Get Started

- Loki architecture
- Loki components
- Loki deployment modes
- Consistent hash rings
- Loki overview
- Label best practices
- Cardinality
- Modify default OpenTelemetry labels
- What is structured metadata
- Quickstart to run Loki locally
- Loki Tutorial

Loki architecture

Grafana Loki has a microservices-based architecture and is designed to run as a horizontally scalable, distributed system. The system has multiple components that can run separately and in parallel. The Grafana Loki design compiles the code for all components into a single binary or Docker image. The `-target` command-line flag controls which component(s) that binary will behave as.

To get started easily, run Grafana Loki in "single binary" mode with all components running simultaneously in one process, or in "simple scalable deployment" mode, which groups components into read, write, and backend parts.

Grafana Loki is designed to easily redeploy a cluster under a different mode as your needs change, with no configuration changes or minimal configuration changes.

For more information, refer to Deployment modes and Components.

Loki components

Storage

Loki stores all data in a single object storage backend, such as Amazon Simple Storage Service (S3), Google Cloud Storage (GCS), Azure Blob Storage, among others. This mode uses an adapter called **index shipper** (or short **shipper**) to store index (TSDB) files the same way we store chunk files in object storage. This mode of operation became generally available with Loki 2.0 and is fast, cost-effective, and simple. It is where all current and future development lies.

Prior to 2.0, Loki had different storage backends for indexes and chunks. For more information, refer to Legacy storage.

Data format

Grafana Loki has two main file types: **index** and **chunks**.

- The **index** is a table of contents of where to find logs for a specific set of labels.
- The **chunk** is a container for log entries for a specific set of labels.

Loki data format: chunks and indexes

The diagram above shows the high-level overview of the data that is stored in the chunk and data that is stored in the index.

Index format

There is one index format that is currently supported as single store with index shipper:

- TSDB

  Time Series Database (or short TSDB) is an index format originally developed by the maintainers of Prometheus for time series (metric) data.
  It is extensible and has many advantages over the deprecated BoltDB index. New storage features in Loki are solely available when using TSDB.

Chunk format

A chunk is a container for log lines of a stream (unique set of labels) of a specific time range.

The following ASCII diagram describes the chunk format in detail.

```
-----------------------------------------------------------------------------
|                        |                       |                          |
|     MagicNumber(4b)    |     version(1b)       |      encoding (1b)       |
|                        |                       |                          |
-----------------------------------------------------------------------------
|                      #structuredMetadata (uvarint)                        |
-----------------------------------------------------------------------------
|      len(label-1) (uvarint)      |          label-1 (bytes)               |
-----------------------------------------------------------------------------
|      len(label-2) (uvarint)      |          label-2 (bytes)               |
-----------------------------------------------------------------------------
|      len(label-n) (uvarint)      |          label-n (bytes)               |
-----------------------------------------------------------------------------
|                      checksum(from #structuredMetadata)                   |
-----------------------------------------------------------------------------
|           block-1 bytes          |           checksum (4b)                |
-----------------------------------------------------------------------------
|           block-2 bytes          |           checksum (4b)                |
-----------------------------------------------------------------------------
|           block-n bytes          |           checksum (4b)                |
-----------------------------------------------------------------------------
|                          #blocks (uvarint)                                |
```

```
-------------------------------------------------------------------------
| #entries(uvarint) | mint, maxt (varint)  | offset, len (uvarint)         |
-------------------------------------------------------------------------
| #entries(uvarint) | mint, maxt (varint)  | offset, len (uvarint)         |
-------------------------------------------------------------------------
| #entries(uvarint) | mint, maxt (varint)  | offset, len (uvarint)         |
-------------------------------------------------------------------------
| #entries(uvarint) | mint, maxt (varint)  | offset, len (uvarint)         |
-------------------------------------------------------------------------
|                          checksum(from #blocks)                       |
-------------------------------------------------------------------------
| #structuredMetadata len (uvarint) | #structuredMetadata offset (uvarint) |
-------------------------------------------------------------------------
|     #blocks len (uvarint)         |       #blocks offset (uvarint)       |
-------------------------------------------------------------------------
```

`mint` and `maxt` describe the minimum and maximum Unix nanosecond timestamp, respectively.

The `structuredMetadata` section stores non-repeated strings. It is used to store label names and label values from structured metadata. Note that the labels strings and lengths within the `structuredMetadata` section are stored compressed.

Block format

A block is comprised of a series of entries, each of which is an individual log line. Note that the bytes of a block are stored compressed. The following is their form when uncompressed:

```
----------------------------------------------------------------------------------------------
------------------------------------------------
|  ts (varint)  |  len (uvarint)  |  log-1 bytes  |  len(from #symbols)  |  #symbols
(uvarint)  |  symbol-1 (uvarint)  | symbol-n*2 (uvarint) |
----------------------------------------------------------------------------------------------
------------------------------------------------
|  ts (varint)  |  len (uvarint)  |  log-2 bytes  |  len(from #symbols)  |  #symbols
(uvarint)  |  symbol-1 (uvarint)  | symbol-n*2 (uvarint) |
----------------------------------------------------------------------------------------------
------------------------------------------------
|  ts (varint)  |  len (uvarint)  |  log-3 bytes  |  len(from #symbols)  |  #symbols
(uvarint)  |  symbol-1 (uvarint)  | symbol-n*2 (uvarint) |
----------------------------------------------------------------------------------------------
------------------------------------------------
```

```
|  ts (varint)  |  len (uvarint)  |  log-n bytes  |  len(from #symbols)  |  #symbols
(uvarint)  |  symbol-1 (uvarint)  | symbol-n*2 (uvarint) |
--------------------------------------------------------------------------------------
-----------------------------------------------
```

`ts` is the Unix nanosecond timestamp of the logs, while `len` is the length in bytes of the log entry.

Symbols store references to the actual strings containing label names and values in the `structuredMetadata` section of the chunk.

Write path

On a high level, the write path in Loki works as follows:

1. The distributor receives an HTTP POST request with streams and log lines.
2. The distributor hashes each stream contained in the request so it can determine the ingester instance to which it needs to be sent based on the information from the consistent hash ring.
3. The distributor sends each stream to the appropriate ingester and its replicas (based on the configured replication factor).
4. The ingester receives the stream with log lines and creates a chunk or appends to an existing chunk for the stream's data. A chunk is unique per tenant and per label set.
5. The ingester acknowledges the write.
6. The distributor waits for a majority (quorum) of the ingesters to acknowledge their writes.
7. The distributor responds with a success (2xx status code) in case it received at least a quorum of acknowledged writes. or with an error (4xx or 5xx status code) in case write operations failed.

Refer to Components for a more detailed description of the components involved in the write path.

Kafka-based write path (experimental)

{{< admonition type="warning" >}} Kafka-based ingestion is an experimental feature. Engineering and on-call support is not available. No SLA is provided. {{< /admonition >}}

Loki can optionally use Apache Kafka, or a Kafka-protocol-compatible system, as a durable buffer on the write path. When you enable Kafka-based ingestion, the write path changes: the distributor writes log streams as Kafka records to a configured topic, instead of, or in addition to, pushing them directly to ingesters over gRPC.

Each ingester consumes from exactly one Kafka partition, determined by its position in the partition ring. This decouples ingestion durability from ingester availability, because Kafka buffers

writes while an ingester is restarting or unavailable.

Enabling Kafka-based ingestion primarily changes the write path. The read path still works the same way, but the querier must be configured to look up ingesters using the partition ring instead of the classic hash ring. For full details, including configuration, monitoring, and migration guidance, refer to Kafka-based ingestion.

Read path

On a high level, the read path in Loki works as follows:

1. The query frontend receives an HTTP GET request with a LogQL query.
2. The query frontend splits the query into sub-queries and passes them to the query scheduler.
3. The querier pulls sub-queries from the scheduler.
4. The querier passes the query to all ingesters for in-memory data.
5. The ingesters return in-memory data matching the query, if any.
6. The querier lazily loads data from the backing store and runs the query against it if ingesters returned no or insufficient data.
7. The querier iterates over all received data and deduplicates, returning the result of the sub-query to the query frontend.
8. The query frontend waits for all sub-queries of a query to be finished and returned by the queriers.
9. The query frontend merges the individual results into a final result and return it to the client.

Refer to Components for a more detailed description of the components involved in the read path.

Multi-tenancy

All data, both in memory and in long-term storage, may be partitioned by a tenant ID, pulled from the `X-Scope-OrgID` HTTP header in the request when Grafana Loki is running in multi-tenant mode. When Loki is **not** in multi-tenant mode, the header is ignored and the tenant ID is set to `fake`, which will appear in the index and in stored chunks.

Loki components

{{< youtube id="_hv4i84Z68s" >}}

Loki is a modular system that contains many components that can either be run together (in "single binary" mode with target `all`), in logical groups (in "simple scalable deployment" mode with targets `read`, `write`, `backend`), or individually (in "microservice" mode). For more information see Deployment modes.

Component	*individual*	`all`	`read`	`write`	`backend`
Distributor	x	x		x	
Ingester	x	x		x	
Query Frontend	x	x	x		
Query Scheduler	x	x			x
Querier	x	x	x		
Index Gateway	x				x
Compactor	x	x			x
Ruler	x	x			x
Pattern ingester	x	x		x	
Bloom Planner (Deprecated)	x				x
Bloom Builder (Deprecated)	x				x
Bloom Gateway (Deprecated)	x				x

This page describes the responsibilities of each of these components.

{{< admonition type="note" >}} Kafka is infrastructure that Loki connects to, not a Loki component binary, so it doesn't have its own row in the table above. When Kafka-based ingestion is enabled, the Distributor and Ingester rows in the table above have optional Kafka write and read behavior. Refer to Kafka-based ingestion for details. {{< /admonition >}}

Distributor

The **distributor** service is responsible for handling incoming push requests from clients. It's the first step in the write path for log data. Once the distributor receives a set of streams in an HTTP request, each stream is validated for correctness and to ensure that it is within the configured tenant (or global) limits. Each valid stream is then sent to `n` ingesters in parallel, where `n` is the replication factor for data. The distributor determines the ingesters to which it sends a stream to using consistent hashing.

A load balancer must sit in front of the distributor to properly balance incoming traffic to them. In Kubernetes, the service load balancer provides this service.

The distributor is a stateless component. This makes it easy to scale and offload as much work as possible from the ingesters, which are the most critical component on the write path. The ability to independently scale these validation operations means that Loki can also protect itself against denial of service attacks that could otherwise overload the ingesters. It also allows us to fan-out writes according to the replication factor.

Validation

The first step the distributor takes is to ensure that all incoming data is according to specification. This includes things like checking that the labels are valid Prometheus labels as well as ensuring the timestamps aren't too old or too new or the log lines aren't too long.

Preprocessing

Currently, the only way the distributor mutates incoming data is by normalizing labels. What this means is making `{foo="bar", bazz="buzz"}` equivalent to `{bazz="buzz", foo="bar"}`, or in other words, sorting the labels. This allows Loki to cache and hash them deterministically.

Rate limiting

The distributor can also rate-limit incoming logs based on the maximum data ingest rate per tenant. It does this by checking a per-tenant limit and dividing it by the current number of distributors. This allows the rate limit to be specified per tenant at the cluster level and enables us to scale the distributors up or down and have the per-distributor limit adjust accordingly. For instance, say we have 10 distributors and tenant A has a 10MB rate limit. Each distributor will allow up to 1MB/s before limiting. Now, say another large tenant joins the cluster and we need to spin up 10 more distributors. The now 20 distributors will adjust their rate limits for tenant A to `(10MB / 20 distributors) = 500KB/s`

. This is how global limits allow much simpler and safer operation of the Loki cluster.

