Prometheus as an OTel native metrics backend

Author: Goutham Veeramachaneni Jesús Vázquez
Date: Sep 7, 2023
Reviewers: *Juraci Paixão Kröhling* , *Matthias Loibl (MetalMatze)* *Jacob Aronoff* *David Ashpole* *… add yourself…*

OpenTelemetry is gaining more and more adoption and people are looking to instrument their applications with the OTel SDKs and use the OTel Collector. There is also a growing mandate in many companies to instrument applications with the OTel SDKs, sometimes over the Prometheus SDKs.

However, OpenTelemetry doesn't include a backend and users are trying to use Prometheus and related projects as the metrics backend. The current experience is quite subpar and it feels like Prometheus isn't a good fit for OTel. The OTel Prometheus WG made massive strides in improving the experience through the specification, but we can make it a lot easier by improving a few things in the core Prometheus project.

This doc lists out a few potential improvements we could make. This isn't an exhaustive list, I fully expect to run into more things that we cannot foresee.

Further, these are just suggestions that are *directionally correct*, and some of the changes require their own design doc. Please don't comment on exact implementation details if something seems off, but focus on "do we want this feature at all?".

Prerequisite reading: 2023-04 [PUBLIC] UX of using target_info
[Public] OTel to Prometheus Usage Issues

OTLP to Prometheus Specification:
https://github.com/open-telemetry/opentelemetry-specification/blob/main/specification/compatibility/prometheus_and_openmetrics.md#otlp-metric-points-to-prometheus

OTel Semantic Conventions:
https://github.com/open-telemetry/semantic-conventions/tree/main/docs

Proposed Changes

Prometheus server

Proper handling of resource attributes

Requirement: MUST

This is the main pain-point when using OTel with Prometheus. I've documented it here:
2023-04 [PUBLIC] UX of using target_info

Any query that requires a "target label" needs a join on `target_info`. The proposal in the doc above was to copy common resource attributes into the metrics. However, doing this today requires configuration of the Collector and I think the right place for it is in the Prometheus server.

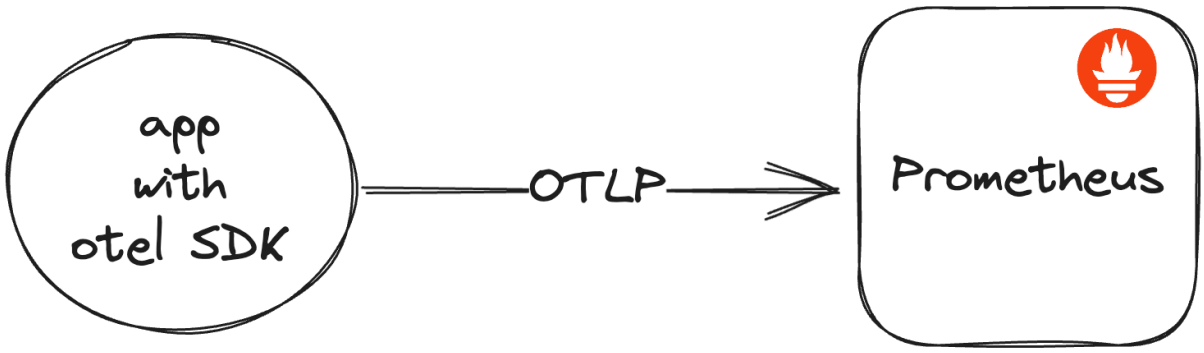
I would like to propose that we have the ability to **copy resource attributes into labels** in the Prometheus OTLP ingest path. I would also recommend adding a `resource_` prefix to these labels to disambiguate between resource attributes and metric attributes.

Open Questions:

- If we prefix the labels with `resource_`, then the labels in the metrics and the labels in `target_info` are different. Joins will still work, because we join on `job` and `instance`, but other things might break.
- Should we even add the `resource_` prefix?

We would need to handle this in both the ingestion paths:

When ingesting OTLP Push



A config that is similar to:

```
None
otlp:
  resource_to_metrics:
    relabel_rules:
      - <relabel_config>
    copy_attributes:
      with_prefix: true
      attributes: [...list of attributes...]
```

For example:

