A distributed and scalable real-time log analysis

Master's Thesis

Rick Pieter Jan Proost

A distributed and scalable real-time log analysis

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER SCIENCE

by

Rick Pieter Jan Proost
born in Noordwijk aan zee, the Netherlands

Software Engineering Research Group
Department of Software Technology
Faculty EEMCS, Delft University of Technology
Delft, the Netherlands
www.ewi.tudelft.nl

Weave
Goudsesingel 78, 3011 KD
Rotterdam, the Netherlands
www.weave.nl

A distributed and scalable real-time log analysis

Author: Rick Pieter Jan Proost
Student id: 4173619
Email: rpjproost@gmail.com

Abstract

Monitoring software behaviour is being done in various ways. Log messages are being output by almost any kind of running software system. Therefore, learning how software behaves from doing analysis over log data can lead to new insights about the system. However, the number of log messages in a computer system grow fast, and analysing the log data by hand is a time-consuming job.
The objective of this study is to propose and implement a scalable architecture for doing real-time log analysis. Log data is structured so that analysis can take place, and the solution is horizontally scalable in every module so that the approach can scale with an ever-growing software solution. The focus of the study is on scalability, and ease-of-use of the implementation of the proposed approach.
The proposed solution can scale horizontally and the test set up showed that reporting features for anomalies remained instantaneous when processing 1.2 million log lines per minute. The usability of the proposed approach is tested in a case study at Weave, where bugs were found by running the proposed solution in a controlled environment.

Keywords: Scalable Log Data Analysis, Prefix Tree, Distributed Systems, Horizontal Scaling, Real-time Log Data Analysis, Software Monitoring

Thesis Committee:

Chair:	Prof. Dr. A. van Deursen, Faculty EEMCS, TU Delft
University supervisor:	Dr. M. Aniche, Faculty EEMCS, TU Delft
Committee Member:	Dr. A. Katsifodimos, Faculty EEMCS, TU Delft

Acknowledgments

First of all I would like to thank Maurício Aniche for sparking my interest in the topic of log analysis, his enthusiasm and contributions to this thesis. Next to that I would like to thank Peter-Jan Karens and Stephan Plat for the collaboration and the resources to make a case study possible for this study.

Rick Pieter Jan Proost
Delft, the Netherlands
October 13, 2020

Contents

List of Figures

Chapter 1

Introduction

Monitoring software behaviour is essential when dealing with small and large systems. Retrieving insights on software relies on monitoring to understand how the system behaves in production. Companies are emerging in the industry focusing on monitoring log data from software [18] [19], proof there exists a growing demand for tools giving insight into software running in production.

Software monitoring can be done in several ways, e.g. keeping track of resources used up by the software by using metrics of CPU utilisation, RAM usage and network throughput. It can also be done in a more specific way, namely monitoring Key Performance Indicators (KPI) of software when they are exported. Typical KPI's for database systems are the number of transactions and queries active, or average performance measured in the duration of queries. When those KPI's are not enough and more knowledge of processes in the software are required, almost all software solutions use logging.

Logging means that the software outputs messages when running that can be useful as they describe the state of process execution in the system at that moment in time. Log messages can be targeted at different goals, problems or warnings about specific situations, detection of security issues or even more general information on ongoing processes. During system development, log messages can be used for debugging purposes such as troubleshooting for detected errors or critical situations, but also for monitoring the system's execution flows when running in production.

However, computer systems often generate enormous amounts of logs, which in turn hold high volumes of possibly interesting and useful information. When computer systems get larger, possibly scaled to multiple nodes on a cluster or consisting of various parts logging their output messages, manually going through logs to extract relevant data becomes difficult and useful information can be overseen. While the fact that a log message alone can tell what the system is doing at that exact moment, in combination with other log messages, it could also explain more about the processes the system is executing in order.

Therefore finding a way to structure log data can help identify latent factors which will

enable better analysis over the running system. When dealing with large amounts of log data coming from multiple sources; however, structuring the data is not straight-forward. And enabling analysis in real-time will require ever-growing hardware resources that are not always accessible.

There are existing solutions like Dapper [41], Jaeger [12] and Zipkin [27], created by Google, Uber and Twitter, respectively. All of them focus much more on network traffic and heavily rely on dependencies like an API or library that needs to be used. The proposed solution will focus on having the least amount of requirements to still work with any kind of log data produced by systems and will do nothing with network calls directly. Though the existing solutions are not equal to what this thesis will offer, they show there exists a need of analysis tools based on data produced in real-time that can be analysed, which the proposed solution will also provide.

A solution for dealing with large amounts of log data, structuring it and enabling a real-time analysis on the data is therefore proposed in this study. The proposed solution works in a distributed system, while the goal of this study is to prove there is a scalable way in which large amounts of log data can be handled and analysed. A cluster is set up as part of this study, together with an overview of all the parts in the architecture needed to create a scalable solution for dealing with large amounts of log data. Next to self-developed tools, out-of-the-box components are used that can be installed on the cluster, like Prometheus [21] and Grafana [10]. The log data is structured by using a prefix tree, which is furtherly discussed in the background and approach chapter. The prefix tree enables the log data to be structured into a graph which can act in a DFA-like manner in which analysis can be conducted. In this study, simple computations will be done on the graph in real-time like measuring the time between log messages in a trace and comparing them against a previous modelled graph for performance degradation. Also, new paths will be discovered in real-time when compared to the previous modelled graph.

The goal of this study is to validate the performance of the proposed approach on accuracy and scalability, where the proposed solution organises and structures large amounts of log data in real-time so it can be analysed in real-time or at a later point in time. Representing software as a state machine opens up the ability to use already existing analysis algorithms, but existing tools do not scale and therefore, do not work in real-time. In this study, we pose the following questions which help assess the performance of the proposed solution. Beginning with what different aspects of log data impact performance of modelling the graph and how does the approach perform with varying quantities of log data. We look at what kind of computational resource metrics give insight on how performant the approach is and if these metrics indicate when the approach should be scaled horizontally. The last question is if performance degrades over a more extended period.

The results of this study are promising as they prove the fact that log data can be structured and analysed in a real-time manner in terms of CPU-utilization. The experiments conducted in a small kubernetes [14] cluster are already handling 1.2 million log messages

a minute. Heavier analysis computational-wise can be added in the future while log messages can be distributed amongst multiple deployments of the solution proposed making it horizontally scalable. The reporting features of the proposed solution like performance degradation and new paths found are still instantaneously reported when being read from the message stream.

This work makes the following contributions:

- A scalable approach for real-time log analysis by presenting an architectural solution for dealing with large amounts of log data
- An empirical study showing how the solution deals with varying inputs of log data
- A case study at Weave showing how well the solution integrates with existing software and how it helps the developer

An implemented prove of concept can be found on github [23].

The thesis is structured as follows: first off background information on the software log data and passive learning is discussed by using existing related work. The data structure used, the prefix tree, is explained and why it is used in the proposed solution. Next, the approach of cluster setup is discussed in detail as well as the implementation used for the experiments. The methodology chapter indicates the research questions and which experiments are conducted to provide results to answer the research questions. The results of the experiments itself can be found in the appendix and are explained in detail in the Results chapter. Finally, conclusions about the solutions are provided in the final chapter, together with future work recommendations on how to improve the proposed solution.

Chapter 2

Background

In this chapter, the background is given for aspects this thesis is expanding on. These aspects consist of log data, data structures, passive learning, distributed systems and detection of anomalies. We assess the state of research in these different categories here and discuss them in this chapter. The research done here is used and resulted in the proposed solution.

2.1 Logging

Nowadays, computer systems provide large amounts of log data; therefore, manual analysis becomes unfeasible. Automating log analysis does come with a multitude of challenges; one of them is while application logs are largely readable for humans, it is not always interpretable for automated systems.

To overcome this challenge methodologies are proposed like log parsing, log clustering and graph construction. This way, log data can be interpreted by automated systems and handled generically to finally extract latent factors that could lead to useful insights. To implement any methodology first needs to be determined what data should be logged at all.

Computer systems log messages in various ways. Often, there is not a single format in which logs are structured. Mostly a log message model consists of a timestamp, the type of the log (e.g. INFO, ERROR) and a message containing parameters instantiated by the context of the running code. To analyse the log data, a generic structure can be thought of to model the messages, so that the constant parts can be divided from dynamically changing parameters. Log parsing is used to transform log messages into these structured models so log data can be effectively processed in the same manner. When the log structure is changed, the log parsing model should also change and adapt, methods proposed by Xu et al [49] infer models from parsing the source code to overcome this challenge. He et al [32] and Lin et al [37] both researched what kind of data should be in log messages to make them effective in terms of resolving issues by reviewing log data. They both propose that an event identifier should be incorporated to make it possible to cluster log messages together that are initiated by the same event; these could also be added without changing the source

code. This identifier can be used in the proposed solution as an identifier to create traces of log messages that originate from the same event or process.

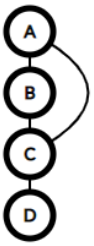
For the proposed solution, traces of log messages could reveal more latent factors, however, it is not trivial on where to log in the source code, and more log output could also mean less performant. Research done in the field of what to log exactly in terms of variables and where in the process should be logged is mainly determined by the rate in which failures are logged by a system so that they can be handled. Cinque et al [30] found that by injecting faults into the software and trying different patterns for putting log statements in the source code, the developer greatly impacts the potential effectiveness of log data. By using simplistic patterns, e.g. log statements after certain branching in the source code, up to 60% [30] [39] of failures could be left undetected, whereas Cinque et al. also describe that when log statements are thought out in the design this resulted in 92% of failures covered.

2.2 Data structures

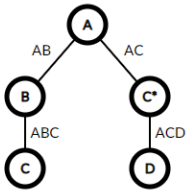
Clustering of log messages can be used to merge correlating log messages together as an extra abstraction layer. Log messages that correlate with each other could possibly be pointing to the same event happening in the code. Therefore if these messages are combined, it could provide new insights when analysing them together. Clustering techniques are applied, like using word frequency count in log data [38] [46] or transforming log data in heuristic feature vectors [40]. When it comes to finding sequences attached to an event with temporal features, graph construction is used.

Graph construction for log messages is done to create a relation between multiple log messages; this provides an extra dimension for log analysis to extract features from. If a graph is constructed, it opens up a broad spectrum of available algorithms to use for analysing the logs. Sun et al [44] for example proposes a technique where a CFG is constructed after using the Log Parsing Approach with Fixed Depth Tree [33]. A CFG is a Time-weighted control flaw graph and is constructed to see what log templates follow each other and in what time interval this happens. Yu et al. [15] give a different approach to graph construction; the log messages are not fully related here by a temporal feature but by co-occurring events. The first action that is required is representing the events and their connections in a model; then a method is needed to extract events similar enough into a pattern set. The graph in the proceeding is constructed in such a way that events that are happening shortly after one another are represented with a shorter graph distance. Both solutions just described focus on finding similarity between log messages; however, for the proposed solution, this can already be deduced when an event or trace identifier is added to the log message.

A directed acyclic graph (DAG) would therefore be what we need to achieve a Deterministic Finite Automaton (DFA) like data structure for the log data. To be able to convert incoming traces into a graph, we can use the unique identifier from a log message to create



(a) Non-existing path creation



(b) Prefix tree

nodes and create edges amongst them. The first challenge here comes from the fact that our solution should incorporate a directed acyclic graph so it would look like a decision tree. Meaning the graph should depict and distinguish paths that can be created from traces so that they are formed in a way that every path that can be extracted from the output of a system to a resulting graph path. The most simple form would be creating nodes for every log identifier and then create edges based on some shared generic information that can later be analysed.

However, if we take a look at the following two traces simplified as ordered log events, creating nodes and edges will result into paths that never existed in the output of the original system as can be seen in 2.1a. The path A B C D is never seen in the original output but is now created in the DAG.

A B C
A C D

To overcome the problem of creating non-existing paths, the data structure prefix tree is chosen [25]. In this data structure, the edges are based on keys, usually, strings, which take into account every part or character in the string and goes a level deeper in the graph. Nodes only point to metadata; the edges define where the data is stored. An example with the same trace events is shown in 2.1b where the edges are defined based on the identifiers of the log lines. Working with the prefix tree data structure does result in correctly created paths; however, it can create large graphs. Paths that end in the same node or have side tracks which do return into an already existing path are separate, unlike a directed acyclic graph. Merging these nodes without introducing non-existing paths is a heavy job in graph theory and in this study is chosen not to do this merging while the result of this Trie holds enough information and most importantly is valid to do analysis on.

2.3 Passive learning

Passive learning is about inferring graph models on the behaviour of the system [47]. In this specific case, this will be done log data, by structuring log data into traces and model them into a graph. The resulting graphs can then be analysed for multiple types of analysis like automation of anomaly detection or measuring performance. Passive learning differs from

active learning through the fact that active learning means that a systems model is actively queried to get the responses to different inputs, and that inferred machines are verified against that system. Passive learning relies on acquired observations and can be compared against newly acquired observations. The work of Wieman et al. [48] introduces three different tools that are in the same line of work of this study. First, the tool Synoptic [28] is discussed which infers a state diagram from log files for debugging purposes. Next from the same authors InvariMint [29] is discussed which models log lines on the edges, more like a prefix tree. And last DFASAT [34], a work of Heule et al. is described, which also uses a prefix tree-like structure to model log traces. All of the above have in common that the models are inferred from historical log data that can be processed later on and try to merge traces in the graph while still retaining unique paths in the graph. These works show that structuring log data into a graph can be used effectively for analysis, and finding a way to model a graph in a scalable way could benefit the industry even more.

2.4 Distributed systems

There are already existing and well-known technologies commonly used in the field of logging and logging analysis. To be able to deal with large amounts of data, multiple kinds of existing technologies can be used. Zhengmin et al. [52] provide, for example, a detailed description of a scalable, high throughput distributed log stream processing system. It immediately sums up many of the most used technologies in the different literature already mentioned. For example Apache Kafka [35], Spark [51], Hadoop [11] and Storm [24] are used by [38] [50] [40] and [52]. The common denominator here is especially the fact that for large amounts of log data, a horizontally scalable and performing solution must be taken into account. Another common factor is that software from Apache is widely used in the research environment.
Dealing with large amounts of log data means using resources effectively when the goal is to process the logs in real-time when the computer system is up and running, it may not disrupt the workings of the system. In the literature is found a clear distinction being made in terms of real-time and post-processing. Mainly because of the fact that some algorithms are heavier to run than others. As already mentioned, algorithms can be resource hungry, and without the necessary precautions, it may be damaging your healthy computer system.

An example of a scalable, high throughput distributed log stream processing system was already mentioned, but it did not yet detect or predict anomalies. Yagoub et al. [50] do provide a high performing anomaly detection system based on Apache Hadoop. It uses machine learning to create time series (Microsoft Time Series algorithm [16]) to not only detect anomalies but also to make a forecast and predict them. The content of the logs is barely used in this log processing method, it uses the time series with MapReduce over logs to monitor KPIs of the system, like disk and CPU usage and creates an anomaly index based on the max and average usage. However, in the end, if an anomaly is found by a sudden change in the anomaly index, it is able to trace it back to the log messages. Not only performance is a challenge in real-time log analysis systems, as already mentioned in both [44] and [42]. Sukmana et al [42] however uses a temporal feature to detect whether

logs correlate with each other. In a real-time system, this means the sliding border problem becomes even more difficult to handle while not all information is known and a decision must be made how big of a window is used to process the logs from.

2.5 Anomalies

This section focuses on anomaly detection and is divided into four subsections. This is because in the field of log analysis literature focuses mainly on one or maybe two of these aspects at once. The different methods used for anomaly detection derive different kinds of anomalies; this section will describe what those different anomalies are and how they are detected. The first subsection describes temporal anomaly detection, meaning the primary part of anomaly detection comes from the timing of log messages. The second one is anomaly detection on performance measurements. The third focuses on the actual content of the application logs. And the last one is visualisation which differs from the previous three sections while it is not automated but requires manual labour to see anomaly trends.

Temporal anomaly detection is based on events occurring within time intervals; this can be two or more related events happening within a certain interval. This interval can then be a threshold or automatically calculated based on previous events. Proceedings that based its anomaly detection this way are the already mentioned [49], [44], [45]. Not only related events are time-boxed however, also frequencies of one kind of log model can be counted in a certain interval like [40], [31] and [36] do. Kubacki et al. [36] creates three types of time series, Pulse (p), Series of pulses (SP) and a period group of pulses (PGP). Where (SP) is a sequence of (P) in a certain interval and the same for (SP) in (PGP). In combination with correlation analysis, the anomaly in the time series can be traced back to the root cause in the application logs, this was not fully evaluated yet, but the proceeding did show that the application logs were synergetic with the time series. Creating time series are popular in the world of log analysis while it is an easy concept to understand and makes sense when working with timestamped data. The challenge here is also the already mentioned, sliding window, and where to cut the borders when comparing your current logs against the subset selected. Notice that any type of log parsing methodology can be chosen for this type of anomaly detection, as long as the temporal factor is taken into account this will be a valid option for log analysis trying to detect anomalies or failures. The proposed solution will also heavily rely on the ordered log traces and by passively learning the graph model, the time frame becomes an important factor on how valid the observations will be when comparing real-time log data against the built model.