{{< admonition type="note" >}} The distributor uses the `ring` component under the hood to register itself amongst its peers and get the total number of active distributors. This is a different "key" than the ingesters use in the ring and comes from the distributor's own ring configuration. {{< /admonition >}}

Forwarding

Once the distributor has performed all of its validation duties, it forwards data to the ingester component which is ultimately responsible for acknowledging the write operation.

Replication factor

In order to mitigate the chance of *losing* data on any single ingester, the distributor will forward writes to a *replication factor* of them. Generally, the replication factor is `3`. Replication allows for ingester restarts and rollouts without failing writes and adds additional protection from data loss for some scenarios. Loosely, for each label set (called a *stream*) that is pushed to a distributor, it will hash the labels and use the resulting value to look up `replication_factor` ingesters in the `ring` (which is a subcomponent that exposes a distributed hash table). It will then try to write the same data to all of them. This will generate an error if less than a *quorum* of writes succeeds. A quorum is defined as `floor( replication_factor / 2 ) + 1`. So, for our `replication_factor` of `3`, we require that two writes succeed. If less than two writes succeed, the distributor returns an error and the write operation will be retried.

{{< admonition type="caution" >}} If a write is acknowledged by 2 out of 3 ingesters, we can tolerate the loss of one ingester but not two, as this would result in data loss. {{< /admonition >}}

The replication factor is not the only thing that prevents data loss, though, and its main purpose is to allow writes to continue uninterrupted during rollouts and restarts. The ingester component now includes a write ahead log (WAL) which persists incoming writes to disk to ensures they are not lost as long as the disk isn't corrupted. The complementary nature of the replication factor and WAL ensures data isn't lost unless there are significant failures in both mechanisms (that is, multiple ingesters die and lose/corrupt their disks).

Hashing

Distributors use consistent hashing in conjunction with a configurable replication factor to determine which instances of the ingester service should receive a given stream.

A stream is a set of logs associated to a tenant and a unique label set. The stream is hashed using both the tenant ID and the label set and then the hash is used to find the ingesters to send the

stream to.

A hash ring, maintained by peer-to-peer communication using the Memberlist protocol, or stored in a Key-Value store such as Consul is used to achieve consistent hashing; all ingesters register themselves into the hash ring with a set of tokens they own. Each token is a random unsigned 32-bit number. Along with a set of tokens, ingesters register their state into the hash ring. The state `JOINING`, and `ACTIVE` may all receive write requests, while `ACTIVE` and `LEAVING` ingesters may receive read requests. When doing a hash lookup, distributors only use tokens for ingesters who are in the appropriate state for the request.

To do the hash lookup, distributors find the smallest appropriate token whose value is larger than the hash of the stream. When the replication factor is larger than 1, the next subsequent tokens (clockwise in the ring) that belong to different ingesters will also be included in the result.

The effect of this hash setup is that each token that an ingester owns is responsible for a range of hashes. If there are three tokens with values 0, 25, and 50, then a hash of 3 would be given to the ingester that owns the token 25; the ingester owning token 25 is responsible for the hash range of 1-25.

Quorum consistency

Since all distributors share access to the same hash ring, write requests can be sent to any distributor.

To ensure consistent query results, Loki uses Dynamo-style quorum consistency on reads and writes. This means that the distributor will wait for a positive response of at least one half plus one of the ingesters to send the sample to before responding to the client that initiated the send.

{{< admonition type="note" >}} When Kafka-based ingestion is enabled, the distributor can also, or instead, write streams to Kafka. Refer to Kafka-based ingestion for details. {{< /admonition >}}

Ingester

The **ingester** service is responsible for persisting data and shipping it to long-term storage (Amazon Simple Storage Service, Google Cloud Storage, Azure Blob Storage, etc.) on the write path, and returning recently ingested, in-memory log data for queries on the read path.

Ingesters contain a *lifecycler* which manages the lifecycle of an ingester in the hash ring. Each ingester has a state of either `PENDING`, `JOINING`, `ACTIVE`, `LEAVING`, or `UNHEALTHY`:

1. `PENDING` is an Ingester's state when it is waiting for a handoff from another ingester that is `LEAVING`. This only applies for legacy deployment modes.
   {{< admonition type="note" >}} Handoff is a deprecated behavior mainly used in stateless deployments of ingesters, which is discouraged. Instead, it's recommended using a stateful deployment model together with the write ahead log. {{< /admonition >}}
2. `JOINING` is an Ingester's state when it is currently inserting its tokens into the ring and initializing itself. It may receive write requests for tokens it owns.
3. `ACTIVE` is an Ingester's state when it is fully initialized. It may receive both write and read requests for tokens it owns.
4. `LEAVING` is an Ingester's state when it is shutting down. It may receive read requests for data it still has in memory.
5. `UNHEALTHY` is an Ingester's state when it has failed to heartbeat. `UNHEALTHY` is set by the distributor when it periodically checks the ring.

Each log stream that an ingester receives is built up into a set of many "chunks" in memory and flushed to the backing storage backend at a configurable interval.

Chunks are compressed and marked as read-only when:

1. The current chunk has reached capacity (a configurable value).
2. Too much time has passed without the current chunk being updated
3. A flush occurs.

Whenever a chunk is compressed and marked as read-only, a writable chunk takes its place.

If an ingester process crashes or exits abruptly, all the data that has not yet been flushed will be lost. Loki is usually configured to replicate multiple replicas (usually 3) of each log to mitigate this risk.

When a flush occurs to a persistent storage provider, the chunk is hashed based on its tenant, labels, and contents. This means that multiple ingesters with the same copy of data will not write the same data to the backing store twice, but if any write failed to one of the replicas, multiple differing chunk objects will be created in the backing store. See Querier for how data is deduplicated.

Timestamp Ordering

Loki is configured to accept out-of-order writes by default.

When not configured to accept out-of-order writes, the ingester validates that ingested log lines are in order. When an ingester receives a log line that doesn't follow the expected order, the line is rejected and an error is returned to the user.

The ingester validates that log lines are received in timestamp-ascending order. Each log has a timestamp that occurs at a later time than the log before it. When the ingester receives a log that does not follow this order, the log line is rejected and an error is returned.

Logs from each unique set of labels are built up into "chunks" in memory and then flushed to the backing storage backend.

If an ingester process crashes or exits abruptly, all the data that has not yet been flushed could be lost. Loki is usually configured with a Write Ahead Log which can be *replayed* on restart as well as with a `replication_factor` (usually 3) of each log to mitigate this risk.

When not configured to accept out-of-order writes, all lines pushed to Loki for a given stream (unique combination of labels) must have a newer timestamp than the line received before it. There are, however, two cases for handling logs for the same stream with identical nanosecond timestamps:

1. If the incoming line exactly matches the previously received line (matching both the previous timestamp and log text), the incoming line will be treated as an exact duplicate and ignored.
2. If the incoming line has the same timestamp as the previous line but different content, the log line is accepted. This means it is possible to have two different log lines for the same timestamp.

Handoff

{{< admonition type="warning" >}} Handoff is deprecated behavior mainly used in stateless deployments of ingesters, which is discouraged. Instead, it's recommended using a stateful deployment model together with the write ahead log. {{< /admonition >}}

By default, when an ingester is shutting down and tries to leave the hash ring, it will wait to see if a new ingester tries to enter before flushing and will try to initiate a handoff. The handoff will transfer all of the tokens and in-memory chunks owned by the leaving ingester to the new ingester.

Before joining the hash ring, ingesters will wait in `PENDING` state for a handoff to occur. After a configurable timeout, ingesters in the `PENDING` state that have not received a transfer will join the ring normally, inserting a new set of tokens.

This process is used to avoid flushing all chunks when shutting down, which is a slow process.

Filesystem support

Ingesters can flush chunks to the filesystem when Loki uses TSDB as the index store. TSDB ships its index files to object storage, including a filesystem object store, so queriers running in a separate

process can read the same data, as long as every process can reach the same directory. This differs from the deprecated, non-shipped BoltDB index store, which only allows one process to hold a lock on the database file at a time.

{{< admonition type="note" >}} When Kafka-based ingestion is enabled, ingesters consume log streams from a Kafka partition instead of, or in addition to, receiving them directly from the distributor over gRPC. Refer to Kafka-based ingestion for details. {{< /admonition >}}

Query frontend

The **query frontend** is an **optional service** providing the querier's API endpoints and can be used to accelerate the read path. When the query frontend is in place, incoming query requests should be directed to the query frontend instead of the queriers. The querier service will be still required within the cluster, in order to execute the actual queries.

The query frontend internally performs some query adjustments and holds queries in an internal queue. In this setup, queriers act as workers which pull jobs from the queue, execute them, and return them to the query frontend for aggregation. Queriers need to be configured with the query frontend address (via the `-querier.frontend-address` CLI flag) in order to allow them to connect to the query frontends.

Query frontends are **stateless**. However, due to how the internal queue works, it's recommended to run a few query frontend replicas to reap the benefit of fair scheduling. Two replicas should suffice in most cases.

Queueing

If no separate query scheduler component is used, the query frontend will also perform basic query queueing.

- Ensure that large queries, that could cause an out-of-memory (OOM) error in the querier, will be retried on failure. This allows administrators to under-provision memory for queries, or optimistically run more small queries in parallel, which helps to reduce the total cost of ownership (TCO).
- Prevent multiple large requests from being convoyed on a single querier by distributing them across all queriers using a first-in/first-out queue (FIFO).
- Prevent a single tenant from denial-of-service-ing (DOSing) other tenants by fairly scheduling queries between tenants.

Splitting

The query frontend splits larger queries into multiple smaller queries, executing these queries in parallel on downstream queriers and stitching the results back together again. This prevents large (multi-day, etc) queries from causing out of memory issues in a single querier and helps to execute them faster.

Caching

Metric queries

The query frontend supports caching metric query results and reuses them on subsequent queries. If the cached results are incomplete, the query frontend calculates the required sub-queries and executes them in parallel on downstream queriers. The query frontend can optionally align queries with their step parameter to improve the cacheability of the query results. The result cache is compatible with any Loki caching backend (currently Memcached, Redis, and an in-memory cache).

Log queries

The query frontend also supports caching of log queries in form of a negative cache. This means that instead of caching the log results for quantized time ranges, Loki only caches empty results for quantized time ranges. This is more efficient than caching actual results because log queries are limited (usually 1000 results) and if you have a query over a long time range that matches only a few lines, and you only cache actual results, you'd still need to process a lot of data in addition to the data from the results cache in order to verify that nothing else matches.

Index stats queries

The query frontend caches index stats query results similar to the metric query results. This cache is only applicable when using single store TSDB.

Log volume queries

The query frontend caches log volume query results similar to the metric query results. This cache is only applicable when using single store TSDB.

Query scheduler

The **query scheduler** is an **optional service** providing more advanced queuing functionality than the query frontend. When using this component in the Loki deployment, query frontend pushes split up queries to the query scheduler which enqueues them in an internal in-memory queue. There is a queue for each tenant to guarantee the query fairness across all tenants. The queriers that connect to the query scheduler act as workers that pull their jobs from the queue, execute

them, and return them to the query frontend for aggregation. Queriers therefore need to be configured with the query scheduler address (via the `-querier.scheduler-address` CLI flag, or the `scheduler_address` field in the `frontend_worker` YAML block) in order to allow them to connect to the query scheduler. As an alternative to a static address, query schedulers can register themselves in a hash ring so that queriers and query frontends discover them automatically. For more information, refer to Scale Loki.

