**Implementation Guide: Grafana Base Approach with CloudWatch Integration**

This guide provides a detailed, granular step-by-step process to implement the "Grafana Base Approach" for monitoring AWS resources (e.g., Lambda, ECS, EC2, DynamoDB, Aurora) using Grafana's built-in CloudWatch data source. This is fully compatible with the free, open-source (OSS) version of Grafana, which you mentioned is already installed and running on your self-hosted server. I've skipped the installation step accordingly and noted it for clarity.

The approach leverages AWS CloudWatch Metrics APIs to pull data directly into Grafana. All steps assume you have administrative access to your Grafana instance and AWS account. If you're new to Grafana, follow each sub-step precisely. I've included references to official documentation, APIs, and tools for verification.

**Prerequisites**:

* Grafana OSS (version 8.x or later recommended for best CloudWatch support; check your version via the Grafana UI under "Help" > "About").
* AWS account with resources emitting metrics to CloudWatch (e.g., EC2 instances with basic monitoring enabled).
* Basic networking: Ensure your Grafana server can reach AWS endpoints (e.g., monitoring.<region>.amazonaws.com) over the internet or via VPC if private.
* AWS credentials: You'll need either IAM access keys or an IAM role attached to the Grafana server's EC2 instance (preferred for security).

**Estimated Effort**: 2-4 hours for basic setup, plus additional time for custom dashboards. This aligns with your outline but is adjusted for your pre-installed Grafana.

**Step 1: Access the Grafana User Interface**

1. Open a web browser and navigate to your Grafana instance's URL (e.g., http://<your-server-ip>:3000 or <https://grafana.yourdomain.com> if configured with SSL).
2. Log in with admin credentials (default is admin/admin if unchanged; change this immediately for security via the user profile settings).
   * Reference: Grafana official docs on logging in - [Grafana Login Guide](https://grafana.com/docs/grafana/latest/administration/user-management/).

**Step 2: Add and Configure the CloudWatch Data Source**

This step sets up direct integration with AWS CloudWatch using Grafana's built-in plugin, which queries the CloudWatch Metrics API.

1. In the Grafana sidebar (left menu), click on the gear icon ("Configuration") > "Data sources".
2. Click "Add data source".
3. Search for and select "CloudWatch" (it's a core plugin in OSS Grafana; no installation needed).
4. In the configuration page:
   * **Name**: Enter a descriptive name, e.g., "AWS-CloudWatch-Prod".
   * **Default Region**: Select your primary AWS region (e.g., us-east-1) from the dropdown. This is where most queries will default to.
   * **Authentication**: Choose one of the following based on your setup (IAM role is recommended for security):
     + **Option A: AWS Access Keys** (use if Grafana is not on an EC2 instance):
       - Enter your AWS Access Key ID and Secret Access Key.
       - Generate these via AWS IAM console: Create a user with "AmazonCloudWatchReadOnlyAccess" policy (least privilege).
       - Reference: AWS docs on creating access keys - [AWS IAM Access Keys](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_credentials_access-keys.html).
     + **Option B: IAM Role (Assume Role)** (use if Grafana runs on an EC2 instance in your AWS account):
       - Leave access keys blank.
       - Enter the IAM Role ARN (e.g., arn:aws:iam::<account-id>:role/GrafanaCloudWatchRole).
       - Attach the "CloudWatchReadOnlyAccess" policy to this role.
       - Reference: AWS docs on IAM roles for EC2 - [AWS IAM Roles for EC2](https://docs.aws.amazon.com/IAM/latest/UserGuide/id_roles_use_switch-role-ec2.html). Grafana will use the EC2 metadata service to assume the role.
   * **Namespaces**: Optionally, pre-select AWS services (e.g., AWS/EC2, AWS/DynamoDB) to limit queries.
   * **Other Settings**: Enable "Statistics" for aggregated metrics if needed (default is fine for starters).
5. Click "Save & Test" at the bottom. Grafana will attempt a test query to CloudWatch's ListMetrics API.
   * If it fails, check: Network connectivity, credentials, or region. Error messages will guide you (e.g., "InvalidSignatureException" means bad keys).
   * Underlying API: This uses AWS CloudWatch's GetMetricData or ListMetrics APIs - [AWS CloudWatch Metrics API Reference](https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/API_GetMetricData.html).
6. Reference: Full Grafana CloudWatch data source docs - [Grafana CloudWatch Plugin Guide](https://grafana.com/docs/grafana/latest/datasources/aws-cloudwatch/).

**Step 3: Query Metrics and Create a Basic Panel**

Now that the data source is connected, test by querying real AWS metrics.

1. In the Grafana sidebar, click "Dashboards" > "New" > "New Dashboard".
2. Click "Add a new panel".
3. In the panel editor:
   * Under "Query", select your CloudWatch data source from the dropdown.
   * **Namespace**: Choose an AWS service, e.g., "AWS/EC2" for EC2 metrics.
   * **Metric Name**: Select from dropdown, e.g., "CPUUtilization" for EC2 CPU usage.
   * **Dimensions**: Filter by resource, e.g., InstanceId = i-1234567890abcdef0 (find your instance IDs in AWS EC2 console).
   * **Statistic**: Choose "Average" or "Maximum" (e.g., average CPU over 5 minutes).
   * **Period**: Set to auto or a value like 5m (5 minutes).
4. Click "Run queries" to preview data. You should see a graph or table of metrics.
5. Customize visualization: Switch to "Graph" mode, add titles, units (e.g., "%" for CPU).
6. Save the panel and dashboard.
   * Example Query for DynamoDB: Namespace = AWS/DynamoDB, Metric = ConsumedReadCapacityUnits, Dimensions = TableName = YourTable.
   * Reference: AWS CloudWatch metrics list by service - [AWS CloudWatch Metrics per Service](https://docs.aws.amazon.com/AmazonCloudWatch/latest/monitoring/aws-services-cloudwatch-metrics.html). Grafana query editor docs - [Grafana Query Language for CloudWatch](https://grafana.com/docs/grafana/latest/datasources/aws-cloudwatch/cloudwatch-queries/).

**Step 4: Build or Import Dashboards**

1. **Build Custom Dashboard**:
   * From the new dashboard, add more panels as in Step 3 (e.g., one for EC2 CPU, another for Lambda invocations via Namespace = AWS/Lambda, Metric = Invocations).
   * Organize into rows/sections for services like EC2, DynamoDB.
2. **Import Community Dashboards**:
   * Go to Grafana sidebar > "Dashboards" > "Import".
   * Search Grafana's community dashboard gallery (via browser: <https://grafana.com/grafana/dashboards/>) for "AWS CloudWatch" (e.g., Dashboard ID 112 for EC2, or 10229 for DynamoDB).
   * Copy the JSON or ID, paste into Grafana's import field, and select your CloudWatch data source.
   * Adjust variables (e.g., region, instance IDs) in the dashboard settings.
   * Reference: Grafana dashboard import guide - [Grafana Import Dashboards](https://grafana.com/docs/grafana/latest/dashboards/export-import/).

**Step 5: Set Up Alerts (Optional)**

1. In a dashboard panel, click the panel title > "Edit".
2. Go to the "Alert" tab in the panel editor.
3. Click "Create Alert Rule".
4. Define conditions: E.g., WHEN avg() OF query(A, 5m, now) IS ABOVE 80 (for CPU >80%).
5. Set evaluation interval (e.g., every 1m) and notifications (requires configuring a notification channel first).
6. Configure notifications: Sidebar > "Alerting" > "Notification channels" > Add channel (e.g., email via SMTP; configure server details).
7. Save and test the alert.
   * Note: Alerts in OSS Grafana are basic; for advanced, consider Grafana Enterprise (paid).
   * Reference: Grafana alerting docs - [Grafana Alerting Guide](https://grafana.com/docs/grafana/latest/alerting/old-alerting/create-alerts/). AWS CloudWatch alarms can complement this but aren't directly integrated here.

**Troubleshooting and Best Practices**

* **Common Issues**: If queries fail, verify AWS credentials have CloudWatch permissions. Use AWS CLI to test: aws cloudwatch list-metrics --namespace AWS/EC2 (install AWS CLI if needed - [AWS CLI Docs](https://docs.aws.amazon.com/cli/latest/userguide/cli-chap-install.html)).
* **Security**: Rotate access keys regularly; prefer IAM roles. Restrict Grafana access with users/roles.
* **Scaling**: For multiple AWS accounts, add separate data sources or use cross-account IAM roles - [AWS Cross-Account Access](https://docs.aws.amazon.com/IAM/latest/UserGuide/tutorial_cross-account-with-roles.html).
* **Testing**: Start with free-tier AWS resources to avoid costs.
* **Next Steps**: If you need logs/traces, explore add-ons like Prometheus for white-box metrics (separate setup).

**Implementation Guide: Grafana Enhanced Approach with Custom Plugins**

This guide provides a detailed, granular step-by-step process to implement the "Grafana Enhanced Approach" by developing and installing custom Grafana plugins for direct integration with AWS APIs. This extends beyond the built-in CloudWatch data source, allowing custom queries (e.g., to AWS DynamoDB's DescribeTable API or Lambda invocation details). It's fully compatible with the free, open-source (OSS) version of Grafana, which you mentioned is already installed and running on your self-hosted server.

The approach uses Grafana's Plugin SDK to create custom data source plugins that call AWS SDK methods directly. This enables tailored data fetching and visualizations for services like DynamoDB, Lambda, ECS, EC2, and Aurora. All steps assume you have developer access to a machine for plugin development (e.g., your local workstation or the Grafana server) and AWS credentials. If you're new to Grafana or plugin development, follow each sub-step precisely, as this involves coding in TypeScript/JavaScript and potentially Go.

**Prerequisites**:

* Grafana OSS (version 8.x or later; plugins work best on recent versions—check via Grafana UI under "Help" > "About").
* Development environment: Node.js (v16+), Yarn (v1.x), Git, and a code editor (e.g., VS Code). Install if needed—[Node.js Download](https://nodejs.org/), then run npm install -g yarn.
* AWS SDK knowledge: You'll use the AWS SDK for JavaScript (installed via Yarn in the plugin).
* AWS account with IAM permissions for targeted APIs (e.g., "dynamodb:DescribeTable" for DynamoDB). Use least-privilege policies.
* Basic networking: Ensure your development machine and Grafana server can reach AWS endpoints (e.g., dynamodb.<region>.amazonaws.com).
* Security note: Plugins handle credentials—use secure methods like environment variables or IAM roles to avoid hardcoding.

**Estimated Effort**: 10-15 steps per plugin, 20-40 hours total (including coding, testing, and iteration for multiple services). This matches your outline but accounts for debugging time.

**Step 1: Set Up Development Environment and Scaffold the Plugin**

This creates a boilerplate for a custom data source plugin using Grafana's official tool.

1. On your development machine, open a terminal.
2. Install Grafana's create-plugin tool globally: Run yarn global add @grafana/create-plugin or npm install -g @grafana/create-plugin.
   * Reference: Grafana plugin scaffolding docs - [Grafana Create Plugin Tool](https://grafana.com/developers/plugin-tools/create-a-plugin/set-up-a-plugin-project).
3. Create a new plugin directory: Run @grafana/create-plugin.
   * Follow prompts: Select "datasource" as plugin type (for querying AWS APIs).
   * Choose "backend" if you need server-side logic (e.g., for secure credential handling in Go); otherwise, "frontend-only" for simpler JS-based queries.
   * Enter plugin details: ID (e.g., "aws-custom-datasource"), name (e.g., "AWS Custom API Data Source"), org (your name).
   * This generates a folder with TypeScript code, a plugin.json manifest, and build scripts.
4. Navigate into the plugin folder: cd <plugin-folder-name>.
5. Install dependencies: Run yarn install (or npm install).
6. If adding backend: Ensure Go is installed (v1.17+—[Go Download](https://go.dev/dl/)), then run mage -v build:backend (Mage is included for Go builds).
   * Reference: Grafana backend plugin docs - [Grafana Backend Plugins](https://grafana.com/developers/plugin-tools/build-a-plugin/develop-a-plugin/backend/introduction-to-backend-plugins).

