Если вы видите что-то необычное, просто сообщите мне.

?????????? ?

??????

- Анализ логов.
- Четыре типа метрик прометея
- Что такое логирование.

?????? ?????.

????????????? ?????

- доступа (access_log) — записывают IP-адрес, время запроса, другую информацию о пользователях;
- ошибок (error_log) — показывают файлы, в которых выявлены ошибки и классифицируют сбои;
- загрузки системы — с его помощью выполняется отладка при появлении проблем, в файл записываются основные системные события, включая сбои;
- основной — содержит информацию о действиях с файерволом, DNS-сервером, ядром системы, FTP-сервисом;
- баз данных — хранит подробности о запросах, сбоях, ошибки в логах сервера отображаются наравне с другой важной информацией;
- веб-сервера — содержит информацию о возникавших ошибках, обращениях;

Для чего необходим анализ логов.

- Анализ работы приложения
- Анализ работы инфраструктуры.

Journalctl - ??????????? ?????? ???????

Systemd

- systemd — менеджер системы и сервисов
- systemctl — утилита для просмотра и управление статусом сервисов
- systemd-analyze — предоставляет статистику по процессу загрузки системы, проверяет корректность unit-файлов и так же имеет возможности отладки systemd

Место где можно найти конфигурации сервисов, демонов `/etc/systemd/system/`

- Journald - системный демон журналов systemd. Systemd спроектирован так, чтобы централизованно управлять системными логами от процессов, приложений и т.д. Все такие события обрабатываются демоном journald, он собирает логи со всей системы и сохраняет их в бинарных файлах.

???????:

??????????? ???? ????? ???????

```
journalctl
```

`-e` - отматает в самый конец `-f` - следить за логами в реальном времени.

?????????? ?? ????????

```
journalctl -p 0
```

Уровни важности:

- 0: emergency (неработоспособность системы)
- 1: alerts (предупреждения, требующие немедленного вмешательства)
- 2: critical (критическое состояние)
- 3: errors (ошибки)
- 4: warning (предупреждения)
- 5: notice (уведомления)
- 6: info (информационные сообщения)
- 7: debug (отладочные сообщения)

?????????? ?? ?????? ???????

```
journalctl -b 0
```

- 0 - текущая загрузка

- -1 - прошлая загрузка

?????????? ????????? ????

```
journalctl -k
```

?????????? ????????? ????????????? ???????

```
journalctl -u NetworkManager.service
```

?????? ???? ?????? ????????

??????? ????????

First things first. There are four types of metrics collected by Prometheus as part of its exposition format:

- Счетчики
- Датчики
- Гистрограммы
- Сводка

Прометей ходит по HTTP точками доступа, которые выставляют метрики. Они могут быть естественно выставлены компонентом, который мониторится или может выставляться через один из сотни экспортеров прометея созданными сообществом. Прометей предоставляет клиентские библиотеки для различных языков программирования, которые вы можете использовать в своейм коде.

Модет сбора работает хорошо, когда мониторится K8s кластер, благодаря обнаружению сервисов и распределенной общей сети внутри кластера, но сложнее мониторить когда это динамический набор машин, AWS Fargate контейнеры или Lambda функции с Прометеем. Почему?

Сложно определить точки сбора метрик, и доступы к этим точкам могут быть ограниченны сетевой политикой. Чтобы решить некоторые из этих проблем, сообщество выпустило Prometheus Agent Mode в конце 2021 года, который собирает только метрики и посылает их в систему мониторинга используя удаленный протокол записи.

Прометей может собирать метрики в обоих видах Prometheus exposition и the OpenMetrics. В обоих случаях, метрики выставленны через HTTP и используют простые форматы основанные на текстах(в основном широко известные) или более эффективные и крепкие форматы буферизировнных протоколов. Одно большое преимущество текстовых форматов то, что они легко воспринимаются человеком, что значит вы можете открыть в браузере или

использвать инструменты типа curl, чтобы получить текущее состояние метрик.

Прометей использует очень простую модель с четырьмя типами метрик, которые поддерживаются в клиентской библиотеке. Все виды метрик отражены в формате выставления использующий один или набор простых типов данных лежащих уровнем ниже. Тип этих данных включает название метрики, набор маркировок, и значения с плавающей запятой. Отметка времени добавляется сервером мониторинга в момент сбора данных.

Каждый уникальный набор имени метрик и набор маркрировок определяет цепь с отметками времени и занченями с плавающей запятой.

Некоторые договоренности используются для отражения различных типов метрик.

Очень полезное свойство форматов Прометея в том, что он может объединять метаданные с метриками для определени их типа и предоставления описания. Например: Прометей делает эту инфу доступной, а Графана использует её, чтобы отражить контекст пользователю, которй помогает им выбрать правильные метрики и применить правильные функции PromQL

Исследование метрик в Графана отражает список прометейских метрик и показывает дополнительный контекс о них.

