AWS CloudWatch: Complete Practical Guide

Table of Contents

What is AWS CloudWatch?

Amazon CloudWatch is AWS's unified monitoring and observability service that provides real-time visibility into your AWS resources, applications, and services. It collects and tracks metrics, monitors log files, sets alarms, and automatically reacts to changes in your AWS environment [1][2].

Key Capabilities

Capability	Description
Metrics Monitoring	Collect and track performance data from AWS services and custom applications
Log Management	Aggregate, store, search, and analyze log data from multiple sources
Alarms & Notifications	Set thresholds and trigger automated actions or notifications
Dashboards	Visualize metrics and logs in customizable, shareable dashboards
Application Monitoring	Track application performance, errors, and latency
Infrastructure Insights	Deep visibility into containers, Lambda, databases, and more

Why CloudWatch Matters for Operations Engineers

- **Unified Observability**: Single pane of glass for all your monitoring needs
- **Proactive Issue Detection**: Catch problems before they impact users
- **Automated Responses**: Trigger actions like auto-scaling or Lambda functions
- **Root Cause Analysis**: Correlate metrics and logs to troubleshoot faster
- **Cost Management**: Track resource utilization and optimize spending

Core Concepts & Terminology

CloudWatch Architecture

CloudWatch follows a hierarchical structure for organizing monitoring data:

```
AWS Account
  └── Namespaces (e.g., AWS/EC2, AWS/Lambda, Custom/MyApp)
       └── Metrics (e.g., CPUUtilization, Errors)
            └── Dimensions (e.g., InstanceId=i-1234567)
                 └── Data Points (timestamp + value + unit)
```

1. Namespaces

Definition: A container for CloudWatch metrics, used to isolate metrics from different applications or services [2].

Key Points:

- AWS services use the prefix `AWS/` (e.g., `AWS/EC2`, `AWS/Lambda`, `AWS/RDS`)
- Custom metrics use custom namespaces (e.g., `MyCompany/MyApp`)
- No default namespace exists—you must always specify one

Example:

```
# AWS service namespace
AWS/EC2

# Custom namespace
Production/WebApp
```

2. Metrics

Definition: A time-ordered set of data points representing a variable you want to monitor (CPU usage, request count, error rate, etc.) [2].

Metric Components:

- **Name**: Identifier for the metric (e.g., `CPUUtilization`)
- **Namespace**: The service or application the metric belongs to
- **Dimensions**: Key-value pairs that identify unique metric streams
- **Timestamp**: When the data point was recorded
- **Value**: The numerical measurement
- **Unit**: The measurement unit (Percent, Bytes, Count, etc.)

Example:

```
{
  "Namespace": "AWS/EC2",
  "MetricName": "CPUUtilization",
  "Dimensions": [
    {"Name": "InstanceId", "Value": "i-1234567890abcdef0"}
  ],
  "Timestamp": "2025-10-28T10:00:00Z",
  "Value": 75.5,
  "Unit": "Percent"
}
```

3. Dimensions

Definition: Name-value pairs that uniquely identify a metric, allowing you to filter and aggregate data [2].

Common Dimensions:

Service	Common Dimensions
EC2	InstanceId, InstanceType, ImageId, AutoScalingGroupName
Lambda	FunctionName, Resource, ExecutedVersion
RDS	DBInstanceIdentifier, DBClusterIdentifier
ELB	LoadBalancerName, AvailabilityZone, TargetGroup

Example:

```
# View CPU for specific EC2 instance
Dimension: InstanceId=i-123456

# View CPU across all instances in Auto Scaling group
Dimension: AutoScalingGroupName=my-asg
```

4. Statistics

Definition: Aggregations of metric data over specified time periods [2].

Available Statistics:

- **Sum**: Total of all values
- **Average**: Mean value
- **Minimum**: Lowest value
- **Maximum**: Highest value
- **SampleCount**: Number of data points
- **pXX (Percentiles)**: p50, p90, p95, p99

When to Use Each:

Statistic	Best For
Average	CPU usage, memory usage
Sum	Request counts, error counts
Maximum	Worst-case latency, peak load
Minimum	Best-case performance
p99	Understanding tail latency

5. Periods

Definition: The length of time to aggregate metric data [2].

