Masaryk University
Faculty of Informatics


# Log management in distributed environment

Bachelor's Thesis

**Ondřej Lomič**

Brno, Spring 2017

# Declaration

Hereby I declare that this paper is my original authorial work, which I have worked out on my own. All sources, references, and literature used or excerpted during elaboration of this work are properly cited and listed in complete reference to the due source.

Ondřej Lomič

**Advisor:** Andriy Stetsko

# Acknowledgement

I thank Zuzana Zatrochová and Andriy Stetsko, for providing me a possibility to write this thesis under their supervision and for their friendly approach.

And I thank you, Eli, my love. For without you this wouldn't be.

# Abstract

This work focuses on collection and processing of logs in distributed environment with help of open-source, free of charge software. The main goal is to create log management system, which will satisfy requirements of common log management goals, such as debugging in production. That was solved by designing generic architecture. It is divided into components, each component can be substituted with a third-party tool, as long as the tool satisfies requirements of given component. Overview of suitable third-party tools follows. These tools are used for creation of prototype log management system, which is tested for required properties. Custom software named Caribou was implemented to allow testing of prototype log management system, but it can be also used for deploying and managing system in production.

## Keywords

logs, architecture design, open-source, distributed system, ELK, Kafka

# Contents

# 1 Introduction

Logs are important part of nearly any application, they can be used to monitor an application in production, debug errors, investigate security breaches or satisfy regulatory compliance [1, p. 75-90]. On the other hand, management of logs evolves together with developed application and logs are usually simply stored in files and reviewed manually during the development [2, p. 26-30]. However, required functionality as filtering, alerting or even collection [3] of logs can get too complex very quickly during the deployment. It may get difficult soon to add new features without breaking the old ones, scale this solution for more data or even preserve current functionality while new log types are added.

There are many third-party tools to help with log management, most of them are paid. Splunk [4] provides most features and has the best processing capabilities, but it is also the most expensive product [5, 6]. There are several options when looking for cheaper alternatives. One of less expensive paid products with limited set of features can be selected. Another option is to choose one of a few free, open-source products or finally custom software can be implemented. No matter the choice, it is good to know what features you may require, so you do not end up with too complex solution or the other way around.

There are two books covering this topic, *Logging and log management* and *The Art of Monitoring* [1, 2]. The first one is more focused on theory, the second one on practical use of selected third-party tools, however both of them are paid. When looking for free of charge alternatives, you can find some useful blog articles by Wilder [7, 8], which can give you brief overview on software you can use. There are also other articles discussing this topic or comparing log management software [9, 10, 11], but many of them are rather trying to promote some software than making proper overview or comparison.

The main goal of this work is to provide information necessary to understand concepts of log management beyond simple collection into files. It describes common log processing goals and their requirements. Based on these requirements, generic architecture for log management is designed. It consists of five independent components, which can be substituted with already existing third-party tools,

as long as selected tool satisfies stated requirements of a given component. Lastly, there is a list of recommended tools suitable for use in the designed generic architecture.

That is followed by description of a prototype log management system. It is implementation of designed generic architecture and it is used to verify required properties, which confirms, that the generic architecture can be used for all common log processing goals.

This work is divided into two main parts. The first, theoretical part, consists of description of common log management goals, design of generic architecture and overview of existing tools. Second, practical part, focuses on implementing the prototype system and verifying required properties.

# 2 Designing the log management system

The first section describes common goals of advanced log management, which represent a wide variety of possible use cases [1, p. 75-90] [12]. Each goal is described as a set of requirements, which the designed system will meet.

It is also important to specify possible properties of such system, such as fault tolerance, before the design itself. Those properties are used to satisfy requirements during the design and many of them can be interpreted in more ways. Depending on that, they might or might not satisfy the requirements they should. So second section is specification of used properties during the design.

The third section describes design of generic architecture for log processing, which meets all discussed requirements, therefore it can be used for any of discussed common goals. This architecture consists of five independent components and all of them can be substituted with already existing third-party tool, as long as the tool holds required properties of given component.

The last section of this chapter is an overview of existing tools suitable for use in the designed architecture.

## 2.1 Goals and requirements

The most common goal of log management is debugging in production, on the other hand, there are many others, like monitoring user activity or collecting audit logs [12]. Moreover, collection or any manipulation with logs comes with specific requirements, which are stated under separate goal. All goals are described in detail below and each of them implicates specific requirements. Their overview can be seen in Table 2.1.

**Manipulation with logs** itself comes with specific requirements, common for all other goals, which originate from nature of logs. Logs are evolving and changing with developed application, new versions produce different logs, therefore system for log management should be **easily changed** and ideally should be able to process logs from different versions at the same time.

| Goals | Architecture requirements |
|---|---|
| log manipulation | • easy to change<br>• should not affect developed application<br>• as simple as possible |
| debugging in production | • full text search<br>• ad hoc queries on indexed data<br>• visualize log statistics |
| prediction of errors | • alerts<br>• no data losses<br>• low latency |
| monitoring users activity | • process large amounts of data |
| audit logs | • no data loss despite failure<br>• limited access |
| statistics, reports | • long term storage |

Table 2.1: Overview of common goals and their requirements

Another common requirement is that a fault of log processing system **should not affect developed application** in any way and collection of logs should interfere with application as little as possible. That is because creation of logs is more like side product of application and mostly their processing is not system critical[1].

Last requirement is keeping log processing **as simple as possible**, since most effort should be put into developing application, not into managing its logs.

**Debugging in production** requires ability to effectively filter and view individual logs as well as view statistics and trends. That means the designed system should provide way to run **ad hoc queries on indexed data**. It should also be able to **visualize log statistics** into graphs, because that is the quickest way to find both trends or sudden changes, which both can be indicator of an error. And since logs often contain text messages describing event in human language, **full text search** is also important.

**Prediction of errors** is the next step after debugging in production. It is solved by automated triggering of **alerts**, which evaluation is based on incoming logs [13]. It requires **low latency** between log creation and log processing, as low as soon the system should react, and also **no data losses**, because incomplete data means untriggered or false alarms.

**Monitoring users activity** is another common goal which can aim for collecting simple statistics as well as for computing complex calculations, like visualizing users interactions into graph. Such data can be useful for business intelligence as well as reports for customers. Because there is usually large number of users, there is also much more logs to process, so the system has to be able to **process large amounts of data**.

**Audit logs** are usually security related logs which guarantee that specific events happened and which non-existence guarantees the opposite. **Access** to such logs should be **limited** only for authorized personal. Audit logs must be saved to more locations to ensure **no data loss despite failure**.

1. Audit logs may be exception [12], see the goal **Audit logs**

**Statistics, reports** are usually simple aggregations, but performed on a large set of data covering long period of time. The process of creating reports is not discussed here, but **long term storage** is required to store such data.

## 2.2 Specification of properties

During the design part, each requirement will implies various architecture/component properties, e.g. fault tolerance. Because some of these properties can have multiple interpretations and therefore might or might not fully satisfy selected requirements, they are specified before the design itself specifically for this system. Solution common for all components, which would satisfy given property, is provided where possible.

**High availability** is property which ensures, that time when system is running and responding to requests, is equal or higher then specified percentage of total elapsed time.

$$\frac{\text{time available}}{\text{total time}} > \text{required availability} \tag{2.1}$$

Required availability is usually specified in nines [14], e.g. three nines mean 99.9% of total time the system has to be available, which would allow maximum of ten minutes of unavailability per week. Note that scheduled maintenance might be included or excluded from the total time, depending on requirements.

That means, that highly available system should be responding to requests even under high load and failover or redundancy is needed to keep system available even when some of its parts fails.

**Fault tolerance** ensures no data loss even when part of the system permanently fails. Data are replicated across multiple machines and in case of single failure, there is always copy on a different one.

Replication factor is a number which specifies number of data copies. There can be up to *n-1* failures for *n* copies without loos-

ing any data, on the other hand it requires $n$ times more storage space.