Пример формата метрик получаемых Прометеем:

```
# HELP http_requests_total Total number of http api requests
# TYPE http_requests_total counter
http_requests_total{api="add_product"} 4633433
```

> “ `# HELP` используется для предоставления описания метрик, а `# TYPE` тип метрики.

Теперь посмотрим подробнее на кажду из метрик Прометея в формате вывода.

????????

Счетчики используются для измерения которые только увеличиваются. Отсюда они всегда складывают их значения и могут только увеличиваться. Одно исключение: когда счетчик перезапущен, в этом случае значение сбрасывается в ноль.

На самом деле значение счетчика не очень ползено само по себе. Значение счетчика часто используется для вычисления разницы между двумя временными отметками или определения изменения во времени.

Например: типичный случай использования счетчика - измерение количества API вызовов, что является измерением которое увеличивается:

```
# HELP http_requests_total Total number of http api requests
# TYPE http_requests_total counter
http_requests_total{api="add_product"} 4633433
```

Имя метрики `http_request_total`, у метрики есть только 1 ярылк называется `api` со значением `add_product` и значение счетчика: 4633433. Это значит, что `add_product` API был взыван 4 633 433 раз с последнего запуска этой метрик. Счетчики обычно помечаются `_total` суфиксом.

Абсолютное число не дает нам какую-то информацию, но когда используется функция rate из PromQL (или любая другая), она помогает нам понять количество запросов в секунду получаемые API. PromQL запрос ниже вычисляет средние запросы в секунду за последние 5 минут.

```
rate(http_requests_total{api="add_product"}[5m])
```

Чтобы вычислить абсолютное изменение за временной период, мы будем использвать функцию которая называется `increase()` в PromQL:

```
increase(http_requests_total{api="add_product"}[5m])
```

Это вернет общее количетсво чисел запросов сделанных за последние 5 минут, и это будет то же что и This would return the total number of requests made in the last five minutes, and it would be the same as multiplying the per second rate by the number of seconds in the interval (five minutes in our case):

```
rate(http_requests_total{api="add_product"}[5m]) * 5 * 60
```

Other examples where you would want to use a counter metric would be to measure the number of orders in an e-commerce site, the number of bytes sent and received over a network interface or the number of errors in an application. If it is a metric that will always go up, use a counter.

Below is an example of how to create and increase a counter metric using the Prometheus client library for Python:

```
from prometheus_client import Counter
api_requests_counter = Counter(
                        'http_requests_total',
                        'Total number of http api requests',
                        ['api']
                       )
api_requests_counter.labels(api='add_product').inc()
```

Note that since counters can be reset to zero, you want to make sure that the backend you use to store and query your metrics will support that scenario and still provide accurate results in case of a counter restart. Prometheus and PromQL-compliant Prometheus remote storage systems like Promscale handle counter restarts correctly.

Gauges

Gauge metrics are used for measurements that can arbitrarily increase or decrease. This is the metric type you are likely more familiar with since the actual value with no additional processing is meaningful and they are often used. For example, metrics to measure temperature, CPU, and memory usage, or the size of a queue are gauges.

For example, to measure the memory usage in a host, we could use a gauge metric like:

```
# HELP node_memory_used_bytes Total memory used in the node in bytes
# TYPE node_memory_used_bytes gauge
```

node_memory_used_bytes{hostname="host1.domain.com"} 943348382

The metric above indicates that the memory used in node host1.domain.com at the time of the measurement is around 900 megabytes. The value of the metric is meaningful without any additional calculation because it tells us how much memory is being consumed on that node.

Unlike when using counters, rate and delta functions don't make sense with gauges. However, functions that compute the average, maximum, minimum, or percentiles for a specific series are often used with gauges. In Prometheus, the names of those functions are avg_over_time, max_over_time, min_over_time, and quantile_over_time. To compute the average of memory used on host1.domain.com in the last ten minutes, you could do this:

```
avg_over_time(node_memory_used_bytes{hostname="host1.domain.com"}[10m])
```

To create a gauge metric using the Prometheus client library for Python you would do something like this:

```
from prometheus_client import Gauge
memory_used = Gauge(
                'node_memory_used_bytes',
                'Total memory used in the node in bytes',
                ['hostname']
              )
memory_used.labels(hostname='host1.domain.com').set(943348382)
```

Histograms

Histogram metrics are useful to represent a distribution of measurements. They are often used to measure request duration or response size.

Histograms divide the entire range of measurements into a set of intervals—named buckets—and count how many measurements fall into each bucket.

A histogram metric includes a few items:

- A counter with the total number of measurements. The metric name uses the _count suffix.

