Enhancing Observability in the JValue Project: Logging for a Microservice-based System

MASTER THESIS

Felix Müller

Submitted on 6 May 2024

Friedrich-Alexander-Universität Erlangen-Nürnberg
Faculty of Engineering, Department Computer Science
Professorship for Open Source Software

Supervisor:
Georg Schwarz
Prof. Dr. Dirk Riehle, M.B.A.

Friedrich-Alexander-Universität
Faculty of Engineering

Declaration of Originality

I confirm that the submitted thesis is original work and was written by me without further assistance. Appropriate credit has been given where reference has been made to the work of others. The thesis was not examined before, nor has it been published. The submitted electronic version of the thesis matches the printed version.

Erlangen, 6 May 2024

License

Erlangen, 6 May 2024

Abstract

This thesis discusses the integration of a log-management and request tracing system into the JValue microservice application using industry standards and open source solutions.
For the log-management, Fluent Bit and Data Prepper are used to collect and aggregate the logs, while OpenSearch is storing them as well as providing the UI for querying said logs. Through the use of pipelines, the logs are enriched and then separated by the kind of application they were generated from.
The request tracing is reusing Data Prepper and OpenSearch for aggregation, storage and UI, but uses OpenTelemetry for trace generation and collection. OpenTelemetry is injected into the JValue microservices using its zero-code instrumentation approach in order to avoid any code modification.
All of these services are deployed to a pre-existing Kubernetes cluster using Helm.

Contents

List of Figures

List of Tables

Acronyms

AWS Amazon Web Services

CRD Custom Resource Definition

ISM Index State Management

K8s Kubernetes

ML Machine Learning

OTEL OpenTelemetry

OTLP OpenTelemetry Protocol

UI User Interface

regex regular expression

1 Introduction

In the microservice landscape one big challenge is the lack of observability caused by the missing debugger. A first step towards improving said observability is the implementation of a log-management system. A good log-management is necessary in order to effectively discover, analyse and debug issues in a system.
Log-management means the collection and aggregation of logs of all the microservices and storing them in a centralized location where they can be analysed, searched and filtered. It also includes a mechanism for log retention.

For the parsing and aggregation of the logs to work as smoothly as possible without the need to adjust every time a new property is added to the logs, structured logging is required. Structured logging refers to logs using a structured format like JSON or XML.

Another big issue, particularly for microservices, is the missing context between separate log entries, especially across different microservices. For example identifying which logs of all the different microservices are part of the same REST call triggered by a user. This can be solved by request tracing, which traces a request by injecting a trace context into all requests.

This thesis will demonstrate a state-of-the-art solution for improving microservice-based system's observability by using current industry standards to implement such a system into the JValue project[1].

1.1 JValue

JValue is a project currently being developed by the Professorship of Open-Source Software of the Friedrich-Alexander-University in Erlangen. The project's goal is the implementation of a platform allowing data scientists an easy way to collaborate on open data projects.

The system currently consists out of the JValue Hub and a domain-specific language called Jayvee. The thesis will focus on the Hub, which consists out of

[1] https://jvalue.org/

the User Interface (UI) and the backend of the project. The Hub is made out of four microservices, all of them implemented using NodeJS and the NestJS[2] framework. The UI is implemented using React, with everything located inside a monorepo managed using Nx[3]. All applications are deployed to a Kubernetes cluster using Helm[4].

1.2 Thesis structure

The thesis starts with the short introduction into the JValue project and introduces log-management and request tracing. The second chapter lists all collected requirements, separated into functional and non-functional. Based upon the requirements, the next chapter contains the literature review done into the subject matter, focusing onto the selected solutions. The next chapter is dedicated to the design of the system, with the fifth chapter containing its implementation. The chapter afterwards contains an evaluation performed with some JValue developers as well as highlight some issues encountered during the implementation. The thesis closes with a conclusion containing some suggestions for future development.

[2]https://nestjs.com/
[3]https://nx.dev/
[4]https://helm.sh/

2 Requirements

This chapter lists the requirements for the system implemented over the course of the thesis. The requirements were created based on a meeting with three JValue developers. During this meeting the developers were asked nine questions which were then discussed. Afterwards the requirements were defined. The chapter is split into functional and non-functional requirements, starting with the functional ones.

2.1 Functional requirements

In total there are three functional requirements that were extracted from the meeting.

The first requirement is the implementation of a separate logging library. This library should be usable by all microservices with minimal code changes and contain most of the configuration in order to avoid duplicate code. The logs generated by the logger of the library should contain at least some kind of application version along with the standard timestamp, message, logging context and, in case of error logs, stack trace. Additionally the logs should contain Kubernetes metadata in case the application is running on a K8s cluster. This metadata has to contain at least the K8s pod name, namespace, container name, container id, container version and the K8s cluster node name the pod is running on.

The next requirement is that the system should collect logs not only from the JValue microservices, but also from Kubernetes system component pods. Kubernetes system component pods are pods running system components of Kubernetes, for example the Kubernetes scheduler, controller manager and API server.

Lastly a request tracing system should be implemented that can trace requests through the different JValue microservices. This request tracing should work without any direct code modifications to the JValue applications and is only required to trace the edge of the applications. This means that there is no need to trace request inside the application, just whenever a request reaches or leaves it. For example all HTTP(S) requests should be traced such that it is possible to

match any incoming request to one microservice to the corresponding outgoing request of the other microservice.

2.2 Non-Functional requirements

During the meeting three non-functional requirements were discovered alongside the functional ones.

Starting with the requirement that any system implemented during the thesis can be hosted on the Kubernetes cluster JValue Hub is running on. Additionally it should also use Helm for its deployment, same as JValue Hub.

The second requirement is that any implemented system, log-management or otherwise, has to be Open-Source. Additionally it was decided to avoid (A)GPLv3 where possible, due to uncertainty regarding the license the logs would fall under.

The last requirement is that the implemented system should be accessible via a single UI. This means that there should be only one UI for both log-management and request tracing.

3 Literature review

This chapter discusses literature relevant for the thesis. It starts with some information about how logging in Kubernetes works, followed by an overview of different log-management and request tracing solutions. The main focus however is on the selected log-management and request tracing solutions.

3.1 Kubernetes Logging

As described in T. K. Authors (2024) Kubernetes differentiates between two types of logging architectures, **node-level logging** and **cluster-level logging**.

The first architecture is built into K8s and describes how each node in a Kubernetes cluster manages the logs generated by their pods. In this architecture the node redirects the stdout and stderr of its pods to a local log file. However that of course requires that the container of the pods uses the widely adopted logging standard of containerized applications to write to the standard output and standard error streams (T. K. Authors, 2024). Afterwards the kubelet rotates the log file and exposes it using the *kubectl logs* command. Noteworthy is that the kubelet retains the logs of one previously terminated container, in case its pod restarts for example. However if a pod gets evicted, all its containers and logs are evicted as well. This architecture also applies to the Kubernetes system components that run inside a container. Examples are the Kubernetes scheduler, controller manager and API server.

The second architecture is not provided by Kubernetes and it involves transmitting the logs to a separate storage in order to allow analysis and querying of long term logs. However the Kubernetes documentation does suggest four different possible cluster-level architectures.
The first one uses a node logging agent, which is a separate pod that tails the log files of the node it is deployed to and transmits them to the logging backend. Because each agent pod can only read the log files of its own node, it is typically deployed as a DeamonSet in order to run on all nodes. There are a few advantages to this solution, with the first one being that it only creates one additional

pod per node. The second one is that the application pods do not require any changes for this solution to work and the last one is that the agent works with all pods whose containers write their logs to the standard output streams.
The second architecture is almost identical to the first one, the only difference is that the application pod has a second (sidecar) container responsible for streaming the application container logs to stdout/stderr. This allows adding support for parts of your application that do not write to the output streams, but for example an internal file. Because this solution is more or less an add-on to the first one, it shares the same advantages with only a slight drop in performance caused by the sidecar. Additionally it has the advantage of being able to support previously mentioned internal log files and similar internal log destinations.
The third architecture uses the sidecar approach of the second architecture to deploy a logging agent as a sidecar instead of as a DeamonSet. The agent then reads the logs of the application container and transmits them to the logging backend. This has the advantage that the agent can be tailored towards the container of the pod it is attached to, meaning it can contain custom parsing and transformation logic. However this approach consumes a lot more resources due to potentially deploying multiple agents per node. In addition to that, because the logs only leave the pod via the agent, they are no longer controlled by the kubelet and thus not readable using *kubectl logs*.
The last architecture does not use any agents or sidecars, but rather opts for the application container to transmit its logs directly to the backend. This increases the complexity of the application container and requires changes as well as restarts if the backend changes. Also because of the increased complexity the container might require more resources, though this approach does save on pods and containers.

3.2 Log-management

As detailed in Schmidt et al. (2012), there are advantages and disadvantages to both buying and building a log-management system. In the case of JValue it was decided to build our own system using free open-source software solutions. This section will detail a few of these solutions and select one of them for this project.

3.2.1 Solutions

There are a lot of different log-management solutions available, both proprietary and open-source. The following will give a rough overview over some of the most well known ones.

The first and probably most well known one is **ELK-Stack**[1]. It is developed

[1] https://www.elastic.co/elastic-stack/

by the company **Elastic** and was open-source until 2021 (Banon, 2021). **ELK** refers to **ElasticSearch** in combination with **Logstash**, **Beats** and **Kibana**. In this stack the logs are collected, aggregated and enriched by Logstash and Beats, while ElasticSearch is responsible for storage and management and Kibana can be used to explore and query the collected data (B.V., 2024b). ElasticSearch is built upon **Apache Lucene**, storing the logs inside Lucene indices for faster querying (B.V., 2024a).

The second solution is **OpenSearch**[2], which is an open-source (Apache-2.0) fork of ElasticSearch created by Amazon Web Services (AWS). For log collection and aggregation as well as UI, there are a few option that can be used. Typically used, including in the documentation, are **Fluent Bit**, **Data Prepper** and **OpenSearch Dashboards**. The first two are for collection, aggregation and enriching while OpenSearch Dashboards is an open-source fork of Kibana.