Also note, that fault tolerance limits availability and the other way around. For example, with three storages and replication factor of three, one failure won't cause data loss, but as long as there will be only two storages, the system won't be available for writes, since at least three storages are needed. Either one storage can be added or replication factor decreased to two to allow fault tolerance and availability at the same time.

**Reliability** simply means no data loss when system is running without failures. What is considered a failure needs to be defined specifically to every system, e.g. it does not have to be clear, if restart of in-memory database should mean loosing stored data or if it should preserve the data in some way. Or if raised exception during parsing a log should mean discarding the log or saving the original for later inspection.

**Scalability** allows us to increase performance by either improving hardware on a single machine (vertical scalability) or adding more instances of the system (horizontal scalability). Since vertical scalability is possible for every system and limited in the sense how much you can scale [15], when scalability is mentioned throughout this work, it is always the horizontal one.

Note that even when the system is not scalable internally, if processed data can be divided into independent parts (e.g. by customer), then the parts can be processed in parallel on independent systems and scalability is achieved.

**Dynamic schema** allows databases to work without predefined schema, it is instead created dynamically based on inserted records. That can simplify database management when storing logs, because they can contain structured part [16, p. 15], which should be parsed as record fields and which parameters' names can change over time.

**Full text search** is not only ability to search through non-structured text, but also index such texts for faster results. Note that not

only existence in a text can be important, but also amount of occurrences or proximity of searched terms.

**Near-real time** systems have latency in order of seconds [17, 18]. There is no specific limit, usually the data are sent as soon as they are ready, with some performance considerations, e.g. flushing to disk in one second batches.

## 2.3 Designing generic architecture

Instead of designing unique architecture for all possible combinations of goals and their requirements, there is going to be one universal, generic architecture, which will meet all requirements from previous section[2]. The design will be iterative, for each requirement the architecture will be extended with only necessary components and properties, so it is clear why are specific components and properties important and also to keep whole architecture as simple as possible[3].

### Collection

First requirement to consider is that processing of logs **should not affect developed application** and the best way how to minimize interference is to do log processing on a separate server. Also, an application is often distributed on more servers, so some kind of log forwarding is needed. The first component of designed architecture is a *collector*, which is responsible for collection of all logs from monitored application and forwarding them to a log management server. This can be usually part of developed application, since change of setting in logging framework is enough, but other third-party tools can be installed to collect and forward various information about system OS, databases etc.

The second component, which ensures separation of log processing, is *queue* and it is located on a log management server. At all times production of logs should be lower than their consumption, otherwise logs will be lost or they will accumulate on application servers and

2. For overview see Table 2.1.
3. KISS principle [19] and also requirement of log processing (page 3.).

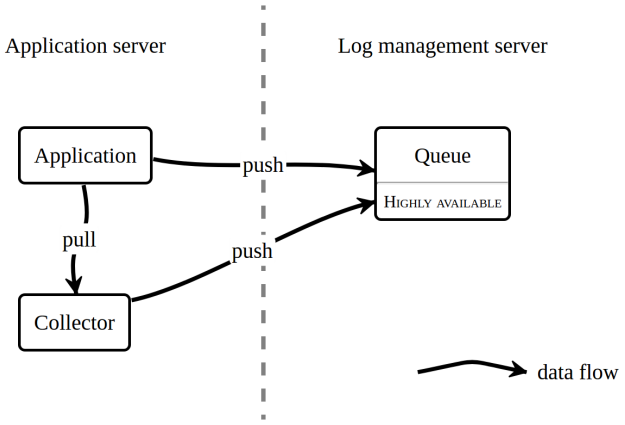


Figure 2.1: Collection of logs

might cause buffer overflow. Since speed of log processing is limited and there might be big spikes in log generation (especially when errors happen), some kind of queue is needed. Presence of queue also means, that if any component but *queue* or *collector* fails, the application won't be affected and no logs will be lost. The *queue* should be highly available, so all request at all times are accepted and there is no data loss.

Note that each component does not have to be represented by a separate tool, e.g. application's logging framework can be reconfigured to send logs directly to *queue* and therefore serve as *collector*. However, that is not always possible for other third party tools like databases, therefore *collector* might be needed anyway (see Figure 2.1).

### Normalization

Each log contains timestamp and usually both unstructured human readable message and structured part. Unfortunately, there is no universally used format, there are some standards, but different applications (or even different parts of one application) can use different log formats [16, 20, 21]. It is necessary to normalize logs for simplification of further log processing [22], even timestamps in different formats would add huge complexity to the rest of the processing (e.g. when computing latency between two different logs).

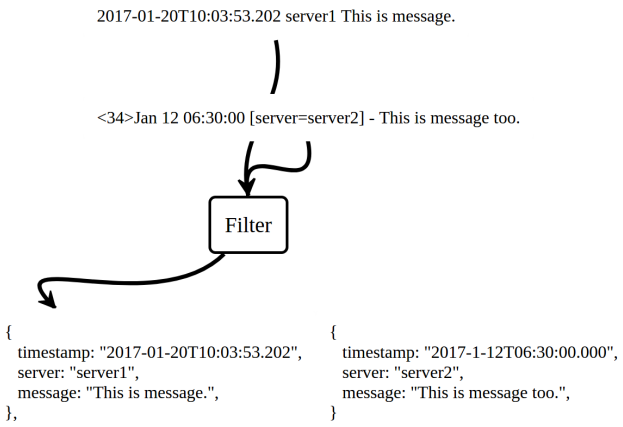


Figure 2.2: Parsing and normalization

To normalize logs and make designed system **easy to change**, component *filter* filters, parses and normalizes logs from *queue* (Figure 2.2). Addition of new log format, or change of existing one, only means adding new parser to *filter* and no other changes. That means, that even with many different log formats the system is very easy to change, and because of normalized filter output the further processing is kept as simple as possible.

### Storage and visualization

The main requirement for debugging in production is being able to run **ad hoc queries on indexed data**, so *database* with appropriate properties should be selected.

Also, even with preceding normalization, logs do not have to have strict schema. Each log can have structured part [16] with named parameters, key-values, which should be parsed too. There can be different parameter names (keys) for different logs. So the *database* should have dynamic schema, to simplify managing of logs.

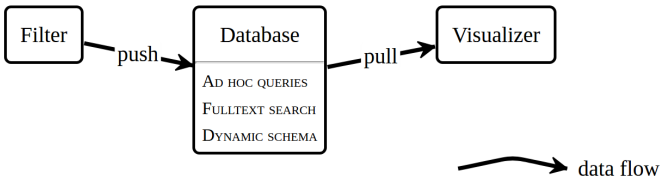


Figure 2.3: Storage and visualization

And lastly, because log's message is unstructured text, the database should provide possibility of full text search.

The last component is a *visualizer*, which is responsible for running predefined queries on the *database* and shows the results in a form of graphs. That way trends and anomalies can be quickly detected and analyzed (Figure 2.3).

### Alerting

There are two distinct parts of alerting, the first part is to create alert messages and the second one is to deliver them to specific outputs.

Creating alert messages can be as simple as filtering logs which contain word error, but aggregating is often involved too. E.g. the filter's condition can be based on an average of a log field value during last hour, but also outputted logs can be joined together into one summary per hour.

Filtering/aggregation can be done in two different ways. The first one is stream (online) processing, which means that logs are processed as they come, before they are saved into database. The advantages are, that each log is processed only once, the latency is as low as possible, the aggregation is not dependent on the database and it can be part of already existing component, *filter*. On the other hand, the computation is generally more difficult, because logs do not have to be processed in order and filter has to store a state, since logs cannot be reprocessed.

The other way of aggregating data is batch (offline) processing. Some component would periodically query the database and aggregate the returned results. It has the opposite advantages than stream

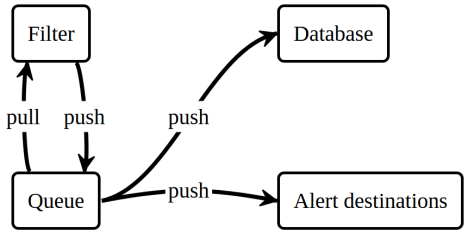


Figure 2.4: Output separation

processing, but because we are trying to achieve **low latency**, batch processing is not appropriate choice.