**Step 2: Implement Data Source Logic Using AWS SDK**

Customize the plugin to call AWS APIs directly. Focus on one service first (e.g., DynamoDB), then iterate for others like Lambda.

1. Install AWS SDK: In the plugin folder, run yarn add @aws-sdk/client-dynamodb (for DynamoDB; add others like @aws-sdk/client-lambda for Lambda).
   * Reference: AWS SDK for JavaScript v3 docs - [AWS SDK JS Getting Started](https://docs.aws.amazon.com/sdk-for-javascript/v3/developer-guide/welcome.html).
2. Edit the data source code:
   * Open src/DataSource.ts (or similar) in your editor.
   * Import AWS SDK: Add import { DynamoDBClient, DescribeTableCommand } from '@aws-sdk/client-dynamodb';.
   * Configure credentials: In the constructor or query method, set up AWS clients using provided options (e.g., access keys from Grafana config). For security, prefer passing credentials via Grafana's secure JSON data.
     + Example code snippet:

typescript

import { DataSourceInstanceSettings, DataQueryRequest, DataQueryResponse } from '@grafana/data';

import { DynamoDBClient, DescribeTableCommand } from '@aws-sdk/client-dynamodb';

export class DataSource {

constructor(instanceSettings: DataSourceInstanceSettings) {

this.awsClient = new DynamoDBClient({

region: instanceSettings.jsonData.region || 'us-east-1',

credentials: {

accessKeyId: instanceSettings.jsonData.accessKey || process.env.AWS\_ACCESS\_KEY\_ID,

secretAccessKey: instanceSettings.jsonData.secretKey || process.env.AWS\_SECRET\_ACCESS\_KEY,

},

});

}

async query(options: DataQueryRequest): Promise<DataQueryResponse> {

// Custom query logic here

const command = new DescribeTableCommand({ TableName: 'YourTableName' });

const response = await this.awsClient.send(command);

// Transform response to Grafana data frame

return { data: [/\* Convert to Field/Frame \*/] };

}

}

* + - Adapt for other APIs: E.g., for Lambda logs, use InvokeCommand from @aws-sdk/client-lambda to fetch invocation details.
    - Handle errors: Add try-catch for AWS API errors (e.g., AccessDeniedException).

1. For backend plugins (Go): Edit pkg/plugin/datasource.go to call AWS APIs server-side, which is safer for credentials.
   * Install AWS SDK for Go: Run go get github.com/aws/aws-sdk-go-v2.
   * Reference: AWS SDK for Go docs - [AWS SDK Go v2](https://docs.aws.amazon.com/sdk-for-go/v2/api/).
2. Define query editor UI: Edit src/QueryEditor.tsx to add form fields (e.g., dropdown for AWS service, input for table name).
   * Use Grafana UI components: Import from @grafana/ui.
   * Reference: Grafana frontend plugin docs - [Grafana Frontend Plugins](https://grafana.com/developers/plugin-tools/build-a-plugin/develop-a-plugin/frontend/introduction-to-frontend-plugins).
3. Transform data: In query(), convert AWS responses to Grafana's DataFrame format for visualization (e.g., table or graph).
   * Reference: Grafana data API - [Grafana Data Frames](https://grafana.com/docs/grafana/latest/developers/plugins/data-frames/).

**Step 3: Test the Plugin Locally Against AWS Services**

1. Start dev server: Run yarn dev (watches for changes and builds).
2. If backend: Run mage -v build:backend then yarn dev.
3. Test queries: Use Grafana's plugin dev mode or a local Grafana instance.
   * Temporarily add to your running Grafana: Copy the dist folder to Grafana's plugins dir (e.g., /var/lib/grafana/plugins/), restart Grafana.
   * In Grafana UI: Add the data source (Configuration > Data sources > Add > Your plugin name).
   * Run test queries: E.g., query DynamoDB DescribeTable; verify via AWS console that the API call succeeds.
4. Debug: Use console logs or Grafana's dev tools (F12 in browser). Check AWS CloudTrail for API calls.
   * Underlying APIs: E.g., DynamoDB DescribeTable - [AWS DynamoDB API Reference](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/API_DescribeTable.html); Lambda Invoke - [AWS Lambda API Reference](https://docs.aws.amazon.com/lambda/latest/dg/API_Invoke.html).
5. Iterate: Add support for multiple services by extending the query logic (e.g., switch based on user input).

**Step 4: Build and Sign the Plugin**

1. Build for production: Run yarn build (bundles JS; for backend, mage -v buildAll).
2. Sign the plugin (required for Grafana 8+ to load unsigned plugins securely): Use Grafana's signing tool.
   * Generate a key if needed: Follow docs to create a manifest.
   * Run npx @grafana/sign-plugin@latest.
   * Reference: Grafana plugin signing docs - [Grafana Plugin Signing](https://grafana.com/developers/plugin-tools/publish-a-plugin/sign-a-plugin).
3. Package: The dist/ folder now contains the plugin.

**Step 5: Install the Plugin in Grafana**

1. Copy the dist/ folder to your Grafana server's plugins directory (default: /var/lib/grafana/plugins/ on Linux; configure in grafana.ini if custom).
2. Restart Grafana: Use systemd (e.g., sudo systemctl restart grafana-server) or your server's method.
3. Verify: In Grafana UI, go to Configuration > Data sources > Add > Search for your plugin name. Configure with AWS region/credentials (securely via JSON data).
   * Reference: Grafana plugin installation docs - [Grafana Install Plugins](https://grafana.com/docs/grafana/latest/administration/plugins-management/).

**Step 6: Create Dashboards with Custom Data**

1. In Grafana sidebar, create a new dashboard.
2. Add panels using your custom data source: Query e.g., DynamoDB table details or Lambda logs.
3. Customize visualizations: Use Grafana's built-in types or create custom visualization plugins if needed (similar scaffolding process).
4. Import/export: Share JSON for reusability.
   * Reference: Same as base approach - [Grafana Dashboards](https://grafana.com/docs/grafana/latest/dashboards/).

**Troubleshooting and Best Practices**

* **Common Issues**: Build errors? Check Node/Yarn versions. API failures? Verify IAM permissions with AWS CLI: aws dynamodb describe-table --table-name YourTable. No internet? Ensure dev machine has access.
* **Security**: Never hardcode credentials—use Grafana's encrypted fields or IAM roles. Audit plugin code for vulnerabilities.
* **Maintenance**: Monitor AWS API changelogs - [AWS What's New](https://aws.amazon.com/new/); update SDK versions periodically.
* **Scaling**: For multiple plugins (one per service), repeat steps. Test against AWS limits (e.g., API throttling) - [AWS Service Quotas](https://docs.aws.amazon.com/general/latest/gr/aws_service_limits.html).
* **Alternatives if Stuck**: If dev effort is too high, fall back to base approach or community plugins (search Grafana plugins catalog).
* **Testing**: Use AWS free-tier resources. For code verification, consider local mocks with AWS SDK stubs.

**Implementation Guide: Grafana Complete Approach with OTEL Collectors and Prometheus**

This guide provides a detailed, granular step-by-step process to implement the "Grafana Complete Approach" using OpenTelemetry (OTEL) Collectors to collect metrics, traces, and logs from AWS services (e.g., Lambda, ECS, EC2, DynamoDB, Aurora), push them to a self-hosted Prometheus instance, and visualize in your existing self-hosted free OSS Grafana. This enables white-box monitoring (e.g., application internals) and goes beyond CloudWatch by supporting custom instrumentation and correlations.

The setup leverages OTEL standards for data collection, Prometheus for storage, and Grafana's built-in Prometheus data source. You can use the open-source OTEL Collector or AWS Distro for OTEL (ADOT) for AWS-optimized features. All steps are compatible with Grafana OSS; no Enterprise features are required. Assume administrative access to your AWS account, hosts/instances, and Grafana server. If you're new to these tools, follow sub-steps precisely—this involves configuration files in YAML and potential AWS resource updates.

**Prerequisites**:

* Grafana OSS (version 9.x or later recommended for best OTEL/Prometheus support; check via Grafana UI under "Help" > "About").
* Server/VM for Prometheus (e.g., EC2 instance; 4GB RAM minimum for small setups).
* AWS resources configured to emit data (e.g., enable detailed monitoring on EC2).
* Development tools: Docker (for running collectors), Git, text editor. Install if needed—[Docker Download](https://docs.docker.com/get-docker/).
* AWS CLI installed and configured—[AWS CLI Docs](https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html).
* IAM permissions: For ADOT, roles with "AWSXrayWriteOnlyAccess" and "CloudWatchAgentServerPolicy"; for general access, "AmazonEC2ReadOnlyAccess" etc.
* Networking: Ensure collectors can reach AWS endpoints and your Prometheus server (use VPC peering if private).

**Estimated Effort**: 15-20 steps, 10-20 hours total (including testing and hardening; more for full service coverage like Lambda/ECS integration). This aligns with your outline but factors in iteration time.

**Step 1: Install OTEL Collectors on Relevant Hosts/Instances**

Install collectors to gather data from AWS resources. Use ADOT for AWS-specific optimizations or standard OTEL for pure OSS.

1. Choose collector type:
   * **Option A: AWS ADOT** (recommended for AWS integration): Download from GitHub releases or use AWS-managed images.
     + On your host/EC2: Run wget https://github.com/aws-observability/aws-otel-collector/releases/latest/download/aws-otel-collector.rpm (for RPM-based systems) or equivalent for DEB/MSI.
     + Install: sudo rpm -i aws-otel-collector.rpm.
     + Start: sudo /opt/aws/aws-otel-collector/bin/aws-otel-collector-ctl -a start.
     + Reference: AWS ADOT docs - [ADOT Getting Started](https://aws-otel.github.io/docs/getting-started/collector).
   * **Option B: Standard OTEL Collector**: Use Docker for simplicity.
     + Pull image: docker pull otel/opentelemetry-collector-contrib:latest.
     + Run: docker run -d -v $(pwd)/config.yaml:/etc/otelcol-contrib/config.yaml otel/opentelemetry-collector-contrib.
     + Reference: OTEL Collector docs - [OTEL Collector Installation](https://opentelemetry.io/docs/collector/installation/).
2. Deploy to relevant AWS resources:
   * For EC2: Install directly on instances.
   * For ECS: Add as a sidecar container in task definitions (detailed in Step 2).
   * For Lambda: Use as a layer/extension (detailed in Step 2).
   * For DynamoDB/Aurora: Use CloudWatch exporter in collector config to scrape metrics.
3. Verify installation: Check logs with docker logs <container-id> or /opt/aws/aws-otel-collector/logs/aws-otel-collector.log. Ensure no errors.

**Step 2: Configure Receivers and Exporters in OTEL Collectors**

Configure collectors to receive data from AWS services and export to Prometheus. Edit YAML config files.

1. Create/edit config file (e.g., config.yaml):
   * For ADOT: Located at /opt/aws/aws-otel-collector/etc/config.yaml.
   * For OTEL: As mounted in Docker.
2. Add receivers for AWS data:
   * **AWS CloudWatch Receiver** (for metrics from EC2, DynamoDB, etc.): Enable to pull from CloudWatch APIs.
     + Example YAML:

yaml

receivers:

awscloudwatch:

region: us-east-1

metrics:

metrics\_collected:

ec2: [CPUUtilization, DiskReadOps]

dynamodb: [ConsumedReadCapacityUnits]

* + - Reference: ADOT CloudWatch receiver docs - [ADOT CloudWatch Receiver](https://aws-otel.github.io/docs/components/cloudwatch-metrics).
  + **OTLP Receiver** (for traces/logs from instrumented apps):

yaml

receivers:

otlp:

protocols:

grpc:

endpoint: 0.0.0.0:4317

http:

endpoint: 0.0.0.0:4318

* + For application instrumentation: Add SDKs to your code (e.g., @opentelemetry/sdk-node for Node.js) to send to collector.
    - Reference: OTEL Instrumentation docs - [OTEL Auto-Instrumentation](https://opentelemetry.io/docs/languages/).

1. Add exporters to push to Prometheus:
   * Use Prometheus Remote Write exporter.

yaml

exporters:

prometheusremotewrite:

endpoint: http://<prometheus-server-ip>:9090/api/v1/write

auth:

authenticator: sigv4auth # For AWS auth if needed

* + - Reference: OTEL Prometheus exporter docs - [OTEL Prometheus Remote Write](https://github.com/open-telemetry/opentelemetry-collector-contrib/tree/main/exporter/prometheusremotewriteexporter).