Anomaly detection can also be done by checking the performance logs from a computer system. Sudden changes in resource usage can indicate that something failed and by time boxing this and correlating it with the application logs several works like in the works of [50] and [36]. This way of detecting anomalies could be applied in almost any computer system while this kind of system logs are very common. Although it may be too late when the resource usages change, it is a method that can extend existing methods of anomaly

detection and prevent future failures by analysing the result of these methods. In a way, the proposed solution will also reveal performance issues in terms of comparing the time between log messages in a trace, which could also indicate high resource usage.

Anomaly detection based on the content of application logs is the most resource-hungry. Modelling log templates can be quite difficult in terms of changing structure. Using a sliding window is taking less information into account, but it is a tradeoff that could be worthwhile if carefully chosen. It is also hard to predict when the content of a message is malicious. In-depth knowledge must be known about the context of the parameters in a log model. Alvarez et al. [38] tried to score log models using LogCluster, but there were too many outliers to detect outliers accurately. The proposed solution will therefore retain selected parts of the log messages, but purely for later manual analysis.

Visualisation is a non-automated process, but anomalies can be detected through manual labour. Visualisation is still applicable in the way that possible anomalies can be represented visually. Suman et al. [43] focused on visualising the logs in such a way that anomaly detection could be done manually. This was done by first parsing logs on topic and then use Term frequency and finally making an intertopic distance map plus a bar graph. This makes manual labour easier when dealing with large amounts of log data. In the proposed solution a visual representation of the graph or parts of the graph will also be taken into account, where a developer is helped to deal with the large structure that is built and still be able to make decisions about it.

2.6 Existing solutions

Golang [8] is chosen for the development of the proposed solution. Golang is a programming language initially developed by Google and is also opensource. One of the reasons golang is chosen as the language to do this experiment with is that it coincides with a cost-effective way of working with a cluster and therefore being a good fit for scalable parts. Another reason is that all of the current software built by Weave [26], the industry partner for this thesis, is currently developed in Golang. Golang is created to be compiled to self-contained binaries, which do not have library dependencies out of the box, which makes docker [4] images to deploy small, namely 13MB for this current solution. Docker [4] can be used for creating images called containers, that incorporate all required software components to run a program. Essentially it can be seen as a tool to preconfigure and tailor an operating system to provide all the needs for a certain deployment and then be able to virtually run it on a machine. Next to the fact that Golang can run in small containers, Golang also offers a very resource-effective method of multithreading called goroutines, built into the runtime instead of being handled by the operating system. This enables running a goroutine for every incoming trace, which makes processing very fast and non-blocking for large amounts of incoming traffic.

Similar solutions to the proposed solution are mainly found in the network monitoring

field. Tools like Dapper [41] developed by Google, Jaeger [12] from Uber and Zipkin from Twitter [27] show that big companies in the industry have the need for analysis tools based on log data. All of the above are mainly in use for tracing network calls, and showing metadata like overall performance and bytes transferred, status codes returned and so on. Some of the tools like Jaeger also enables the developer to create traces inside the source code by use of a specific library, which needs to be taken into account when developing the system. By using the standard OpenTracing [20] spans can be created inside of your program, which can then be traced by Jaeger in the same way it traces network calls. For network traffic however it is not needed to adjust the source code, tools like Envoy [5] by lyft can reroute HTTP and RPC traffic and adjust them, by for example adding the needed headers for the tracing tools.

2.7 Conclusion

All of the research being done into log analysis shows there are valid methods of analysing log data that can reveal important latent factors when developing or monitoring a system. The fact that almost any computer system outputs log data makes this a research field which can potentially benefit many in the industry. However, it can also be seen that there are no solutions that are solely reliant on log data for real-time analysis, only for post-processing or with the use of external libraries. This is mainly because of the challenges which arise by analysing large amounts of log data produced and unstructured data is also more complex to process. Existing tools do exist, but they do not scale or do not work in real-time, where creating a scalable real-time log monitoring solution is the goal of this study. Therefore, this study is an extension to the field of log analysis in such a way that it can provide a proof of concept that real-time analysis can be done and potentially benefit the industry more than existing techniques.

Chapter 3

Approach

In this chapter, we describe the proposed solution in an abstract and generic way. Different parts of the proposed solution are distinguished and defined as modules which are either required or optional. For every part or module in the proposed solution, a definition will be given and also the way it interacts with the other different parts of the system. Next, the implementation of this described approach will be given in detail, like the technologies and languages used and why they were chosen as a valid option to experiment with. The proposed solution is based on a few preset requirements. This gives insight into what the proposed solution is able to do and also describe the research that this study focuses on:

1. Collect log output of running systems
2. Model log messages in a structured manner
3. Form traces of collected log data
4. Process log traces in a distributed manner, so that horizontal scaling is an option
5. Create a DFA-like data structure from log traces which enables analysis
6. Able to store the state of the system as a modelled graph
7. Able to compare the current state of a system against a previously stored state in terms of new paths
8. Able to detect performance degradation in real-time in terms of timing between log messages in a trace

The proposed solution focuses on a distributed system which enables processing of a large input of log data. Therefore the proposed solution depends on many other factors, such as the environment it will run on and how the log lines and/or traces are being fed into this solution. This chapter will, therefore depict a systematical diagram 3.1 of how all parts of an environment should interact and explain how all these parts work in detail.

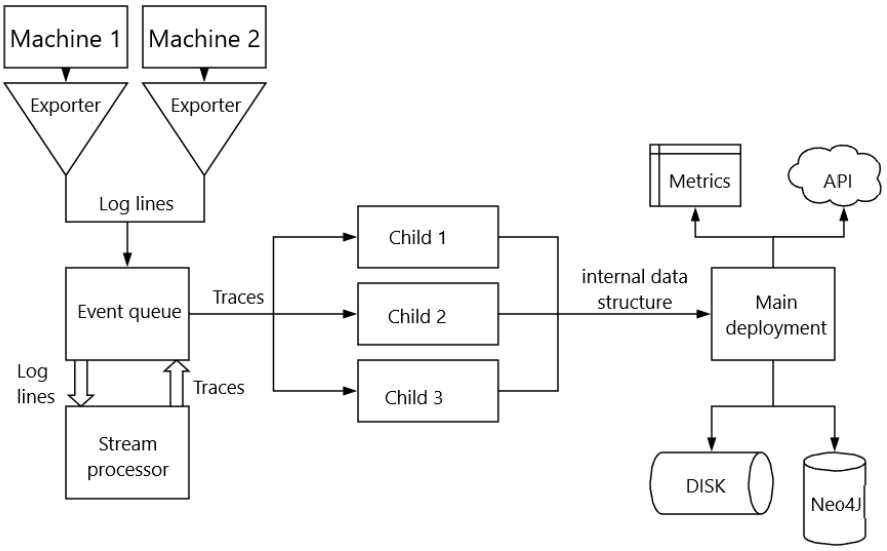


Figure 3.1: Overview of all parts needed for distributed real-time log analysis

3.1 Architecture

Nowadays there are many systems in production that produce high amounts of log output like systems of companies, e.g. Google, Facebook, Adyen, Amazon. When structuring traces in real-time and performing real-time analysis on those traces, it requires scaling up resources. If there is only one machine able to run a program for processing this stream of traces, it means that the only option is to scale vertically, meaning more power to this one machine in terms of CPU or RAM. Scaling up vertically can mean prices for the required resources can go very high, and eventually there will be a limit to what one machine can take in terms of hardware. Another way of scaling is horizontal scaling, which requires adding more machines that can process the same input, which will be cheaper in many cases and the limit will depend on how many machines can be added. Therefore this architecture focuses on every part to be horizontally scalable, which will make it easier to deal with large amounts of log data.

There are many variations in which systems produce log output. It can be one program on the machine producing the output to collect; it can be multiple programs on one machine or even multiple programs on multiple machines. While there are many variations, all produced log output data should be collected in some generic way and from there it can be further processed. In the proposed solution, this means having a central hub in the form of a Pub/Sub queueing system in the environment, which still can be scaled horizontally. A publish/subscriber system is able to receive all log lines and traces, after collecting it is able to redistribute them using an algorithm like round-robin to subscribers which can further process the data stream. Popular event queueing systems are Apache Kafka [35], RabbitMQ [22], Apache ActiveMQ [1] and NATS [17].

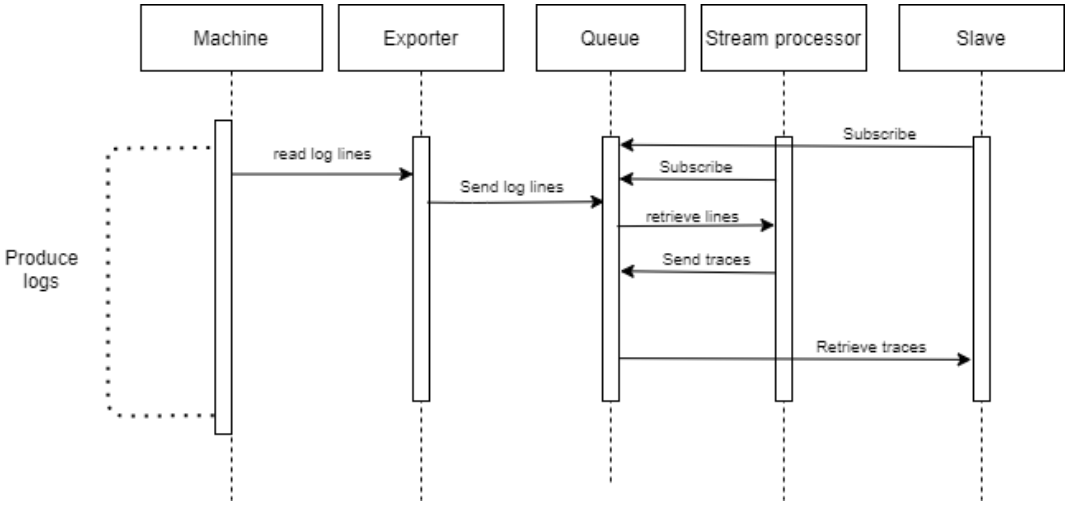


Figure 3.2: Sequence diagram data flow through system

There are multiple ways in which an existing program can interact with such a Pub/Sub system. One way can be done without alterations to the existing codebase of a system by implementing an exporter in the required environment. Figure 3.1 shows this by having Machine one and two producing log data with an exporter sending it to the event queue on the top left. Exporters such as LogStash [15] and Fluentd [7] gather logs in a system based on set configurations and export those too, for example, an event queuing system also based on configurations. Another way would be sending produced log data directly to the queueing system. In figure 3.2, a sequence diagram can be seen that shows how log data flows through the system.

The process continues after the collection by finding traces from the produced log lines, which means transforming single log lines to merged ones which represent traces. In figure 3.1 this can be seen in the bottom left, where log lines are sent to the stream processor, and the resulting traces are sent back to the pub/sub system. The proposed solution requires an environment where these log lines can be time-windowed with the help of a data streaming processor like Apache Flink [6] or Benthos [2]. A data streaming processor will need to work with a sliding-window like mechanism to be able to process ever-growing incoming data and bundle log lines together as one trace. With the help of data streaming processors, we are able to divide log lines by time and by preconfiguring a time limit between log lines a complete trace can be found. In the example diagram 3.3 this is made clear by a limit of twenty seconds. It displays a logline with trace identifier 'A' and 'B' in the first time window. By not detecting trace identifier 'A' in the next time window, by definition, this trace is now finished and can be collected. In the end, this means that a trace will at maximum be processed the preconfigured limit in seconds later than the last log line with its trace identifier.

The last part of the data processing is the throughput of the traced log data to the child

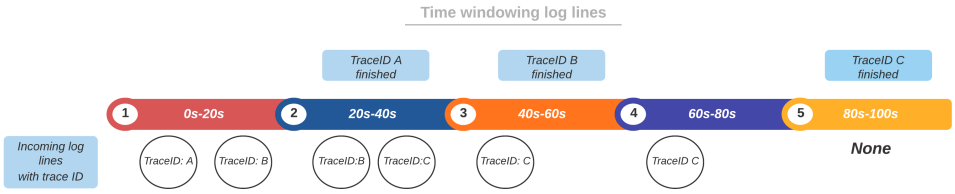


Figure 3.3: Logs divided into time windows of 20 seconds each

instances, as can be seen in the middle in figure 3.1, there are two parts in this final process. One part, defined as the main instance from this point, is the overall state manager, which is able to store the state, merge new states, export the entire state and able to handle specific requests. The other part, defined as child instances, are scalable units which are able to receive traces, build a graph dynamically and be able to do real-time analysis on them. Child instances subscribe themselves to the queue and can be deployed separately, all of the childs retrieve traces based on a round-robin algorithm, so computational-wise the load will be spread. Any number of childs can be deployed, which makes the complete solution horizontally scalable and every child can do its own computations on the incoming trace, built graph model and comparison to the previous base graph. The main instance then does the merging of the partial trees of the children and offers API access, metric exposure and export possibilities.

The scalable child deployments are able to do simple computations on traces, these can be extended, but now are limited to:

- Computing time difference between log lines in a trace
- Comparing average time difference between loglines against the modelled base graph
- Reporting degradation in time difference between log lines
- Reporting new paths found compared to modelled base graph
- Reporting spikes in amount of specific path processed
- Reporting specific identifiers/payload in paths (e.g. following user identifier in log data)

The main instance is therefore relieved of the above-mentioned computation responsibilities and is just responsible for merging complete graph structures that it retrieves from the child instances. It can then store or export its current state so it can become a new baseline for the child instances to compare their computations with. However, there are a few minor computations that the main instance is still responsible for, such as when the child instaces find a new, it reports to the main instance, so it will only flag a new path once,

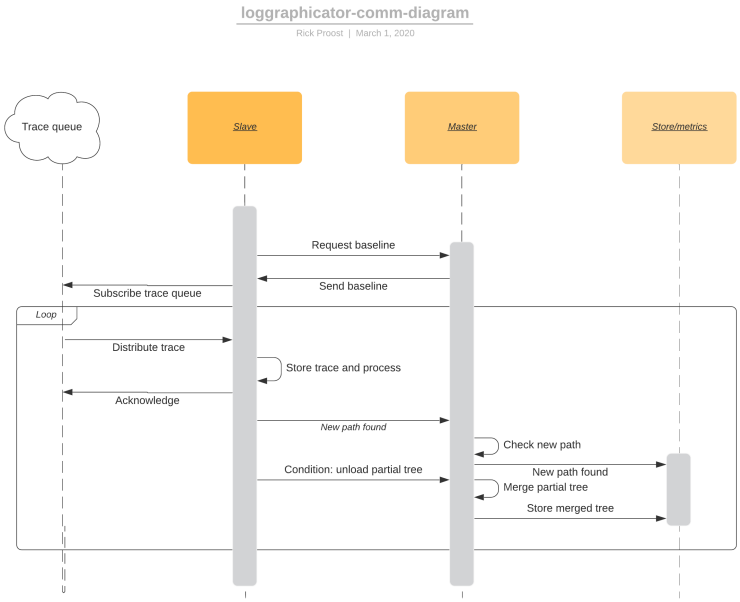


Figure 3.4: Sequence diagram queue/child/main communication

instead of every child flagging it. These are very small operations; however, which only account to exporting metrics which all the child parts are doing as well.

The child parts are stateless, and in fact, they reach out to the main instance when they start up to retrieve a baseline. When retrieved, they run on their own by subscribing to the trace queue. This means that child parts can be turned on and off at any moment and can scale independently from the main deployment. The main instance can also reboot at any time; it will only read in in the latest or preconfigured baseline, child parts will retry sending their information if they won't succeed. Figure 3.4 depicts how the different parts of the system interact, starting with requesting the baseline, then subscribing to the trace queue and afterwards processing every sent trace.

In summary, all the different parts of an ecosystem to deal with large amounts of log data are now described. Starting with the programs or machines that output raw log data, which are then collected by an exporter that sents the log data to a queueing system. From there it is processed by scalable stream processors which merge the separate log lines to time-windowed batches where traces can be extracted. Then these traces can be redistributed to the scalable child deployments by the publish/subscribe queueing system. The child deployments will do the computations and building of the graph structure, and the

main deployment will receive the partial trees when a preconfigured condition is met and will merge, store and potentially export them.