- A counter with the sum of the values of all measurements. The metric name uses the _sum suffix.
- The histogram buckets are exposed as counters using the metric name with a _bucket suffix and a le label indicating the bucket upper inclusive bound. Buckets in Prometheus are inclusive, that is a bucket with an upper bound of N (i.e., le label) includes all data points with a value less than or equal to N. For example, the summary metric to measure the response time of the instance of the add_product API endpoint running on host1.domain.com could be represented as:

```
# HELP http_request_duration_seconds Api requests response time in seconds
# TYPE http_request_duration_seconds histogram
http_request_duration_seconds_sum{api="add_product" instance="host1.domain.com"} 8953.332
http_request_duration_seconds_count{api="add_product" instance="host1.domain.com"} 27892
http_request_duration_seconds_bucket{api="add_product" instance="host1.domain.com" le="0"}
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com",
le="0.01"} 0
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com",
le="0.025"} 8
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com",
le="0.05"} 1672
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com", le="0.1"}
8954
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com",
le="0.25"} 14251
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com", le="0.5"}
24101
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com", le="1"}
26351
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com", le="2.5"}
27534
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com", le="5"}
27814
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com", le="10"}
27881
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com", le="25"}
27890
http_request_duration_seconds_bucket{api="add_product", instance="host1.domain.com",
le="+Inf"} 27892
```

The example above includes the sum, the count, and 12 buckets. The sum and count can be used to compute the average of a measurement over time. In PromQL, the average duration for the last five minutes will be computed as follows:

```
rate(http_request_duration_seconds_sum{api="add_product", instance="host1.domain.com"}[5m]) /
rate(http_request_duration_seconds_count{api="add_product", instance="host1.domain.com"}[5m])
```

It can also be used to compute averages across series. The following PromQL query would compute the average request duration in the last five minutes across all APIs and instances:

```
sum(rate(http_request_duration_seconds_sum[5m])) /
sum(rate(http_request_duration_seconds_count[5m]))
```

With histograms, you can compute percentiles at query time for individual series as well as across series. In PromQL, we would use the histogram_quantile function. Prometheus uses quantiles instead of percentiles. They are essentially the same thing but quantiles are represented on a scale of 0 to 1 while percentiles are represented on a scale of 0 to 100. To compute the 99th percentile (0.99 quantile) of response time for the add_product API running on host1.domain.com, you would use the following query:

```
histogram_quantile(0.99, rate(http_request_duration_seconds_bucket{api="add_product",
instance="host1.domain.com"}[5m]))
```

One big advantage of histograms is that they can be aggregated. The following query returns the 99th percentile of response time across all APIs and instances:

```
histogram_quantile(0.99, sum by (le) (rate(http_request_duration_seconds_bucket[5m])))
```

In cloud-native environments, where there are typically many instances of the same component running, the ability to aggregate data across instances is key.

Histograms have three main drawbacks:

- First, buckets must be predefined, requiring some upfront design. If your buckets are not well defined, you may not be able to compute the percentiles you need or would consume unnecessary resources. For example, if you have an API that always takes more than one second, having buckets with an upper bound (le label) smaller than one second would be

useless and just consume compute and storage resources on your monitoring backend. On the other hand, if 99.9 % of your API requests take less than 50 milliseconds, having an initial bucket with an upper bound of 100 milliseconds will not allow you to accurately measure the performance of the API.

- Second, they provide approximate percentiles, not accurate percentiles. This is usually fine as long as your buckets are designed to provide results with reasonable accuracy.
- And third, since percentiles need to be calculated server-side, they can be very expensive to compute when there is a lot of data to be processed. One way to mitigate this in Prometheus is to use recording rules to precompute the required percentiles. The following example shows how you can create a histogram metric with custom buckets using the Prometheus client library for Python:

```
from prometheus_client import Histogram
api_request_duration = Histogram(
                        name='http_request_duration_seconds',
                        documentation='Api requests response time in seconds',
                        labelnames=['api', 'instance'],
                        buckets=(0.01, 0.025, 0.05, 0.1, 0.25, 0.5, 1, 2.5, 5, 10, 25 )
                       )
api_request_duration.labels(
    api='add_product',
    instance='host1.domain.com'
).observe(0.3672)
```

Summaries

Like histograms, summary metrics are useful to measure request duration and response sizes.