Query schedulers are **stateless**. However, due to the in-memory queue, it's recommended to run more than one replica to keep the benefit of high availability. Two replicas should suffice in most cases.

Querier

The **querier** service is responsible for executing Log Query Language (LogQL) queries. The querier can handle HTTP requests from the client directly (in "single binary" mode, or as part of the read path in "simple scalable deployment") or pull subqueries from the query frontend or query scheduler (in "microservice" mode).

It fetches log data from both the ingesters and from long-term storage. Queriers query all ingesters for in-memory data before falling back to running the same query against the backend store. Because of the replication factor, it is possible that the querier may receive duplicate data. To resolve this, the querier internally **deduplicates** data that has the same nanosecond timestamp, label set, and log message.

Index Gateway

The **index gateway** service is responsible for handling and serving metadata queries. Metadata queries are queries that look up data from the index. The index gateway is only used by single store TSDB.

The query frontend queries the index gateway for the log volume of queries so it can make a decision on how to shard the queries. The queriers query the index gateway for chunk references for a given query so they know which chunks to fetch and query.

The index gateway can run in `simple` or `ring` mode. In `simple` mode, each index gateway instance serves all indexes from all tenants. In `ring` mode, index gateways use a consistent hash ring to distribute and shard the indexes per tenant amongst available instances.

Compactor

The **compactor** service is used by single store TSDB, to compact the multiple index files produced by the ingesters and shipped to object storage into single index files per day and tenant. This makes index lookups more efficient.

To do so, the compactor downloads the files from object storage in a regular interval, merges them into a single one, uploads the newly created index, and cleans up the old files.

Additionally, the compactor is also responsible for log retention and log deletion.

In a Loki deployment, the compactor service is usually run as a single instance.

Ruler

The **ruler** service manages and evaluates rule and/or alert expressions provided in a rule configuration. The rule configuration is stored in object storage (or alternatively on local file system) and can be managed via the ruler API or directly by uploading the files to object storage.

Alternatively, the ruler can also delegate rule evaluation to the query frontend. This mode is called remote rule evaluation and is used to gain the advantages of query splitting, query sharding, and caching from the query frontend.

When running multiple rulers, they use a consistent hash ring to distribute rule groups amongst available ruler instances.

Pattern ingester

The optional **pattern ingester** component receives log data from the ingesters and scans the logs to detect and aggregate patterns. This can be useful for understanding the structure of your logs at scale. The pattern ingester is used by the pattern feature in Logs Drilldown, which lets you detect similar log lines and add or exclude them from your search.

The ingester uses a drain algorithm to identify related logs that share the same pattern, and maintain their counts over time. Patterns consist of a number, a string, and a Loki series identifier.

The pattern ingester exposes a query API, so you can fetch detected patterns. This API is used by the Patterns tab in the Grafana Logs Drilldown plugin.

This component is disabled by default and must be enabled in your Loki config file.

Kafka-based ingestion (experimental)

{{< admonition type="warning" >}} This feature is an experimental feature. Engineering and on-call support is not available. No SLA is provided. {{< /admonition >}}

Loki can optionally use Apache Kafka, or a Kafka-protocol-compatible system, as a durable buffer on the write path, instead of, or in addition to, the direct gRPC path between the distributor and the ingester.

When you enable Kafka-based ingestion:

- The distributor writes each log stream as one or more records to a Kafka topic.
- Each ingester consumes records from exactly one Kafka partition of that topic.
- Ingesters use the **partition ring**, a hash ring similar to the one ingesters use for gRPC writes, to coordinate which ingester owns which partition. Unlike the main hash ring, the partition ring assigns one partition to one ingester at a time, rather than distributing hash ranges across replicas.

Because Kafka durably stores records until an ingester consumes them, ingesters can restart or roll out without losing or blocking writes: on restart, an ingester resumes consuming from the last offset it committed to Kafka.

Enabling Kafka-based ingestion primarily changes the write path. The read path still works the same way, but the querier must be configured to look up ingesters using the partition ring instead of the classic hash ring. For details, refer to Kafka-based ingestion.

For configuration, monitoring, and migration guidance, refer to Kafka-based ingestion.

Bloom Planner

{{< admonition type="warning" >}} This feature is deprecated and will be removed in a future release. Engineering and on-call support is not available. No SLA is provided. {{< /admonition >}}

The Bloom Planner service is responsible for planning the tasks for blooms creation. It runs as a singleton and provides a queue from which tasks are pulled by the Bloom Builders. The planning

runs periodically and takes into account what blooms have already been built for a given day and tenant and what series need to be newly added.

This service is also used to apply blooms retention.

Bloom Builder

{{< admonition type="warning" >}} This feature is deprecated and will be removed in a future release. Engineering and on-call support is not available. No SLA is provided. {{< /admonition >}}

The Bloom Builder service is responsible for processing the tasks created by the Bloom Planner. The Bloom Builder creates bloom blocks from structured metadata of log entries. The resulting blooms are grouped in bloom blocks spanning multiple series and chunks from a given day. This component also builds metadata files to track which blocks are available for each series and TSDB index file.

The service is stateless and horizontally scalable.

Bloom Gateway

{{< admonition type="warning" >}} This feature is deprecated and will be removed in a future release. Engineering and on-call support is not available. No SLA is provided. {{< /admonition >}}

The Bloom Gateway service is responsible for handling and serving chunks filtering requests. The index gateway queries the Bloom Gateway when computing chunk references, or when computing shards for a given query. The gateway service takes a list of chunks and a filtering expression and matches them against the blooms, filtering out any chunks that do not match the given label filter expression.

The service is horizontally scalable. When running multiple instances, the client (Index Gateway) shards requests across instances based on the hash of the bloom blocks that are referenced.

Loki deployment modes

Loki is a distributed system consisting of many microservices. It also has a unique build model where all of those microservices exist within the same binary.

You can configure the behavior of the single binary with the `-target` command-line flag to specify which microservices will run on startup. You can further configure each of the components in the `loki.yaml` file.

Because Loki decouples the data it stores from the software which ingests and queries it, you can easily redeploy a cluster under a different mode as your needs change, with minimal or no configuration changes.

Comparison of deployment modes

	Monolithic	HA Monolithic	Microservices
Durability	□	□	□
High availability	□	□	□
Separation of execution paths	□	□	□
Operational complexity	□□ low	□□ medium	□□ high
Scalability	□□ low	□□ medium	□□ high

Monolithic mode

Sometimes also referred to as **single binary** deployment.

The simplest mode of operation is the monolithic deployment mode. You enable monolithic mode by setting the `-target=all` command line parameter. This mode runs all of Loki's microservice components inside a single process as a single binary or Docker image.

monolithic mode diagram

Monolithic mode is useful for getting started quickly to experiment with Loki, as well as for small read/write volumes of up to approximately 20GB per day.

Query parallelization is limited by the number of instances and the setting `max_query_parallelism` which is defined in the `loki.yaml` file.

HA Monolithic mode

Sometimes also referred to as **HA single binary** deployment. This mode replaces the deprecated Simple Scalable Deployment (SSD), which will be removed in Loki 4.0.

HA monolithic mode sits between monolithic mode and microservices mode: it provides high availability and moderate horizontal scalability without the operational complexity of managing individual microservices. If you need fine-grained, per-component scaling, use microservices mode instead.

You can horizontally scale a monolithic mode deployment to more instances by using a shared object store, and by configuring the `ring` section of the `loki.yaml` file to share cluster state between all instances.

Configuration requirements

The following requirements must be met to run Loki in HA monolithic mode:

1. **Run every instance with `-target=all`.** All replicas use the single-binary target so that every process serves every component.
2. **Shared object storage.** All instances must read from and write to the same S3-compatible bucket (or equivalent) for chunk data and, if rulers are used, a separate bucket for ruler data. Local disk is only used for ephemeral caches and the write-ahead log (WAL).
3. **Memberlist for ring state.** Configure the `memberlist` block so that every instance knows the addresses of all other instances. Set `common.ring.kvstore.store: memberlist` so that all internal rings (ingester, distributor, and so on) use the shared memberlist cluster.
4. **Replication factor of 3.** Set `common.replication_factor: 3` and run **at least three** instances so that the cluster can tolerate the loss of one instance without data loss.
5. **Dedicated compactor instance.** Designate one instance as the main compactor by setting `compactor.horizontal_scaling_mode: main` on one instance and `worker` on all others. The address of the main compactor needs to be set in `common.compactor_grpc_address`.
6. **Load balancer in front of all instances.** Route all inbound push and query traffic to every replica in round-robin order.

See the Loki HA Monolithic example for a more complete configuration.

Microservices mode

Sometimes also referred to as **distributed** deployment.

The microservices deployment mode runs components of Loki as distinct processes. The microservices deployment is also referred to as a Distributed deployment. Each process is invoked specifying its `target`. For release 3.3 the components are:

- Bloom Builder (deprecated)
- Bloom Gateway (deprecated)
- Bloom Planner (deprecated)
- Compactor
- Distributor
- Index Gateway
- Ingester
- Overrides Exporter
- Querier
- Query Frontend
- Query Scheduler
- Ruler
- Table Manager (deprecated)

{{< admonition type="tip" >}} You can see the complete list of targets for your version of Loki by running Loki with the flag `-list-targets`, for example:

```
docker run docker.io/grafana/loki:3.2.1 -config.file=/etc/loki/local-config.yaml -list-targets
```

{{< /admonition >}}

Microservices mode diagram

Running components as individual microservices provides more granularity, letting you scale each component as individual microservices, to better match your specific use case.

Microservices mode deployments can be more efficient Loki installations. However, they are also the most complex to set up and maintain.

Microservices mode is only recommended for very large Loki clusters or for operators who need more precise control over scaling and cluster operations.

Microservices mode is designed for Kubernetes deployments. A community-supported Helm chart is available for deploying Loki in microservices mode.

Consistent hash rings

Consistent hash rings are incorporated into Loki cluster architectures to

- aid in the sharding of log lines
- implement high availability
- ease the horizontal scale up and scale down of clusters. There is less of a performance hit for operations that must rebalance data.

Hash rings connect instances of a single type of component when

- there are a set of Loki instances in monolithic deployment mode
- there are multiple read components or multiple write components in simple scalable deployment mode
- there are multiple instances of one type of component in microservices mode

Not all Loki components are connected by hash rings. These components need to be connected into a hash ring:

- distributors
- ingesters
- query schedulers
- compactors
- rulers

These components can optionally be connected into a hash ring:

- index gateway

In an architecture that has three distributors and three ingesters defined, the hash rings for these components connect the instances of same-type components.

Distributor and ingester rings

Each node in the ring represents an instance of a component. Each node has a key-value store that holds communication information for each of the nodes in that ring. Nodes update the key-value store periodically to keep the contents consistent across all nodes. For each node, the key-value store holds:

- an ID of the component node
- component address, used by other nodes as a communication channel
- an indication of the component node's health

Configuring rings

Define ring configuration within the `common.ring` block.

Use the `memberlist` key-value store type unless there is a compelling reason to use a different key-value store type. `memberlist` uses a gossip protocol to propagate information to all the nodes to guarantee the eventual consistency of the key-value store contents.

There are additional configuration options for distributor rings, ingester rings, and ruler rings. These options are for advanced, specialized use only. These options are defined within the `distributor.ring` block for distributors, the `ingester.lifecycler.ring` block for ingesters, and the `ruler.ring` block for rulers.

About the distributor ring

Distributors use the information in their key-value store to keep a count of the quantity of distributors in the distributor ring. The count further informs cluster limits.

