Loki User Guide

Document Level Classification

200

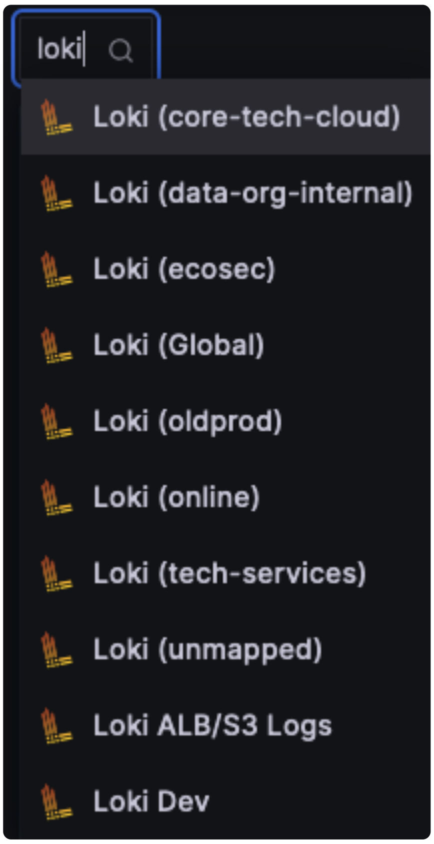
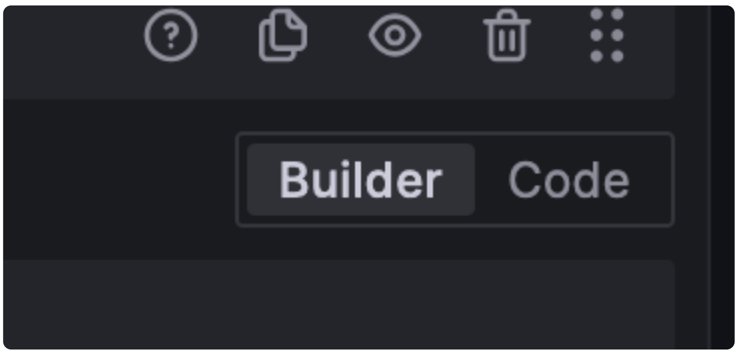
Discovery & Usage

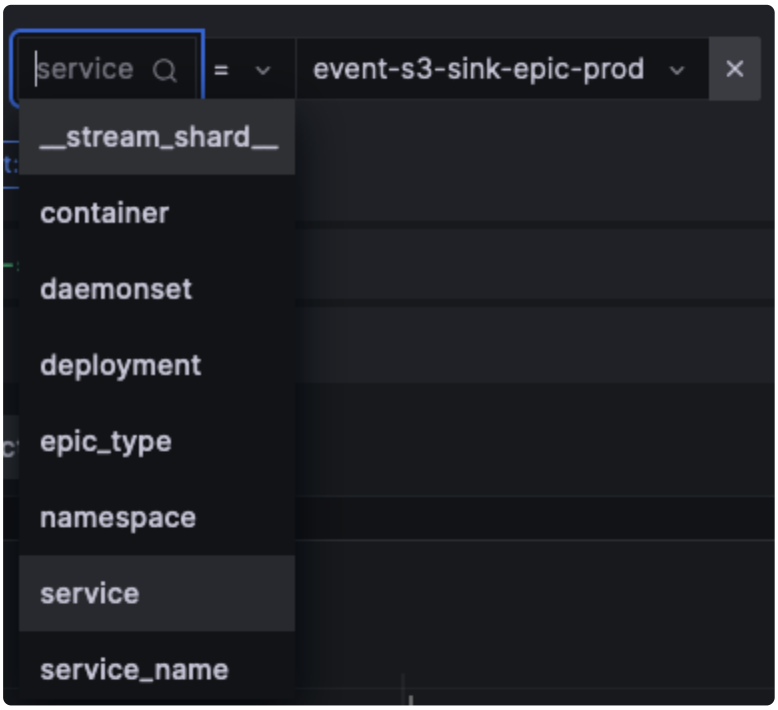
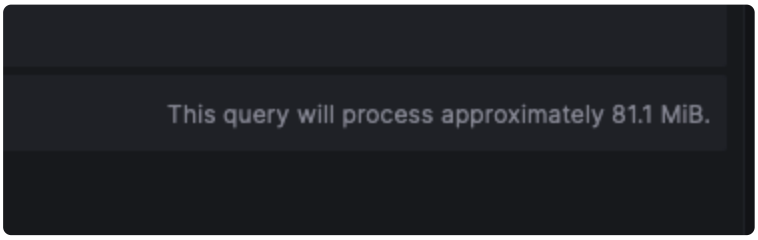
If you do not have Grafana access yet you can request it though SailPoint. There are multiple options here but the main one most everyone will need is **Grafana - Epic Live**. The correct role to request is **Grafana Epic editor** as this includes the ability to view the Loki Logs Explore page mentioned throughout this guide.