### Output separation

Note, that since now, the parsed logs outputted by filter went straight to the database. The alerts can be sent directly from *filter* to *database* and other outputs too, but failure of one output would mean halting all other ones. To achieve output separation, parsed logs and alerts from *filter* are sent back to *queue* (see Figure 2.4). Independent connection from queue is created for each output, which means that failure of one won't affect the other, it will only cause accumulation of logs in the queue.

The queue, as a central communication hub, also ensures independence of all other components on each other. New component can be added or existing one can be changed or entirely switched by different one arbitrary, only log format has to be the same.

### Management

Even while trying to keep whole architecture as simple as possible, it consists of five components. *Collector* is going to be deployed together with application, because for each application there should be one, but the rest has to be deployed independently. *Database* and *visualizer*, although dependent on each other, are usually two separate tools and

although *filter* and *queue* can be implemented as one tool, they are often separated too.

That leaves us with four tools, which are to be deployed on different servers, some in multiple instances, to provide scalability, high availability or fault tolerance. To keep deployment and configuring of these tools simple, there should be some kind of manager, which will allow management of deployed tools, their configuration from one place and monitoring of whole system.

### Remaining requirements

**No data losses** are ensured when both connections between components and components itself are reliable.

**Low latency** is guaranteed when all components work in near-real time.

**Large amounts of data** can be sooner or later only processed in parallel, which means that all components have to be scalable.

**Fault tolerance** is relevant to components, which store or process data. *Database* and *queue* can achieve fault tolerance using replication, fault tolerant filter usually relies on its source, in case of failure it reprocesses lost messages [23, 24].

**Limited access** can be implemented most easily for whole log management architecture, securing the connection from *collector* to *queue* and restricting access to whole system. If more advanced security is needed, as per-record access or different user roles, appropriate features have to be provided by the *database* component.

**Long term storage** is simply added as another component, filter can send data both to *database* for searching and to *long term storage*.

### Complete architecture

The overview of whole generic architecture is shown in figure 2.5. Logs are collected from application by *collector* and securely send to *queue*.

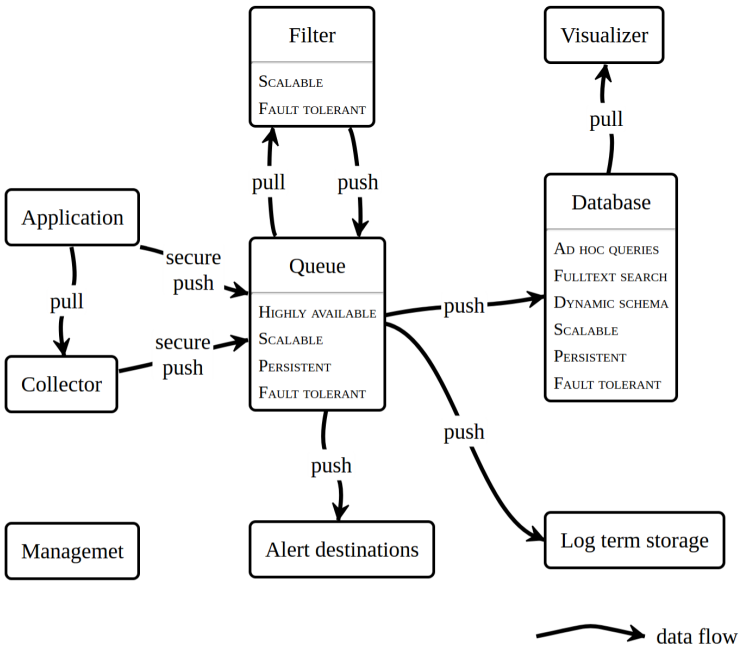


Figure 2.5: Complete generic architecture with all properties

*Filter* is used to aggregate and normalize logs, and to create alerts. Data from *filter* are send back to *queue*, where they are independently forwarded to their destination. Logs can be pushed to *database*, which indexes them to make search through *visualizer* fast, to quickly see statistics, trends or individual logs.

## 2.4 Software overview

There is already a lot of third-party tools, which can be used for log processing. Since it is easier to reuse existing tool instead of implementing a new one, there is an overview of existing suitable software.

To keep this overview simple, a detailed overview of recommended tool per component is provided, always complemented by other alternatives. Also note that all selected software is open-source and free of charge for commercial usage, there is a summary of paid software at the end of this section.

### Queue

There are many properties required by design and Kafka [25] is one of few tools, which satisfies all of them. Moreover, Kafka is ideal as fault tolerant scalable queue, because it was designed for that specific purpose [26].

Kafka can be described as distributed streaming platform, which means, that it cannot manipulate with individual records (logs), but it processes them as a stream. That allows many performance optimizations which grants Kafka high throughput even while all records are saved persistently to disk.

#### Alternatives

RabbitMQ [27], ActiveMQ Artemis [28] and ZeroMQ [29] are popular and widely used message queues, which are able to satisfy our requirements. However, they provide many other features, therefore they are unnecessarily complex. Persistence and replication are implemented per message, not per stream, which makes them much slower compared to Kafka [30].

### Collector

The simplest and easiest way how to forward logs is to use logging framework directly. The only requirement is that used logging framework supports outputting logs to selected *queue*, but there is mostly such support for recommended queue, Kafka. If such support is missing, it is always possible to extend logging framework by another handler.

#### Alternatives

Filebeat [31] is an example of log shipper. Files with logs are specified in simple configuration file together with output specification. It is lightweight, supports Kafka out of the box, runs on any OS and has no dependencies.

Telegraf [32] is mainly used for collection of metrics, but (as many other tools) it can be configured to collect log messages as well.

### Filter

Logstash [33] is the first of recommended tools. It is not fault tolerant, but on the other hand, it is really simple to use and has many predefined inputs, filters and outputs. It also supports easy way to parse log messages into structured data. It does not provide any advanced features, but it is appropriate tool to start with.

Samza [34] is example of advanced stream processor. It works on top of Kafka, is managed by YARN (resource manager) and provides features as durability, fault tolerance and it can store and update internal state based on incoming events. That allows aggregations of multiple events and other advanced metrics.

#### Alternatives

Storm [35] and Flink [36] provides very similar features as Samsa and they are good alternative with no major disadvantage.

Spark [37] is a tool for processing batches. However, it provides library called Spark Streaming, which divides incoming stream into small batches and processes them in Spark. That allows running same algorithms on both batches and streams or even combine them.

Fluendt [38] and Heka [39] are alternatives to Logstash, as filters without fault tolerance, but with simple configuration and deployment.

### Database

Elasticsearch [40] is a perfect fit for our requirements. It is focused on full text search and it was designed to be distributed. It also supports dynamic schema, ad hoc queries with aggregations, there is big active community and a very good documentation.

#### Alternatives

Solr [41] is very similar database as Elasticsearch, mainly because both of them are using the same full text search library, Lucene [42]. The main difference is, that there is no up to date visualizer for Solr, there is old fork of Kibana called Banana [43], but it is not actively developed.

### Visualizers

Kibana [44] is an official visualizer for Elasticsearch. Grafana [45] is based on Kibana's core, it is focused on InfluxDB, but it also supports viewing and aggregating logs from Elasticsearch. Both applications support creating and managing dashboards with predefined visualizations, graphs and metrics, but Kibana excels in viewing and custom filtering of individual logs. On the other hand, Grafana implements user management, authentication and various permissions.

That is why both of these application are recommended. Grafana can be used for showing statistics to customers, meanwhile Kibana can serve developers as a quick and easy to use debugging tool.

### Alerting

As suggested when designing architecture, it is better to create alerts in *filter* as a part of stream processing, because of much lower latency. Created alerts are sent to *queue*, where they can be independently forwarded to various outputs.

Alternatives

ElastAlert [46] is a simple application for generating alerts using batches, it can be only used with Elasticsearch.