Available Periods:

- **High-resolution metrics**: 1 second, 5 seconds, 10 seconds, 30 seconds
- **Standard resolution**: 1 minute, 5 minutes, 15 minutes, 30 minutes, 1 hour, etc.

Example:

```
# Get average CPU over 5-minute periods
Period: 300 seconds (5 minutes)

# High-resolution metric every 1 second
Period: 1 second
```

6. Resolution

Definition: The granularity at which metrics are stored [2].

Types:

- **Standard Resolution**: 1-minute granularity (free for AWS service metrics)

- **High Resolution**: Up to 1-second granularity (custom metrics only, additional cost)

Use Cases:

- **Standard**: Most monitoring scenarios, cost-effective
- **High Resolution**: Real-time monitoring, rapid auto-scaling decisions

CloudWatch Logs

Log Hierarchy

CloudWatch Logs uses a three-tier hierarchy [3]:

```
Log Group
  └── Log Stream 1
       └── Log Event 1 (timestamp + message)
       └── Log Event 2
       └── Log Event 3
  └── Log Stream 2
       └── Log Event 1
       └── Log Event 2
```

1. Log Events

Definition: A single log entry with a timestamp and raw log message [4].

Components:

- **Timestamp**: When the event occurred (milliseconds since epoch)
- **Message**: The actual log text (up to 256 KB)

Example:

```
{
  "timestamp": 1698480000000,
  "message": "2025-10-28 10:00:00 ERROR Database connection failed"
}
```

2. Log Streams

Definition: A sequence of log events from the same source [3][4].

Key Characteristics:

- One stream per source (e.g., one Lambda execution environment, one EC2 instance)
- Events in a stream are ordered by timestamp
- Automatically created when logs are sent

Naming Convention:

```
# Lambda function
2025/10/28/[$LATEST]a1b2c3d4

# EC2 instance
i-1234567890abcdef0

# Custom application
webserver-01/access.log
```

3. Log Groups

Definition: A container for log streams with shared retention, monitoring, and access control settings [3].

Key Features:

- **Retention Policies**: 1 day to 10 years, or indefinite
- **Encryption**: Optional KMS encryption at rest
- **Metric Filters**: Extract metrics from log data
- **Subscriptions**: Stream logs to other services (Kinesis, Lambda, Elasticsearch)

Naming Convention for AWS Services:

```
# Lambda functions
/aws/lambda/function-name

# RDS databases
/aws/rds/instance/db-instance-name/error

# API Gateway
/aws/apigateway/api-id

# ECS containers
/ecs/container-name
```

Creating Log Groups

Method 1: AWS Console

Step-by-Step:

1. Navigate to CloudWatch Console → **Log groups**
2. Click **Actions** → **Create log group**
3. Enter log group name (e.g., `/myapp/production/api`)
4. (Optional) Configure retention period
5. (Optional) Enable KMS encryption
6. Click **Create log group**

Method 2: AWS CLI

```
# Create a log group
aws logs create-log-group \
    --log-group-name /myapp/production/api \
    --region us-east-1

# Create with tags
aws logs create-log-group \
    --log-group-name /myapp/production/api \
    --tags Environment=Production,Application=MyApp \
    --region us-east-1

# Set retention policy (7 days)
aws logs put-retention-policy \
    --log-group-name /myapp/production/api \
    --retention-in-days 7
```

Method 3: Terraform

```
resource "aws_cloudwatch_log_group" "app_logs" {
  name              = "/myapp/production/api"
  retention_in_days = 7

  kms_key_id = aws_kms_key.log_encryption.arn

  tags = {
    Environment = "Production"
    Application = "MyApp"
  }
}
```

Sending Logs to CloudWatch

Method 1: CloudWatch Agent

Installation:

```
# Download agent (Amazon Linux 2)
wget https://s3.amazonaws.com/amazoncloudwatch-agent/amazon_linux/amd64/latest/amazon-clo
sudo rpm -U ./amazon-cloudwatch-agent.rpm

# Configure agent
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-config-wizard
```