To get started, enter Grafana from the Okta tile and then find the Explore menu item from the hamburger menu in the top left. This will take you directly to the data interface to find Loki. You should end up at https://grafana.ol.epicgames.net/explore

Finding your logs

Once in the Explore interface, select the Loki data source that maps to your	

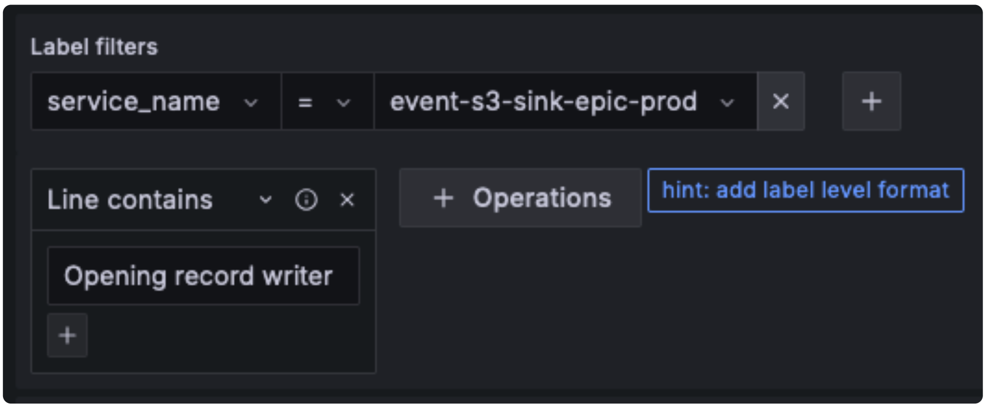
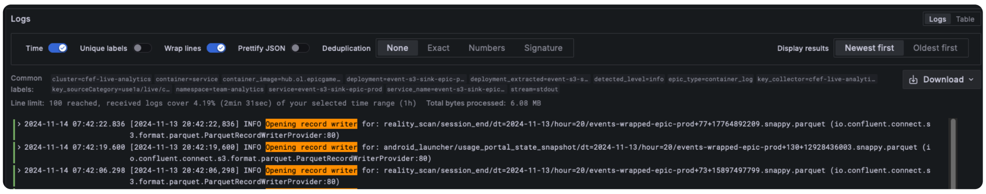
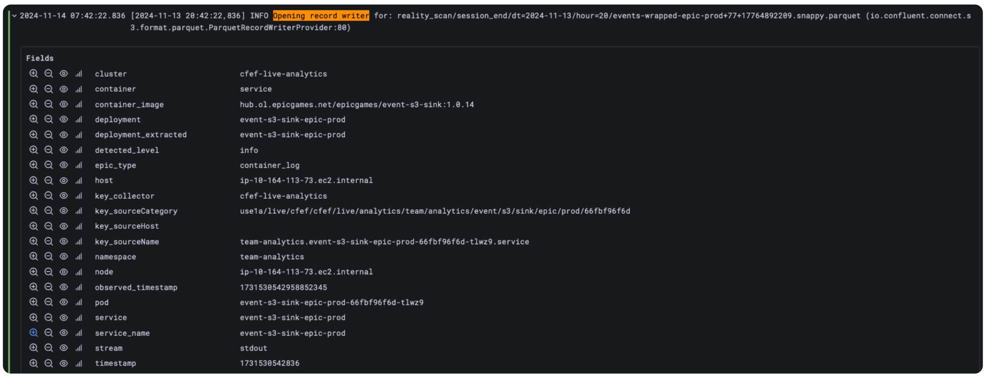
tenant. If you aren't sure or your expected tenant isn't present, try looking in the `Loki (unmapped)` data source, or check the tenant mapping, here. To follow along with the below examples, use the `Loki (data-org-internal)` tenant, and select the `event-s3-sink-epic-prod` service name	
Select the 'Builder' button on the right side of the query panel. This is a much easier way to get started with Loki rather than having to learn all of the LogQL syntax right away	

Select `service_name` under the 'Label filters' menu, then select the desired service from the value drop down. If you don't see your service, it is most likely in a different tenant. You may need to switch Loki data sources	
Loki will now give you an estimated volume of logs that will be searched based upon your selected labels + time range.	