### Management

Unfortunately, there is no software which would allow easy deployment, configuration and monitoring of recommended applications. There are two ways to solve this issue, the first is to use extensible management software, make sure that it provides required functionality and then write custom plugins for unsupported software. For now, Ambari [47] is the the only software, that has this required functionality.

The second option is to implement custom software. Already preexisting software can used for specific parts, for example Salt or Puppet can be used for simple deployment and configuration, but a monitoring of whole cluster would still have to be implemented.

### All in one

Graylog [48] is a software for complete management of logs. It internally uses Kafka, Elasticsearch and MongoDB [49], comes with its own filtering capabilities, alerts, simple monitoring of whole cluster, collector and even its own log format. It satisfies all required properties, but everything is managed through web interface, which limits some of its original component settings. For example, log parsing capabilities are limited compared to Logstash or it is very hard to add unsupported features. But for many deployments it can be sufficient solution.

### Paid solution

The second to none paid software for processing logs is Splunk [4]. However, it is also the most expensive one, but for some environments the price can be acceptable and purchasing complete software and just send logs into it might be better than spend time deploying free of charge software. For example, when development of log management has to be setup quickly with guaranteed result, this would be better solution then starting to develop a custom, open-source one.

### Alternatives

Elastic [50] is a company, which provides free of charge Beats (various collectors), Logstash, Elasticsearch and Kibana. These components deployed together are usually called ELK stack and Elastic company provides support for deploying ELK and also paid extensions like Shield, which provides security access to Elasticsearch.

Loggly [51], NXLog [52], Nagios [53], Stackify [54], Sematext [55] or Lucidworks [56], all of them provide their own software for log processing or even only consultations about how to deploy your own log management software.

# 3 Implementation, deployment and testing

To make deployment and therefore testing of log processing system simple, a software for deploying, configuring and monitoring all used tools is needed. As mentioned in software overview in previous chapter (page 18), it would be possible to use Ambari [47] and extend it to support selected tools. On the other hand, it would require additional software to allow automated testing and Ambari would be another complex tool, which would have to be managed.

Therefore the other option was selected, implementation of custom tool. It is named Caribou and it was created from scratch. It focuses to be simple to use and easily extensible and customizable.

There are three main sections. The first one contains description of implemented Caribou, its usage and features. The second section describes the prototype log management system and its deployment. Finally, the prototype system is tested for required properties. If these properties are met, system meets its requirements and therefore can be used for any of discussed common goals from first chapter (page 4).

## 3.1 Implementation of Caribou

Caribou is implemented in python. Framework Django [57] is used to provide additional functionality, as easily accessible database or web server. SSH protocol is used to access remote hosts and Caribou, as well as managed tools, are supposed to run on Linux. No other libraries or dependencies are used, which makes it easier to learn and use.

### Component abstraction

Caribou aims not only for simplicity in design, but also for simplicity in usage. To allow that, configuration and management of used tools is independent. That allows you to run the same test with multiple configurations as well as run the same configuration among multiple tests.

That is possible because of component abstraction. Component's name describes multiple tool's instances across several hosts, but with

the same configuration. That corresponds with horizontal scaling, all tool's instances run on different hosts, but their configuration is still the same.

Hosts are specified for each component name, together with specific tool and its configuration. Like that an instance of a class `Layout` is created. The layout contains complete description and configuration of all tools, which are to be managed. It can be used in tests or to create an instance of a class `Manager`, to manage tools manually.

Below is short but working example of usage. Firstly host is defined for filter component, then Logstash is set as specific tool to be managed. Further manager is manually created and Logstash, being the only tool, is started with default configuration.

```
layout = Layout({"filter": ["10.0.10.14"]})
layout.filter.set_app(Logstash)
manager = Manager(layout)
manager.start()
```

### Used terms

There is an overview of used terms during detailed description of Caribou, to clarify and distinguish them.

**Tool** is a third-party software, which is used to manage logs.

`Application` is a python class, which represent generic tool and provides general methods for tool's management.

**Handle** is a common name for any of `Application`'s subclasses, which implements `Application` abstract methods and therefore can be used to manage specific tool.

### Layout configuration

Layout is a configuration of whole cluster. It is defined in one file for static deployments, but it can be also created dynamically for purpose of tests.

At the beginning, components and hosts are defined.

```
layout = Layout({
    "filter": ["10.0.10.14", "10.0.10.15"],
```

```
    "db": ["10.0.10.14"]
})
```

This is the only place where hosts are defined, which makes it very easy to change them. Further in the layout configuration, hosts can be accessed as `layout.`*`component_name`*`.ips`.

This is followed by assignment of handlers.

```
layout.filter.set_app(Logstash)
layout.db.set_app(Elasticsearch)
```

Handlers `Logstash` and `Elasticsearch` are python classes, they are described in detail in `Application` subsection (page 26). For now, the important information is that they contain class `Settings`. When method `set_app()` is called, instance of corresponding `Settings` is saved to `layout.`*`component_name`*`.settings`, and its parameters are set to default. `Logstash.Settings` can be viewed to get information about various parameters and like that, Logstash's parameter value can be changed and it will be used when filter will be (re)configured.

```
layout.filter.settings.pipeline_size = 1000
```

### Manager and parallel execution

Manager is used to create all handlers' instances and run their methods in parallel (or other synchronization modes). Firstly, manager is created with a specific layout.

```
manager = Manager(layout)
```

It creates instances of all handlers specified in the layout. All handler's methods can be called through manager, for example `manager.start()` would run method start on all handlers. To allow method execution on specific components, manager also implements methods `filter` and `exclude`, which return copy of original manager, but with limited set of handlers to manage.

```
manager_of_all_filters
       = manager.filter(name="filter")
manager_of_applications_not_on_ip1
       = manager.exclude(ip="ip1")
manager_of_filter_on_ip2
       = manager_of_all_filters.filter(ip="ip2")
```

But the main advantage of manager is making parallel execution much easier.

### Parallel execution - default behavior

By default, each method execution is done in parallel on all handlers and they are immediately synchronized when the result is returned.

```
manager.start()
manager.stop()
```

Like this, all components are started, and only when all of them are running, then they are stopped.

### Parallel execution - chained

There is also a way, how to run handlers' methods without synchronization in between. That is beneficial when we want to speed up more independent methods.

```
manager.chained(
    lambda manager: manager.start().stop()
)
```

Like this, when component is started, it is stopped immediately afterwards, without any synchronization with other components in the middle. Of course, all handlers are synchronized at the end.

### Parallel execution - parallel

This type of execution allows to run more managers, and therefore more methods in parallel.

```
manager.parallel(
    lambda m: m.filter(name="filter").send(),
    lambda m: m.exclude(name="filter").receive(),
)
```

Each parameter of method `parallel()` is a function, which is executed in parallel with copy of original manager. Like that, two different methods can run on different parts of cluster easily.

### `Application`

Each custom handler (`Logstash`, `Elasticsearch`) is a python class which inherits from `Application` class. This general class implements most methods, which makes adding new custom handlers rather simple, but all of its methods can be overwritten, to be adjustable to custom handler needs.

### Execution of commands

Because Caribou and all managed tools are supposed to run only on Linux, shell commands are used to manage them. There is a set of functions, which format parameters to appropriate string, which is run in a shell. This solution is dependent on operating system, but on the other hand it is very simple to learn, change or debug, which it aims for.

### Installation and structure of folders

Installation is also kept as simple as possible. In settings file, there are two specified folders. Tools should be downloaded and extracted into the first one, and the second one specifies where tools are installed, it is on the same path for all hosts. The only requirements for installation are that SSH connection without password is set up to managed hosts and logged in user has rights to access the folder, where the tools are to be installed.

Installation is rather a simple process, firstly folder with component's name is created. Four other folders are created inside of it, `app/`, `logs/`, `data/` and `configs/`. App folder contains copies of extracted files, tool's logging should be redirected into `logs/` folder, tool's data and configurations correspondingly. That provides us with separation of all tool's data from the rest of the system and also gives us ability to use general methods, for example showing all logs simply means showing all files in corresponding folder, or deleting whole component with all data means deleting whole component folder.

