OneAgent Ruby Agent Documentation

Spis treści

1. Licensing

Our agent is licensed based on RAM consumption in similar way as Dynatrace OneAgent:

Max. RAM	License
1.6 GB	0.10
4 GB	0.25
8 GB	0.50
16 GB	1.0
32 GB	2.0
48 GB	3.0
64 GB	4.0
80 GB	5.0
96 GB	6.0
112 GB	7.0
nx16 GB	n

Single license cost is calculated on OneAgent Host Unit license and normally it's 25% of Agent Cost.

2. Installation

Supported version of ruby is 1.9 +. Right know jRuby is not supported. Ruby Agent is distributed as GEM file with native extension. This means that it must be installed in the way that will execute native code building.

First you have to add a reference to our library into your Gemfile:

```
gem 'onesdk_ruby'
```

Our gem is not available in any public repository, so you cannot just use bundle install command.

Easiest way is create directory in root of application:
`/your_app_dir/vendor/cache` (if it's not exist) and put your gemfile there. Then you can just execute:

```
bundle install --local
```

and gem should be installed without any issues.

Alternative way: You can use local repository if available in your environment. In such case you will be able to install gemfile without copying file.

You can install as well our gem from public repo with credentials sent via e-mail:

Installation for you will need few steps (it's not in docs because it's special for you):

1. You need authenticate in gemfury (our gem artifactory)
   `bundle config gem.fury.io KEY_RECEIVED_VIA_EMAIL` or
   `export BUNDLE_GEM__FURY__IO=KEY_RECEIVED_VIA_EMAIL`
2. Add in you Gemfile this definition (you need to have open traffic to this url):
```
source 'https://gem.fury.io/spicasolutions/' do
      gem 'onesdk_ruby', '~> 1.0.28'
end
```
3. `Execute bundle install`
4. `Run your application on Kubernetes cluster`

3. License

License has to be puted to onesdk.yaml file

```
license: LICENSEKEY
```

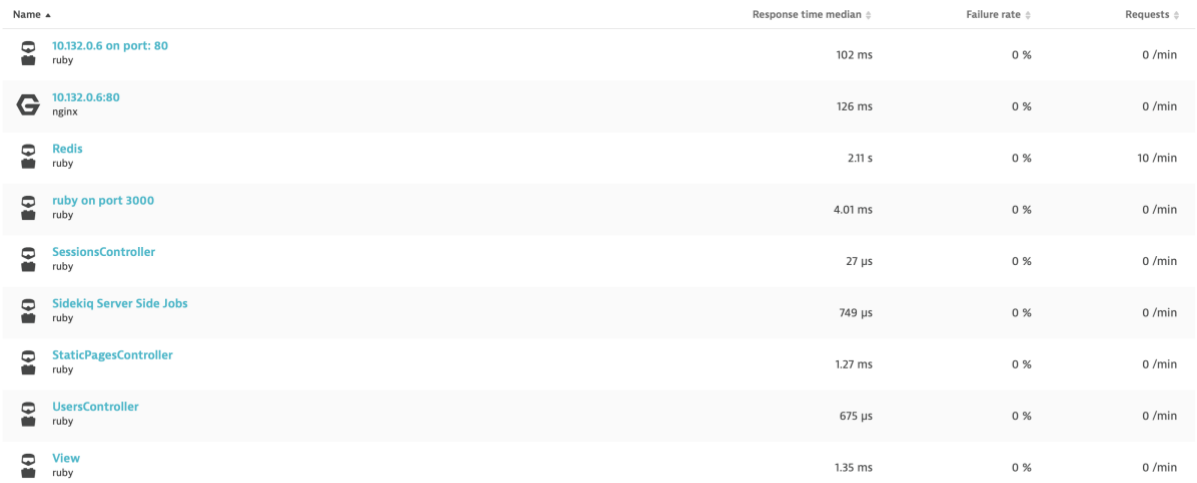
or via environment variable:

ONESDK_LICENSE=LICENSEKEY

4. Operating

4.1. Configuration

Our agent is made over OneAgent SDK. In such case you don't have to create any extra configuration. You should install OneAgent in Full Stack mode to run Ruby Agent properly and restart the application. After that and detection of traffic by OneAgent you should be able to see detected ruby's services in Dynatrace.