Configuration File (`/opt/aws/amazon-cloudwatch-agent/etc/config.json`):

```
{
  "logs": {
    "logs_collected": {
```

```
      "files": {
        "collect_list": [
          {
            "file_path": "/var/log/app/application.log",
            "log_group_name": "/myapp/production/api",
            "log_stream_name": "{instance_id}",
            "timezone": "UTC"
          }
        ]
      }
    }
  }
}
```

Start Agent:

```
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl \
    -a fetch-config \
    -m ec2 \
    -s \
    -c file:/opt/aws/amazon-cloudwatch-agent/etc/config.json
```

Method 2: AWS CLI

```
# Create log stream
aws logs create-log-stream \
    --log-group-name /myapp/production/api \
    --log-stream-name webserver-01

# Send log events
aws logs put-log-events \
    --log-group-name /myapp/production/api \
    --log-stream-name webserver-01 \
    --log-events \
        timestamp=1698480000000,message="Application started" \
        timestamp=1698480060000,message="User logged in: user123"
```

Method 3: Python Boto3

```
import boto3
import time

# Create CloudWatch Logs client
logs = boto3.client('logs', region_name='us-east-1')

log_group = '/myapp/production/api'
log_stream = f'python-app-{int(time.time())}'

# Create log stream
logs.create_log_stream(
    logGroupName=log_group,
```

```
    logStreamName=log_stream
)

# Send log events
logs.put_log_events(
    logGroupName=log_group,
    logStreamName=log_stream,
    logEvents=[
        {
            'timestamp': int(time.time() * 1000),
            'message': 'Application started successfully'
        },
        {
            'timestamp': int(time.time() * 1000) + 1000,
            'message': 'Database connection established'
        }
    ]
)
```

Viewing and Searching Logs

Console Method

1. Navigate to **CloudWatch** → **Log groups**
2. Select your log group
3. Choose a log stream
4. Use the filter box to search:

```
ERROR
"Database connection failed"
[level = ERROR]
```

CloudWatch Logs Insights

What is it?: A powerful query language for analyzing log data [2].

Access: CloudWatch Console → **Logs** → **Insights**

Example Queries:

```
# Find all ERROR logs in the last hour
fields @timestamp, @message
| filter @message like /ERROR/
| sort @timestamp desc
| limit 20

# Count errors by type
fields @message
| filter @message like /ERROR/
| parse @message "ERROR: * - *" as errorType, errorMsg
| stats count() by errorType
```

```
# Calculate average response time
fields @timestamp, responseTime
| filter @message like /Request completed/
| parse @message "Request completed in *ms" as responseTime
| stats avg(responseTime) as avgResponseTime

# Find slowest requests
fields @timestamp, @message, duration
| filter @type = "REPORT"
| sort duration desc
| limit 10
```

Log Retention

Setting Retention:

```
# Set to 7 days
aws logs put-retention-policy \
    --log-group-name /myapp/production/api \
    --retention-in-days 7

# Set to never expire
aws logs delete-retention-policy \
    --log-group-name /myapp/production/api
```

Available Retention Periods:

- 1, 3, 5, 7, 14, 30, 60, 90, 120, 150, 180, 365, 400, 545, 731, 1827, 2192, 2557, 2922, 3288, 3653 days
- Never expire (indefinite)

Cost Impact: Longer retention = higher storage costs

CloudWatch Metrics

Understanding Metrics

Metrics are the fundamental concept in CloudWatch for tracking performance data [2].

Built-in AWS Service Metrics

AWS automatically publishes metrics for many services at no additional charge:

Service	Key Metrics	Frequency
EC2	CPUUtilization, NetworkIn/Out, DiskReadOps, StatusCheckFailed	5 minutes (1 minute with detailed monitoring)
Lambda	Invocations, Duration, Errors, Throttles, ConcurrentExecutions	Real-time

Service	Key Metrics	Frequency
RDS	CPUUtilization, DatabaseConnections, ReadLatency, WriteLatency, FreeStorageSpace	1 minute
ELB/ALB	RequestCount, TargetResponseTime, HTTPCode_Target_2XX_Count, UnHealthyHostCount	1 minute
DynamoDB	ConsumedReadCapacityUnits, ConsumedWriteCapacityUnits, UserErrors, SystemErrors	1 minute
S3	BucketSizeBytes, NumberOfObjects, AllRequests, 4xxErrors, 5xxErrors	Daily

