This Document Is Shared Publicly

Loki: like Prometheus, but for logs.

Design Document
Tom Wilkie & David Kaltschmidt, March 2018

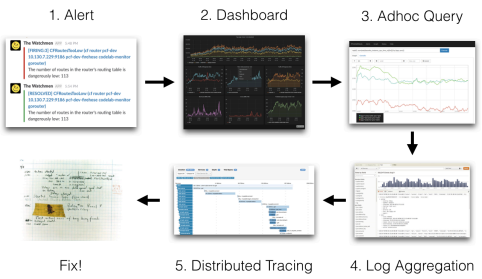
This document aims to explain the motivations for, and design of, the Grafana Loki service. This document does not attempt to describe in depth every possible detail of the design, but hopefully explains the key points and should allow us to spot any obvious mistakes ahead of time.

This document aims to answer questions not only about how we're going to build this, but also why we're building it, what it will be used for, and who will be using it.

Background & Motivations

Incident Response & Context Switching

Metrics are key to incident response; alerts are typically written as conditions over time series. But metrics can only be used to expose anticipated behaviour, due to their nature (they need to be pre-declared and limited in cardinality). Therefore metrics can only tell half the story; to get a complete picture of the cause of an incident, engineers typically resort to logs (textual output of a program) to get more detailed information.



It follows then that commonly incident response first starts with an alert, then some dashboard consultation (perhaps evolving the query in an ad hoc fashion) before pinpointing the exact service, host or instance that the errors are coming from. The engineer will then attempt to find the logs for that service/host/instance and time range to find the root cause. As the current status quo is for metrics and logs to be stored in two disparate systems, it is up to the engineer to translate the query from one language and interface to another.

Therefore the first hypothesis of this design is that minimising the cost of the context switching between logs and metrics will help reduce incident response times and improve user experience.

Existing Solutions

Log aggregation is not a new idea; like time series monitoring, there are many SaaS vendors and open source projects competing for attention. Almost all existing solutions involve using full-text search systems to index logs; at first glance this seems like obvious solutions, with a rich and powerful feature set allowing for complex queries.

These existing solutions are complicated to scale, resource intensive and difficult to operate. As mentioned above, an increasingly common pattern is the use of time series monitoring in combination with log aggregation - therefore the flexibility and sophistication offered by the query facilities often go unused; the majority of queries just focusing on a time range and some simple parameters (host, service etc). Using these systems for log aggregation is akin to using a sledgehammer to crack a nut.

The challenges and operational overheads of existing systems have led many buyers into the hands of SaaS operators. Therefore the second hypothesis of this design is that a different trade off can be struck between ease of operation and sophistication of the query language, one favouring ease of operation.

Cost Efficiency

With the migration towards SaaS log aggregation, the excessive costs of such systems has become more obvious. This cost arises not just from the technology used to implement full-text search - scaling and sharding an inverted index is hard; either writes touch each shard, or reads must - but also from the operation complexity.

A common experience amongst buyers looking for a log aggregation system, after receiving an initial quote for their existing logging load at an order-of-magnitude more than they are willing to spend, is to turn to the engineers and ask them to log less. As logging exists to cover the

unanticipated errors (see above), a typical response from engineers is one of disbelief - “what’s the point in logging if I have to be conscious of what I log?”.

Some systems have recently emerged offering a different trade off here. The open source OKLOG project by Peter Bourgon (now archived), eskews all forms of indexing beyond time-based, and adopts a eventually consistent, mesh-based distribution strategy. These two design decisions offer massive cost reduction and radically simpler operations, but in our opinion don’t meet our other design requirements - queries are not expressive enough and too expensive. We do however recognise this as being an attractive on-premise solution.

Therefore the third hypothesis is that a significantly more cost effective solution, with a slightly different indexing trade off, would be a really big deal...

Kubernetes

An interesting aside is to consider how logging has changed in the modern cloud native / microservices / containerised workloads. The standard pattern is now for applications to simply writes logs to STDOUT or STDERR. Platforms such as Kubernetes and Docker build on this to offer limited log aggregation features; logs are stored locally on nodes and can be fetched and aggregated on demand, using label selectors.

But with these simple systems, logs are often lost when a pod or node disappears. This is often one of the first triggers for a buyer to realise they need log aggregation - a pod or node mysteriously dies and no logs are available to diagnose why.