3.2 Log structure and parsing

Log data can vary in structure among every computer system and program out there, but there are a few components in log data that are shared (standards) among many solutions already in production. Therefore the proposed solution requires three different parts in a log message to be able to infer a graph from them. The following properties are determined as required for the proposed solution to be able to create a graph:

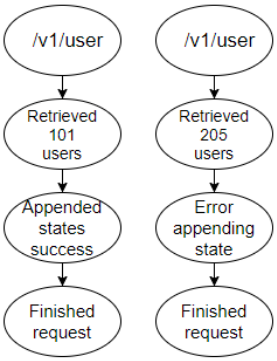
- Timestamp
- Static log ID
- Trace ID

The timestamp is needed to make sure logs can be sorted inside a trace based on execution timing while it is not known whether log lines will be delivered in the correct time sequence. A static log ID is required to identify a log output in the systems codebase where the message could differ every time based on dynamic value input. Lastly, a trace ID will be needed to identify which log lines were put out by the same execution process or flow, which eventually can be ordered on the timestamp and the result will be one execution path in the graph. All other kinds of payloads like the message or maybe JSON [13] defined fields can still be taken into account but are not needed to create an execution path graph.

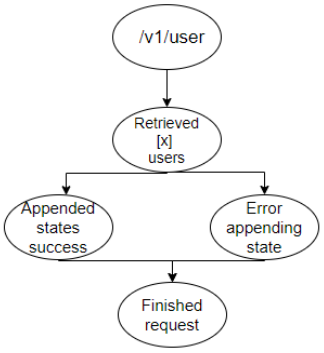
The log lines below depict an example of the required properties together with a dynamically created message on runtime. The lines are formed of timestamp, log identifier, trace identifier and message part, respectively. It can be seen that there are two different traces in this example while there are two different trace identifiers, furthermore, there are five different log identifiers. The importance of the log identifier is shown by the two messages 'Retrieved 601 users' and 'Retrieved 602 users' which differ from each other in message but are actually coming from the same place in the codebase. Hence the same log identifier 'bZRjxAwnwe' which enables the graph to form a static path from log output in the codebase instead of basing it upon dynamically changing messages.

```
[2019-10-28 10:05:01:736] {MRAjWwhTHc} {cdddbcdf} Started Request [GET]: /v1/user
[2019-10-28 10:05:02:126] {bZRjxAwnwe} {cdddbcdf} Retrieved 601 users
[2019-10-28 10:05:02:220] {MRAjWwhTHc} {6ce65118} Started Request [GET]: /v1/user
[2019-10-28 10:05:02:223] {TndUJHiQec} {cdddbcdf} Appended states to users succesfully
[2019-10-28 10:05:02:289] {uXvgxEqdoX} {cdddbcdf} Finished Request [GET]: /v1/user
[2019-10-28 10:05:03:120] {bZRjxAwnwe} {6ce65118} Retrieved 602 users
[2019-10-28 10:05:03:120] {tCjySfUrie} {6ce65118} Error appending state to user X
[2019-10-28 10:05:03:160] {uXvgxEqdoX} {6ce65118} Finished Request [GET]: /v1/user
```

Both the traces in the example begin with the same log identifiers, but they follow different paths in the codebase. Below in figure 3.5a and 3.5b a resulting graph is depicted from these example log lines as traces and merged in one graph. It also shows the importance of finding a correct data structure to hold this information and being able to quickly store and access these parsed log lines and resulting traces. While the traces can be depicted as single



(a) An example of separate traces as one path and together in a graph



(b) An example of traces in a graph

paths in a tree, but ideally, the nodes of the graphs that are the same are merged when they still depict the correct paths.

3.3 Building prefix trees

For structuring traces in a scalable way, implementation is needed for the prefix tree that can be shared amongst multiple deployments of the solution. A prefix tree works via the creation of paths in a graph on information stored in the edges. The nodes of a prefix tree can be seen as a placeholder for data, but do not have true meaning in terms of the representation of the graph. That means that an alphabet can be chosen as symbols for each edge in a prefix tree, where for each symbol in the chosen alphabet, an edge is created with that symbol from the previous symbol edge. Meaning that when structuring the prefix tree, a word in the chosen alphabet will be extracted into symbols, and for each symbol read from the beginning of the word, an edge will be created and the next symbol will be one level deeper. Figure 2.1b in the background chapter already shows how a prefix tree is built when the words 'ABC' and 'ACD' are structured into a prefix tree.

Because this thesis is about processing large amounts of log messages and model them into a prefix tree, the implementation is based on a thread-safe prefix tree structure. The prefix tree implemented data structure works with mutual exclusion locks which make inserting new paths and accessing existing paths faster by the fact that is concurrently accessible by multiple threads. Below is a pseudocode implementation 1 of the algorithm which incorporates this data structure and works with mutually exclusive access:

3.4 Combining prefix trees

All the child deployments will build their own prefix tree based on the traces they receive. Building the prefix tree, computing and adding metadata can get resource-hungry depending on what kind of computations are done. However, the main deployment, which is respon-

Algorithm 1 Process traces algorithm

```
1:  procedure RETRIEVING TRACE
2:      trace ← queue
3:      goroutine processTrace.
4:  processTrace:
5:      Sort trace by timestamp
6:      for lineIndex in trace do
7:          trace(lineIndex+1).Performance = delta(trace(lineIndex),trace(lineIndex+1))
8:          start putLine trace(lineIndex).
9:      end for
10: putLine:
11:     node ← trie
12:     logIdentifier ← line.static
13:     for char in logIdentifier do
14:         node.MutexLock
15:         child ← node.Children(char)
16:         if child eq empty then
17:             child = Node()
18:         end if
19:         node.MutexUnLock
20:         node = child
21:     end for
22:     node.MutexLock              ▷ Extra computation can be done on node values here
23:     node.Value = NewValue
24:     node.MutexUnLock
25: end procedure
```

sible for creating the eventual prefix tree, based on the partial ones it receives over time, is not scalable. Therefore the computational complexity of combining prefix trees must not be high, in order to make the proposed solution a valid scalable way to process large amounts of log data.

The main deployment will periodically receive partial prefix trees from its children. Therefore a queue will be build to merge one at a time with the latest main tree known. To combine the main tree with a received partial tree, a new prefix tree is created. Beginning in the root of the main tree, all edges from the root will be checked if they are known in the received partial tree, if not, the entire path from the root can immediately be inserted into the new prefix tree. If the edge is found in the received partial tree, the computed variables will be aggregated, and the edge will be inserted in the new prefix tree. From the mutual edges the next edges, children, will go through the same process, if not found in the received tree, immediately add the remaining path from that edge, otherwise aggregate.
The only remaining step is then adding the complete paths that are found in the received partial tree, but not in the previous main tree. Adding these can be done by doing a second

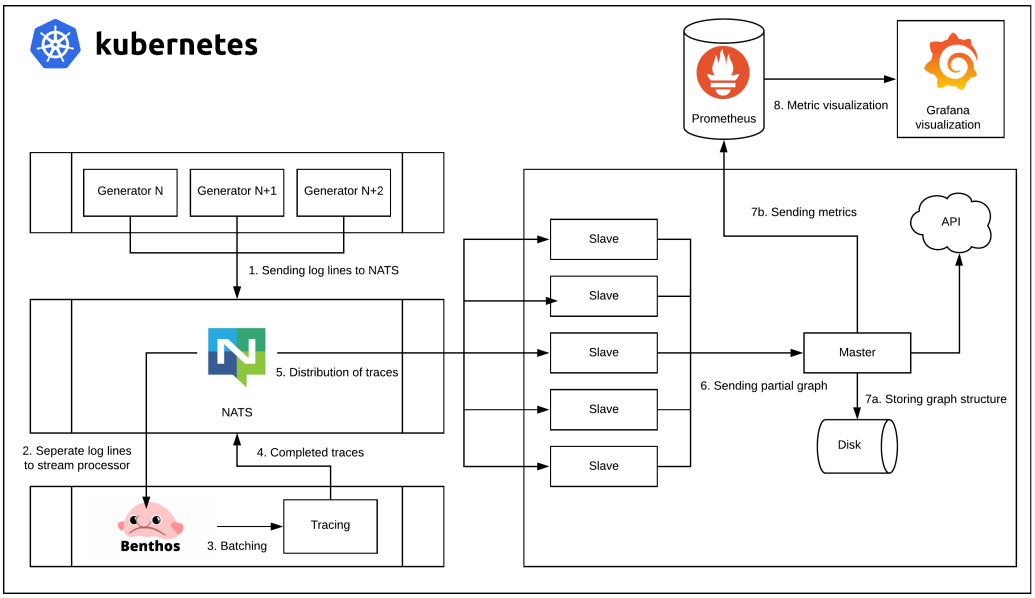


Figure 3.6: Architecture of the complete environment with interactions of the actual implementation

round over the received partial tree, without adding the mutual edges.

This means that in the worst-case scenario, you go over all the edges of the main tree (Em) and traverse all the edges of the partial received tree (Ep), coming down to a complexity of O(Em+Ep) to combine two prefix trees.

3.5 Implementation

The current implementation is created with focus on reproducibility. Everything required to run the experiments detailed in the next chapter is available on Github [23]. This section will describe what technologies are chosen to fulfil the described solution at the beginning of this chapter. Every decision that is made in this process will be discussed in detail and explained why this method was chosen. The complete overview of the architecture can be seen in 3.6 where all subsequent interactions are described by a number following what action is taking place between the different components of the system.

3.5.1 kubernetes

The first part of the ecosystem starts with kubernetes. Originally created by Google [9] and now maintained by the Native Computing Foundation [3]. Kubernetes is a platform for automated application deployment, scaling and management. Kubernetes works via custom configuration resources which indicate how an application should be deployed, for

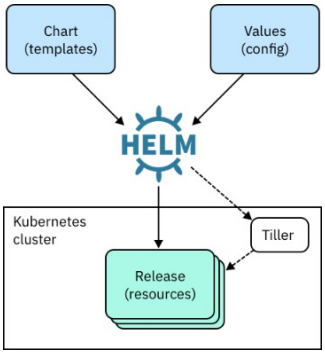


Figure 3.7: Helm chart interaction with kubernetes cluster

example, which environment variables the application should have and what kind of hard disk an application should be using. It is a widely used platform for exactly the kind of challenges this study faces, namely deploying a lot of different components in one shared cluster and being able to scale them up with configurations. These configurations can be premade by using Helm charts, which are a sort of templates for deployments where you can edit parameters in the template quite easily for changing your configurations easily. These helm charts are also available for easy reproduction of the experiments and for later use of the scalable logging analysis by deploying the proposed solution. In figure 3.7, a diagram is shown how helm charts are created and how they interact with the kubernetes cluster.

3.5.2 Event queueing system

As already described there are various Event queueing systems available, while this implementation works with kubernetes the choice went to NATS [17] and NATS Streaming which are very easy to deploy on kubernetes and have a self-developed and documented library for Golang. Furthermore, the differences between various event queueing systems are not important for this study while they all should be able to offer and achieve the same results with minor differences in performance, depending on what is needed. For this solution a combination between NATS and NATS Streaming is used, while NATS is used for At Most Once Delivery and NATS Streaming is used for At Least Once Delivery. NATS handles the load of the experiment and could be interchanged with NATS Streaming if a one hundred percent guarantee is needed when delivering log lines, for the experiment, there were no losses in log lines when using NATS. Therefore NATS is chosen for handling the deliverance of the separate log lines while it is more performant. NATS Streaming is an extension on NATS; NATS Streaming offers durable subscriptions and more options in the redistribution of queue items. This is needed for the Round-Robin distribution of traces to every child instance that subscribes to the trace queue.

3.5.3 Streaming processor

For the streaming processor, Benthos [2] was chosen while it is a light-weight single purpose stream processor which can be extended by Golang code which makes it easier to integrate

with the other parts in terms of programming. Benthos handles simple operations over data streams from all kind of sources; simple operations include batching and merging multiple objects into one on predetermined parts of a message like a trace identifier. This makes Benthos a great candidate for batching and already pre-grouping log lines which belong to each other in a trace. Additionally, a plugin is written, which takes two batches following each other and outputs traces to one queue and keeps the remainder in memory until a new batch comes in like described in the previous section.

3.5.4 Metrics

To keep track of all different parts of the proposed ecosystem, the system should be monitored in a way that gives the user an overview of what is happening. This can give a grip on which part is using what resources and bottlenecks can be determined. To monitor the system, multiple metrics can shine a light on how well a system is interacting. Hardware metrics are most common to monitor, but also the throughput, latency and other kinds of custom software metrics can give better insight. One universally used solution for keeping track of metrics in a kubernetes cluster is by using Prometheus [21]. There is a wide range of exporters available for Prometheus, which measure all kinds of software and hardware metrics and sends them to Prometheus. Prometheus offers a query language to get an insight into collected metrics. Grafana [10] comes in to visualize these queryable metrics from Prometheus, where dashboards can be created with multiple graphs to get an overview of what is happening in a cluster in real-time.

In this solution, the main instance as well as the child instances export metrics to Prometheus, like the amount of traces processed and new paths that are encountered. Furthermore, exporters are installed for NATS so that throughput can be measured in real-time.

In the given implementation Grafana is included, and the dashboard keeps track of the following metrics in real-time:

- New paths encountered against the baseline
- Deterioration of performance in specific paths
- Amount of traces processed
- Amount of log lines processed
- Test identifier counter
- NATS RTT
- NATS cluster network throughput
- NATS messages throughput
- NATS connection amount

A small example of how this dashboard looks can be found in figure 3.8

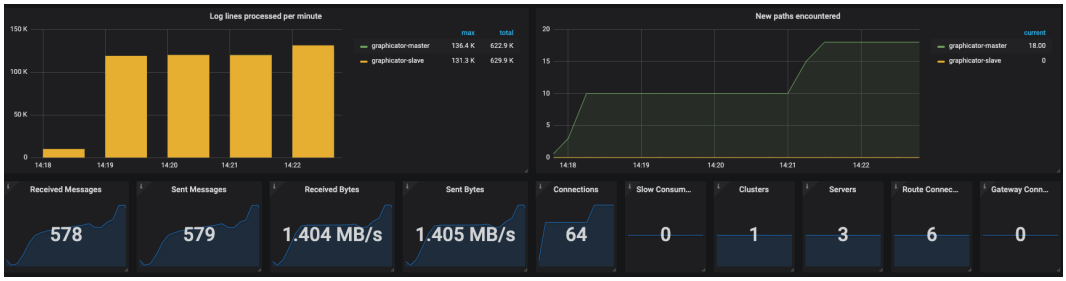


Figure 3.8: Grafana metric visualization

Chapter 4

Research methodology

This chapter will describe the goals, research questions and methodology of how the results will be acquired and benefit the analysis. Every research question is described on how to experiment and evaluate results so the research question can ultimately be answered.

4.1 Goal

The goal of this study is to validate the performance of the proposed approach on accuracy and scalability, where the proposed solution organises and structures large amounts of log data in real-time so it can be analysed in real-time or at a later point in time. To assess there exists a way to structure a large amount of log data in real-time, distributed programming patterns are used to be able to cope with scaling problems when limits in hardware resources are reached. Dealing with log data in real-time means that the system should handle and process log data quick enough so that no latency builds up.
This chapter, therefore, describes how implementing a distributed system to deal with log data can overcome the challenge of running behind in processing log data. This results in a system that models the data into a graph and can do simple computations on the graph in real-time. Firstly the research questions will be listed and explained how they would give insight into the performance of the developed solution. Then the methodology is described on how to answer these research questions.

4.2 Research Questions

The research questions that will be approached or answered in this thesis are:

- RQ #1: What different aspects of log data impact performance of modelling the graph?
- RQ #2: How does the approach perform with varying quantities of log data?
- RQ #3: What kind of computational resource metrics give insight on how performant the approach is?

- RQ #4: When and how much does the approach scale horizontally?
- RQ #5: Does the performance of the approach degrade over time?

All the above-mentioned questions will help conclude whether the proposed solution will be successful in dealing with large amounts of log data and whether the proposed solution actually scales when the log data size will get bigger.

4.3 Methodology

This section will describe the methodology which will be used to answer the questions above. Using the implementation of the previous chapter, several metrics will be collected and researched to report findings which will prove or disprove theories about the developed system. The reasoning behind the choices made when implementing the system accompanied by the findings on researching the performance of the system will eventually lead to a conclusion on the ultimate goal, which is to find a scalable solution when dealing with large amounts of log data.

