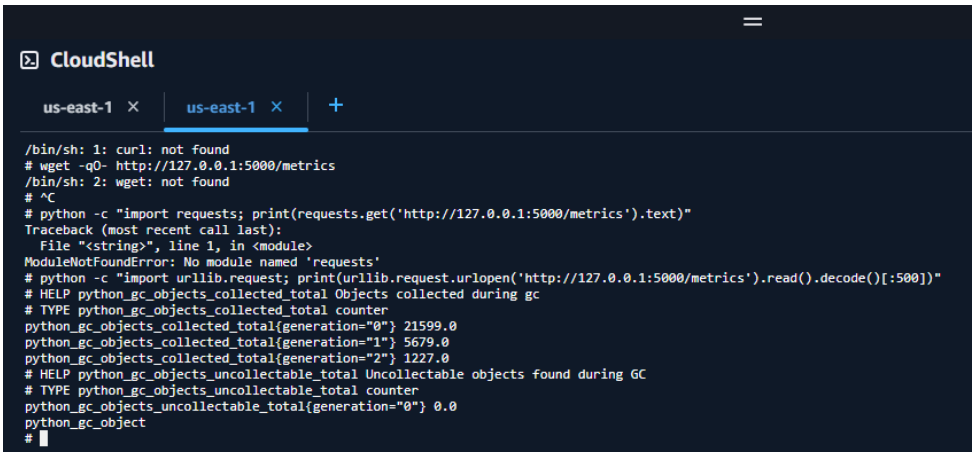
- Infrastructure: Amazon ECS (Fargate) with ADOT Sidecar.
- Storage & Evaluation: Amazon Managed Prometheus (AMP).
- Notification: AlertManager integrated with Amazon SNS.

The system is now capable of detecting service downtime immediately, triggering an alert, and notifying the operations team via email in real-time. It also supports auto-resolution notifications when the service recovers.

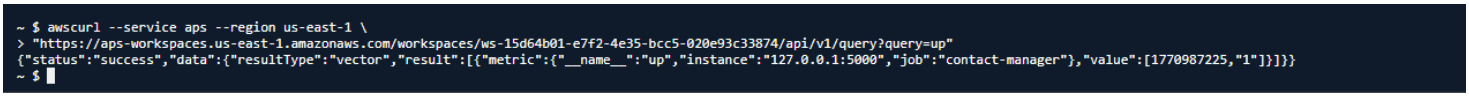
Today's Learning: Today's implementation provided deep practical experience in:

- Instrumenting Python Flask applications with Prometheus client libraries.
- Configuring the OpenTelemetry (ADOT) collector for metric scraping and forwarding.
- Setting up Amazon Managed Prometheus workspaces and rule groups.
- Writing PromQL queries for both health checks and alerting rules.
- Integrating AWS AlertManager with SNS for external notifications.
- Debugging distributed observability pipelines end-to-end.

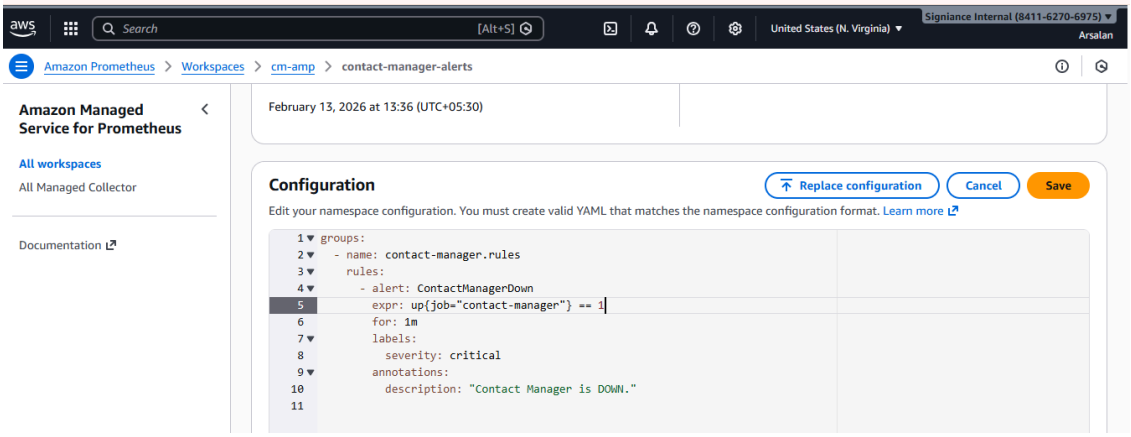
Screenshots:



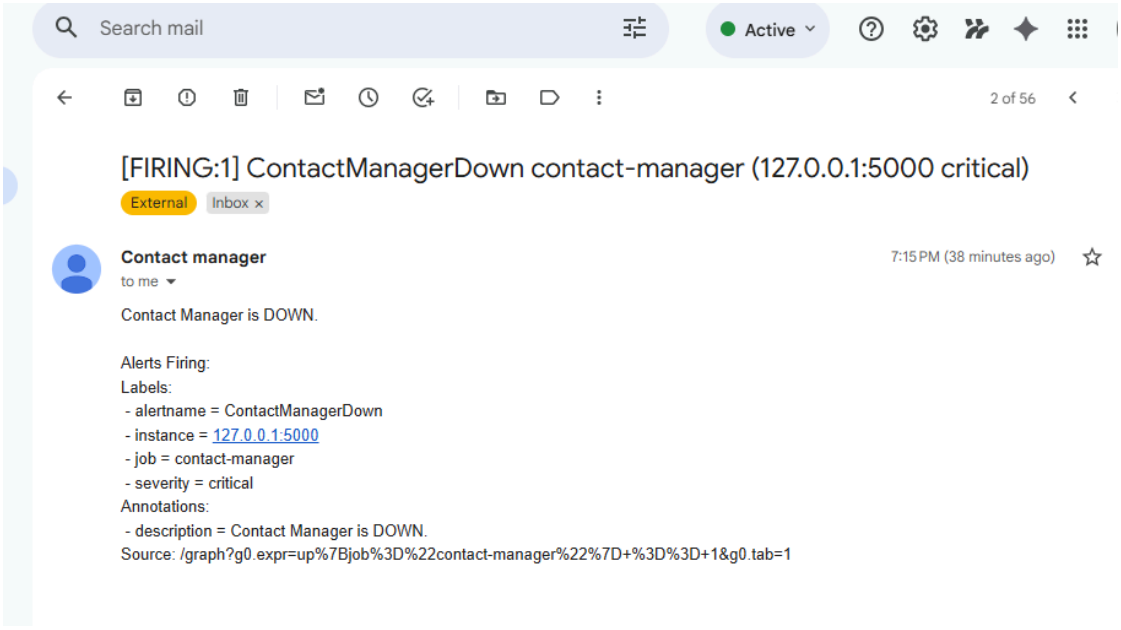
CloudShell terminal showing a local metrics endpoint (http://127.0.0.1:5000/metrics) being queried and returning Prometheus metrics like python_gc_objects_collected_total.



AWS CLI query to Amazon Managed Prometheus returning a successful result for the up{job="contact-manager"} metric with value 1.



Amazon Managed Service for Prometheus alert rule configuration defining a ContactManagerDown alert with expression up{job="contact-manager"} == 1 and severity critical.



Email alert notification showing a firing alert [FIRING:1] ContactManagerDown indicating the Contact Manager service is reported as DOWN.

Prepared by: ArsalanSharief
DevOps Intern – Signiance Technology