Grafana Loki[3] is a newer log management solution developed by Grafana and is open-source under AGPLv3. Loki is typically used with **Promtail**, a custom log collector developed specifically for Loki, and Grafana as its UI. As described in Labs (2024b), unlike the previous two solutions, Loki does not index its logs, but groups them into streams which are indexed with labels.

The third solution is proprietary and provided by the company **Sumo Logic**[4]. Due to its closed source nature, not much is known about its technical details, but it provides detailed log monitoring and analysis for cloud based applications supporting a wide variety of collectors.

The last solution is **Graylog Open**[5] which is also a proprietary product. It contains a UI and collector sidecars in addition to the main backend, as well as a few more "Content packs" that provide additional optional functionalities.

From the solutions, OpenSearch was selected due to the requirements, mainly that it is the only one with a non (A)GPLv3 open-source license. The following section contains some literature review about OpenSearch and its architecture.

3.2.2 OpenSearch

Starting with OpenSearch, the most important information concerning its logging architecture is how it stores its logs.
Generally speaking, when installing OpenSearch an **OpenSearch cluster** is created. As described in contributors (2024d), cluster consists out of at last one

[2]https://opensearch.org/
[3]https://grafana.com/oss/loki/
[4]https://www.sumologic.com/solutions/log-management/
[5]https://graylog.org/products/source-available/

OpenSearch node in its *single-node* configuration and multiple in its standard configuration. OpenSearch stores the logs inside **OpenSearch indices**, which are split over a configurable number of **shards** upon their creation. Each of these shards holds a subset of the documents of the OpenSearch index. This results into a basic architecture as shown in the following figure.

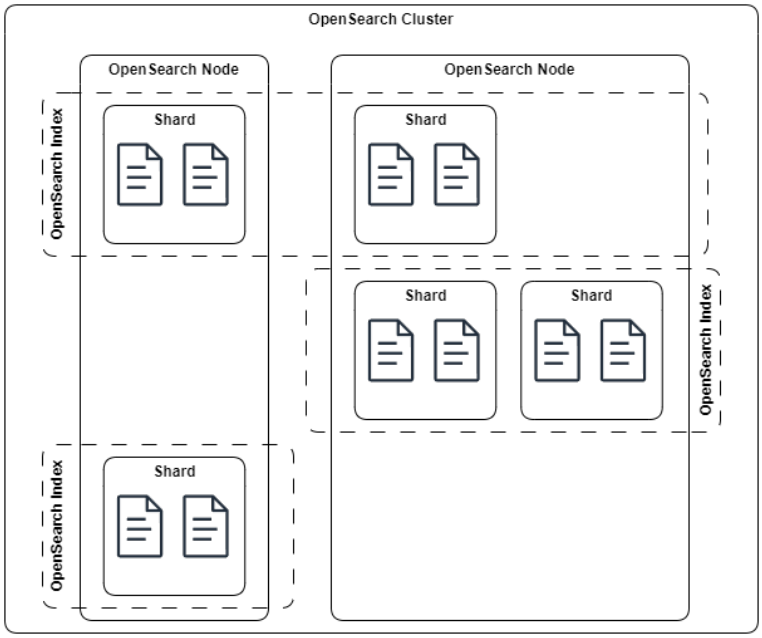

Figure 3.1: OpenSearch basic index architecture

Nodes

Concerning nodes, each node can be configured with one or more types that describes what kind of tasks that node will handle. Possible node types, as listed in contributors (2024a), are:

- **cluster-manager**: Manages cluster and its state. This includes creation of new indices, tracking the nodes of the cluster and their health as well as allocating shards to nodes.
- **cluster-manager-eligible**: Marks node as possible cluster manager, elected through voting process.
- **data**: Worker nodes storing and searching data. They handle all data related operations like indexing, searching and aggregating on their local shards. Due to this they require more disk space.
- **ingest**: Node that pre-processes data before ingestion into cluster by running the data through a pipeline.

- **coordinating**: Responsible for delegating client requests to the data nodes as well as aggregating the results into a combined response for the client.
- **dynamic**: Node assigned for specific custom work like Machine Learning (ML) in order to avoid taking resources from the data nodes.
- **search**: Responsible for providing access to searchable snapshots. Searchable snapshots are snapshots stored remotely (for example in AWS S3[6]) that can be queried despite the data not being stored in cluster storage.

By default each node is assigned to be *cluster-manager-eligible*, *data*, *ingest* and *coordinating*.

Shards
The first thing to note about shards is that, despite being part of an OpenSearch index, each shard is a fully functional Apache Lucene index on its own. Additionally shards can be categorized as either a *primary* or *replica* shard. The former one are the main shards storing the logs and other data. Replica shards contain the same data as their primary counterpart and are used to also improve the speed of search request but mainly for backups in case of node failure. Because of this replica shards are located to a different node than their primary shard. They can also be configured separately with the default being one replica for each primary shard. If, for example, an index is created with 3 shards and 1 replica shard for each primary it will result into 6 shards in total. Concerning how many shards an index should be split into, it has to be noted that more is not necessarily better in this case. This is due to, as mentioned, each shard being an Apache Lucene index which consumes CPU and memory and thus straining the cluster. A good measure is to split the index based on the estimated size with the suggestion of trying to get each shard to store 10-50GB (contributors, 2024d).

Features
As listed in contributors (2024c), OpenSearch offers a variety of features, some part of the core installation and some via optional plugins. One core feature is the ability to create **Index-** and **Component-Templates**. With these it is possible to configure indices even before their creation. Index templates, for example, allow the configuration of the number of primary and replica shards, as well as describing the properties, and their types, of the expected log data. Component templates can be used to share configurations between multiple index templates. Another core feature is the possibility to create so-called **Tenants** which are more or less like user groups. They restrict what indices and other data a user can see and access.
The third core feature are the **Index State Management (ISM) Policies**.

[6]https://aws.amazon.com/s3/

This feature allows the description of the life cycle of indices, which can involve increasing or decreasing the number of replicas up to deleting them outright.

Feature provided via plugins are, for example, an ML powered search as well as anomaly detection and most importantly support for request tracing.

3.2.3 Data Prepper

Data Prepper is a server-side data collector capable of filtering, enriching, transforming, normalizing, and aggregating data for downstream analytics and visualization(contributors, 2024b). It is commonly used for trace and log analytics and operates using pipelines. Each Data Prepper instance can run one or more pipelines simultaneously.

As detailed in contributors (2024b), a pipeline consists out of two required and two optional components, as well as an optional conditional routing, all chained as displayed in 3.2. The first component is the **Source**, which is the input of a pipeline with each pipeline having exactly one. There are multiple different kind of sources supported, like a more generic one consuming HTTP(S) requests, as well as sources for Kafka event queues, AWS S3 or OpenTelemetry (OTEL). Each of these source types has its own configuration as required by the underlying application providing the input event.
The second component is an optional one, called **Buffer**. It buffers the event received by the source until it is piped into the sink. There are two supported types of buffers, in-memory or disk based. If no buffer is specified, Data Prepper uses the default in-memory `bounded_blocking` buffer.
The next component is the optional **Processor** responsible for filtering, transforming and enriching the event. In case no processor is defined, the event will simply be passed to the sink unmodified. However it is also possible to define more than one processor, in which case the processors are executed in the order they are defined in the pipeline configuration, allowing for advanced processor chaining. Common processors are, for example, `parse_json` for parsing JSON inside the event, `rename_keys` for renaming keys inside the event, `add_entries` for enriching the data or `drop_events` which allows dropping events if a certain expression of the built-in expression syntax evaluates to `true`.
The last component is the **Sink** which defines the output where Data Prepper should send the event that ran through this pipeline. Similar to the source there are different kind of sinks, for example to write to OpenSearch, AWS S3 or even another Data Prepper pipeline. A pipeline can have more than one sink to send the event to. This is where the conditional routing comes into play. It allows selecting one or more sinks based on the current event content using the expression syntax. It also allows specifying a default route to which the event will be sent in case no route could be determined.

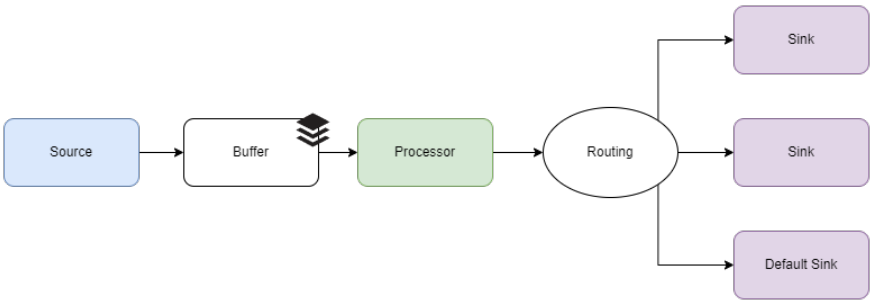

Figure 3.2: Data Prepper pipeline definition

3.2.4 Fluent Bit

Fluent Bit[7] is an open-source Telemetry Agent for logs, metrics and traces made with a focus on lightweight and fast performance. It is capable of collecting data from all major operating systems like Windows, Linux and macOS, as well as complex cloud infrastructure like Kubernetes. Because of this it is supported by a wide variety of telemetry ecosystems such as, for example, Prometheus[8]. Like 3.2.3, Fluent Bit also works using pipelines which are composed of the following components, allowing to parse and filter incoming events (In the concept of Fluent Bit, filtering means alter, enrich or dropping events). A typical Fluent Bit pipeline is structured as described in the following paragraph as well as F. B. Authors (2024a) and visualized in 3.3.