4.3.1 What different aspects of log data impact performance of modelling the graph?

It is essential to know which factors degrade performance while scaling up in resources can be reduced by using efficient formatted log data. To determine what kind of factors impact peformance when dealing with log data, the log data itself, the algorithm and data structure are looked at. Parameters that describe log data and traces in a measurable fashion are determined as follows:

- #1: The length of the static character part of the log identifier
- #2: The length of a path in a graph
- #3: The number of different paths in a graph
- #4: A combination of the number of paths and the length of paths
- #5: The amount of traces processed

To analyse the impact of these different aspects of log data, a number of tests will be run with a varying input. These tests will be run on a single instance by running unit tests. Results are measured in log lines processed each second and can be used when determining how many instances need to be deployed, while results depict when the single instance reaches its limits. The table 4.3.1 gives the varying input per experiment that is run.

Exp Nr.	Nr. of paths	Depth of path	Static part length	Nr. of traces range
#1	1	10	10 - 1.000.000	100 - 1.000.000
#2	1	10 - 1000	10	100 - 1.000.000
#3	10 - 1000	10	10	100 - 1.000.000
#4	10 - 1000	10 - 1000	10	100 - 1.000.000

The length of the static character part of the log identifier is taken into account while the prefix tree is using the characters of the static part in log data to construct the paths in the graph. Expected is that when the static part of a logline becomes longer, the computation to process and insert it into the graph will take longer as well. Measuring this can be done by generating log lines with a fixed length static part and measuring how many lines will be processed in a time range. This process will be repeated with different fixed lengths and comparing the number of log lines processed per second for each of the procedures.

The length of a path in a graph is defined by the number of log lines in a trace. While there can be reasoned about the amount of logging in production, there is no standard in how much log lines a trace will contain. By constructing a test where the amount of log lines in each trace is fixed, the number of log lines processed per time unit can be measured again. Changing the amount of the log lines per trace in every repeated procedure will result in gaining insight on what the depth of a path contributes to the improvement or degradation of performance in processing log data.

The amount of different paths in a graph is defined by the number of different sequences in all traces processed. The number of different paths can be generated by creating traces that have a fixed amount of different sequences in static log parts following each other. By repeating this experiment multiple times for a growing number of different sequences, the output can again be measured in log lines processed per time range. While we are testing the different amount of paths, it should be noteworthy that the experiment for 1.000 paths can only begin at 1.000 different traces.

The amount of traces processed will be measured for every experiment described above. This will mean that every experiment will be run with a growing amount of traces as input. To reduce the factor of input lag, all the traces will be generated before the experiment and stored in Random Access Memory (RAM) and then be fed to the benchmarking experiment. This will result in measurements that solely rely on processing traces instead of creating the traces, and if there is degrading performance, it means it can only be blamed on the processing part.

A combination of the number of paths and the length of paths will be experimented with to get an overview of the performance when multiple factors are enlarged together. This could mean it will result in the same performance, or it will degrade even faster.

All of these experiments are benchmarks which will be available in the codebase and can be validated locally by running 'go test -v'.

4.3.2 How does the approach perform with varying quantities of log data?

To test the scalability of the implemented solution, the approach will be tested against varying quantities of log data. This will be done in an environment that is set up with log data generators, developed to output a specified amount of log messages with variable input parameters based on the described log data factors above. By feeding log data into the system in a controlled way, the output can be measured in terms of log lines processed per minute and reporting features can be controlled and measured in time to deduct latency of the implemented solution.

4.3.3 What kind of computational resource metrics give insight on how performant the approach is?

To obtain insight about the performance of the system, the cluster described in the previous chapter will be set up. By doing this, Google Cloud Monitoring gives insight into CPU usage time and utilisation accumulated per minute. Besides that, it will also provide bytes used in Random Access Memory (RAM) and Disk usage. Furthermore the exporters for Prometheus that will be set up provide additional information about the deployed NATS cluster in terms of the number of messages received and sent, connections made, number of bytes passing through the cluster and Round-Trip Time of messages. In addition, the solution created will also export log lines, trace and new paths processed to Prometheus. In the end, all of these metrics can be reviewed in either Google's Cloud Console interface or Grafana that visualises the metrics exported to Prometheus.

In summary, the following metrics will be reviewed to gain insight on what resources are being used to indicate which parts of the cluster need to be scaled up at what point in time to deal with large amounts of log data:

- CPU usage time
- CPU utilisation
- RAM usage
- Disk usage
- Log lines processed
- Traces processed
- New paths encountered
- NATS RTT
- NATS Network usage

Primarily the focus for performance measurement will be on the self-created implementation, while other parts of the system are interchangeable with other solutions available in today's market. However it is important to keep in mind that these parts are crucial for the end-to-end process of dealing with log data, so measurements will be done for these parts, and they will be reviewed to decide whether they will become a bottleneck or not in the entire pipeline. In the end, it must be pointed out that the architecture of the complete pipeline is set up in a way that every module can be scaled up while they are all selected on the capability to support distributed processing.

4.3.4 When and how much does the approach scale horizontally?

To measure the performance on how the system will handle the amounts of log data that is fed into the system, the metrics in the previous subsection are used. While several components like the stream processor and NATS are out-of-the-box solutions and can be interchanged with other solutions in the field, the focus of the distributed programming experiments will primarily focus on the solution implemented in this study. To focus on the solution implemented in this study, the complete traces will be generated at once, instead of separate log lines, to reduce the amount and size of the message that passes through NATS and STAN. This will result in being able to handle more traces, and loglines by the solution implemented and therefore, will be a better indication of how the solution implemented responds to large amounts of log data. To find out how the distributed methodology used affects this performance, the child parts of the system are scaled individually. The metrics like log lines processed each second but also all the aforementioned metrics can then be reviewed individually per deployed instance and compared against each other. This enables the system to provide information on whether log lines are distributed equally among the individual instances and how this affects resource usage.

4.3.5 Does the performance of the approach degrade over time?

In this part, the system will be stress-tested as part of the experiment to see what amount of log data the system can handle when running on the machines that are set up for this experiment. By feeding as much traces as possible to the system, the data structure will be enlarged, and performance degradation can be measured by using the aforementioned metrics. Although the first research question will already be able to give insight on the most significant performance measurements for dealing with log data, running and measuring it in a full set up environment will be able to validate or invalidate previous findings.

In the current implementation, a few features are built in to get insight on latent factors about the behaviour of the software. Investigating the Round-Trip-Time of log data being fed into the solution until it results in structured data enables research on how well the solution is able to report anomalies in real-time. Currently one of the features is measuring the performance in nanoseconds from node to node in the resulting graph, where new paths processed can trigger reports on degrading performance. The Round-Trip-Time of a degraded trace can then be defined by the delta of the exact moment of feeding the degraded trace

and the exact moment of reporting back the metric. In addition to the prerequisites for this experiment, one trace should be defined which differs in delta with a preconfigured boundary in the delta between nodes When this is set up, the experiment can be done repeatedly with a different amount of log lines being fed to the solution each time, to see whether the Round-Trip Time degrades when handling larger amounts of log data.

Another feature is comparing the base graph to incoming traces in terms of new paths. In this case, also the Round-Trip-Time of new traces being fed to the system can be measured until it reports a new path. This will indicate again on how the system is dealing with log data in real-time. The experiment can be conducted by using the prerequisites mentioned above and sending traces with a different path compared to the base graph to the system. This metric will enable the solution to report differences in software's behaviour in a manner that it can find changes in paths for different releases.

Chapter 5

Results

In this chapter, results will be concluded from the conducted experiments. By comparing and reviewing the gathered data, the research questions are tried to be answered.

5.1 Research questions

The following research questions were posed:

- RQ #1: What different aspects of log data impact performance of modelling the graph?
- RQ #2: How does the approach perform with varying quantities of log data?
- RQ #3: What kind of computational resource metrics give insight on how performant the approach is?
- RQ #4: When and how much does the approach scale horizontally?
- RQ #5: Does performance of the approach degrade over time?

5.1.1 RQ 1: What different aspects of log data impact performance of modelling the graph?

It can be clearly seen in 5.1 that the depth of the paths in the graph has the most significant impact in performance for the given approach. By enlarging the traces, the depth of the graph gets bigger and more operations are done to build the graph. However, the input that is processed is mostly the same paths which lock the data structure. When the input is distributed amongst multiple deployments, the locking of the data structure becomes less of a problem, and performance will go up again.
Next to the size of the traces, enlarging the static identifier results in the same performance degradation. The Trie edges are built from the static parts, so this comes down to the same issue as described above. In real-world scenarios, the static identifier will be chosen as small as possible not to clutter the log data, so this will be an issue for the given approach.

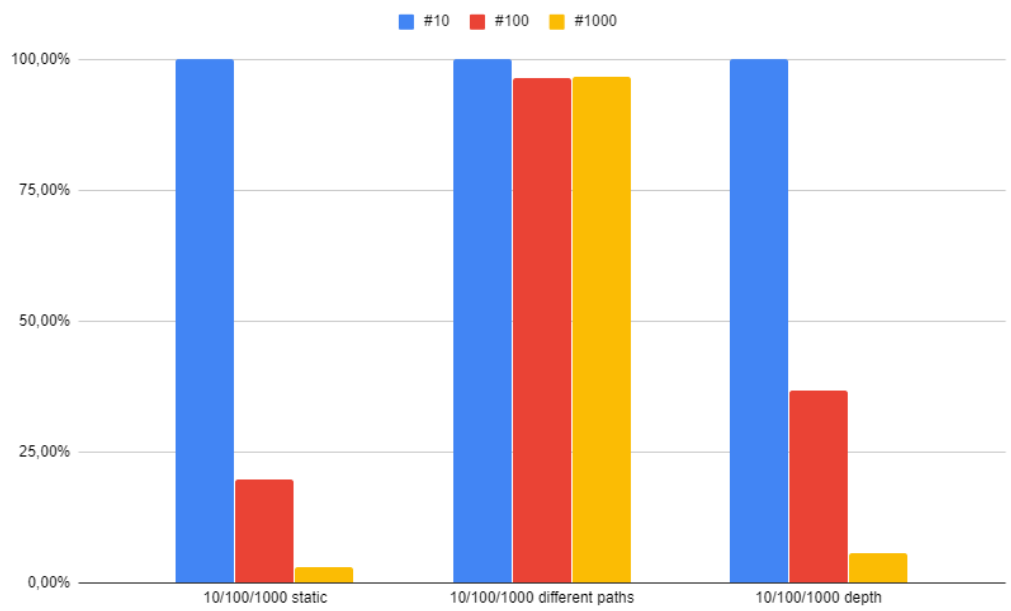


Figure 5.1: Performance at different input for specific log data factors, static identifier, different paths and depth op paths respectively

In 5.1 it can also be seen that multiple paths do not impact the performance of the given approach in processing speed, the graph does increase in size, so it results in more RAM used. This, however, does not decrease the processing speed. Figures X to X in the appendix support the results described here.

RQ1: The depth of the tree structure in the graph has the most significant impact in performance. Longer traces result in heavier computational needs of resources. Different traces, resulting in different paths in the data structure, do not cause the solution to slow down.

5.1.2 RQ 2: How does the approach perform with varying quantities of log data?

The performance does not degrade when more log data is being processed which can be seen in 5.2. One child instance is depicted 5.2 in Grafana, which processes a growing number of log data that is being fed into the approach. It shows that the number of log lines processed per minute remains stable over time for any given input. The main deployment does more merges when large quantities of log data are being processed, but merging is fast enough to not be a factor in scaling the approach. Reporting of paths and anomalies show almost no latency regardless of the amount of log data that is being processed. Figure 5.3 shows round-trip-time (RTT) for messages passing through NATS and getting acknowledged. The

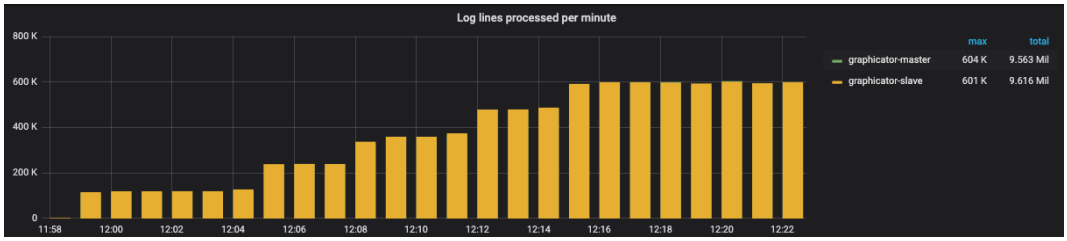


Figure 5.2: Log lines processed per minute in grafana with one child deployment

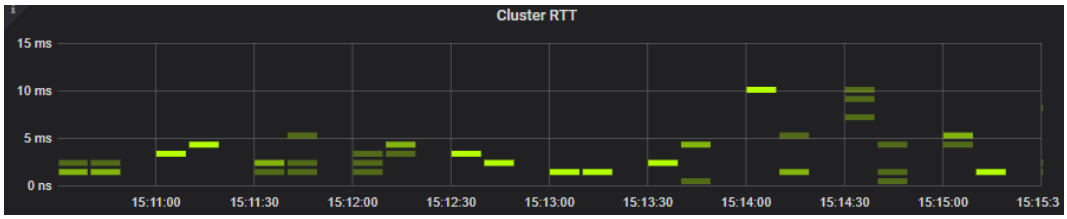


Figure 5.3: STAN Round-Trip-Time request to client

amount of time it takes the approach to report anomalies are almost equal to the time-window for batching messages and the given RTT, which means the deployments report almost instantaneously.

> RQ2: The performance does not degrade when more log data is being processed. Processing log data remains stable at whatever quantity is fed into the proposed solution. Reporting of paths and anomalies show almost no latency even for growing quantities of log data fed into the solution.

5.1.3 RQ 3: What kind of computational resource metrics give insight on how performant the approach is?

The kind of metrics that impact the cluster the most when dealing with large amounts of log data is CPU usage time, RAM usage and disk usage for NATS/STAN. For every instance, the CPU usage time and RAM memory used are gathered and compared to the number of log lines processed per minute in the figures A.8 and A.9. The first instance that responds to the log lines generated is the pipeline stream processor in combination with the tracer. Usage of CPU is quite low, explained by the fact it does very little computation, corresponding with the expectation. RAM usage doubles from 70k lines to 140k lines through to the fact that the stream processor batches the number of incoming messages and then passes them through to the tracer. The stream processor 'Benthos' should be scaled up earlier than the tracer itself but is not the computational heavyweight when we look at the other components in the cluster.

NATS and it's counter component STAN take far more resources of the cluster to handle the log lines. NATS, however, is just a portal for message throughput without any complex

Figure 5.4: RAM usage with two childs, l/m = logs per minute

control on the messages passing, while STAN takes persistence into account. A lot of the CPU usage by NATS and STAN can be explained through the fact that limits are configured for the amount of messages that are stored to disk and that it is constantly checked whether to remove older messages. Both deployments are using far more disk I/O operations which do not benefit performance in this specific cluster while disk operations are slow against the disks mounted. Waiting for these operations to finish, explain parts of the higher amount of CPU usage time.
The actual implementation of the trace structuring solution uses far fewer resources than NATS. Results are shown in the figures A.8 and A.9 clearly indicate that the child deployment is doing the computational part, and the main instance is using very little resources in the cluster. The child deployment, however, does use a lot more RAM than its main counterpart, while it holds more information in run-time to be able to do computations on performance measurement and path detection.

RQ3: CPU usage time is one of the metrics with the biggest impact on whether to scale resources. The solution benefits from distributed workloads and uses little CPU to do the standard computations to build the prefix trie structure. RAM usage stays equal among the main instance and all child instances.

5.1.4 RQ 4: When and how much does the approach scale horizontally?

It is difficult to pinpoint an exact trigger on when the approach has to scale up for performance improvement; this also depends on the resources that are available. However, the most important thing to notice is that RAM used is not distributed in the given approach 5.4 and every deployed instance, child and main, will have similar memory usage. It can also be seen that the RAM usage is low, but theoretically, when doing more complex computations, this could grow and must be taken into account when adding these computations.
CPU usage is almost evenly divided among the child deployments; child deployments should scale up in terms of CPU usage quite easily when they reach a certain preconfigured threshold. Another reason to scale up child deployments is when log data input is largely the same in large amounts, multiple deployments can be faster in terms of less locking of the data structure in run-time. This will be a trade-off against the extra RAM the deployments will be using. The main deployment doe not use up a lot of resources for the merging of partial graphs, and is close to being a non-factor in terms of scalability of the approach.

> RQ4: CPU usage is almost evenly divided among the child deployments, RAM usage is not divided. Automatic scaling can be set by taking account the CPU resource limits of the machine a child deployment is running on. Another reason to scale up child deployments is when log data input is largely the same in large amounts, while this can speed up processing the log data.

