Complete Monitoring and Observability Setup Guide

This guide provides step-by-step instructions to implement comprehensive monitoring for ThreatCompass using the provided configurations.

Phase 1: Prerequisites and Environment Setup

1. Install Required Python Packages

Add these dependencies to your `requirements.txt`:

txt

```
# Logging and monitoring
python-json-logger==2.0.7
sentry-sdk[flask]==1.32.0

# AWS CloudWatch
boto3==1.28.57
watchtower==3.0.1

# Performance monitoring (optional)
opentelemetry-api==1.20.0
opentelemetry-sdk==1.20.0
opentelemetry-instrumentation-flask==0.41b0
opentelemetry-instrumentation-requests==0.41b0
opentelemetry-instrumentation-sqlalchemy==0.41b0
```

2. Environment Variables

Add these to your production environment:

```bash
# Sentry Configuration
SENTRY_DSN=https://your-sentry-dsn@sentry.io/project-id
SENTRY_ENVIRONMENT=production
APP_VERSION=1.0.0

# CloudWatch Configuration
CLOUDWATCH_LOG_GROUP=/aws/application/threatcompass
AWS_REGION=us-east-1

# Alert Configuration
ALERT_EMAIL=admin@yourdomain.com
SLACK_WEBHOOK_URL=https://hooks.slack.com/services/YOUR/SLACK/WEBHC
```

Phase 2: Implement Structured Logging

1. Update Your Flask Application

Modify your main `app.py` file:

python

```python
# app.py
from flask import Flask
from logging_config import setup_logging, setup_correlation_id_middleware
from cloudwatch_metrics import setup_flask_metrics_middleware, metrics
from sentry_integration import setup_sentry

def create_app():
    app = Flask(__name__)

    # Load configuration
    app.config.from_object('config.ProductionConfig')

    # Set up structured logging
    logger = setup_logging(app)

    # Set up correlation ID middleware
    setup_correlation_id_middleware(app)

    # Set up metrics collection
    setup_flask_metrics_middleware(app)

    # Set up error tracking
    setup_sentry(app)

    # Register blueprints
    from routes import main_bp
    from api import api_bp
    app.register_blueprint(main_bp)
```

```
    app.register_blueprint(api_bp)

    return app

if __name__ == '__main__':
    app = create_app()
    app.run()
```

2. Update Celery Configuration

Modify your `celery_app.py`:

python

```python
# celery_app.py
from celery import Celery
from logging_config import setup_celery_logging, setup_celery_correlation_signals
from cloudwatch_metrics import setup_celery_metrics
from sentry_integration import setup_celery_sentry_context

def create_celery_app():
    celery = Celery('threatcompass')
    celery.config_from_object('celeryconfig')

    # Set up structured logging for Celery
    setup_celery_logging()

    # Set up correlation ID signals
    setup_celery_correlation_signals(celery)

    # Set up metrics collection
    setup_celery_metrics(celery)

    # Set up error tracking
    setup_celery_sentry_context(celery)

    return celery

celery = create_celery_app()
```

3. Update Your Route Handlers

Example of instrumenting your API routes:

python

```python
# routes.py
from flask import Blueprint, request, jsonify
from flask_login import login_required, current_user
from logging_config import ThreatCompassLogger, log_performance_metric, log_
from cloudwatch_metrics import ThreatCompassMetrics, measure_time
from sentry_integration import track_errors, track_performance
import time

main_bp = Blueprint('main', __name__)
logger = ThreatCompassLogger(__name__)

@main_bp.route('/api/v1/iocs', methods=['POST'])
@login_required
@track_errors('ioc_creation', 'api')
@track_performance(threshold_ms=3000)
@measure_time('IOCCreationTime', 'API')
def create_ioc():
    start_time = time.time()

    try:
        # Your existing IOC creation logic
        ioc_data = request.json
        ioc = process_ioc(ioc_data)

        # Log business event
        log_business_event(
            'ioc_created',
            ioc_type=ioc.type,
```

```
            ioc_source=ioc.source,
            user_id=current_user.id,
            tenant_id=current_user.tenant_id
        )

        # Record CloudWatch metrics
        ThreatCompassMetrics.record_ioc_processed(
            ioc_type=ioc.type,
            source=ioc.source,
            tenant_id=current_user.tenant_id,
            success=True
        )

        # Log successful creation
        duration = (time.time() - start_time) * 1000
        logger.info(
            "IOC created successfully",
            ioc_id=ioc.id,
            ioc_type=ioc.type,
            duration_ms=duration,
            user_id=current_user.id,
            tenant_id=current_user.tenant_id
        )

        return jsonify({"status": "success", "id": ioc.id})

    except ValidationError as e:
        # Record failure metrics
        ThreatCompassMetrics.record_ioc_processed(
```

```
            ioc_type=ioc_data.get('type', 'unknown'),
            source=ioc_data.get('source', 'unknown'),
            tenant_id=current_user.tenant_id,
            success=False
        )

        logger.error(
            "IOC creation validation failed",
            validation_errors=str(e),
            user_id=current_user.id,
            tenant_id=current_user.tenant_id
        )

        return jsonify({"error": "Validation failed"}), 400

    except Exception as e:
        # Log and track the error
        duration = (time.time() - start_time) * 1000
        logger.error(
            "IOC creation failed",
            error_message=str(e),
            error_type=type(e).__name__,
            duration_ms=duration,
            user_id=current_user.id,
            tenant_id=current_user.tenant_id
        )

        return jsonify({"error": "Internal server error"}), 500
```

