LogCLI tutorial

- LogCLI tutorial
  - Prerequisites
  - Scenario
  - Setup
    - Connecting LogCLI to Loki
  - Querying logs
    - Find all critical packages
    - Metric queries
    - Instant metric queries
    - Writing query results to a file
  - Meta queries
    - Checking series cardinality
    - Detected fields
    - Checking query performance
    - Checking the log volume
  - Queries against static log files
  - Conclusion

LogCLI tutorial

This LogCLI tutorial will walk you through the following concepts:

- Querying logs
- Meta Queries against your Loki instance
- Queries against static log files

Prerequisites

Before you begin, you need to have the following:

- Docker
- Docker-compose
- LogCLI installed on your machine (see LogCLI installation)

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

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 >}}

Scenario

You are a site manager for a new logistics company. The company uses structured logs to keep track of every shipment sent and received. The payload format looks like this:

```
{"timestamp": "2024-11-22T13:22:56.377884", "state": "New York", "city": "Buffalo", "package_id": "PKG34245", "package_type": "Documents", "package_size": "Medium", "package_status": "error", "note": "Out for delivery", "sender": {"name": "Sender27", "address": "144 Elm St, Buffalo, New York"}, "receiver": {"name": "Receiver4", "address": "260 Cedar Blvd, New York City, New York"}}
```

The logs are processed from Grafana Alloy to extract labels and structured metadata before they're stored in Loki. You have been tasked with monitoring the logs using the LogCLI and build a report on the overall health of the shipments.

Setup

To get started, we need to clone the Alloy Scenario repository and start the mail-house example:

1. Clone the repository:

   ```
   git clone https://github.com/grafana/alloy-scenarios.git
   ```

2. Start the mail-house example:

   ```
   docker compose -f alloy-scenarios/mail-house/docker-compose.yml up -d
   ```

This will start the mail-house example and expose the Loki instance at `http://localhost:3100`. We have also included a Grafana instance to verify the LogCLI results which can be accessed at http://localhost:3000.

Connecting LogCLI to Loki

To connect LogCLI to the Loki instance, you need to set the `LOKI_ADDR` environment variable:

{{< admonition type="tip" >}}

If you are running this example against your own Loki instance and have configured authentication, you will also need to set the `LOKI_USERNAME` and `LOKI_PASSWORD` environment variables.

{{< /admonition >}}

```
export LOKI_ADDR=http://localhost:3100
```

Now let's verify the connection by running the following command:

```
logcli labels
```

This should return an output similar to the following:

```
http://localhost:3100/loki/api/v1/labels?end=1732282703894072000&start=1732279103894072000
package_size
service_name
state
```

This confirms that LogCLI is connected to the Loki instance and we now know that the logs contain the following labels: `package_size`, `service_name`, and `state`. Let's run some queries against Loki to better understand our package logistics.

Querying logs

As part of our role within the logistics company, we need to build a report on the overall health of the shipments. Unfortunately, we only have access to a console and cannot use Grafana to visualize the data. We can use LogCLI to query the logs and build the report.

Find all critical packages

To find all critical packages in the last hour (the default lookback time), we can run the following query:

```
logcli query '{service_name="Delivery World"} | package_status="critical"'
```

This will return all logs where the `service_name` is `Delivery World` and the `package_status` is `critical`. The output will look similar to the following:

```
http://localhost:3100/loki/api/v1/query_range?direction=BACKWARD&end=1732617594381712000&limit
=30&query=%7Bservice_name%3D%22Delivery+World%22%7D+%7C+package_status%3D%22critical%22&start=
1732613994381712000
Common labels: {package_status="critical", service_name="Delivery World"}
2024-11-26T10:39:52Z {package_id="PKG79755", package_size="Small", state="Texas"}
{"timestamp": "2024-11-26T10:39:52.521602Z", "state": "Texas", "city": "Dallas", "package_id":
"PKG79755", "package_type": "Clothing", "package_size": "Small", "package_status": "critical",
"note": "In transit", "sender": {"name": "Sender38", "address": "906 Maple Ave, Dallas,
Texas"}, "receiver": {"name": "Receiver41", "address": "455 Pine Rd, Dallas, Texas"}}
2024-11-26T10:39:50Z {package_id="PKG34018", package_size="Large", state="Illinois"}
{"timestamp": "2024-11-26T10:39:50.510841Z", "state": "Illinois", "city": "Chicago",
"package_id": "PKG34018", "package_type": "Clothing", "package_size": "Large",
"package_status": "critical", "note": "Delayed due to weather", "sender": {"name": "Sender22",
"address": "758 Elm St, Chicago, Illinois"}, "receiver": {"name": "Receiver10", "address":
"441 Cedar Blvd, Naperville, Illinois"}}
```

Lets suppose we want to look back for the last 24 hours, we can use the `--since` flag to specify the time range:

```
logcli query --since 24h '{service_name="Delivery World"} | package_status="critical"'
```

This will query all logs for the `package_status` `critical` in the last 24 hours. However it will not return all of the logs, but only the first 30 logs. We can use the `--limit` flag to specify the number of logs to return:

```
logcli query --since 24h --limit 100 '{service_name="Delivery World"} |
package_status="critical"'
```