5.1.5 RQ 5: Does the performance of the approach degrade over time?

Performance does not degrade over time while the data structure is released to the main instance by preconfiguring, and it can keep up with incoming log data without using excessive resources. In figureA.17 can be seen that when running for half an hour against 1.2 million log lines per minute, which accounts for 10.000 unique nodes and 200 different paths in the graph, the CPU resources required do not degrade over time. Figures A.18 and A.19 show resources used by one of the four child deployments over time. It can be concluded that the CPU resources required remain stable over time when the same amount of log lines are processed per minute. RAM resources used stay even over time, while no new paths and nodes are introduced while the traces processed stay the same over time. Therefore we can conclude that software running in production will eventually reach a certain resource usage in terms of log processing that can cover the highs and lows in the software usage itself. So it is possible to find an optimal setup for a cluster to deal with the amount of logs software in production outputs. Figures A.20 and A.21 show resource usage over time for the main deployment. The main instance actually uses a bit more ram than the child deployments, this while it stores multiple partial graph structures in memory for a short period of time when merging, and therefore it allocates more memory. This behaviour can also be seen in CPU resources used, while the CPU usage is very low, it uses bursts of CPU when partial graph structures are received for the merging of the main graph and the partial graph ab-

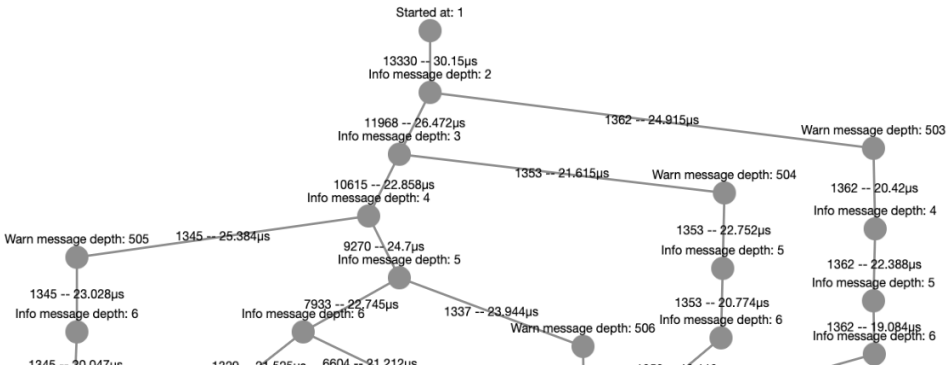


Figure 5.5: Example of resulting Trie structure

sorbed.
While in this experiment a cluster was used running NATS with a normal hard disk mounted, it can be seen that the throughput of messages cannot be processed in time anymore in terms of disk operations. Figure A.22 shows the RAM usage spiking suddenly when running already for 15 minutes.

In Grafana the metrics of log lines processed per minute A.23, and network throughput A.24 were tracked, and it can be seen that after 25 minutes processing 1.2 million log lines per minute it starts throttling because of STAN. However STAN can be configured differently like with a Solid-State Drive (SSD) or a database to improve throughput, best performance will be when NATS and STAN are running on dedicated machines and not in a cloud cluster provided by Google while STAN especially is disk intensive.
By sending a specific trace multiple times when already processing 1.2 million log lines per minute and have the main instance reporting the exact moment in time when it was received, it was concluded that the average time between sending the trace and the report was 8ms. Figure 5.3 explains why there is a small delay in sending and receiving messages, but it indicates that even when dealing with large amounts of log data, the anomaly detection is still near instantaneous when a trace is received by one of the child deployments running.

RQ5: Performance does not degrade over time. Running the solution for half an hour against 1.2 million log lines per minute kept performances equal to the start.

5.2 Discussion

An example of a resulting graph can be reviewed in figure 5.5 where the nodes labelled with the log message and the edges are annotated with the number of times connected and the performance in time.

Looking at performance, namely processing log data per minute, the biggest impact comes from the depth of the tree. Very deep paths also make the graph very tedious to work

with and identify abnormalities. Both challenges could potentially be solved by transforming the incoming traces in real-time. The given approach can do computations before and after insertion in the graph. This means that a sequence of log lines can be transformed into one predefined one, e.g. 'A,B,C,D' could be transformed into 'Successful payment of invoice'. Making it easier to understand the graph, while some sequence of events can be translated into some understandable business ruling and making the graph less deep.

While executing the benchmarks for one instance, limiting the number of concurrent threads impacted the results in a positive matter. In the figures A.2 and A.3 the same three experiments are conducted. Respectively the traces from experiment one to three have 10, 100 or 1000 paths in the resulting tree. Again the performance is measured by processed log lines per seconds. In the first figure A.2, the performance degraded harshly after the first two input sizes. By locking the data structure using mutual exclusion and spawning a lot of goroutines (threads), context switching becomes a relatively big overhead. By rate-limiting the number of goroutines that can spawn by the number of CPU cores, the results improved as can be seen in figure A.3. When more depth is added in the graph structure by lengthier traces, there are more different nodes and edges in the data structure. Not every goroutine touches the mutual-exclusive locks at the same time anymore, so setting a core limit has less effect. This means that when you deal with large quantities of log data that often exists from the same sequence of events, it can speed up the processing of log data by scaling the child deployments at an early stage.

The given approach can scale at every part in the environment, the large data traffic over the pub/sub system is one of the factors that need to be taken into account when implementing the given approach. Also, when doing different kinds of computations in real-time by the scalable deployments, RAM usage needs to be taken into account while memory is not shared amongst the child deployments. Reporting features of the given approach give feedback in real-time without noticeable latency, and performance does not degrade over time.

Chapter 6

Case study

Besides the scalability possibilities of the proposed solution, we also want to know if the solution is usable in the industry. Therefore the solution is used in an existing project of our industry partner in this thesis: Weave. In this chapter, the methodology of the case study is explained, the findings of the developers are shared and the results are discussed. In the end, the next two questions will be focused on in this case study.

- RQ #6: To which extend does structuring log data into a graph in real-time help giving insight to software's behaviour?
- RQ #7: How well does the proposed solution integrate with an existing project?

6.1 Weave

Weave is a company that develops custom software solutions for clients in all sorts of industries. It is a company that heavily relies on cloud-native solutions for deploying their software, like Google cloud products and Kubernetes. Often the software that is written uses a microservice architecture where the logs that are outputted are separated from each other. At this point, Weave does not implement analysis over log data, however, they do want a generic solution to keep track of a system's health. Log data is a common denominator in all the projects, so the implemented solution deemed a perfect fit for Weave.
For this case study, an energy supplier platform which runs on top of kubernetes using about 20 different microservices is made available to deploy the solution in. The energy supplier platform includes automated contract creation, invoicing, payments, usage retrieval, dynamic cost calculation and more. The platform is primarily used to automate onboarding of new customers, give the customers insight about their energy usage and costs and help them save money. The novel idea of the energy supplier is that you pay the exact same per hour that energy costs on a specific hour on the energy market and only pay a subscription fee each month. All the software is written in Golang and logging is done by one logging library used by all microservices.

6.2 Methodology

6.2.1 Setup

For the case study, we opted for a controlled environment where integration tests are run. This is chosen because this makes the output more predictable and we can conclude fast where changes are coming from. The integration test environment is also redeployed every time integration tests are run, this also makes it easier to deal with cutting off a constant stream of logs. By deploying the proposed solution in this environment, it also adds a layer on top of the integration tests to give more context on what parts of the software are actually interacting when running certain scenarios. To experiment with the multiple features, the proposed solution brings we deployed the solution in the environment and reran with different steps to assess usability and accuracy of the solution.

The idea behind running the solution against the integration test environment is that there is immediate feedback of added features to the codebase and how it differs from the previous version of the software. We first run the integration tests with the proposed solution integrated so it can passively learn and model the graph. This result is stored on a persistent disk, and whenever new features and integration tests are released, it can rerun this against the previously modelled graph. Rerunning the integration tests for this case study was done for a week during active development of the platform.

- #1: Deploy the solution and run passively against the integration tests available
- #2: Save the created graph on a persistent disk
- #3: Review the created graph
- #4: Repeat integration tests with new features and integration tests that are being released
- #5: Review the results reported by the proposed solution

To review the results reported by the proposed solution, a small tool is built as an extension that interacts with the solution. It visualises the complete graph, only the paths containing error level log line messages or the reported new- and performance degraded paths one-by-one. With the help of this visualisation tool and an active developer of the platform, the accuracy of the created graph and reported paths are determined. In cooperation with a developer of the platform, the initial base graph is reviewed as well as the reported paths in following runs of the integration tests. The created graph will be reviewed briefly, while it will be quite big, the focus will be on the paths containing error level messages. Then the reported paths will be discussed and flagged as ”correct”, ”bug”, ”incident”. Correct can be seen here as the correct/expected execution flow. A bug means that this specific path should not have existed and an incident means the path is not the ideal or expected execution flow but is acceptable for the developer.

With this tool also comes active learning, by the fact that the developer can accept the reported paths when he or she feels like this is a correct change of the execution the

software performs. Accepting the reported paths will store them on the persistent disk for the following run of the integration tests.

6.2.2 Implementation

To be able to integrate the solution into the existing project's environment, small changes in the codebase are needed. First off the library that is used for logging is adjusted, so it generates a trace identifier whenever there is none in the context of the execution path. While NATS is already used in the environment, the logger is adjusted to immediately publish log messages to NATS. Third, the internal requests from microservice to microservice are appended with a trace header including the trace identifier present in the context of the execution path. Lastly, API middleware is added, so incoming requests are checked for the trace header, if it is present, it puts it in the context of the execution path for the logger. This enables traces to contain execution paths of multiple microservices that are run sequentially and therefore, should provide more context of what the actual execution flow is.

6.3 Results

6.3.1 RQ 6: To which extend does structuring log data into a graph in real-time help giving insight to software's behaviour?

The structured model together with the reporting and visualisation tool helps to determine the correct functionality of the platform. It can be used in a controlled environment to quickly gather feedback on what changes in execution are happening by changes in the source code. Reported paths containing an error message are often in need to be analysed in more detail. Together with the merge request of the source code and the visualisation tool for reported different paths, it helps to assess whether the new changes are acceptable.

Looking at the results of the case study, the findings are promising. Even when running in a smaller controlled environment, bugs are detected and solved. The visualisation tool, however, is required to make the solution usable for the developer. It provides an extra layer of information which is easy to walk through and helped to determine in this case whether feature branches change more than the expected flows.

The graph itself gets large very quickly and is not likely to be analysed by hand. Suggestions here are that the graph could be queryable, like searching for some payload in the log message, filtering by log level or by some condition between nodes, e.g. microservice switch or performance. Most of the by hand analysis requires in-depth knowledge about the system; the reported paths by the solutions are there to make the detection of anomalies easier and more accurate. By reviewing the numbers, we can conclude new paths are mostly correct but can indicate anomalies. By running the solution in the integration test environment, there are not that many new paths per run, so it does not take too much time of the developer to review them. Error paths almost always indicate a bug or an incident.

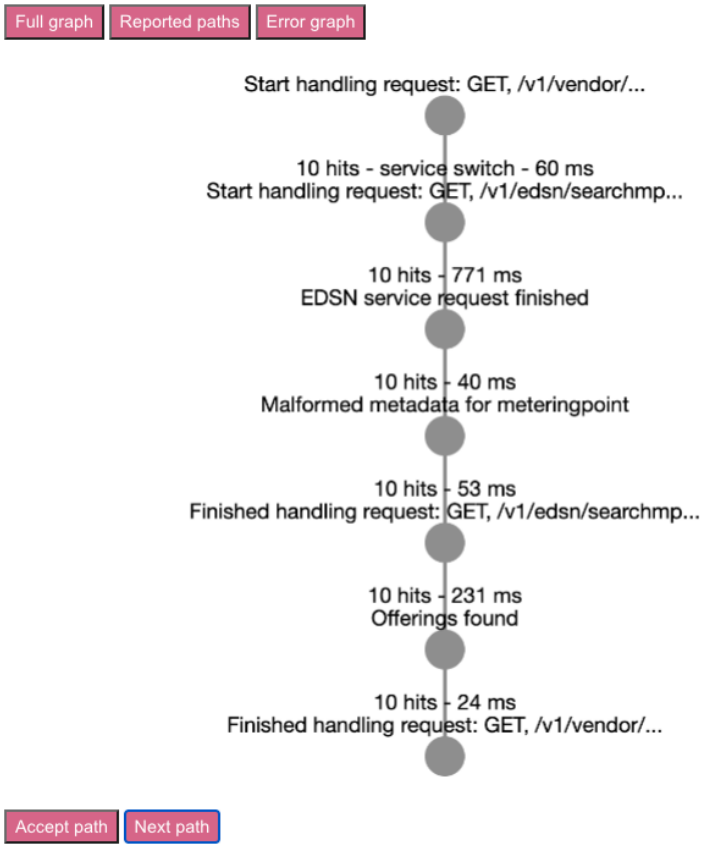


Figure 6.1: Bug reported by visualization tool

Reviewing the modelled graph in its entirety is still a job what takes a lot of time. In total, after running the integration tests, there are 362 different nodes, 254 edges with a total of 114 different paths. Next to the ten error paths in the existing graph, 28 newly reported paths are examined. From the ten error paths examined, nine of them were incidents only happening in the controlled environment by missing optional data in the database; the remaining one was classified as a bug. The 28 newly reported paths were all found correct but one classified as an incident, only happening because of a missing migration in the controlled environment.

After modelling the graph for the first time with the integration tests that existed the focus was on the paths that contained log lines with log level ”error”. There a bug was found that was actually in the issue list of the project, that was occurring on production, but still needed to be analysed. Figure 6.1 displays the found path as an incoming API request internally calling a microservice which throws an error but still returned the correct result.

Therefore this flow was deemed correct by the integration tests, but still was behaving

unexpectedly. By reviewing the path in the tree, the execution path was straight-forward, and the bug came from newly imported objects from an external API that was not found in the database of the platform yet. In the source code, this object was first tried to be retrieved from the database, in which case it gave an error and then still inserted the new object retrieved from the external API. This was then fixed by starting a transaction where the insertion and retrieval were done to always return an object without logging an error. In the end, two separate transactions are transformed into one which saves thousands of database calls a day, and an error log message was removed while it was not actually breaking. By reviewing this in the visualisation tool, it became in his own words very easy for the developer to fix, while the representation was straight-forward with the way the source code was executing the software. Not only the execution in the API could be followed now, but also from the API to the microservice it was calling.

Another finding by the log analysis solution was that after a release of a feature for emailing reminders for settlement invoices, old invoices were picked up by the cron job. This was actually not tested by the integration tests, but before the integration tests are run, the database is seeded with data required for the integration tests. The invoices seeded were not updated, and therefore the log analysis solution reported new paths that were unexpectedly executed. The seeded invoices were therefore updated, but there was already a migration for production, so this would not have happened on the production environment altogether. Still, the solution gave extra information about what changes the feature actually had brought with it, and the feedback the solution reported was actually found correct by the developers.

Finally, the solution was set up to report degradation of performance when the time between nodes doubled or more, this was not a correct setting, while most of the nodes were less than a millisecond apart. Therefore the log analysis solution reported many degraded paths, while they were actually behaving correctly, within acceptable boundaries. When the degradation threshold was preset, there were no more degraded paths reported.

Reviewing the execution path, not only in one component of the system but in the context of multiple components, the analysis of unexpected execution paths becomes easier, while more information is given. The user experience of the visualisation tool can still be greatly improved, but in this minimal version, it already shows the developer accurate context about execution throughout a microservice architectural platform. This made further analysis of bugs and incidents easier, though they were missing the reference to the source of the actual message in the codebase.

RQ6: Looking at the results of the case study, the findings are promising, bugs are detected and solved. The visualisation tool, however, is required to make the solution usable for the developer. Reviewing the modelled graph in its entirety is still a job what takes a lot of time, reported paths create a smaller scope.

6.3.2 RQ 7: How well does the proposed solution integrate with an existing project

One of the objectives of the solution is that it can be integrated into existing platforms. While almost all software projects work with log data, the solution should work by changing the source code as little as possible. The requirements of the log messages are already given chapter 33, and at Weave this meant changing one logger that is universally used within projects to output a trace identifier. The rest of the components were already in the log messages, like timestamp and a static part.

Attaching a trace identifier to the log message is done by attaching an identifier to the execution context whenever it is not present yet. In practice, this meant for the energy supplier platform at Weave that all incoming API requests created a trace identifier in the context, and when starting a cron job, a trace identifier is created. After that, the identifier is passed through the execution path.

To also make chaining of execution paths in the graph possible between microservices, also the middleware of the service call had to be adjusted. This is done through setting the header with the trace identifier in the context, and the receiving microservice will put this header into its execution context.