Querying Logs

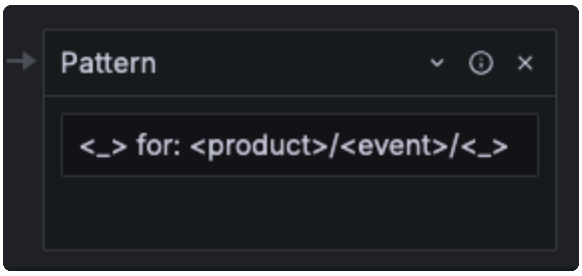
Line Contains

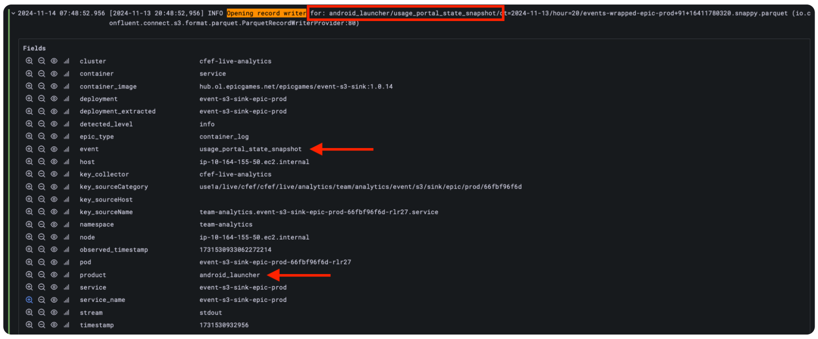
Once the service has been identified by label, a query can be composed. Line Contains is the most basic type of search operation, returning all lines that contain the specified string. For more advanced searching (regex, etc.), you can click the drop-down arrow on the Line Contains box.

Click the "*Operations*" button → *Line Filters* → *Line Contains*, and enter a string to search for in the prompt	
This will return any matching records in the table below	
Click on one of the lines to expand it and see what [structured metadata]() is attached to it	

Line Parsing (Pattern Matching)

Now, create some new structured metadata by parsing the log line. In this example, the product and event will be extracted from the log line.

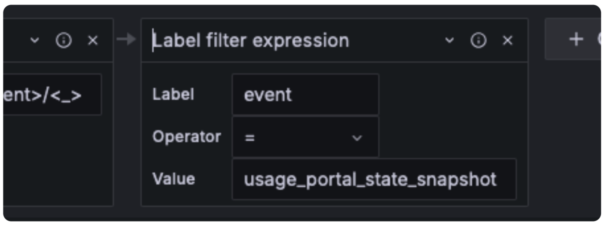
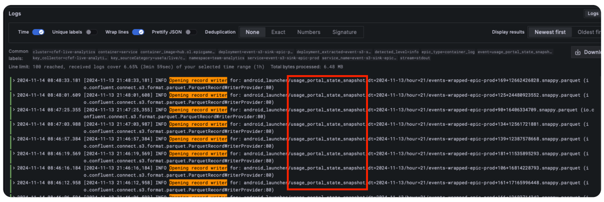
Click the *Operations* button → *Format* → *Pattern*, and enter *`: <_> for: <product>/<event>/<_>`*. This will look for the string "` for: `", then assign anything between that and the next forward slash to a `product` field. It will then assign anything up to the next forward slash to an `event` field. The `<_>` indicates "match anything, but don't collect as structured metadata". For more information, see the [`pattern`]() LogQL function	
When the query returns, new metadata will be	

decorating the log line:	

(For details on parsing JSON, see the section below)

Label Filtering

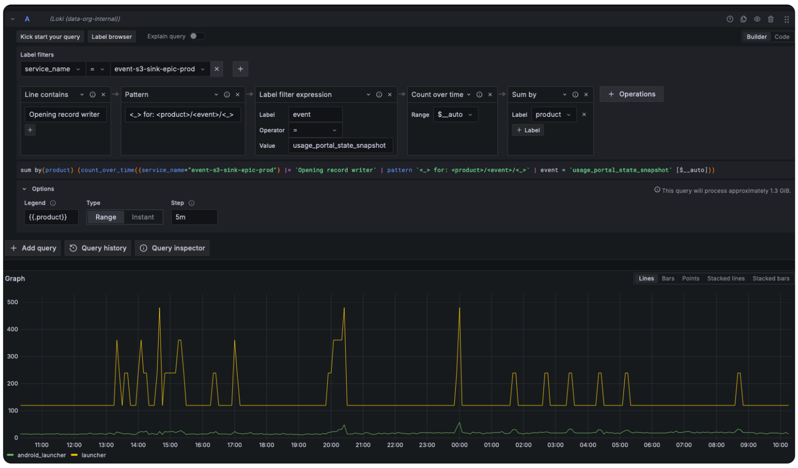
Now query for a specific event type.