1. Configure pipelines to connect receivers to exporters:

yaml

service:

pipelines:

metrics:

receivers: [awscloudwatch, otlp]

exporters: [prometheusremotewrite]

traces:

receivers: [otlp]

exporters: [logging] # Or to Jaeger if set up

logs:

receivers: [otlp]

exporters: [logging]

1. AWS-specific integrations:
   * **For ECS**: Update task definition in AWS console or CLI: Add collector container with shared volumes for logs/metrics.
     + Example CLI: aws ecs register-task-definition --cli-input-json file://task-def.json (include ADOT sidecar).
     + Reference: ADOT for ECS - [ADOT ECS Guide](https://aws-otel.github.io/docs/getting-started/ecs).
   * **For Lambda**: Add ADOT layer via AWS console: Search for "AWSOtelLambda" layer ARN, attach to function. Configure via environment vars (e.g., OPENTELEMETRY\_COLLECTOR\_CONFIG\_FILE).
     + Reference: ADOT for Lambda - [ADOT Lambda Guide](https://aws-otel.github.io/docs/getting-started/lambda).
2. Restart collector: sudo /opt/aws/aws-otel-collector/bin/aws-otel-collector-ctl -a restart or restart Docker container.
3. Test: Use OTEL tools to send sample data (e.g., curl -X POST http://localhost:4318/v1/metrics with JSON payload). Check collector logs for exports.

**Step 3: Set Up Prometheus to Receive from OTEL**

Install and configure Prometheus to store data from collectors.

1. Install Prometheus on a dedicated server/EC2:
   * Download: wget https://github.com/prometheus/prometheus/releases/download/v2.47.0/prometheus-2.47.0.linux-amd64.tar.gz (adjust for OS).
   * Extract and run: ./prometheus --config.file=prometheus.yml.
   * For production: Use systemd service—create /etc/systemd/system/prometheus.service.
   * Reference: Prometheus installation docs - [Prometheus Getting Started](https://prometheus.io/docs/prometheus/latest/installation/).
2. Configure prometheus.yml for remote write reception:

yaml

remote\_write:

- url: http://localhost:9090/api/v1/write # OTEL pushes here

scrape\_configs: [] # Optional for additional scraping

1. If using AWS auth: Add SigV4 for secure writes.
2. Start Prometheus and verify: Access UI at http://<prometheus-ip>:9090. Query metrics (e.g., up) to confirm data ingestion.
   * Reference: Prometheus remote write docs - [Prometheus Remote Write](https://prometheus.io/docs/prometheus/latest/configuration/configuration/#remote_write).

**Step 4: Add Prometheus Data Source to Grafana**

Since Grafana is already installed, focus on integration.

1. In Grafana sidebar, click gear icon > "Data sources" > "Add data source".
2. Select "Prometheus".
3. Configure:
   * URL: http://<prometheus-ip>:9090
   * Access: Browser or Server (use Server for production).
   * Scrape interval: 15s.
4. Save & Test: Grafana queries Prometheus API (e.g., /api/v1/query).
   * Reference: Grafana Prometheus data source docs - [Grafana Prometheus Guide](https://grafana.com/docs/grafana/latest/datasources/prometheus/).

**Step 5: Develop Minimal Custom Plugins if Gaps Exist (Optional)**

If standard Prometheus queries lack AWS-specific visualizations, create lightweight plugins (similar to Option 2 but minimal).

1. Scaffold: Use @grafana/create-plugin for a panel or data source extension (e.g., for custom trace views).
2. Implement: Add logic for OTEL data parsing (e.g., using @opentelemetry/api).
3. Build/install: Follow Option 2's build steps; add to Grafana plugins dir.
   * Effort: 5-10 hours if needed; skip if Prometheus suffices.
   * Reference: Grafana plugin docs - [Grafana Plugin Development](https://grafana.com/developers/plugin-tools/).

**Step 6: Build Dashboards and Alerts**

1. Create dashboard in Grafana: Add panels with Prometheus queries (e.g., rate(aws\_ec2\_cpuutilization\_average[5m])).
2. Import community templates: From <https://grafana.com/grafana/dashboards/> (e.g., ID 1860 for Prometheus, or OTEL-specific like 15983).
3. Set alerts: In panel editor > Alert tab, define thresholds (e.g., CPU >80%).
4. Correlate data: Use variables for services; add traces/logs if exporters configured.
   * Reference: Grafana dashboards docs - [Grafana Dashboards](https://grafana.com/docs/grafana/latest/dashboards/); Alerting - [Grafana Alerting](https://grafana.com/docs/grafana/latest/alerting/).

**Troubleshooting and Best Practices**

* **Common Issues**: Collector not exporting? Check YAML indentation and ports. Prometheus not receiving? Verify firewall (e.g., open 9090). Use AWS CLI to test: aws cloudwatch get-metric-statistics --namespace AWS/EC2 --metric-name CPUUtilization.
* **Security**: Use TLS for endpoints; IAM roles over keys. Monitor with Prometheus itself.
* **Scaling**: For large setups, federate Prometheus or use Thanos—[Prometheus Scaling](https://prometheus.io/docs/prometheus/latest/federation/).
* **Testing**: Start with one service (e.g., EC2). Use free-tier AWS to minimize costs.
* **References**: OTEL Docs - [opentelemetry.io](https://opentelemetry.io/docs/); Prometheus Docs - [prometheus.io](https://prometheus.io/docs/introduction/overview/); ADOT Docs - [aws-otel.github.io](https://aws-otel.github.io/); Grafana Docs - [grafana.com/docs](https://grafana.com/docs/grafana/latest/).

**Implementation Guide: Creating Action Buttons in Grafana OSS to Trigger AWS EventBridge and SSM Documents**

This guide provides a detailed, granular step-by-step process to create interactive buttons in your self-hosted free open-source (OSS) version of Grafana. These buttons will allow NOC/Command Center teams to send requests to AWS EventBridge, which in turn triggers an AWS Systems Manager (SSM) document for automated issue resolution (e.g., clearing a cache, restarting a stuck service, failing over resources, or terminating stalled SQL jobs).

Grafana OSS does not have native support for action buttons that trigger external APIs. To achieve this, we'll use two free community plugins:

* **Infinity Datasource Plugin**: Enables arbitrary REST API calls, including POST requests to AWS EventBridge's PutEvents API. It supports AWS authentication.
* **Button Panel Plugin**: Creates clickable buttons in dashboards that execute queries against datasources (in this case, Infinity to trigger the API call).

The workflow is: Button click → Query to Infinity datasource → POST to EventBridge PutEvents → EventBridge rule matches event → Triggers SSM Automation document.

**Prerequisites**:

* Grafana OSS (version 10.x or later recommended for best plugin support; check via Grafana UI under "Help" > "About"). Ensure it's running on a server with internet access for plugin downloads.
* AWS account with resources to automate (e.g., EC2 for service restarts).
* AWS CLI installed and configured on your workstation for testing—[AWS CLI Docs](https://docs.aws.amazon.com/cli/latest/userguide/getting-started-install.html).
* IAM permissions: Create a user/role with policies like "AmazonEventBridgeFullAccess", "AmazonSSMFullAccess", and specific actions (e.g., "events:PutEvents", "ssm:StartAutomationExecution").
* Security: If Grafana runs on an EC2 instance, attach an IAM role for credential-less access (preferred). Otherwise, use access keys (rotate regularly).
* Networking: Ensure Grafana server can reach AWS endpoints (e.g., events.<region>.amazonaws.com).
* Basic tools: Git, terminal access to Grafana server.

**Estimated Effort**: 15-20 steps, 4-8 hours (including testing; more for custom SSM documents).

**Limitations**:

* Plugins are community-maintained; test thoroughly.
* Infinity requires enabling "dangerous" HTTP methods (POST) via config, which could pose security risks if misused.
* AWS costs for API calls/EventBridge/SSM executions.
* No direct JS execution in panels due to Grafana sandboxing; relies on plugins.
* For complex auth, Infinity uses access keys (SigV4 signing implied but not explicit in docs).

**Step 1: Set Up AWS Resources for EventBridge and SSM**

Configure AWS to receive events and trigger automations. This ensures buttons have a target.

1. Log in to AWS Management Console (console.aws.amazon.com).
2. Create an SSM Automation Document:
   * Go to Systems Manager > Automation > Documents > Create document.
   * Choose "Automation" type.
   * In the editor, define YAML/JSON for your action (e.g., restart service on EC2).
     + Example for restarting a service (adapt for your use case like clearing cache):

yaml

description: Restart stuck service on EC2

schemaVersion: '0.3'

assumeRole: '{{ assumeRole }}'

parameters:

instanceId:

type: String

description: EC2 Instance ID

serviceName:

type: String

description: Service to restart (e.g., nginx)

mainSteps:

- name: restartService

action: aws:runCommand

inputs:

DocumentName: AWS-RunShellScript

InstanceIds: ['{{ instanceId }}']

Parameters:

commands: ['sudo systemctl restart {{ serviceName }}']

* + - Name it (e.g., "RestartServiceAutomation").
    - Create the document.
  + Reference: AWS SSM Documents docs - [SSM Automation Documents](https://docs.aws.amazon.com/systems-manager/latest/userguide/automation-documents.html).

1. Create an EventBridge Rule to Trigger SSM:
   * Go to EventBridge > Rules > Create rule.
   * Name it (e.g., "GrafanaTriggerSSM").
   * Event source: Other (custom event).
   * Event pattern: Define to match your custom event (e.g., from Grafana button).
     + Example JSON pattern:

json

{

"source": ["grafana.actions"],

"detail-type": ["RestartService"]

}

* + Target: Systems Manager Automation.
    - Select your SSM document (e.g., "RestartServiceAutomation").
    - Configure input: Transformer or constant (pass parameters like instanceId from event detail).
  + Create the rule.
  + Reference: EventBridge rules docs - [EventBridge Rules](https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-rules.html); SSM integration - [EventBridge SSM Targets](https://docs.aws.amazon.com/eventbridge/latest/userguide/eb-targets.html#targets-ssm).

1. Test via AWS CLI: Send a sample event.
   * aws events put-events --entries '[{"Source":"grafana.actions","DetailType":"RestartService","Detail":"{\"instanceId\":\"i-1234567890abcdef0\",\"serviceName\":\"nginx\"}"}]'
   * Verify in SSM Executions console that it triggered.
   * Reference: PutEvents API - [EventBridge PutEvents](https://docs.aws.amazon.com/eventbridge/latest/APIReference/API_PutEvents.html).

**Step 2: Install the Infinity Datasource Plugin in Grafana**

This plugin enables API calls to EventBridge.

1. On your Grafana server, open a terminal as admin (e.g., SSH to the host).
2. Install via Grafana CLI: Run grafana-cli plugins install yesoreyeram-infinity-datasource.
3. Restart Grafana: sudo systemctl restart grafana-server (adjust for your OS/init system).
4. Verify: In Grafana UI, go to Configuration > Data sources > Add data source > Search for "Infinity". If listed, it's installed.
   * Reference: Grafana plugin installation docs - [Install Plugins](https://grafana.com/docs/grafana/latest/administration/plugins-management/); Infinity docs - [Infinity Plugin Docs](https://grafana.github.io/grafana-infinity-datasource/).

**Step 3: Configure the Infinity Datasource for EventBridge API Calls**

Set up the datasource to handle POST requests to EventBridge.

1. In Grafana UI, go to Configuration > Data sources > Add data source > Select "Infinity".
2. Configure basic settings:
   * Name: "EventBridgeTrigger".
   * URL: Leave blank (or set to base if needed; queries will specify full URLs).
   * Allowed HTTP Methods: Enable "Allow dangerous methods (POST, PUT, DELETE, PATCH)" in the settings (this is required for PutEvents).
3. Set authentication for AWS:
   * Auth Type: AWS.
   * Enter AWS Access Key ID and Secret Access Key (from IAM user with PutEvents permission).
   * Region: Your AWS region (e.g., us-east-1).
   * If Grafana on EC2: Use IAM role instead (leave keys blank; ensure role has permissions).
   * Note: Infinity handles AWS SigV4 signing when keys/role provided.
4. Save & Test: Click to verify connection (it tests basic auth).
   * Reference: Infinity AWS auth - Search changelog for "AWS authentication" in plugin docs.