Integrating the solution into the energy supplier platform was, therefore, not that time-consuming. However, when an existing project is using a different pub/sub system, a different tracer must be used and written.

RQ7: The proposed solution depends on a pub/sub system and log collection in a kubernetes environment. The software to analyse only requires a logger that needs to be set up with three required fields. Therefore making the solution work for the software did not present problems and should not in many cases.

6.3.3 Controlled environment versus production environment

The proposed solution can be helpful in a controlled environment and work as a tool to help developers understand changes made to the codebase and help identify unexpected behaviour when implementing new features. However, the proposed solution is built for being scalable and be deployable in production environments. There are a few differences to consider when comparing the case study to a production environment, namely the timeframe in which to model the graph, deploying new releases, and analysis of reported paths. The proposed solution works best when the modelled graph contains all execution paths at least once. However, this is not a trivial action to perform.
A timeframe must be chosen in which the proposed solution will learn its initial graph, only with in-depth knowledge of the system this timeframe can be chosen. Reported paths are dependent on the modelled graph and will therefore be more accurate and useful when the

modelled graph is complete. Also accepting new paths by the visualisation tool may be time-consuming depending on how many paths are reported and why so more research will be needed to conclude anything meaningful about the reported paths in a production environment.

Chapter 7

Conclusions and Future Work

Computer systems output enormous amounts of log statements, and the log statements contain information about the state of a running system. Monitoring software behaviour is essential, and in many cases, it can be learned from the log statements. Manually going through logs to extract relevant data becomes difficult when the number of log statements grows, and useful information can be overseen. Therefore a systematic way of dealing with large amounts of log data is proposed in this thesis.
The approach given in this thesis is about scaling a solution that structures and models log data so that further analysis is made available through distributed deployments. A complete architecture is drawn to set up an environment where, together with the implemented solution log data analysis can scale horizontally to deal with large quantities of log statements. The proposed architecture includes multiple parts, which are all horizontally scalable. The implemented solution works via scalable units that all build a partial graph and can do computations on them, and one main instance which periodically retrieves the partial graphs and controls reporting of anomalies. Existing tools from the industry like Prometheus are utilised to analyse statistics in a scalable way.

In the first part of the thesis, we pose research questions, beginning with, what different aspects of log data impact performance of modelling the graph. We find that the depth of the paths in the graph, which is equal to the length of the log traces, have the biggest impact on the performance of processing log data. Longer traces result in heavier computational needs of resources. Different traces, resulting in different paths in the data structure, do not cause the solution to slow down.

The second research question we posed was, how does the approach perform with varying quantities of log data. The performance does not degrade when more log data is being processed, processing log data remains stable at whatever quantity is fed into the proposed solution. Also, latency for reporting remains within milliseconds.

The third question we posed was, what kind of computational resource metrics give insight into how performant the approach is. We found that CPU- and disk utilisation for the pub/sub system in the proposed architecture was the heaviest in resource usage. Child de-

ployments of the implemented solution share CPU- usage when scaled horizontally, RAM usage, however, is not, but memory usage, in general, is low for large graphs. When doing more complex computations in the child deployments, RAM usage, however, should be taken into account while it is not shared.

The fourth question, when and how much does the approach scale horizontally, depends on the needs of your system. When large quantities of the same sequence of log statements are being read, early scaling of the child deployments can reduce the latency of log processing. In most cases, the most important metric to take into account is CPU- usage, and child deployments could be scaled based on threshold.

The final question that we posed was, does performance of the approach degrade over time. We found that the performance does not degrade over time when stress testing on the relatively small cluster available, the pub/sub system was the first part of the architecture that began throttling.

Next, to determine the usability and accuracy of the tool, a case study was done at Weave, a company that builds custom cloud-native software solutions. With the help of a developer and a visualisation tool findings in an actively developed energy platform were discussed in a controlled environment. The case study resulted in an extra layer on top of integration tests which helped give insight to what new and changed paths source code adjustments were leading to. The tool helped to trace back bugs and other findings and also gave more detailed and more precise information than the integration tests by themselves.
Two questions were posed to determine the usability, the first one being, to which extend does structuring log data into a graph in real-time help giving insight to software's behaviour. Looking at the results of the case study, the findings are promising, bugs are detected and solved. The visualisation tool, however, is required to make the solution usable for the developer. Reviewing the modelled graph in its entirety is still a job what takes a lot of time, reported paths create a smaller scope.

The second question we posed was, what kind of computational resource metrics give insight on how performant the approach is. The proposed solution depends on a pub/sub system and log collection in a kubernetes environment. The software to analyse only requires a logger that needs to be set up with three required fields. Therefore making the solution work for the software did not present problems and should not in many cases.

The proposed solution enables a scalable way of real-time log analysis on a graph. At this point, however, only simple computations are done, and the output can still be hard to understand by itself. The solution incorporates a reporting feature which helps to identify anomalies in running software systems in a very generic way. To get a better understanding of where anomalies may lie, an active developer of its system mostly has the best knowledge for this. Dynamically adding computations and new reporting rules to the proposed solution can, therefore make this tool much more relevant in the industry. Not only reporting anomalies but also statistics about a running system can be extracted from already existing

log output by dynamically adding specific rules for a specific system.
Next, the graph itself can be smaller if paths are merged that are similar without creating non-existing execution paths. Tools like DFASAT [34] already do this kind of merging, but within the proposed solution this could potentially be done in a divide and conquer like algorithm.
This could make it possible in real-time, and more existing analysis techniques will become available.

Bibliography

[1] ActiveMQ open source multi-protocol messaging. https://activemq.apache.org/. Accessed: 2020-02-03.

[2] Benthos. https://www.benthos.dev/. Accessed: 2020-02-03.

[3] CNCF cloud native computing foundating. https://cncf.io/. Accessed: 2020-02-03.

[4] Docker a container technology for linux that allows a developer to package up an application. https://docker.com/. Accessed: 2020-02-03.

[5] Envoy envoy is an open source edge and service proxy, designed for cloud-native applications. https://www.envoyproxy.io/. Accessed: 2020-02-03.

[6] Apache Flink. https://flink.apache.org/. Accessed: 2020-02-03.

[7] FluentD. https://www.fluentd.org/. Accessed: 2020-02-03.

[8] GO go is an open source programming language that makes it easy to build simple, reliable, and efficient software. https://golang.org/. Accessed: 2020-02-03.

[9] Google cloud software company. https://google.com/. Accessed: 2020-02-03.

[10] GRAFANA. https://grafana.com/. Accessed: 2020-02-03.

[11] HADOOP the apache hadoop project develops open-source software for reliable, scalable, distributed computing. https://hadoop.apache.org/. Accessed: 2020-02-03.

[12] Jaeger a distributed tracing system. https://www.jaegertracing.io/docs/1.18/architecture/. Accessed: 2020-02-03.

[13] JSON. https://www.json.org/json-en.html. Accessed: 2019-10-28.

[14] PROMETHEUS. https://kubernetes.io/. Accessed: 2020-02-03.

[15] LOGSTASH. https://logz.io/logstash-td/?utm_source=google&utm_medium=cpc&utm_campaign=Logstash_Global&utm_term=logstash&utm_content=logstash-exact&gclid=EAIaIQobChMIzI_e94GW4wIVTOh3Ch1FHAYNEAAYASAAEgK5k_D_BwE. Accessed: 2020-02-03.

[16] Microsoft SQL. https://docs.microsoft.com/en-us/sql/sql-serverlsql-server-technical-documentation. Accessed: 2020-02-03.

[17] NATS open source cloud native messaging system. https://nats.io/. Accessed: 2020-02-03.

[18] Sumo Logic cloud based log analysis platform. https://techcrunch.com/2017/06/27/sumo-logic-lands-75-million-series-f-on-the-road-to-ipo/?guccounter=1, . Accessed: 2020-08-01.

[19] Elastic software monitoring stack. https://www.investors.com/news/technology/elastic-ipo-initial-public-offering/, . Accessed: 2020-08-01.

[20] OpenTracing vendor-neutral apis and instrumentation for distributed tracing. https://opentracing.io/. Accessed: 2020-02-03.

[21] PROMETHEUS. https://www.prometheus.io/. Accessed: 2020-02-03.

[22] RabbitMQ open source message broker. https://www.rabbitmq.com/. Accessed: 2020-02-03.

[23] Source code source code for the approach. https://github.com/rpjproost/log-analysis. Accessed: 2020-10-06.

[24] HADOOP apache storm is a free and open source distributed realtime computation system. https://storm.apache.org/. Accessed: 2020-02-03.

[25] Trie in computer science, a trie, also called digital tree or prefix tree. https://en.wikipedia.org/wiki/Trie. Accessed: 2020-02-03.

[26] Weave software industry partner for this thesis. https://weave.nl/. Accessed: 2020-02-03.

[27] Zipkin zipkin is a distributed tracing system. https://zipkin.io/. Accessed: 2020-02-03.

[28] Ivan Beschastnikh, Yuriy Brun, Sigurd Schneider, Michael Sloan, and Michael D Ernst. Leveraging existing instrumentation to automatically infer invariant-constrained models. In *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering*, pages 267–277, 2011.

[29] Ivan Beschastnikh, Yuriy Brun, Jenny Abrahamson, Michael D Ernst, and Arvind Krishnamurthy. Unifying fsm-inference algorithms through declarative specification. In *2013 35th International Conference on Software Engineering (ICSE)*, pages 252–261. IEEE, 2013.

[30] Marcello Cinque, Domenico Cotroneo, and Antonio Pecchia. Event logs for the analysis of software failures: A rule-based approach. *IEEE Transactions on Software Engineering*, 39(6):806–821, 2012.

[31] Perttu Halonen, Markus Miettinen, and Kimmo Hätönen. Computer log anomaly detection using frequent episodes. In *IFIP International Conference on Artificial Intelligence Applications and Innovations*, pages 417–422. Springer, 2009.

[32] Pinjia He, Jieming Zhu, Shilin He, Jian Li, and Michael R Lyu. An evaluation study on log parsing and its use in log mining. In *2016 46th Annual IEEE/IFIP International Conference on Dependable Systems and Networks (DSN)*, pages 654–661. IEEE, 2016.

[33] Pinjia He, Jieming Zhu, Zibin Zheng, and Michael R Lyu. Drain: An online log parsing approach with fixed depth tree. In *2017 IEEE International Conference on Web Services (ICWS)*, pages 33–40. IEEE, 2017.

[34] Marijn JH Heule and Sicco Verwer. Exact dfa identification using sat solvers. In *International Colloquium on Grammatical Inference*, pages 66–79. Springer, 2010.

[35] Jay Kreps, Neha Narkhede, Jun Rao, et al. Kafka: A distributed messaging system for log processing. In *Proceedings of the NetDB*, volume 11, pages 1–7, 2011.

[36] Marcin Kubacki and Janusz Sosnowski. Multidimensional log analysis. In *2016 12th European Dependable Computing Conference (EDCC)*, pages 193–196. IEEE, 2016.

[37] Qingwei Lin, Hongyu Zhang, Jian-Guang Lou, Yu Zhang, and Xuewei Chen. Log clustering based problem identification for online service systems. In *2016 IEEE/ACM 38th International Conference on Software Engineering Companion (ICSE-C)*, pages 102–111. IEEE, 2016.

[38] Rafael P Martínez-Álvarez, Carlos Giraldo-Rodríguez, and David Chaves-Diéguez. Large scale anomaly detection in data center logs and metrics. In *Proceedings of the 12th European Conference on Software Architecture: Companion Proceedings*, pages 1–4, 2018.

[39] Antonio Pecchia, Marcello Cinque, Gabriella Carrozza, and Domenico Cotroneo. Industry practices and event logging: Assessment of a critical software development process. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 2, pages 169–178. IEEE, 2015.

[40] Tao Qin, Yuli Gao, Lingyan Wei, Zhaoli Liu, and Chenxu Wang. Potential threats mining methods based on correlation analysis of multi-type logs. *IET Networks*, 7(5): 299–305, 2017.

[41] Benjamin H. Sigelman, Luiz André Barroso, Mike Burrows, Pat Stephenson, Manoj Plakal, Donald Beaver, Saul Jaspan, and Chandan Shanbhag. Dapper, a large-scale distributed systems tracing infrastructure. Technical report, Google, Inc., 2010. URL `https://research.google.com/archive/papers/dapper-2010-1.pdf`.

[42] Muhammad IH Sukmana, Kennedy A Torkura, Feng Cheng, Christoph Meinel, and Hendrik Graupner. Unified logging system for monitoring multiple cloud storage providers in cloud storage broker. In *2018 International Conference on Information Networking (ICOIN)*, pages 44–49. IEEE, 2018.

[43] Reeta Suman, Behrouz Far, Emad A Mohammed, Ashok Nair, and Sanaz Janbakhsh. Visualization of server log data for detecting abnormal behaviour. In *2018 IEEE International Conference on Information Reuse and Integration (IRI)*, pages 244–247. IEEE, 2018.

[44] Kang Sun, Luoming Meng, Shaoyong Guo, Siya Xu, Ying Wang, and Weijian Li. An approach of anomaly diagnosis with logs for distributed services in communication network information system. In *2017 IEEE International Symposium on Parallel and Distributed Processing with Applications and 2017 IEEE International Conference on Ubiquitous Computing and Communications (ISPA/IUCC)*, pages 938–940. IEEE, 2017.

[45] Erika Sy, Sam Ade Jacobs, Aldo Dagnino, and Yu Ding. Graph-based clustering for detecting frequent patterns in event log data. In *2016 IEEE International Conference on Automation Science and Engineering (CASE)*, pages 972–977. IEEE, 2016.

[46] Risto Vaarandi and Mauno Pihelgas. Logcluster-a data clustering and pattern mining algorithm for event logs. In *2015 11th International conference on network and service management (CNSM)*, pages 1–7. IEEE, 2015.

[47] Neil Walkinshaw, Bernard Lambeau, Christophe Damas, Kirill Bogdanov, and Pierre Dupont. Stamina: a competition to encourage the development and assessment of software model inference techniques. *Empirical software engineering*, 18(4):791–824, 2013.

[48] Rick Wieman. *What Does Passive Learning Bring To Adyen?* PhD thesis, Master's thesis, Delft University of Technology, the Netherlands, 2017.

[49] Wei Xu, Ling Huang, Armando Fox, David Patterson, and Michael I Jordan. Detecting large-scale system problems by mining console logs. In *Proceedings of the ACM SIGOPS 22nd symposium on Operating systems principles*, pages 117–132, 2009.

[50] Imam Yagoub, Muhammad Asif Khan, and Li Jiyun. It equipment monitoring and analyzing system for forecasting and detecting anomalies in log files utilizing machine learning techniques. In *2018 International Conference on Advances in Big Data, Computing and Data Communication Systems (icABCD)*, pages 1–6. IEEE, 2018.

[51] Matei Zaharia, Reynold S Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J Franklin, et al. Apache spark: a unified engine for big data processing. *Communications of the ACM*, 59(11):56–65, 2016.

[52] Jingfen Zhao, Peng Zhang, Yong Sun, Qingyun Liu, Guolin Tan, and Zhengmin Li. A high throughput distributed log stream processing system for network security analysis. In *2017 IEEE 9th International Conference on Communication Software and Networks (ICCSN)*, pages 1092–1096. IEEE, 2017.