Click on *Operations* → *Label Filter* → *Label Filter Expression*. Enter `event` in the Label field, and `usage_portal_state_snapshot` in the value field	
Only lines where the parsed event field match the specified value will be displayed	

Grouping Labels

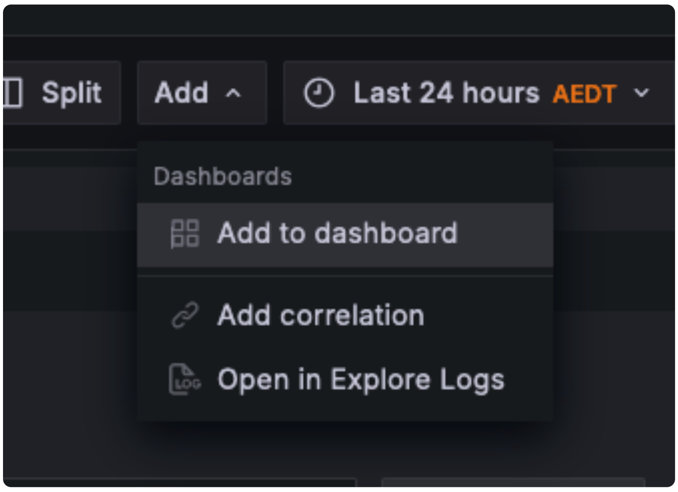
Next, group by product, so that we can see the count over time of logs generated.

- Click *Operations → Range Functions → Count Over Time*
- Click *Operations → Aggregations → Sum*
- In the Sum by 'Label' field, type in `product`
- Expand the Options bar below the query
- In the Legend field, enter `{{.product}}`
- In the Step field, enter `5m`



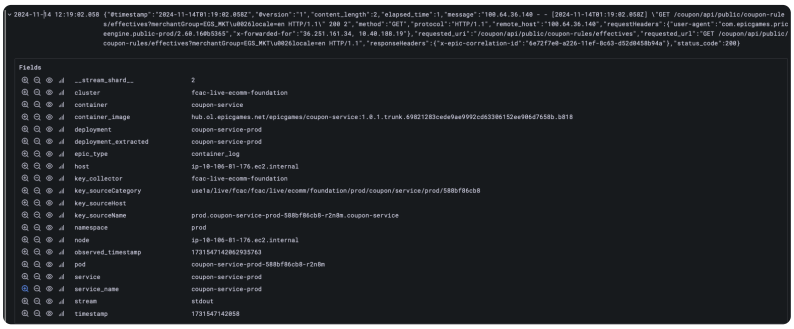
Adding To A Dashboard

Loki queries can be added to Grafana dashboards as standard panels / visualizations. You can add a query created in Grafana's Explore view to a new / existing dashboard by clicking *Add → Add To Dashboard* in the upper right corner



Line Parsing (JSON)

The above example used the `pattern` parser to decorate the log lines with additional structured metadata. For those logs that are emitting as JSON, a native JSON parser can be used to achieve the same effect! Let's take a look at an example:

• Select the `Loki (online)` data source • Select `service_name = coupon-service-prod`	
You can see that while the log line itself is JSON, none of the values are yet part of the structured	

metadata. To fix this: • Click *Operations → Formats → JSON* and repeat the search	

Now all fields within the JSON log message have been embedded as structured metadata and can be filtered / processed just as any other label would. **Note** that objects are flattened (e.g. `requestHeaders_user_agent`), and arrays / lists are completely skipped. More information on the JSON parser can be found [here]

Getting Data In

Sending Your Data

Substrate

If your service is in Substrate, congratulations, you're on the golden path! Your substrate cluster automatically forwards any logs emitted to stdout/stderr to Loki. You can split this into more streams to improve search capability, see the **Enhancing Your Data** section below.

Oldprod

Oldprod hosts use a Docker syslog driver to write to local rsyslog, which forwards to a special OpenTelemetry collector fleet that receives and processes logs for legacy services. These logs arrive in the Loki (Oldprod) tenant. While we support ingestion of data for oldprod services, we don't plan any improvements to this pipeline.

External/Other Use Cases

We maintain a pair of OpenTelemetry collectors for "all other use cases":

- Internal: https://internal-logs-collector.fbfb.live.use1a.on.epicgames.com
- External: https://external-logs-collector.fbfb.live.use1a.on.epicgames.com

Logs sent to these collectors MUST:

- be in OTLP format (ideally coming from another OpenTelemetry collector)
- provide basic auth credentials - see #ct-obs-support-ext for help with getting credentials
- provide an **X-Epic-Tenant-ID** header
- set a `service.name` resource attribute on all incoming logs. This should be in the format <your_service>-<env> i.e. foo-gamedev, foo-live

Enhancing Your Data

Your logging life will be better if you send structured log messages using JSON. See OATS Standard for Logging - we are optimizing our ingestion pipeline to support any standards that OATS adopts.