Name	Response time median	Failure rate	Requests
10.132.0.6 on port: 80 ruby	102 ms	0 %	0 /min
10.132.0.6:80 nginx	126 ms	0 %	0 /min
Redis ruby	2.11 s	0 %	10 /min
ruby on port 3000 ruby	4.01 ms	0 %	0 /min
SessionsController ruby	27 µs	0 %	0 /min
Sidekiq Server Side Jobs ruby	749 µs	0 %	0 /min
StaticPagesController ruby	1.27 ms	0 %	0 /min
UsersController ruby	675 µs	0 %	0 /min
View ruby	1.35 ms	0 %	0 /min

From this moment agent will be running with your application after each restart like regular OneAgent.

Our agent is instrumenting / enables visibility:

- Incomming WebRequests

- Controllers with actions
- Views
- Database Calls
- Redis Calls
- External WebRequests
- Exceptions
- Sidekiq Jobs

4.2. Debug logs

By default, debug logs are disabled. During troubleshooting you can enable those logs using flag in onesdk.yaml config file:

To start collecting debug logs with our agent you have to create **onesdk.yaml** file in config directory and put there such key:

debug: true

As alternative you can use environment variable **ONESDK_DEBUG=true**. Environment variables has higher priority than config file

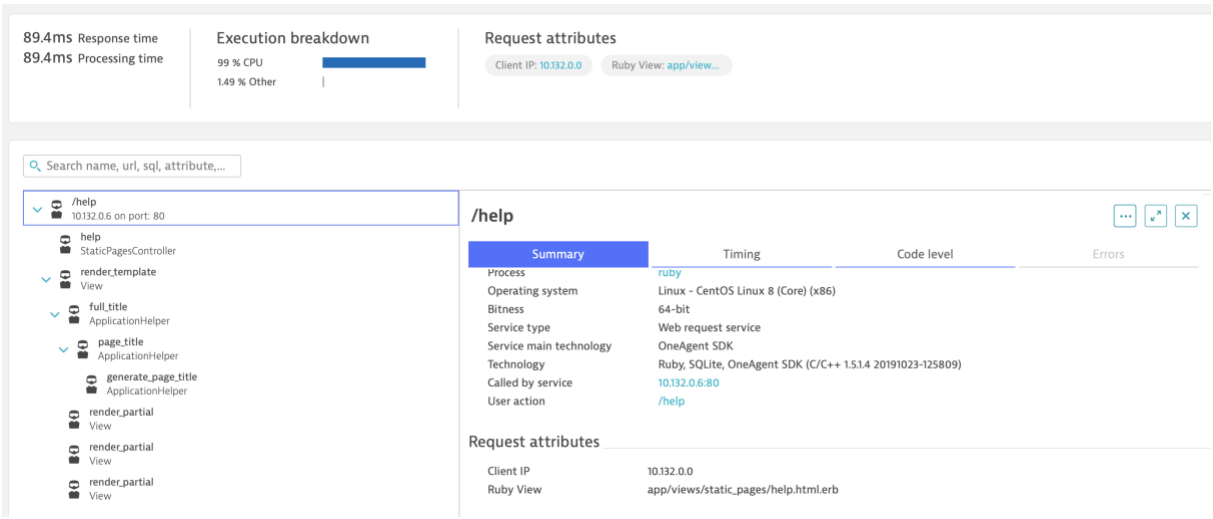
4.3. Disabling agent

Agent can be deactivated using environment variable **ONESDK_MONITORING_DISABLED=true** or via config file **monitoring_disabled: true**

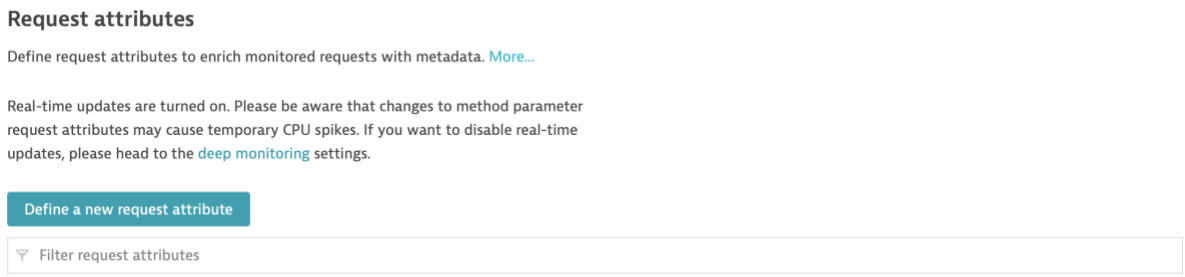
When application starts, this configuration will prevent app instrumentation process. Agent will not perform any changes in code, SDK will not be initialized as well. After switching value to false or remove flag, application restart is needed.