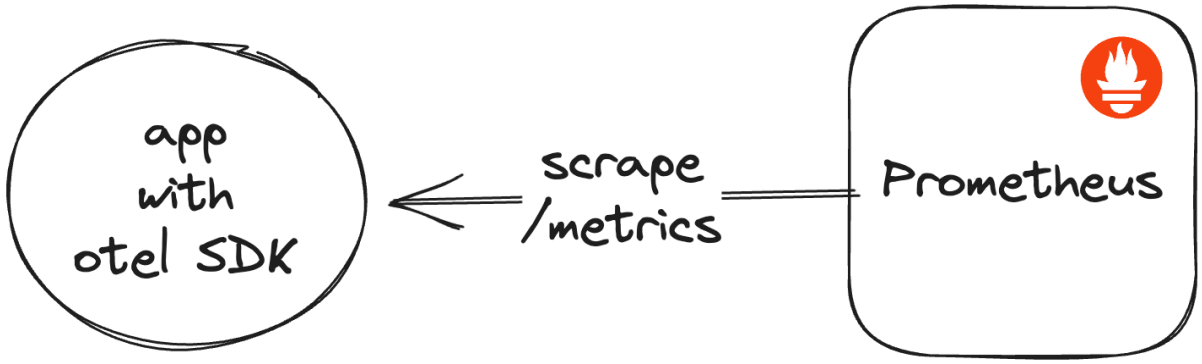
```
None
otlp:
  resource_to_metrics:
    copy_attributes:
      attributes:
        - k8s.cluster.name
        - k8s.namespace.name
        - cloud.availability_zone
```

Would produce the following labels:

```
None
...{resource_k8s_cluster_name="", resource_k8s_namespace_name="",
resource_cloud_availability_zone=""}
```

The label names are quite verbose but it would be staying consistent with the OTel semantic conventions and also helps distinguish between resource and metric attributes. We can work around the verbosity in the UI layer imo.

When scraping OTel SDK instrumented applications



Applications instrumented with the OTel SDK can also expose Prometheus metrics on `/metrics` that Prometheus can scrape. In this case, the resource attributes are put into a `target_info` metric. For example, the `/metrics` page looks something like:

```
None
....
# HELP target_info Target metadata
# TYPE target_info gauge
target_info{cloud_availability_zone="us-central1-a",cloud_provider="gcp",deploy
ment_environment="production",k8s_cluster_name="prod-us-central-0",k8s_namespac
e_name="ecommerce",k8s_pod_name="frontend-6b75f8d456-7j8dr",service_instance_id
="frontend-6b75f8d456-7j8dr",service_name="frontend",service_namespace="ecommer
ce"} 1
....
```

We need to handle this at scrape time and copy the labels from `target_info` into the metrics.

```
None
<scrape_config>:
  target_info_handling:
    resource_to_metrics:
      relabel_rules:
        - <relabel_config>
      copy_attributes:
        with_prefix: true
        attributes: [...list of attributes...]
```

One issue is that this would involve updating the labels of all metrics based on the labels of another metric in the scrape. This has performance implications and needs careful design.

Open Questions:

- How do we handle this in federation?
- Should we include *target labels* in *target_info*? See: https://github.com/prometheus/prometheus/issues/11362

Improve specification

The OTel to Prometheus specification isn't finalized yet and maybe we can solve the problems as part of the specification rather than in the Prometheus server. This could involve making it easy to copy over attributes as part of the SDK config.

Sane defaults

The approaches proposed above require users to always configure something to get to a usable state. There are currently 103 resource attributes documented. Maybe we can pick a few resource attributes that are copied over by default. See Appendix for an idea.

People can override this list but it gives them a usable solution out of the box.

Make resource attributes a first class citizen

A lot of this is a workaround for the concept of "resource attributes" not existing in Prometheus. As a Prometheus 3.0 (likely 4.0) change, we should consider including "resource attributes" as a first-class citizen.

Attribute compatibility

Requirement: MUST

OTLP SDKs allow users to configure `.` and `/` characters in the metric and label names which get converted to `_` when converting to Prometheus metrics. This is causing confusion as the metrics that users declare in the code, are not the same metrics that end up in Prometheus.

There are efforts underway to fix this:
https://github.com/ywwg/proposals/blob/utf8/proposals/2023-08-21-utf8.md

See: [Public] OTel to Prometheus Usage Issues

Out of Order metric handling

Requirement: MUST

The OTEL collector encourages batching up writes for various reasons such as optimized compression and less network overhead. This leads to a higher rate of out-of-order ingestion. Fortunately last year out-of-order for Prometheus was released and we can enable it today to make sure we don't lose any samples.

A window like 5-10 minutes will probably do it for you and this will probably not hurt any caching systems you may have built on top of Prometheus.

Out-of-order still remains experimental because some more polishing needs to happen on the TSDB. That said, we've been running it at scale at Grafana Labs with no issues.