Appendix A

Experiment Results

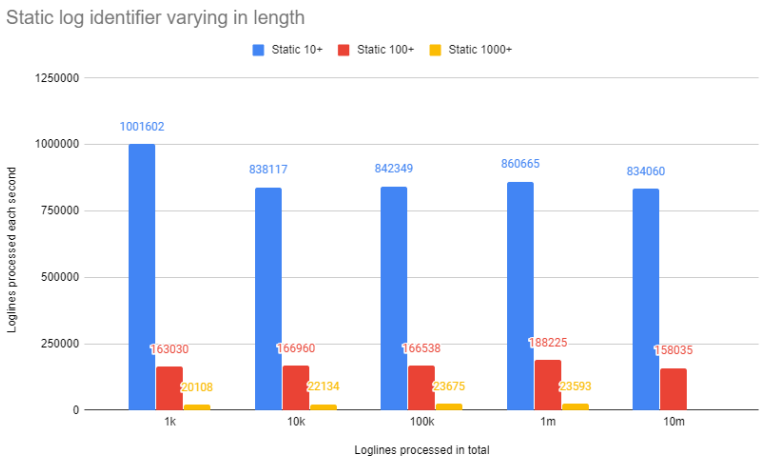


Figure A.1: Three experiments are conducted, where each have a static part varying in size appended on top of their respective static parts

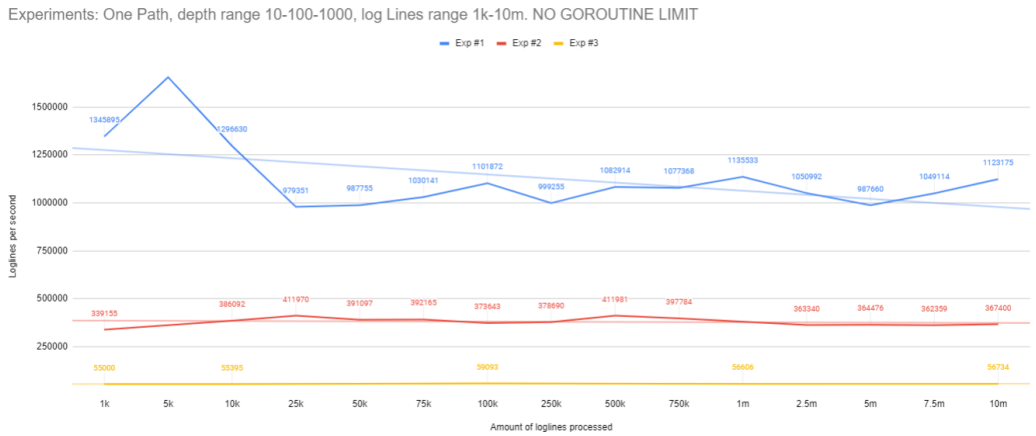


Figure A.2: Experiment with ranging path depth conducted without goroutine limit

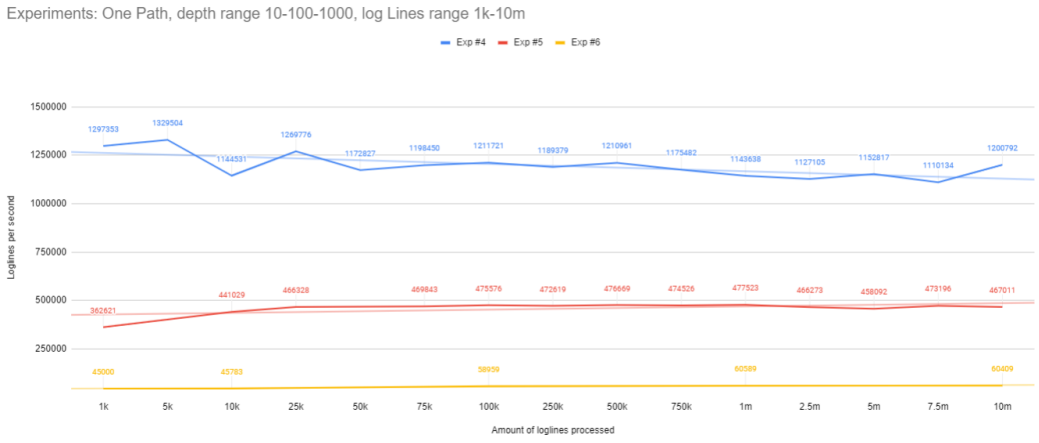


Figure A.3: Experiment with ranging path depth conducted with goroutine limit based on available cores/HT

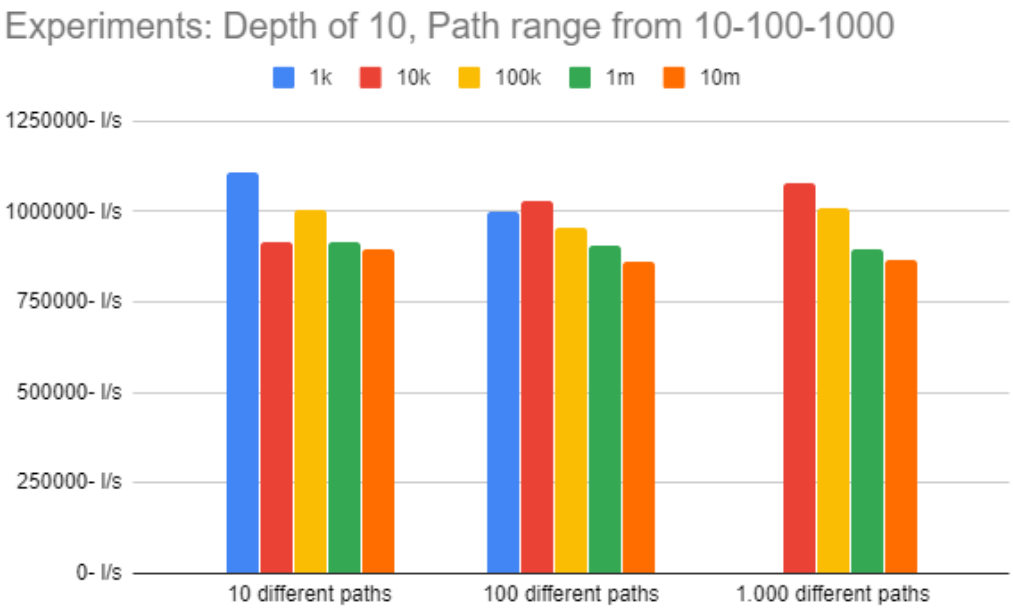


Figure A.4: Experiments with varying amount of paths

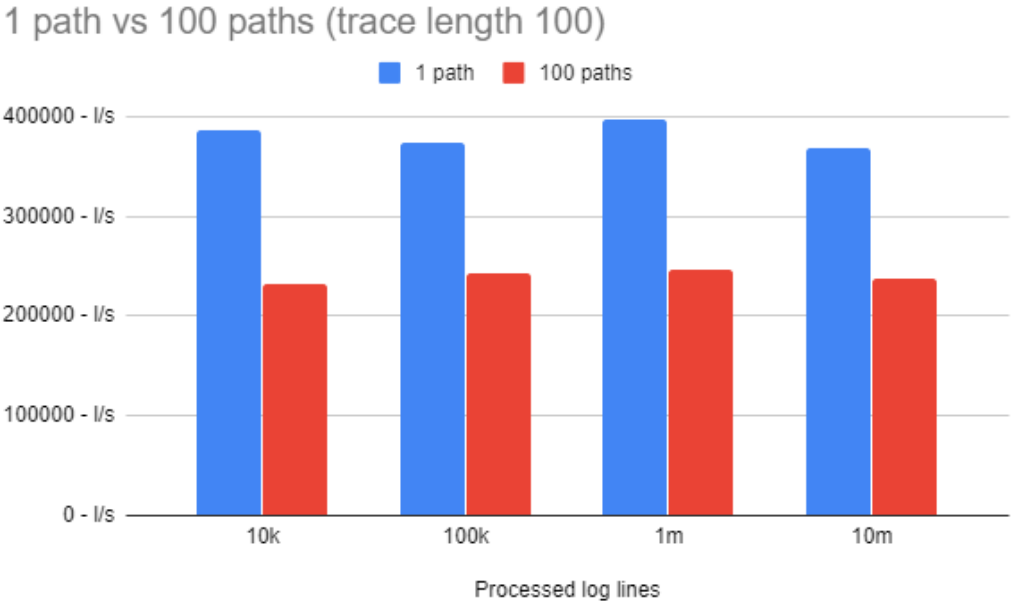


Figure A.5: Performance comparison 1 path vs 100 paths with path depth set to 100

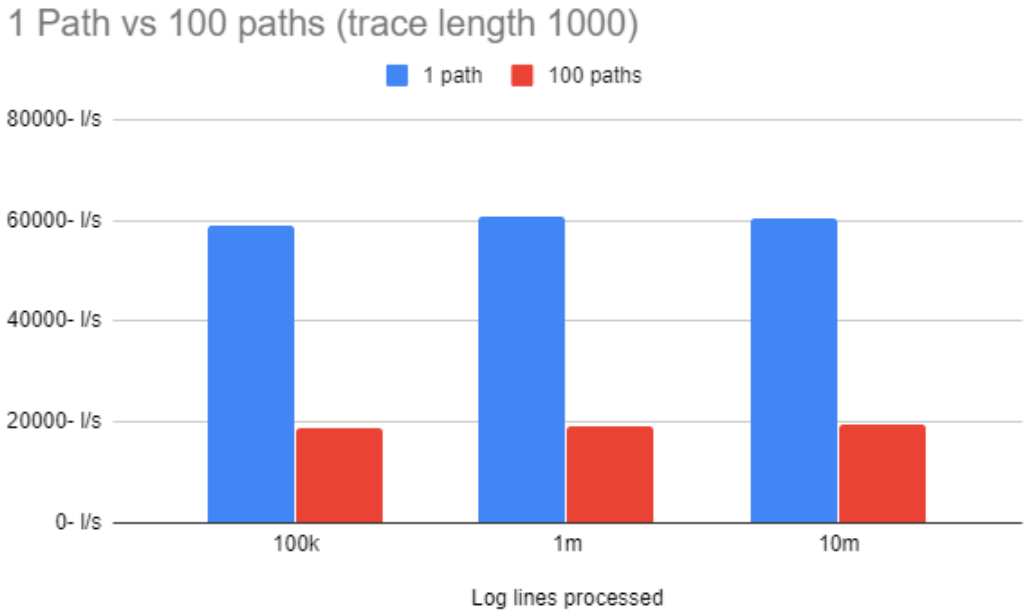


Figure A.6: Performance comparison 1 path vs 100 paths with path depth set to 1000

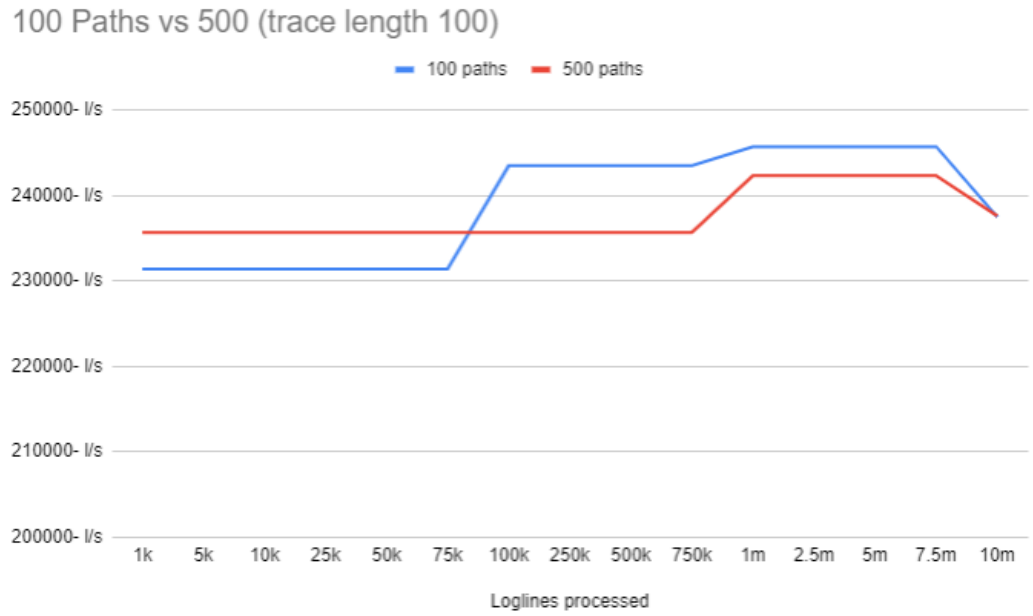


Figure A.7: Performance comparison 100 paths vs 500 paths with path depth set to 100

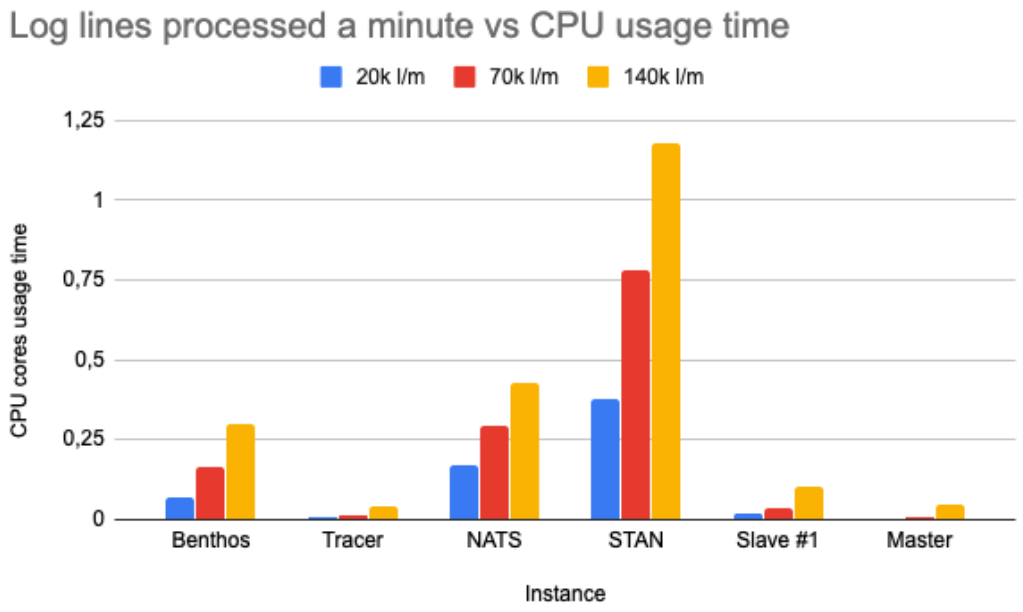


Figure A.8: CPU metrics for cluster instances against lines processed per minute

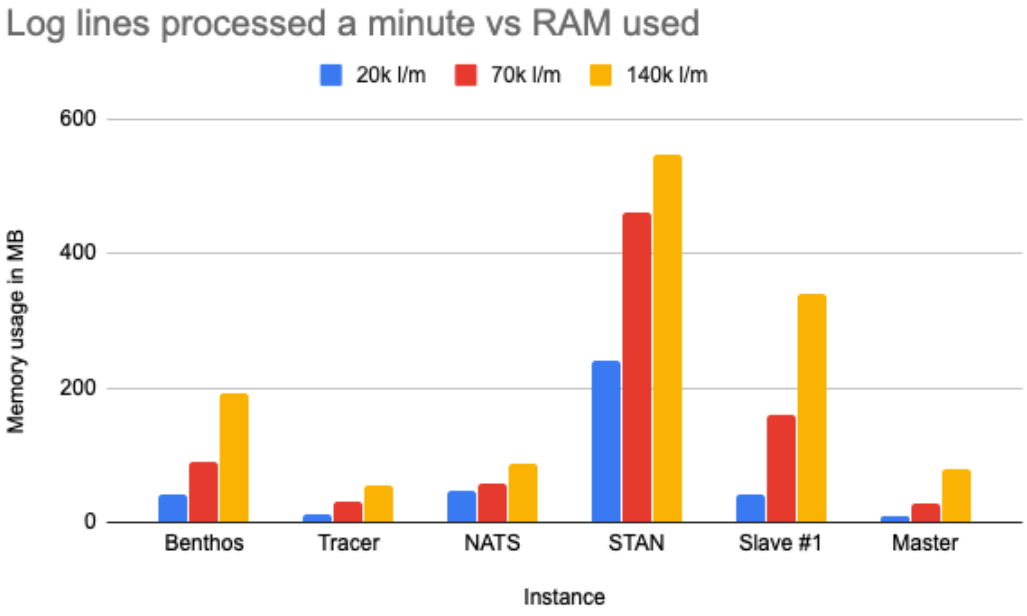


Figure A.9: RAM metrics for cluster instances against lines processed per minute

Figure A.10: CPU metrics STAN, one slave and master against lines processed per minute

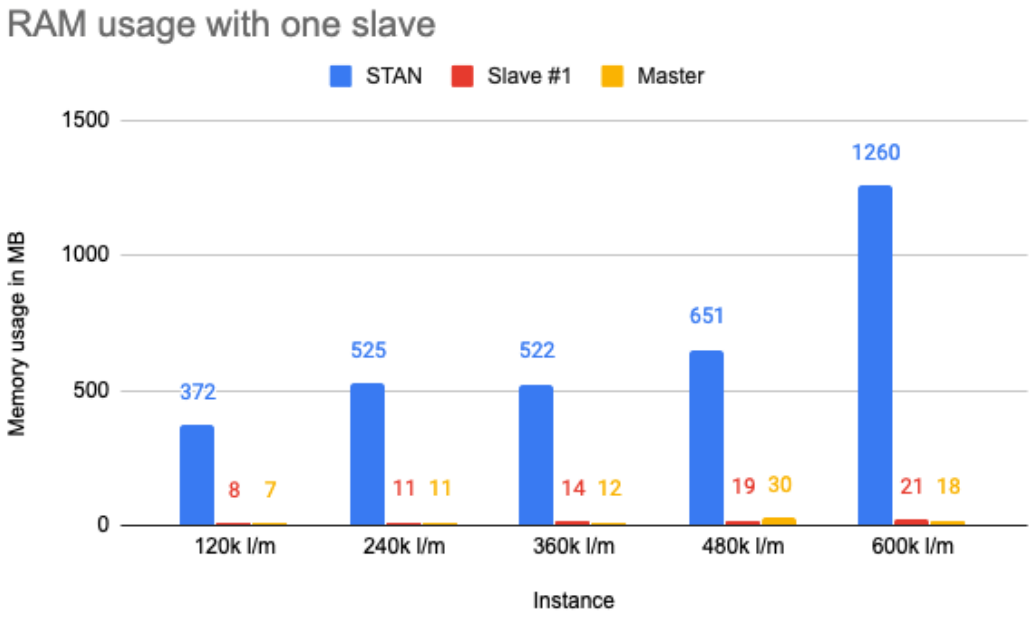


Figure A.11: RAM metrics STAN, one slave and master against lines processed per minute