About the ingester ring

Ingester ring information in the key-value stores is used by distributors. The information lets the distributors shard log lines, determining which ingester or set of ingesters a distributor sends log lines to.

About the query scheduler ring

Query schedulers use the information in their key-value store for service discovery of the schedulers. This allows queriers to connect to all available schedulers, and it allows schedulers to connect to all available query frontends, effectively creating a single queue that aids in balancing the query load.

About the compactor ring

Compactors use the information in the key-value store to identify a single compactor instance that will be responsible for compaction. The compactor is only enabled on the responsible instance, despite the compactor target being on multiple instances.

About the ruler ring

The ruler ring is used to determine which rulers evaluate which rule groups.

About the index gateway ring

The index gateway ring is used to determine which gateway is responsible for which tenant's indexes when queried by rulers or queriers.

Loki overview

Loki is a horizontally-scalable, highly-available, multi-tenant log aggregation system inspired by Prometheus. Loki differs from Prometheus by focusing on logs instead of metrics, and collecting logs via push, instead of pull.

Loki is designed to be very cost effective and highly scalable. Unlike other logging systems, Loki does not index the contents of the logs, but only indexes metadata about your logs as a set of labels for each log stream.

A log stream is a set of logs which share the same labels. Labels help Loki to find a log stream within your data store, so having a quality set of labels is key to efficient query execution.

Log data is then compressed and stored in chunks in an object store such as Amazon Simple Storage Service (S3) or Google Cloud Storage (GCS), or even, for development or proof of concept, on the filesystem. A small index and highly compressed chunks simplify the operation and significantly lower the cost of Loki.

{{< figure src="../loki-overview-2.png" caption="**Loki logging stack**" >}}

A typical Loki-based logging stack consists of 3 components:

- **Agent** - An agent or client, for example Grafana Alloy. The agent scrapes logs, turns the logs into streams by adding labels, and pushes the streams to Loki through an HTTP API.
- **Loki** - The main server, responsible for ingesting and storing logs and processing queries. It can be deployed in three different configurations, for more information see deployment modes.
- **Grafana** for querying and displaying log data. You can also query logs from the command line, using LogCLI or using the Loki API directly.

Loki features

- **Scalability** - Loki is designed for scalability, and can scale from as small as running on a Raspberry Pi to ingesting petabytes a day. Loki can run as a single binary for simple setups, in HA monolithic mode for moderate horizontal scalability without added operational complexity, or as fine-grained microservices designed to run natively within Kubernetes for the largest, highest-scale installations.

{{< admonition type="note" >}} Simple Scalable Deployment (SSD) mode, which decoupled requests into separate read and write paths, is deprecated and will be removed in Loki 4.0. The new HA monolithic mode will be the recommended replacement for most SSD use cases. See deployment modes for details. {{< /admonition >}}

- **Multi-tenancy** - Loki allows multiple tenants to share a single Loki instance. With multi-tenancy, the data and requests of each tenant is completely isolated from the others. Multi-tenancy is configured by assigning a tenant ID in the agent.
- **Third-party integrations** - Several third-party agents (clients) have support for Loki, via plugins. This lets you keep your existing observability setup while also shipping logs to Loki.
- **Efficient storage** - Loki stores log data in highly compressed chunks. Similarly, the Loki index, because it indexes only the set of labels, is significantly smaller than other log aggregation tools. By leveraging object storage as the only data storage mechanism, Loki inherits the reliability and stability of the underlying object store. It also capitalizes on both the cost efficiency and operational simplicity of object storage over other storage mechanisms like locally attached solid state drives (SSD) and hard disk drives (HDD). The compressed chunks, smaller index, and use of low-cost object storage, make Loki less expensive to operate.
- **LogQL, the Loki query language** - LogQL is the query language for Loki. Users who are already familiar with the Prometheus query language, PromQL, will find LogQL familiar and flexible for generating queries against the logs. The language also facilitates the generation of metrics from log data, a powerful feature that goes well beyond log aggregation.
- **Alerting** - Loki includes a component called the ruler, which can continually evaluate queries against your logs, and perform an action based on the result. This allows you to monitor your logs for anomalies or events. Loki integrates with Prometheus Alertmanager, or the alert manager within Grafana.
- **Grafana integration** - Loki integrates with Grafana, Mimir, and Tempo, providing a complete observability stack, and seamless correlation between logs, metrics and traces.

Frequently asked questions

{{< qa-list >}} {{< qa question="What is a good open source tool for aggregating and querying logs?" >}} Grafana Loki is a horizontally scalable, highly available log aggregation system built for the cloud-native era. Instead of expensively indexing every line of your logs, Loki indexes just a small set of labels and stores the rest in cheap object storage such as Amazon S3, Google Cloud Storage, or Azure Blob Storage, dramatically lowering your costs while scaling effortlessly from a Raspberry Pi to petabytes a day. Paired with the powerful LogQL query language and deeply integrated with Grafana, Loki brings your logs, metrics, and traces together in one seamless observability experience. {{< /qa >}} {{< qa question="What are the best open source log

aggregation tools for Kubernetes?" >}} The strongest open source option for most cloud-native teams, Grafana Loki is the best choice. Purpose-built for Kubernetes, Loki indexes only a small set of labels, rather than the full contents of every log line, then stores compressed logs in cheap object storage such as Amazon S3, Google Cloud Storage, or Azure Blob Storage. The result is dramatically lower cost and effortless scaling—from a single binary up to petabytes a day. With the Prometheus-inspired LogQL query language, built-in alerting, and seamless integration with Grafana, Mimir, and Tempo, Loki delivers cost-efficient, Kubernetes-native log aggregation inside a complete observability stack. {{< /qa >}} {{< qa question="What is Grafana Loki and how does it work?" >}} Grafana Loki is a horizontally scalable, highly available, multi-tenant log aggregation system inspired by Prometheus. Unlike traditional logging tools that index the full contents of every log line, Loki indexes only a small set of labels (metadata) for each log stream, while the log data itself is compressed and stored in chunks in low-cost object storage such as Amazon S3, Google Cloud Storage, or Azure Blob Storage, keeping the index small and making Loki cheaper to operate and easier to scale. In practice, an agent like Grafana Alloy scrapes your logs, attaches labels to turn them into streams, and pushes them to Loki over HTTP; Loki then ingests and stores those streams and serves queries written in LogQL, its Prometheus-inspired query language. You can explore the results in Grafana, alert on log patterns using the built-in ruler, and correlate your logs with metrics and traces for a complete observability stack. {{< /qa >}} {{< qa question="How does Grafana Loki store and query logs?" >}} Grafana Loki separates a small index from the bulk log data to keep storage cheap and scalable. Rather than indexing the full contents of your logs, Loki groups incoming logs into streams identified by a set of labels and indexes only that metadata; the log lines themselves are compressed and written as chunks to low-cost object storage such as Amazon S3, Google Cloud Storage, or Azure Blob Storage. When you run a query, Loki uses the labels to quickly locate the relevant streams, then fetches and decompresses only the matching chunks, which is why a well-chosen, low-cardinality set of labels is key to fast queries. You express those queries in LogQL, Loki's Prometheus-inspired query language, using label matchers to select streams, line and label filters to narrow results, and pipeline stages that can even generate metrics from your log data. {{< /qa >}} {{< /qa-list >}}

Label best practices

Grafana Loki is under active development, and we are constantly working to improve performance. But here are some of the most current best practices for labels that will give you the best experience with Loki.

Static labels are good

Use labels for things like regions, clusters, servers, applications, namespaces, and environments. They will be fixed for a given system/app and have bounded values. Use static labels to make it easier to query your logs in a logical sense (for example, show me all the logs for a given application and specific environment, or show me all the logs for all the apps on a specific host).

Use dynamic labels sparingly

Too many label value combinations leads to too many streams. The penalties for that in Loki are a large index and small chunks in the store, which in turn can actually reduce performance.

To avoid those issues, don't add a label for something until you know you need it! Use filter expressions (`|= "text"`, `|~ "regex"`, ...) and brute force those logs. It works -- and it's fast.

If you often parse a label from a log line at query time, the label has a high cardinality, and extracting that label is expensive in terms of performance; consider extracting the label on the client side attaching it as structured metadata to log lines .

From early on, we have set a label dynamically using client pipelines for `level`. This seemed intuitive for us as we often wanted to only show logs for `level="error"`; however, we are re-evaluating this now as writing a query. `{app="loki"} |= "level=error"` is proving to be just as fast for many of our applications as `{app="loki",level="error"}`.

This may seem surprising, but if applications have medium to low volume, that label causes one application's logs to be split into up to five streams, which means 5x chunks being stored. And loading chunks has an overhead associated with it. Imagine now if that query were `{app="loki",level!="debug"}`. That would have to load **way** more chunks than `{app="loki"} != "level=debug"`.

Above, we mentioned not to add labels until you *need* them, so when would you *need* labels?? A little farther down is a section on `chunk_target_size`. If you set this to 1MB (which is reasonable),

this will try to cut chunks at 1MB compressed size, which is about 5MB-ish of uncompressed logs (might be as much as 10MB depending on compression). If your logs have sufficient volume to write 5MB in less time than `max_chunk_age`, or **many** chunks in that timeframe, you might want to consider splitting it into separate streams with a dynamic label.

What you want to avoid is splitting a log file into streams, which result in chunks getting flushed because the stream is idle or hits the max age before being full. As of Loki 1.4.0, there is a metric which can help you understand why chunks are flushed `sum by (reason) (rate(loki_ingester_chunks_flushed_total{cluster="dev"}[1m]))`.

It's not critical that every chunk be full when flushed, but it will improve many aspects of operation. As such, our current guidance here is to avoid dynamic labels as much as possible and instead favor filter expressions. For example, don't add a `level` dynamic label, just `|= "level=debug"` instead.

Here are some best practices for using dynamic labels with Loki:

- Ensure the labels have low cardinality, ideally limited to tens of values.
- Use labels with long-lived values, such as the initial segment of an HTTP path: `/load`, `/save`, `/update`.
  - Do not extract ephemeral values like a trace ID or an order ID into a label; the values should be static, not dynamic.
- Only add labels that users will frequently use in their queries.
  - Don't increase the size of the index and fragment your log streams if nobody is actually using these labels. This will degrade performance.

Label values must always be bounded

If you are dynamically setting labels, never use a label which can have unbounded or infinite values. This will always result in big problems for Loki.

Try to keep values bounded to as small a set as possible. We don't have perfect guidance as to what Loki can handle, but think single digits, or maybe 10's of values for a dynamic label. This is less critical for static labels. For example, if you have 1,000 hosts in your environment it's going to be just fine to have a host label with 1,000 values.

As a general rule, you should try to keep any single tenant in Loki to less than **100,000 active streams**, and less than a million streams in a 24-hour period. These values are for HUGE tenants, sending more than **10 TB** a day. If your tenant is 10x smaller, you should have at least 10x fewer labels.

Be aware of dynamic labels applied by clients

Loki has several client options: Grafana Alloy (which also supports systemd journal ingestion and TCP-based syslog ingestion), Fluentd, Fluent Bit, a Docker plugin, and more.

Each of these come with ways to configure what labels are applied to create log streams. But be aware of what dynamic labels might be applied. Use the Loki series API to get an idea of what your log streams look like and see if there might be ways to reduce streams and cardinality. Series information can be queried through the Series API, or you can use logcli.