In addition to service_name, Loki also indexes by default on the following fields, which are pulled from your logs:

- **level** or **log.level -** the log level associated with a message. This gets indexed as the **level** label in Loki, and can help you filter your data. Supported values for this are **"info", "warn", "error", "debug", and "trace".**
- **category** - this is used to identify different streams in your logs. Current supported values for this are **application** and **access**. If

your service sends access logs alongside its application logs, this can help you separate that data.

We will be enhancing our pipeline in the coming months to support additional labels or structured metadata (which power bloom filters to make your searches even faster) - stay tuned.

Glossary

Tenant

Similarly to the way that SumoLogic uses partitions to segment data into more logical groupings, Loki has a concept of a tenant. A tenant is ostensibly an isolated collection of data to help partition workloads away from one another. That way, if you are part of a small tenant generating a small amount of logs, the amount of data that must be queried for your tenant is significantly less than if your workload was coupled in with a large tenant who generates significantly more load.

This also functions as an RBAC mechanism, so that sensitive data (e.g. EOS) can be logically separated from non-EOS data, and only authorized groups in Grafana are granted permission to query specific Loki tenants

At Epic, the use of tenants both in the scope of Loki and in a larger sense hasn't been well defined, yet. Currently all tenants are defined here. For Loki, tenants are currently scoped by AWS account (i.e. All EKS clusters within the same account will share the same tenant)

Labels

Within a tenant, Loki further partitions data by a number of higher-level constructs known as labels. Each unique combination of labels defines a stream (described below), so the more labels you add to your query, the less data needs to be searched. Currently we define the following labels:

- service_name
- namespace

- cluster
- deployment
- container
- epic_type
- daemonset
- statefulset
- cronjob

Generally speaking this will allow you to specify enough labels to identify your specific application or service, so that way if you are co-located in the same cluster as a 'noisy', high volume service, it will not impact performance when querying your service.

Note: *For the 'oldprod' data tenant,* `service_name` *and* `source` *are the only available labels to choose from*

Structured Metadata

Structured Metadata are like attributes associated with a log line that can be used for query or filtering purposes. They function identically to labels in this regard. The key difference is that labels are explicitly used to define Loki streams and should be low cardinality (this is not something that as an end-user you need to worry about). Structured metadata fields can be high cardinality however, as they are used exclusively to decorate individual log lines for query purposes. Each log line that's ingested has a number of structured metadata fields already populated. All labels are presented as structured metadata.

LogQL

Loki's query language is known as LogQL. It's "inspired by" PromQL in that it retains similar syntax for label selection, functions, range vectors, etc., but also builds upon it by allowing for a series of chained filters to be applied to each query. More information can be found here

Chunks

Loki stores all ingested data as S3 objects in what it calls 'chunks'. A chunk is a binary representation of the ingested log data that is persisted in object storage (S3).

Data Streams

Chunks are partitioned into data streams by their unique combination of labels when they are ingested. A data stream therefore uniquely identifies all logs for a specific service. This is how the estimated amount of data to read is determined as you add labels to your query. This unique combination of labels per stream is what allows a low volume service to not be impacted by a high volume service that may be co-located on the same cluster; Loki will only ever need to query that small amount of data due to the data stream that identifies that particular service

Concepts

Query Performance

Loki doesn't perform any pre-indexing of data upon ingest (unlike a system such as ElasticSearch). This makes it suitable for scaling to extremely large data sets, but comes with the trade off that queries are effectively "brute force" and must scan all chunks within a data stream until the desired number of matches are found. Work is being done to implement bloom filters into Loki to reduce the number of chunks that need to be scanned for a given stream, but for now it's safe to assume that all chunks for a stream must be scanned.

When a query is sent to Loki, it is broken down into smaller queries based on time called "splits". As of writing, each split covers a 5 minute window, and each split is further broken down into additional sub-queries called "shards". It's a bit hand-wavey as to how many shards are generated from each split (this article somewhat explains it), but for optimal query performance you want more splits. More splits means more shards, more shards means more parallelization across query nodes.

Let's say that your service generates 100GB of data spread evenly over a 1 hour window. Your query will first be broken down into 20 splits (1h / 5m), then further broken down into 1->n * 20 shards, with each shard being executed on a querier in parallel. Assuming 20 shards per split, this means you'll have 400 separate queries being executed in parallel to process that 100GiB of data. (~250MiB each)

Conversely, if you generate 100GiB of data over a 5 minute period, then querying that same amount of data will be much slower, as you'll only have a single split and its 20 shards, or 20 separate queries executing in parallel (~5GiB each).

