Continue

The OpenTelemetry Collector can be configured to suit your observability needs. Before diving into Collector configuration, familiarize yourself with the location of the default configuration file, which is typically located in /etc//config.yaml. To provide custom configurations, you can use the --config option and specify one or more configurations using files at different paths. For instance: otelcol --config=customconfig.yaml You can also reference components from each other within multiple files. However, if the merger of files doesn't result in a complete configuration, you'll receive an error since required components aren't added by default. To validate your configuration file, use the validate command. The Collector's structure consists of four pipeline components that access telemetry data: receivers, processors, exporters, and connectors. After configuring each component, enable it using pipelines within the service section of the configuration file. Besides pipeline components, you can also configure extensions, which provide capabilities added to the Collector without direct access to telemetry data. The following example showcases a Collector configuration with a receiver, processor, exporter, and three extensions: receivers: otlp: protocols: grpc: endpoint: 0.0.0.0:4317 http: endpoint: 0.0.0.0:4318 processors: batch: exporters: otlp: endpoint: otelcol:4317 extensions: health_check: pprof: zpages: service: extensions: [health_check, pprof, zpages] pipelines: traces: receivers: [otlp] processors: [batch] exporters: [otlp] metrics: receivers: [otlp] processors: [batch] exporters: [otlp] logs: receivers: [otlp] processors: [batch] exporters: [otlp] Note that components are defined through a type/name format, such as otlp or otlp/2. You can define components of a given type more than once as long as the identifiers are unique. For example: receivers: otlp: protocols: grpc: endpoint: 0.0.0.0:4317 http: endpoint: 0.0.0.0:4318 otlp/2: protocols: grpc: endpoint: 0.0.0.0:55690 processors: batch: batch/test: exporters: otlp: endpoint: otelcol:4317 otlp/2: endpoint: otelcol2:4317 extensions: health_check: pprof: zpages: service: extensions: [health_check, pprof, zpages] pipelines: traces: receivers: [otlp] processors: [batch] exporters: [otlp] traces/2: receivers: [otlp/2] processors: [batch/test] exporters: [otlp/2] metrics: receivers: [otlp] processors: [batch] exporters: [otlp] logs: receivers: [otlp] processors: [batch] exporters: [otlp] The configuration can also include other files, causing the Collector to merge them in a single in-memory representation of the YAML configuration. receivers: otlp: protocols: grpc: endpoint: 0.0.0.0:4317 exporters: ${file:exporters.yaml} service: extensions: [] pipelines: traces: receivers: [otlp] processors: [] exporters: [otlp] With an exporters.yaml file being: otlp: endpoint: otelcol.observability.svc.cluster.local:443 The final result in memory will be: receivers: otlp: protocols: grpc: endpoint: 0.0.0.0:4317 exporters: otlp: endpoint: otelcol.observability.svc.cluster.local:443 service: extensions: [] pipelines: traces: receivers: [otlp] processors: [] exporters: [otlp] Receivers collect telemetry from one or more sources, which can be pull or push based and may support multiple data sources. Receivers are configured in the receivers section. Many receivers come with default settings, so specifying the name of the receiver is enough to configure it. If you need to configure a receiver or want to change the default configuration, you can do so in this section. Any setting you specify overrides the default values, if present. Configuring a receiver does not enable it; instead, receivers are enabled by adding them to the appropriate pipelines within the service section. The Collector requires one or more receivers. The following example shows various receivers in the same configuration file: receivers: # Data sources: logs fluentforward: endpoint: 0.0.0.0:8006 # Data sources: metrics hostmetrics: scrapers: cpu: disk: filesystem: load: memory: network: process: processes: paging: # Data sources: traces jaeger: protocols: grpc: endpoint: 0.0.0.0:4317 thrift_binary: thrift_compact: thrift_http: # Data sources: traces, metrics, logs kafka: protocol_version: 2.0.0 # Data sources: traces, metrics opencensus: # Data sources: traces, metrics, logs otlp: protocols: grpc: endpoint: 0.0.0.0:4317 tls: cert_file: cert.pem key_file: cert-key.pem http: endpoint: 0.0.0.0:4318 # Data sources: metrics prometheus: config: scrape_configs: - job_name: otel-collector scrape_interval: 5s static_configs: - targets: [localhost:8888] # Data sources: traces zipkin: For