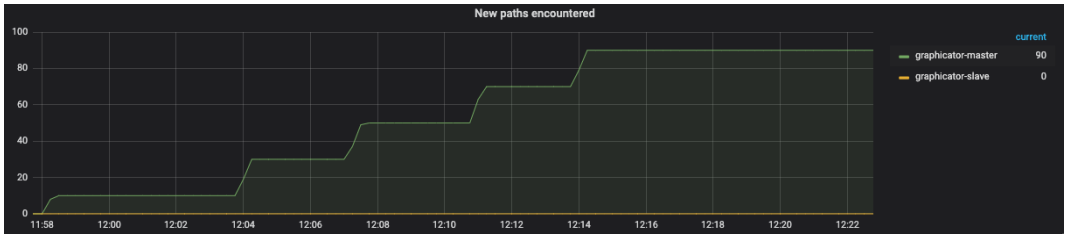


Figure A.12: New paths encountered showing in grafana with one slave

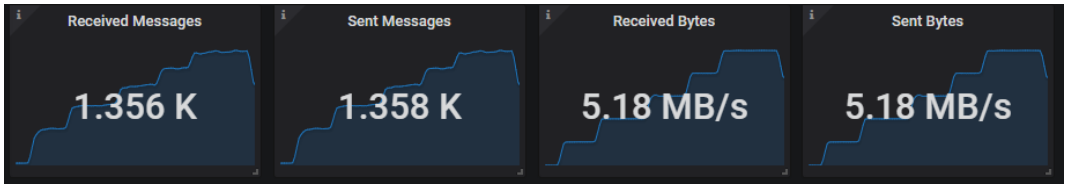


Figure A.13: STAN cluster network metrics with 600k lines per second

Figure A.14: CPU core usage time with two slaves

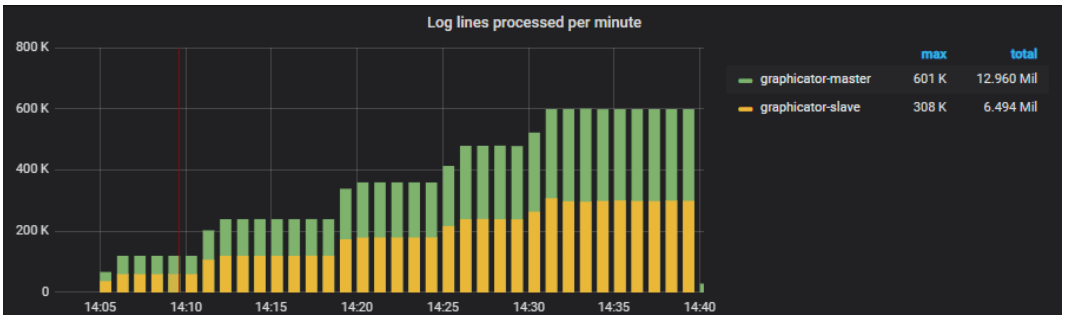


Figure A.15: Log lines processed per minute with two slaves

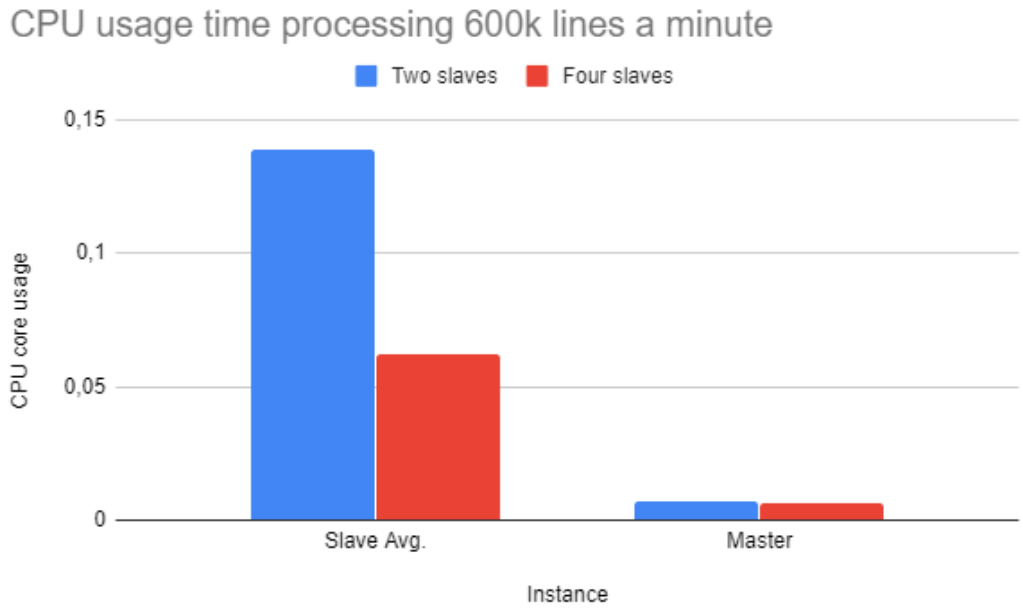


Figure A.16: Log lines processed per minute with two slaves

Figure A.17: CPU core usage time with four slaves

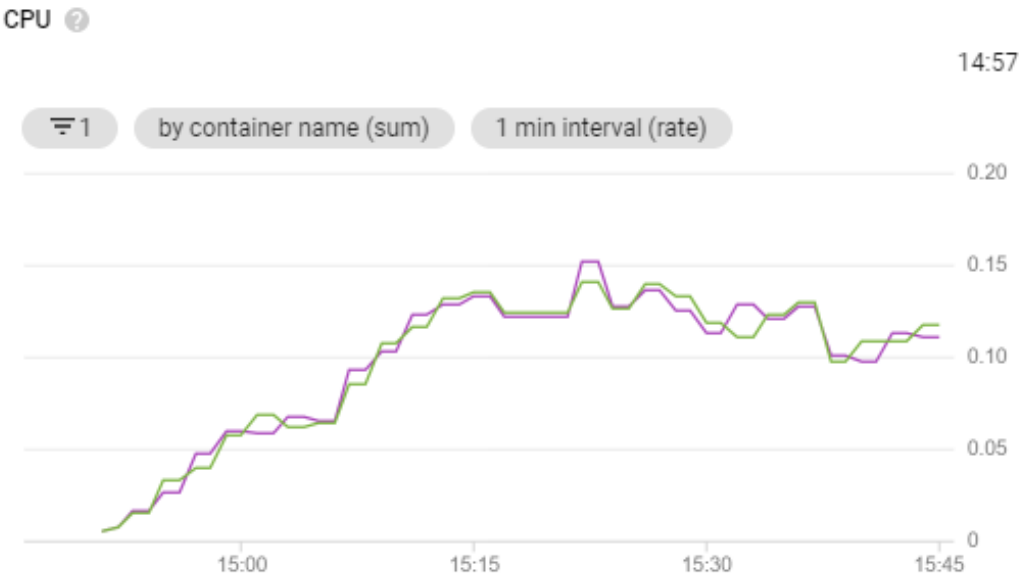


Figure A.18: Google cloud provided CPU chart of one slave

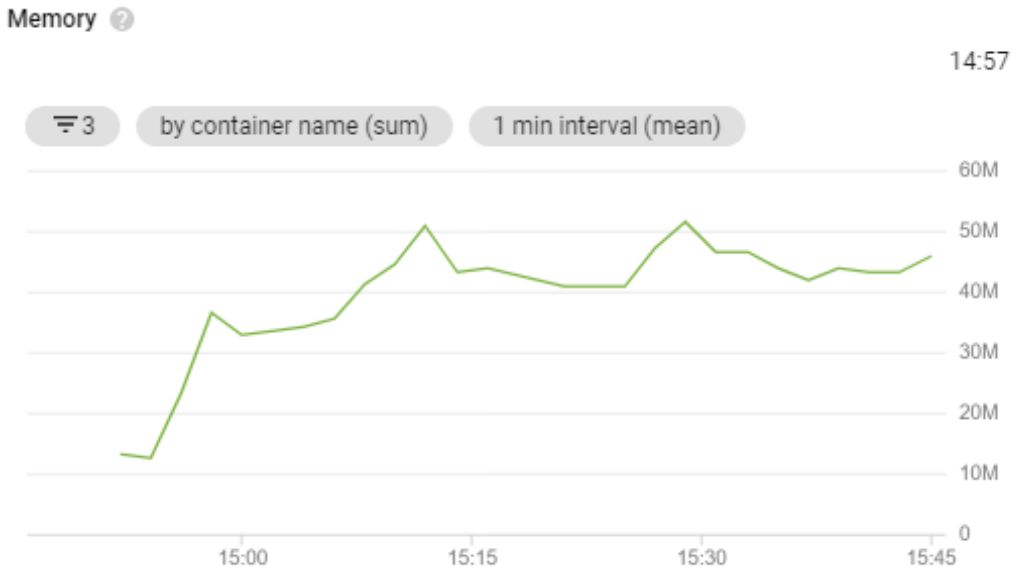


Figure A.19: Google cloud provided RAM chart of one slave

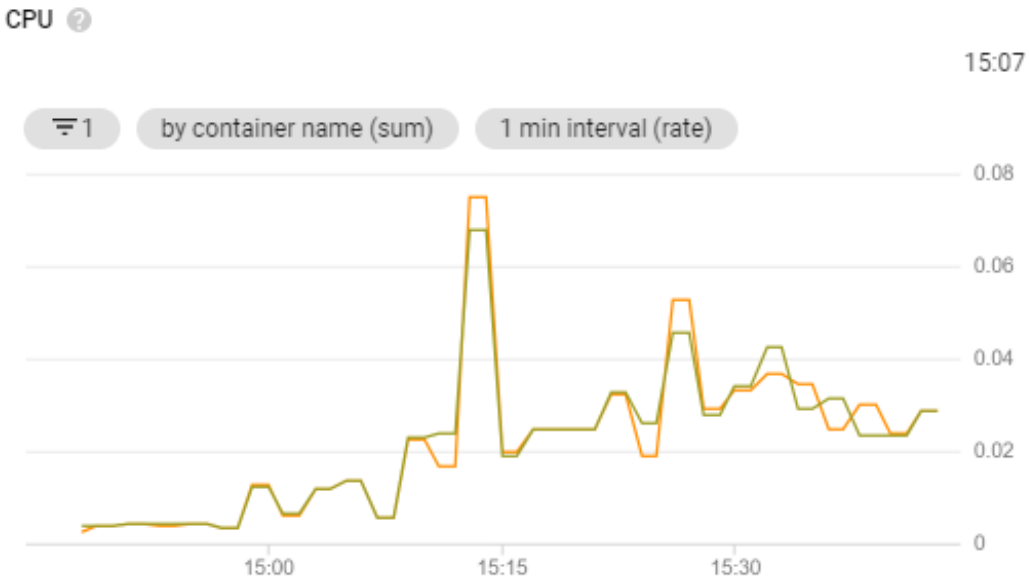


Figure A.20: Google cloud provided CPU chart of master

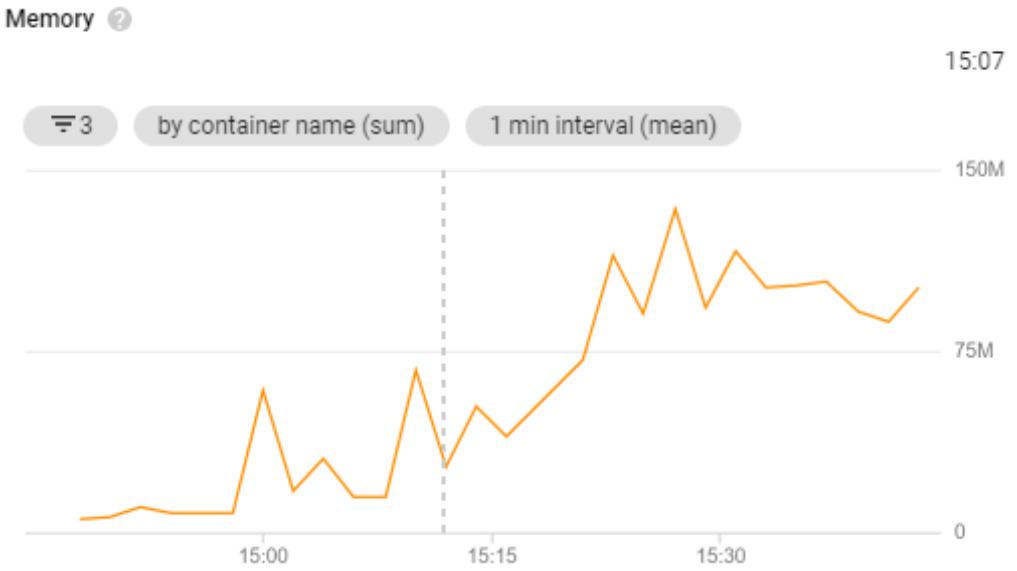


Figure A.21: Google cloud provided RAM chart of master

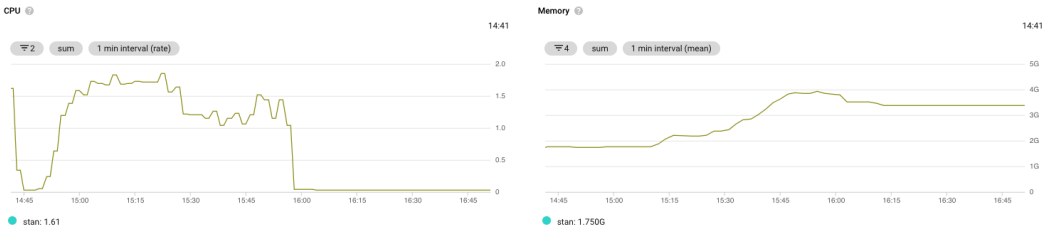


Figure A.22: Google cloud provided CPU and RAM chart of STAN

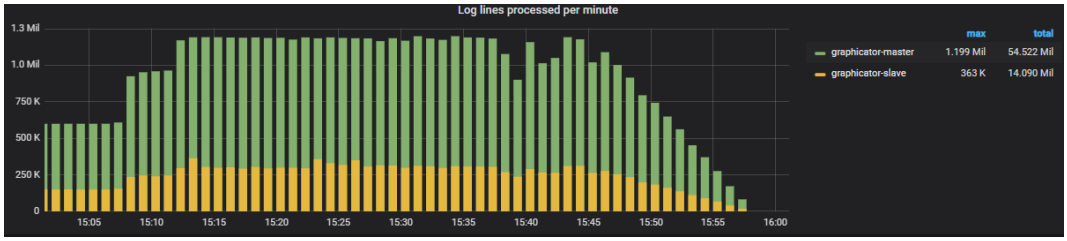


Figure A.23: Throttling of STAN

Figure A.24: STAN network throughput with 1.2 million log lines per minute

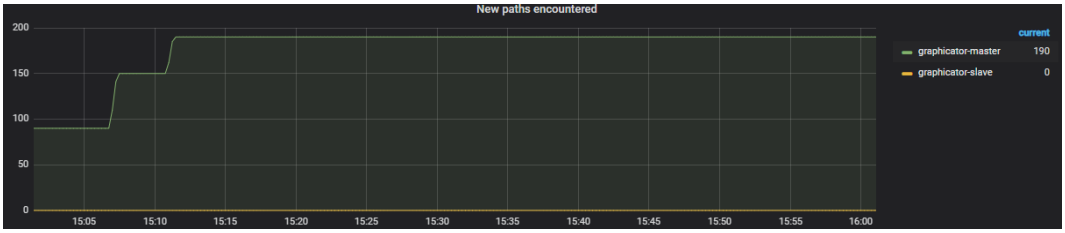


Figure A.25: New path reporting with 1.2 million log lines per minute