**Step 4: Install the Button Panel Plugin in Grafana**

This plugin provides the buttons.

1. On Grafana server terminal: Run grafana-cli plugins install speakyourcode-button-panel.
2. Restart Grafana: sudo systemctl restart grafana-server.
3. Verify: In a dashboard, add a new panel > Search visualizations for "Button". If available, it's installed.
   * Reference: Button Panel docs - [Button Panel Plugin](https://grafana.com/grafana/plugins/speakyourcode-button-panel/); General installation as above.

**Step 5: Create a Dashboard with Action Buttons**

Build the interactive dashboard.

1. In Grafana sidebar, click "Dashboards" > "New" > "New Dashboard".
2. Add a panel: Click "Add a new panel".
3. Switch visualization: In panel editor, search and select "Button" (from the plugin).
4. Configure buttons:
   * In the "Buttons" section, add a new button.
     + Label: Descriptive text (e.g., "Restart Stuck Service").
     + Action Type: Query (to datasource).
     + Datasource: Select "EventBridgeTrigger" (your Infinity datasource).
     + Query: Define as JSON for Infinity to make a POST to EventBridge.
       - Example for restarting service:

json

{

"type": "json",

"url": "https://events.us-east-1.amazonaws.com/",

"method": "POST",

"data": "{\"Entries\":[{\"Source\":\"grafana.actions\",\"DetailType\":\"RestartService\",\"Detail\":\"{\\\"instanceId\\\":\\\"i-1234567890abcdef0\\\",\\\"serviceName\\\":\\\"nginx\\\"}\"}]}",

"parser": "backend"

}

* + - * + Adapt "url" to your region, "data" to match your event pattern/parameters (e.g., for terminating SQL job, change Detail to include job ID).
        + Use Grafana variables (e.g., ${instanceId}) for dynamic inputs from dashboard dropdowns.
    - Ref ID: Auto-generated, but ensure unique.

1. Add more buttons: Repeat for other actions (e.g., "Clear Cache" with different event DetailType like "ClearCache").
   * Arrangement: Choose horizontal/vertical layout in panel options.
2. Test the button: Save panel, return to dashboard, click button. Monitor AWS CloudTrail/EventBridge for the PutEvents call, and SSM for execution.
   * If fails: Use Grafana's Query Inspector (in panel editor) to debug the request.
3. Enhance: Add dashboard variables (top of dashboard > Settings > Variables) for user inputs (e.g., dropdown for instanceId).
   * Reference: Button Panel query examples - Plugin page examples for InfluxDB/PostgreSQL (adapt to Infinity JSON); Infinity query docs - [Infinity Queries](https://grafana.github.io/grafana-infinity-datasource/docs/query_types/json).

**Step 6: Set Up Alerts or Additional Safeguards (Optional)**

1. To prevent accidental clicks, add confirmation: Button Panel doesn't support natively, but use dashboard notes or separate panels.
2. Integrate with Grafana alerts: If button tied to metrics, trigger alerts that notify before manual action.
   * Reference: Grafana alerting - [Alerting Guide](https://grafana.com/docs/grafana/latest/alerting/).

**Troubleshooting and Best Practices**

* **Common Issues**: Plugin not loading? Check Grafana logs (/var/log/grafana/grafana.log) for errors; ensure versions compatible. API call fails? Test with AWS CLI first; verify IAM permissions with aws sts get-caller-identity.
* **Security**: Restrict dashboard access via Grafana users/roles. Use short-lived credentials. Audit EventBridge rules to prevent unauthorized triggers.
* **Testing**: Use AWS free-tier or test accounts. Monitor costs via AWS Cost Explorer.
* **Scaling**: For multiple actions, create dedicated dashboards per team. If plugins insufficient, consider custom plugin development (similar to previous guides).
* **Alternatives if Plugins Fail**: Use Text panel with HTML links to a custom web app that triggers APIs, but this is less integrated.

**Implementation Guide: Integrating Grafana OSS with ServiceNow for Automatic Ticket Creation**

This guide provides a detailed, granular step-by-step process to integrate your self-hosted free open-source (OSS) version of Grafana with ServiceNow to automatically create incident tickets (e.g., based on alerts like high CPU usage or service downtime). The integration uses Grafana's built-in alerting system and webhook notification channels to send POST requests directly to ServiceNow's REST API (Table API for the "incident" table). No additional plugins are required, as this leverages core Grafana OSS features (available in version 8.x or later; alerting is fully supported in OSS since v8).

When a Grafana alert fires (or resolves), it will send a webhook payload to ServiceNow, creating a new incident ticket with details from the alert (e.g., metric name, value, timestamp). This is "automatic" in the sense that alerts trigger the process without manual intervention.

**Prerequisites**:

* Grafana OSS (version 9.x or later recommended for improved alerting; check via Grafana UI under "Help" > "About"). Ensure alerting is enabled (default in OSS).
* ServiceNow instance (e.g., a developer or production instance like <https://dev12345.service-now.com>). You need admin access or permissions to create API users.
* AWS or on-prem server running Grafana with internet access to reach your ServiceNow instance URL.
* Basic networking: Ensure Grafana server can reach ServiceNow endpoints (e.g., over HTTPS port 443; check firewalls/VPCs).
* ServiceNow API knowledge: Familiarity with REST APIs; we'll use Basic Authentication (username/password) for simplicity. For production, consider OAuth (requires additional setup in ServiceNow).
* Test data: An existing Grafana dashboard with metrics (e.g., from CloudWatch as in previous guides) to set up alerts.
* Tools: Web browser for Grafana/ServiceNow UIs, terminal for Grafana server (to restart if needed), and optionally curl/AWS CLI/Postman for testing API calls.
* Security note: Use a dedicated ServiceNow user with minimal permissions (e.g., "itil" role for incident creation). Avoid using admin credentials in production.

**Estimated Effort**: 15-20 steps, 2-4 hours (including testing; more if customizing fields or handling OAuth).

**Limitations**:

* Grafana OSS alerting is rule-based; it doesn't support ad-hoc ticket creation without alerts.
* Webhook sends on alert state changes (firing, resolving); no direct "button" for manual tickets (see previous guide for buttons via plugins).
* ServiceNow costs for API usage/incidents; test in a dev instance.
* If you have ServiceNow Event Management module, there's a built-in webhook integration for Grafana alerts (per ServiceNow docs), but this guide uses the direct API for broader compatibility without additional modules.

**Step 1: Set Up API Access in ServiceNow**

Create a dedicated user and verify API endpoint for incident creation.

1. Log in to your ServiceNow instance (e.g., https://<your-instance>.service-now.com) with admin privileges.
2. Create a new user for API access:
   * Navigate to System Security > Users and Groups > Users > New.
   * Fill in: User ID (e.g., "grafana\_api\_user"), First name/Last name (descriptive), Password (strong; note it down securely).
   * Assign roles: At minimum, "itil" (for incident management) and "rest\_service" (for API access). If restricted, ensure permissions for "incident" table create.
   * Save the user.
   * Reference: ServiceNow user creation docs - [ServiceNow Users](https://docs.servicenow.com/bundle/zurich-platform-administration/page/administer/users-and-groups/task/t_CreateAUser.html).
3. (Optional but recommended) Test the user login: Log out and log in as the new user to confirm.
4. Explore the REST API endpoint:
   * Navigate to System Web Services > REST > REST API Explorer.
   * Select Namespace: "now", API Name: "Table API", API Version: "v1".
   * Select HTTP Method: "POST", Table Name: "incident".
   * In the request builder, add sample fields (e.g., short\_description: "Test Incident", description: "From Grafana").
   * Click "Send" to test (use the new user's credentials if prompted).
   * Note the endpoint URL generated: https://<your-instance>.service-now.com/api/now/table/incident.
   * Reference: ServiceNow REST API Explorer docs - [REST API Explorer](https://docs.servicenow.com/bundle/zurich-application-development/page/integrate/inbound-rest/task/t_GetStartedWithRESTAPIExplorer.html).
5. Verify incident creation manually via curl (from your terminal or Postman):
   * Install curl if needed (e.g., on Linux: sudo apt install curl).
   * Run:

text

curl -X POST "https://<your-instance>.service-now.com/api/now/table/incident" \

-H "Accept: application/json" \

-H "Content-Type: application/json" \

-H "Authorization: Basic $(echo -n '<username>:<password>' | base64)" \

-d '{"short\_description":"Test from Curl","description":"This is a test incident","priority":"3"}'

* + - Replace <your-instance>, <username>, <password> with your values.
    - Expected response: JSON with the new incident details (e.g., {"result": {"number": "INC0010001", ...}}).
  + Check in ServiceNow: Go to Incident > All to see the new ticket.
  + If fails: Check errors (e.g., 401 Unauthorized means bad credentials; 403 Forbidden means insufficient roles).
  + Reference: ServiceNow Table API docs - [Table API](https://docs.servicenow.com/bundle/zurich-api-reference/page/integrate/inbound-rest/reference/r_TableAPI.html).

**Step 2: Configure a Webhook Notification Channel in Grafana for ServiceNow**

Set up the channel to POST alert data to ServiceNow's API.

1. Log in to Grafana UI (e.g., http://<grafana-ip>:3000) with admin credentials.
2. Navigate to alerting configuration: Sidebar > Alerting > Notification channels (in older versions) or Contact points (in v9+; this guide uses v10+ terminology—adjust if on older OSS).
3. Click "New contact point" (or "Add channel" in older versions).
4. Configure the contact point:
   * Name: "ServiceNow Incident Creator".
   * Integration: Select "Webhook".
   * URL: https://<your-instance>.service-now.com/api/now/table/incident (from Step 1).
   * HTTP Method: POST.
   * Basic Auth: Enable, then enter Username (e.g., "grafana\_api\_user") and Password (from ServiceNow user).
   * Custom Headers: Add two headers:
     + Header 1: Name = "Accept", Value = "application/json".
     + Header 2: Name = "Content-Type", Value = "application/json".
   * Skip TLS Verification: Enable if your ServiceNow instance uses self-signed certs (not recommended for prod).
5. Customize the payload (message template) to map alert data to ServiceNow incident fields:
   * In the "Webhook settings" > "Custom payload" or "Message" field, use Grafana's template syntax to create a JSON body.
   * Example template (paste this into the payload field):

text

{

"short\_description": "Grafana Alert: {{ .RuleName }}",

"description": "Alert fired at {{ .StartsAt }}.\nState: {{ .Status }}.\nMessage: {{ .Annotations.summary }}.\nValue: {{ .Values.A }}.\nLabels: {{ range .LabelsSorted }}{{ .Name }}={{ .Value }} {{ end }}",

"priority": "{{ if eq .Status \"firing\" }}2{{ else }}4{{ end }}",

"category": "Monitoring",

"subcategory": "Grafana Alert",

"impact": "3",

"urgency": "3",

"comments": "Generated automatically by Grafana OSS"

}

* + - This uses Grafana variables (e.g., {{ .RuleName }} for alert name, {{ .Status }} for firing/resolved). Adapt fields to your ServiceNow schema (e.g., priority: 1=Critical, 5=Planning).
    - Reference: Grafana template variables - [Alert Notification Templates](https://grafana.com/docs/grafana/latest/alerting/fundamentals/notifications/message-templating/).

1. Save the contact point.
2. Test the contact point: In the configuration page, click "Test" and send a sample notification. Check ServiceNow for a new incident with test data.
   * If fails: Use Grafana logs (/var/log/grafana/grafana.log) or Query Inspector in a test panel to debug. Common errors: Invalid URL (check region/instance), auth issues (verify user/pass), or JSON syntax (validate with jsonlint.com).

**Step 3: Create or Configure an Alert Rule in Grafana**

Tie the notification to alerts for automatic triggering.

