Loki components

- Distributor
  - Validation
  - Preprocessing
  - Rate limiting
  - Forwarding
    - Replication factor
  - Hashing
  - Quorum consistency
- Ingester
  - Timestamp Ordering
  - Handoff
  - Filesystem support
- Query frontend
  - Queueing
  - Splitting
  - Caching
    - Metric queries
    - Log queries
    - Index stats queries
    - Log volume queries
- Query scheduler
- Querier
- Index Gateway
- Compactor
- Ruler
- Pattern ingester
- Kafka-based ingestion (experimental)
- Bloom Planner

- Bloom Builder
- Bloom Gateway

{{< 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.

□□□ #2
□ □□□ □□□ 2026-10-31 13:44:46 UTC
□ □□□ □□□ 2026-10-31 13:45:07 UTC