I think this needs to be marked as "stable" and automatically turned on if we recognise OTLP Pushes.

Open Questions:

- What to do with recording rules and alerts? In Mimir, we currently evaluate with a 60s delay to make sure all the samples make it. Do we need to do the same here?

Up and *staleness* for PUSH

Requirement: Maybe

This comes down to the push vs pull debate. Prometheus is a Pull based system with the main advantages being staleness and service discovery (up == 0). This sets Prometheus apart and it's important to preserve these benefits even with OTLP Push.

OTLP Pushes happen on a periodic interval (60s by default). This means we can still configure service discovery, detect when a service is down and insert `up` and staleness markers. i.e, we can mark a service as down (`up` = 0) if we see that a Push has been missed two consecutive intervals. We can also achieve the same staleness handling we had before.

However, this is harder to do correctly because we need to configure both ends (sender and receiver), but it is possible. It is what Monarch does if I understand correctly.

Remote Write via OTLP

Requirement: Maybe

OTLP is about 50% more efficient than Prometheus remote write 1.0. A lot of this is due to gzip and as a result it takes more CPU. However, OTLP has an upcoming Arrow implementation that is more efficient and faster:

```
None
For univariate time series, OTel Arrow is 2 to 2.5 better in terms of bandwidth
reduction ... and the end-to-end speed is 3.1 to 11.2 times faster

For multivariate time series, OTel Arrow is 3 to 7 times better in terms of bandwidth
reduction ... Phase 2 has [not yet] been .. estimated but similar results are expected
```

We should evaluate if remote write v2 could be the Arrow based implementation of OTLP.

Separate from this, we should also consider exporting OTLP *in addition to* remote write: https://github.com/prometheus/prometheus/issues/12633

Delta compatibility

Requirement: Maybe

OpenTelemetry also has support for delta temporality where instead of pushing cumulative values, the applications will push “deltas”. We should evaluate if we want to support this in Prometheus or in a proxy layer in front.

See: https://github.com/prometheus/prometheus/issues/12763

Workarounds in Prometheus where there are issues with the SDKs

Requirement: Maybe

We had an issue some time ago where one of the SDKs sent labels with high cardinality and it wasn’t straightforward to disable this through the SDK’s configuration. We all know how high cardinality data can be challenging for Prometheus so this situation was not ideal.

The final solution to this problem was for the SDK to fix the cardinality issue but in the meantime we asked ourselves if there is something Prometheus could’ve done as a workaround. And I think this opens a very good question for us, we can’t simply blame SDKs and we might need to go the extra mile sometimes for our users.

For this particular case, the solution that was considered was to implement label filtering on ingestion, just like we do for scraping or remote writing. So in essence, to provide the user with more knobs in the configuration file.

SDKs

OpenTelemetry also has SDKs to instrument applications, but these SDKs are complicated. Prometheus SDKs can provide a simple and straightforward way to instrument applications while also integrating with the rest of the OTel ecosystem.

Easily plug into OTel SDK / auto-instrumentation for custom metrics

Our Java SDK is setting the gold standard for how to integrate and inter-operate with OTel: https://groups.google.com/g/prometheus-team/c/TuT0xlBPkB0/m/9KqZaxV_AAAJ

The new version of the Java Prometheus client lets users use the Prometheus API while also seamlessly integrating into the OTel SDK. For example, exemplars are added automatically and you can export the Prometheus metrics over `OTLP` protocol.

Expose `OTLP` alongside /metrics

This is similar to above, but I think it makes sense to add the ability to export `OTLP` metrics to all our SDKs.

Instrumentation wrappers for libraries with semantic conventions

The OTel SDKs come with a lot of easy instrumentation for the most common packages. I think Prometheus can also benefit from this approach. We can start a `client_golang-contrib` repo where we collect Prometheus instrumentations with the semantic conventions from OTel. We could also potentially look into reusing and contributing to the instrumentations from OTel.

Exporters

The OTel Collector also has a set of infrastructure “receivers” which play the same role as our exporters.

However, the metric names are completely different. This means there will be two different ways to monitor infrastructure components that will cause confusion and an ecosystem split.