In Loki 1.6.0 and newer the logcli series command added the `--analyze-labels` flag specifically for debugging high cardinality labels:

```
Total Streams:  25017
Unique Labels:  8

Label Name  Unique Values  Found In Streams
requestId   24653          24979
logStream   1194           25016
logGroup    140            25016
accountId   13             25016
logger      1              25017
source      1              25016
transport   1              25017
format      1              25017
```

In this example you can see the `requestId` label had a 24653 different values out of 24979 streams it was found in, this is bad!!

This is a perfect example of something which should not be a label, `requestId` should be removed as a label and instead filter expressions should be used to query logs for a specific `requestId`. For example if `requestId` is found in the log line as a key=value pair you could write a query like this: `{logGroup="group1"} |= "requestId=32422355"`

Cardinality

The cardinality of a data attribute is the number of distinct values that the attribute can have. For example, a boolean column in a database, which can only have a value of either `true` or `false` has a cardinality of 2.

High cardinality refers to a column or row in a database that can have many possible values. For an online shopping system, fields like `userId`, `shoppingCartId`, and `orderId` are often high-cardinality columns that can have hundreds of thousands of distinct values.

Other examples of high cardinality attributes include the following:

- Timestamp
- IP addresses
- Kubernetes pod names
- User ID
- Customer ID
- Trace ID

When we talk about *cardinality* in Loki we are referring to the combination of labels and values and the number of log streams they create. In Loki, the fewer labels you use, the better. This is why Loki has a default limit of 15 index labels.

High cardinality can result from using labels with a large range of possible values, **or** combining many labels, even if they have a small and finite set of values, such as combining `status_code` and `action`. A typical set of status codes (200, 404, 500) and actions (GET, POST, PUT, PATCH, DELETE) would create 15 unique streams. But, adding just one more label like `endpoint` (/cart, /products, /customers) would triple this to 45 unique streams.

To see an example of series labels and cardinality, refer to the LogCLI tutorial. As you can see, the cardinality for individual labels can be quite high, even before you begin combining labels for a particular log stream, which increases the cardinality even further.

To view the cardinality of your current labels, you can use logcli.

```
logcli series '{}' --since=1h --analyze-labels
```

Impact of high cardinality in Loki

High cardinality causes Loki to create many streams, especially when labels have many unique values, and when those values are short-lived (for example, active for seconds or minutes). This causes Loki to build a huge index, and to flush thousands of tiny chunks to the object store.

Loki was not designed or built to support high cardinality label values. In fact, it was built for exactly the opposite. It was built for very long-lived streams and very low cardinality in the labels. In Loki, the fewer labels you use, the better.

High cardinality can lead to significant performance degradation.

Avoiding high cardinality

To avoid high cardinality in Loki, you should:

- Avoid assigning labels with unbounded values, for example timestamp, trace ID, order ID.
- Prefer static labels that describe the origin or context of the log message, for example, application, namespace, environment.
- Don't assign "dynamic" labels, which are values from the log message itself, unless it is low-cardinality, or a long-lived value.
- Use structured metadata to store frequently-searched, high-cardinality metadata fields, such as customer IDs or transaction IDs, without impacting Loki's index.

{{< admonition type="note" >}} Structured metadata is a feature in Loki and Cloud Logs that allows customers to store metadata that is too high cardinality for log lines, without needing to embed that information in log lines themselves.
It is a great home for metadata which is not easily embeddable in a log line, but is too high cardinality to be used effectively as a label. Query acceleration with Blooms also utilizes structured metadata. {{< /admonition >}}

Modify default OpenTelemetry labels

When Grafana Loki first started supporting OpenTelemetry, we selected a set of default resource attributes to promote to labels. We included some labels such as k8s.pod.name to be consistent with the scrape configs we have been suggesting for many years for Grafana Alloy. However, Loki has evolved a lot over the years and with the introduction of Structured Metadata we no longer recommend indexing a few of the labels we included in the defaults as they often have very high cardinality and Structured Metadata is a much better place for them. We no longer recommend the following as default labels:

- `k8s.pod.name`
- `service.instance.id`

Because removing these resource attributes from labels would be a breaking change for existing users, they have not yet been deprecated. If you are a new user of Grafana Loki, we recommend that you modify your Grafana Alloy or OpenTelemetry Collector configuration to convert these resource attributes from index labels to structured metadata.

Alloy configuration file example

If you are using Grafana Alloy to collect logs, the following example shows how to update your `alloy.config` file to demote `k8s.pod.name` from an index label to structured metadata.

```
// discovery.kubernetes allows you to find scrape targets from Kubernetes resources.
// It watches cluster state and ensures targets are continually synced with what is currently
running in your cluster.
discovery.kubernetes "pod" {
  role = "pod"
}

// discovery.relabel rewrites the label set of the input targets by applying one or more
relabeling rules.
// If no rules are defined, then the input targets are exported as-is.
```

```
discovery.relabel "pod_logs" {
  targets = discovery.kubernetes.pod.targets

  // Label creation - "namespace" field from "__meta_kubernetes_namespace"
  rule {
    source_labels = ["__meta_kubernetes_namespace"]
    action = "replace"
    target_label = "namespace"
  }

  // Label creation - "pod" field from "__meta_kubernetes_pod_name"
  //rule {
  //  source_labels = ["__meta_kubernetes_pod_name"]
  //  action = "replace"
  //  target_label = "pod"
  //}


  // Label creation - "container" field from "__meta_kubernetes_pod_container_name"
  rule {
    source_labels = ["__meta_kubernetes_pod_container_name"]
    action = "replace"
    target_label = "container"
  }

  // Label creation -  "app" field from "__meta_kubernetes_pod_label_app_kubernetes_io_name"
  rule {
    source_labels = ["__meta_kubernetes_pod_label_app_kubernetes_io_name"]
    action = "replace"
    target_label = "app"
  }

  // Label creation -  "job" field from "__meta_kubernetes_namespace" and
"__meta_kubernetes_pod_container_name"
  // Concatenate values __meta_kubernetes_namespace/__meta_kubernetes_pod_container_name
  rule {
    source_labels = ["__meta_kubernetes_namespace", "__meta_kubernetes_pod_container_name"]
    action = "replace"
    target_label = "job"
```

```
    separator = "/"
    replacement = "$1"
  }

  // Label creation - "container" field from "__meta_kubernetes_pod_uid" and
"__meta_kubernetes_pod_container_name"
  // Concatenate values __meta_kubernetes_pod_uid/__meta_kubernetes_pod_container_name.log
  rule {
    source_labels = ["__meta_kubernetes_pod_uid", "__meta_kubernetes_pod_container_name"]
    action = "replace"
    target_label = "__path__"
    separator = "/"
    replacement = "/var/log/pods/*$1/*.log"
  }

  // Label creation -  "container_runtime" field from "__meta_kubernetes_pod_container_id"
  rule {
    source_labels = ["__meta_kubernetes_pod_container_id"]
    action = "replace"
    target_label = "container_runtime"
    regex = "^(\\S+):\\/\\/.+$"
    replacement = "$1"
  }
}

// loki.source.kubernetes tails logs from Kubernetes containers using the Kubernetes API.
loki.source.kubernetes "pod_logs" {
  targets    = discovery.relabel.pod_logs.output
  forward_to = [loki.process.pod_logs.receiver]
}

// loki.process receives log entries from other Loki components, applies one or more
processing stages,
// and forwards the results to the list of receivers in the component's arguments.
loki.process "pod_logs" {
  stage.static_labels {
      values = {
        cluster = "<CLUSTER_NAME>",
      }
```

```
  }

  forward_to = [loki.write.<WRITE_COMPONENT_NAME>.receiver]
}
```

Kubernetes Monitoring Helm chart configuration example

If you are using the Kubernetes Monitoring Helm chart to collect Kubernetes logs to send to Loki, the following example shows how to update the `values.yaml` file for your Helm chart to demote `k8s.pod.name` and `service.instance.id` from index labels to structured metadata when gathering Pod logs via the Kubernetes API (the `podLogsViaKubernetesApi` feature).

```
# Enable Pod log collection for the cluster. Collects logs from all Pods in both the meta and
loki namespaces.
podLogsViaKubernetesApi:
  enabled: true
  # collector must match a name defined under your own top-level collectors: map.
  collector: alloy-singleton
  namespaces:
    - meta
    - loki
  # This is the section that specifies to send k8s.pod.name and service.instance.id to
structured metadata
  # instead of promoting them to indexed labels. Format is <key>: <extracted_key>.
  structuredMetadata:
    k8s.pod.name: k8s.pod.name
    service.instance.id: service.instance.id
```

{{< admonition type="note" >}} Chart version 4.x replaced the old `labelsToKeep` filter list with explicit `labels:` and `annotations:` promotion maps: instead of filtering down a larger generated label set, you opt in to the specific Pod labels and annotations you want copied to log labels. By default, `podLogsViaKubernetesApi` already keeps `k8s.pod.name` and `service.instance.id` out of the indexed labels and sends them as structured metadata; the configuration above shows how to set this explicitly. For the full set of available `labels:`, `annotations:`, and `structuredMetadata:` options, refer to the feature-pod-logs-via-kubernetes-api documentation. {{< /admonition >}}

Loki configuration example

If you are running Loki in a Docker container or on-premises, here is an example of how to modify your Loki `values.yaml` file to demote `k8s.pod.name` and `service.instance.id` from index labels to structured metadata.

```
distributor:
  otlp_config:
    # List of default otlp resource attributes to be picked as index labels - EDIT TO REMOVE k8s.pod.name AND service.instance.id FROM THE LIST
    # CLI flag: -distributor.otlp.default_resource_attributes_as_index_labels
    default_resource_attributes_as_index_labels: [service.name service.namespace deployment.environment deployment.environment.name cloud.region cloud.availability_zone k8s.cluster.name k8s.namespace.name k8s.container.name container.name k8s.replicaset.name k8s.deployment.name k8s.statefulset.name k8s.daemonset.name k8s.cronjob.name k8s.job.name]
```

Kubernetes example `values.yaml`:

```
loki:
  distributor:
    otlp_config:
    # List of default otlp resource attributes to be picked as index labels - EDIT TO REMOVE k8s.pod.name AND service.instance.id FROM THE LIST
    # CLI flag: -distributor.otlp.default_resource_attributes_as_index_labels
      default_resource_attributes_as_index_labels: [service.name service.namespace deployment.environment deployment.environment.name cloud.region cloud.availability_zone k8s.cluster.name k8s.namespace.name k8s.container.name container.name k8s.replicaset.name k8s.deployment.name k8s.statefulset.name k8s.daemonset.name k8s.cronjob.name k8s.job.name]
```

What is structured metadata

{{< admonition type="warning" >}} Structured metadata was added to chunk format V4, which is used if the schema version is greater than or equal to `13`. Structured metadata also requires the `tsdb` index type; it is not supported by the deprecated `boltdb-shipper` store. See Storage schema for more details about schema versions and index types. {{< /admonition >}}

Selecting proper, low cardinality labels is critical to operating and querying Loki effectively. Some metadata, especially infrastructure related metadata, can be difficult to embed in log lines, and is too high cardinality to effectively store as indexed labels (and therefore reducing performance of the index).

Structured metadata is a way to attach metadata to logs without indexing them or including them in the log line content itself. Examples of useful metadata are kubernetes pod names, process ID's, or any other label that is often used in queries but has high cardinality and is expensive to extract at query time.

Structured metadata can also be used to query commonly needed metadata from log lines without needing to apply a parser at query time. Large json blobs or a poorly written query using complex regex patterns, for example, come with a high performance cost. Examples of useful metadata include container_IDs or user IDs.