Phase 3: Deploy Infrastructure Monitoring

1. Apply Terraform Configuration

Add the new monitoring resources to your Terraform:

```bash
# Add to your existing Terraform configuration
cp cloudwatch_dashboards.tf terraform/
cp cloudwatch_alarms.tf terraform/

# Add variables to terraform.tfvars
echo 'alert_email = "admin@yourdomain.com"' >> terraform/terraform.tfvars
echo 'slack_webhook_url = "https://hooks.slack.com/services/YOUR/SLACK/WEBH
echo 'enable_enhanced_alerting = true' >> terraform/terraform.tfvars

# Apply the changes
cd terraform
terraform plan
terraform apply
```

2. Configure Sentry

1. **Create Sentry Project**:
   - Sign up at sentry.io
   - Create a new project for ThreatCompass

- Copy the DSN

2. **Add Sentry DSN to Secrets Manager**:

bash

```
aws secretsmanager update-secret \
  --secret-id threatcompass-production/app/secrets \
  --secret-string '{
    "flask_secret_key": "your-existing-key",
    "sentry_dsn": "https://your-sentry-dsn@sentry.io/project-id",
    "virustotal_api_key": "your-vt-key",
    "abuseipdb_api_key": "your-abuse-key"
  }'
```

3. Update Docker Images

Rebuild and deploy your Docker images with the new monitoring code:

bash

```
# Build new images with monitoring
docker build -f docker/Dockerfile.flask-app -t threatcompass-flask-app:monitoring
docker build -f docker/Dockerfile.celery-worker -t threatcompass-celery-worker:m
docker build -f docker/Dockerfile.celery-beat -t threatcompass-celery-beat:monitc

# Push to ECR and deploy via your CI/CD pipeline
# Or manually update ECS services
```

Phase 4: Configure Dashboards and Alerts

1. Access CloudWatch Dashboards

After Terraform applies the configuration, you'll have several dashboards:

1. **Application Health**: `threatcompass-production-application-health`
2. **Application Performance**: `threatcompass-production-application-performance`
3. **Database Performance**: `threatcompass-production-database-performance`
4. **Background Tasks**: `threatcompass-production-background-tasks`
5. **Security Audit**: `threatcompass-production-security-audit`
6. **Cost Optimization**: `threatcompass-production-cost-optimization`

2. Configure Alert Recipients

Set up SNS topic subscriptions:

bash

```
# Subscribe to critical alerts
aws sns subscribe \
    --topic-arn arn:aws:sns:us-east-1:123456789012:threatcompass-production-cr
    --protocol email \
    --notification-endpoint admin@yourdomain.com

# Subscribe to warning alerts
aws sns subscribe \
    --topic-arn arn:aws:sns:us-east-1:123456789012:threatcompass-production-w
    --protocol email \
    --notification-endpoint devops@yourdomain.com

# Confirm subscriptions via email
```

3. Set Up Slack Integration

Create a Lambda function for Slack notifications:

python

```python
# slack_alert_processor.py
import json
import urllib3
import os

def lambda_handler(event, context):
    """Process CloudWatch alarms and send to Slack."""

    webhook_url = os.environ['SLACK_WEBHOOK_URL']

    # Parse SNS message
    sns_message = json.loads(event['Records'][0]['Sns']['Message'])

    alarm_name = sns_message.get('AlarmName', 'Unknown')
    alarm_description = sns_message.get('AlarmDescription', '')
    new_state = sns_message.get('NewStateValue', 'UNKNOWN')
    reason = sns_message.get('NewStateReason', '')

    # Determine color based on alarm state
    color = {
        'ALARM': 'danger',
        'OK': 'good',
        'INSUFFICIENT_DATA': 'warning'
    }.get(new_state, 'warning')

    # Create Slack message
    slack_message = {
        'text': f'ThreatCompass Alert: {alarm_name}',
```

```
    'attachments': [
        {
            'color': color,
            'fields': [
                {
                    'title': 'Alarm',
                    'value': alarm_name,
                    'short': True
                },
                {
                    'title': 'State',
                    'value': new_state,
                    'short': True
                },
                {
                    'title': 'Description',
                    'value': alarm_description,
                    'short': False
                },
                {
                    'title': 'Reason',
                    'value': reason,
                    'short': False
                }
            ]
        }
    ]
}
```

```
    # Send to Slack
    http = urllib3.PoolManager()
    response = http.request(
        'POST',
        webhook_url,
        body=json.dumps(slack_message).encode('utf-8'),
        headers={'Content-Type': 'application/json'}
    )

    return {
        'statusCode': 200,
        'body': json.dumps('Alert sent to Slack')
    }
```

