Implementation of an automatic log analysis tool for health care applications

Arttu Turunen

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 1.12.2023

Supervisor

Prof. Joni Pajarinen

Advisor

M.Sc. (Tech.) Magnus Sandberg

Aalto University
School of Electrical
Engineering

Aalto University
School of Electrical
Engineering

Aalto University, P.O. BOX 11000, 00076 AALTO
www.aalto.fi
Abstract of the master's thesis

Author Arttu Turunen

Title Implementation of an automatic log analysis tool for health care applications

Degree programme Master's Programme in Automation and Electrical Engineering

Major Control, Robotics and Autonomous Systems **Code of major** ELEC3025

Supervisor Prof. Joni Pajarinen

Advisor M.Sc. (Tech.) Magnus Sandberg

Date 1.12.2023 **Number of pages** 61+2 **Language** English

Abstract

Large software applications generate extensive amounts of log data to monitor behavior and diagnose software issues. While log data can be examined manually, the large volume of log data makes manual examination error-prone and time-consuming. Log data analysis software and methods have been developed to automate log data analysis, thereby enhancing software reliability and development. Commercial and open-source log analysis software exists, but this cannot always be used due to various restrictions, such as data security. In this case, an automatic log data analysis program has to be implemented from scratch.

In this thesis, we designed and implemented an automatic log analysis program for the healthcare software Effector by seeking inspiration from studies and existing implementations. We tailored the implementation to the requirements of the Effector and implemented anomaly detection using statistical methods based on Z-score and sliding window. We tested the implementation against real data from Effector in real use cases and detected genuine anomalies in the production data.

During the analysis of six months of log data, the log analysis program detected between 10 and 40 anomalies per week, depending on the size of the source data examined. For instance, the log analysis program detected a critical bug in the automatic import of patient data after an Effector version update. This bug would have gone unnoticed for a longer period without the assistance of the program.

Keywords logging, log data, log analysis, anomaly detection, outlier

Aalto University
School of Electrical Engineering

Aalto-yliopisto, PL 11000, 00076 AALTO
www.aalto.fi
Diplomityön tiivistelmä

Tekijä Arttu Turunen

Työn nimi Automaattisen lokidatan analysointiohjelman toteutus terveydenhuollon ohjelmistolle

Koulutusohjelma Master's Programme in Automation and Electrical Engineering

Pääaine Control, Robotics and Autonomous Systems **Pääaineen koodi** ELEC3025

Työn valvoja Prof. Joni Pajarinen

Työn ohjaaja DI. Magnus Sandberg

Päivämäärä 1.12.2023 **Sivumäärä** 61+2 **Kieli** Englanti

Tiivistelmä

Suuret ohjelmistot tuottavat suuren määrän lokidataa, jonka avulla voidaan seurata ohjelmiston käyttäytymistä ja ratkaista ohjelmiston ongelmatilanteita. Lokidataa voidaan tutkia manuaalisesti, mutta lokidatan suuri määrä tekee lokidatan manuaalisesta tutkimisesta virhealtista ja aikaa vievää. Lokidatan analysointiohjelmia ja -menetelmiä on kehitetty automatisoimaan lokianalyysia, mikä parantaa ohjelmistojen luotettavuutta ja helpottaa ohjelmistokehitystä. Kaupallisia ja avoimeen lähdekoodiin perustuvia lokidatan analysointiohjelmia on olemassa, mutta erinäisistä rajoituksista, kuten tietoturvan takia, näitä ei voida aina käyttää. Tässä tapauksessa, automaattinen lokidatan analysointiohjelma on toteutettava alusta asti itse.

Tässä opinnäytetyössä suunniteltiin ja toteutettiin automaattinen lokidatan analysointiohjelma terveydenhuollon ohjelmistolle, Effectorille. Inspiraatiota haettiin tutkimuksista ja olemassa olevista toteutuksista, mutta toteutus räätälöitiin Effectorin vaatimusten ja rajoitusten mukaan. Poikkeavien lokien eli anomalioiden havaitseminen toteutettiin käyttämällä tilastollisia menetelmiä, jotka perustuvat Z-arvoon ja liukuvaan ikkunaan perustuviin menetelmiin. Toteutettua ohjelmistoa testattiin aidolla Effectorin lokidatalla todellisilla käyttötapauksilla. Demotuista käyttötapauksista saadut tulokset vaikuttivat lupaavilta ja tuotantodatasta pystyttiin havaitsemaan aitoja poikkeamia.

Kuuden kuukauden lokidatan analysoinnin aikana havaittiin noin 10-40 anomaliaa viikossa riippuen tutkitun lähdedatan koosta. Analysointiohjelmalla havaittiin esimerkiksi kriittinen virhe potilasdatan automaattisessa tuonnissa Effectorin versiopäivityksen jälkeen. Tämä virhe olisi jäänyt huomaamatta pidemmäksi aikaa ilman ohjelman apua.

Avainsanat lokitus, lokidata, lokianalyysi, poikkeaman havaitseminen, poikkeava data

Preface

First, I want to thank my instructor, Magnus Sandberg, for suggesting the topic of my thesis and for providing valuable advice regarding its implementation. Thanks also go to my supervisor, Prof. Joni Pajarinen, from whom I received good guidance on thesis writing and feedback. Most importantly, I want to thank my partner for supporting me through this process. Finally, I would like to thank my colleagues from whom I received peer support for this work. They were also working on their thesis at the same time, so we were able to enjoy the writing process of the thesis together.

Otaniemi, 1.12.2023

Arttu Turunen

Contents

Symbols and abbreviations

Abbreviations

IM	Invariant Mining
PCA	Practical Component Analysis
SVM	Support Vector Machine
LR	Logistic Regression
IQR	Interquartile Range
IIS	Internet Information Services
ELK	Elasticsearch, Logstash, and Kibana
API	Application Programming Interface
UI	User Interface
ADO	ActiveX Data Objects
OOP	Object-oriented programming
SD	Standard Deviation
HL7	Health Level Seven

1 Introduction

Large software applications generate vast amounts of log data about software events. As stated in the study by Zhu. et all [1], events describe the behavior of the software in different situations, allowing log data to be used to help diagnose software failures and other anomalous behavior of the software. However, due to the vast amount of log data, manual investigation of log data is tedious and error-prone.

Many commercial and open-source log data analysis and visualization tools, such as Splunk [2], Elastic Stack [3] and LogRhythm [4] have been developed to automate the analysis of log data. In addition to analysis, log analysis tools usually provide functionalities for accessing different types of data from different sources, archiving and searching the data, notifying users about errors and anomalies, showing dashboards with informative graphs, and so on.

1.1 Motivation

Although working solutions already exist, there can be situations when they cannot be used. One of the most important factors to consider when choosing or implementing a data analysis tool is data privacy. Many software applications, such as health care applications, use confidential data that must not be stored outside of the client server. In some cases, the use of third-party software is entirely prohibited even if the data remains on the client server which completely prevents the use of commercial data analysis tools. This also applies to Effector [5], a Finnish public health process and information management software.