The exception here is when running aggregation queries and adjusting the Step value of the Grafana panel (e.g. setting a `step` value of `1h` instead of auto). This is somewhat analogous to the `timeslice` function in SumoLogic. In this case, if you're querying 24h of data with a Step value of `1h`, you'll only end up with 24 splits (24h / 1 hour) and their corresponding shards instead of 288 splits (24h / 5m) + shards. This leads to lower parallelization, which leads to more data being processed per querier, which leads to slower performance.

In general, if you find yourself frequently making these sorts of large step-window aggregation queries, consider emitting schematized events to Data Router and using a dedicated analytics engine such as Databricks to perform your queries instead. Performance will be orders of magnitude better for large window aggregations.

TL;DR

Treat Loki like distributed grep, not as a data warehouse Error

Alerting Guidelines

Don't use Loki for Alerts!

Loki is primarily optimized for ad-hoc log analysis and troubleshooting. In order to crunch TBs of arbitrarily structured log data quickly, we run a large fleet of (expensive) compute that is ready to work on-demand. Alerting acts as a continuous drain on these resources, and is one of the most expensive ways to continuously monitor your applications. Instead, use one of the following:

- If you routinely want to know about a specific error condition, emit a metric when it occurs, and alert against that. Link to specific log searches in your runbook.
- Use Sentry for general exception/error tracking.

In the near future, we will be taking direct action to pause or remove Loki-based alerts. This is necessary to keep costs in check and ensure Loki is available for its intended purpose.

Query Tips

Writing efficient searches

Some rules to keep in mind when writing searches:

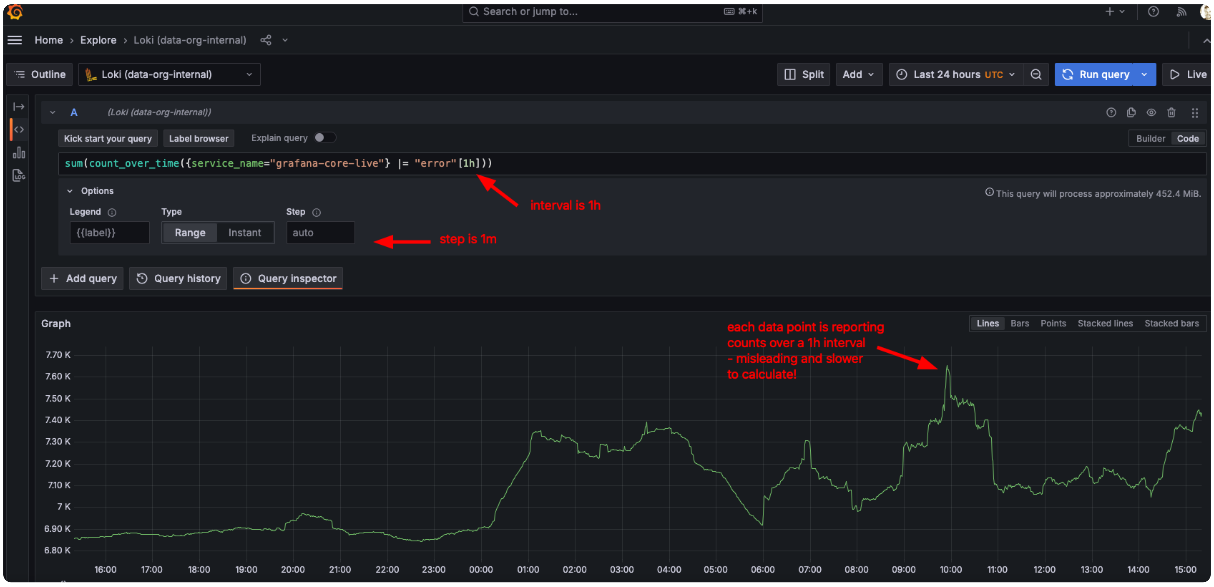
1. Reduce data as much as possible using labels, structured metadata, and simple text search before doing compute intensive operations like parsing or aggregations
2. Simple text filters are your friend. Put these first in your search pipeline. Avoid doing text-insensitive or regex search as a first filtering operation if possible