Phase 5: Testing and Validation

1. Test Structured Logging

```
bash

# Check CloudWatch logs
aws logs tail /ecs/threatcompass-production/flask-app --follow

# Look for JSON-formatted logs with correlation IDs
```

2. Test Custom Metrics

Create a test script to verify metrics:

```python
# test_metrics.py
from cloudwatch_metrics import ThreatCompassMetrics
import time

# Test business metrics
ThreatCompassMetrics.record_ioc_processed('IP_ADDRESS', 'test', tenant_id=1, s
ThreatCompassMetrics.record_playbook_generated(tenant_id=1, step_count=5)

# Wait and check CloudWatch console
time.sleep(60)
print("Check CloudWatch metrics console for ThreatCompass namespace")
```

3. Test Error Tracking

Generate a test error to verify Sentry integration:

```python
# In your Flask route, temporarily add:
@app.route('/test-error')
def test_error():
    raise Exception("Test error for Sentry integration")
```

4. Test Alerting

Trigger an alarm to test the alerting system:

```bash
# Temporarily scale down Flask app to trigger alarm
aws ecs update-service \
  --cluster threatcompass-production-cluster \
  --service threatcompass-production-flask-app \
  --desired-count 0

# Wait for alarm to trigger, then restore
aws ecs update-service \
  --cluster threatcompass-production-cluster \
  --service threatcompass-production-flask-app \
  --desired-count 2
```

Phase 6: Ongoing Monitoring Operations

1. Daily Monitoring Checklist

- [ ] Check CloudWatch dashboards for anomalies
- [ ] Review Sentry error reports
- [ ] Verify all ECS services are healthy
- [ ] Check RDS and Redis performance metrics
- [ ] Review security logs for suspicious activity

2. Weekly Monitoring Tasks

- [ ] Analyze performance trends
- [ ] Review and tune alarm thresholds

- [ ] Check cost optimization dashboard
- [ ] Update alert recipient lists
- [ ] Review and clear old logs

3. Monthly Monitoring Review

- [ ] Analyze month-over-month metrics trends
- [ ] Review and optimize dashboard layouts
- [ ] Update monitoring documentation
- [ ] Conduct incident response drills
- [ ] Plan capacity scaling based on trends

Phase 7: Advanced Configuration Options

1. Custom CloudWatch Metrics for Business Logic

python

```python
# business_metrics.py
from cloudwatch_metrics import metrics

def track_user_onboarding_completion(tenant_id, completion_time_minutes):
    """Track onboarding completion metrics."""
    metrics.put_metric(
        'OnboardingCompletion',
        1,
        'Count',
        dimensions={
            'TenantId': str(tenant_id),
            'CompletionTimeRange': get_time_range(completion_time_minutes)
        }
    )

    metrics.put_metric(
        'OnboardingDuration',
        completion_time_minutes,
        'Minutes',
        dimensions={'TenantId': str(tenant_id)}
    )

def get_time_range(minutes):
    """Categorize completion times."""
    if minutes < 10:
        return 'Fast'
    elif minutes < 30:
        return 'Normal'
```

```
    else:
        return 'Slow'
```

2. Enhanced Sentry Configuration

```
python

# sentry_enhanced.py
import sentry_sdk
from sentry_sdk.integrations.sqlalchemy import SqlalchemyIntegration

# Custom integration for database query tracking
class DatabaseQueryIntegration(SqlalchemyIntegration):
    def __init__(self, **kwargs):
        super().__init__(**kwargs)

    def setup_once(self):
        super().setup_once()
        # Add custom database monitoring
```

This comprehensive monitoring setup provides:

✅ **Complete Observability**: Logs, metrics, traces, and error tracking
✅ **Proactive Alerting**: Real-time notifications for issues
✅ **Performance Monitoring**: Detailed application and infrastructure metrics
✅ **Security Monitoring**: Audit trails and security event tracking

✅ **Cost Optimization**: Resource utilization monitoring

✅ **Business Intelligence**: Customer usage and feature adoption metrics

Your ThreatCompass platform now has enterprise-grade monitoring that will help you maintain high availability, quickly diagnose issues, and continuously optimize performance.