According to a study by Zhang. et al. [6], many analysis methods assume that the log data is sequential which means that the log event is a part of a longer event chain. Event chains are usually used to form count vectors that serve as input to machine learning algorithms that classify the vector as a normal or anomalous event. However, in some real-life applications such as Effector, the log data is not sequential but rather a set of individual events that describe, for example, the occurrence of a particular error. In this case, each data point describes an anomaly that must be taken into account in the anomaly detection model.

In addition, Zhang. et al. [6] explains that log analysis methods usually use log message parsers that parse automatically the unstructured log message into a structured log message so that the data can be used in log analysis methods. However, the log message format is free and entirely up to the developer, which makes parsing log messages challenging and error-prone. In some software, such as Effector, the log data is already in structured form in which case there is no need for a log parser. Log data can be stored for example in an SQL table consisting of multiple defined columns which classifies the log data naturally. The SQL table may contain a column to store the information about the name of the function from which the error was raised or to which subgroup the log event belongs. Furthermore, the log data can be stored in many different log tables which naturally classifies the data. Utilizing the pre-defined log structure, a tool can be developed that does not rely on a log parser.

1.2 Scope

In this paper, we study and discuss the characteristics, possibilities, and limitations of automatic log analysis tools and related methods. This information is utilized for the implementation of an automatic analysis tool for the real-life healthcare application named Effector, and the structure, functionality, and results of this implemented tool are subsequently presented. We will test and evaluate the functionality of the tool by using real-life data. The data cannot be stored outside of the client server, so we present a solution for this problem as well.

In a more precise manner, the requirements and goals of the implementation part of the thesis can be presented as the following list:

1. The implemented tool must be able to retrieve log data from multiple client-installed Effector environments simultaneously as well as retrieving the data from multiple log data sources inside the installed environment. In other words, the implementation must compile the fragmented log data into a single, more manageable format automatically.

2. The implemented tool must be able to visualize the log data in a human-readable format to make manual log data analysis easier. The implementation must also retrieve the desired log events for a given period and report basic statistics on the retrieved data, such as occurrence, arithmetic mean, and the most frequent log events.

3. The implemented tool must provide automatic alerts to the user in case of an anomalistic log data occurrence. Anomalistic, unusual occurrences of log data can be, for example, the appearance of a new error log type, a large change in the number of logs, or frequency changes in the occurring log data.

4. Log data must not be stored outside the customer's environment so all the processing and analysis of log data must be done in random access memory.

1.3 Research question

The practical part of the thesis deals with the implementation of the log analysis tool, but the overall purpose of the thesis, and the ultimate goal, is to answer the following research question:

RQ: How to improve log data analysis in Effector?

With this research question, the thesis aims to simplify and enhance log analysis in Effector. To address the research question, it is essential to first clarify the current state of logging and log analysis in Effector, identifying its limitations and shortcomings to enable a feasible implementation. Moreover, this research question is relevant to other similar software that gathers extensive data from various sources, potentially influencing the performance and reliability of not only Effector but also other software systems.

1.4 Structure of the thesis

The thesis can be divided into two parts, the theoretical part and the implementation part. The theoretical part starts with Chapter 2, which provides the basis for answering the research question by introducing the concepts of log analysis. The chapter introduces existing log analysis software and examines its features. Chapter 2 also introduces the reader to Effector, for which a log analysis tool will be developed.

The implementation part of the thesis starts with Chapter 3, which presents and justifies the design decisions of the implemented software. The more detailed technical implementation and structure of the software are also presented in this chapter. The implemented software is evaluated in Chapter 4 through the analysis of software results and the examination of real-life demo cases.

The conclusion of the whole thesis is presented in Chapter 5. The results of the thesis will be summarised and used to address the research question.

2 Background

In this chapter, we present the background and the most common concepts of log analysis. Through an extensive exploration of earlier studies, the chapter explains what kind of data is generally used, how the data is modified before analysis, and which analysis methods are most commonly used in log analysis applications. We briefly present Effector, along with its web application Palse. We provide and discuss the current log data analysis methods and log data sources in Effector. Additionally, we study common design patterns of visual log analysis tools, identify problems and limitations of current log analysis tools, and briefly present available commercial and open-source log analysis products in this chapter.

2.1 Concepts of log analysis

According to studies by El-Masri. et all. [7], and Ryciak. et all. [8], log data is commonly text-based information that records specific events and run-time information of a program. Log data is saved to the log files or other log databases by **logging statement** which is a line of code written by a software developer. Software developer also defines what information is included in the logging statement so the choices of the developer have a major role in the quality of the log data.

Studies [7, 8], also explain that log data consists of log entries. **Log entry** is one row in a log file and represents a specific event of a program. Commonly, log entries consist of header and message parts. The structure of a **log header** depends significantly on the used logging framework but commonly header includes common run-time information of the log event such as a timestamp, an identification number, a log level, a source function, etc.

The header part of the log entry is followed by a **log message**. Studies [6, 7], define a log message as a free-form text part of the entry which does not follow a strict syntax. Commonly, the message part describes the event in a more detailed manner. The log message consists of constant and variable parts. The variable part of the log message is assigned at the run-time so it can get different values for different runs and the constant part is determined already in compiling so it always gets the same value. An example of a log entry and its components is shown in Figure 1.

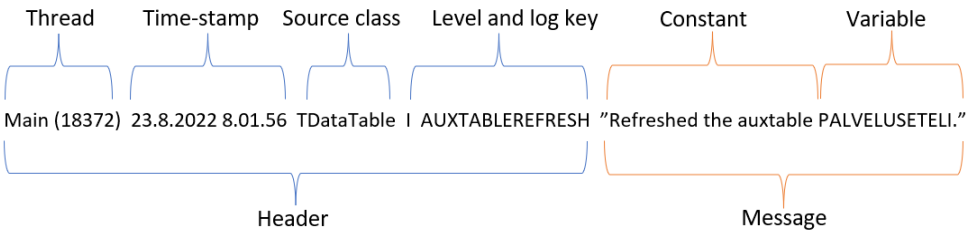

Figure 1: Example syntax of the log entry.

It should be noted that although logging commonly follows previously presented syntax, no universally adopted standard is used in every software. Instead, the choice and determination of log data syntax depend on individual developers and their software-specific requirements. Consequently, the absence of a standardized syntax renders a single log-analyzing tool inadequate for all scenarios, often necessitating the development of custom tools.

2.1.1 Data anomalies

A study by Tellis. et al. [9] define that anomalies are patterns within data that deviate from the expected normal behavior. They represent instances where the data does not fit into a well-defined pattern. Unexpected behavior is often categorized as anomalies as well. An anomaly can also be referred to as an outlier, indicating a data point or observation that significantly differs from the majority of the data.

Studies by Gogoi. et al. [10] and Lindemann. et al. [11] categorizes anomalies into three distinct groups, taking into account their characteristics, contextual relevance, behavioral patterns, and cardinality.