Examples

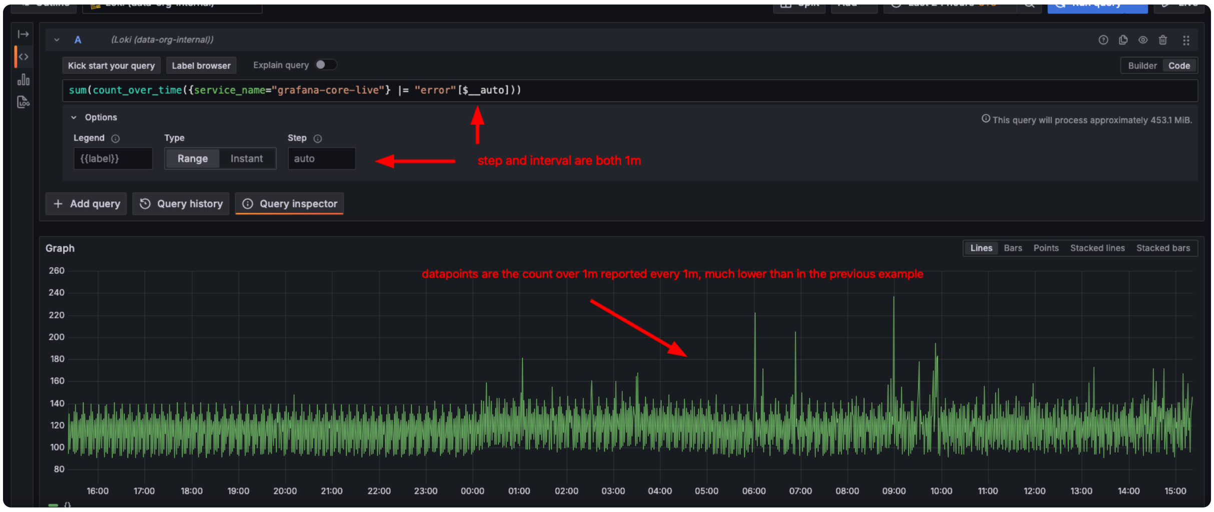
```
// BAD - every line is run through a JSON parser (expensive/slow) befor
{service_name="grafana-core-live"} | json | level="error", msg="error r

{service_name="account-service-prod"} | json | CORRID="FN-rLvmBQNNZ8ub_

// GOOD - filtering up front with appropriate log level and simple stri
// You can still do precise filtering on JSON fields on the reduced set
{service_name="grafana-core-live", level="error"} |= "NerdGraph" | json

{service_name="account-service-prod", level="info"} |= "FN-rLvmBQNNZ8ub

// BAD - this regex case-insensitive search is more compute intensive a
{service_name="grafana-core-live"} |~ "(?i)cbd8fd48-894f-4470-8456-08d5

// GOOD - just use a simple string search
{service_name="grafana-core-live"} |= "cbd8fd48-894f-4470-8456-08d5ba58
```

Use $__auto with step for metrics queries

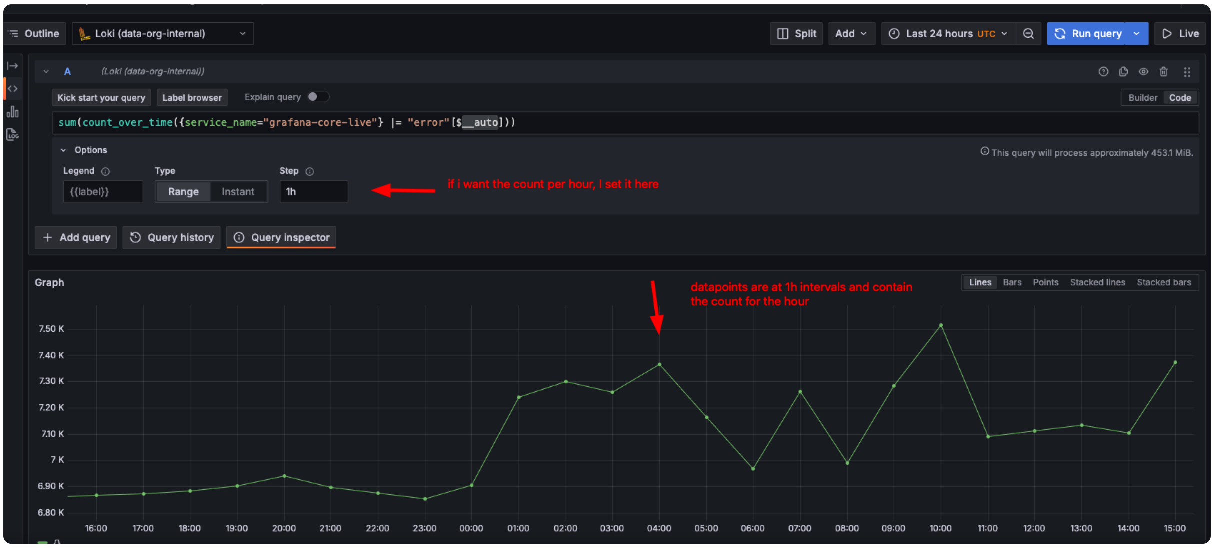
Grafana/Loki provide a helper variable, $__auto, which automatically adjusts to the "Step" argument in your query. **You should use this** unless you have a specific reason to override it. Setting the interval and step improperly can result in slower queries **and** misleading data when doing range queries. Here's an example. I've run a count_over_time for the last hour, but ignored the `Step` setting. Grafana sets this to 1m. Each datapoint is thus the count over time hour for each data point. For a 24h period with 1 minute interval,s that's 1400 data points, and Loki has to perform a `count_over_time()[1h]` starting at t, t-1, t-2, t-1440. This gives back a time series graph that appears to report values of 7000 requests per minute - but this is misleading, each data point is showing the count over an hour.