When to use structured metadata

You should only use structured metadata in the following situations:

- If you are ingesting data in OpenTelemetry format, using Grafana Alloy or an OpenTelemetry Collector. Structured metadata was designed to support native ingestion of OpenTelemetry data.
- If you have high cardinality metadata that should not be used as a label and does not exist in the log line. Some examples might include `process_id` or `thread_id` or Kubernetes pod names.
- If you are using Logs Drilldown to visualize and explore your Loki logs. You must enable `pattern_ingester.enabled`, `allow_structured_metadata`, `volume_enabled`, and `discover_log_levels` in your Loki configuration. For the full setup, refer to Access or install Grafana Logs Drilldown.
- If you are a large-scale customer, who is ingesting more than 75TB of logs a month and are using Bloom filters (deprecated), starting in Loki 3.3 Bloom filters now utilize

structured metadata.

Enable or disable structured metadata

You enable structured metadata in the Loki config.yaml file.

```
limits_config:
  allow_structured_metadata: true
```

If you are also using Logs Drilldown, you need additional settings. The following example enables structured metadata along with the other settings required by Logs Drilldown:

```
pattern_ingester:
  enabled: true
limits_config:
  allow_structured_metadata: true
  volume_enabled: true
  discover_log_levels: true
```

You can disable Structured Metadata by setting `allow_structured_metadata: false` in the `limits_config` section or set the command line argument `-validation.allow-structured-metadata=false`. Note structured metadata is required to support ingesting OTLP data.

Attaching structured metadata to log lines

You have the option to attach structured metadata to log lines in the push payload along with each log line and the timestamp. For more information on how to push logs to Loki via the HTTP endpoint, refer to the HTTP API documentation.

Alternatively, you can use Grafana Alloy to extract and attach structured metadata to your log lines. See the Alloy `stage.structured_metadata` block for more information.

With Loki version 1.2.0, support for structured metadata has been added to the Logstash output plugin. For more information, see logstash.

{{< admonition type="warning" >}} Structured metadata size is taken into account while asserting ingestion rate limiting. Along with that, there are separate limits on how much structured metadata can be attached per log line.

```
# Maximum size accepted for structured metadata per log line.
# CLI flag: -limits.max-structured-metadata-size
[max_structured_metadata_size: <int> | default = 64KB]

# Maximum number of structured metadata entries per log line.
# CLI flag: -limits.max-structured-metadata-entries-count
[max_structured_metadata_entries_count: <int> | default = 128]
```

Log lines that exceed either limit are rejected by the distributor with an HTTP 400 response, and the discarded entries are counted in the `loki_discarded_samples_total` metric with reason `structured_metadata_too_large` or `structured_metadata_too_many`. A large OpenTelemetry log attribute such as `exception.stacktrace` can be the cause — refer to Handle large attributes such as stack traces for mitigations. {{< /admonition >}}

Querying structured metadata

Structured metadata is extracted automatically for each returned log line and added to the labels returned for the query. You can use labels of structured metadata to filter log line using a label filter expression.

For example, if you have a label `pod` attached to some of your log lines as structured metadata, you can filter log lines using:

```
{job="example"} | pod="myservice-abc1234-56789"
```

Of course, you can filter by multiple labels of structured metadata at the same time:

```
{job="example"} | pod="myservice-abc1234-56789" | trace_id="0242ac120002"
```

Note that since structured metadata is extracted automatically to the results labels, some metric queries might return an error like `maximum of series (50000) reached for a single query`. You can use the Keep and Drop stages to filter out labels that you don't need. For example:

```
count_over_time({job="example"} | trace_id="0242ac120002" | keep job  [5m])
```

Quickstart to run Loki locally

Quickstart to run Loki locally

If you want to experiment with Loki, you can run Loki locally using the Docker Compose file that ships with Loki. It runs Loki in the simple scalable deployment mode and includes a sample application to generate logs. Simple Scalable Deployment (SSD) mode is deprecated and scheduled for removal in Loki 4.0; for production deployments, see deployment modes for currently recommended options.

The Docker Compose configuration runs the following components, each in its own container:

- **flog**: which generates log lines. flog is a log generator for common log formats.
- **Grafana Alloy**: which scrapes the log lines from flog, and pushes them to Loki through the gateway.
- **Gateway** (nginx) which receives requests and redirects them to the appropriate container based on the request's URL.
- **Loki read component**: which runs a Query Frontend and a Querier.
- **Loki write component**: which runs a Distributor and an Ingester.
- **Loki backend component**: which runs an Index Gateway, Compactor, Ruler, Bloom Planner (deprecated), Bloom Builder (deprecated), and Bloom Gateway (deprecated).
- **Minio**: which Loki uses to store its index and chunks.
- **Grafana**: which provides visualization of the log lines captured within Loki.

{{< figure max-width="75%" src="/media/docs/loki/get-started-flog-v3.png" caption="Getting started sample application" alt="Getting started sample application" >}}

Before you begin

Before you start, you need to have the following installed on your local system:

- Install Docker
- Install Docker Compose

{{< admonition type="tip" >}} Alternatively, you can try out this example in our interactive learning environment: Loki Quickstart Sandbox.

It's a fully configured environment with all the dependencies already installed.

Interactive

Provide feedback, report bugs, and raise issues in the Grafana Killercoda repository. {{< /admonition >}}

Install Loki and collecting sample logs

{{< admonition type="note" >}} This quickstart assumes you are running Linux. {{< /admonition >}}

To install Loki locally, follow these steps:

1. Create a directory called `evaluate-loki` for the demo environment. Make `evaluate-loki` your current working directory:

   ```
   mkdir evaluate-loki
   cd evaluate-loki
   ```

2. Download `loki-config.yaml`, `alloy-local-config.yaml`, and `docker-compose.yaml`: {{< tabs >}} {{< tab-content name="wget" >}}

   ```
   wget https://raw.githubusercontent.com/grafana/loki/main/examples/getting-started/loki-config.yaml -O loki-config.yaml
   wget https://raw.githubusercontent.com/grafana/loki/main/examples/getting-started/alloy-local-config.yaml -O alloy-local-config.yaml
   wget https://raw.githubusercontent.com/grafana/loki/main/examples/getting-started/docker-compose.yaml -O docker-compose.yaml
   ```

   {{< /tab-content >}} {{< tab-content name="curl" >}}

   ```
   curl https://raw.githubusercontent.com/grafana/loki/main/examples/getting-started/loki-config.yaml --output loki-config.yaml
   curl https://raw.githubusercontent.com/grafana/loki/main/examples/getting-started/alloy-local-config.yaml --output alloy-local-config.yaml
   curl https://raw.githubusercontent.com/grafana/loki/main/examples/getting-started/docker-compose.yaml --output docker-compose.yaml
   ```

   {{< /tab-content >}} {{< /tabs >}}

{{< docs/ignore >}}

```
wget https://raw.githubusercontent.com/grafana/loki/main/examples/getting-started/loki-config.yaml -O loki-config.yaml
wget https://raw.githubusercontent.com/grafana/loki/main/examples/getting-started/alloy-local-config.yaml -O alloy-local-config.yaml
wget https://raw.githubusercontent.com/grafana/loki/main/examples/getting-started/docker-compose.yaml -O docker-compose.yaml
```

{{< /docs/ignore >}}

3. Deploy the sample Docker image.
   With `evaluate-loki` as the current working directory, start the demo environment using `docker compose`:

   ```
   docker compose up -d
   ```

   At the end of the command, you should see something similar to the following:

   ```
   ✔ Network evaluate-loki_loki          Created      0.1s
   ✔ Container evaluate-loki-minio-1     Started      0.6s
   ✔ Container evaluate-loki-flog-1      Started      0.6s
   ✔ Container evaluate-loki-backend-1   Started      0.8s
   ✔ Container evaluate-loki-write-1     Started      0.8s
   ✔ Container evaluate-loki-read-1      Started      0.8s
   ✔ Container evaluate-loki-gateway-1   Started      1.1s
   ✔ Container evaluate-loki-grafana-1   Started      1.4s
   ✔ Container evaluate-loki-alloy-1     Started      1.4s
   ```

4. (Optional) Verify that the Loki cluster is up and running.
   - The read component returns `ready` when you browse to http://localhost:3101/ready. The message `Query Frontend not ready: not ready: number of schedulers this worker is connected to is 0` shows until the read component is ready.
   - The write component returns `ready` when you browse to http://localhost:3102/ready. The message `Ingester not ready: waiting for 15s after being ready` shows until the write component is ready.
5. (Optional) Verify that Grafana Alloy is running.
   - You can access the Grafana Alloy UI at http://localhost:12345.
6. (Optional) You can check all the containers are running by running the following command:

   ```
   docker ps -a
   ```

View your logs in Grafana

After you have collected logs, you will want to view them. You can view your logs using the command line interface, LogCLI, but the easiest way to view your logs is with Grafana.

1. Use Grafana to query the Loki data source.

   The test environment includes Grafana, which you can use to query and observe the sample logs generated by the flog application.

   You can access the Grafana cluster by browsing to http://localhost:3000.

   The Grafana instance in this demonstration has a Loki data source already configured.
   {{< figure src="/media/docs/loki/grafana-query-builder-v2.png" caption="Grafana Explore" alt="Grafana Explore" >}}
2. From the Grafana main menu, click the **Explore** icon (1) to open the Explore tab.

   To learn more about Explore, refer to the Explore documentation.
3. From the menu in the dashboard header, select the Loki data source (2).
   This displays the Loki query editor.

   In the query editor you use the Loki query language, LogQL, to query your logs. To learn more about the query editor, refer to the query editor documentation.
4. The Loki query editor has two modes (3):
   - Builder mode, which provides a visual query designer.
   - Code mode, which provides a feature-rich editor for writing LogQL queries.

   Next we'll walk through a few simple queries using both the builder and code views.
5. Click **Code** (3) to work in Code mode in the query editor.
   Here are some sample queries to get you started using LogQL. These queries assume that you followed the instructions to create a directory called `evaluate-loki`.
   If you installed in a different directory, you'll need to modify these queries to match your installation directory.
   After copying any of these queries into the query editor, click **Run Query** (4) to execute the query.
   1. View all the log lines which have the container label `evaluate-loki-flog-1`:

      ```
      {container="evaluate-loki-flog-1"}
      ```

      In Loki, this is a log stream.

      Loki uses labels as metadata to describe log streams.
      Loki queries always start with a label selector. In the previous query, the label selector is `{container="evaluate-loki-flog-1"}`.
   2. To view all the log lines which have the container label `evaluate-loki-grafana-1`:

      ```
      {container="evaluate-loki-grafana-1"}
      ```

3. Find all the log lines in the `{container="evaluate-loki-flog-1"}` stream that contain the string `status`:

   ```
   {container="evaluate-loki-flog-1"} |= `status`
   ```

4. Find all the log lines in the `{container="evaluate-loki-flog-1"}` stream where the JSON field `status` has the value `404`:

   ```
   {container="evaluate-loki-flog-1"} | json | status=`404`
   ```

5. Calculate the number of logs per second where the JSON field `status` has the value `404`:

   ```
   sum by(container) (rate({container="evaluate-loki-flog-1"} | json | status=`404`
   [$__auto]))
   ```

   The final query is a metric query which returns a time series. This makes Grafana draw a graph of the results.
   You can change the type of graph for a different view of the data. Click **Bars** to view a bar graph of the data.

6. Click the **Builder** tab (3) to return to builder mode in the query editor.
   1. In builder mode, click **Kick start your query** (5).
   2. Expand the **Log query starters** section.
   3. Select the first choice, **Parse log lines with logfmt parser**, by clicking **Use this query**.
   4. On the Explore tab, click **Label browser**, in the dialog select a container and click **Show logs**.

For a thorough introduction to LogQL, refer to the LogQL reference.