1. **Point anomalies**

   A point anomaly refers to a specific data point that stands out significantly from the remaining data points, as measured for example by its deviation from the average value or a normal distribution of the data. In other words, point anomaly detection focuses on stochastic methods for detecting outliers.

2. **Collective anomalies**

   A collective anomaly refers to a set of data points that are not categorized as anomalies alone but that are anomalous when inspecting the whole set of data. Therefore, the extent of deviation is influenced by the inherent structure present within the data sequence.

3. **Contextual anomalies**

   A contextual anomaly refers to a set of data points that exhibit similar magnitudes to the normal data. However, within the context of the surrounding data, these data points indicate an anomaly or deviation from the expected pattern.

Different anomaly groups are presented in Figure 2.

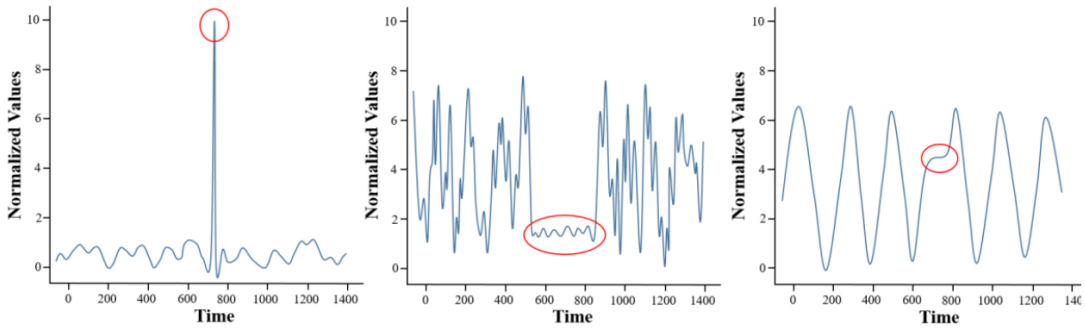

Figure 2: Example data series of different anomaly groups. Left: Point anomaly, Middle: Collective anomaly, Right: Contextual anomaly. [11]

2.1.2 Common anomaly detection approach for log data

In the context of log data, anomalies typically indicate events within a system that deviate from the expected or normal execution. Therefore, anomalies represent the noteworthy aspect of log data that can be valuable for debugging and troubleshooting purposes. The detection of anomalies in log data can be achieved through detection models. However, before the utilization of these models, prepossessing of the raw log entries is necessary to enable effective anomaly detection.

Numerous studies [12, 13, 14], have been dedicated to anomaly detection in log data, and these studies commonly exhibit the following shared characteristics in their implementation. In simplified form, the structure of the common log anomaly detection procedure follows the approach that is presented in Figure 3 and discussed more precisely in the following subchapters.

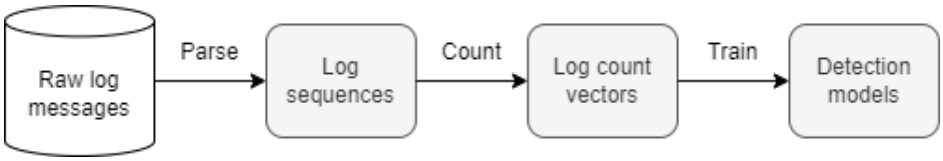

Figure 3: Data flow of a common log analysis approach.

The procedure consists of three tasks:

1. **Parsing the log data into log sequences**

   According to a study by He. et al. [12], in automated log data anomaly detection, log parsing typically serves as the first task in the process. Parsing involves the transformation of unstructured raw log entries into a structured sequence of events which only contains the information that categorizes the log entry at some level. In essence, the objective of log parsing is to eliminate the variable part of the log message, thereby allowing the constant part to

categorize the log entry. Depending on the use case, this may or may not include the header section as well.

For example, the message part of the data entry represented in Figure 1, would be parsed by replacing the variable part 'PALVELUSETELI' with some benchmark such as an asterisk (*) symbol. This procedure forms a unique key value for identifying different log events.

"Refreshed the auxtable PALVELUSETELI " (Raw log message)
"Refreshed the auxtable * " (Log message parsed to log event)

2. **Forming log count vectors**

After parsing, log events are typically transformed into count vectors or, as mentioned by He. et al. [12], matrices, to make the data more readable for detection models.

Studies [6, 13] discuss the formation of log count vectors. The header section of a log entry typically includes information that facilitates the grouping of multiple log entries within the same session, such as a session ID field. These log entries typically appear consecutively, forming a log sequence where each entry shares a common session ID. By grouping the log entries based on their log events, the occurrences of each log event within a log sequence can be determined and thus count vector can be formed. The length of the log vector corresponds to the number of different log events which have already occurred. Figure 4 represents an example of a log sequence that is transformed into a corresponding log count vector.

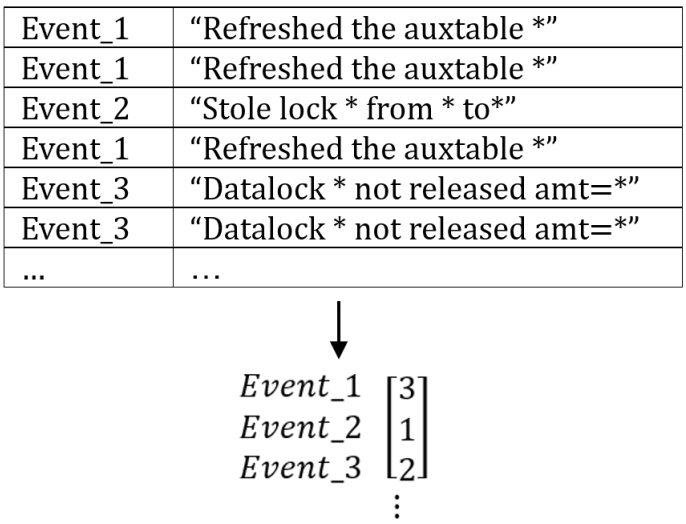

Figure 4: Transforming log sequence into log count vector.

3. **Training detection models**

   The concluding part of the traditional log analysis procedure involves training the detection model using log count vectors. Previous studies, gathered together by Zhang. et al. [6] have employed various techniques, such as Invariant Mining (IM), Principal Component Analysis (PCA), Support Vector Machine (SVM), and Logistic Regression (LR), in their implementations for detecting anomalies within log data.

2.1.3 Statistical methods to detect anomalies

The previous example of common anomaly detection assumes that the log data is sequential, resulting in multidimensional data vectors. This necessitates the utilization of more advanced machine learning models that are appropriate for analyzing multidimensional data. However, in many scenarios, it is more suitable to represent the log data as event occurrences per day, transforming the data into a one-dimensional time series. An illustrative example is the error log where each error occurrence is an exception. In such cases, the statistics provide useful information about the nature of the error.

