Metrics monitoring and alerting system

Eg Datadog, grafana

Provide clear visibility to infra health to ensure high availability and reliability

Step 1: problem and design scope

Qu: is it in-house system for large company like fb, google or for internal use eg data dog, splunk?
Ans: internal use only

Qu: metric to collect?
Ans: operational system metric (eg CPU load, memory use, disk space consumption, service request per sec, running server count of web pool.

Qu: infra scale:
Ans: 100 mn daily active user, 1000 server pool, 100 machines per pool

qu: data retention
and: 1 year

qu: want to reduce resolution for long term storage
and: yes, new data for 7 days, 1 min resolution for 30 dats, 1 hour resolution after that

qu: alert channel
ans: email, phone, pagerduty, webhook

qu: collect log, error or access log
ans: no

qu: need to support distributed system tracing
ans: no

Requirements and assumptions

functional

100 mn daily active user
1000 server pool, 100 machine per pool, 100 metric per machine -> ~10 million metric
1 year data retention with 7 days, 30 days (1 min), 1 year(1hr) resolution
metric - CPU usage, request count, memory usage, message count in msg queue

non function

- scalable
- low latency
- reliability to avoid missing critical alerts
- flexible

out of scope requirement

- log monitoring: elastic search, log stash, cabana
- distributed system trace

Step2: HLD

5 component - data collection, data transmission, data storage, alert to different communication channel, visualization via graph, chart etc.

data model: recorded as time series

Example 1: What is the CPU load on production server instance i631 at 20:00?

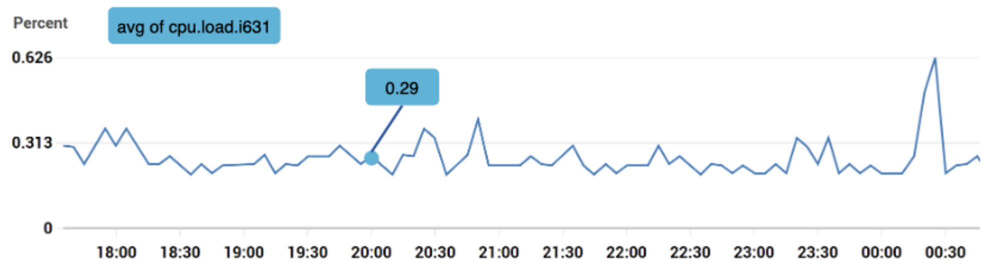

Figure 3 CPU load

data access pattern

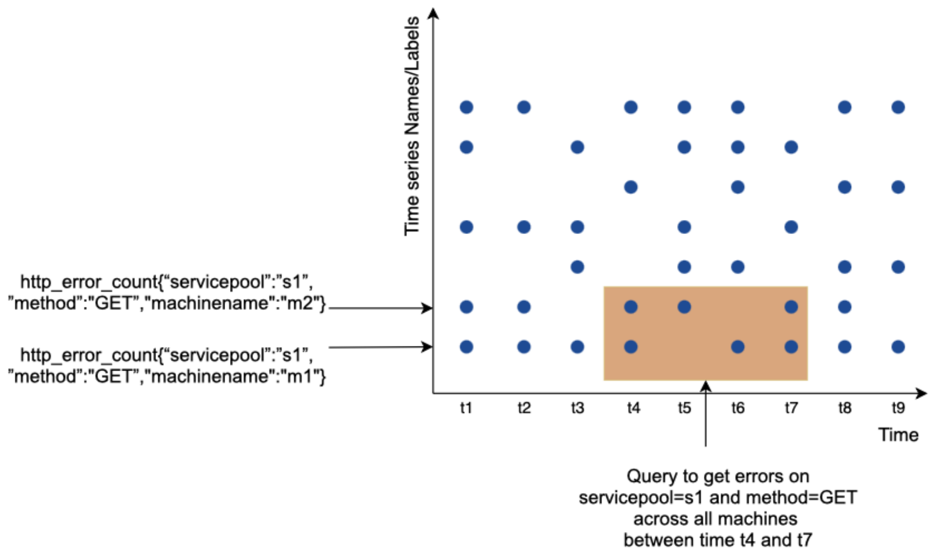
Time series Names/Labels
http_error_count{"servicepool":"s1", "method":"GET","machinename":"m2"}
http_error_count{"servicepool":"s1", "method":"GET","machinename":"m1"}
t1
t2
t3
t4
t5
t6
t7
t8
t9
Time
Query to get errors on servicepool=s1 and method=GET across all machines between time t4 and t7

Figure 4 Data access pattern

NOTE: system is under constant heavy write load, while read load is spiky

data storage system

options not good for time series data -

relational db is not optimized: eg to compute moving average in rolling time window, or support tagging/labeling of data, we need to add index to each tag, also not good for constant heavy write load

noSQL - few could handle eg Cassandra, Bigtable - but req deep internal working knowledge to device scalable system, not good for industrial scale.

preferred options

OpenTSDB - distributed time series db (based on Hadoop, HBase), thus add complexity, Twitter use MetricsDB, Amazon has timestream

<u>most popular</u> - influxDB and Prometheus - for real time analysis, for large data, primarily rely on in memory cache and on-disk storage

InfluxDB build index on labels - for fast lookup of time series. Make sure each label has low cardinality

has custom query interface to analyze and summarize large amount of time series data by labels.