Custom Metrics

Use Cases:

- Application-specific metrics (cart abandonment rate, login failures)
- Business metrics (revenue, user signups)
- System metrics not provided by AWS (application memory usage, queue depth)

Publishing Custom Metrics

Method 1: AWS CLI

```
# Publish a single metric
aws cloudwatch put-metric-data \
    --namespace "MyApp/Production" \
    --metric-name "PageLoadTime" \
    --value 245 \
    --unit Milliseconds \
    --dimensions Page=Homepage,Region=us-east-1

# Publish metric with timestamp
aws cloudwatch put-metric-data \
    --namespace "MyApp/Production" \
    --metric-name "ActiveUsers" \
    --value 1523 \
    --timestamp 2025-10-28T10:00:00Z \
    --dimensions Environment=Production
```

Method 2: Python Boto3

```
import boto3
from datetime import datetime

cloudwatch = boto3.client('cloudwatch', region_name='us-east-1')

# Publish metric
cloudwatch.put_metric_data(
    Namespace='MyApp/Production',
    MetricData=[
```

```
        {
            'MetricName': 'PageLoadTime',
            'Value': 245,
            'Unit': 'Milliseconds',
            'Timestamp': datetime.utcnow(),
            'Dimensions': [
                {'Name': 'Page', 'Value': 'Homepage'},
                {'Name': 'Region', 'Value': 'us-east-1'}
            ]
        }
    ]
)

# Publish multiple metrics at once
cloudwatch.put_metric_data(
    Namespace='MyApp/Production',
    MetricData=[
        {
            'MetricName': 'RequestCount',
            'Value': 150,
            'Unit': 'Count',
            'Timestamp': datetime.utcnow()
        },
        {
            'MetricName': 'ErrorCount',
            'Value': 3,
            'Unit': 'Count',
            'Timestamp': datetime.utcnow()
        },
        {
            'MetricName': 'AverageResponseTime',
            'Value': 125.5,
            'Unit': 'Milliseconds',
            'Timestamp': datetime.utcnow()
        }
    ]
)
```

Method 3: CloudWatch Agent

Configure in `/opt/aws/amazon-cloudwatch-agent/etc/config.json`:

```
{
  "metrics": {
    "namespace": "MyApp/Production",
    "metrics_collected": {
      "mem": {
        "measurement": [
          {
            "name": "mem_used_percent",
            "rename": "MemoryUtilization",
            "unit": "Percent"
          }
        ],
        "metrics_collection_interval": 60
```

```
      },
      "disk": {
        "measurement": [
          {
            "name": "used_percent",
            "rename": "DiskUtilization",
            "unit": "Percent"
          }
        ],
        "metrics_collection_interval": 60,
        "resources": ["/"]
      }
    }
  }
}
```

Metric Math

Purpose: Perform calculations on multiple metrics to create derived metrics [5].

Common Use Cases:

- Error rate percentage: `(Errors / Invocations) * 100`
- Success rate: `((RequestCount - ErrorCount) / RequestCount) * 100`
- Average across multiple instances: `AVG([m1, m2, m3])`

Example: Calculate Error Rate

```
# Using AWS CLI with metric math
aws cloudwatch get-metric-statistics \
    --namespace AWS/Lambda \
    --metric-name Invocations \
    --dimensions Name=FunctionName,Value=MyFunction \
    --start-time 2025-10-28T00:00:00Z \
    --end-time 2025-10-28T23:59:59Z \
    --period 3600 \
    --statistics Sum

# Create alarm with metric math (in console or via API)
# Expression: (m1 / m2) * 100
# Where m1 = Errors, m2 = Invocations
```

CloudWatch Alarms

Alarms watch metrics and trigger actions when thresholds are breached [5].