4.4. Collecting names of ruby Views as Request Attributes

To see names of views on PurePaths you must create request attribute in Dynatrace.

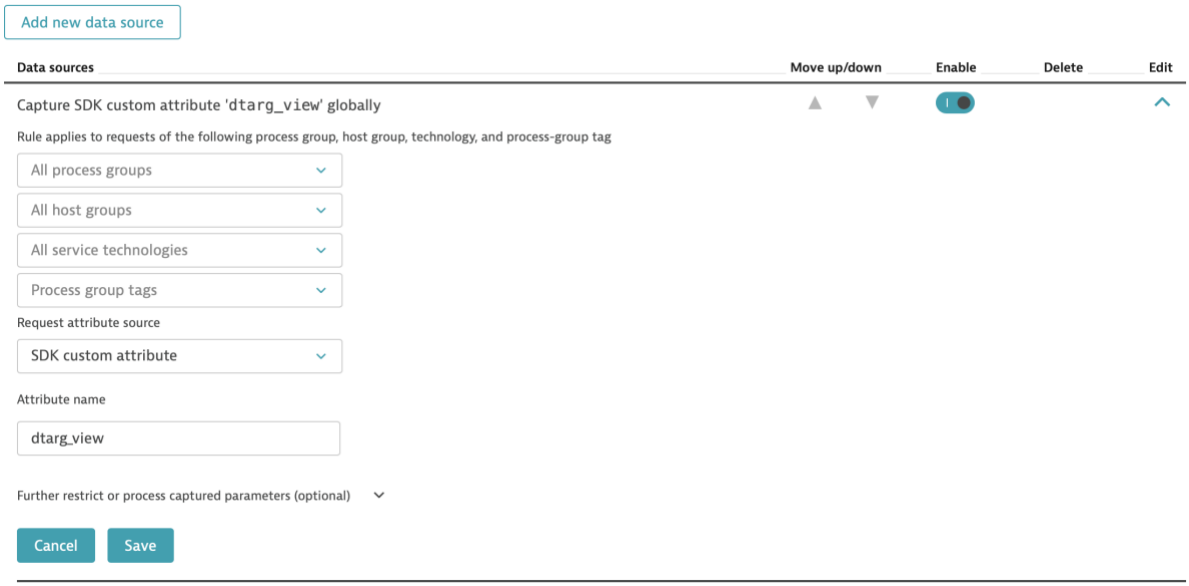


In such case you will be able to filter transactions using its value and see it in particular transactions.
To start collecting those values you have to go to Settings -> Server-side service monitoring -> Request attributes:

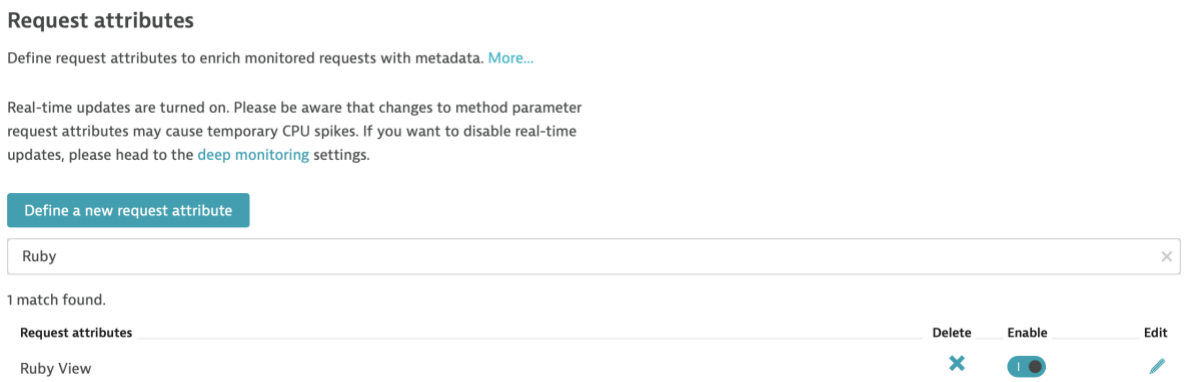


Click on “Define a new request attribute”:

Name it: Ruby View (or like you want) and add new data source:



Filled it like on screen above. Attribute name should be set as: **dtarg_view.** Click on “Save on data source” and “Request attribute” as well. As result you should be able to see your newly created request attribute in Dynatrace:



You don’t need restart your process if it’s already monitored with our agent.