The first component is the **Input** component which is responsible for gathering the data from the corresponding source. Unlike the Data Prepper, a Fluent Bit pipeline can have multiple inputs. Examples of available inputs are `tail` which is used to tail a (log) file, `systemd` which collects data from Linux systemd, `kubernetes_events` collecting Kubernetes (K8s) events or `kafka` listening to a Kafka event queue.
The second component is the **Parser**, which is optional. This component is used to convert the incoming events from unstructured to structured data. Available parsers are, for example, `json` which parses the incoming JSON and `regex` which allows the parsing of the incoming data using a regular expression (regex).
After this comes the most important component, the **Filter**. It is responsible for altering the event data before sending them to the output. Similar to the Data Prepper Processor, it is possible to specify multiple filters which are then executed in the order of definition inside the pipeline configuration. Possible filters are, for example, `kubernetes` which enriches the event with K8s metadata from the configured cluster, `modify` which allows the modification of certain properties of an event and `nest` allowing flattening and nesting of properties. Another note-

[7]https://fluentbit.io/
[8]https://prometheus.io/

worthy filter is the `lua` filter which allows the developer to specify a custom Lua script that is used to alter the event.
The next component is the **Buffer** component which offers in-memory or disk based buffering like the Data Prepper Buffer. It has to be noted that in case of Fluent Bit, the events are not stored in plain text during the buffering, but rather Fluent Bit's internal binary representation (F. B. Authors, 2024b).
Last comes the **Output** describing the destination where Fluent Bit is supposed to send the final event. Similar to Data Prepper, Fluent Bit also offers to describe multiple outputs, as well as a routing. The difference is that the Fluent Bit routing is a lot simpler, operating on tags which are assigned by the Input component and a corresponding match on the output. These matches only supports regex and wildcards for the tags. Fluent Bit offers a vast number of possible outputs, amongst them are `cloudwatch_logs` (AWS CloudWatch), `s3` (AWS S3), `es` (ElasticSearch), `loki`, `opensearch` as well as more generic ones like `http` and `kafka`.

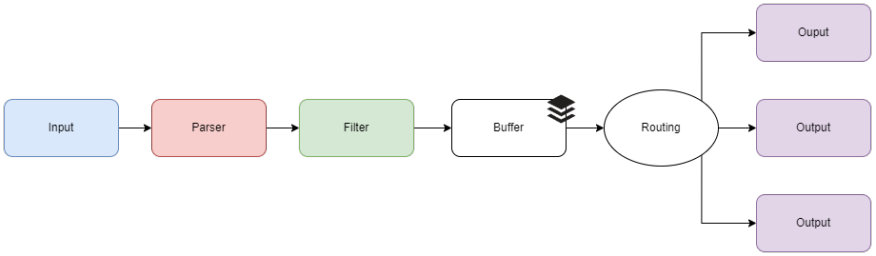

Figure 3.3: Fluent Bit pipeline definition

3.3 Request tracing

There are various different kinds of request tracing, also called observability, solutions. They could be split into two broad categories: **Application based** and **Network based**. The first category uses a language specific SDK to create code that generates traces. This is often paired with offering implementations for popular packages and using code injection to enable them. Solutions of this category sometimes require code changes or at least some kind of instrumentation code that enables the offered implementations. However because of this they have the advantage of being able to trace application internal requests. A solution belonging to this category is, for example, OTEL.
The other category does not modify an application's code and is often installed by a system administrator rather than a developer. Solutions of this category can only trace external requests by injecting a `requestId` into the requests and listening to the traffic entering and leaving all applications. This has the advantage that they are independent of the applications and their programming languages, but can not trace what happens inside these applications as well as can only trace

the type of request transportation they were designed to handle (for example HTTP). An example for this kind of request tracing would be the HTTP request tracing offered by **Istio** (T. I. Authors, 2024).

3.3.1 Solutions

As mentioned there are a lot of providers offering different kinds of request tracing solutions, however in the scope of this thesis the focus will be on the application based ones.

One of the most well known request tracing solutions is **OTEL**[9], which was briefly mentioned above. It is an open-source observability framework and toolkit designed to create and manage telemetry data such as traces, metrics, and logs (O. Authors, 2024d). This means that it focuses on trace generation using its SDKs and does not offer any storage or visualization options. However it can be used with a wide variety of observability backends due to its widely supported open-source OpenTelemetry Protocol (OTLP). Additionally the OTEL SDKs provide implementations for most of the popular packages of each language that handle, for example, HTTP(S) requests. It sometimes even provides implementations for popular frameworks like NestJS for its Javascript SDK. The OTEL SDKs are even capable of collecting environment specific metadata and adding them to their traces, for example enriching them with Docker container metadata in case the application is dockerized.

Another solution is **Zipkin**[10], licensed under the open-source Apache-2.0 license. Unlike OTEL, Zipkin provides not only a trace generation SDK, but also an observability backend and UI (Z. Authors, 2024a). Like the first solution, Zipkins protocol is supported by a wide variety of other observability backends meaning that using its own is not required. That is not the only similarity, Zipkin also offers SDKs for a number of programming languages with support for various transport modes like HTTP and Kafka, as well as many frameworks. However unlike OTEL, it currently does not support NestJS, only cujoJS, express and restify(Z. Authors, 2024b).

The third solution is the Apache-2.0 open-source **Jaeger**[11], with its documentation distributed under CC-BY-4.0. As detailed in T. J. Authors (2024), Jaeger is a distributed tracing platform offering an observability backend and UI. Its SDKs however are currently being deprecated in favour of using the OTEL provided ones. Before that it also had SDKs for the most popular programming languages, but that is now outdated due to the deprecation.

[9] https://opentelemetry.io/
[10] https://zipkin.io/
[11] https://www.jaegertracing.io/

Lastly, for completion's sake, there is also **Grafana Tempo**[12], which uses the same AGPLv3 license as the other Grafana products mentioned in this thesis. However this solution does not provide any trace generator but rather functions purely as a tracing backend. This means that it requires one of the above-mentioned solutions to provide the traces (Labs, 2024c) as well as a UI. In addition to that it is also recommended to use an external object storage like AWS S3, although it does support local storage (Labs, 2024a). However it has excellent integration with the other Grafana products.

For this thesis OTEL was selected due to its focus on trace generation and its well documented integration into the OpenSearch log-management system. This enables fulfilling the requirement of only one UI for both log-management and request tracing. Additionally it offers SDKs for most popular programming languages and was chosen over Zipkin due to it supporting a larger number of Javascript packages including NestJS. Aside from that OTEL has become the go-to solution to the point that it is supported by most request tracing systems and Jaeger even having deprecated its own SDKs in favour of OTEL.

3.3.2 OpenTelemetry

OpenTelemetry consists out of several major components as described in O. Authors (2024b), with the **Instrumentation** and the **Collector** being the components that have to be installed.
The instrumentation is used to instrument the code such that it generates traces. In case of OTEL there are a few different approaches available. The first one is to instrument your application manually using the SDK. This is also called *code-based instrumentation* (O. Authors, 2024c). In this case the developers directly write code that creates and sends traces, offering a lot of control over the trace generation.
The second approach is the so called *zero-code instrumentation* (also called *auto-instrumentation* and *automatic instrumentation*). Using this approach an official instrumentation script is injected using language specific code injection, which instruments the application using the provided instrumentations of supported libraries. In case of NodeJS the `require` flag is used, which preloads the instrumentation script before the application and thus modifies the libraries with their instrumented implementations (N. Contributers, 2024a). However when using this approach only the edges of the application are instrumented, due to the automatic instrumentation not being able to know where to generate traces inside the application specific code. Nevertheless this makes the second approach perfect for starting with instrumentation or in case it is not possible, or wanted, to modify the application with explicit trace generation. Additionally it is also possible to write a custom instrumentation script and use this for the auto-instrumentation,

[12] https://grafana.com/oss/tempo/

allowing some customisation. However a lot can also be configured using environment variables, though that depends on the auto-instrumentation library of the corresponding language, with for example NodeJS only supporting trace but not log and metrics configuration.
In both of these approaches it is possible configure the provided library instrumentations, as well as various other things, like the *resource detectors*. These are used to detect resources from the environment the application is running in. This includes environment variables, Docker metadata but also special resources in case the application is running in a cloud like AWS. Aside from that it is also possible to chose between using HTTP(S) or gRPC as the underlying transport protocol for OTLP(O. Authors, 2024c).

The other component, the collector as described in O. Authors (2024a), is used to collect the traces of the application and send them to the configured observability backend. Strictly speaking it is not necessary to use a collector, due to the instrumentations being able to send the data directly to most types of backends. However, as mentioned in O. Authors (2024a), it is recommended to use a collector in order to offload the data as quickly as possible from the application. The collector will then take care of handling retries, encryption and other such tasks.
If a collector is used, it can be deployed in two different ways. The first option is to deploy it as an **agent**, meaning that the collector will run on the same host as an application, or another collector instance, which transmits the traces to the collector. This pattern is very easy to configure and assigns a collector to each trace generating application. However that also makes it very hard to scale, should the host have too many trace generating applications that overwhelm the collector.
The other pattern is to deploy the collector as a **gateway**. In this case all applications would send their telemetry data to a load balancer which balances the load upon a set of collectors responsible for sending the data to the backend. This option has the advantages of separating the collector from the applications host and having a centralized location where all collectors are running (rather than one collector per application host). However this separation does introduce latency and causes a higher resource cost.
The structure of the collector configuration is similar to the previously shown Fluent Bit or Data Prepper configurations. It too uses a pipeline consisting out of three parts, albeit the parts are a bit simpler. The first part is the **Receiver** which configures where the telemetry data is coming from. Typically this is the OTLP protocol using HTTP and/or gRPC. However there are also other receivers like `jaeger`, `kafka` and `zipkin`.
The next part is the **Processor** which can transform the telemetry data. This part is optional, although, as described in O. Contributers (2024b), some are recommended, like the `batch` processor for batching.

The third part is the **Exporter** which configures the endpoint to where the collector should send the telemetry data. Supported are, for example, `file` for writing to a file, `otlp/jaeger` for sending to a Jaeger backend, `zipkin` for Zipkin backend or `otlp` for sending to a specified endpoint via OTLP.
Lastly it is also possible to chain pipelines via **Connectors** which are specified as the exporter of one pipeline and the receiver of another pipeline. These connectors can be used to summarize, replicate or route telemetry data.

4 Design

This chapter details the design of the structured logging implementation, as well as the deployment of the OpenSearch log-management system and OpenTelemetry request tracing. As mentioned previously, both OpenSearch and OpenTelemetry are to be deployed to a Kubernetes cluster using Helm.