1. Go to a dashboard with metrics (e.g., from previous CloudWatch setup).
2. Edit a panel: Click panel title > Edit.
3. Go to the "Alert" tab (if no metrics, add a query first).
4. Create a new alert rule:
   * Rule Name: Descriptive (e.g., "High CPU Alert").
   * Query: Use your existing metric query (e.g., EC2 CPUUtilization > 80%).
   * Conditions: Set threshold (e.g., WHEN avg() OF query(A, 5m, now) IS ABOVE 80).
   * Evaluation: Every 1m, for 5m.
   * No Data/Error Handling: Set to "Alerting" or "Keep Last State" as needed.
5. Add notifications:
   * In the "Notifications" section, select your "ServiceNow Incident Creator" contact point.
   * Optionally, add message overrides or tags.
6. Save the panel and dashboard.
   * Reference: Grafana alerting docs - [Create Alert Rules](https://grafana.com/docs/grafana/latest/alerting/alerting-rules/create-grafana-managed-rule/).

**Step 4: Test the Integration**

Trigger an alert to create a ticket automatically.

1. Force an alert: Temporarily lower the threshold in your alert rule to make it fire (e.g., CPU > 0%), save, and wait for evaluation.
2. Monitor Grafana: Go to Alerting > Alert rules to see state change to "Alerting".
3. Verify in ServiceNow: Go to Incident > All; search for the short\_description matching your alert name. Check details populated from the template.
4. Resolve the alert: Adjust threshold back; wait for "OK" state. If configured, this can update the ticket (e.g., add a comment via resolved status in template).
5. Debug if needed: Check Grafana notification history (Alerting > Notifications) or ServiceNow logs (System Logs > Transactions) for errors.

**Step 5: Production Hardening and Customization (Optional)**

1. Customize fields: Add more ServiceNow fields (e.g., "assignment\_group": "Monitoring Team") via template. Reference ServiceNow incident schema - [Incident Table](https://docs.servicenow.com/bundle/zurich-it-service-management/page/product/incident-management/reference/r_IncidentTable.html).
2. Handle resolutions: Modify template to POST to update existing incidents (change URL to /api/now/table/incident/<sys\_id>, method PATCH).
3. OAuth instead of Basic Auth: In ServiceNow, create OAuth app (System OAuth > Application Registry > New > Create an OAuth API endpoint). In Grafana webhook, use custom headers for Bearer token (requires token refresh logic, possibly via script).
4. Rate limiting: Monitor API calls; ServiceNow limits apply.
5. Security: Encrypt passwords in grafana.ini; use HTTPS only.
   * Reference: Grafana webhook docs - [Webhook Notifications](https://grafana.com/docs/grafana/latest/alerting/contact-points/notifiers/webhook/).

**Implementation Guide: Creating Modular Dashboards in Grafana OSS**

This guide provides a detailed, granular step-by-step process to create modular dashboards in your self-hosted free open-source (OSS) version of Grafana. "Modular dashboards" refer to designing reusable, composable components that can be shared across multiple dashboards, reducing duplication and easing maintenance. In Grafana OSS (version 8.3 or later required for key features like library panels; check your version via Grafana UI under "Help" > "About"), this is achieved through:

* **Dashboard variables and templating**: For dynamic, parameterized queries.
* **Library panels**: Reusable panels that update everywhere when edited.
* **Repeating rows and panels**: For modular layouts based on variables.
* **Dashboard organization**: Using folders, playlists, and links.
* **Dashboards as code**: Exporting/importing JSON for version control and automation (optional advanced step using free tools like grafanalib).

This approach allows you to build dashboards like building blocks (e.g., a reusable CPU panel module used in multiple service dashboards). All features are available in Grafana OSS without plugins or Enterprise licensing.

**Prerequisites**:

* Grafana OSS (version 8.3+; if older, upgrade via your package manager, e.g., sudo apt upgrade grafana on Ubuntu—backup first).
* Administrative access to Grafana UI and server (for file-based provisioning if needed).
* A data source configured (e.g., CloudWatch from previous guides) with sample metrics.
* Basic tools: Text editor for JSON, Git for version control (optional).
* Networking: Grafana server access to its own API if automating imports.

**Estimated Effort**: 10-15 steps, 2-4 hours for basic modularity (more for advanced code-based approaches).

**Notes**:

* Modularity in OSS is UI-driven with some automation; for fully programmatic generation, use external OSS tools.
* Test in a non-production dashboard to avoid disruptions.
* References are to official Grafana docs (grafana.com/docs) unless noted.

**Step 1: Set Up Dashboard Variables for Dynamic Modularity**

Variables allow dashboards to be reused across contexts (e.g., different servers or regions) by parameterizing queries.

1. Log in to Grafana UI (e.g., http://<grafana-ip>:3000) with admin credentials.
2. Create or open a dashboard: Sidebar > Dashboards > New > New Dashboard (or edit an existing one).
3. Add variables: Dashboard top-right > Settings (gear icon) > Variables > Add variable.
4. Configure a variable:
   * Name: Descriptive (e.g., "instanceId").
   * Type: Query (for dynamic values from data source) or Custom (static list).
   * Example for Query type (using CloudWatch data source):
     + Data source: Select your CloudWatch source.
     + Query: Use Grafana's query editor, e.g., for EC2 instances: Namespace = AWS/EC2, Metric = CPUUtilization, Dimension = InstanceId (this auto-populates from AWS).
     + Refresh: On dashboard load.
     + Multi-value: Enable if users can select multiple (e.g., multiple instances).
     + Include All option: Enable for an "All" wildcard.
5. Add more variables as needed (e.g., "region" as Custom: us-east-1,us-west-2).
6. Save variables and dashboard.
7. Use in panels: In a panel query, reference via ${variableName} (e.g., Dimension filter: InstanceId = ${instanceId}).
8. Test: Reload dashboard; change variable values in the top dropdown—panels should update dynamically.
   * Reference: Grafana variables docs - [Dashboard Variables](https://grafana.com/docs/grafana/latest/dashboards/variables/add-template-variables/).

**Step 2: Create Library Panels for Reusable Modules**

Library panels are standalone, editable components that can be inserted into multiple dashboards. Changes to a library panel propagate everywhere.

1. In an existing dashboard, create or edit a panel (e.g., a graph showing CPU utilization).
2. Configure the panel fully: Add query, visualization options, thresholds.
3. Save as library panel: Panel top-right menu (three dots) > More > New library panel.
4. In the dialog:
   * Name: Descriptive (e.g., "EC2 CPU Utilization Module").
   * Folder: Select or create a folder for organization (e.g., "Reusable Modules").
   * Save.
5. The panel is now in the library; original dashboard panel becomes a linked instance.
6. Add to another dashboard: In a new/existing dashboard, click "Add" > "Panel from library".
7. Search/select your library panel > Add.
8. Edit library panel centrally: Sidebar > Dashboards > Manage library panels > Select your panel > Edit > Make changes (e.g., add alert) > Save. All instances update.
9. Unlink if needed: In a dashboard, panel menu > More > Unlink library panel (creates a local copy).
10. Test: Modify the library panel; verify changes in all dashboards using it.
    * Reference: Grafana library panels docs - [Manage Library Panels](https://grafana.com/docs/grafana/latest/dashboards/build-dashboards/manage-library-panels/).

**Step 3: Use Repeating Rows and Panels for Layout Modularity**

Repeats create modular sections that duplicate based on variables (e.g., one row per selected instance).

1. In a dashboard, add a row: Click "Add" > "New row" (or convert a panel to row via menu).
2. Add panels to the row (e.g., insert a library panel from Step 2).
3. Enable repeat: Row top-right menu > Repeat options.
4. Configure:
   * Repeat for: Select a multi-value variable (e.g., ${instanceId}).
   * Direction: Horizontal or Vertical.
5. For individual panels: Panel menu > Repeat options > Repeat by variable (same as above).
6. Save and test: Select multiple values in the variable dropdown—rows/panels duplicate automatically, each filtered to one value.
7. Combine with variables: Use ${\_\_value} in panel titles/queries for context (e.g., title: "CPU for ${instanceId}").
   * Reference: Grafana repeating docs - [Repeat Panels or Rows](https://grafana.com/docs/grafana/latest/dashboards/build-dashboards/repeat-panels-or-rows/).

**Step 4: Organize Dashboards with Folders, Playlists, and Links for Modular Navigation**

Group and link dashboards for a modular ecosystem.

1. Create folders: Sidebar > Dashboards > New > New folder.
   * Name: E.g., "Modular Components".
   * Move dashboards: In dashboard list, select and move to folder.
2. Add dashboard links: In a dashboard > Settings > Links > Add link.
   * Type: Dashboard (link to another dashboard) or Panels (link to specific panel).
   * Use variables: E.g., link URL with ?var-instanceId=${instanceId} for passing context.
3. Create playlists: Sidebar > Dashboards > Playlists > New playlist.
   * Add dashboards from folders.
   * Set interval (e.g., cycle every 30s) for auto-rotation.
   * Use for NOC views combining modular dashboards.
4. Test navigation: Click links in one dashboard; ensure variables pass correctly.
   * Reference: Grafana organization docs - [Dashboard Folders](https://grafana.com/docs/grafana/latest/dashboards/dashboard-folders/); [Playlists](https://grafana.com/docs/grafana/latest/dashboards/playlists/); [Links](https://grafana.com/docs/grafana/latest/dashboards/links/dashboard-links/).

**Step 5: Implement Dashboards as Code for Advanced Modularity and Version Control (Optional)**

Manage dashboards as JSON files in Git for programmatic modularity (e.g., generate variants).

1. Export a dashboard: Dashboard top-right > Share > Export > View JSON > Copy or Download.
2. Save as file: E.g., modular-dashboard.json in a Git repo.
3. Edit JSON manually: Use a text editor to add modularity (e.g., insert library panel UIDs under "panels").
   * Reference JSON structure in Grafana API docs.
4. Import back: Sidebar > Dashboards > Import > Upload JSON file > Select file > Import.
5. For automation, use grafanalib (OSS Python library):
   * Install locally: pip install grafanalib (on your workstation, not Grafana server).
   * Create Python script:

python

from grafanalib.core import Dashboard, TimeSeries, Target

dashboard = Dashboard(

title="Modular Dashboard",

panels=[

TimeSeries(

title="Reusable CPU Panel",

dataSource="CloudWatch",

targets=[Target(

namespace="AWS/EC2",

metricName="CPUUtilization",

dimensions={"InstanceId": "${instanceId}"}

)]

)

]

).auto\_panel\_ids()

print(dashboard.to\_json\_data()) # Output JSON to file

* + Run: python script.py > dashboard.json.
  + Import the generated JSON into Grafana.

1. Version control: Commit JSON to Git; use CI/CD (e.g., GitHub Actions) to push to Grafana API.
   * API import: Use curl with API key (create via Configuration > API Keys).
     + Example: curl -X POST -H "Authorization: Bearer <api-key>" -H "Content-Type: application/json" --data @dashboard.json http://<grafana-ip>:3000/api/dashboards/db.
2. Test: Generate variants (e.g., change titles in script), import, verify modularity.
   * Reference: Grafana provisioning docs - [Provision Dashboards](https://grafana.com/docs/grafana/latest/administration/provisioning/#dashboards); grafanalib GitHub - [grafanalib Repo](https://github.com/weaveworks/grafanalib).

**Troubleshooting and Best Practices**

* **Issues**: Variables not resolving? Check query syntax. Library panels not updating? Ensure version 8.3+. JSON import fails? Validate with jsonlint.com.
* **Security**: Use API keys with minimal permissions. Version control sensitive data.
* **Scaling**: Start small (one variable/library panel). For teams, use folders for access control.
* **Testing**: Duplicate dashboards for experiments.

**Implementation Guide: Creating Modular Alerts/Monitors in Grafana OSS with ServiceNow Integration**

This guide provides a detailed, granular step-by-step process to create modular alerts (also referred to as monitors) in your self-hosted free open-source (OSS) version of Grafana, with configurations to push notifications to ServiceNow for automatic ticket creation. "Modular alerts" means designing reusable, version-controlled alert rules that can be easily replicated, updated, or shared across environments. In Grafana OSS (version 8.x or later required for unified alerting; version 9+ recommended for provisioning enhancements—check via Grafana UI under "Help" > "About"), this is achieved through:

* **File provisioning**: Define alert rules, contact points (e.g., ServiceNow webhook), and notification policies as YAML files. This allows modularity via code (e.g., Git version control, templating with tools like Helm for reuse).
* **Alert rule groups**: Group related rules into namespaces/folders for organization and reuse.
* **Reusable queries**: Use data source UIDs and parameterized queries where possible (e.g., via labels or dimensions).
* **Integration with ServiceNow**: Build on webhook notifications (from previous guides) to create incidents, but provision them modularly.