4.5. Coexistence of other APM Agent

It’s highly not recommended to use our agent on the same application with other solutions like NewRelic, Instana, DataDog, ScoutAPM etc. The pattern of the instrumentation is similar in all of mentioned agents, so there may produce some issues on your application.

5. Custom metrics collection

Our gem is able to collect some extra metrics from GC and Threads. List of available metrics is listed below:

onesdk.ruby_count (GC count),
onesdk.ruby_heap_allocated_pages
onesdk.ruby_heap_sorted_length
onesdk.ruby_heap_allocatable_pages
onesdk.ruby_heap_available_slots
onesdk.ruby_heap_live_slots
onesdk.ruby_heap_free_slots
onesdk.ruby_heap_final_slots
onesdk.ruby_heap_marked_slots
onesdk.ruby_heap_eden_pages
onesdk.ruby_heap_tomb_pages
onesdk.ruby_total_allocated_pages
onesdk.ruby_total_freed_pages
onesdk.ruby_total_allocated_objects
onesdk.ruby_total_freed_objects
onesdk.ruby_malloc_increase_bytes
onesdk.ruby_malloc_increase_bytes
onesdk.ruby_minor_gc_count
onesdk.ruby_major_gc_count
onesdk.ruby_compact_count
onesdk.ruby_remembered_wb_unprotected_objects
onesdk.ruby_remembered_wb_unprotected_objects_limit
onesdk.ruby_old_objects
onesdk.ruby_old_objects_limit
onesdk.ruby_oldmalloc_increase_bytes
onesdk.ruby_oldmalloc_increase_bytes_limit
onesdk.gc_total_time (ms)
onesdk.ruby_thread_count

IMPORTANT:

To start collecting metrics you should enable OneAgent metric API according to this doc: https://www.dynatrace.com/support/help/shortlink/local-api#enable-the-oneagent-metric-api

To start collecting those metrics with our agent you have to create **onesdk.yaml** file in config directory and put there such keys:

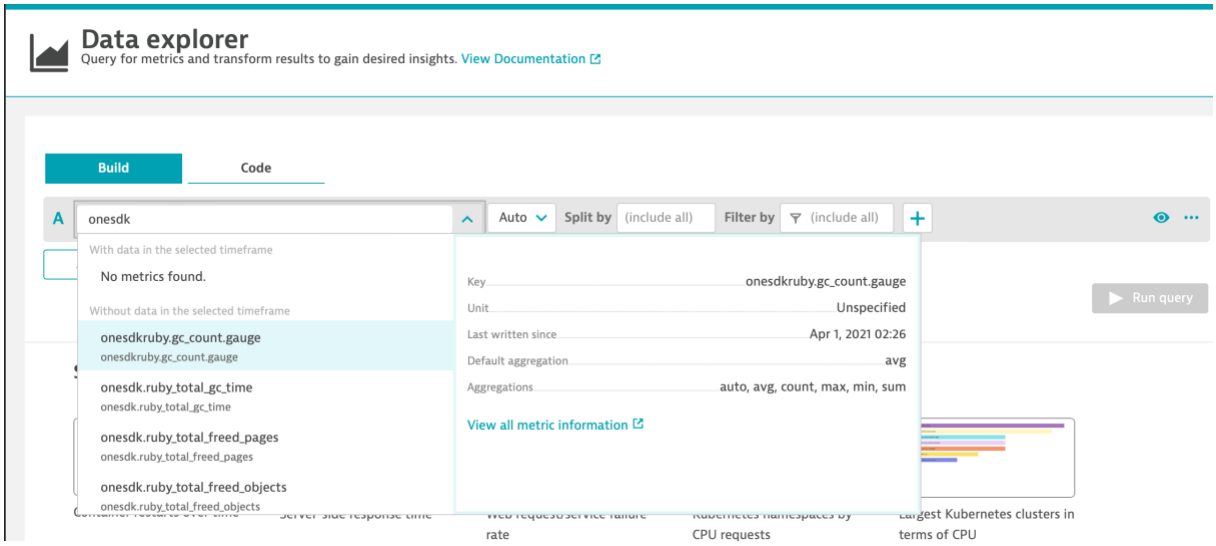
```
metrics_ingest: true
```

As alternative you can setup Environent Variable **ONESDK_METRICS_INGEST=true** which has higher priority than config file. You can switch metrics ingest with setting its value to false.