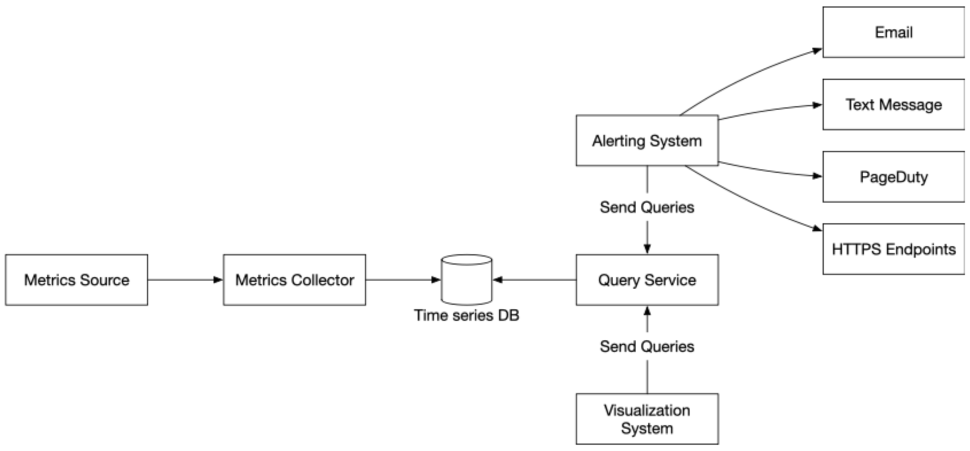

Figure 6 High-level design

Step 3: design deep dive

1. Metric collection -

pull model:

zookeeper or etc use service discover to get list of all the source (web server, db cluster, queue cluster, cache cluster)

metadata - pulling interval, IP address, timeout and retry parameters etc.

this pull metric data from predefined HTTP endpoint.

for large scale, we must use pool of metric collector, but this might lead to duplicate data - use some coordination scheme to avoid this. Eg designate each collector to a range of hash ring and map every server being monitored by unique name in hash ring

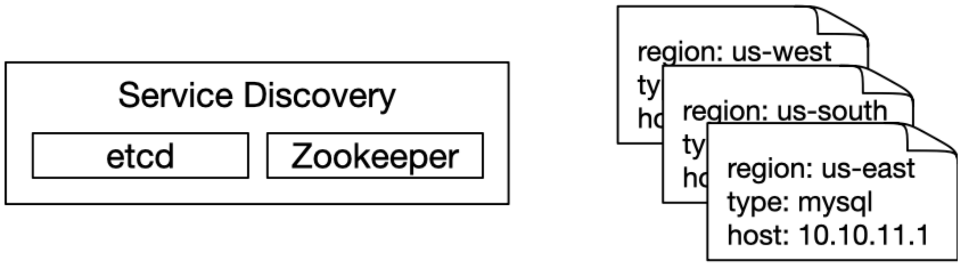

Figure 9 Service discovery

Figure 10 explains the pull model in detail.

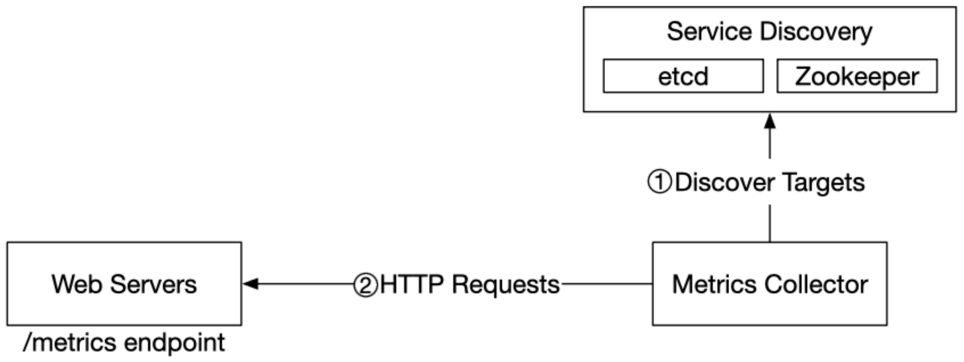

Figure 10 Pull model in detail

push model

collection agent commonly installed on every server being monitored - push metric periodically to metric collector.

can use aggregation to reduce data volume

- for high traffic, keep small buffer of data locally and resend them later.
- but for autoscaling group with data rotated frequently, data loss might happen, so here use load balancer based on which cluster will scale up and down

when to use what?