4.1 Structured logging

The structured logging is implemented in a separate library inside the existing JValue Hub monorepo. This library contains a single function to create a new logger with a basic configuration. Using the configuration, the logger's log level can be configured with *info*, *warn*, *error* and *debug* as the possible log levels. Additionally it is also possible to en- or disable structured logging with the configuration. If the option is enabled, the logger will write all logs as JSONs instead of raw strings. The other possible configuration options most logger libraries offer, like configuring the output type and location of the logs as well as their properties and its format, are hardcoded. In this case the logger writes all logs to the console meaning stdout and stderr depending on the log level. The library also contains a log formatter that includes a timestamp, log level, log context, as well as the message and, in case of errors, the stack. However due to JValue using an Nx monorepo, the application version could not be used. This is because Nx can not support different versions for each project in the monorepo, only one global version, as discussed in N. Contributers (2024b). Because of that, and the commitment to always deploy the different microservices with an explicit version, it was determined that the Docker image version would suffice as an alternative. The Docker image version is injected by the Fluent Bit `kubernetes` filter.
The logger of this library is then integrated into the JValue NestJS applications using either the NestJS `logger` configuration property of `NestFactory.create` or the `app.useLogger` function.

4.2 Log-management

The basic architecture of the log management system is displayed in 4.1. In the following additional design choices for each component of the log-management system are described.

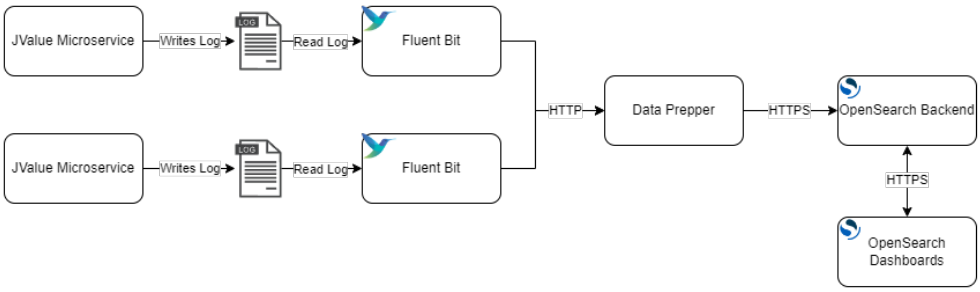

Figure 4.1: Log-management basic architecture

4.2.1 OpenSearch

The OpenSearch cluster is deployed in its single-node configuration. This was chosen due to the JValue project being fairly small with resources being a larger issue at the time than the potential performance issues caused by having only a single node.
The cluster is preconfigured with a custom admin and a data-prepper user, with the latter being used by the Data Prepper for log transmission. Additionally the cluster is configured to use SSL for communicating with both the Data Prepper and its UI. Lastly the UI does not have any Ingress[1] configured and thus can only be accessed using port-forwarding. This was chosen in order to prevent leaking any log data due to the data now only being reachable from the university network.
Concerning the index architecture it was decided to split the logs into four indices. The first one is the *jvalue-application* index. This index is the main index that contains the logs of all the JValue microservices.
The next index is the *postgres-operator* which is the destination for all logs created by the different postgres databases running on the K8s cluster that were created by the Postgres Operator[2]. This index is for easily debugging the databases and is separated from the first index due to the databases being viewed as a service that is used by the JValue application rather than an application of it.
The third index is the *longhorn* index, storing the logs of all the different Longhorn[3] services running in the cluster. This index was created due to Longhorn

[1]https://kubernetes.io/docs/concepts/services-networking/ingress/
[2]https://github.com/zalando/postgres-operator
[3]https://longhorn.io/

being a large source of potential issues and, similar to the databases, being viewed as an external infrastructure related service.
The last index is the default index called *kubernetes_ cluster*. This index contains all the collected logs that do not belong into any of the previously mentioned indices, including, for example, Ingress or Kubernetes system logs.

4.2.2 Data Prepper

The Data Prepper is deployed with only a single instance, receiving the data from all Fluent Bit instances. This was chosen to simplify the deployment. Additionally in case a single instance does no longer suffice in the future, it is possible to simply deploy more instances with a LoadBalancer configured between them and Fluent Bit.
The pipeline configuration of the Data Prepper consists out of five pipelines. The first pipeline is the *entry-pipeline* which receives all the logs via HTTP and then uses routing to forward each log to one of the other four pipelines, depending on where the log originated from as described in 4.2. The main pipeline, for parsing the logs of the different JValue applications, receives the data from the entry pipeline and then processes them using the `parse_json` processor. This processor is used due to the structured logging that the microservices use, as described in 4.1. After the processing is done, the pipeline sends the data to the corresponding OpenSearch index (4.3).

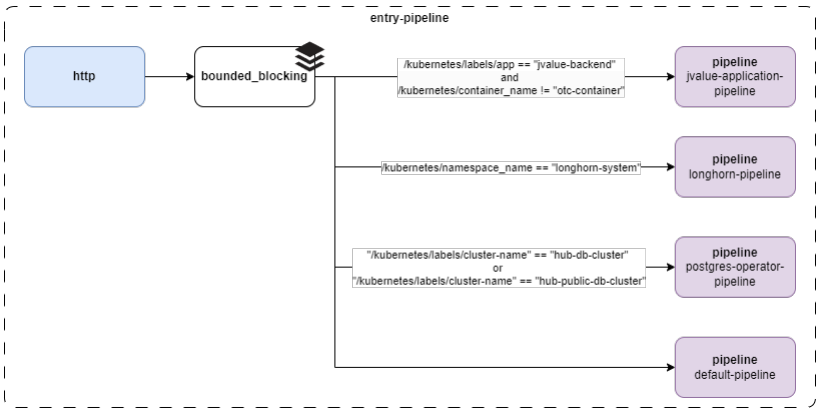

Figure 4.2: Data Prepper log entry pipeline

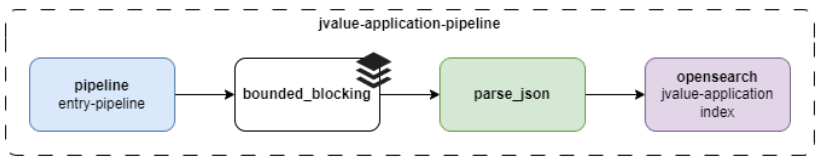

Figure 4.3: Data Prepper JValue log pipeline

The other pipelines also simply parse the logs using a corresponding processor and send their data to the corresponding OpenSearch index. Diagrams for them can be found in the appendix A.

4.2.3 Fluent Bit

Fluent Bit collects the logs with a pipeline consisting out of two inputs (4.4). The first input is the *tail* input that tails a log file, with the second collecting its data from Linux *systemd*. The reason for the systemd input is that the containers are running inside a Linux environment and thus certain system logs are written to systemd. Afterwards the collected logs are enriched with Kubernetes metadata using the correspondingly named filter. At the end the logs are sent to the Data Prepper using the `http` output. Fluent Bit is not configured to encrypt its data to the Data Prepper in order to quickly offload its logs. The encryption will happen between Data Prepper and OpenSearch.

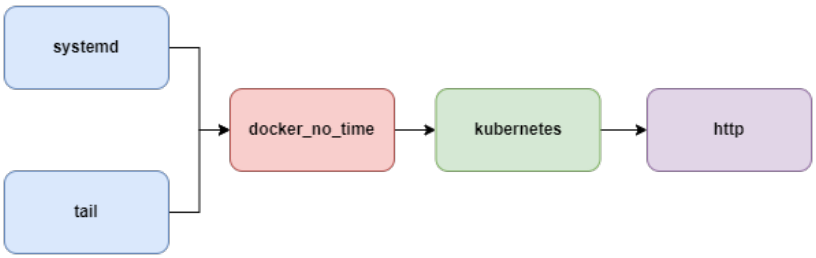

Figure 4.4: Fluent Bit log pipeline

4.3 Request tracing

For the request tracing the basic architecture as described in 4.5 was chosen. The following subsections describe in more detail the design decisions for each component.

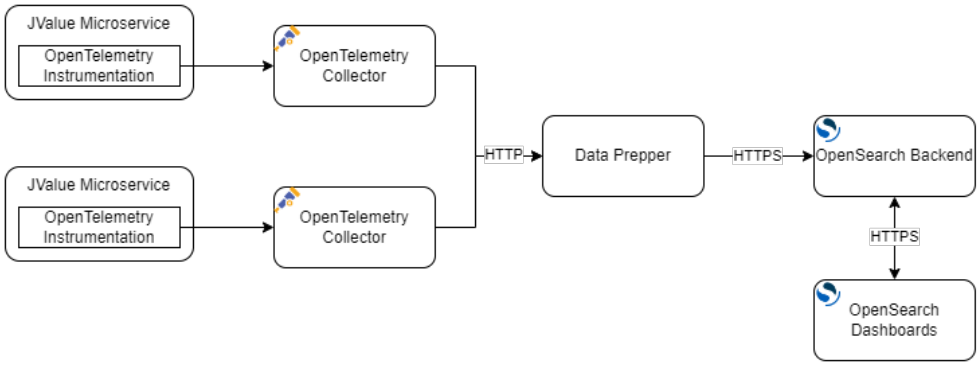

Figure 4.5: Request tracing basic architecture

4.3.1 OpenTelemetry

The OTEL collector is deployed using the agent pattern, which was chosen due the four JValue applications being the only ones where tracing is wanted and this approach allows the fast offloading of traces.

For the instrumentation itself, the *zero-code instrumentation* approach was chosen, mainly due to the trace generation of the auto-instrumentation library sufficing for the scope of this thesis.