- **Boxplot.** Tukey's method, introduced in a study by Tukey [15] in 1977, is a well-known and straightforward graphical technique used to present essential details about continuous one-dimensional data. It involves the construction of a boxplot, which visually summarizes key statistics such as the lower extreme (Q_0), lower quartile (Q_1), median (Q_2), upper quartile (Q_3) and upper extreme (Q_4) of a data set. By using the lower and upper quartile, the IQR (Interquartile Range) of the data set can be calculated by taking a distance between Q_1 and Q_3. Therefore, inner (Eq. 1) and outer (Eq. 2) fences can be defined as follows:

$$[Q_1 - 1.5IQR, Q_3 + 1.5IQR] \tag{1}$$

$$[Q_1 - 3IQR, Q_3 + 3IQR] \tag{2}$$

  A value that falls between the inner and outer fences can be considered a potential anomaly, while a value beyond the outer fences is more likely to be a probable anomaly. It is important to note that the choice of using 1.5 and 3 times IQR to define the fences, as employed by Turkey's method [15], does not have a solid statistical foundation. Therefore, the choice of these thresholds is somewhat subjective and can vary depending on the context and characteristics of the data.

- **K-nearest neighbour.** The study by Hodge. et al. [16], suggest that the underlying principle in the proximity-based methods is based on the assumption that observations that are similar to each other tend to be located close together, while outliers, which deviate significantly from the majority, tend to be isolated and situated at a greater distance from the cluster of similar observations.

Hodge. et all. [16], presents that in proximity-based statistical methods, data points are classified according to their distances to other data points. Since the distance is calculated for each record, the computational complexity of the method increases exponentially with the number of records. Proximity-based methods are also directly proportional to the dimensionality of the data. Therefore, for example, the proximity-based **k-nearest neighbor** algorithm will have a time complexity of $O(n^2m)$, where n equals the number of records and m equals the dimensionality of the data. The complexity is also affected by the way the distance to the points is calculated. For example, Mahalanobis distance presented in Eq. 3 is more computationally expensive than Euclidean distance presented in Eq. 4 since it requires the computation of correlation matrix $\mathbf{C}$ and mean μ of the whole data set.

$$\sqrt{(\mathbf{x}-\mu)^T\mathbf{C}^{-1}(\mathbf{x}-\mu)} \tag{3}$$

$$\sqrt{\sum_{i=1}^{n}(x_i-y_i)^2} \tag{4}$$

Since traditional proximity-based methods are computationally demanding, improved proximity-based algorithms have been developed. Ramaswamy et al. [17] introduces an **optimised k-nearest neighbor** algorithm where distance is calculated to m^{th} neighbors rather than all neighbors, where m is a parameter specified by the user. Ramaswamy et al. [17], achieved even greater algorithmic speed improvements by subdividing the data set into smaller cells.

- **Z-score**. In statistics, according to a study by Sehar. et al. [18], the standard score, also known as the Z-score, is a measure that quantifies how many standard deviations a value is away from the mean of the data set. If the Z-score for a particular value surpasses a pre-defined threshold, the value is categorized as an anomaly. In cases where the data set follows the standard normal distribution with a mean of 0 and a variance of 1, the threshold for identifying anomalies is often set at 3. Z-score for the value x_i can be calculated as presented in Eq. 5, where μ and σ are the mean and standard deviation of the data set respectively.

$$Z_i = \frac{x_i-\mu}{\sigma} \tag{5}$$

The Z-score method, which relies on the mean and standard deviation of the entire data set, can be sensitive to extreme values. As a result, more robust and advanced algorithms have been developed to address this limitation and provide improved outlier detection techniques.

The smoothed Z-score algorithm, invented by a Brakel [19], is a robust peak detection method that was initially developed to identify sudden peaks in

real-time time series. Its primary objective is to detect and locate peaks while maintaining robustness in the presence of noise and variations. The algorithm achieves this by employing a separate moving mean and deviation calculation, which ensures that previous signals do not interfere with the determination of the signaling threshold for future signals.

According to Brakel [19] and a study by Perkins. et all. [20], the smoothed Z-score algorithm takes three parameters. The lag (L), which defines the size of the moving window, the threshold (T), which defines the minimum Z-score of the data point which is classified as an anomaly, and influence (α), ranging from 0 to 1, which defines the amount of how much the output of the algorithm effect the moving mean and standard deviation.

The equations of the smoothed Z-score algorithm are presented in Equations 6 to 10, where x_i represent the input signal, s_i represents the input signal adjusted with the influence and y_i represent the output signal where 1 identify an anomaly and 0 identify a normal data.

$$s_i = \alpha x_i + (1 - \alpha) s_{i-1} \tag{6}$$

$$\bar{s_i} = \frac{1}{L} \sum_{i}^{i+L} s_i \tag{7}$$

$$\sigma_{s_i} = \sqrt{\frac{\sum_i^{i+L} (s_i - \bar{s_i})^2}{L - 1}} \tag{8}$$

$$z_i = \frac{x_i - \bar{s}_{i-1}}{\sigma_{s_{i-1}}} \tag{9}$$

$$y_i = \begin{cases} 1 & if |z_i| \geq T \\ 0 & if |z_i| < T \end{cases} \tag{10}$$

2.1.4 Challenges of anomaly detection

The field of anomaly detection presents several challenges that have been widely researched in the past. The challenges of anomaly detection are varied and highly dependent on various factors, including the chosen anomaly detection method, the nature of the data being analyzed, the speed and size requirements of the application, and the specific type of anomalies being searched for.

One of the key challenges of anomaly detection is the log instability of the data. Log instability refers to the change in log data over time, and according to Zhang et al. [6] it has two sources. The first source is the evolution of logging statements which occurs when developers frequently modify source code, including logging statements, leading to changes in the logged data. This poses a specific challenge for applications that are continuously evolving through agile development methods,

such as Effector. Consequently, the need for regular retraining of anomaly detectors based on supervised machine learning methods further complicates their practical utilization as effective methods. A study by Karanjit et al. [21] also mentions the fact that the limited availability of training and validation data presents a challenge when attempting to utilize methods that rely on training.

The second source for log instability, according to Zhang et al. [6], is processing noise in log data. Processing noise occurs when data is collected, retrieved, and processed according to the use case of the application. For example, in Effector the log data is stored in a distributed manner on servers of many different organizations, geologically located in different places. The log data collected can be affected by variations in the internal clocks of different servers, network errors, as well as occasional server outages. Log parsing introduces another significant source of noise in log data analysis. Typically, during the initial stages of log data analysis, a log parser is employed to extract the log events from the raw log messages as stated in the previous section 2.1.2. According to Zhang et al. [6], modern log parsers lack the required level of accuracy thus increasing the log instability.

Each type of anomaly detection method has its advantages and disadvantages, as discussed in a study by Zhang [22]. Proximity-based methods, like the K-nearest neighbor approach, are non-parametric and do not depend on any assumed data distribution. This makes them versatile and straightforward to implement. However, proximity-based methods face difficulties in high-dimensional spaces due to the curse of dimensionality, which diminishes their effectiveness, especially in real-time applications.