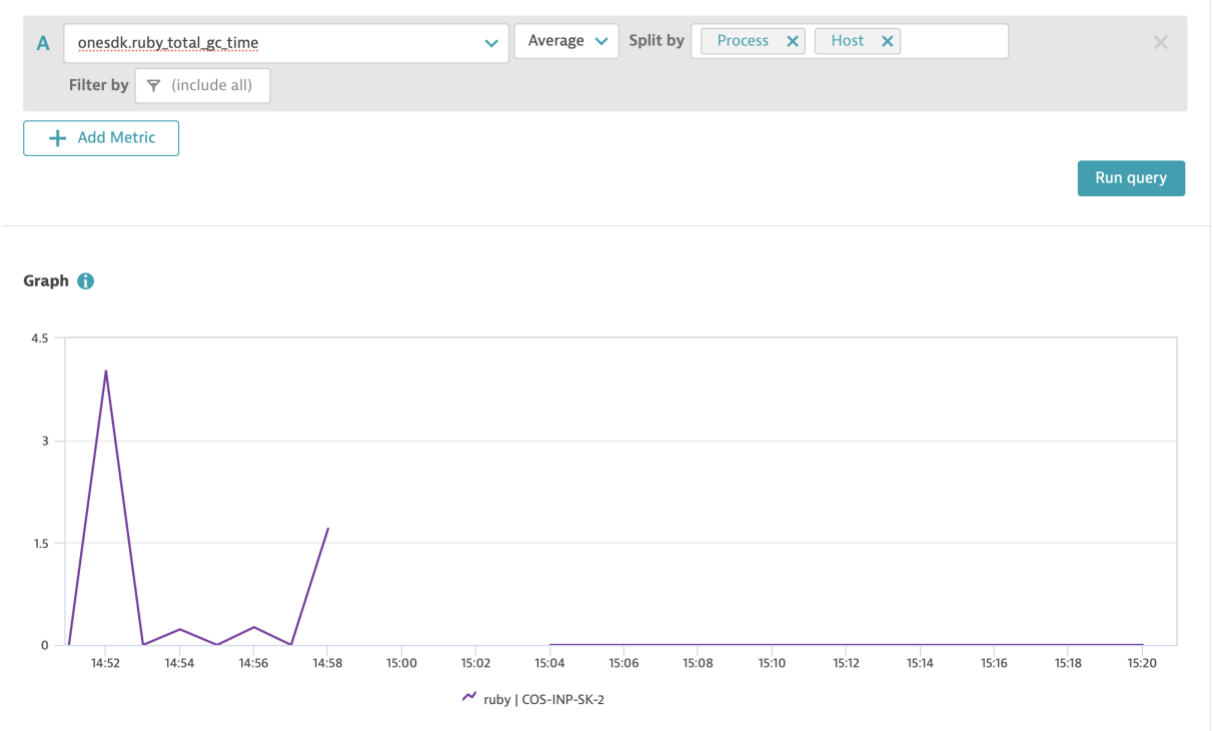
Metric_ingest key true value is enabling metrics collection.

Because of Dynatrace limitations, right now metrics will not show up in further details tab on process group instance dashboard. They will be assigned to proper process group despite that. We cannot setup units for those metrics automatically so all of them are generic ones. This behavior will change in the future. To access collected metrics, you must:

Create custom chart or enter Data Explorer:



All metrics will be available on key “onesdk.metric_name”.



You can split values by process group and host. Such metric can be pinned to your dashboard.

6. Custom instrumentation

It is always possible to instrument extra pieces of code if initial instrumentation is not enough. It may happen if your application is using multiple custom classes that should be exposed as Custom services.