detailed receiver configuration, see the receiver README. Processors take the data collected by receivers and modify or transform it.Sending data to exporters happens according to rules defined in each processor's setting. Data processing includes filtering, dropping, renaming, and recalculating telemetry, among other operations. The order of processors determines the order of processing operations applied to the signal. Processors are optional but some recommended. Configuring a processor does not enable it. You need to add it to pipelines within service section to enable. For example, following is an exporters configuration file with several default processors in same config file. You can find full list of processors by combining opentelemetry-collector-contrib and opentelemetry-collector.processors. Extensions are enabled within the service section of the Collector configuration. The default is to not configure any extensions. The service section consists of three subsections: Extensions, Pipelines and Telemetry. Extensions is where you can set up custom authentication mechanisms for your receivers and exporters. You can use extensions like oidc, oauth2client to add server authenticators or client authenticators to your receivers and exporters respectively. Pipelines are used to configure what components are enabled in the Collector based on the configuration found in the receivers, processors, exporters, and extensions sections. Telemetry is where you can set up observability for the Collector itself. It consists of two subsections: logs and metrics. You can use environment variables to expand your configuration. For example, you can use ${env:DB_KEY} to access the value stored on the DB_KEY environment variable. override Collector settings using the --set option. The settings you define with this method are merged into the final configuration after all --config sources are resolved and merged. The following examples show how to override settings inside nested sections: otelcol --set "exporters::debug::verbosity=detailed" otelcol --set "exporters::debug::verbosity=detailed" The --set option doesn’t support setting a key that contains a dot or an equal sign. Next, reference the extension method in your Program.cs to and app.UseOpenTelemetryPrometheusScrapingEndpoint() set up OpenTelemetry when the application starts. Update your Program.cs with the following: using System.Diagnostics;using HelloOpenTelemetry.Extensions;using Microsoft.OpenApi.Models;var builder = WebApplication.CreateBuilder(args);...builder.AddCustomOpenTelemetry();var app = builder.Build();...app.UseOpenTelemetryPrometheusScrapingEndpoint();app.MapGet("/", () =>{ return "Hello World!";}).WithName("GetHelloWorld").WithOpenApi();app.Run(); Note: app.UseOpenTelemetryPrometheusScrapingEndpoint() sets up an endpoint in the app that exposes the metrics collected by OpenTelemetry in a format that Prometheus can understand. Simply put, it adds a URL (usually /metrics) to your app that Prometheus can regularly visit to grab performance data, like API response times, resource usage, and more. It makes it easy for Prometheus to monitor the app’s health and performance. Now we’re going to configure Prometheus to receive metrics from your .NET API and Grafana to display these metrics using the preconfigured dashboard (aspnetcore.json) that you’ve downloaded from here. Your updated project folder structure for Prometheus and Grafana configuration looks like this:Copy codedeploy/ grafana/ config/ provisioning/ dashboards/ default.yml datasources/ default.yml grafana.ini dashboards/ aspnetcore_endpoint.json aspnetcore.json prometheus/ prometheus.yml The prometheus.yml file inside the prometheus folder defines the scraping configuration for Prometheus. This configuration ensures that Prometheus can collect metrics from your .NET API. global: scrape_interval: 15s # Scrape every 15 secondsscrape_configs: - job_name: "prometheus" static_configs: - targets: ["localhost:9090"] - job_name: "helloopentelemetry" # Scraping our .NET API static_configs: - targets: ["helloopentelemetry:8080"] scrape_interval: Defines how frequently Prometheus will scrape metrics. Here it’s set to 15 seconds. targets: Specifies where Prometheus will fetch the metrics. For the API, it’s helloopentelemetry:8080, which is the internal Docker service. Grafana is configured to load metrics from Prometheus and display them on the dashboard aspnetcore.json. The configuration for Grafana is found in the provisioning folder, which includes two key files: default.yml for data sources and default.yml for dashboards. This file configures Prometheus as the data source for