Using `$__auto` - Grafana adjusts the interval to 1m, and each data point is now the result of `count_over_time()[1m]` - much more accurate.



If I want the actual, hourly count, I adjust the step to 1h. Grafana adjusts the query to `count_over_time()[1h]` but also only requests 24 data points. The results here are still accurate - around 7000 requests per HOUR, not per minute as it appeared in the first graph.



Use sum instead of count

It's easy to use confuse **count** with **sum** - generally you want **sum**. Per the docs:

- `sum`: Calculate sum over labels
- `count`: Count number of elements in the vector

In practice we've found **sum** to be much faster/more efficient.

Counting high cardinality series

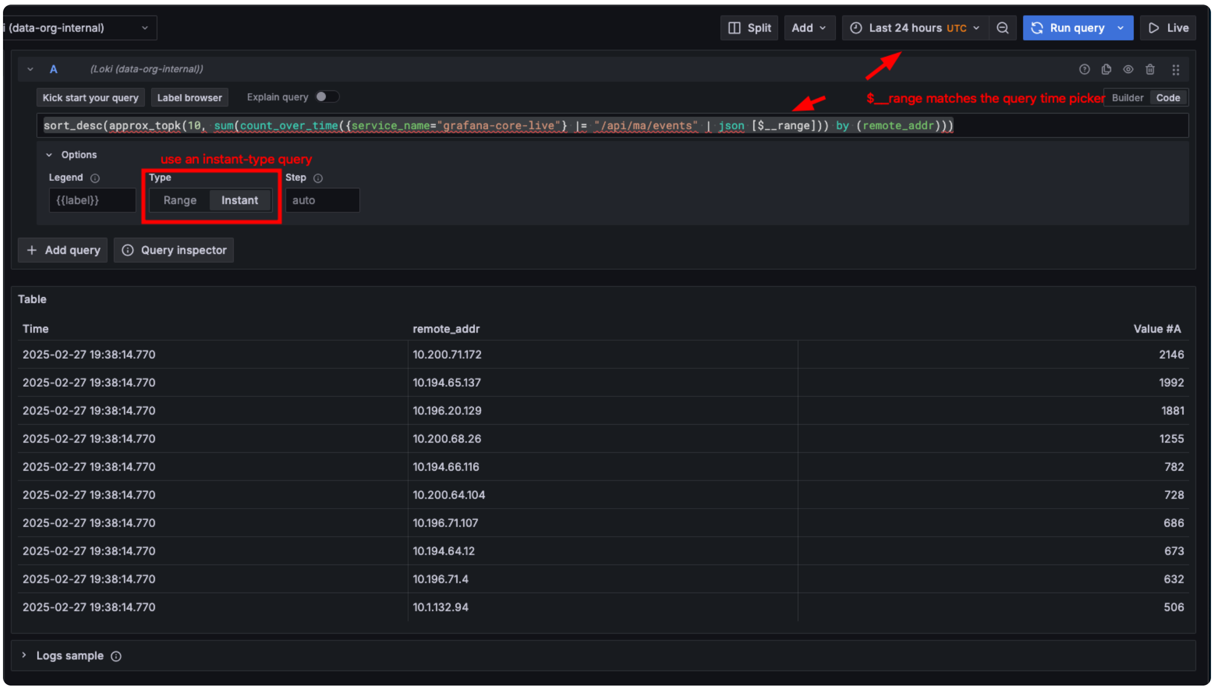
Loki will refuse to count more than 5000 unique series, so this makes it challenging to run "top 10 users" or "top 10 ip" style queries. Grafana introduced a new function, approx_topk, which can help in these specific scenarios. Here's an example, where we do a topk looking at the remote IPs that have hit Grafana.

```
sort_desc(approx_topk(10, sum(count_over_time({service_name="grafana-co
```

Some notes:

- Use $__range here to match the query time range you specify in the UI

- These queries only work as "instant" queries.. If you get a "count min sketches are only supported on instant queries", remember to set the query type from **range** to **instant**.



Page Information:
Page ID: 81068511
Space: Cloud Developer Platform
Downloaded: 2025-07-12 04:08:30