This setup enables you to define a "module" (e.g., a YAML file for CPU alerts) that can be copied/adapted for different services (e.g., EC2 vs. Lambda) and ensures pushes to ServiceNow are consistent.

**Prerequisites**:

* Grafana OSS (version 9+; if older, upgrade via package manager, e.g., sudo apt upgrade grafana on Ubuntu—backup your database first via grafana-cli admin data-export).
* Administrative access to Grafana server (e.g., SSH) and file system (default config at /etc/grafana/grafana.ini on Linux).
* A configured data source (e.g., CloudWatch from earlier guides) with metrics to monitor.
* ServiceNow setup from previous guide: API endpoint (e.g., https://<your-instance>.service-now.com/api/now/table/incident), dedicated user with credentials, and tested incident creation.
* Tools: Text editor (e.g., vim/nano on server), Git (optional for version control), terminal access.
* Security: Use environment variables or secrets management (e.g., via Docker if containerized) for ServiceNow credentials to avoid plain text in files.
* Networking: Grafana server must reach ServiceNow API (HTTPS port 443).

**Estimated Effort**: 15-20 steps, 3-6 hours (including testing; more for custom templating or Git integration).

**Notes**:

* Alert rules in OSS are standalone (not directly tied to dashboard variables), but provisioning makes them modular through file reuse.
* For advanced modularity (e.g., generating YAML dynamically), use external OSS tools like Terraform or Python scripts (optional Step 5).
* Test in a non-production environment; provisioning reloads on Grafana restart or API calls.
* Limitations: OSS lacks Enterprise features like alert silencing rules or advanced RBAC for alerts; modularity is file-based, not UI-templated.

**Step 1: Enable File Provisioning in Grafana**

Configure Grafana to load alerting resources from YAML files for modularity.

1. SSH to your Grafana server.
2. Edit the configuration file: Open /etc/grafana/grafana.ini (or your custom path) with sudo (e.g., sudo nano /etc/grafana/grafana.ini).
3. Under [unified\_alerting] section (add if missing):
   * Set enabled = true (default in v8+).
4. Under [alerting] section (for legacy compatibility, but unified is preferred):
   * Ensure it's not conflicting; comment out if using unified.
5. Create provisioning directories: Run sudo mkdir -p /etc/grafana/provisioning/alerting (default path; customizable).
6. In grafana.ini, under [paths] section:
   * Add or edit provisioning = /etc/grafana/provisioning (points to your dir).
7. Restart Grafana: sudo systemctl restart grafana-server (adjust for your init system).
8. Verify: In Grafana UI, go to Alerting > Alert rules—if no errors in logs (/var/log/grafana/grafana.log), provisioning is enabled.
   * Reference: Grafana file provisioning docs - [Provision Alerting Resources](https://grafana.com/docs/grafana/latest/alerting/set-up/provisioning-alerting-resources/file-provisioning/).

**Step 2: Provision a Modular Contact Point for ServiceNow Push**

Define the ServiceNow webhook as a reusable YAML module. This contact point can be referenced in multiple alert rules.

1. On the server, navigate to /etc/grafana/provisioning/alerting.
2. Create a file: sudo nano contact-points.yaml.
3. Add YAML content for the ServiceNow webhook (adapt from previous guide; this is modular as you can copy this file across Grafana instances):

yaml

apiVersion: 1

contactPoints:

- orgId: 1 # Default org; change if multi-org

name: ServiceNow-Incident-Creator

type: webhook

settings:

url: https://<your-instance>.service-now.com/api/now/table/incident

httpMethod: POST

username: <your-servicenow-username> # E.g., grafana\_api\_user; use env vars for security

password: <your-servicenow-password> # Secure this!

headers:

- name: Accept

value: application/json

- name: Content-Type

value: application/json

body: | # JSON template for incident; use Grafana placeholders

{

"short\_description": "Grafana Alert: {{ .RuleName }}",

"description": "Alert: {{ .Status }} at {{ .StartsAt }}. Message: {{ .Annotations.summary }}. Value: {{ .ValueString }}",

"priority": "{{ if eq .Status \"firing\" }}2{{ else }}4{{ end }}",

"category": "Monitoring",

"impact": "3",

"urgency": "3"

}

* + Replace placeholders (<your-instance>, etc.). For security, use Grafana secrets: Set password via env var in grafana.ini or Docker.

1. Set permissions: sudo chown -R grafana:grafana /etc/grafana/provisioning (ensure Grafana user can read).
2. Restart Grafana: sudo systemctl restart grafana-server.
3. Verify: In UI, Alerting > Contact points—see "ServiceNow-Incident-Creator" listed. Test by clicking "Test" (sends sample payload; check ServiceNow for test incident).
   * Reference: Grafana contact point provisioning - [Provision Contact Points](https://grafana.com/docs/grafana/latest/alerting/set-up/provisioning-alerting-resources/file-provisioning/#contact-points).

**Step 3: Provision Modular Alert Rule Groups**

Create reusable alert rules in YAML. Group them into modules (e.g., one file per monitor type like "CPU" or "DynamoDB").

1. In /etc/grafana/provisioning/alerting, create a file: sudo nano alert-rules-cpu.yaml (this is a module for CPU monitors; create more files for other types).
2. Add YAML for a rule group (reusable by copying/adapting the file):

yaml

apiVersion: 1

groups:

- orgId: 1

name: EC2-CPU-Monitors # Group name for modularity

folder: AWS-Monitors # Namespace/folder for organization

interval: 1m # Evaluation interval

rules:

- uid: cpu-high-alert # Unique ID for reference/reuse

title: High CPU Utilization

condition: B # Refers to query B below

data:

- refId: A

datasourceUid: <your-datasource-uid> # Get from UI: Configuration > Data sources > Copy UID

model: # Query (adapt for CloudWatch; make reusable with dimensions)

editorMode: code

expression: ""

hide: false

intervalMs: 1000

maxDataPoints: 43200

namespace: AWS/EC2

metricName: CPUUtilization

statistic: Average

dimensions:

InstanceId: "\*" # Wildcard for modularity; or parameterize if scripting

period: 300

region: default

- refId: B

queryType: ""

relativeTimeRange:

from: 600

to: 0

datasourceUid: \_\_expr\_\_

model:

conditions:

- evaluator:

params: [80]

type: gt

operator:

type: and

query:

params: [A]

reducer:

type: avg

type: query

datasource:

type: \_\_expr\_\_

uid: \_\_expr\_\_

expression: A

hide: false

intervalMs: 1000

maxDataPoints: 43200

type: reduce

execErrState: Error

for: 5m # Pending duration

annotations:

summary: CPU >80% on EC2 instance

labels:

severity: critical

service: ec2 # Labels for filtering/reuse

noDataState: NoData

* + Adapt query for your data source (e.g., add more rules in the group for low CPU, etc.). UID makes it reusable.

1. Create another modular file (e.g., sudo nano alert-rules-dynamodb.yaml) with similar structure but for DynamoDB metrics (e.g., ConsumedReadCapacityUnits > threshold).
2. Restart Grafana to load.
3. Verify: UI > Alerting > Alert rules—see groups/rules listed under "AWS-Monitors" folder. Simulate high CPU (e.g., stress an EC2) to test firing; check ServiceNow for ticket.
4. For modularity: Copy alert-rules-cpu.yaml to alert-rules-lambda.yaml, edit queries/dimensions, restart—new module created.
   * Reference: Grafana alert rule provisioning - [Provision Alert Rules](https://grafana.com/docs/grafana/latest/alerting/set-up/provisioning-alerting-resources/file-provisioning/#alert-rules).

**Step 4: Provision Notification Policies for Modular Routing to ServiceNow**

Link alerts to the contact point modularly.

1. Create file: sudo nano notification-policies.yaml.
2. Add YAML:

yaml

apiVersion: 1

policies:

- orgId: 1

receiver: ServiceNow-Incident-Creator # References your contact point

group\_by: [alertname, grafana\_folder, service] # Groups by labels for modularity

matchers:

- severity =~ "warning|critical" # Applies to labeled alerts

group\_wait: 30s

group\_interval: 5m

repeat\_interval: 1h

1. Restart Grafana.
2. Verify: UI > Alerting > Notification policies—see policy routing to ServiceNow. Test by firing an alert.
   * Reference: Grafana notification policy docs - [Provision Notification Policies](https://grafana.com/docs/grafana/latest/alerting/set-up/provisioning-alerting-resources/file-provisioning/#notification-policies).

**Step 5: Advanced Modularity with Version Control and Automation (Optional)**

Make modules reusable across teams/environments.

1. Initialize Git: In /etc/grafana/provisioning/alerting, run git init; git add .; git commit -m "Initial modular alerts".
2. For dynamic generation: On your workstation, use Python to template YAML (e.g., with Jinja2—install via pip if needed).
   * Example script (run locally):

python

import yaml

from jinja2 import Template

template = Template(open('alert-template.yaml.j2').read()) # Create a Jinja template file

rendered = template.render(service='ec2', threshold=80)

with open('alert-rules-ec2.yaml', 'w') as f:

yaml.dump(rendered, f)

* + Copy generated YAML to server, restart Grafana.

1. Use Terraform (OSS) for infra-as-code: Install Terraform, define resources like grafana\_alert\_rule.
   * Example TF file (from docs):

terraform

resource "grafana\_alert\_rule" "cpu\_high" {

name = "High CPU"

folder = "AWS-Monitors"

# ... (adapt from YAML)

}

* + Run terraform apply to provision via API.

1. Test: Update a module in Git, pull to server, restart—changes apply modularly.
   * Reference: Grafana Terraform provider - [Terraform Provisioning](https://grafana.com/docs/grafana/latest/alerting/set-up/provision-alerting-resources/terraform-provisioning/); API - [Alerting API](https://grafana.com/docs/grafana/latest/developers/http_api/alerting_provisioning/).

**Troubleshooting and Best Practices**

* **Issues**: Rules not loading? Check logs for YAML syntax errors (use yamllint.com). Conflicts? UIDs must be unique.
* **Security**: Avoid credentials in YAML; use Grafana encrypted secrets or env vars.
* **Scaling**: Use folders for namespaces; version YAML in Git for audits.
* **Testing**: Use Grafana's "Test rule" button in UI before provisioning.

This completes the implementation. Your alerts are now modular, provisioned as code, and integrated with ServiceNow. If you need YAML examples for specific monitors or tool setups, provide details.

**Implementation Guide: Promoting Non-Prod Grafana to Prod with Dashboards and Alerts as Code in a Fully Automated Telemetry Pipeline (Grafana OSS)**

This guide provides a detailed, granular step-by-step process to use your non-prod (e.g., development or staging) self-hosted free open-source (OSS) Grafana instance to safeguard production by treating dashboards and alerts as code. This creates a fully automated telemetry pipeline for promotion, ensuring changes are tested, version-controlled, and deployed safely. The approach leverages Grafana's file-based provisioning (YAML/JSON files), Git for version control (as code), and CI/CD tools (e.g., GitHub Actions or Jenkins, which are free/OSS) for automation. This is "safeguarding" via non-prod testing, pull request reviews, and automated deployments to prevent direct prod edits.

Key concepts:

* **Dashboards/Alerts as Code**: Export to JSON/YAML, store in Git, provision via files.
* **Telemetry Pipeline**: A GitOps workflow where non-prod changes are promoted to prod via automated CI/CD, with safeguards like tests and approvals.
* **Automation**: No manual UI changes in prod; everything flows from non-prod through code.
* All features are in Grafana OSS (version 9+ recommended for unified alerting provisioning; check via UI "Help" > "About"). No plugins needed.