Prometheus and Cortex

Lastly it is worth covering how Prometheus fits into this picture. Prometheus is a monitoring system centered around a time series database. The TSDB indexes collections of samples (a time series) using a set of key-value pairs. Queries over these time series can be made by specifying a subset of these labels (matchers), returning all time series which match these labels. This differentiates from the likes of legacy Graphite hierarchical labels by making queries robust to the presence of new or changing labels.

In Prometheus (and Cortex), these labels are stored in an inverted index, making queries against these labels fast. This inverted index in Cortex exists in memory for recent data, and in a distributed KV store (BigTable, DynamoDB or Cassandra) for historic data. The Cortex index scales linearly by both retention and throughput, but by design is limited in cardinality of any given label.

The Prometheus system contains many components, but one noteworthy component for this discussion is mtail (https://github.com/google/mtail). Mtail allows you to “*extract whitebox monitoring data from application logs for collection in a time series database”.* This allows you

to build time series monitoring and alerts for applications which do not expose any metrics natively.

Stories

- After receiving an alert on my service and drilling into the query associated with said alert, I want to quickly see the logs associated with the jobs which produced those timeseries at the time of the alert.

- After a pod or node disappears, I want to be able to retrieve logs from just before it died, so I can diagnose why it died.

- After discovering an ongoing issue with my service, I want to extract a metric from some logs and combine it with my existing time series data.

- I have a legacy job which does not expose metrics about errors - it only logs them. I want to build an alert based on the rate of occurrences of errors in the log.

Non Goals

One common use case for log aggregation systems is to store structured, event-based data - for instance emitting an event for every request to a system, and including all the request details and metadata. With these kind of deployments comes the ability to ask questions like “show me top 10 users with highest 99th percentile latency”, something that you typically cannot do with time series metrics system due to the high cardinality of users. Whilst this use case is totally valid, it is not something we are targeting with this system.

Proposed Solution

We will build a hosted log aggregation system (*the system)* which indexes metadata associated with those log streams, rather than indexing the contents of the log streams themselves. This metadata will take the form of Prometheus-style multi-dimensional labels. These labels will be consistent with the labels associated with time series/metrics ingested from a job, such that the same labels can be used to find logs from a job as can be used to find time series from said job, enabling quick context switching in the UI.

The system will not solve many of the complicated distributed systems and storage challenges typically associated with log aggregation, but rather will offload them to existing distributed databases and object storage systems. This will reduce operational complexity by having the majority of the systems services be stateless and ephemeral, and allowing operators of the system to use the hosted services offered by cloud vendors.

By only indexing metadata associated with log streams, the system will reduce the load on the index by many orders of magnitude - I expect there for be ~1KB of metadata for potentially 100MBs of log data. The actual log data will be stored in a hosted object storage service (S3, GCS etc), which are experiencing massive downward cost pressure due to competition between vendors. We will be able to pass on these savings and offer the system at a price a few orders of magnitude lower than competitors. For example, GCS cost $0.026/GB/month, whereas Loggly costs ~$100/GB/month.

By virtue of this being a hosted system, logs will be trivially available after clients hosts or pods fail. An agent will be deployed to each node in a client's system to ship logs to our service, and ensure metadata is consistent with metrics.

Architecture

This section is a work in progress.

Agent

The first challenge is obtaining reliable metadata that is consistent with the metadata associated with the time series / metrics. To achieve this we will use the same service discovery and label relabelling libraries as Prometheus. This will be packaged up in a daemon that discovers targets, produces metadata labels and tails log files to produce streams of logs, which will be momentarily buffered on the client side and then sent to the service. Given the requirement about having recent logs on node failure, there is a fundamental limit to the amount of batching it can do.

This component exists and is called Promtail.

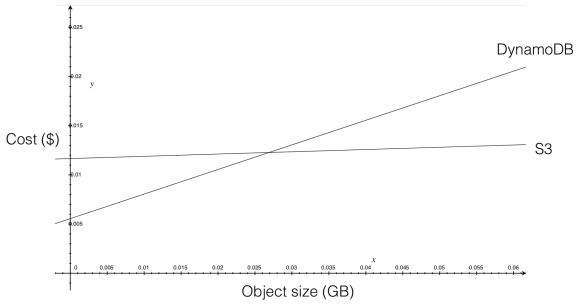
Life of a Write Request

The server-side components on the write path will mirror the Cortex architecture:

- Writes will first hit the Distributor, which is responsible for distributing and replicating the writes to the ingesters. We will use the Cortex consistent hash ring; we will distribute writes based on a hash of the entire metadata (including user id), as there is no convenient metric name for log streams.
- Next writes will hit a "log ingester" which batches up writes for the same stream in memory into "log chunks". When chunks hit a predefined size or age, they are periodically flushed to the Cortex chunk store.
- The Cortex chunk store will be updated to reduce copying of chunk data on the read and write path, and add support for writing chunks of GCS.