### Running applications

Managing of components is done by custom django management command `clm`.

```
python manage.py clm start
python manage.py clm --layout l_name --filter ip=ip1 stop
```

This command corresponds with simple usage of `Manager`. Specified (or default) layout is loaded, manager is created, filters are applied and finally selected method is invoked. That allows simple control of whole deployment as well as individual tools.

Note, that there is no continuously running service to provide tool management. When a tool is started, its process id (pid) is saved to database. When stop is called (or any other function), while an `Application` is initializing, it checks the database for pid of already running tool. If so, it uses that pid and it does not create another tool.

### Adding handlers

Since generic class `Application` provides most of the functionality, adding support for specific tool is rather simple. Class `MyTool` needs to be created and it has to inherit from `Application`.

```
1 class MyTool(Application):
2     installation_folder = 'my_tool-v1.5.1'
3     start_command = '{self.app_path}/bin/start_script ' \
4                     '--param {self.settings.param}'
5
6     class Settings(object):
7         def __init__(self):
8             self.param = 'default'
```

**Installation folder** specifies name of downloaded and extracted folder, which contains MyTool's files.

**Start commad** is used to start the tool.

**Settings** contains parameter `param`, which can be changed in layout configuration and is used when tool is started.

### Additional features

Caribou contains more additional features which simplify system management, such as web overview of managed tools, support for periodic health check with alerting and layout custom methods. These features are described in detail in an attachment, Guide to Caribou.

## 3.2 Deployment

The specific deployment, which is used for testing, is described here. It corresponds with the generic architecture designed in the first chapter and it uses tools from software overview.

Below is definition of whole deployment, as it is written in Caribou.

```
# components and hosts
layout = Layout({
    'queue': ['10.0.10.18'],
    'zookeeper': ['10.0.10.18'],
    'parser': ['10.0.10.18'],
    'forwarder': ['10.0.10.18'],
    'db': ['10.0.10.19', '10.0.10.20'],
    'visualizer': ['10.0.10.18'],
    'monitoring': ['10.0.10.18'],
})

# setting applications
layout.queue.set_app(Kafka)
layout.zookeeper.set_app(Zookeeper)
layout.parser.set_app(Logstash)
layout.forwarder.set_app(Logstash)
layout.db.set_app(Elasticsearch)
layout.visualizer.set_app(Kibana)
layout.monitoring.set_app(CaribouServer)

# custom application settings

# configuration of parser, which converts text messages to json
layout.parser.settings.pipeline_config = input_(
```

```
    kafka_input(
        bootstrap_servers=Kafka.format_ips(layout.queue),
        topics=['logs'],
    )
) + filter_(
    # parse message to fields
    grok_filter(
        match={
            'message': '\A<%{LOGLEVEL:loglevel} ?> '
                       '%{TIMESTAMP_ISO8601:timestamp} '
                       '%{NOTSPACE:host} '
                       '%{NOTSPACE:component} '
                       '%{GREEDYDATA:message}'
        },
        overwrite=['message']
    ),
    # set timestamp to @timestamp
    date_filter(
        match=['timestamp', 'ISO8601'],
        remove_field=['timestamp']
    ),
    # parse key values into data
    kv_filter(allow_duplicate_values=True),
    # remove key values from message
    mutate_filter(
        gsub=['message', '\w+=\[.*?\] ?', ''],
        remove_field=['@version'],
    )
) + output_(
    kafka_output(
        bootstrap_servers=Kafka.format_ips(layout.queue),
        topic_id='parsed_logs',
        codec=json_codec()
    )
)

layout.forwarder.settings.pipeline_config = input_(
    kafka_input(
```

```
63          bootstrap_servers=Kafka.format_ips(layout.queue),
64          topics=['parsed_logs']
65      )
66  ) + output_(
67      elasticsearch_output(
68          hosts=layout.db.ips,
69          index='logs-%{+YYYY.MM.dd}'
70      )
71  )
```

As can be seen from the configuration, Kafka is used as a queue. Running Zookeeper is a requirement for Kafka, it is used for managing Kafka's configuration. Logstash both parses logs and forwards them from queue into database, because current version of Kafka Connect does not support newest release of Elasticsearch, which is used as database. Logs are visualized with Kibana and CaribouServer is Django's lightweight server, which allows quick overview of cluster health through web interface.

Note that most of the layout configuration consists of Logstash's setting named `pipeline_config` (line 21 and below). It is configuration of Logstash's inputs, filters and outputs, written with help of created library, which transforms python code into Logstash configuration string. More information can be found in already mentioned attachment, Guide to Caribou.

### Caribou installation and deployment

Caribou and all managed tools run only on Linux, specifically on Ubuntu 14.04 or Ubuntu 16.04. To run Caribou, it has to be extracted from archive and python 2.7 and Django 1.9.4 has to be installed.

Next step is to modify file with settings, located at `cluster_management/cluster_settings.py`. There are three settings, which has to be changed:

**ssh_login_name** is a name of user, which can access managed hosts through passwordless ssh connection.

**installation_path** is a directory where components are installed, it has to exist on each of managed hosts. User, logged through

ssh, has to have rights to read files, write into them and execute them.

**installation_files_origin_path** is a directory on the host, where Caribou is installed. All used tools have to be downloaded and extracted into this folder.

Finally, command `python manage.py clm reset` can be called to install, configure and start whole system. There is a web interface with quick overview of system health on the host were Caribou is installed, accessible through port 8000. Or a command `python manage.py clm is_running` can be run to get quick info, that all components are running.

More information relating to installation and configuration of Caribou can be found in an attachment, Guide to Caribou.

## 3.3 Tests

Following tests of the deployment specified above do not only verify functionality, but also required properties of individual components and therefore all requirements stated at the beginning of this work[1].

At the beginning of any test, the whole deployment is reinstalled, which means that all data are removed and therefore any test cannot affect the next one. Also, all tests are automated, `python manage.py test_properties` command executes all tests one by one.

Results of tests shown below are averages of five test runs. All results are shown in an attachment named results of tests.

### Functionality test

This test is generic, it can be applied to any layout. It is used for verification of components' configuration by executing as many `Application` methods as possible. That is allowed because of decorator `@check_status`.

Any component is always in one of three statuses, not installed, running or waiting (installed, but not running). Decorator `@check_status(status)` can be added in front of any method in an `Application`

1. Overview of requirements can be found in table 2.1 on page 4.

class. Every time, before such method is executed, component's status is checked and if it does not correspond with required status, exception is raised.

When running functionality test, all components are gradually removed, installed and then started. For each status, methods with corresponding decorator are executed. If these methods raise an exception, or cause component failure, test is unsuccessful.

This test does not check return values of used methods, but it is nevertheless useful, since many incorrect configurations will cause component failure and methods raise exception when unexpected behavior is encountered.

Since configuration of tested system is valid, functionality tests were always successful.

### Testing scalability

At first, throughput is measured as time, which is required to process $n$ messages. Message is processed, when it is parsed and stored in database.

To test scalability, throughput is measured while log management system is gradually deployed on one, two and finally three servers, with all components deployed on each server. Number of messages is proportional to number of servers, therefore if three servers can process messages in the same time as one server, the system is fully scalable, if it takes three times longer, it is not scalable at all.

Figure 3.1 shows averages times to process six hundred thousand messages per server, together with theoretical optimal performance (100% scalability) and no scalability. Time is measured from moment, when test generator is started, till the queue is empty. Test generator startup time is included in measured time, it takes approximately 15 seconds. Queue is checked in 30 seconds intervals, therefore up to 15 seconds can be added to real test time. That corresponds with maximum of two percent error for the shortest test, which took 912 seconds.