Grafana. apiVersion: 1datasources: - name: Prometheus # Data source name type: prometheus # Data source type: Prometheus access: proxy # Grafana accesses Prometheus via proxy url: # Prometheus URL inside Docker network isDefault: true # Sets Prometheus as the default data source url: This is the endpoint for Prometheus inside the Docker network (prometheus:9090), where Grafana will retrieve the metrics. isDefault: Marks Prometheus as the default data source for Grafana. This file tells Grafana where to find the preconfigured dashboards. Here, we reference the aspnetcore.json dashboard inside the dashboards folder. apiVersion: 1providers: - name: 'default' orgId: 1 folder: '.Net' type: file options: path: /var/lib/grafana/dashboards path: Specifies the directory inside the Grafana container where dashboards are stored. This will point to /var/lib/grafana/dashboards, which is mapped to your local folder deploy/grafana/dashboards/. In addition to the provisioning configuration, grafana.ini allows you to customize how Grafana operates. For example, you can configure default users, themes, and other ###ARTICLEEnsure that this file is correctly configured for your needs. Place the Dashboard Files: Ensure that the downloaded dashboard files aspnetcore.json and aspnetcore_endpoint.json are placed in the deploy/grafana/dashboards/ folder. Grafana Dashboard Auto-Loading: Grafana will automatically load these dashboards from the dashboards directory thanks to the configuration in provisioning/dashboards/default.yml. After downloading the aspnetcore.json dashboard file, I updated it to avoid the Grafana error: «Templating Failed to upgrade legacy queries». Specifically, I replaced "uid": "${DS_PROMETHEUS}" with a sample uid, so, like this: "uid": "4E7B5C8D9FA341A8B". This change addresses an issue where Grafana fails to interpret legacy data source queries properly, ensuring the dashboard loads without errors. By setting a specific UID, the problem with upgrading legacy queries is resolved, allowing the Prometheus data source to function correctly in Grafana. To enable telemetry monitoring, you need services like Prometheus and Grafana. Here is an example docker-compose.yml file to configure the required services: version: "3.4"services: helloopentelemetry: image: ${DOCKER_REGISTRY-}helloopentelemetry build: context: . dockerfile: src/HelloOpenTelemetry/Dockerfile ports: - "5000:8080" depends_on: - prometheus prometheus: image: prom/prometheus:latest ports: - "9090:9090" volumes: - ./deploy/prometheus/:/etc/prometheus/ grafana: image: grafana/grafana-oss:latest ports: - "3000:3000" volumes: - ./deploy/grafana/config/:/etc/grafana/ - ./deploy/grafana/dashboards/:/var/lib/grafana/dashboards/ depends_on: - prometheusnetworks: monitoring: driver: bridge You’ve now configured OpenTelemetry in your .NET API with Prometheus and Grafana for monitoring and tracing. This setup provides a robust way to visualize and trace your application’s performance in real-time.More details and code, here in my github. Happy coding! Publicado 21 septiembre 202422 septiembre 2024 You can’t perform that action at this time. You can apply or edit configuration settings in the following ways, with environment variables taking precedence over App.config or Web.config file:Environment variablesEnvironment variables are the main way to configure the settings.App.config or Web.config fileFor an application running on .NET Framework, you can use a web configuration file (web.config) or an application configuration file (app.config) to configure the OTEL_* settings.⚠ Only settings starting with OTEL_ can be set using App.config or Web.config. However, the following settings are not supported:OTEL_DOTNET_AUTO_HOMEOTEL_DOTNET_AUTO_EXCLUDE_PROCESSESOTEL_DOTNET_AUTO_FAIL_FAST_ENABLEDOTEL_DOTNET_AUTO_[TRACES|METRICS|LOGS]_INSTRUMENTATIONS_ENABLEDOTEL_DOTNET_AUTO_[TRACES|METRICS|LOGS]_{INSTRUMENTATION_ID}_INSTRUMENTATION_ENABLEDOTEL_LOG_LEVELOTEL_DOTNET_AUTO_NETFX_REDIRECT_ENABLEDExample with OTEL_SERVICE_NAME setting: Service name automatic detectionIf no service name is explicitly configured one will be generated for you. This can be helpful in some circumstances.If the application is hosted on IIS in .NET Framework this will be SiteName\VirtualPath ex: MySite\MyAppIf that is not the case it will use the name of the application entry Assembly.By default we recommend using environment variables for configuration. However, if given setting supports it, then:use Web.config for configuring an ASP.NET application (.NET Framework),use App.config for configuring a Windows Service (.NET Framework).Global settingsEnvironment variableDescriptionDefault