Sample queries (code view)

Here are some more sample queries that you can run using the Flog sample data.

To see all the log lines that flog has generated, enter the LogQL query:

```
{container="evaluate-loki-flog-1"}
```

The flog app generates log lines for simulated HTTP requests.

To see all `GET` log lines, enter the LogQL query:

```
{container="evaluate-loki-flog-1"} |= "GET"
```

To see all `POST` methods, enter the LogQL query:

```
{container="evaluate-loki-flog-1"} |= "POST"
```

To see every log line with a 401 status (unauthorized error), enter the LogQL query:

```
{container="evaluate-loki-flog-1"} | json | status="401"
```

To see every log line that doesn't contain the text `401`:

```
{container="evaluate-loki-flog-1"} != "401"
```

For more examples, refer to the query documentation.

Loki data source in Grafana

In this example, the Loki data source is already configured in Grafana. This can be seen within the `docker-compose.yaml` file:

```
  grafana:
    image: grafana/grafana:latest
    environment:
      - GF_PATHS_PROVISIONING=/etc/grafana/provisioning
      - GF_AUTH_ANONYMOUS_ENABLED=true
      - GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
    depends_on:
      - gateway
    entrypoint:
      - sh
      - -euc
      - |
        mkdir -p /etc/grafana/provisioning/datasources
        cat <<EOF > /etc/grafana/provisioning/datasources/ds.yaml
        apiVersion: 1
        datasources:
          - name: Loki
            type: loki
            access: proxy
            url: http://gateway:3100
```

```
      jsonData:
        httpHeaderName1: "X-Scope-OrgID"
      secureJsonData:
        httpHeaderValue1: "tenant1"
    EOF
    /run.sh
```

Within the entrypoint section, the Loki data source is configured with the following details:

- `Name: Loki` (name of the data source)
- `Type: loki` (type of data source)
- `Access: proxy` (access type)
- `URL: http://gateway:3100` (URL of the Loki data source. Loki uses an nginx gateway to direct traffic to the appropriate component)
- `jsonData.httpHeaderName1: "X-Scope-OrgID"` (header name for the organization ID)
- `secureJsonData.httpHeaderValue1: "tenant1"` (header value for the organization ID)

It is important to note when Loki is configured in any other mode other than monolithic deployment, you are required to pass a tenant ID in the header. Without this, queries will return an authorization error.

Complete metrics, logs, traces, and profiling example

You have completed the Loki Quickstart demo. So where to go next?

{{< docs/ignore >}}

Back to docs

Head back to where you started from to continue with the Loki documentation: Loki documentation . {{< /docs/ignore >}}

If you would like to run a demonstration environment that includes Mimir, Loki, Tempo, and Grafana, you can use Introduction to Metrics, Logs, Traces, and Profiling in Grafana. It's a self-contained environment for learning about Mimir, Loki, Tempo, and Grafana.

The project includes detailed explanations of each component and annotated configurations for a single-instance deployment. You can also push the data from the environment to Grafana Cloud.

Loki Tutorial

Loki Tutorial

This quickstart guide will walk you through deploying Loki in single binary mode (also known as monolithic mode) using Docker Compose. Grafana Loki is only one component of the Grafana observability stack for logs. In this tutorial we will refer to this stack as the **Loki stack**.

{{< figure max-width="100%" src="/media/docs/loki/getting-started-loki-stack-3.png" caption="Loki stack" alt="Loki stack" >}}

The Loki stack consists of the following components:

- **Alloy**: Grafana Alloy is an open source telemetry collector for metrics, logs, traces, and continuous profiles. In this quickstart guide Grafana Alloy has been configured to tail logs from all Docker containers and forward them to Loki.
- **Loki**: A log aggregation system to store the collected logs. For more information on what Loki is, see the Loki overview.
- **Grafana**: Grafana is an open-source platform for monitoring and observability. Grafana will be used to query and visualize the logs stored in Loki.

Before you begin

Before you start, you need to have the following installed on your local system:

- Install Docker
- Install Docker Compose

{{< admonition type="tip" >}} Alternatively, you can try out this example in our interactive learning environment: Loki Quickstart Sandbox.

It's a fully configured environment with all the dependencies already installed.

Interactive

Provide feedback, report bugs, and raise issues in the Grafana Killercoda repository. {{< /admonition >}}

Deploy the Loki stack

{{< admonition type="note" >}} This quickstart assumes you are running Linux or MacOS. Windows users can follow the same steps using Windows Subsystem for Linux. {{< /admonition >}}

To deploy the Loki stack locally, follow these steps:

1. Clone the Loki fundamentals repository and check out the getting-started branch:

   ```
   git clone https://github.com/grafana/loki-fundamentals.git -b getting-started
   ```

2. Change to the `loki-fundamentals` directory:

   ```
   cd loki-fundamentals
   ```

3. With `loki-fundamentals` as the current working directory deploy Loki, Alloy, and Grafana using Docker Compose:

   ```
   docker compose up -d
   ```

   After running the command, you should see a similar output:

   ```
   ✔ Container loki-fundamentals-grafana-1  Started  0.3s
   ✔ Container loki-fundamentals-loki-1     Started  0.3s
   ✔ Container loki-fundamentals-alloy-1    Started  0.4s
   ```

4. With the Loki stack running, you can now verify each component is up and running:
   - **Alloy**: Open a browser and navigate to http://localhost:12345/graph. You should see the Alloy UI.
   - **Grafana**: Open a browser and navigate to http://localhost:3000. You should see the Grafana home page.
   - **Loki**: Open a browser and navigate to http://localhost:3100/metrics. You should see the Loki metrics page.

Since Grafana Alloy is configured to tail logs from all Docker containers, Loki should already be receiving logs. The best place to verify log collection is using the Grafana Logs Drilldown feature.

To do this, navigate to http://localhost:3000/drilldown. Select **Logs**. You should see the Grafana Logs Drilldown page.

{{< figure max-width="100%" src="/media/docs/loki/get-started-drill-down.png" caption="Grafana Logs Drilldown" alt="Grafana Logs Drilldown" >}}

If you have only the getting started demo deployed in your Docker environment, you should see three containers and their logs; `loki-fundamentals-alloy-1`, `loki-fundamentals-grafana-1` and `loki-fundamentals-loki-1`. In the `loki-fundamentals-loki-1` container, click **Show Logs** to drill down into the logs for that container.

{{< figure max-width="100%" src="/media/docs/loki/get-started-drill-down-container.png" caption="Grafana Drilldown Service View" alt="Grafana Drilldown Service View" >}}

We will not cover the rest of the Grafana Logs Drilldown features in this quickstart guide. For more information on how to use the Grafana Logs Drilldown feature, refer to Get started with Grafana Logs Drilldown.

Collect logs from a sample application

Currently, the Loki stack is collecting logs about itself. To provide a more realistic example, you can deploy a sample application that generates logs. The sample application is called **The Carnivorous Greenhouse**, a microservices application that allows users to login and simulate a greenhouse with carnivorous plants to monitor. The application consists of seven services:

- **User Service:** Manages user data and authentication for the application. Such as creating users and logging in.
- **Plant Service:** Manages the creation of new plants and updates other services when a new plant is created.
- **Simulation Service:** Generates sensor data for each plant.
- **WebSocket Service:** Manages the websocket connections for the application.
- **Bug Service:** A service that when enabled, randomly causes services to fail and generate additional logs.
- **Main App:** The main application that ties all the services together.
- **Database:** A PostgreSQL database that stores user and plant data.

The architecture of the application is shown below:

{{< figure max-width="100%" src="/media/docs/loki/get-started-architecture.png" caption="Sample Microservice Architecture" alt="Sample Microservice Architecture" >}}

To deploy the sample application, follow these steps:

1. With `loki-fundamentals` as the current working directory, deploy the sample application using Docker Compose:

   ```
   docker compose -f greenhouse/docker-compose-micro.yml up -d --build
   ```

   {{< admonition type="note" >}} This may take a few minutes to complete since the images for the sample application need to be built. Go grab a coffee and come back. {{< /admonition >}}
   Once the command completes, you should see a similar output:

   ```
   ✔ Container greenhouse-websocket_service-1   Started   0.7s
   ✔ Container greenhouse-db-1                  Started   0.7s
   ✔ Container greenhouse-user_service-1        Started   0.8s
   ✔ Container greenhouse-bug_service-1         Started   0.8s
   ✔ Container greenhouse-plant_service-1       Started   0.8s
   ✔ Container greenhouse-simulation_service-1  Started   0.7s
   ✔ Container greenhouse-main_app-1            Started   0.7s
   ```

2. To verify the sample application is running, open a browser and navigate to http://localhost:5005. You should see the login page for the Carnivorous Greenhouse application.
   Now that the sample application is running, run some actions in the application to generate logs. Here is a list of actions:
   1. **Create a user:** Click **Sign Up** and create a new user. Add a username and password and click **Sign Up**.
   2. **Login:** Use the username and password you created to login. Add the username and password and click **Login**.
   3. **Create a plant:** Once logged in, give your plant a name, select a plant type and click **Add Plant**. Do this a few times if you like.

Your greenhouse should look something like this:

{{< figure max-width="100%" src="/media/docs/loki/get-started-greenhouse.png" caption="Greenhouse Dashboard" alt="Greenhouse Dashboard" >}}

Now that you have generated some logs, you can return to the Grafana Logs Drilldown page http://localhost:3000/drilldown. You should see seven new services such as `greenhouse-main_app-1`, `greenhouse-plant_service-1`, `greenhouse-user_service-1`, etc.

Querying logs

At this point, you have viewed logs using the Grafana Logs Drilldown feature. In many cases this will provide you with all the information you need. However, we can also manually query Loki to

ask more advanced questions about the logs. This can be done via **Grafana Explore**.

1. Open a browser and navigate to http://localhost:3000 to open Grafana.
2. From the Grafana main menu, click the **Explore** icon (1) to open the Explore tab.

   To learn more about Explore, refer to the Explore documentation.
   {{< figure src="/media/docs/loki/grafana-query-builder-v2.png" caption="Grafana Explore" alt="Grafana Explore" >}}
3. From the menu in the dashboard header, select the Loki data source (2).
   This displays the Loki query editor.

   In the query editor you use the Loki query language, LogQL, to query your logs. To learn more about the query editor, refer to the query editor documentation.
4. The Loki query editor has two modes (3):
   - Builder mode, which provides a visual query designer.
   - Code mode, which provides a feature-rich editor for writing LogQL queries.

   Next we'll walk through a few queries using the code view.
5. Click **Code** (3) to work in Code mode in the query editor.
   Here are some sample queries to get you started using LogQL. After copying any of these queries into the query editor, click **Run Query** (4) to execute the query.
   1. View all the log lines which have the `container` label value `greenhouse-main_app-1`:

      ```
      {container="greenhouse-main_app-1"}
      ```

      In Loki, this is a log stream.

      Loki uses labels as metadata to describe log streams.
      Loki queries always start with a label selector. In the previous query, the label selector is `{container="greenhouse-main_app-1"}`.
   2. Find all the log lines in the `{container="greenhouse-main_app-1"}` stream that contain the string `POST`:

      ```
      {container="greenhouse-main_app-1"} |= "POST"
      ```

Extracting attributes from logs

Loki by design does not force log lines into a specific schema format. Whether you are using JSON, key-value pairs, plain text, Logfmt, or any other format, Loki ingests these logs lines as a stream of characters. The sample application we are using stores logs in Logfmt format:

```
ts=2025-02-21 16:09:42,176 level=INFO line=97 msg="192.168.65.1 - - [21/Feb/2025 16:09:42]
"GET /static/style.css HTTP/1.1" 304 -"
```

To break this down:

- `ts=2025-02-21 16:09:42,176` is the timestamp of the log line.
- `level=INFO` is the log level.
- `line=97` is the line number in the code.
- `msg="192.168.65.1 - - [21/Feb/2025 16:09:42] "GET /static/style.css HTTP/1.1" 304 -"` is the log message.