Results show, that the time is increased by 20 percent with replication factor of one, but more importantly, higher replication factor does not significantly increase throughput. Even fully scalable system could have three times worse throughput for replication factor of three,

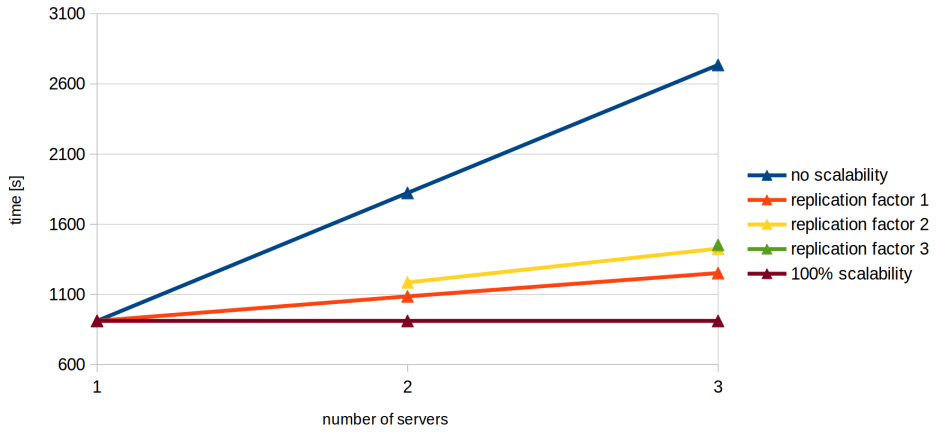


Figure 3.1: Throughput results

simply because three times more data needs to be saved. This shows that system is not fully scalable, but handles replication very well.

Scalability results in figure 3.2 shows, that designed system scales well (80% scalability), possibly even better for more instances. Although logs are independent, 100% scalability is not achieved because of overhead caused by synchronization and balancing load.

### Testing reliability

The system is considered reliable, when no logs are lost even during restart of log management system. Reliability test starts all components on each of three servers and creates continuous stream of one hundred logs per one hundred milliseconds to simulate load. Then all instances of given component are restarted one by one. There is no pause between stopping and starting a tool, but the test waits sixty seconds before restarting next component. Finally, stream of test logs is stopped and when queue is empty, messages in database are counted.

#### Logstash's reliability

There are four main parts in Logstash's processing pipeline, inputs, in-memory queue, filters and outputs. When Logstash is stopped,

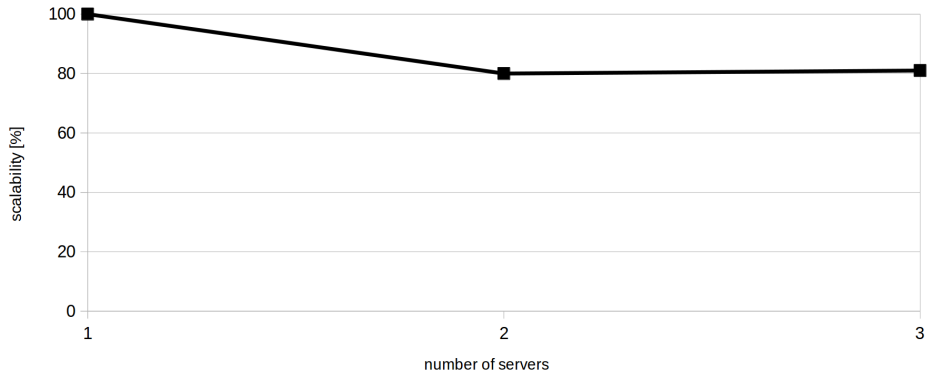


Figure 3.2: Scalability results

inputs are stopped and filtering and outputting of all stored logs is finished before Logstash exits, therefore no logs are lost.

Logstash's input commits received logs in five second interval. Commit marks logs in Kafka as processed. In case of Logstash's restart, it starts to process logs from first unprocessed. That creates window of five seconds, when logs can be received, filtered and outputted, but not committed, so restarted Logstash can reprocess these logs again. Amount of such logs can be lowered by decreasing commit interval.

Testing discovered, that approximately in one tenth of cases up to sixty thousands logs per restart were duplicated. There are only five runs of whole test, but Logstash is present in two instances (as a parser and a forwarder) and also reliability is tested for more different replication settings, which does not affect Logstash. Otherwise there was no loss of logs, therefore Logstash is reliable.

### Elasticsearch's reliability

Elasticsearch, as well as Logstash, supports graceful shutdown, however ten logs in average were duplicated per restart, up to one hundred. I could not find the reason for such behavior, however it is a small amount of logs and no were lost, therefore Elasticsearch is reliable.

#### Zookeeper's reliability

Zookeeper does not process any logs, Kafka uses it to manage its configurations. Zookeeper requires, that at least majority of its instances are running, otherwise it stops operating. However, Kafka's default timeout, one minute, was in all tests more then enough for Zookeeper to restart and reconfigure, therefore no logs were lost.

#### Kafka's reliability

Kafka itself is reliable, it flushes all new logs to disk before fully stopping. However, logs generated by producers[2] can get lost while Kafka is down.

There are two ways to prevent that. The first requires reconfiguration of producers, they have attributes `retires` and `retry.backoff.ms`. For example, these could be set to fifty and one thousand, which would force producer wait up to fifty seconds before logs would be lost.

The other way is to make Kafka highly available, specifically available even when one instance is down. This is the setting with which was Kafka tested, for more information see subsection Testing Kafka, on page 36.

### Testing fault tolerance

Fault tolerance is tested in very similar way as reliability, only instead of restarting a tool, the tool is killed, fully removed and reinstalled, which simulates permanent failure with quick replacement.

#### Logstash's fault tolerance

Logstash is not fault tolerant, which means, that it can either loss or duplicate logs on failure. However, amount of lost logs is always limited by size of internal buffer, which is 32 MB by default. That corresponds with approximately two hundred thousand logs.

Moreover, such failure would have to happen right after Logstash would commit logs received from Kafka and all of them were lost

2. Tools, which push logs into Kafka.

before delivered to output. Such scenario can happen, when logs are not outputted as fast as received and logs accumulate in Logstash.

During tests, outputs always processed logs faster then they were received, but one case, when over sixty thousand logs were lost. I could not manage to repeat this case, but I suppose that Elasticsearch could not receive logs fast enough, therefore they accumulated in Logstash and were lost when Logstash was killed. Otherwise, no logs were lost, only duplicated, because of the same reason as when restarting Logstash. (see Testing of Logstash's reliability, page 32).

### Zookeeper's fault tolerance

Zookeeper is designed to be fault tolerant, when three or more servers are running [58]. However, even tests with one or two Zookeeper's instances did not affect Kafka's log processing and none logs were lost.

### Elasticsearch's fault tolerance

Elasticsearch provides easily manageable fault tolerance settings. For each index[3], number of replicas is specified. Replica is a copy of original primary index, which is kept on a different instance than original. If there is not enough running instances of Elasticsearch — replicas are not created. If primary index with existing replica fails — the replica will become new primary index, and if possible, another replica is made on a different, available instance of Elasticsearch.

Tests proves, that Elasticsearch is fault tolerant. No logs were lost, when at least two instances of Elasticsearch were running and number of replicas was set to at least one. Only few messages were duplicated, the same amount as when testing reliability (see page 33).

### Kafka's fault tolerance

Kafka's fault tolerance works very similarly as Elasticsearch, except much more in depth knowledge is required to understand all settings. That is why all results of Kafka's measurements are discussed in next separate section, Testing Kafka.

---

3. Elasticsearch's index is logical part of database, similar to table in SQL databases

### Testing Kafka

It is important to mention, that the main goal of these tests is not to completely verify selected properties for all possible cases, there are several research papers and articles to analyze that [59].

The main goal is to setup and test Kafka's configuration to satisfy discussed properties for common cases, like failure of one instance only. The result is description of important configuration options together with specific verified Kafka's configuration, which can be reused.

#### Configuration overview

**Replication factor** sets number of log copies, which should be stored among multiple Kafka instances. If there is not enough of running instances, less replicas (copies) are made.

**Min-insync-replicas** sets minimum number of required replicas to accept an incoming log. If number of accessible replicas is lower than this setting, messages are refused. Also log delivery is confirmed only when at least min-insync-replicas confirmed its receipt, like this Kafka provides strong delivery guarantee.