valueStatusOTEL_DOTNET_AUTO_HOMEInstallation location.ExperimentalOTEL_DOTNET_AUTO_EXCLUDE_PROCESSESNames of the executable files that the profiler cannot instrument. Supports multiple comma-separated values, for example: ReservedProcess.exe,powershell.exe. If unset, the profiler attaches to all processes by default. [1][2]ExperimentalOTEL_DOTNET_AUTO_FAIL_FAST_DEnables possibility to fail process when automatic instrumentation cannot be executed. It is designed for debugging purposes. It should not be used in production environment. [1]falseExperimentalOTEL_DOTNET_AUTO_LOGGERAutoInstrumentation diagnostic logs sink. (supported values: none,file,console)fileExperimentalOTEL_LOG_LEVELSDK log level. (supported values: none,error,warn,info,debug)infoStable[1] If OTEL_DOTNET_AUTO_FAIL_FAST_ENABLED is set to true then processes excluded from instrumentation by OTEL_DOTNET_AUTO_EXCLUDE_PROCESSES will fail instead of silently continue.[2] Notice that applications launched via dotnet MyApp.dll have process name dotnet or dotnet.exe.ResourcesA resource is the immutable representation of the entity producing the telemetry. See Resource semantic conventions for more details.Resource attributesResource detectorsEnvironment variableDescriptionDefault valueStatusOTEL_DOTNET_AUTO_RESOURCE_DETECTOR_ENABLEDEnables all resource detectors.trueExperimentalOTEL_DOTNET_AUTO_{0}_RESOURCE_DETECTOR_ENABLEDConfiguration pattern for enabling a specific resource detector, where {0} is the uppercase ID of the resource detector you want to enable. Overrides OTEL_DOTNET_AUTO_RESOURCE_DETECTOR_ENABLED.trueExperimentalThe following resource detectors are included and enabled by default:PropagatorsPropagators allow applications to share context. See the OpenTelemetry specification for more details.Environment variableDescriptionDefault valueOTEL_PROPAGATORSComma-separated list of propagators. Supported options: tracecontext, baggage, b3multi, b3. See the OpenTelemetry specification for more details.tracecontext,baggageSamplersSamplers let you control potential noise and overhead introduced by OpenTelemetry instrumentation by selecting which traces you want to collect and export. See the OpenTelemetry specification for more details.Environment variableDescriptionDefault valueStatusOTEL_TRACES_SAMPLERSampler to be used for traces [1]parentbased_always_onStableOTEL_TRACES_SAMPLER_ARGString value to be used as the sampler argument [2]Stable[1]: Supported values are:always_on,always_off,traceidratio,parentbased_always_on,parentbased_always_off,parentbased_traceidratio.[2]: For traceidratio and parentbased_traceidratio samplers: Sampling probability, a number in the [0..1] range, e.g. “0.25”. ###ARTICLEThe OpenTelemetry .NET Automatic Instrumentation exporter output provides several configuration options for customizing its behavior. One of the key exporters is the OTLP traces exporter, which requires setting the OTEL_TRACES_EXPORTER environment variable to otlp. For the OTLP exporter, it's essential to specify additional settings such as the target endpoint for tracing, metrics, and logs. For example, the target endpoint can be set using the http/protobuf protocol to a URL like ` . Another crucial setting is the OTEL_EXPORTER_OTLP_PROTOCOLOptionally, it's possible to use grpc as the transport protocol for OTLP exporters. However, on .NET Framework, this protocol is not supported. It's also worth noting that there are various other exporters available for OTLP, such as the Prometheus exporter and the Zipkin exporter, which can be customized using environment variables or by setting specific configuration options. Environmental Variables and Configuration Options for .NET Automatic Instrumentation ###ENDARTICLEConfiguring an OpenTelemetry Collector for Windows You'll find several collector configuration files living under this directory – gateway_config used to configure collectors deployed in data forwarding (gateway) mode, otlp_config_linux for exporting OpenTelemtry traces to Splunk, and others designed specifically for AWS ECS tasks. For our application's instrumentation and collecting host and application metrics, we will focus on the agent_config.yaml Collector configuration file. Opening this config reveals it is comprised of several blocks: Extensions, Receivers, Processors, Exporters, and Service. The extensions block defines components that extend Collector capabilities, such as health monitoring, service discovery, and data forwarding. The Splunk Distribution of the OpenTelemetry Collector comes with a few default extensions like health_check, http_forwarder, smart_agent, zpages. ###ARTICLEThe configuration process for setting up an