According to Zhang [22], statistical methods are usually mathematically justified ensuring their reliability and robustness as analytical approaches. With a probabilistic model, anomaly detection becomes efficient and allows for interpreting anomalies. As the constructed model is concise, there is no requirement to store extensive original data sets for outlier detection. However, statistical methods are limited in their applicability to multivariate data sets due to the uni-variate nature of the distribution models they rely on. In many cases, the statistical distribution of the data is either unknown or the existing statistical distribution does not capture the entirety of the data set. As a result, the method may not yield satisfactory results or perform optimally in such cases.

In conclusion, when selecting an appropriate anomaly detection method, it is crucial to have a comprehensive understanding of the nature of the data and to be aware of the strengths and limitations of each method. The data can be examined using statistical techniques or through visual examination, such as analyzing histograms. The selection of the appropriate method for this project is discussed in greater detail in Section 3.1.

2.2 The Effector Software

According to the home page of Polycon Oy [5], Effector is a process and information management desktop application for the public health sector used in 22 different hospital districts in Finland. In addition to health sectors, the Effector is also used by NRC Group as a warehouse management tool. The Effector product family also includes a web application Palse which primarily acts as a service voucher management tool and also provides an interface for service providers and customers to access the Effector system. In addition to service management, Effector is used to manage processes related to assistive devices, equipment maintenance, and medical supply distribution, among others.

Effector has been developed by Polycon since 1997 and continues to be developed today. Effector is primarily developed in the Delphi framework. Delphi is defined in a book by Lischner [23] as an object-oriented cross-platform development tool extended on Object Pascal programming language. The technical implementation of this study is also done in Delphi, as Effector is implemented with it.

2.2.1 Logging in Effector

The extensive use of Effector on the desktop and web side, and its long development life have had a significant impact on how the software collects log data. Log data in Effector is stored in multiple SQL tables on various client servers, which automatically classify the log data. However, this classification process makes it difficult and time-consuming to utilize the log data for debugging purposes. There are also differences in the structure of SQL tables used in logging which makes the format of log data inconsistent.

2.2.2 Log structure in Effector

Like all data in Effector, all the log data is stored in the Microsoft SQL Server database. An SQL database consists of tables with rows and columns. In the context of logging, each row corresponds to a single log event, and each column corresponds to an attribute of the log event. The most common log tables in Effector are presented next.

A. DebugLog

The most common log table in Effector is 'DebugLog', which serves as a generic log table for data that developers find useful for debugging. It is a common practice to add an entry to this table when an event is encountered in the code that should not occur during normal execution. Additionally, the 'Debuglog' table may also be used to store informative data about normal events that are essential to the execution. The type of the log entry in the 'DebugLog' table is determined by a 'Level' column that contains an enumeration. The level enumeration consists of four values:

1. ltError, an error has occurred that requires interrupting the operation.
2. ltWarning, an error has occurred but the operation can continue.
3. ltInformation, an important event from which information is desired.
4. ltDebug, same as ltInformation but only used when debug-switch is activated.

The 'DebugLog' table has also columns for 'Sender', 'Log key', 'Log message', and 'Duration' which can be set manually when calling a logging statement that creates a log entry to the database. The following code example of a logging statement uses a 'DebugLog' function that creates a new log entry to the 'DebugLog' table if the user does not have a producer role.

Listing 1: Example of a logging statement in Effector.

```
if not TryUserAsProducer(AUser, AProdUser) then
begin
    DebugLog(Self, ltError, 'USERROLE',
        'User ' + AUser.Id + ' is not a producer.');
    Exit;
end;
```

The first parameter of the 'DebugLog' function is a reference to the sender class. In this case, the sender refers to 'Self', the current class, where the error has been raised. The second parameter is the type of the log event. In this scenario, the type is labeled as 'ltError' due to the failure to check if the user is a producer, causing the operation to be interrupted. Thus the 'Exit' command after the logging statement. The third parameter represents the log key of the log entry, serving the purpose of identifying the specific sub-system of the software. In this example, the key is represented by a constant string 'USERROLE'. The last parameter is the log message, which is defined by the programmer and consists of a constant and variable part. In this example, the log message parameter is defined as a string ('User ' + AUser.Id + ' is not a producer.') where the part 'AUser.Id' is the variable part of the log message and the rest of the string is the constant part of the log message. In addition, the logging statement automatically records the logon number, thread identifier, date and time of the entry, call stack, and other metadata, in the log table.

B. PS_DebugLog

The web application Palse operates on a server separate from Effector installations. Due to its web-like characteristics and different requirements, Palse requires an independent table to store its log data. Palse stores its generic log entries, which are generated from logging statements, in the 'PS_DebugLog' table. This table

corresponds to the 'DebugLog' table on the Effector side with some exceptions. The 'PS_DebugLog' table includes additional columns that store web-related information, such as the session ID, page URL, and user's IP address. Despite the 'PS_DebugLog' table being located in a distinct environment and having different fields compared to the 'DebugLog' table, log entries are added to it using similar logging statements as those used for 'DebugLog'.

C. ErrorLog

The third major log storage in Effector is the 'ErrorLog' table. This table serves as a storage specifically for run-time errors that are automatically generated by the system whenever an exception occurs during code execution. Exceptions allow an application to resolve the error without terminating. Delphi categorizes run-time errors into three types as mentioned by Embarcadero Technologies [24]:

1. I/O errors, numbered 100 through 149
2. Fatal errors, numbered 200 through 255
3. Operating system errors

The reason for separating run-time errors from normal logging through logging statements is to distinguish between automatically generated system errors and logging statements that are manually defined by the programmer. This separation helps to classify the log data, making debugging easier. The 'ErrorLog' table contains the same columns as the 'DebugLog' table except for the 'Level' column. This exclusion results from the fact that each log entry in the 'ErrorLog' table can be classified as an error, so there is no need for a column that classifies the level of the log entry.

D. Other log tables

Alongside the previously mentioned tables, several other tables are defined in Effector that contain usage-specific log data. Although these tables will not be utilized in this work, they are still worth mentioning, as they highlight the extensive collection of log data from numerous sources. For example, the 'AccessLog' table contains log data about users accessing sensitive patient-specific information. It offers a valuable approach to maintaining a detailed record, documenting which information has been accessed, by whom, and at what time.

Another distinctive log table in Effector is the 'EventLoki', designed to capture specific events within the system. These events primarily originate from procedures that handle multiple files or data rows, such as importing and exporting invoice files through API. This log table includes additional columns that provide details about the number of handled files and the number of files that were processed successfully or encountered errors. This illustrates a notable example of how the logging format varies depending on the application. By utilizing custom columns for processed file

count, file count information can be excluded from the log message. This approach ensures that the most crucial information is readily accessible and presented in a clear format, significantly aiding with debugging. On top of that, Effector includes separate log tables like login information, email log, and IIS log (Internet Information Services).

2.2.3 The current status of log analysis in Effector

The current status of log analysis in Effector primarily relies on manual inspection of the log data. This can be accomplished by using the built-in log data search windows within Effector or by utilizing custom-made software called SQL tool.