4.3.2 Data Prepper

In the scope of request tracing the Data Prepper configuration is extended by three additional pipelines. The first pipeline is the *otel-trace-pipeline* which contains the `otel_trace_source` listening to OTEL traces (4.6). This pipeline then sends the data to the other two pipelines.
One of these pipelines is the *raw-pipeline* which uses the `otel_trace_raw` and `otel_trace_group` processors to parse the traces and enrich them with additional data from the metadata already stored in OpenSearch(4.7). After the enrichment the pipeline sends the traces to the OpenSearch index for raw trace data that is created by the observability plugin.
The other pipeline is the *service-map-pipeline* which is responsible for extracting the necessary data from the traces to create OpenSearch's service map visualization (4.8). This is done using the `service_map_stateful` processor and transmitting the data to the corresponding index created by the observability plugin.

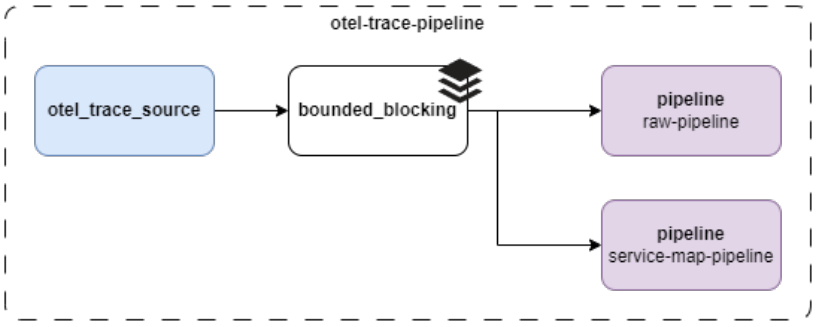

Figure 4.6: Data Prepper OTEL entry pipeline

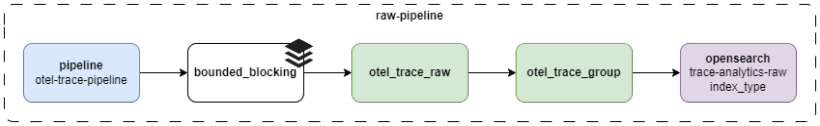

Figure 4.7: Data Prepper OTEL raw pipeline

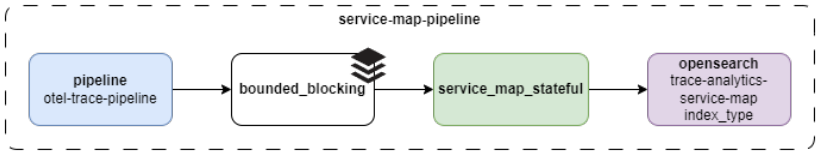

Figure 4.8: Data Prepper OTEL service pipeline

5 Implementation

This chapter details the implementation and deployment of the log-management and request tracing systems. It has to be noted that during the implementation some issues were encountered that forced some changes to the architecture described in the previous chapter.

5.1 Structured logging

The structured logging library was implemented using the **Winston**[1] library. This library was chosen due to their large feature set and support of Javascript and Typescript in general but also NestJS specifically.

The library was created in the *libs* subdirectory and called *nestjs-shared* due to the intention of later including other functionality that all NestJS applications require that are not JValue domain specific.
The exported main function has an optional `LoggerOptions` input that defines whether debug logging and/or structured logging as JSON should be enabled. The default value is `false` for both due to structured logging mainly being used when running in K8s. The function then returns a Winston `Logger` instance configured with a log level based on the debug flag and a format constructed using the provided `format.timestamp()`, `format.errors()` and either `format.json()` or `format.colorize()` in combination with a custom formatter, depending on whether structured logging is enabled or not. The custom formatter has the following simple structure.

```
1 let formattedMessage = `${timestamp} [${context}] ${level}: ${
    message}`;
2 if (stack !== undefined) {
3   formattedMessage += ` - ${stack.toString()}`;
4 }
```

[1] https://www.npmjs.com/package/winston

The main function is then used during the NestJS initialization of each JValue application. The created Winston logger is then wrapped using the *Winston-Module*[2] and passed to the `logger` configuration property of `NestFactory.create`. The configuration of whether structured logging should be enabled is done via an environment variable called `LOG_AS_JSON`.

5.2 Log-management

For the log-management, *OpenSearch Kubernetes Operator*[3] is used to deploy OpenSearch. However this operator does not support deployment of a single-node cluster and thus the standard multi-node configuration is used. The following diagram shows the basic Kubernetes pod architecture of the log management system.

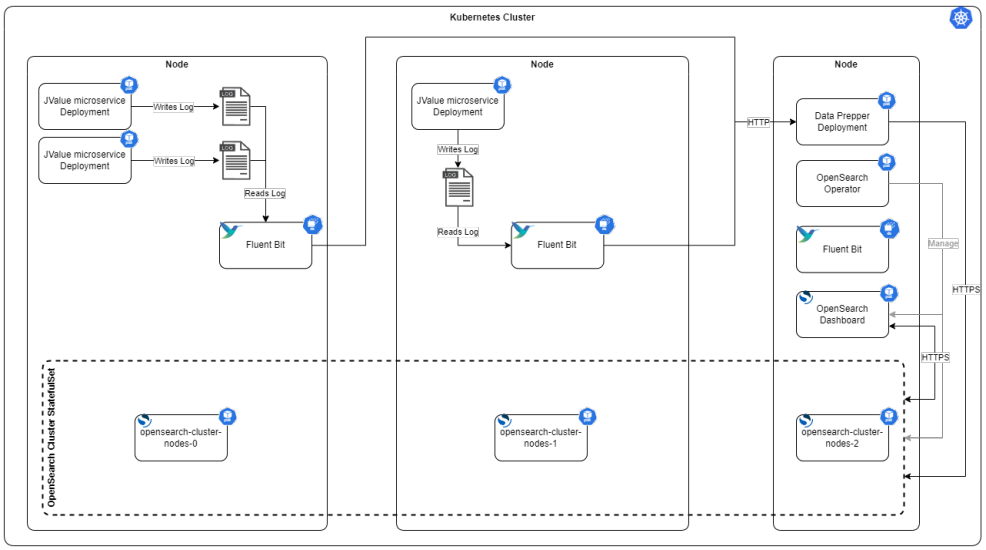

Figure 5.1: Log-management K8s pod architecture

5.2.1 OpenSearch

As mentioned above, the OpenSearch cluster is installed using the OpenSearch Kubernetes Operator. The operator itself is installed via its Helm Chart[4] without any changes to the default configuration.
The OpenSearch cluster is then installed using the *OpenSearchCluster* Custom Resource Definition (CRD) of the operator. As mentioned this CRD does not

[2]https://www.npmjs.com/package/nest-winston
[3]https://github.com/opensearch-project/opensearch-k8s-operator
[4]https://github.com/opensearch-project/opensearch-k8s-operator/tree/main/charts/opensearch-operator

support the single-node deployment and requires at least three `cluster_manager` nodes to run. All of these OpenSearch nodes have both the `cluster_manager` as well as the `data` role. Additionally they are configured to use the already present Longhorn StorageClass using a configurable disk size. Lastly the operator installs all available plugins, for both the cluster and the Dashboards, by default. This includes the *opensearch-observability* and *dashboards-observability* which are required for request tracing.
For the creation of the two initial users a few concessions had to be made. The operator offers two different approaches on how to create users. The first one is to define them inside the `securityConfigSecret` where it is also possible to define roles, tenants and basic configuration. While this allows for some more flexibility, it does however require the security config to contain the password hash of each user.
The other approach is to use the *OpensearchUser* CRD. In this approach the password is retrieved from the specified K8s secret. Unfortunately these approaches cannot be used at the same time and creating a custom admin user is only supported by using the security config, which is also why this approach was chosen. The issue with the password hashes was solved by adding a *values.secret.yaml* Helm values file. This values file contains the hashes and is specified inside the *.gitignore* to prevent accidental publishing. Another, albeit smaller issue, with this approach is that the security config requires a certain minimal configuration for each of the configurations during the initial deployment. At the same time some of these configurations, like tenants, can be edited via the UI and would be override by the minimal config in case the chart gets updated. Because of this a separate step is necessary during deployment that redeploys the chart without the minimal config to allow the editing via the UI. For both the admin and data prepper user there is also a secret containing the username and password. These secrets are used by the OpenSearch Dashboards and the Data Prepper respectively to access the cluster. Aside from specifying this secret, the only other configuration made to the UI is enabling TLS with auto generated certificates.

Other CRDs provided by the operator are *OpensearchIndexTemplate* and *OpensearchComponentTemplate*. With these it is possible to create Index- and Component-Templates as they are described in 3.2.2. However shortly after starting the implementation of these templates using the CRDs, they were no longer available. The reason for that was that the company originally developing the operator, Opster, was bought by Elastic in late November 2023(Elastic, 2024). The operator was forked into a repository under the opensearch-project group, however in the process a few things broke, amongst them these two CRDs.
Due to this the index and component templates were refactored into JSON files that are manually deployed after the cluster creation using a custom bash script. This bash script collects all the index and component templates from their re-

spective directories and deploys them by using *curl*[5] to execute a REST API call to the clusters index template API. This request uses the JSON file content as the request body and the name of the file as the name of the template. This results into the following command for a file called *example-index-template.json*: `curl -s -k -u "$user"-H "Content-Type: application/json"-X PUT https://localhost:$port/_index_template/example-index-template -d "$template_json "`. The `$user` and `$port` are required inputs of the script and `$template_json` is the content of the JSON file. The inputs are passed to the script using the `-u` and `-p` flags resulting into the following command for executing the script `opensearch-templates.sh -p <LOCAL_PORT> -u "<OPENSEARCH_ADMIN_USER>:<OPENSEARCH_ADMIN_PASSWORD >"`. The reason for using `localhost` as the endpoint of the cluster is that the cluster can only be reached using port-forwarding. This means that a port-forwarding has to be started prior to executing this script, although it does not matter which OpenSearch cluster node is being used. The complete code of the bash script can be found in B with the directory structure being detailed in C.
Currently there are four index templates, one for each index mentioned in the previous chapter, and four component templates. The first component template contains a default shard and replica count configuration and the second one common properties expected by all logs, namely `logLevel` and `timestamp`. The other two templates are for Fluent Bit properties with the first one containing common properties of Fluent Bit, like `date`, `time`, `log` and `stream`. The second one describes some of the properties the `kubernetes` filter injects, for example `container_name`, `container_image`, `namespace_name`, `pod_name` and `labels`.
The index template for the *jvalue-application* index uses all of these four component templates and adds the `message`, `context`, `traceId`, `spanId` as well as `traceFlags` properties.
The second index template, for the *longhorn* index, also uses the four component templates and specifies `message`, `backup`, `controller`, `error` and `node` as properties.
The template for the *postgres-operator* index only uses all of the component templates, without any additional properties being specified.
The last index template is for the default index and because of that it does not use the component template specifying common properties due to the unknown structure and content of the logs in this index.
All of the above index templates also specify a corresponding index pattern which defines to which index this template is to be applied to. Additionally they also contain an alias for their index, which is a virtual index than can point to another index. This is useful in case the logs are spread over multiple indices, for example one index for each month. Using an alias allows querying the alias instead of the index and thus only requiring to move the alias to the newest month instead of adjusting the query.