Metric queries

We can also use LogCLI to query logs based on metrics. For instance as part of the site report we want to count the total number of packages sent from California in the last 24 hours in 1 hour intervals. We can use the following query:

```
logcli query --since 24h 'sum(count_over_time({state="California"}[1h]))'
```

This will return a JSON object containing a list of timestamps (Unix format) and the number of packages sent from California in 1 hour intervals. Since we summing the count of logs over time, we will see the total number of logs steadily increase over time. The output will look similar to the following:


```
[
  {
    "metric": {},
    "values": [
      [
        1733913765,
        "46"
      ],
      [
        1733914110,
        "114"
      ],
      [
        1733914455,
        "179"
      ],
      [
        1733914800,
        "250"
      ],
      [
        1733915145,
        "318"
      ],
      [
        1733915490,
        "392"
```

```
      ],
      [
        1733915835,
        "396"
      ]
    ]
  }
]
```

We can take this a step further and filter the logs based on the `package_type` label. For instance, we can count the number of documents sent from California in the last 24 hours in 1 hour intervals:

```
logcli query --since 24h  'sum(count_over_time({state="California"}| json | package_type=
"Documents" [1h]))'
```

This will return a similar JSON object above but will only show a trend of the number of documents sent from California in 1 hour intervals.

Instant metric queries

Instant metric queries are a subset of metric queries that return the value of the metric at a specific point in time. This can be useful for quickly understanding an aggregate state of the stored logs.

For instance, we can use the following query to get the number of packages sent from California in the last 5 minutes:

```
logcli instant-query 'sum(count_over_time({state="California"}[5m]))'
```

This will return a result similar to the following:

```
[
  {
    "metric": {},
    "value": [
      1732702998.725,
      "58"
    ]
  }
]
```

Writing query results to a file

Another useful feature of LogCLI is the ability to write the query results to a file. This can be useful for downloading the results of our inventory report:

First we need to create a directory to store the logs:

```
mkdir -p ./inventory
```

Next we can run the following query to write the logs to the `./inventory` directory:

```
logcli query \
   --timezone=UTC \
   --output=jsonl \
   --parallel-duration="12h" \
   --parallel-max-workers="4" \
   --part-path-prefix="./inventory/inv" \
   --since=24h \
   '{service_name="Delivery World"}'
```

This will write all logs for the `service_name` `Delivery World` in the last 24 hours to the `./inventory` directory. The logs will be split into two files, each containing 12 hours of logs. Note that we do not need to specify `--limit` as this is overridden by the `--parallel-duration` flag.

Meta queries

As site managers, it's essential to maintain good data hygiene and ensure Loki operates efficiently. Understanding the labels and log volume in your logs plays a key role in this process. Beyond querying logs, LogCLI also supports meta queries on your Loki instance. Meta queries don't return log data but provide insights into the structure of your logs and the performance of your queries. The following examples demonstrate some of the core meta queries we run internally to better understand how a Loki instance is performing.

Checking series cardinality

One of the most important aspects of keeping Loki healthy is to monitor the series cardinality. This is the number of unique series in your logs. A high series cardinality can lead to performance issues and increased storage costs. We can use LogCLI to check the series cardinality of our logs.

To start let's print how many unique series we have in our logs:

```
logcli series '{}'
```

This will return a list of all the unique series in our logs. The output will look similar to the following:

```
{package_size="Small", service_name="Delivery World", state="Florida"}
{package_size="Medium", service_name="Delivery World", state="Florida"}
{package_size="Small", service_name="Delivery World", state="California"}
{package_size="Large", service_name="Delivery World", state="New York"}
{package_size="Small", service_name="Delivery World", state="Illinois"}
{package_size="Large", service_name="Delivery World", state="Florida"}
{package_size="Medium", service_name="Delivery World", state="Illinois"}
{package_size="Large", service_name="Delivery World", state="Texas"}
{package_size="Medium", service_name="Delivery World", state="California"}
{package_size="Medium", service_name="Delivery World", state="Texas"}
{package_size="Small", service_name="Delivery World", state="Texas"}
{package_size="Large", service_name="Delivery World", state="Illinois"}
{package_size="Small", service_name="Delivery World", state="New York"}
{package_size="Medium", service_name="Delivery World", state="New York"}
{package_size="Large", service_name="Delivery World", state="California"}
```

We can further improve this query by adding `--analyze-labels`:

```
logcli series '{}' --analyze-labels
```

This will return a summary of the unique values for each label in our logs. The output will look similar to the following:

```
Label Name    Unique Values  Found In Streams
state         5              15
package_size  3              15
service_name  1              15
```

Detected fields

Another useful feature of LogCLI is the ability to detect fields in your logs. This can be useful for understanding the structure of your logs and the keys that are present. This will let us detect keys which could be promoted to labels or to structured metadata.

```
logcli detected-fields --since 24h '{service_name="Delivery World"}'
```