**Prerequisites**:

* Two Grafana OSS instances: Non-prod (e.g., on a dev server) and Prod (on a secure server). Both self-hosted, with identical versions.
* Git installed on your workstation/server (free; e.g., sudo apt install git on Linux).
* CI/CD tool: GitHub (free for public/private repos) with Actions, or self-hosted Jenkins/GitLab (OSS).
* Administrative access to both Grafana servers (SSH for file copying/restarts).
* Data sources configured similarly in both (e.g., CloudWatch UIDs must match for alert portability).
* Tools: Text editor, terminal. Optional: Terraform (OSS) for infra-as-code.
* Security: Use IAM roles/environment variables for credentials; restrict prod access.
* Networking: Non-prod and prod can be isolated; CI/CD bridges them via secure file pushes (e.g., SCP/SSH).

**Estimated Effort**: 15-20 steps, 4-8 hours initial setup (plus ongoing for pipeline runs). Assumes basic Git/CI knowledge; adapt for your stack.

**Notes**:

* This builds on previous guides (e.g., provisioning alerts/dashboards as code).
* Safeguards: Non-prod testing prevents prod breakage; Git reviews catch issues; rollbacks via Git revert.
* Limitations: OSS lacks built-in dashboard testing; use manual validation or scripts. No native hot-reload for all resources (requires restart or API calls).

**Step 1: Set Up Version Control for Dashboards and Alerts as Code**

Export resources from non-prod Grafana and store in Git for modularity and promotion.

1. Log in to non-prod Grafana UI.
2. Export dashboards: For each dashboard > Top-right > Share > Export > View JSON > Download (or copy). Save as <dashboard-name>.json.
3. Export alerts (if not already provisioned): UI > Alerting > Alert rules > Select rule > Edit > Copy JSON (or use API: curl with API key—create key via Configuration > API Keys).
   * For full export: Use Grafana API—curl -H "Authorization: Bearer <api-key>" http://<non-prod-ip>:3000/api/v1/provisioning/alert-rules > alerts.json.
4. Initialize Git repo on your workstation: git init grafana-telemetry; cd grafana-telemetry.
5. Organize files modularly:
   * Create dirs: mkdir -p dashboards alerting/contact-points alerting/policies alerting/rules.
   * Place dashboard JSON in dashboards/ (e.g., ec2-monitoring.json).
   * Convert alerts to YAML (preferred for provisioning): Use a text editor or script to format (example from previous guides).
     + Example alerting/rules/ec2-cpu.yaml (adapt UIDs/queries):

yaml

apiVersion: 1

groups:

- orgId: 1

name: EC2-Monitors

folder: AWS

interval: 1m

rules:

- uid: ec2-cpu-high

title: High CPU

# ... (full rule as in previous guide)

* + Add contact points/policies similarly (e.g., alerting/contact-points/servicenow.yaml).

1. Commit: git add .; git commit -m "Initial export from non-prod".
2. Push to remote: Create a GitHub repo (private), add remote (git remote add origin <url>), push (git push -u origin main).
3. Test import back to non-prod: Copy files to non-prod server's /etc/grafana/provisioning/ dirs, restart Grafana (sudo systemctl restart grafana-server), verify in UI.
   * Reference: Grafana file provisioning docs - [Provision Dashboards](https://grafana.com/docs/grafana/latest/administration/provisioning/#dashboards); [Provision Alerting](https://grafana.com/docs/grafana/latest/alerting/set-up/provisioning-alerting-resources/file-provisioning/).

**Step 2: Configure File Provisioning on Both Non-Prod and Prod Grafana Instances**

Enable Grafana to load resources from files, ensuring prod is read-only from code.

1. On both servers, edit /etc/grafana/grafana.ini (sudo access):
   * Under [paths]: provisioning = /etc/grafana/provisioning.
   * Under [unified\_alerting]: enabled = true.
2. Create provisioning dirs: sudo mkdir -p /etc/grafana/provisioning/{dashboards,alerting}.
3. For dashboards, add /etc/grafana/provisioning/dashboards/providers.yaml:

yaml

apiVersion: 1

providers:

- name: default

type: file

updateIntervalSeconds: 30

options:

path: /etc/grafana/provisioning/dashboards/files # Sub-dir for JSON files

foldersFromFilesStructure: true

1. For alerting, files go directly in /etc/grafana/provisioning/alerting/ (no separate provider YAML needed; Grafana scans for \*.yaml).
2. Set ownership: sudo chown -R grafana:grafana /etc/grafana/provisioning.
3. Restart both instances: sudo systemctl restart grafana-server.
4. Verify on non-prod: Manually copy a test JSON/YAML file, restart, check UI for loaded resources.
   * Reference: Same as above; for alerting - [File Provisioning for Alerting](https://grafana.com/docs/grafana/latest/alerting/set-up/provisioning-alerting-resources/file-provisioning/).

**Step 3: Develop and Test Changes in Non-Prod**

Use non-prod as a safe sandbox; commit tested code to Git.

1. Make changes in non-prod UI (e.g., edit dashboard, create alert).
2. Export as in Step 1; overwrite Git files.
3. Commit/push: git commit -am "Updated EC2 dashboard"; git push.
4. Test safeguards:
   * Simulate failure: Intentionally break a query in non-prod UI, verify it doesn't affect prod.
   * Validate: Manually import to a temp non-prod dashboard, check alerts fire correctly (e.g., use test data or simulators).
5. Review: Use GitHub pull requests—branch from main (git checkout -b feature/new-alert), push, create PR, review code (e.g., check queries for efficiency).
6. Merge to main after approval/testing.
   * Reference: GitHub PR docs - [Pull Requests](https://docs.github.com/en/pull-requests).

**Step 4: Build the Automated Telemetry Pipeline with CI/CD**

Automate promotion from Git (post-non-prod merge) to prod Grafana files.

1. Set up CI/CD: Use GitHub Actions (free tier sufficient).
   * In your GitHub repo, create .github/workflows/deploy-to-prod.yaml:

yaml

name: Deploy to Prod Grafana

on:

push:

branches: [main] # Triggers on merge to main

jobs:

deploy:

runs-on: ubuntu-latest

steps:

- uses: actions/checkout@v4

- name: Deploy Files to Prod Server

uses: appleboy/scp-action@v0.1.4 # Free action for SCP

with:

host: ${{ secrets.PROD\_SERVER\_IP }}

username: ${{ secrets.PROD\_SSH\_USER }}

key: ${{ secrets.PROD\_SSH\_KEY }} # Store in GitHub Secrets

source: "dashboards/\*,alerting/\*"

target: /etc/grafana/provisioning/

- name: Restart Grafana on Prod

uses: appleboy/ssh-action@v1.0.3

with:

host: ${{ secrets.PROD\_SERVER\_IP }}

username: ${{ secrets.PROD\_SSH\_USER }}

key: ${{ secrets.PROD\_SSH\_KEY }}

script: sudo systemctl restart grafana-server

1. Add secrets to GitHub repo: Settings > Secrets and variables > Actions > New repository secret (e.g., PROD\_SERVER\_IP, PROD\_SSH\_USER, PROD\_SSH\_KEY—generate SSH key pair, add public to prod server ~/.ssh/authorized\_keys).
2. Test pipeline: Push a small change to main; monitor Actions tab for success. Verify prod UI updates without manual intervention.
3. Add safeguards to workflow:
   * Tests: Add step for JSON/YAML validation (e.g., use yamllint action: uses: ibiqlik/action-yamllint@v3).
   * Approvals: Require PR approvals before merge (repo Settings > Branches > Add rule for main).
   * Rollback: If issues, revert commit in Git, push—pipeline redeploys old files.
4. For advanced automation: Use Terraform provider for Grafana (OSS)—define resources like grafana\_dashboard in TF files, run in CI/CD to apply via API instead of file copy.
   * Install Terraform locally, add to workflow: uses: hashicorp/setup-terraform@v3.
   * Example TF:

terraform

provider "grafana" {

url = "http://<prod-ip>:3000"

auth = var.api\_key # From secrets

}

resource "grafana\_dashboard" "ec2" {

config\_json = file("dashboards/ec2-monitoring.json")

}

* + Reference: Grafana Terraform provider - [Terraform for Grafana](https://registry.terraform.io/providers/grafana/grafana/latest/docs); GitHub Actions docs - [Workflow Syntax](https://docs.github.com/en/actions/using-workflows/workflow-syntax-for-github-actions).

**Step 5: Monitor and Maintain the Pipeline**

Ensure ongoing safeguards.

1. Set up logging: In GitHub Actions, enable debug; on prod server, tail Grafana logs (journalctl -u grafana-server) post-deploy.
2. Test end-to-end: In non-prod, trigger an alert > Verify ServiceNow ticket (from previous integration) > Promote via pipeline > Confirm prod alert fires similarly.
3. Handle drifts: Disable UI edits in prod (via read-only users); if drifts occur, re-export and commit.
4. Scale: Add branches for environments (e.g., staging between non-prod and prod); use tags for releases.
5. Backup: Git backs up code; also backup Grafana DB periodically (grafana-cli admin data-export).

**Troubleshooting and Best Practices**

* **Issues**: Pipeline fails? Check SSH permissions or secrets. Resources not loading? Validate YAML/JSON syntax (use online tools like yamllint.com). API errors? Ensure UIDs unique.
* **Security**: Use encrypted secrets; least-privilege SSH users; monitor for unauthorized merges.
* **Safeguards**: Always test in non-prod; use canary deploys if scaling (e.g., partial file updates).
* **References**: Grafana provisioning - [Provisioning Overview](https://grafana.com/docs/grafana/latest/administration/provisioning/); Alerting API - [Provisioning API](https://grafana.com/docs/grafana/latest/developers/http_api/alerting_provisioning/); GitOps for Grafana - Community articles (search "Grafana GitOps").