[5] https://curl.se/

5.2.2 Data Prepper

The Data Prepper is deployed using a custom Helm chart that uses the *opensearchproject/data-prepper:latest* Docker image. Originally it was planned to add support for deploying the Data Prepper to the OpenSearch Operator, however due to the aforementioned repository migration it is unknown where that is still planned or when it is implemented. In this Helm chart the image is deployed as a *Deployment* with *Services* for the pipeline inputs. The pipeline configuration is done via a *ConfigMap* that contains the pipeline definitions. This ConfigMap, along with the certificate for connecting to OpenSearch, are mounted via volume mounts.
The log pipelines are implemented as described in the previous chapter. However two adjustments have been made, one to the default pipeline and one to the jvalue-application pipeline. For the default pipeline the `drop_events` processor was added which simply drops all log events of this pipeline. This was added to conserve storage space due to the limited amount of disk space available to the Kubernetes cluster.

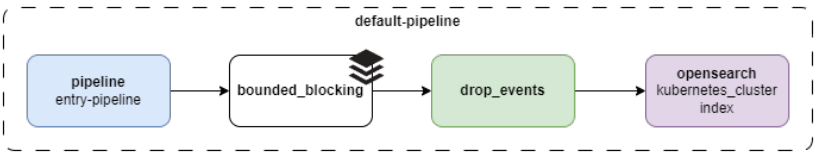

Figure 5.2: Data Prepper default log pipeline (adjusted)

In case of the jvalue-application pipeline a second processor of type `rename_keys` was added. This processor renames the properties `level`, `trace_id`, `span_id` and `trace_flags` to `logLevel`, `traceId`, `spanId` and `traceFlags`. The rename of `level` is purely for consistency as all the other services call the property `logLevel`. The other renames are necessary due to the OTEL instrumentation of Winston, which injects the trace properties into the logs, using snake case, while OpenSearch expects them in camel case.

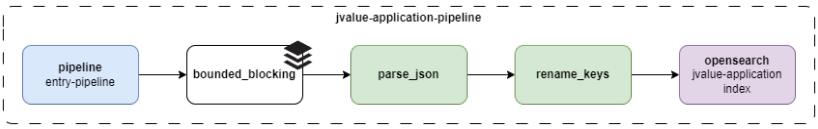

Figure 5.3: Data Prepper JValue log pipeline (adjusted)

Lastly for all of the four pipelines sending data to OpenSearch, the index used is suffixed with the current year and month. Because the `opensearch` sink automatically creates indices if they do not yet exist, the logs for each month are separated from each other. This was chosen to simplify the deletion of older logs

and to prevent indices from growing too large. The time period of one month was decided due to the lack of time to experiment how much data is being generated in what time frame and is thus only an initial suggestion.

5.2.3 Fluent Bit

For the installation of the Fluent Bit collector the official Helm chart[6] is used. This chart already has the two inputs of the previous chapter preconfigured, as well as the Kubernetes filter. However this filter was overwritten in order to disable including annotations. The reason for that is an issue encountered for the labels that also applies to the annotations. This issue had to be fixed manually and fixing it for annotations was not deemed worth the effort. The issue is that OpenSearch interprets a dot inside a property name as a nested object, meaning, for example, the label `app.kubernetes.io` is interpreted as the object `app` containing the object `kubernetes` with the property `io`. That in itself is not an issue, however the JValue deployments also have the label `app`. Because of this OpenSearch cannot determine the type of `app` due to finding both an object and a string with this name.
This resulted into the fix shown in 5.4. Summarized the fix uses five additional filters which first un-nest the labels object, then replaces the dot in some known and wanted labels with an underscore, deleting the rest, and lastly nests the renamed properties again. This un-nesting and nesting is necessary due to Fluent Bit being unable to directly rename nested keys. At the end the HTTP output is added which is configured to send the data to the Data Prepper Service for the logs mentioned in previous subsection.

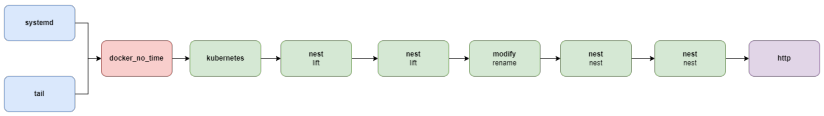

Figure 5.4: Fluent Bit log pipeline (adjusted)

5.3 Request tracing

There also were a few adjustments that had to be made to the request tracing part of the architecture. These were also caused due to the usage of an operator, namely the *OpenTelemetry Operator*[7], and are highlighted in the following subsections with the diagram displaying the adjusted architecture being located below.

[6]https://github.com/fluent/helm-charts/tree/main/charts/fluent-bit
[7]https://github.com/open-telemetry/opentelemetry-collector

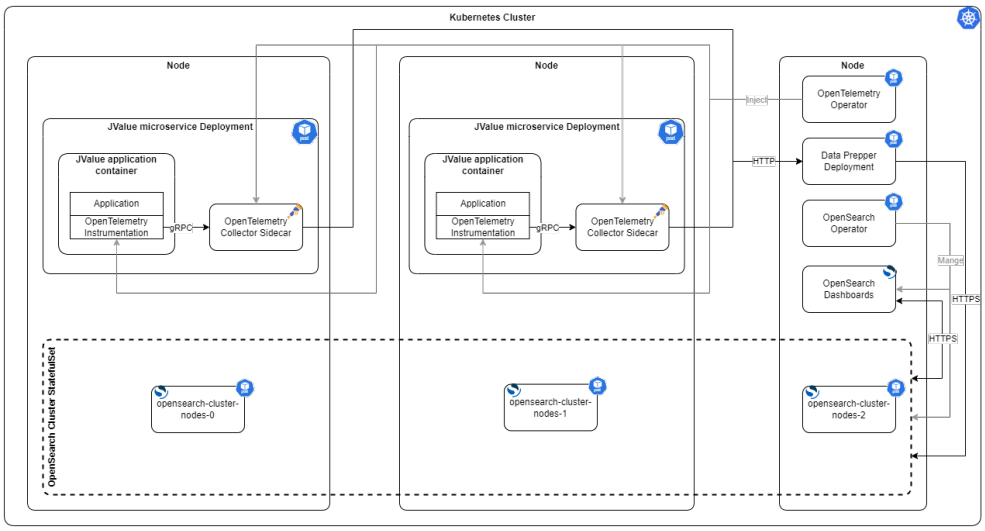

Figure 5.5: Request tracing K8s pod architecture (Operator revised)

5.3.1 OpenTelemetry

OpenTelemetry offers an operator for deploying the collector as well as the instrumentations. This operator also has a Helm chart[8] for its deployment.
The OTEL collector is deployed using the operators *OpenTelemetryCollector* CRD. It is configured to use the *Sidecar* deployment type to fit the design discussed prior. This approach also requires less configuration due to the collector being reachable using *localhost* which is the default endpoint of the instrumentations.

The operator also offers a CRD for instrumentations. Using this instrumentation the operator injects the pod with an init container that contains the instrumentation, as well as whatever else is needed to activate it. In the case of JValue, which uses the NodeJS instrumentations, this entails an init container that copies and mounts the instrumentation script to the application container. It also adds the `NODE_ENV` environment variable that activates the instrumentation as explained in 3.3.2.
The images for the init containers are provided by the operator, but can be overwritten with custom containers. This is also necessary for JValue, due to the official images having two issues. The first and minor issue is that, at the time of implementation, the official image does not allow disabling certain instrumentations and for the NodeJS instrumentation it is recommended to disable *@opentelemetry/instrumentation-fs* as discussed in O. Contributers (2024a). The second issue is that the official image uses the new `NodeSDK` to initialize the instru-

[8]https://github.com/open-telemetry/opentelemetry-helm-charts/tree/main/charts/opentelemetry-operator

mentation and there seems to be some issue with NestJS, because only a fraction of the expected traces are generated. Because of these issues a custom image has been created, and specified in the operators deployment configuration, that disables the mentioned instrumentation and uses the old `registerInstrumentations` implementation with an issue being created in the GitHub repository[9]. The image also allows en-/disabling of OTELs debug mode via an environment variable which was added while debugging the instrumentation issue. The code for this image is located inside the *otel-autoinstrumentation-node* project which in turn is located inside the monorepo's Kubernetes infrastructure directory, which also contains the Helm charts. However this project is a standalone project and not part of the JValue Nx monorepo and should be deprecated as soon as the mentioned issues of the official image are fixed.

Both the collector and instrumentation CRD are enabled by annotating the pod with the corresponding annotations. This prompts the operator to inject the instrumentation and sidecar. For example the following annotation injects the NodeJS instrumentation with the name `nodejs-instrumentation` that was created in the `monitoring` namespace: `instrumentation.opentelemetry.io/inject-nodejs: 'monitoring/nodejs-instrumentation'`. In comparison `sidecar.opentelemetry.io/inject: 'monitoring/opentelemetry-collector-sidecar-agent'` injects the sidecar named `opentelemetry-collector-sidecar-agent` from the `monitoring` namespace.

5.3.2 Data Prepper

For the request tracing, the Data Prepper deployment was extended with an additional *Service* for receiving the traces from the OTEL collectors, as well as the pipelines described in the previous chapter. This time no adjustments had to be made to the pipelines.