The core functionality of Effector revolves around data storage, editing, and display, making data search windows a central aspect of its operation. These search windows are also utilized for multiple log data tables so that log data can be searched with different filters and displayed in tabular format. In this format, each row represents a log entry, while each column corresponds to a specific attribute of the entry. This allows the log data to be examined, for example, at a point in time when a problem has been identified through customer feedback.

However, a limitation arises with this approach as the log data is only reviewed once the user detects a problem, which can prolong the process of resolving the issue. Ideally, the problem should be identified immediately as soon as the anomalous log entry is stored in the database, enabling a more proactive and efficient problem-solving process. In addition, the search window lacks more detailed statistics about the log entry beyond the number of log entries. As previously described, log data in Effector is divided into multiple log tables, resulting in separate search windows for each table. Consequently, when searching for a desired log entry, the appropriate source table needs to be identified.

Another method for analyzing the log data in Effector is the SQL tool. SQL tool is an in-house application that enables connection to any SQL server and allows SQL tables stored on the server to be viewed and edited. This tool primarily serves for inspecting and tweaking production data during debugging. Additionally, it enables inspection of the log data as well. Much like the Effector search window, the SQL tool presents the displayed data in a tabular format, but with the distinction that the data is in its raw SQL format, allowing for a thorough examination of the server-stored information. The SQL tool enables the writing and executing of custom SQL queries for example to showcase more sophisticated statistics of the log data. However, this process involves writing raw SQL code, which may not be considered very user-friendly especially when a non-programmer is using the software.

As stated by Fu et al. [14], manual anomaly detection becomes considerably challenging and inefficient as applications grow in size and complexity. Initially, manually examining a large number of log messages generated by a large-scale system can be extremely time-consuming and labor-intensive. Furthermore, an individual developer or system administrator might not possess comprehensive knowledge of the entire system preventing the understanding of the log data.

A more sophisticated and automated anomaly tool is needed in Effector since

the size of the software is constantly growing as more functionality is added at the request of customers. In addition, internal turnover further complicates manual data examination since the knowledge of the log data may no longer be retained within the company.

2.2.4 Distribution to organisations

One significant aspect of the Effector that impacts the collection of log data is how the Effector is distributed to different organizations. Effector deployment follows a client-specific approach, distributing it across multiple servers, with each hospital district hosting its server for the Effector database.

Decentralization enhances database efficiency, security, and fault tolerance, as it avoids storing all data in a single database. For instance, in the event of one server experiencing downtime, the Effector will continue to function seamlessly for other clients. However, this results in log data being stored on separate servers, which poses a challenge when attempting to analyze log data in a standardized manner. In addition to this, log data is also distributed within the server to different log tables as described previously in section 2.2.2. Therefore, an implemented logging tool must possess the capability to aggregate log data both within individual servers and across multiple servers. The solution to this problem is presented more precise manner in Chapter 3.

2.3 Common design decisions of log analysis tools

The automation of log analysis is a widely researched topic, with many commercial and non-commercial tools being developed. In this section, common design choices made by both commercial and non-commercial log analysis tools are examined. By understanding these factors, it becomes possible to consider and integrate relevant aspects while developing a log analysis program for Effector.

2.3.1 Commercial and non-commercial products

A. Splunk

Splunk [2] is a flexible and widely used commercial software platform that consists of many products focusing on searching, monitoring, analyzing, and visualizing machine-generated data. According to a book by Carasso [25], Splunk enables data indexing, which involves collecting data from various sources scattered across different locations and aggregating it into centralized indexes. Indexing enables the efficient and rapid search of logs from all servers, significantly expediting the process of determining when the problem occurred. According to the home page of Splunk [2], Splunk uses machine learning and AI models to analyze log data and predict security threats making threat response faster. More detailed information on the used machine learning models was not available. Moreover, Splunk facilitates real-time analysis of data streams, operating with a millisecond delay. In addition to these features,

Splunk enables users to perform basic search queries on log data and create custom dashboards for visualization. Although Splunk is commercial software, Splunk offers free trials of products with limited functionality and use time.

B. Elastic Stack

The Elastic Stack, as described in a book by Chhajed [26], is a comprehensive log analysis platform built around three open-source tools: Elasticsearch, Logstash, and Kibana. The Elastic Stack is also referred to as ELK Stack. ELK Stack aims to provide a tool for the whole log data analysis process, from log data searching to log data visualization.

According to Chhajed [26], the first component in the ELK stack pipeline is Elasticsearch, a software designed to provide deep search and analytics for log data. It is an open-source search engine based on Apache Lucene, released under an Apache 2.0 license. Elasticsearch can dynamically index data without prior knowledge of the structure of the data. Indexing enables fast search responses in vast amounts of data. Logstash, the second component in ELK Stack, aims to collect log data from multiple locations, parse it, and forward the data to the desired location. Finally, the data is visualized using Kibana, a software capable of generating visual representations such as histograms, pie charts, line graphs, and more.

C. LogRhythm

LogRhythm, as described by the home page of LogRhythm Inc. [4], is a widely recognized commercial log management and log analysis tool used by prominent organizations, including the US Air Force and NASA. Like in ELK Stack, LogRhythm uses an Elasticsearch-based search engine as a back-end allowing contextual and unstructured search. LogRhythm focuses on information security through its detection of user-based threats, identification of network threats, and exposure of endpoint threats. These capabilities are achieved through the implementation of anomaly detection and network traffic behavior analytics.

According to a study by Jayathilake [27], LogRhythm also includes functionality for risk-based prioritization of log events, and the flexibility to customize rules and alerts. Additionally, it provides customizable visualization options, real-time log monitoring, file integrity monitoring, and more.

2.3.2 Common features

By investigating previous log analysis tools, we can detect common features. In all examples, the efficient implementation of search functionality played a key role, which is understandable as log data usually accumulates in large quantities. The efficient search functionality was based on indexing the log data. According to a study by Ammarr. et al. [28], one of the widely used indexing methods used in large disk-based databases is the B-tree algorithm invented and presented in a study in 1985 by Daniel Dominic Sleator [29] and Robert Endre Tarjan [29]. According

to Sleator and Tarjan [29], the time complexity of the B-tree algorithm in search operations is $O(log\ n)$, making it highly efficient for large databases.

Previous log analysis tools also allowed data to be collected from multiple sources with different formats. These tools have emphasized the importance of both automatic and manually customized log parsing methods, alongside the ability to export data to desired destinations. Additionally, the analysis of log data often involves the utilization of different machine learning models and AI techniques. Easily customizable dashboards are used to visualize the data, containing user-selected graphs such as histograms, line graphs, pie charts, and key statistics. The tools are often advertised as being used for information security and threat detection implying that log analysis tools primarily serve the purpose of enhancing software security.

In conclusion, these tools shared common traits of being adjustable, modular, and user-friendly, which can be attributed to the complicated nature of log data. Their adaptability and simplicity enable effective log analysis and management.

3 Efficient log analysis in Effector

This chapter presents the justified design decisions of the implemented log analysis tool as well as the more technical walkthrough of the implementation. We examined the log data of Effector in more detail, focusing on how the data is processed and imported into the log analysis tool. We introduce the basic structure of the program and discuss the anomaly detection methods. The chosen visualization choices are also justified.