When querying Loki, you can pipe the result of the label selector through a parser. This extracts attributes from the log line for further processing. For example, lets pipe `{container="greenhouse-main_app-1"}` through the `logfmt` parser to extract the `level` and `line` attributes:

```
{container="greenhouse-main_app-1"} | logfmt
```

When you now expand a log line in the query result, you will see the extracted attributes.

{{< admonition type="tip" >}} Before we move on to the next section, let's generate some error logs. To do this, enable the bug service in the sample application. This is done by setting the `Toggle Error Mode` to `On` in the Carnivorous Greenhouse application. This will cause the bug service to randomly cause services to fail. {{< /admonition >}}

Advanced and Metrics Queries

With Error Mode enabled the bug service will start causing services to fail, in these next few LogQL examples we will track down some of these errors. Lets start by parsing the logs to extract the `level` attribute and then filter for logs with a `level` of `ERROR`:

```
{container="greenhouse-plant_service-1"} | logfmt | level="ERROR"
```

This query will return all the logs from the `greenhouse-plant_service-1` container that have a `level` attribute of `ERROR`. You can further refine this query by filtering for a specific code line:

```
{container="greenhouse-plant_service-1"} | logfmt | level="ERROR", line="58"
```

This query will return all the logs from the `greenhouse-plant_service-1` container that have a `level` attribute of `ERROR` and a `line` attribute of `58`.

LogQL also supports metrics queries. Metrics are useful for abstracting the raw log data aggregating attributes into numeric values. This allows you to utilize more visualization options in Grafana as well as generate alerts on your logs.

For example, you can use a metric query to count the number of logs per second that have a specific attribute:

```
sum(rate({container="greenhouse-plant_service-1"} | logfmt | level="ERROR" [$__auto]))
```

It worth changing the visualization from `lines` to `bars` to visualize the error rate over time since the error count is quite low.

Another example is to get the top 10 services producing the highest rate of errors:

```
topk(10, sum(rate({env="production"} | logfmt | level="ERROR" [5m])) by (service_name))
```

{{< admonition type="note" >}} `service_name` is a label created by Loki when no service name is provided in the log line. It will use the container name as the service name. A list of all automatically generated labels can be found in Labels. {{< /admonition >}}

Finally, lets take a look at the total log throughput of each container in our production environment:

```
sum by (service_name) (rate({env="production"} | logfmt [$__auto]))
```

This is made possible by the `service_name` label and the `env` label that we have added to our log lines. Note that `env` is a static label that we added to all log lines as they are processed by Alloy.

A look under the hood

At this point you will have a running Loki stack and a sample application generating logs. You have also queried Loki using Grafana Logs Drilldown and Grafana Explore. In this next section we will take a look under the hood to understand how the Loki stack has been configured to collect logs, the Loki configuration file, and how the Loki data source has been configured in Grafana.

Grafana Alloy configuration

Grafana Alloy is collecting logs from all the Docker containers and forwarding them to Loki. It needs a configuration file to know which logs to collect and where to forward them to. Within the `loki-fundamentals` directory, you will find a file called `config.alloy`:

```
// This component is responsible for discovering new containers within the Docker environment
discovery.docker "getting_started" {
	host             = "unix:///var/run/docker.sock"
	refresh_interval = "5s"
}

// This component is responsible for relabeling the discovered containers
discovery.relabel "getting_started" {
```

```
  targets = []

  rule {
    source_labels = ["__meta_docker_container_name"]
    regex         = "/(.*)"
    target_label  = "container"
  }
}

// This component is responsible for collecting logs from the discovered containers
loki.source.docker "getting_started" {
  host             = "unix:///var/run/docker.sock"
  targets          = discovery.docker.getting_started.targets
  forward_to       = [loki.process.getting_started.receiver]
  relabel_rules    = discovery.relabel.getting_started.rules
  refresh_interval = "5s"
}

// This component is responsible for processing the logs (In this case adding static labels)
loki.process "getting_started" {
    stage.static_labels {
    values = {
      env = "production",
    }
}
    forward_to = [loki.write.getting_started.receiver]
}

// This component is responsible for writing the logs to Loki
loki.write "getting_started" {
  endpoint {
    url  = "http://loki:3100/loki/api/v1/push"
  }
}

// Enables the ability to view logs in the Alloy UI in realtime
livedebugging {
  enabled = true
}
```

This configuration file can be viewed visually via the Alloy UI at http://localhost:12345/graph.

{{< figure max-width="100%" src="/media/docs/loki/getting-started-alloy-ui.png" caption="Alloy UI" alt="Alloy UI" >}}

In this view you can see the components of the Alloy configuration file and how they are connected:

- **discovery.docker**: This component queries the metadata of the Docker environment via the Docker socket and discovers new containers, as well as providing metadata about the containers.
- **discovery.relabel**: This component converts a metadata (`__meta_docker_container_name`) label into a Loki label (`container`).
- **loki.source.docker**: This component collects logs from the discovered containers and forwards them to the next component. It requests the metadata from the `discovery.docker` component and applies the relabeling rules from the `discovery.relabel` component.
- **loki.process**: This component provides stages for log transformation and extraction. In this case it adds a static label `env=production` to all logs.
- **loki.write**: This component writes the logs to Loki. It forwards the logs to the Loki endpoint `http://loki:3100/loki/api/v1/push`.

View Logs in realtime

Grafana Alloy provides a built-in real time log viewer. This allows you to view current log entries and how they are being transformed via specific components of the pipeline. To view live debugging mode open a browser tab and navigate to:

http://localhost:12345/debug/loki.process.getting_started.

Loki Configuration

Grafana Loki requires a configuration file to define how it should run. Within the `loki-fundamentals` directory, you will find a file called `loki-config.yaml`:

```
auth_enabled: false


server:
  http_listen_port: 3100
  grpc_listen_port: 9096
  log_level: info
  grpc_server_max_concurrent_streams: 1000
```

```
common:
  instance_addr: 127.0.0.1
  path_prefix: /tmp/loki
  storage:
    filesystem:
      chunks_directory: /tmp/loki/chunks
      rules_directory: /tmp/loki/rules
  replication_factor: 1
  ring:
    kvstore:
      store: inmemory

query_range:
  results_cache:
    cache:
      embedded_cache:
        enabled: true
        max_size_mb: 100

limits_config:
  metric_aggregation_enabled: true
  allow_structured_metadata: true
  volume_enabled: true
  retention_period: 24h   # 24h

schema_config:
  configs:
    - from: 2020-10-24
      store: tsdb
      object_store: filesystem
      schema: v13
      index:
        prefix: index_
        period: 24h

pattern_ingester:
  enabled: true
  metric_aggregation:
    loki_address: localhost:3100
```

```
ruler:
  enable_alertmanager_discovery: true
  enable_api: true

frontend:
  encoding: protobuf

compactor:
  working_directory: /tmp/loki/retention
  delete_request_store: filesystem
  retention_enabled: true
```

To summarize the configuration file:

- **auth_enabled**: This is set to `false`, meaning Loki does not need a tenant ID for ingest or query. Note that this is not recommended for production environments. When deploying the Loki Helm chart, this is set to `true` by default.
- **server**: Defines the ports Loki listens on, the log level, and the maximum number of concurrent gRPC streams.
- **common**: Defines the common configuration for Loki. This includes the instance address, storage configuration, replication factor, and ring configuration.
- **query_range**: This is configured to tell Loki to use inbuilt caching for query results. In production environments of Loki this is handled by a separate cache service such as memcached.
- **limits_config**: Defines the global limits for all Loki tenants. This includes enabling specific features such as metric aggregation and structured metadata. Limits can be defined on a per tenant basis, however this is considered an advanced configuration and for most use cases the global limits are sufficient.
- **schema_config**: Defines the schema configuration for Loki. This includes the schema version, the object store, and the index configuration.
- **pattern_ingester**: Enables pattern ingesters which are used to discover log patterns. Mostly used by Grafana Logs Drilldown.
- **ruler**: Enables the ruler component of Loki. This is used to create alerts based on log queries.
- **frontend**: Defines the encoding format for the frontend. In this case it is set to `protobuf`.
- **compactor**: Defines the compactor configuration. Used to compact the index and manage chunk retention.

The above configuration file is a basic configuration file for Loki. For more advanced configuration options, refer to the Loki Configuration documentation.

Grafana Loki Data source

The final piece of the puzzle is the Grafana Loki data source. This is used by Grafana to connect to Loki and query the logs. Grafana has multiple ways to define a data source:

- **Direct**: This is where you define the data source in the Grafana UI.
- **Provisioning**: This is where you define the data source in a configuration file and have Grafana automatically create the data source.
- **API**: This is where you use the Grafana API to create the data source.

In this case we are using the provisioning method. Instead of mounting the Grafana configuration directory, we have defined the data source in this portion of the `docker-compose.yml` file:

```
  grafana:
    image: grafana/grafana:latest
    environment:
      - GF_FEATURE_TOGGLES_ENABLE=grafanaManagedRecordingRules
      - GF_AUTH_ANONYMOUS_ORG_ROLE=Admin
      - GF_AUTH_ANONYMOUS_ENABLED=true
      - GF_AUTH_BASIC_ENABLED=false
    ports:
      - 3000:3000/tcp
    entrypoint:
      - sh
      - -euc
      - |
         mkdir -p /etc/grafana/provisioning/datasources
         cat <<EOF > /etc/grafana/provisioning/datasources/ds.yaml
         apiVersion: 1
         datasources:
         - name: Loki
           type: loki
           access: proxy
           orgId: 1
           url: 'http://loki:3100'
           basicAuth: false
           isDefault: true
           version: 1
           editable: true
         EOF
         /run.sh
```

```
    networks:
      - loki
```

The entrypoint overrides the default startup command to first create the datasource provisioning file `ds.yaml`, which configures Loki as the default datasource. It then calls `/run.sh`, the default Grafana startup script included in the `grafana/grafana` Docker image, which starts the Grafana server. Since Loki is running in the same Docker network as Grafana, the datasource URL uses the service name `loki`.

What next?

{{< docs/ignore >}}

Back to docs

Head back to where you started from to continue with the Loki documentation: Loki documentation . {{< /docs/ignore >}}

You have completed the Loki Quickstart demo. So where to go next? Here are a few suggestions:

- **Deploy:** Loki can be deployed in multiple ways. For production use cases we recommend deploying Loki via the Helm chart.
- **Send Logs:** In this example we used Grafana Alloy to collect and send logs to Loki. However there are many other methods you can use depending upon your needs. For more information see send data.
- **Query Logs:** LogQL is an extensive query language for logs and contains many tools to improve log retrieval and generate insights. For more information see the Query section.
- **Alert:** Lastly you can use the ruler component of Loki to create alerts based on log queries. For more information see Alerting.

Complete metrics, logs, traces, and profiling example

If you would like to run a demonstration environment that includes Mimir, Loki, Tempo, and Grafana, you can use Introduction to Metrics, Logs, Traces, and Profiling in Grafana. It's a self-contained environment for learning about Mimir, Loki, Tempo, and Grafana.

The project includes detailed explanations of each component and annotated configurations for a single-instance deployment. You can also push the data from the environment to Grafana Cloud.