Pull	push
Easy debug, can view metric anytime.	If metric collector not receive metric, problem might be caused by network issues.

Easy health check (if server not respond to pull -> server is down)	If metric doesn't receive metric, problem might caused by network issues
Not good for short lived jobs	Good for short lived batch jobs
Complicated network setup with multiple data center	If metric collector setup with load balancer, it is possible to receive data rom anywhere.
Performance - use TCP	Use UDP (low latency, but high effort)

Scale metric transmission pipeline

Introduce queue component (eg kafkaesque) - then consumer or stream process service (storm, flint, spark) push data to time series db

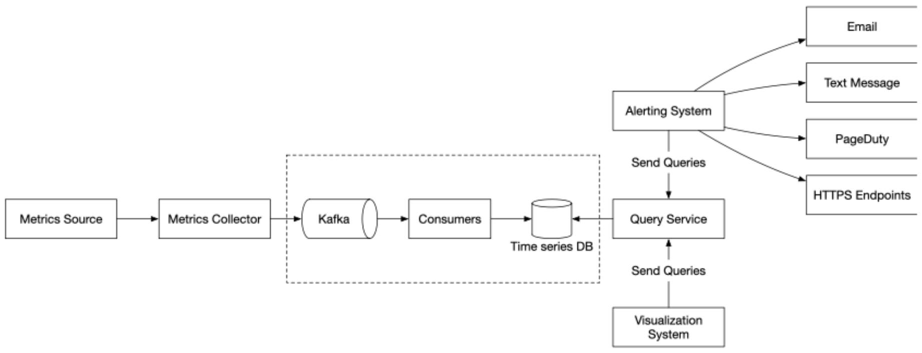

Figure 15 Add queues

scale through kafka
- configure partition based on throughput requirements.
- further partition metrics data with tags / labels
- prioritize metric to consume imp metric first.

kafkaesque alternative
- difficult to maintain production scale kafka
- alternative = fb gorilla in-memory time-series database

Where aggregation can happen?

Collection agent -

support simple aggregation logic (eg: aggregate counter every minute before sent to metrics collector)

ingestion pipeline -

aggregate before writing to storage, eg Flink - only write calculated results.

Query side -

raw data can be aggregated over given time period at query time, but query speed might be slower.

Query service - cluster of query server to access time-series database, handle request from visualization or alert system

cache layer - load time series db and make query service more performant.

NOTE: most popular metrics monitoring system eg prometheus and influxes don't use sql, but have their own query language eg hard to build sql query over time series data

Storage layer

1. Use time series db importantly.
2. Space optimization
   - data encoding and compression
   - downsampling (7 days - no sample, 30 days, downsample to 1 min, 1 year to downsample.1 hour resolution)
3. Cold storage - for inactive data.

Alert system

1. Load config files to cache server.
2. Alert manager fetch alert config from cache
3. Config rules - alert manager call query service at predefined interval - if violate threshold : alert event created
4. Retry : alert manager check alert state and ensure notification send at least once
5. Alert store.- key-value database eg Cassandra that keep state of all alerts.
6. Eligible alert inserted into kafka
7. Alert consumer process alert and send notification to different channel

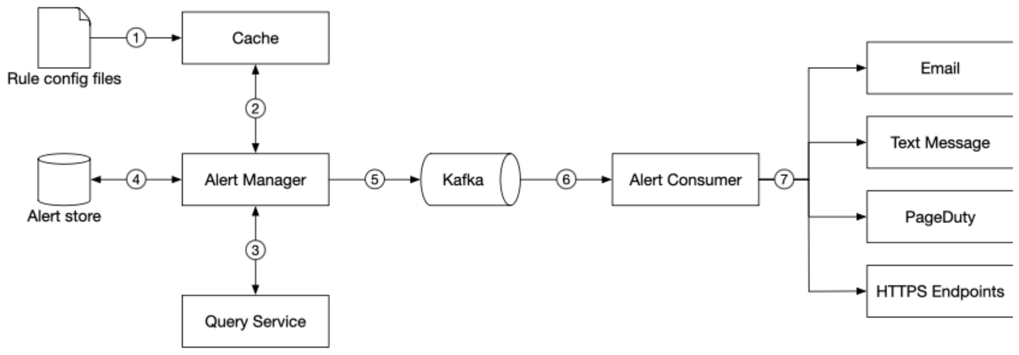

Figure 19 Alerting system

Visualization system

use Grafana for this purpose - integrates well with popular time series database which you can buy.

Wrap up

1. Pull vs push model for collecting metric data
2. Utilize kafkaesque to scale system
3. Choose right time-series database
4. Use downsampling to reduce data store
5. B**uild vs buy option for alerting and visualization system.**