[9] https://github.com/open-telemetry/opentelemetry-operator/issues/2510

6 Evaluation

This chapter discusses the evaluation conducted and highlights the identified remaining issues.
The evaluation was conducted with three JValue developers using the Thinking-aloud method.

Thinking-aloud refers to a method that involves asking participants to solve a set of questions while saying out loud what they are thinking. The method is used both for psychological and educational research, as well as to build knowledge-based computer systems (M.W. van Someren, 1994). In computer science this method has been in use for usability testing for a long time (McDonald et al., 2020).

This evaluation method was chosen, due to various reasons. The main reason is that this method exposes the individual's thought process and the results should therefore relate mainly to the working memory (McDonald et al., 2020).
The second reason is that thinking-aloud evaluations are fairly fast and easy to setup and conduct. This is due to them only requiring a list of tasks and one examiner that logs the thoughts of the evaluation participants.

In the process of this evaluation each participant got five tasks in total to fulfill. The tasks were split into two before and three after tasks. It has to be noted that due to time constraints the before and after tasks had to be done in the same meeting. Additionally before and after had two tasks in common for better results comparison.

The evaluation was started with the two before tasks in which the participants were asked to fulfill them the way they used to. In our case that meant using the Lens application for all participants. During the after tasks, the participants used the deployed log-management and request tracing solutions.

The first task was to find a log entry explaining an error received by a user of the JValue application who tried to initiate a pipeline run (see *appendix*). The reason for this task was to test how fast the participants could find specific errors in the logs.

The second task was to query the logs of all *longhorn-manager* instances for any errors that have occurred the last 12 hours. This task was chosen to evaluate how easy it is to query logs of deployments that have multiple running instances. The third and fourth tasks were identical to the first two, just for the after evaluation. The last task was to use the request tracing solution to find a trace that represents the successful creation of a JValue pipeline run. This was chosen to demonstrate the request tracing.

6.1 Evaluation results

At the end the participants were asked to score from 1 (very negative) to 5 (very positive) how satisfied they were with the new log-management and request tracing solution. The results are visualized in the following graphic.

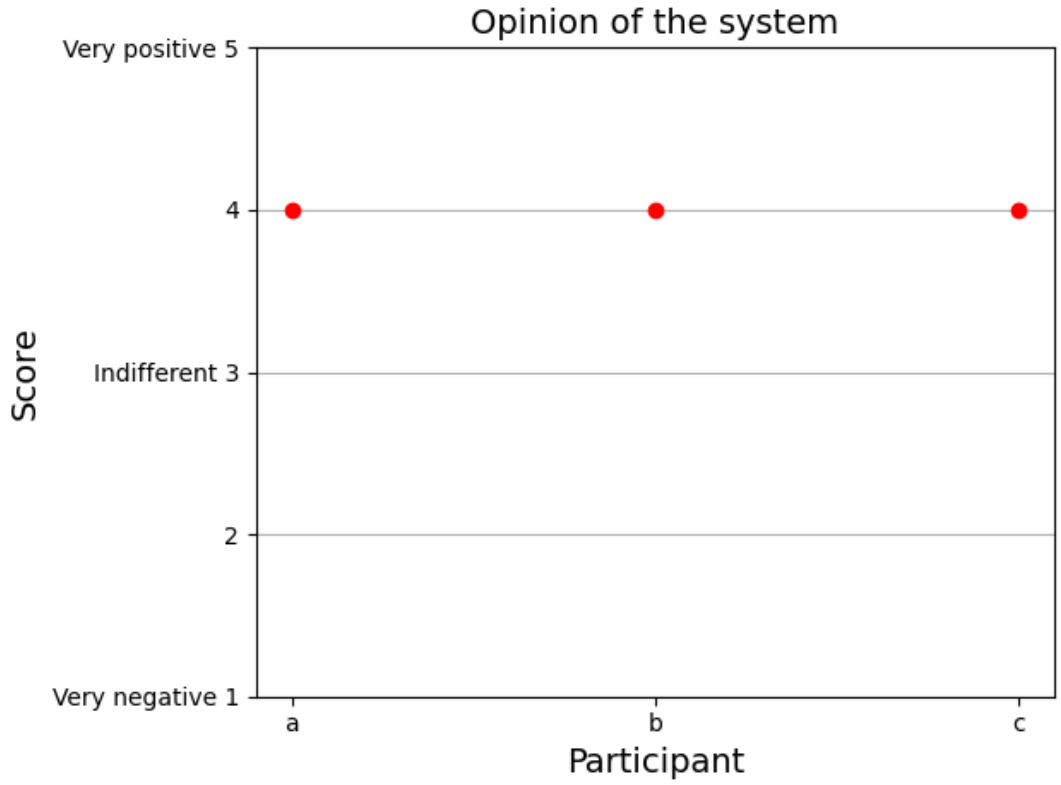

Figure 6.1: Evaluation score

Additionally some positive and negative feedback was given.
For the negatives the most prevalent one was that the UI was too overwhelming without any sort of prior introduction. The only other one, that most participants

wished for, was that a dashboard would have been already configured for a fast information overview.
Concerning the positives, one was that the UI allowed for advanced queries using keywords and time filters as well as the configuration of said dashboards. The most important thing was that the traces were unanimously mentioned as highly valuable for debugging.

6.2 Remaining issues

Next the remaining issues are highlighted, split into their corresponding subsection log-management and request tracing.

6.2.1 Log-management

The largest outstanding issues are caused by the aforementioned migration of the OpenSearch Operator from the Opster to the official OpenSearch-Project repository.
The first one is the, at the time of implementation, missing support for creating Index- and Component-Templates via CRDs, which required the deployment of those using a script. The second one is that also missing are CRDs for ISM policies. Because of that and time constraints, there is currently no automatic index deletion. This causes the indicies to gradually consume more and more storage until none is left. This is also the reason why, at the moment, the logs that would be written to the *kubernetes_ cluster* index are being dropped instead of stored.

Another issue concerns the SSL certificates. As of time of development, the operator did not offer any automatic certificate renewal for auto-generated SSL certificates. One possible solution for this would be to replace the auto-generated certificates with manual ones, however this was not done due to time constraints. On the subject of SSL, there is also an undiagnosed issue with the SSL connection from the UI to the backend after a few minutes of use, that causes all connections to fail until the port-forwarding has been restarted. This issue only arouse after the development was finished and could not be resolved thus far.

The last and least important issue is the Data Prepper throwing JSON parse errors, even though the data is sent to OpenSearch. From the research conducted this seems to be an underlying Data Prepper issue where recursive parsing errors are falsely logged.

6.2.2 Request tracing

Concerning the request tracing implementation there have been three issues identified.

The most important issue is that, as mentioned in 5.3.1, there is currently an unresolved issue with the official OTEL NodeJS auto-instrumentation image. The image does not correctly initialize and generates only a very small amount of (mostly) unimportant traces. The reason why most of these traces are not important is, that they were created by the `fs` instrumentation which, as previously mentioned, should be disabled. An issue has been created in the OpenTelemetry Operator GitHub repository.

The second issue is that the OTEL collector sidecar requires a manual pod restart in case the sidecars gets updated. The reason for that is that the operator currently does not remember where it injected which sidecar. This means that the only way to update a running sidecar is by restarting the pod and thus forcing the pod to get the new sidecar version injected by the operator.

The last issue is that the instrumentations vastly increase the application startup time. This is due to OTEL instrumentations patching the corresponding NodeJS packages and patching NestJS takes especially long. As a rough estimate, the `hub-backend` took up to two minutes longer during its start-up.

7 Conclusions

This thesis discussed and detailed the deployment of the OpenSearch log-management system to a Kubernetes cluster using Helm. In addition to that it also detailed the deployment of OpenTelemetry as a request tracing system to said Kubernetes cluster and its configuration for NestJS. It also detailed how said request tracing system can be integrated into OpenSearch to allow managing both logs and traces via a single UI. Lastly the thesis included the implementation of a logging library supporting structured logging for NestJS using Winston, although the application version had to be substituted with the docker image version as mentioned.
However the thesis also highlighted some outstanding issues and in the following a few suggestions for the future will be given.
The first improvement suggestion is to do some more experimentation concerning how long the logs should be stored and create an ISM policy for automatic log retention.
The second one, which is a result of the evaluation, is the introduction of a small "cookbook" to introduce the developers into the log-management UI which should prevent overwhelming them.
The last and most important improvement suggestion is to create a separate Kubernetes cluster just for running OpenSearch. This would reduce the resource consumption of the current cluster and, more importantly, allows the usage of a single OpenSearch cluster for all deployment environments of JValue. These could be separated using the currently untouched Tenants provided by OpenSearch. In fact using them it would even be possible to use this OpenSearch cluster for multiple projects of the department chair, although that would probably require somebody responsible for the cluster.