This will return a list of all the keys detected in our logs. The output will look similar to the following:

```
label: city                   type: string  cardinality: 15
label: detected_level         type: string  cardinality: 3
label: note                   type: string  cardinality: 7
label: package_id             type: string  cardinality: 994
label: package_size_extracted type: string  cardinality: 3
label: package_status         type: string  cardinality: 4
label: package_type           type: string  cardinality: 5
label: receiver_address       type: string  cardinality: 991
label: receiver_name          type: string  cardinality: 100
label: sender_address         type: string  cardinality: 991
label: sender_name            type: string  cardinality: 100
label: state_extracted        type: string  cardinality: 5
label: timestamp              type: string  cardinality: 1000
```

You can now see why we opted to keep `package_id` in structured metadata and `package_size` as a label. Package ID has a high cardinality and is unique to each log entry, making it a good candidate for structured metadata since we potentially may need to query for it directly. Package size, on the other hand, has a low cardinality, making it a good candidate for a label.

Checking query performance

Another important aspect of keeping Loki healthy is to monitor the query performance. We can use LogCLI to check the query performance of our logs.

{{< admonition type="note" >}} The LogCLI can only return statistics for queries that touch object storage. In this example we force the Loki ingesters to flush chunks every 5 minutes which isn't recommended for production use. When running this demo if you don't see any statistics returned, try running the command again after a few minutes. {{< /admonition >}}

To start lets print the query performance of our logs:

```
logcli stats --since 24h '{service_name="Delivery World"}'
```

This will provide a JSON object containing statistics on the amount of data queried. The output will look similar to the following:

```
http://localhost:3100/loki/api/v1/index/stats?end=1732639430272850000&query=%7Bservice_name%3D
%22Delivery+World%22%7D&start=1732553030272850000
{
```

```
  bytes: 12MB
  chunks: 63
  streams: 15
  entries: 29529
}
```

This will return the total number of bytes queried, the number of chunks queried, the number of streams queried, and the number of entries queried. If we narrow down the query by specifying a secondary label, we can see the performance of the query:

```
logcli stats --since 24h '{service_name="Delivery World", package_size="Large"}'
```

This will return the statistics for the logs where the `service_name` is `Delivery World` and the `package_size` is `Large`. The output will look similar to the following:

```
{
  bytes: 4.2MB
  chunks: 22
  streams: 5
  entries: 10198
}
```

As you can see, we touched far fewer streams and entries by narrowing down the query.

Checking the log volume

We may also want to check the log volume in our logs. This can be useful for understanding the amount of data being ingested into Loki. We can use LogCLI to check the log volume in our logs.

```
logcli volume --since 24h '{service_name="Delivery World"}'
```

This returns the total number of logs ingested for the label `Delivery World` in the last 24 hours. The output will look similar to the following:

```
[
  {
    "metric": {
      "service_name": "Delivery World"
    },
    "value": [
      1732640292.354,
```

```
      "11669299"
    ]
  }
]
```

The result includes the timestamp and the total number of logs ingested.

We can also return the log volume over time by using `volume_range`:

```
logcli volume_range --since 24h --step=1h '{service_name="Delivery World"}'
```

This will provide a JSON object containing the log volume for the label `Delivery World` in the last 24 hours. `--step` will aggregate the log volume into 1 hour buckets. Note that if there are no logs for a specific hour, the log volume for that hour will not return a value.

We can even aggregate the log volume into buckets based on a specific labels value:

```
logcli volume_range --since 24h --step=1h --targetLabels='state' '{service_name="Delivery World"}'
```

This will provide a similar JSON object but will aggregate the log volume into buckets based on the `state` label value.

Queries against static log files

In addition to querying logs from Loki, LogCLI also supports querying static log files. This can be useful for querying logs that are not stored in Loki. Earlier in the tutorial we stored the logs in the `./inventory` directory. Lets run a similar query but pipe it into a log file:

```
logcli query \
   --timezone=UTC \
   --parallel-duration="12h" \
   --parallel-max-workers="4" \
   --part-path-prefix="./inventory/inv" \
   --since=24h \
   --merge-parts \
   --output=raw \
   '{service_name="Delivery World"}' > ./inventory/complete.log
```

Next lets run a query against the static log file:

```
cat ./inventory/complete.log |  logcli --stdin query '{service_name="Delivery World"} | json | package_status="critical"'
```

Note that since we are querying a static log file, labels are not automatically detected:

- `{service_name="Delivery World"}` is optional in this case but is recommended for clarity.
- `json` is required to parse the log file as JSON. This lets us extract the `package_status` field.

For example, suppose we try to query the log file without the `json` filter:

```
cat ./inventory/complete.log | logcli --stdin query '{service_name="Delivery World"} | package_status="critical"'
```

This will return no results as the `package_status` field is not detected.

Conclusion

In this tutorial as site manager for a logistics company, we have successfully used LogCLI to query logs and build a report on the overall health of the shipments. We have also used meta queries to better understand our data cleanliness and query performance. The LogCLI is a powerful tool for understanding your logs and how they are stored in Loki, as you continue to scale your solution remember to keep LogCLI in mind to monitor cardinality and query performance.

□□□ #2
□ □□□ □□□ 2026-10-31 13:47:49 UTC
□ □□□ □□□ 2026-10-31 13:47:50 UTC