Log Chunks

The chunk format will be important to the cost and performance of the system. A chunk is all logs for a given label set over a certain period. The chunks must support appends, seeks and streaming reads. Assuming an "average" node will produce 10 GB of logs per day, and runs on average 30 containers, then each log stream will write at 4 KB/s. Expected compression rates should be around 10x for log data.



When picking the optimal chunk size we need to consider:

- Per operation cost vs storage costs; at small object size, per-operation costs dominate and its cheaper to store them in a database eg Bigtable.
- Per chunk index load - each chunk will require entries in the index; experience running Cortex tells us this is the largest cost component in running the system, but given a larger chunk size I suspect this won't be the case here.
- Cost of memory for building chunks and risk of loss. This will probably be the limiting factor. We should expect to be able to deal with streams for 1000s of hosts per machines to be able to cost effective run the service; if each stream need 1MB in memory and its 30 streams per host, this means 30GB of memory (and similar for WAL) per ingester. 1000 hosts also means 130MB/s inbound and outbound bandwidth and ingoing compression, which is a push.
- Compression effectiveness - at very small sizes (10s of bytes), compression is not effective; log lines will need to be batched together to achieve closer to optimal compression.

For example, 12 hours of log data will produce chunks around ~100 MB uncompressed, and ~10 MB compressed. 12 hours is also the upper limit for chunk length we use in Cortex. Given the memory requirements for building these, a chunk size closer to 1 MB (compressed) looks more likely.

The proposal is for a chunk to consist of a series of blocks; the first block is a gorilla-style encoded time index, and the subsequent blocks contain compressed log data. Once enough blocks have been produced to form a big enough chunk, they will be appended together to form a chunk. Some experimentation will be necessary to find the right chunk format, input here is welcome.

Life of a Query Request

As chunks are many orders of magnitude larger than Prometheus/Cortex chunks (Cortex chunks are maximum 1KB in size), it won't be possible to load them and decompress them in

their entirety. Therefore we will need to support streaming and iterating over them, only decompressing the parts we need as we go. Again, lots of details to work out here, but I suspect aggressive use of gRPC streaming and heaps will be in order (see the new Cortex iterating query PR)

TBD how to implement mtail like functionality - extra PromQL functions? The ruler?

API

- Log write
    - *???*
- Log range query
    - Returns streams that match the selector and the time range.
    - Request parameters
        - selector: Label set to identify the streams, e.g., {job=”webapp2”}, if no selector is given, all streams are returned
        - start: Start time of the interval in Epoch seconds
        - end: End time of the interval in Epoch seconds
    - Response fields
        - streams: List of streams, each with a metadata and the data sub-fields
- Log stream query
    - *Can this be modelled with the Log range query?*
    - *Should this be websockets?*
- Label space query
    - Returns available label keys for a given time range
    - Request parameters
        - start: Start time of the interval in Epoch seconds
        - end: End time of the interval in Epoch seconds
    - Response fields
        - keys: List of available label keys
- Label match query
    - Returns available label sets for a given selector and time range
    - Request parameters
        - selector: Label set to prefilter the labels (facetting), e.g., {job=”webapp2”} *(open question if empty selectors are too expensive to return all label sets)*
        - start: Start time of the interval in Epoch seconds
        - end: End time of the interval in Epoch seconds
    - Response fields
        - labels: List of label sets

Future Ideas

- We could potentially extract extra metadata from the logs on ingest, for inclusion in the index, for example log level. We would need to be mindful or cardinality explosion.
- I want to tap into a real time stream of logs of a specific service/instance, so i can get more familiar with how it behaves and/or check assumptions. This could be for a new instance that just came online, or from an already running one.

FAQ

This section is a work in progress. Feel free to add questions not explained.

- Where will the code live?
- Who is working on this?
- How will users deploy it?
- How will we bill for it?
- How will this work on-prem?

Alternatives Considered

This section is a work in progress.

- Running a hosted Elastic.
- Using OKLOG.
- Doing full text indexing using the Cortex index.

Compression Exploration

We experimented with various compression schemes and block sizes on some example log data:
https://docs.google.com/spreadsheets/d/1zTtln0kKspSWsHtwmamZzSnsHspHQigd-1jotZ8I9gc/edit?usp=sharing