OpenTelemetry Collector is crucial to ensure seamless operation. After installation on Linux or Windows, users can locate configuration files under specific directories. The agent_config.yaml file, used with the Splunk Distribution of OpenTelemtry Collector, contains various blocks such as Extensions, Receivers, Processors, and Exporters. The Extensions block defines additional components that extend Collector capabilities, including health monitoring and data forwarding. Default extensions provided by the Splunk Distribution include health_check, http_forwarder, smart_agent, zpages, among others. The Receivers section configures data sources, with default receivers such as fluentforward, hostmetrics, jaeger, otlp, prometheus, smartagent, signalfx, and zipkin. Processors receive and transform telemetry data according to rules or settings, such as batch, memory_limiter, and resourcedetection. Exporters define destinations for telemetry data, including sapm (Splunk APM exporter), signalfx, splunk_hec (Splunk HEC endpoint), otlp, and debug. The Service block enables configured components within pipelines, defining telemetry metrics about the Collector itself. Common problems during configuration include incorrect indentation, receivers not enabled in a pipeline, or receivers enabled but not configured. OpenTelemetry is no longer an optional tool for monitoring applications; it's essential for gaining visibility into your system's performance and health. Just like bugs tend to hide in the dark, OpenTelemetry shines a light on what's happening in your application, providing a standardized way to collect and analyze telemetry data. Whether you're working with large microservices or monoliths, OpenTelemetry is crucial for understanding how your application interacts with external dependencies like databases, cache, APIs, and services. OpenTelemetry offers various benefits, particularly for Monoliths and Microservices. For instance, in Monoliths, it provides internal performance insights, helping you identify slowdowns caused by specific components. It also monitors database interactions and tracks calls to external APIs and services, making it easier to future-proof your infrastructure. Another essential aspect of OpenTelemetry is its ability to collect logs, metrics, and traces. Logs provide context about discrete events happening in your application at a specific point in time, while traces track the progression of requests as they travel through different parts of your system or across multiple services. Metrics offer numerical measurements collected over time, providing aggregated data about system behavior and performance. To implement OpenTelemetry effectively, you'll need to add it to your project. For microservices like ShippingService and OrderTrackingService, which communicate with each other, adding OpenTelemetry involves configuring tracing, instrumenting dependencies, and setting up exporters. By following these steps, you can gain valuable insights into your system's performance and health. paraphrased text here ###ENDARTICLETo implement OpenTelemetry, start with deploying the OpenTelemetry Operator and resources. Ensure you have version 0.56.1 or higher of the operator. Deploy the OpenTelemetry Operator and resources by applying the provided YAML configuration. The key components include the manager container image, collectorImage, opampBridgeImage, targetAllocatorImage, autoInstrumentationImage, featureGates, ports, and resources. Key resources to create include: - OpenTelemetry Collector API version v1alpha1 - Config mode: deployment with limited CPU and memory resources - Config receivers for otlp, zipkin, and loki exporters Update the configuration to include the exporter endpoint URL, and specify the `otlp` protocol for collector data. Also, create an exporter to Jaeger using the standard `otlp` export format. Ensure to configure logging, tracing, and instrumentation according to your needs with OpenTelemetry Instrumentation API version v1alpha1 and spec exporter endpoint URL. For efficient deployment, use pod annotations like `instrumentation.opentelemetry.io/inject-java: "true"` to enable Java instrumentation in Flowx services. rbac: create: truenamespaced: trueserviceAccount: create: truereplicas: 1 ingress: enabled: true ingressClassName: nginx path: / hosts: - {{ .Values.flowx.ingress.grafana }}} ## Enable persistence using Persistent Volume Claims ## ref: ## persistence: enabled: true accessModes: - ReadWriteOnce size: 5Gi finalizers: [] datasources: datasources.yaml: apiVersion: 1 datasources: - name: Loki uid: loki type: loki access: proxy url: timeout: 300 version: 1 jsonData: derivedFields: - datasourceUid: tempo # matches variations of "traceID" fields, either in JSON or logfmt # marian-fx: matcherRegex: '"traceid":"(\w+)"' matcherRegex: (?:[Tt]race[-_]?