Alarm States

State	Description
OK	Metric is within defined threshold
ALARM	Metric breached threshold
INSUFFICIENT_DATA	Not enough data to determine state (alarm just created, or metric has no data)

Alarm Components

1. **Metric**: The metric to monitor
2. **Threshold**: The value to compare against
3. **Comparison Operator**: How to compare (>, <, >=, <=)
4. **Evaluation Period**: How many periods to evaluate
5. **Datapoints to Alarm**: How many periods must breach to trigger alarm
6. **Actions**: What to do when alarm state changes

Creating Alarms

Example 1: High CPU Alert

Console Steps:

1. Navigate to **CloudWatch → Alarms → Create alarm**
2. Select **EC2** namespace
3. Choose **CPUUtilization** metric
4. Select specific instance or all instances
5. Set conditions:
   - **Threshold type**: Static
   - **Comparison**: Greater than
   - **Threshold value**: 80
   - **Period**: 5 minutes
   - **Datapoints**: 3 out of 3
6. Configure actions (SNS notification)
7. Name alarm and create

AWS CLI:

```
aws cloudwatch put-metric-alarm \
    --alarm-name "HighCPU-i-1234567890" \
    --alarm-description "Alert when CPU exceeds 80%" \
    --metric-name CPUUtilization \
```

```
    --namespace AWS/EC2 \
    --statistic Average \
    --period 300 \
    --threshold 80 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 3 \
    --datapoints-to-alarm 3 \
    --dimensions Name=InstanceId,Value=i-1234567890abcdef0 \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:MyTopic \
    --treat-missing-data notBreaching
```

Example 2: Lambda Error Rate Alarm

```
aws cloudwatch put-metric-alarm \
    --alarm-name "HighLambdaErrors-MyFunction" \
    --alarm-description "Alert when error rate exceeds 5%" \
    --metrics '[
        {
            "Id": "e1",
            "Expression": "(m1/m2)*100",
            "Label": "ErrorRate",
            "ReturnData": true
        },
        {
            "Id": "m1",
            "MetricStat": {
                "Metric": {
                    "Namespace": "AWS/Lambda",
                    "MetricName": "Errors",
                    "Dimensions": [
                        {"Name": "FunctionName", "Value": "MyFunction"}
                    ]
                },
                "Period": 300,
                "Stat": "Sum"
            },
            "ReturnData": false
        },
        {
            "Id": "m2",
            "MetricStat": {
                "Metric": {
                    "Namespace": "AWS/Lambda",
                    "MetricName": "Invocations",
                    "Dimensions": [
                        {"Name": "FunctionName", "Value": "MyFunction"}
                    ]
                },
                "Period": 300,
                "Stat": "Sum"
            },
            "ReturnData": false
        }
    ]' \
    --threshold 5 \
```

```
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 2 \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:DevOpsAlerts
```

Alarm Actions

Supported Actions:

1. **SNS Notifications**: Send email, SMS, or trigger Lambda
2. **Auto Scaling**: Scale EC2 instances up or down
3. **EC2 Actions**: Stop, terminate, reboot, or recover instances
4. **Systems Manager**: Run automation documents

Example: Auto Scaling Action

```
aws cloudwatch put-metric-alarm \
    --alarm-name "ScaleUp-HighCPU" \
    --metric-name CPUUtilization \
    --namespace AWS/EC2 \
    --statistic Average \
    --period 300 \
    --threshold 70 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 2 \
    --dimensions Name=AutoScalingGroupName,Value=my-asg \
    --alarm-actions arn:aws:autoscaling:us-east-1:123456789012:scalingPolicy:policy-id
```

Composite Alarms

Purpose: Combine multiple alarms using AND/OR logic to reduce false positives [5].

Example: Alert only when both CPU is high AND memory is high

```
aws cloudwatch put-composite-alarm \
    --alarm-name "CriticalResourceUsage" \
    --alarm-description "CPU AND Memory are both high" \
    --actions-enabled \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:CriticalAlerts \
    --alarm-rule "ALARM(HighCPU) AND ALARM(HighMemory)"
```

CloudWatch Dashboards

Dashboards provide visual representation of your metrics and logs [2].