Therefore, when there is at least $r$ replicas and also min-insync-replicas is set to $i$ where $r$ is greater or equal to $i$, up to $i$-1 failures can happen without loosing any data. On the other hand up to $r$-$i$ instances can fail, until Kafka stops being available.

To provide both availability and fault tolerance, replication factor is set to three and min-insync-replicas to two. System is fault tolerant, in case of permanent Kafka's instance failure, there is always a copy, as well as enough replicas to keep Kafka available.

#### Reliability and fault tolerance results

The results of both reliability and fault tolerance tests are the same, there is approximately one hundred lost logs per restart/kill of Kafka instance. The source of logs loss is not Kafka, but Logstash. Logs, which were being saved by just failed Kafka and which acknowledgment was not sent back, are not resent by Logstash to another Kafka instance,

even when Logstash's configuration `retries` is set. Instead only error message is logged and these logs are lost [60].

Otherwise Kafka is both reliable and fault tolerant, there is no data loss when Logstash is stopped before testing either reliability or fault tolerance.

#### Availability results

The first important notice is, that only Kafka is tested for availability, because it is the only tool which is connected to collectors and which has to be available. If any other component stops working, logs accumulate in Kafka and they will be processed when failed component is fixed.

Availability test works very similarly as reliability test, it restarts Kafka's instances one by one, it only waits before starting Kafka for thirty seconds. At the end, if test generator logs contain message with `NOT_ENOUGH_REPLICAS` error, the test fails.

Results of availability test confirms so far gathered information. This test fails, when number of min-insync-replicas is the same as replication factor, when one instance of Kafka is stopped, there is one less replica than required and Kafka stops being available.

### Verifying remaining properties

The rest of required properties has to be verified manually. All of them are properties of *database*, specifically ability to run ad hoc queries, providing full text search and dynamic schema.

Command `python manage.py test -a DatabaseTest` deploys prototype system on localhost and sends predefined logs into database. These can be manually viewed and manipulated through Kibana, which is by default listening on port 5601.

As can be seen from figure 3.3, logs can be simply filtered by time or by term (1, 2), furthermore full text search is applied (3) and also `data.correlation` (4) is a field, which was automatically parsed and added to log schema.

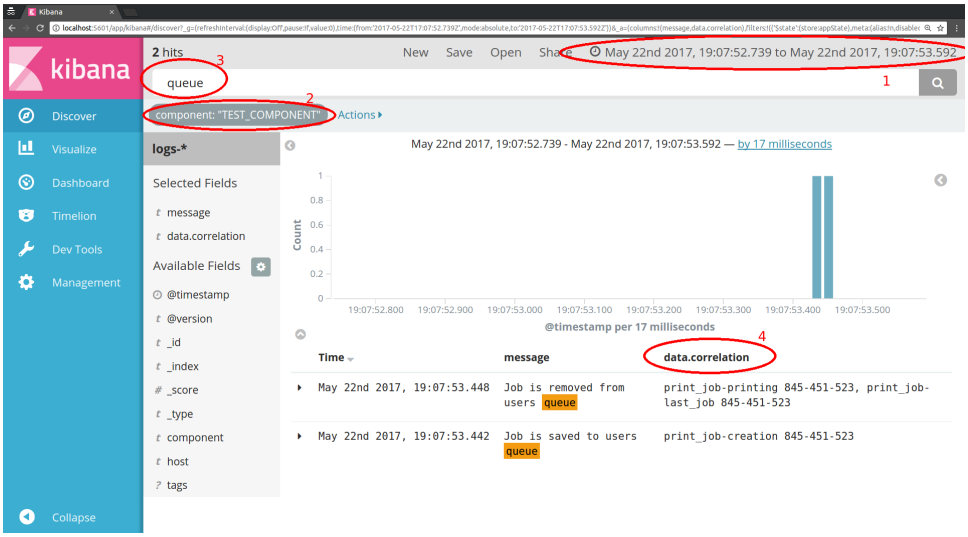


Figure 3.3: Kibana screenshot

# 4 Conclusion

My design of generic architecture was successful, all tests of prototype log management system confirmed required properties, which means that the architecture should be easily usable for all common goals of log processing.

Tests results also provided deeper insight into managed tools, for example increasing replication factor from one to three increased throughput only by sixteen percent.

Also Caribou, a tool for deployment, configuration and monitoring of designed system, was successful. It provided a way to simply change configuration of whole cluster and deploy various layouts for purpose of tests. It also allowed automated testing, through monitoring of deployed tools.

Whole solution is used in company Y Soft for debugging of their product, SafeQ. Deployed system was already expanded by stream processor Flink to allow correlation of processed logs and both designed system and Caribou proved to be easily extensible.

I hope that in the future the designed system will be used and furthermore extended and improved as a part of SafeQ's monitoring.

# Bibliography


1. CHUVAKIN, Anton A.; SCHMIDT, Kevin J. *Logging and log management: the authoritative guide to understanding the concepts surrounding logging and log management*. Waltham, Mass: Syngress, 2013. ISBN 978-1597496353.
2. TURNBULL, James. *The Art of Monitoring: a hands-on introductory book on the art of modern application and infrastructure monitoring and metrics*. [online]: Turnbull, James, 2014. ISBN 0988820242.
3. TAMURA, Kiyoto. Unified Logging Layer: Turning Data into Action [online] [visited on 2017-05-06]. Available from: `http://www.fluentd.org/blog/unified-logging-layer`.
4. *Splunk* [online]. Splunk Inc, 2017 [visited on 2017-05-06]. Available from: `https://www.splunk.com/`.
5. DREYFUSS, Josh. *Log Management Tools Face-Off: Splunk vs. Logstash vs. Sumo Logic* [online]. OverOps, Inc., 2017 [visited on 2017-05-06]. Available from: `http://blog.takipi.com/log-management-tools-face-off-splunk-vs-logstash-vs-sumo-logic/`.
6. UPGUARD. Splunk vs ELK [online]. 2017 [visited on 2017-05-06]. Available from: `https://www.upguard.com/articles/splunk-vs-elk`.
7. WILDER, Jason. Centralized Logging [online] [visited on 2017-05-06]. Available from: `http://jasonwilder.com/blog/2012/01/03/centralized-logging/`.
8. WILDER, Jason. Centralized Logging Architecture [online] [visited on 2017-05-06]. Available from: `http://jasonwilder.com/blog/2013/07/16/centralized-logging-architecture/`.
9. *Intro to Log Management* [online]. Logentries, inc., 2017 [visited on 2017-05-06]. Available from: `https://docs.logentries.com/docs/log-management`.
10. SISSEL, Jordan; KOOPMANN, Lennart. *OSDC 2014: Intro to Log Management* [online]. NETWAYS, 2014 [visited on 2017-05-06]. Available from: `https://www.youtube.com/watch?v=-ys68KHaZ6E`.