A summary metric includes these items:

- A counter with the total number of measurements. The metric name uses the _count suffix.
- A counter with the sum of the values of all measurements. The metric name uses the _sum suffix. Optionally, a number of quantiles of measurements exposed as a gauge using the metric name with a quantile label. Since you don’t want those quantiles to be

measured from the entire time an application has been running, Prometheus client libraries use streamed quantiles that are computed over a sliding time window (which is usually configurable). For example, the summary metric to measure the response time of the instance of the add_product API endpoint running on host1.domain.com could be represented as:

```
# HELP http_request_duration_seconds Api requests response time in seconds
# TYPE http_request_duration_seconds summary
http_request_duration_seconds_sum{api="add_product" instance="host1.domain.com"} 8953.332
http_request_duration_seconds_count{api="add_product" instance="host1.domain.com"} 27892
http_request_duration_seconds{api="add_product" instance="host1.domain.com" quantile="0"}
http_request_duration_seconds{api="add_product" instance="host1.domain.com" quantile="0.5"}
0.232227334
http_request_duration_seconds{api="add_product" instance="host1.domain.com" quantile="0.90"}
0.821139321
http_request_duration_seconds{api="add_product" instance="host1.domain.com" quantile="0.95"}
1.528948804
http_request_duration_seconds{api="add_product" instance="host1.domain.com" quantile="0.99"}
2.829188272
http_request_duration_seconds{api="add_product" instance="host1.domain.com" quantile="1"}
34.283829292
```

This example above includes the sum and count as well as five quantiles. Quantile 0 is equivalent to the minimum value and quantile 1 is equivalent to the maximum value. Quantile 0.5 is the median and quantiles 0.90, 0.95, and 0.99 correspond to the 90th, 95th, and 99th percentile of the response time for the add_product API endpoint running on host1.domain.com.

Like histograms, summaries include sum and count that can be used to compute the average of a measurement over time and across time series.

Summaries provide more accurate quantiles than histograms but those quantiles have three main drawbacks:

- First, computing the quantiles is expensive on the client-side. This is because the client library must keep a sorted list of data points overtime to make this calculation. The implementation in the Prometheus client libraries uses techniques that limit the number of data points that must be kept and sorted, which reduces accuracy in exchange for an increase in efficiency. Note that not all Prometheus client libraries support quantiles in

summary metrics. For example, the Python library does not have support for it.

- Second, the quantiles you want to query must be predefined by the client. Only the quantiles for which there is a metric already provided can be returned by queries. There is no way to calculate other quantiles at query time. Adding a new quantile requires modifying the code and the metric will be available from that time forward.
- And third and most important, it’s impossible to aggregate summaries across multiple series, making them useless for most use cases in dynamic modern systems where you are interested in the view across all instances of a given component. Therefore, imagine that in our example the add_product API endpoint was running on ten hosts sitting behind a load balancer. There is no aggregation function that we could use to compute the 99th percentile of the response time of the add_product API endpoint across all requests regardless of which host they hit. We could only see the 99th percentile for each individual host. Same thing if instead of the 99th percentile of the response time for the add_product API endpoint we wanted to get the 99th percentile of the response time across all API requests regardless of which endpoint they hit. The code below creates a summary metric using the Prometheus client library for Python:

```
from prometheus_client import Summary
api_request_duration = Summary(
                        'http_request_duration_seconds',
                        'Api requests response time in seconds',
                        ['api', 'instance']
                       )
api_request_duration.labels(api='add_product', instance='host1.domain.com').observe(0.3672)
```

The code above does not define any quantile and would only produce sum and count metrics. The Prometheus client library for Python does not have support for quantiles in summary metrics.

??? ????? ???????????.

Известно, что программисты проводят много времени, отлаживая свои программы, пытаясь разобраться, почему они не работают — или работают неправильно. Когда говорят про отладку, обычно подразумевают либо отладочную печать, либо использование специальных программ – дебагеров. С их помощью отслеживается выполнение кода по шагам, во время которого видно, как меняется содержимое переменных. Эти способы хорошо работают в небольших программах, но в реальных приложениях быстро становятся неэффективными.

??????????

И для всего этого многообразия систем существует единое решение — логирование. В простейшем случае логирование сводится к файлу на диске, куда разные программы записывают (логируют) свои действия во время работы. Такой файл называют логом или журналом. Как правило, внутри лога одна строчка соответствует одному действию.

?????? ???????????

- debug
- info
- warning
- error

??????? ?????

Со временем количество логов становится большим, и с ними нужно что-то делать. Для этого используется ротация логов. Иногда за это отвечает сама программа, но чаще — внешнее приложение, задачей которого является чистка. Эта программа по необходимости разбивает логи на более мелкие файлы, сжимает, перемещает и, если нужно, удаляет. Подобная система встроена в любую операционную систему для работы с логами самой

системы и внешних программ, которые могут встраиваться в нее.

С веб-сайтами все еще сложнее. Даже на небольших проектах используется несколько серверов, на каждом из которых свои логи. А в крупных проектах тысячи серверов. Для управления такими системы созданы специализированные программы, которые следят за логами на всех машинах, скачивают их, складывают в заточенные под логи базы данных и предоставляют удобный способ поиска по ним.