Creating Dashboards

Console Method

1. Navigate to **CloudWatch → Dashboards → Create dashboard**
2. Enter dashboard name
3. Add widgets:
   - **Line graph**: Trend over time
   - **Number**: Single current value
   - **Gauge**: Value with min/max
   - **Bar chart**: Compare multiple metrics
   - **Pie chart**: Proportional distribution
   - **Logs table**: Display log query results

Programmatic Creation

```
import boto3
import json

cloudwatch = boto3.client('cloudwatch')

dashboard_body = {
    "widgets": [
        {
            "type": "metric",
            "properties": {
                "metrics": [
                    ["AWS/EC2", "CPUUtilization", {"stat": "Average"}]
                ],
                "period": 300,
                "stat": "Average",
                "region": "us-east-1",
                "title": "EC2 CPU Utilization",
                "yAxis": {"left": {"min": 0, "max": 100}}
            }
        }
    ]
}

cloudwatch.put_dashboard(
    DashboardName='MyProductionDashboard',
    DashboardBody=json.dumps(dashboard_body)
)
```

Sharing Dashboards

Methods:

1. **Email**: Generate shareable link
2. **Public URL**: Share publicly (be cautious with sensitive data)
3. **SSO**: Share with specific users in your organization
4. **Cross-account**: Share across AWS accounts

Practical Hands-On Examples

Example 1: Monitor EC2 Instance

Scenario: Monitor CPU, memory, and disk for an EC2 instance.

Step 1: Install CloudWatch Agent

```
# On EC2 instance
wget https://s3.amazonaws.com/amazoncloudwatch-agent/amazon_linux/amd64/latest/amazon-clo
sudo rpm -U ./amazon-cloudwatch-agent.rpm
```

Step 2: Configure Agent

Create `/opt/aws/amazon-cloudwatch-agent/etc/config.json`:

```
{
  "metrics": {
    "namespace": "CustomMetrics/EC2",
    "metrics_collected": {
      "cpu": {
        "measurement": [
          {"name": "cpu_usage_active", "rename": "CPUUtilization", "unit": "Percent"}
        ],
        "metrics_collection_interval": 60,
        "totalcpu": false
      },
      "mem": {
        "measurement": [
          {"name": "mem_used_percent", "rename": "MemoryUtilization", "unit": "Percent"}
        ],
        "metrics_collection_interval": 60
      },
      "disk": {
        "measurement": [
          {"name": "used_percent", "rename": "DiskUtilization", "unit": "Percent"}
        ],
        "metrics_collection_interval": 60,
        "resources": ["/"]
      }
    }
```

```
  },
  "logs": {
    "logs_collected": {
      "files": {
        "collect_list": [
          {
            "file_path": "/var/log/messages",
            "log_group_name": "/aws/ec2/system-logs",
            "log_stream_name": "{instance_id}"
          }
        ]
      }
    }
  }
}
```

Step 3: Start Agent

```
sudo /opt/aws/amazon-cloudwatch-agent/bin/amazon-cloudwatch-agent-ctl \
    -a fetch-config \
    -m ec2 \
    -s \
    -c file:/opt/aws/amazon-cloudwatch-agent/etc/config.json
```

Step 4: Create Alarms

```
# High CPU alarm
aws cloudwatch put-metric-alarm \
    --alarm-name "EC2-HighCPU" \
    --metric-name CPUUtilization \
    --namespace CustomMetrics/EC2 \
    --statistic Average \
    --period 300 \
    --threshold 80 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 2 \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:Alerts

# High memory alarm
aws cloudwatch put-metric-alarm \
    --alarm-name "EC2-HighMemory" \
    --metric-name MemoryUtilization \
    --namespace CustomMetrics/EC2 \
    --statistic Average \
    --period 300 \
    --threshold 90 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 2 \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:Alerts
```

Example 2: Monitor Lambda Function

Scenario: Track invocations, errors, duration, and create alerts.

Step 1: View Built-in Metrics

Lambda automatically sends metrics to CloudWatch:

- Invocations
- Errors
- Duration
- Throttles
- ConcurrentExecutions

Step 2: Add Custom Logging

```
import json
import logging

logger = logging.getLogger()
logger.setLevel(logging.INFO)

def lambda_handler(event, context):
    logger.info(f"Function invoked with event: {json.dumps(event)}")

    try:
        # Your business logic
        result = process_request(event)
        logger.info(f"Request processed successfully: {result}")
        return {
            'statusCode': 200,
            'body': json.dumps(result)
        }
    except Exception as e:
        logger.error(f"Error processing request: {str(e)}")
        raise
```

Step 3: Create CloudWatch Alarm for Errors

```
aws cloudwatch put-metric-alarm \
    --alarm-name "Lambda-HighErrors-MyFunction" \
    --metric-name Errors \
    --namespace AWS/Lambda \
    --statistic Sum \
    --period 300 \
    --threshold 5 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 1 \
    --dimensions Name=FunctionName,Value=MyFunction \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:DevOps
```

Step 4: Query Logs with Insights

```
# Find all errors in last hour
fields @timestamp, @message
| filter @message like /ERROR/
| sort @timestamp desc

# Calculate error rate
stats count() as total,
      count(@message like /ERROR/) as errors,
      (errors / total) * 100 as error_rate
```

Example 3: Create Application Dashboard

Scenario: Build a dashboard showing API health metrics.