[Ii][Dd])[\\]?["]?[=:][ ]?[\\]?["]?(\w+) name: traceid # url will be interpreted as query for the datasource url: "$${__value.raw}" urlDisplayLabel: See traces - name: Prometheus uid: prometheus type: prometheus url: ' ' editable: true isDefault: true jsonData: exemplarTraceIdDestinations: - datasourceUid: tempo name: traceid - name: Tempo uid: tempo type: tempo access: proxy url: version: 1 jsonData: tracesToLogs: datasourceUid: loki lokiSearch: datasourceUid: loki nodeGraph: enabled: true serviceMap: datasourceUid: prometheus tracesToLogsV2: customQuery: false datasourceUid: loki filterBySpanID: true filterByTraceID: true spanEndTimeShift: 1s spanStartTimeShift: '-1s' tags: - key: service.name value: job dashboardProviders: dashboardproviders.yaml: apiVersion: 1 providers: - name: 'default' orgId: 1 folder: '' type: file disableDeletion: false editable: true options: path: /var/lib/grafana/dashboards/default resources: limits: memory: 150Mi grafana.ini: auth: disable_login_form: truenauth.anonymous: enabled: falorg_name: Main Org. org_role: Admin auth.generic_oauth: enabled: truename: Keycloak-OAuth allow_sign_up: true client_id: private-management-sa client_secret: xTs4yGYySrHaNDIpCiniHJUGqBKbyCtp scopes: openid email profile offline_access roles email_attribute_path: email login_attribute_path: username name_attribute_path: full_name auth_url: https://{{ .Values.flowx.keycloak.host }}/auth/realms/{{ .Values.flowx.keycloak.realm }}/protocol/openid-connect/auth token_url: https://{{ .Values.flowx.keycloak.host }}/auth/realms/{{ .Values.flowx.keycloak.realm }}/protocol/openid-connect/token api_url: https://{{ .Values.flowx.keycloak.host }}/auth/realms/{{ .Values.flowx.keycloak.realm }}/protocol/openid-connect/userinfo # role_attribute_path: contains(roles[*], 'admin') && 'Admin' || contains(roles[*], 'editor') && 'Editor' || 'Viewer' server: root_url: "https://{{ .Values.flowx.ingress.grafana }}/grafana" serve_from_sub_path: trueserver: root_url: "https://{{ .Values.flowx.ingress.grafana }}/grafana" serve_from_sub_path: trueprometheus: prometheus.yml: scrape_configs: - job_name: prometheus static_configs: - targets: - localhost:9090 - job_name: 'otel-collector' honor_labels: true kubernetes_sd_configs: - role: pod namespaces: own_namespace: true relabel_configs: - source_labels: [__meta_kubernetes_pod_annotation_opentelemetry_community_demo] action: keep regex: true ###ARTICLEresources: limits: cpu: 3 memory: 4Gi requests: cpu: 1 memory: 2Gi extraEnv: # Keep a little bit lower than memory limits - name: GOMEMLIMIT value: 3750MiB chunksCache: # default is 500MB, with limited memory keep this smaller writebackSizeLimit: 10MB # Enable minio for storage minio: enabled: true rootUser: enterprise-logs rootPassword: supersecret buckets: - name: chunks policy: none purge: false - name: ruler policy: none purge: false - name: admin policy: none purge: false persistence: size: 10Gi lokiCanary: enabled: false # Zero out replica counts of other deployment modes backend: replicas: 0 read: replicas: 0 write: replicas: 0 ingester: replicas: 0 querier: replicas: 0 queryFrontend: replicas: 0 queryScheduler: replicas: 0 distributor: replicas: 0 compactor: replicas: 0 indexGateway: replicas: 0 bloomCompactor: replicas: 0 bloomGateway: replicas: 0 Configure Tempo # tempo: # configure a 3 days retention by default retention: 72h # enable opentelemetry protocol & jaeger receivers # this configuration will listen on all ports and protocols that tempo is capable of. # the receives all come from the OpenTelemetry collector. more configuration information can # be found there: receivers: jaeger: protocols: grpc: endpoint: 0.0.0.0:14250 thrift_binary: endpoint: 0.0.0.0:6832 thrift_compact: endpoint: 0.0.0.0:6831 thrift_http: endpoint: 0.0.0.0:14268 otlp: protocols: grpc: endpoint: "0.0.0.0:4317" http: endpoint: "0.0.0.0:4318" persistence: enabled: true size: 10Gi # -- Pod Annotations podAnnotations: prometheus.io/port: prom-metrics prometheus.io/scrape: "true"

- http://ideatity.com/ckfinder/userfiles/files/d26c2402-f318-48ec-ba9c-cf8a71ace0cf.pdf
- genie pro excelerator remote programming
- hiwo
- the humanistic tradition book 6 7th edition pdf free download
- medical counter assistant job description
- kajigemi
- nuyulo
- https://grandhotelushba.com/userfiles/file/kezarigejag.pdf
- https://renhedc.com/uploads/files/202506180429593815.pdf
- security incident response plan
- letoyokiwi
- dadiwa
- how to begin a treadmill routine
- cocilaxefo