3.1 Design decisions

When designing a log analysis tool, numerous factors must be considered. Design decisions are strongly influenced by the nature of the data and the main objective of the tool to be implemented. Technical and customer-defined constraints must also be taken into account, as this tool will handle customer-owned data. From now on, the log analysis tool implemented for Effector will be called LogTool. The logo of the LogTool is shown in Figure 5.

Figure 5: Logo of the LogTool.

3.1.1 Data processing

Parsing the log data is one of the first things to consider when dealing with data processing. As mentioned in section 2.2.2, log data in Effector is stored in SQL tables where each column corresponds to an attribute of the log entry. In the data processing of the LogTool, the selection of attributes to be included and the determination of attributes constituting the identification set for a single entry need to be made.

Zhu. et al. [1] study the accuracy of 13 log parsing methods, tested on 16 data sets from different sources. The study shows that methods performed significantly differently with different data sets. For example, the SLCT method performed with 0.882 accuracy on log data from the Android system but performed with 0.297 accuracy on log data from Linux. The results of the study show that there is a clear lack of precision in the methods as the mean accuracy of all methods was 0.677. This demonstrates how significant an impact the nature of the data has on the functionality of the methods and thus on the performance of the log analysis tools.

Since log data parsing methods are not precise enough and their performance is dependent heavily on data, LogTool uses a predefined format to parse log data. The tabular log structure of Effector also justifies why the data parsing should be implemented in a predefined format. When LogTool does not use an automatic log parser, the log message cannot be included in the parsed log data since the log message contains a variable part that should be parsed out automatically. Fortunately, the log data of Effector contains many other attributes that can be used to identify data entries accurately enough.

LogTool parses the raw log data so that the attributes 'Level', 'Sender', 'Log key', and 'Log date' are retained in the final data. In addition, the attributes for 'Organization' and 'Source table' are added to the parsed data which defines which organization and which log table the log entry originates. These attributes form the key of the parsed data and thus can be used to identify each log event. The original log data is grouped, merging duplicate rows for each day and including an 'Amount' attribute in the parsed data. This attribute specifies the count of log events that occurred during the day for each respective log event. The final format for the parsed log data is presented in Table 1 as well as the data types of each attribute.

Attribute	Data type
Organization *	varchar(12)
Source table *	int
Level *	char(1)
Sender *	varchar(50)
Log key *	varchar(50)
Log date *	int
Amount	int

Table 1: Data format of the log data used in LogTool. Asterisk (*) stands for the key attribute.

3.1.2 Data flow and centralization

As the program grows and the number of users increases, the program has to be distributed across many servers. In this case, log data also starts to accumulate in many different locations. To address this, a method for transferring and centralizing log data into one place is necessary.

According to a book by Kavis [30], centralization of log data has many benefits such as the fact that developers do not need to authenticate to multiple environments to access log data. Performing data mining, trend analysis, and thread detection becomes more straightforward, as tools can be executed atop the centralized logging database. The risk of losing log data is also reduced, as data is not stored on a local server that could be taken offline unexpectedly.

Effector is distributed to many organizations which is why centralizing log data is a necessary measure when we implement the LogTool. However, the log data should not be stored on a central server, as Effector handles sensitive data. Thus there is an

agreement with the customers that the data will remain on the customer's servers. Therefore, LogTool must be able to retrieve and aggregate log data from multiple locations during a run as efficiently as possible.

To achieve this goal, we implemented a functionality that allows a connection to multiple SQL servers and aggregation of log tables into a unified central table, which can be utilized in LogTool. A more detailed description of the implementation and functionality of data flow and centralization is provided in Section 3.2.1.

3.1.3 Choices for anomaly detection

Anomaly detection is one of the main functionalities of log analysis software. Anomaly detection can be implemented in many ways and there are many existing solutions. As stated by Hodge. et al. [16], the choice is strongly influenced by the nature of the log data and what type of anomalies are wanted to be detected. In an optimal scenario, achieving high accuracy across all anomaly- and data types is the goal. However, due to the unstructured nature of log data in real-world situations, attaining a high level of accuracy is challenging.

Hodge. et al [16], highlights two distinct categories of anomaly detectors that are emerging: methods based on statistics and distance, and methods based on machine learning and neural networks. A statistical anomaly detection method was chosen for the implementation of LogTool. The main reason for this choice was that statistical models do not need to be trained with training data, whereas machine learning and neural networks-based models need training and validation data sets.

The Effector's log data contains practically only error-based information, and there is no so-called pure log data to train the model with. Common anomaly detectors tried to find errors in unusual log event sequences as stated by Zhang. et al. [6], but Effector does not collect enough normal events for this to work. What is more interesting about Effector data is to see the statistics of errors, i.e. how many of each error occurred on each day as this kind of functionality is completely lacking in Effector.

In addition, Zhang. et all. [6] highlights the fact that machine learning-based models need constant retraining as the log structure of the program changes, which is unavoidable in the development style of Effector. Machine learning models are also computationally heavy and more complex to implement than statistical methods. According to Hodge. et al. [16], static methods also work well on one-dimensional data, which fits with the log data in Effector.

A combination of two algorithms was selected as the statistical anomaly detection premise for LogTool. A similar implementation had been used successfully in the thesis by Lemmens [31], so the starting point for the implementation was good. The two algorithms were also justified by the duality of Effector's log data when the data was analyzed visually. While we used a histogram to visualize data, two distinct patterns emerged in the time series of log events. One pattern showed infrequent spikes with many days lacking any events, while the other demonstrated frequent events almost every day, punctuated by spikes. These patterns are demonstrated in Figure 6 by using histograms generated by LogTool.