Python Script:

```
import boto3
import json

cloudwatch = boto3.client('cloudwatch', region_name='us-east-1')

dashboard_config = {
    "widgets": [
        {
            "type": "metric",
            "width": 12,
            "height": 6,
            "properties": {
                "metrics": [
                    ["AWS/ApiGateway", "Count", {"stat": "Sum", "label": "Total Requests"
                    [".", "4XXError", {"stat": "Sum", "label": "Client Errors"}],
                    [".", "5XXError", {"stat": "Sum", "label": "Server Errors"}]
                ],
                "view": "timeSeries",
                "region": "us-east-1",
                "title": "API Request Metrics",
                "period": 300
            }
        },
        {
            "type": "metric",
            "width": 12,
            "height": 6,
            "properties": {
                "metrics": [
                    ["AWS/ApiGateway", "Latency", {"stat": "Average", "label": "Avg Laten
                    ["...", {"stat": "p99", "label": "P99 Latency"}]
                ],
                "view": "timeSeries",
                "region": "us-east-1",
                "title": "API Latency",
                "period": 300,
```

```
                "yAxis": {"left": {"label": "ms"}}
            }
        },
        {
            "type": "log",
            "width": 24,
            "height": 6,
            "properties": {
                "query": "fields @timestamp, @message | filter @message like /ERROR/ | so
                "region": "us-east-1",
                "title": "Recent Errors",
                "logGroupNames": ["/aws/lambda/MyAPIFunction"]
            }
        }
    ]
}

response = cloudwatch.put_dashboard(
    DashboardName='API-Production-Dashboard',
    DashboardBody=json.dumps(dashboard_config)
)

print(f"Dashboard created: {response}")
```

Best Practices

1. Metric Best Practices

✓ **Use appropriate resolution**: Standard for cost savings, high-resolution for critical real-time monitoring
✓ **Choose right statistics**: Average for CPU, Sum for counts, p99 for latency
✓ **Add dimensions**: Enable filtering and aggregation
✓ **Use metric math**: Calculate derived metrics (error rates, success rates)
✓ **Set retention appropriately**: Balance cost vs. historical analysis needs

2. Alarm Best Practices

✓ **Set meaningful thresholds**: Based on historical data and SLOs
✓ **Use multiple datapoints**: Avoid false positives from temporary spikes
✓ **Configure alarm actions**: SNS notifications, auto-scaling, incident response
✓ **Use composite alarms**: Reduce alert fatigue
✓ **Treat missing data appropriately**: `notBreaching`, `breaching`, `ignore`, or `missing`
✓ **Test alarms**: Verify they trigger correctly

3. Log Best Practices

✅ **Use structured logging**: JSON format for easy parsing
✅ **Set retention policies**: Avoid indefinite storage costs
✅ **Create metric filters**: Extract metrics from logs
✅ **Use Log Insights**: Query logs instead of downloading
✅ **Tag log groups**: Organize and control costs
✅ **Encrypt sensitive logs**: Use KMS encryption

4. Dashboard Best Practices

✅ **Create role-specific dashboards**: Dev, Ops, Business
✅ **Use consistent naming**: Standardize across organization
✅ **Include key metrics only**: Avoid cluttered dashboards
✅ **Add annotations**: Explain anomalies and changes
✅ **Share dashboards**: Improve visibility and collaboration