Appendices

A Design: Additional Data Prepper pipelines

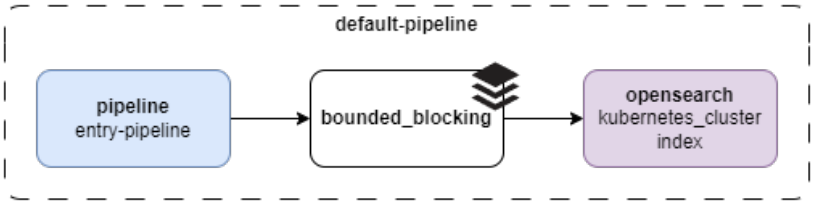

Figure 1: Data Prepper default log pipeline

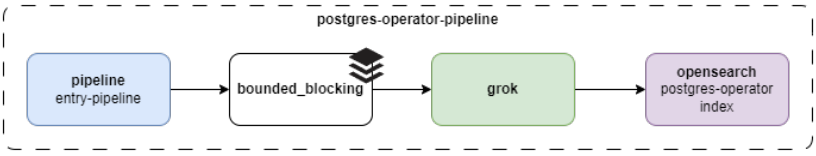

Figure 2: Data Prepper Postgres Operator log pipeline

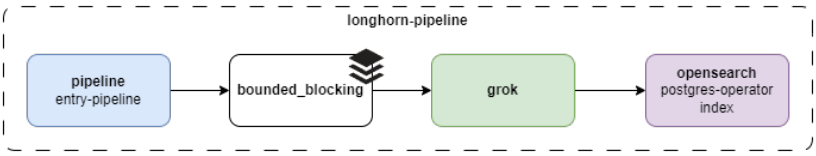

Figure 3: Data Prepper Longhorn log pipeline

B Implementation: Index Template deployment script

```
#! /bin/bash

while getopts p:u: flag
do
    case "${flag}" in
        p) port=${OPTARG};;
        u) user=${OPTARG};;
    esac
done

usage="[USAGE] opensearch-templates.sh -p [OPENSEARCH_API_PORT] -
    u [CURL_USER]"
```

```
12
13 script_dir=$( cd -- "$( dirname -- "${BASH_SOURCE[0]}" )" &> /dev
      /null && pwd )
14 component_templates_dir="$script_dir/opensearch-templates/
      component"
15 index_templates_dir="$script_dir/opensearch-templates/index"
16
17 opensearch_endpoint="https://localhost:$port"
18 opensearch_index_template_path="_index_template"
19 opensearch_component_template_path="_component_template"
20 curl_basic_flags=(-s -k -u "$user" -H "Content-Type: application/
      json")
21
22 if [ -z "$port" ] || [ -z "$user" ]
23 then
24       echo "$usage"
25       exit -1
26 fi
27
28 function create_index_template() {
29     template_name=$1
30     template_json=$2
31     echo "Creating component template $template_name"
32     curl=$(curl "${curl_basic_flags[@]}" -X PUT
          $opensearch_endpoint/$opensearch_index_template_path/
          $template_name -d "$template_json")
33     echo $curl
34 }
35 function create_component_template() {
36     template_name=$1
37     template_json=$2
38     echo "Creating index template $template_name"
39     curl=$(curl "${curl_basic_flags[@]}" -X PUT
          $opensearch_endpoint/$opensearch_component_template_path/
          $template_name -d "$template_json")
40     echo $curl
41 }
42
43 for json_file in $(find $component_templates_dir -type f -name "
      *.json") ;
44 do
45     template_name=$(basename $json_file .json)
46     template_json=$(cat $json_file | tr '[:space:]' ' ')
47     create_component_template "$template_name" "$template_json"
48 done;
49 for json_file in $(find $index_templates_dir -type f -name "*.
      json") ;
50 do
51     template_name=$(basename $json_file .json)
52     template_json=$(cat $json_file | tr '[:space:]' ' ')
53     create_index_template "$template_name" "$template_json"
```

```
done;
```

C Implementation: Index Template deployment directory structure

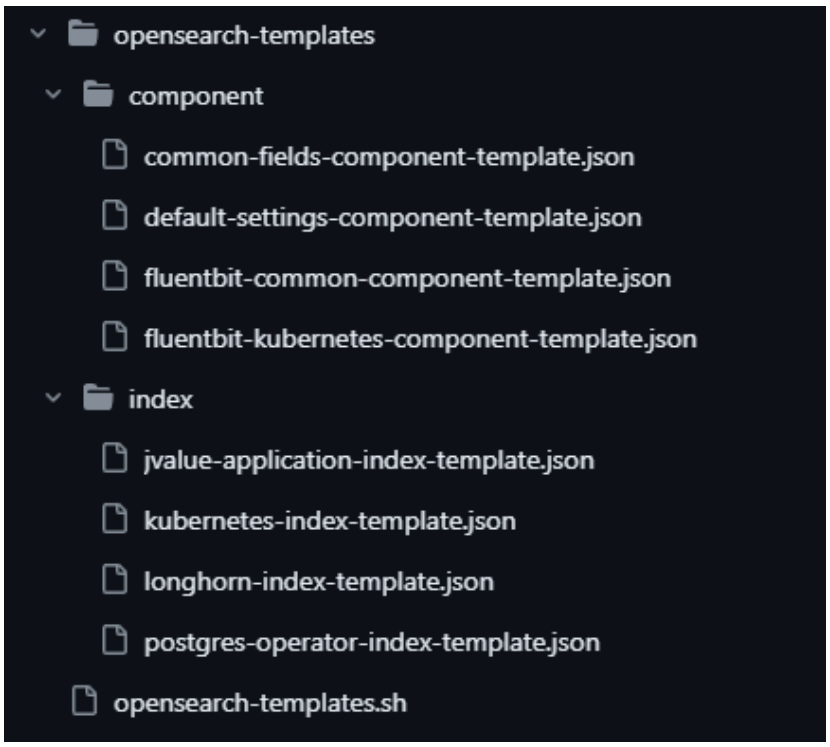

Figure 4: Index Template deployment directory structure

D Evaluation JValue failed request .har snippet

```
{
  "request": {
    "method": "POST",
    "url": "https://dev.jvalue.com/hub-backend/runs",
    "httpVersion": "http/2.0",
    "bodySize": 53,
    "postData": {
      "mimeType": "application/json",
      "text": "{\"instanceId\":\"c3b491e9-d030-434b-8668-7
        d3296dd8f83\"}"
    }
  },
  "response": {
```

```
    "status": 500,
    "statusText": "",
    "httpVersion": "http/2.0",
    "content": {
      "size": 52,
      "mimeType": "application/json",
      "text": "{\"statusCode\":500,\"message\":\"Internal server
         error\"}"
    }
  },
  "startedDateTime": "2024-01-28T17:36:06.480Z",
  "time": 955.9730000037234
}
```

References

Authors, F. B. (2024a, April). *Data pipeline | 2.1 | fluent bit: Official manual*. https://docs.fluentbit.io/manual/v/2.1/concepts/data-pipeline

Authors, F. B. (2024b, April). *Fluent bit v2.1 documentation | 2.1 | fluent bit: Official manual*. https://docs.fluentbit.io/manual/v/2.1/

Authors, O. (2024a, April). *Collector | opentelemetry*. https://opentelemetry.io/docs/collector/

Authors, O. (2024b, April). *Components | opentelemetry*. https://opentelemetry.io/docs/concepts/components/

Authors, O. (2024c, April). *Instrumentation | opentelemetry*. https://opentelemetry.io/docs/concepts/instrumentation/

Authors, O. (2024d, April). *What is opentelemetry? | opentelemetry*. https://opentelemetry.io/docs/what-is-opentelemetry/

Authors, T. I. (2024, April). *Istio / overview*. https://istio.io/latest/docs/tasks/observability/distributed-tracing/overview/

Authors, T. J. (2024, April). *Jaeger documentation*. https://www.jaegertracing.io/docs/1.56/

Authors, T. K. (2024, April). *Logging architexture | kubernetes*. https://kubernetes.io/docs/concepts/cluster-administration/logging/

Authors, Z. (2024a, April). *Architecture • openzipkin*. https://zipkin.io/pages/architecture.html

Authors, Z. (2024b, April). *Tracers and instrumentation • openzipkin*. https://zipkin.io/pages/tracers_instrumentation.html

Banon, S. (2021, February). *Introducing elastic license v2, simplified and more permissive; sspl remains an option*. https://www.elastic.co/blog/elastic-license-v2

B.V., E. (2024a, April). *Information out: Search and analyze | elasticsearch guide [8.13] | elastic*. https://www.elastic.co/guide/en/elasticsearch/reference/current/search-analyze.html

B.V., E. (2024b, April). *What is elasticsearch? | elasticsearch guide [8.13] | elastic*. https://www.elastic.co/guide/en/elasticsearch/reference/current/elasticsearch-intro.html

Contributers, N. (2024a, April). *Command-line api | node.js v22.1.0 documentation*. https://nodejs.org/api/cli.html#-r---require-module

Contributers, N. (2024b, April). *Using different versions of a lib or a package • issue #309 • nrwl/nx • github*. https://github.com/nrwl/nx/issues/309

Contributers, O. (2024a, April). *High startup memory when using @opentelemetry/instrumentation-fs • issue #1344 • open-telemetry/opentelemetry-js-contrib • github*. https://github.com/open-telemetry/opentelemetry-js-contrib/issues/1344

Contributers, O. (2024b, April). *Opentelemetry-collector/processor at main • open-telemetry/opentelemetry-collector • github*. https://github.com/open-telemetry/opentelemetry-collector/tree/main/processor%5C#recommended-processors

contributors, O. (2024a, April). *Creating a cluster - opensearch documentation*. https://opensearch.org/docs/2.13/tuning-your-cluster/

contributors, O. (2024b, April). *Data prepper - opensearch documentation*. https://opensearch.org/docs/latest/data-prepper/

contributors, O. (2024c, April). *Getting started - opensearch documentation*. https://opensearch.org/docs/2.13/about/

contributors, O. (2024d, April). *Intro into opensearch - opensearch documentation*. https://opensearch.org/docs/2.13/getting-started/intro/

Elastic. (2024, April). *Elastic - elastic completes acquisition of opster*. https://ir.elastic.co/news/news-details/2023/Elastic-Completes-Acquisition-of-Opster/default.aspx

Labs, G. (2024a, April). *Configure tempo | grafana tempo documentation*. https://grafana.com/docs/tempo/latest/configuration/

Labs, G. (2024b, April). *Grafana loki oss | log aggregation system*. https://grafana.com/oss/loki/

Labs, G. (2024c, April). *Instrument for distributed tracing | grafana tempo documentation*. https://grafana.com/docs/tempo/latest/getting-started/instrumentation/

McDonald, S., Cockton, G., & Irons, A. (2020). The impact of thinking-aloud on usability inspection. *Proc. ACM Hum.-Comput. Interact.*, *4*(EICS). https://doi.org/10.1145/3397876

M.W. van Someren, J. S., Y.F. Barnard. (1994). *The think aloud method: A practical approach to modelling cognitive processes*. Londen: Academic Press.

Schmidt, K., Phillips, C., & Chuvakin, A. (2012). *Logging and log management: The authoritative guide to understanding the concepts surrounding logging and log management*. Newnes.