11. IDAN, Henn. *How to Choose the Right Log Management Tool?: Sumo Logic vs Graylog vs Loggly vs PaperTrail vs Logentries vs Stackify* [online]. OverOps, Inc., 2017 [visited on 2017-05-06]. Available from: http://blog.takipi.com/how-to-choose-the-right-log-management-tool/.
12. BUSTAMANTE, Michele Leroux. *The Ultimate Logging Architecture: You KNOW You Want It* [online]. NDC Conferences, 2014 [visited on 2017-05-06]. Available from: https://vimeo.com/113510470.
13. SALFNER, Felix; TSCHIRPKE, Steffen. Error Log Processing for Accurate Failure Prediction [online]. 2017 [visited on 2017-05-06]. Available from: https://www.usenix.org/legacy/event/wasl08/tech/full_papers/salfner/salfner_html/index.html.
14. RANTS, Maths. Nines of Nines [online] [visited on 2017-05-06]. Available from: http://www.joshdeprez.com/post/67-nines-of-availability/.
15. SHEVAT, Amir. Scale out versus scale up: How to scale your application [online] [visited on 2017-05-06]. Available from: http://spacebug.com/scale-out-versus-scale-up-html/.
16. GERHARDS, Rainer. *The Syslog Protocol* [online]. IETF Trust, 2009 [visited on 2017-05-06]. Available from: http://www.rfc-base.org/txt/rfc-5424.txt.
17. *Telecommunications: Glossary of Telecommunication Terms* [online]. Federal Standard 1037C, 1996 [visited on 2017-05-06]. Available from: https://www.its.bldrdoc.gov/fs-1037/fs-1037c.htm.
18. *Getting started: Basic concepts* [online]. Elasticsearch, 2017 [visited on 2017-05-06]. Available from: https://www.elastic.co/guide/en/elasticsearch/reference/current/_basic_concepts.html.
19. *KISS* [online]. Apache [visited on 2017-05-06]. Available from: http://people.apache.org/~fhanik/kiss.html.
20. *Logging Control In W3C httpd: The Common Logfile Format* [online]. W3C, 1995 [visited on 2017-05-06]. Available from: https://www.w3.org/Daemon/User/Config/Logging.html#common-logfile-format.
21. *Brief introduction of log file formats* [online]. Nihuo Software Inc., 2001–2014 [visited on 2017-05-06]. Available from: http://www.loganalyzer.net/log-analysis/log-file-format.html.

22. CHUVAKIN, Anton. Five mistakes of log analysis [online]. 2017 [visited on 2017-05-06]. Available from: http://www.computerworld.com/article/2567666/security0/five-mistakes-of-log-analysis.html.

23. RAMESH, Navina. Apache Samza: LinkedIn's Framework for Stream Processing [online]. 2017 [visited on 2017-05-06]. Available from: https://thenewstack.io/apache-samza-linkedins-framework-for-stream-processing/.

24. *Fault Tolerance Guarantees of Data Sources and Sinks* [online]. Apache Software Foundation, 2017 [visited on 2017-05-06]. Available from: https://ci.apache.org/projects/flink/flink-docs-release-1.2/dev/connectors/guarantees.html.

25. *Kafka* [online]. Apache Software Foundation, 2016 [visited on 2017-05-06]. Available from: https://kafka.apache.org/.

26. *Introduction: Apache Kafka$^{TM}$ is a distributed streaming platform. What exactly does that mean?* [online]. Apache Software Foundation, 2016 [visited on 2017-05-06]. Available from: https://kafka.apache.org/intro.

27. *RabbitMQ* [online]. Pivotal Software, 2017 [visited on 2017-05-06]. Available from: https://www.rabbitmq.com/.

28. *ActiveMQ Artemis* [online]. Apache Software Foundation, 2016 [visited on 2017-05-06]. Available from: https://activemq.apache.org/artemis/.

29. *ZeroMQ* [online]. iMatix Corporation, 2014 [visited on 2017-05-06]. Available from: http://zeromq.org/.

30. WARSKI, Adam. Evaluating persistent, replicated message queues [online]. 2017 [visited on 2017-05-06]. Available from: https://softwaremill.com/mqperf/.

31. *Filebeat: Lightweight Shipper for Logs* [online]. Elasticsearch, 2017 [visited on 2017-05-06]. Available from: https://www.elastic.co/products/beats/filebeat.

32. *Telegraf* [online]. InfluxData, Inc., 2017 [visited on 2017-05-06]. Available from: https://www.influxdata.com/telegraf/.

33. *Logstash: Centralize, Transform & Stash Your Data* [online]. Elasticsearch, 2017 [visited on 2017-05-06]. Available from: `https://www.elastic.co/products/logstash`.

34. *Samza* [online]. Apache Software Foundation, 2017 [visited on 2017-05-06]. Available from: `http://samza.apache.org/`.

35. *Storm* [online]. Apache Software Foundation, 2015 [visited on 2017-05-06]. Available from: `http://storm.apache.org/`.

36. *Flink* [online]. Apache Software Foundation, 2014–2017 [visited on 2017-05-06]. Available from: `https://flink.apache.org/`.

37. *Spark Streaming* [online]. Apache Software Foundation, 2017 [visited on 2017-05-06]. Available from: `http://spark.apache.org/streaming/`.

38. *Fluentd* [online]. Treasure Data, 2014–2017 [visited on 2017-05-06]. Available from: `http://www.fluentd.org/`.

39. *Heka* [online]. Mozilla, 2014 [visited on 2017-05-06]. Available from: `https://hekad.readthedocs.io/en/v0.10.0/`.

40. *Elasticsearch: The Heart of the Elastic Stack* [online]. Elasticsearch, 2017 [visited on 2017-05-06]. Available from: `https://www.elastic.co/products/elasticsearch`.

41. *Solr* [online]. Apache Software Foundation, 2017 [visited on 2017-05-06]. Available from: `http://lucene.apache.org/solr/`.

42. *Lucene* [online]. Apache Software Foundation, 2011–2016 [visited on 2017-05-06]. Available from: `https://lucene.apache.org/core/`.

43. Banana for Solr [online]. 2017 [visited on 2017-05-06]. Available from: `https://github.com/lucidworks/banana`.

44. *Kibana: Your Window into the Elastic Stack* [online]. Elasticsearch, 2017 [visited on 2017-05-06]. Available from: `https://www.elastic.co/products/kibana`.

45. *Grafana: The open platform for beautiful analytics and monitoring* [online]. Grafana Labs, 2017 [visited on 2017-05-06]. Available from: `https://grafana.com/`.

46. *ElastAlert* [online]. Yelp, 2014 [visited on 2017-05-06]. Available from: `https://elastalert.readthedocs.io/en/latest/`.

47. *Ambari* [online]. Apache Software Foundation, 2017 [visited on 2017-05-06]. Available from: https://ambari.apache.org/.

48. *Graylog: Trusted full-featured log management* [online]. Graylog, Inc., 2017 [visited on 2017-05-06]. Available from: https://www.graylog.org/.

49. *MongoDB* [online]. MongoDB, Inc., 2017 [visited on 2017-05-06]. Available from: https://www.mongodb.com/.

50. *Elastic* [online]. Elasticsearch, 2017 [visited on 2017-05-06]. Available from: https://www.elastic.co/.

51. *Loggly* [online]. Loggly, Inc., 2017 [visited on 2017-05-06]. Available from: https://www.loggly.com/.

52. *NXLog* [online]. NXLog Ltd., 2016 [visited on 2017-05-06]. Available from: https://nxlog.co/.

53. *Nagios* [online]. Nagios Enterprises, 2009–2017 [visited on 2017-05-06]. Available from: https://www.nagios.org/.

54. *Stackify* [online]. Stackify, 2017 [visited on 2017-05-06]. Available from: https://stackify.com/.

55. *Sematext* [online]. Sematext Group, Inc., 2017 [visited on 2017-05-06]. Available from: https://sematext.com/.

56. *Lucidworks* [online]. Lucidworks, 2017 [visited on 2017-05-06]. Available from: https://lucidworks.com/.

57. *Django: The web framework for perfectionists with deadlines* [online]. Django Software Foundation, 2005–2017 [visited on 2017-05-06]. Available from: https://www.djangoproject.com/.

58. *ZooKeeper Administrator's Guide: A Guide to Deployment and Administration* [online]. Apache Software Foundation, 2008–2013 [visited on 2017-05-06]. Available from: https://zookeeper.apache.org/doc/trunk/zookeeperAdmin.html#Communication+using+the+Netty+framework.

59. *Kafka papers and presentations* [online]. Apache Software Foundation, 2016 [visited on 2017-05-06]. Available from: https://cwiki.apache.org/confluence/display/KAFKA/Kafka+papers+and+presentations.

60. *Problems detecting when Kafka server is down* [online]. GitHub, Inc., 2017 [visited on 2017-05-06]. Available from: `https://github.com/logstash-plugins/logstash-output-kafka/issues/48`.

# Attachments

- Caribou's source code
- text document, Guide to Caribou
- results of tests
- thesis in the PDF format