Common Use Cases

1. Auto Scaling Based on Metrics

Scenario: Scale EC2 instances based on CPU utilization.

```
# Create scale-up policy
aws autoscaling put-scaling-policy \
    --auto-scaling-group-name my-asg \
    --policy-name scale-up \
    --scaling-adjustment 1 \
    --adjustment-type ChangeInCapacity

# Create alarm to trigger scale-up
aws cloudwatch put-metric-alarm \
    --alarm-name cpu-high \
    --metric-name CPUUtilization \
    --namespace AWS/EC2 \
    --statistic Average \
    --period 300 \
    --threshold 70 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 2 \
    --dimensions Name=AutoScalingGroupName,Value=my-asg \
    --alarm-actions arn:aws:autoscaling:us-east-1:123456789012:scalingPolicy:policy-id
```

2. Billing Alerts

Scenario: Get notified when AWS costs exceed threshold.

```
aws cloudwatch put-metric-alarm \
    --alarm-name "HighBilling" \
```

```
    --metric-name EstimatedCharges \
    --namespace AWS/Billing \
    --statistic Maximum \
    --period 21600 \
    --threshold 1000 \
    --comparison-operator GreaterThanThreshold \
    --evaluation-periods 1 \
    --dimensions Name=Currency,Value=USD \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:BillingAlerts
```

3. Database Monitoring

Scenario: Monitor RDS database performance.

Key Metrics:

- CPUUtilization
- DatabaseConnections
- FreeStorageSpace
- ReadLatency / WriteLatency
- DiskQueueDepth

```
# Low disk space alarm
aws cloudwatch put-metric-alarm \
    --alarm-name "RDS-LowDisk" \
    --metric-name FreeStorageSpace \
    --namespace AWS/RDS \
    --statistic Average \
    --period 300 \
    --threshold 10737418240 \
    --comparison-operator LessThanThreshold \
    --evaluation-periods 1 \
    --dimensions Name=DBInstanceIdentifier,Value=mydb \
    --alarm-actions arn:aws:sns:us-east-1:123456789012:DBAAlerts
```

Cost Optimization

CloudWatch Pricing Components

Component	Pricing
Standard Metrics	Free (AWS service metrics)
Custom Metrics	$0.30 per metric per month (first 10,000)
API Requests	$0.01 per 1,000 requests
Logs Ingestion	$0.50 per GB
Logs Storage	$0.03 per GB per month

Component	Pricing
Dashboards	$3 per dashboard per month (first 3 free)
Alarms	$0.10 per alarm per month (first 10 free)

Cost Optimization Strategies

✓ **Set log retention**: Don't store logs indefinitely
✓ **Archive old logs to S3**: $0.023/GB vs. $0.03/GB in CloudWatch
✓ **Use metric filters wisely**: Extract only necessary metrics from logs
✓ **Aggregate metrics**: Reduce custom metric cardinality
✓ **Delete unused alarms and dashboards**
✓ **Use standard resolution**: Avoid high-resolution unless necessary

Example: Export Logs to S3

```
aws logs create-export-task \
    --log-group-name /aws/lambda/MyFunction \
    --from 1698480000000 \
    --to 1698566399000 \
    --destination s3-bucket-name \
    --destination-prefix lambda-logs/
```

Summary

Amazon CloudWatch is a comprehensive monitoring service that provides:

✓ **Metrics**: Track performance of AWS services and custom applications
✓ **Logs**: Centralize log collection, storage, and analysis
✓ **Alarms**: Automated alerts and actions based on thresholds
✓ **Dashboards**: Visual representation of system health
✓ **Insights**: Powerful query capabilities for troubleshooting

For Junior Operations Engineers

Must-Know Skills:

1. View and interpret CloudWatch metrics
2. Create and manage log groups
3. Query logs using CloudWatch Logs Insights
4. Set up alarms with appropriate thresholds
5. Build basic dashboards for monitoring
6. Install and configure CloudWatch agent

Interview Preparation:

- Understand the hierarchy: Namespace → Metric → Dimension

- Know the difference between log groups, streams, and events
- Be able to explain alarm states and evaluation periods
- Practice creating alarms using CLI
- Understand cost implications of retention and custom metrics

References

[1] AWS CloudWatch Documentation - What is CloudWatch
[2] AWS CloudWatch User Guide - Core Concepts
[3] AWS CloudWatch Logs - Working with Log Groups and Streams
[4] AWS CloudWatch Logs - Best Practices
[5] AWS CloudWatch Alarms - Configuration Guide