Project	**OTel Receiver**	**Prometheus Exporter**
Aerospike	OTel	Prometheus --> from Aerospike
Apache Web Server	OTel	Prometheus

Apache Spark	OTel	Prometheus (native)
CouchDB	OTel	Prometheus
Docker	OTel	Prometheus (native)
Flink	OTel	Prometheus (native)
HAProxy	OTel	Prometheus (native)
Host / Node	OTel	Prometheus official
Kubernetes	OTel	Prometheus ~official
Memcached	OTel	Prometheus official
MySQL	OTel	Prometheus official
Nginx	OTel	Prometheus --> from nginx
OracleDB	OTel	Prometheus
Podman	OTel	Prometheus --> from podman
Pulsar	OTel	Prometheus native
cAdvisor	OTel	Prometheus

Semantic conventions

One way to manage this is to update the Prometheus exporters to emit the metrics based on semantic conventions, and then make them importable as “OTel Collector receivers”. This way the same code is reused by both ecosystems, and there is little confusion.

Mixins

Standard dashboards for common semantic conventions

We should also produce standard dashboards for the common semantic conventions. This will help users get out of the box monitoring when using Prometheus and will help with adoption.

Appendix

Resource attributes

List of all resource attributes

Reference: Resource Semantic Conventions 1.21.0

1. service.name
2. service.version
3. service.namespace
4. service.instance.id
5. telemetry.sdk.name
6. telemetry.sdk.language
7. telemetry.sdk.version
8. telemetry.auto.version
9. container.name
10. container.id
11. container.runtime
12. container.image.name
13. container.image.tag
14. container.image.id
15. container.command
16. container.command_line
17. container.command_args
18. faas.name
19. faas.version
20. faas.instance
21. faas.max_memory
22. process.pid
23. process.parent_pid
24. process.executable.name
25. process.executable.path
26. process.command
27. process.command_line
28. process.command_args
29. process.owner
30. process.runtime.name
31. process.runtime.version
32. process.runtime.description
33. webengine.name
34. webengine.version
35. webengine.description

36. host.id
37. host.name
38. host.type
39. host.arch
40. host.image.name
41. host.image.id
42. host.image.version
43. os.type
44. os.description
45. os.name
46. os.version
47. device.id
48. device.model.identifier
49. device.model.name
50. device.manufacturer
51. cloud.provider
52. cloud.account.id
53. cloud.region
54. cloud.resource_id
55. cloud.availability_zone
56. cloud.platform
57. deployment.environment
58. k8s.cluster.name
59. k8s.cluster.uid
60. k8s.node.name
61. k8s.node.uid
62. k8s.namespace.name
63. k8s.pod.uid
64. k8s.pod.name
65. k8s.container.name
66. k8s.container.restart_count
67. k8s.replicaset.uid
68. k8s.replicaset.name
69. k8s.deployment.uid
70. k8s.deployment.name
71. k8s.statefulset.uid
72. k8s.statefulset.name
73. k8s.daemonset.uid
74. k8s.daemonset.name
75. k8s.job.uid
76. k8s.job.name
77. k8s.cronjob.uid
78. k8s.cronjob.name
79. browser.brands

80. browser.platform
81. browser.mobile
82. browser.language
83. user_agent.original
84. webengine.name
85. webengine.version
86. webengine.description

Cloud-Provider-Specific Attributes

87. aws.log.group.names
88. aws.log.group.arns
89. aws.log.stream.names
90. aws.log.stream.arns
91. aws.ecs.cluster.arn
92. aws.ecs.launchtype
93. aws.ecs.task.arn
94. aws.ecs.task.family
95. aws.ecs.task.revision
96. aws.eks.cluster.arn
97. gcp.cloud_run.job.execution
98. gcp.cloud_run.job.task_index
99. gcp.gce.instance.name
100. gcp.gce.instance.hostname
101. heroku.release.creation_timestamp
102. heroku.release.commit
103. heroku.app.id

List of resource attributes to use as labels

Here is the list of resource attributes that make sense to use as indexed labels sorted in descending order by priority.

1. service.name (resource_service_name)
2. service.namespace (resource_service_namespace)
3. service.instance.id (resource_service_instance_id)
4. deployment.environment (resource_deployment_environment)
5. cloud.region (resource_cloud_region)
6. cloud.availability_zone (resource_cloud_availability_zone)
7. k8s.cluster.name (resource_k8s_cluster_name)
8. k8s.namespace.name (resource_k8s_cluster_namespace_name)
9. k8s.pod.name (resource_k8s_pod_name)
10. k8s.container.name (resource_k8s_container_name)
11. k8s.node.name (resource_k8s_node_name)

12. container.name (resource_container_name)
13. container.id (resource_container_id)
14. One of:
    a. k8s.replicaset.name
    b. k8s.deployment.name
    c. k8s.statefulset.name
    d. k8s.daemonset.name
    e. k8s.cronjob.name
    f. k8s.job.name

Right now only `service.name` has status stable, other resource attributes in experimental status, which means breaking changes are allowed