6.1. Manual instrumentation

If agent is initialized and it is working properly on your application, you can always import our class and use it to instrument your code manually. Below you can find example:

First you have to import onesdk on top of your file:

```
require 'onesdk'
```

Then inside your method initialize tracer:

```
tracer = Onesdk.onesdk_customservicetracer_create(Onesdk.onesdk_asciistr(“Method name”),
Onesdk.onesdk_asciistr("Custom Service Name”))
```

Start the tracer

```
Onesdk.onesdk_tracer_start(tracer)
```

Then execute your method code.

If you want to report exception you can do this like that (before end of the tracer).

```
Onesdk.onesdk_tracer_error(tracer, Onesdk.onesdk_asciistr(“Exception Class Name”),
Onesdk.onesdk_asciistr(“Exception message”))
```

Exception can be reported only once per tracer, if you have several exceptions you have to combined them as single string.

Before method finish add line like below:

```
Onesdk.onesdk_tracer_end(tracer)
```

If you would like to report request attributes, add a below command before finishing tracer:

```
Onesdk.onesdk_customrequestattribute_add_string(Onesdk.onesdk_asciistr("KEY"),
Onesdk.onesdk_asciistr(“STRING_VALUE”));
```

To expose them in Dynatrace, you have to create proper Request Attribute configuration.

6.2. Instrumentation using config file

To use this feature you have to create yaml file in [app_dir]/config/onesdk.yaml

Sample content is:

```
custom_instrumentation: True/False (enables disables custom instrumentation)
methods:
  - class_module: string (Module::Class or Module or Class)
    method: string (method name)
```

Sample file content:

```
custom_instrumentation: True
methods:
  - class_module: User
    method: send_password_reset_email

  - class_module: User
    method: send_activation_email

  - class_module: User
    method: activate

  - class_module: User
    method: authenticated?

  - class_module: App::ApplicationHelper
```

```
    method: full_title
```

If instrumentation go fine, you will be able to see log entries related to particular method instrumentation. You can select multiple methods from single module/class

After changing config file, you have to restart ruby application to see effect.

6.3. Instrumentation environment variable

If you prefer using environment variable for config changes, you can setup instrumentation of custom methods in the same way as via vonfig file.
First you have to export environment variable that will enable this module:

```
export ONESDK_CUSTOM_INSTRUMENTATION=true
```

Second step is environment variable that will list Classes/Models and methods to be instrumented:

```
export
ONESDK_METHODS="App::ApplcationHelper|full_title\\User|send_password_reset_ema
il\\User|send_activation_email\\User|authenticated?"
```

For this environment variable syntax is

“ModuleOrClass|actionOrMethos\\AnotherModuleOrClass|anotherActionOrMethod”. You can chain multiple configurations like that.

After you add environment variable, you have to restart ruby application to see effect.

7. Sample full YAML file content

```
---
debug: true
custom_instrumentation: true
methods:
  -
    class_module: User
    method: send_password_reset_email
  -
    class_module: User
    method: send_activation_email
  -
    class_module: User
    method: activate
  -
    class_module: User
    method: authenticated?
  -
    class_module: ApplicationHelper
    method: full_title
  -
    class_module: ApplicationHelper
    method: page_title
  -
    class_module: ApplicationHelper
    method: generate_page_title
metrics_ingest: true
pgi_id: PROCESS_GROUP_INSTANCE-9BE796BA4609E25F
```

8. Sample Environment Variable Config

```
ONESDK_DEBUG=true/false (default false)
ONESDK_METRICS_INGEST=true/false (default false)
ONESDK_CUSTOM_INSTRUMENTATION=true/false (default false)
ONESDK_METHODS="App::ApplcationHelper|full_title\\User|send_password_reset_email\\User|send_activation_email\
\User|authenticated?" (default empty)
```

9. EarlyAccess

Early access features are not tested well but should be working fine. By default, those features are disabled. In future releases this behavior will be changed to enable by default.

9.1. Kafka instrumentation (tested on ruby 2.5+)

Kafka instrumentation can be turned on by setting up environment variable **ONESDK_KAFKA_INSTRUMENTATION=true** or in config yaml by setting **kafka_instrumentation: true**

9.1. RabbitMQ (Bunny) instrumentation (tested on ruby 1.9+)

Kafka instrumentation can be turned on by setting up environment variable **ONESDK_RABBITMQ_INSTRUMENTATION=true** or in config yaml by setting **bunny_instrumentation: true**