# 📊 AWS APIs for Grafana Custom Plugins \*\*Status:\*\* TECHNICAL REFERENCE \*\*Last Updated:\*\* September 26, 2025 \*\*Author:\*\* DevOps Team \*\*Version:\*\* 1.0 --- ## 📚 Table of Contents - [Overview](#overview) - [CloudWatch APIs](#cloudwatch-apis) - [Aurora DB APIs](#aurora-db-apis) - [DynamoDB APIs](#dynamodb-apis) - [ECS APIs](#ecs-apis) - [Implementation Guide](#implementation-guide) - [Best Practices](#best-practices) --- ## Overview This document provides a comprehensive reference for AWS APIs that can be accessed through Grafana custom plugins. Each section includes the API methods, expected data structures, and practical examples for integration. > \*\*ℹ️ Key Information:\*\* All API responses follow AWS SDK response formats. Ensure proper IAM permissions are configured for the APIs you intend to access. --- ## CloudWatch APIs ### Primary Monitoring APIs #### \*\*GetMetricData\*\* Retrieves metric data points for multiple metrics with a single request. \*\*Request Structure:\*\* ```json { "MetricDataQueries": [ { "Id": "m1", "MetricStat": { "Metric": { "Namespace": "AWS/EC2", "MetricName": "CPUUtilization", "Dimensions": [ { "Name": "InstanceId", "Value": "i-1234567890abcdef0" } ] }, "Period": 300, "Stat": "Average" }, "ReturnData": true } ], "StartTime": "2025-09-26T00:00:00Z", "EndTime": "2025-09-26T23:59:59Z" } ``` \*\*Response Structure:\*\* ```json { "MetricDataResults": [ { "Id": "m1", "Label": "CPUUtilization", "Timestamps": [ "2025-09-26T00:00:00Z", "2025-09-26T00:05:00Z" ], "Values": [ 45.5, 47.2 ], "StatusCode": "Complete" } ], "Messages": [] } ``` #### \*\*ListMetrics\*\* Lists available metrics for a given namespace and dimensions. \*\*Request Structure:\*\* ```json { "Namespace": "AWS/Lambda", "MetricName": "Duration", "Dimensions": [ { "Name": "FunctionName" } ] } ``` \*\*Response Structure:\*\* ```json { "Metrics": [ { "Namespace": "AWS/Lambda", "MetricName": "Duration", "Dimensions": [ { "Name": "FunctionName", "Value": "my-function" } ] } ], "NextToken": null } ``` #### \*\*GetMetricStatistics\*\* Gets statistics for a specific metric. \*\*Request Structure:\*\* ```json { "Namespace": "AWS/ECS", "MetricName": "CPUUtilization", "Dimensions": [ { "Name": "ServiceName", "Value": "my-service" }, { "Name": "ClusterName", "Value": "my-cluster" } ], "StartTime": "2025-09-26T00:00:00Z", "EndTime": "2025-09-26T23:59:59Z", "Period": 3600, "Statistics": ["Average", "Maximum", "Minimum"] } ``` --- ## Aurora DB APIs ### RDS/Aurora Monitoring APIs #### \*\*DescribeDBInstances\*\* Retrieves information about Aurora DB instances. \*\*Request Structure:\*\* ```json { "DBInstanceIdentifier": "aurora-instance-1", "Filters": [ { "Name": "engine", "Values": ["aurora-mysql", "aurora-postgresql"] } ] } ``` \*\*Response Structure:\*\* ```json { "DBInstances": [ { "DBInstanceIdentifier": "aurora-instance-1", "DBInstanceClass": "db.r5.large", "Engine": "aurora-mysql", "DBInstanceStatus": "available", "AllocatedStorage": 100, "DBInstanceArn": "arn:aws:rds:us-east-1:123456789012:db:aurora-instance-1", "Endpoint": { "Address": "aurora-instance-1.xyz.us-east-1.rds.amazonaws.com", "Port": 3306 }, "DBClusterIdentifier": "aurora-cluster-1", "MonitoringInterval": 60, "PerformanceInsightsEnabled": true, "StatusInfos": [] } ] } ``` #### \*\*DescribeDBClusterMetrics (Performance Insights)\*\* Retrieves Performance Insights metrics for Aurora clusters. \*\*Request Structure:\*\* ```json { "ServiceType": "RDS", "Identifier": "aurora-cluster-1", "MetricQueries": [ { "Metric": "db.SQL.Innodb\_rows\_read.avg", "GroupBy": { "Group": "db.SQL\_TOKENIZED" } } ], "StartTime": "2025-09-26T00:00:00Z", "EndTime": "2025-09-26T23:59:59Z", "PeriodInSeconds": 3600 } ``` ### Aurora Metrics Reference | Metric Category | Common Metrics | Data Type | |-----------------|----------------|-----------| | Database Connections | DatabaseConnections | Count | | CPU | CPUUtilization, CPUCreditBalance | Percentage | | Memory | FreeableMemory, SwapUsage | Bytes | | Storage | FreeStorageSpace, VolumeBytesUsed | Bytes | | I/O | ReadIOPS, WriteIOPS, ReadLatency, WriteLatency | Count/Second, Milliseconds | --- ## DynamoDB APIs ### Table and Performance APIs #### \*\*DescribeTable\*\* Returns information about a DynamoDB table. \*\*Request Structure:\*\* ```json { "TableName": "my-table" } ``` \*\*Response Structure:\*\* ```json { "Table": { "TableName": "my-table", "TableStatus": "ACTIVE", "CreationDateTime": "2025-01-15T10:30:00Z", "ProvisionedThroughput": { "ReadCapacityUnits": 10, "WriteCapacityUnits": 10 }, "TableSizeBytes": 1024000, "ItemCount": 250, "TableArn": "arn:aws:dynamodb:us-east-1:123456789012:table/my-table", "BillingModeSummary": { "BillingMode": "PROVISIONED" }, "GlobalSecondaryIndexes": [ { "IndexName": "GSI1", "IndexStatus": "ACTIVE", "ProvisionedThroughput": { "ReadCapacityUnits": 5, "WriteCapacityUnits": 5 }, "IndexSizeBytes": 512000, "ItemCount": 100 } ] } } ``` #### \*\*DescribeTimeToLive\*\* Returns TTL settings for a DynamoDB table. \*\*Response Structure:\*\* ```json { "TimeToLiveDescription": { "TimeToLiveStatus": "ENABLED", "AttributeName": "ttl" } } ``` #### \*\*ListTables\*\* Lists all DynamoDB tables in the region. \*\*Response Structure:\*\* ```json { "TableNames": [ "table-1", "table-2", "table-3" ], "LastEvaluatedTableName": null } ``` #### \*\*DescribeContributorInsights\*\* Returns Contributor Insights status and rules. \*\*Response Structure:\*\* ```json { "TableName": "my-table", "ContributorInsightsStatus": "ENABLED", "ContributorInsightsRuleList": [ "DynamoDBContributorInsights-PKT-my-table", "DynamoDBContributorInsights-SKT-my-table" ] } ``` > 📌 \*\*Note on DynamoDB Metrics:\*\* > DynamoDB detailed metrics are primarily available through CloudWatch. Use the CloudWatch GetMetricData API with namespace "AWS/DynamoDB" for metrics like: > - ConsumedReadCapacityUnits / ConsumedWriteCapacityUnits > - UserErrors / SystemErrors > - ThrottledRequests > - SuccessfulRequestLatency --- ## ECS APIs ### Container and Service APIs #### \*\*DescribeClusters\*\* Describes one or more ECS clusters. \*\*Request Structure:\*\* ```json { "clusters": ["my-cluster"], "include": ["ATTACHMENTS", "SETTINGS", "STATISTICS", "TAGS"] } ``` \*\*Response Structure:\*\* ```json { "clusters": [ { "clusterArn": "arn:aws:ecs:us-east-1:123456789012:cluster/my-cluster", "clusterName": "my-cluster", "status": "ACTIVE", "registeredContainerInstancesCount": 3, "runningTasksCount": 10, "pendingTasksCount": 0, "activeServicesCount": 5, "statistics": [ { "name": "CPUUtilization", "value": "45.2" }, { "name": "MemoryUtilization", "value": "68.5" } ], "capacityProviders": ["FARGATE", "FARGATE\_SPOT"] } ] } ``` #### \*\*DescribeServices\*\* Describes specified services within an ECS cluster. \*\*Request Structure:\*\* ```json { "cluster": "my-cluster", "services": ["my-service"], "include": ["TAGS"] } ``` \*\*Response Structure:\*\* ```json { "services": [ { "serviceArn": "arn:aws:ecs:us-east-1:123456789012:service/my-cluster/my-service", "serviceName": "my-service", "clusterArn": "arn:aws:ecs:us-east-1:123456789012:cluster/my-cluster", "status": "ACTIVE", "desiredCount": 3, "runningCount": 3, "pendingCount": 0, "taskDefinition": "my-task:5", "deployments": [ { "id": "ecs-svc/123456789", "status": "PRIMARY", "taskDefinition": "my-task:5", "desiredCount": 3, "runningCount": 3, "createdAt": "2025-09-26T10:00:00Z" } ], "loadBalancers": [ { "targetGroupArn": "arn:aws:elasticloadbalancing:us-east-1:123456789012:targetgroup/my-targets/12345", "containerName": "web", "containerPort": 80 } ], "healthCheckGracePeriodSeconds": 60 } ] } ``` #### \*\*DescribeTasks\*\* Describes specified tasks or all tasks in a cluster. \*\*Request Structure:\*\* ```json { "cluster": "my-cluster", "tasks": ["arn:aws:ecs:us-east-1:123456789012:task/my-cluster/1234567890abcdef"] } ``` \*\*Response Structure:\*\* ```json { "tasks": [ { "taskArn": "arn:aws:ecs:us-east-1:123456789012:task/my-cluster/1234567890abcdef", "taskDefinitionArn": "arn:aws:ecs:us-east-1:123456789012:task-definition/my-task:5", "clusterArn": "arn:aws:ecs:us-east-1:123456789012:cluster/my-cluster", "lastStatus": "RUNNING", "desiredStatus": "RUNNING", "cpu": "256", "memory": "512", "containers": [ { "containerArn": "arn:aws:ecs:us-east-1:123456789012:container/abc123", "name": "web", "image": "nginx:latest", "lastStatus": "RUNNING", "networkInterfaces": [ { "privateIpv4Address": "10.0.1.100" } ], "cpu": "256", "memory": "512", "memoryReservation": "512" } ], "startedAt": "2025-09-26T10:30:00Z", "connectivity": "CONNECTED", "healthStatus": "HEALTHY" } ] } ``` #### \*\*DescribeContainerInstances\*\* Describes container instances for EC2-backed ECS. \*\*Response Structure:\*\* ```json { "containerInstances": [ { "containerInstanceArn": "arn:aws:ecs:us-east-1:123456789012:container-instance/my-cluster/abc123", "ec2InstanceId": "i-1234567890abcdef0", "status": "ACTIVE", "agentConnected": true, "runningTasksCount": 2, "pendingTasksCount": 0, "remainingResources": [ { "name": "CPU", "integerValue": 1024 }, { "name": "MEMORY", "integerValue": 3072 } ], "registeredResources": [ { "name": "CPU", "integerValue": 2048 }, { "name": "MEMORY", "integerValue": 8192 } ] } ] } ``` --- ## Implementation Guide ### 🛠️ Plugin Development Steps 1. Initialize plugin using `@grafana/create-plugin` 2. Install AWS SDK: `npm install aws-sdk` 3. Implement data source with AWS credential management 4. Create query editor component for API selection 5. Transform AWS responses to Grafana data frames ### Sample Plugin Code Structure ```typescript // datasource.ts import { DataSourceInstanceSettings, DataQueryRequest, DataQueryResponse } from '@grafana/data'; import AWS from 'aws-sdk'; export class AWSDataSource { private cloudWatch: AWS.CloudWatch; private ecs: AWS.ECS; private dynamoDB: AWS.DynamoDB; private rds: AWS.RDS; constructor(instanceSettings: DataSourceInstanceSettings) { const credentials = { accessKeyId: instanceSettings.jsonData.accessKey, secretAccessKey: instanceSettings.decryptedSecureJsonData.secretKey, region: instanceSettings.jsonData.region }; this.cloudWatch = new AWS.CloudWatch(credentials); this.ecs = new AWS.ECS(credentials); this.dynamoDB = new AWS.DynamoDB(credentials); this.rds = new AWS.RDS(credentials); } async query(options: DataQueryRequest): Promise { // Transform queries to AWS API calls // Convert responses to Grafana DataFrames } } ``` ### Data Frame Transformation Example ```typescript // Transform CloudWatch response to Grafana DataFrame function transformCloudWatchData(response: AWS.CloudWatch.GetMetricDataOutput): DataFrame { const fields: Field[] = [ { name: 'Time', type: FieldType.time, values: new ArrayVector(response.MetricDataResults[0].Timestamps) }, { name: response.MetricDataResults[0].Label || 'Value', type: FieldType.number, values: new ArrayVector(response.MetricDataResults[0].Values) } ]; return { name: 'CloudWatch Metrics', fields, length: response.MetricDataResults[0].Values.length }; } ``` --- ## Best Practices ### 🔒 Security - Use IAM roles when possible instead of access keys - Implement least privilege principle for API permissions - Store credentials securely using Grafana's encrypted storage - Rotate access keys regularly ### ⚡ Performance - Implement caching for frequently accessed metadata - Use batch APIs (GetMetricData) instead of multiple individual calls - Respect AWS API rate limits and implement exponential backoff - Consider using pagination for large result sets ### 🔍 Monitoring - Log API errors and latencies for debugging - Monitor plugin memory usage - Track API quota usage - Implement health checks for AWS connectivity --- ## ⚠️ Important Considerations - \*\*Rate Limiting:\*\* AWS APIs have rate limits. Implement proper throttling and retry logic. - \*\*Cost:\*\* Some APIs (especially GetMetricData) can incur costs. Monitor usage carefully. - \*\*Regional Endpoints:\*\* Ensure you're connecting to the correct AWS region for your resources. - \*\*Time Zones:\*\* AWS uses UTC for all timestamps. Handle timezone conversions appropriately. --- ## 📚 Additional Resources - [CloudWatch API Reference](https://docs.aws.amazon.com/AmazonCloudWatch/latest/APIReference/) - [RDS (Aurora) API Reference](https://docs.aws.amazon.com/AmazonRDS/latest/APIReference/) - [DynamoDB API Reference](https://docs.aws.amazon.com/amazondynamodb/latest/APIReference/) - [ECS API Reference](https://docs.aws.amazon.com/AmazonECS/latest/APIReference/) - [Grafana Plugin Development Guide](https://grafana.com/docs/grafana/latest/developers/plugins/)