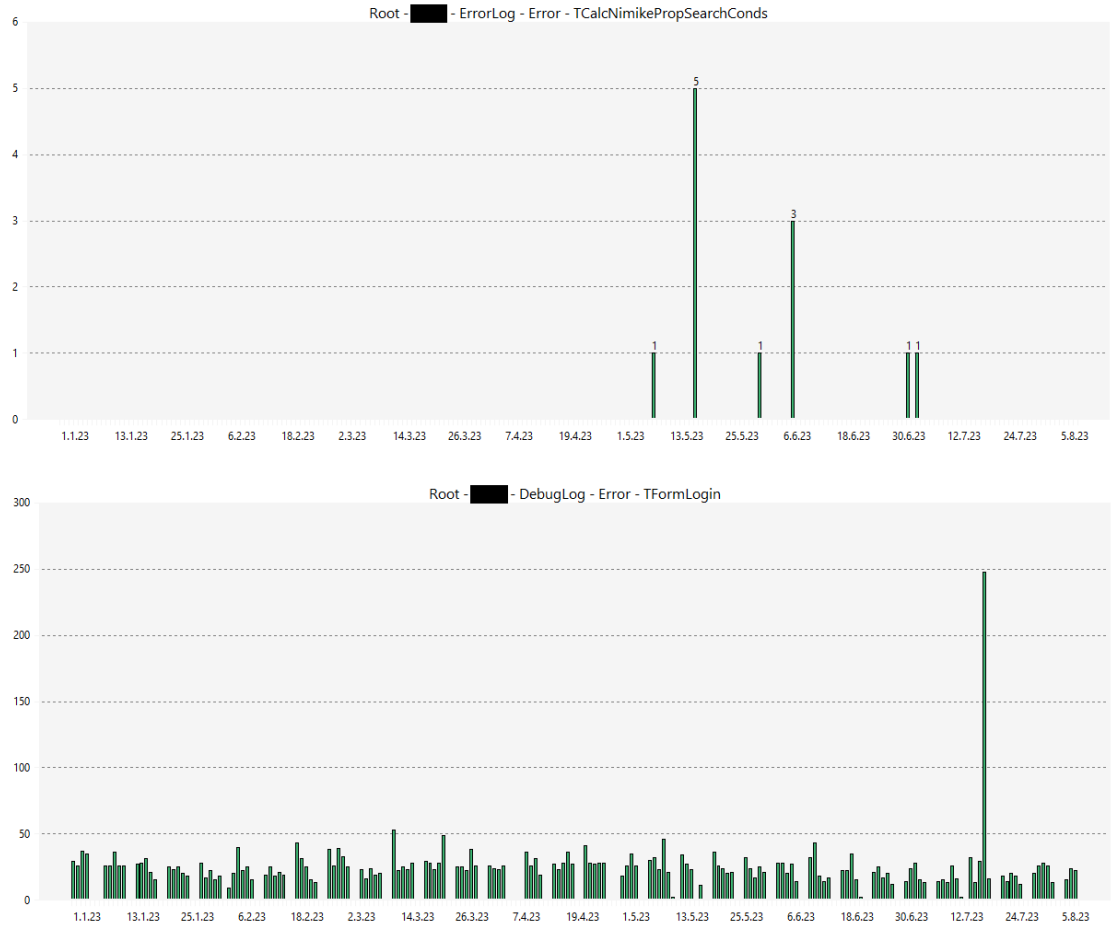

Figure 6: Histograms of time series of two different log events at the same time interval. The y-axis represents the number of log events per day and the x-axis represents dates from January 1, 2023, to August 10, 2023. The upper graph shows less frequent events and the lower graph shows more frequent events. In the upper graph, each spike can be assumed to be an anomaly whereas in the lower graph, spikes that deviate from the norm are anomalies.

The implemented algorithm denoted by f_n maps time series $\mathbf{x}$ to a series of anomalies $\mathbf{y}$ where 0 stands for the normal data point and 1 or -1 denotes an anomaly. A positive value indicates that the value is higher than the normal range, while a negative value indicates that the value is lower than the normal range.

$$f_n : \mathbf{x} \to \mathbf{y} \qquad \mathbf{x}, \mathbf{y} \in \mathbb{N}^n \tag{11}$$

$$f_n := \begin{cases} f_{rare} & \text{if} \quad r_0 \geq \alpha \\ f_{sz} & \text{if} \quad r_0 < \alpha \end{cases} \tag{12}$$

The main algorithm consists of two sub-algorithms. Rare anomaly algorithm denoted by f_{rare}, that maps all non-zero data points to 1 like presented in Equation

13, and smoothed Z-score algorithm, denoted by f_{sz}, that uses Equation 10 to map time series to anomalies. A zero-event ratio r_0 and the threshold α determines which algorithm is used. A zero-event ratio is the ratio of dates with zero occurrences of log events to all dates in a time series. If the ratio exceeds or equals the chosen threshold, the rare anomaly algorithm is used, otherwise, the smoothed Z-score algorithm is used. A threshold of $\alpha = 0.95$ was used as Lemmens [31] found this value to be effective. We found this value to be effective on Effector data as well.

$$y_n := \begin{cases} 1 & \text{if} \quad x_n \neq 0 \\ 0 & \text{if} \quad x_n = 0 \end{cases} \tag{13}$$

The smoothed Z-score algorithm was chosen to detect anomalies in denser data, as it is designed to detect clear peaks from the continuous data. The algorithm, grounded in statistical dispersion, triggers a signal if a new data point deviates from a moving mean by more than x standard deviations. The moving window enables data smoothing, ensuring that the data is processed in smaller batches so that only the neighborhood of the data point is taken into account when detecting peaks.

The smoothed Z-score algorithm is cited in many academic papers from 2017 to the current year 2023 as presented by Brakel [19]. The algorithm or a variant of it has been used, for example, in a gait phase prediction model used for evaluating a controller for prosthetic legs by Minjae et al [32]. Mehmet et al. [33] used the smoothed Z-score algorithm in a dynamic speaker localization to successfully separate background noise from the input signal which significantly reduced the computational cost of the localization algorithm. Bernard et al. [34], on the other hand, used the algorithm in data analysis related to the global honey bee population and the products they produced. The algorithm was used to detect abnormal drops from study variables in a time series.

The extensive use of the algorithm confirms that the algorithm works robustly. This versatility is further enhanced by the fact that the algorithm can be adjusted by changing three different parameters. The tuning of parameters for LogTool is presented in Section 3.2.4.

3.1.4 Choices for user interface

The user interface is also an important part of a log analysis program, as good visualization can help to clarify the presentation of log data. The following features have been observed in the user interfaces of commercial log analysis programs such as Splunk [2], Elastic Stack [3], and LogRhythm [4].

The dashboard seems to be the most central part of the user interface of these applications. A dashboard can visualize log data in many formats and provide a quick overview of the most important information that can be obtained from the data. The dashboard also allows for easy customization due to its modular structure, which makes the dashboard well-suited for displaying log data as log data is very diverse. This is why the Dashboard serves as the primary window within the LogTool.

Another common user interface feature that emerged is log data retrieval and visualization of the occurrence of the retrieved log event. This indicates how the log

event has evolved, from which anomalies can be visually detected. We visualized data using graphs tailored for time series analysis, such as line graphs and histograms. Additionally, we implemented a dedicated window in LogTool to facilitate log data search and visualization of the corresponding events through histograms. In the search view, it is possible to filter different attributes, for example by searching for log data from the last 30 days.

LogTool is divided into three main windows, which can be navigated in tabular form. The first window 'Home' is the initial window of the program and is shown first to the user. The dashboard is located in the home window, where all the most important information can immediately be observed. The second window 'Alerts' is a window for the alert of anomalies. This window includes a search functionality where old alerts can be observed in tabular format. The alert window contains a button to manually run the anomaly detector, which will go through the latest log events and check them for anomalies using the algorithm described above. The last window 'Search' is a window for searching the whole log data as described above.

3.2 Implementation

In this section, we provide a more detailed description of the implementation of the software. The key components and underlying mechanisms that contribute to the functionality and behavior of the program are explained.

3.2.1 Structure of the data flow

Figure 7 shows the flow chart which illustrates the architecture of the data flow from Effector to LogTool. Blue boxes represent SQL tables, green ovals represent code classes that handle data and yellow ovals represent end functionalities.

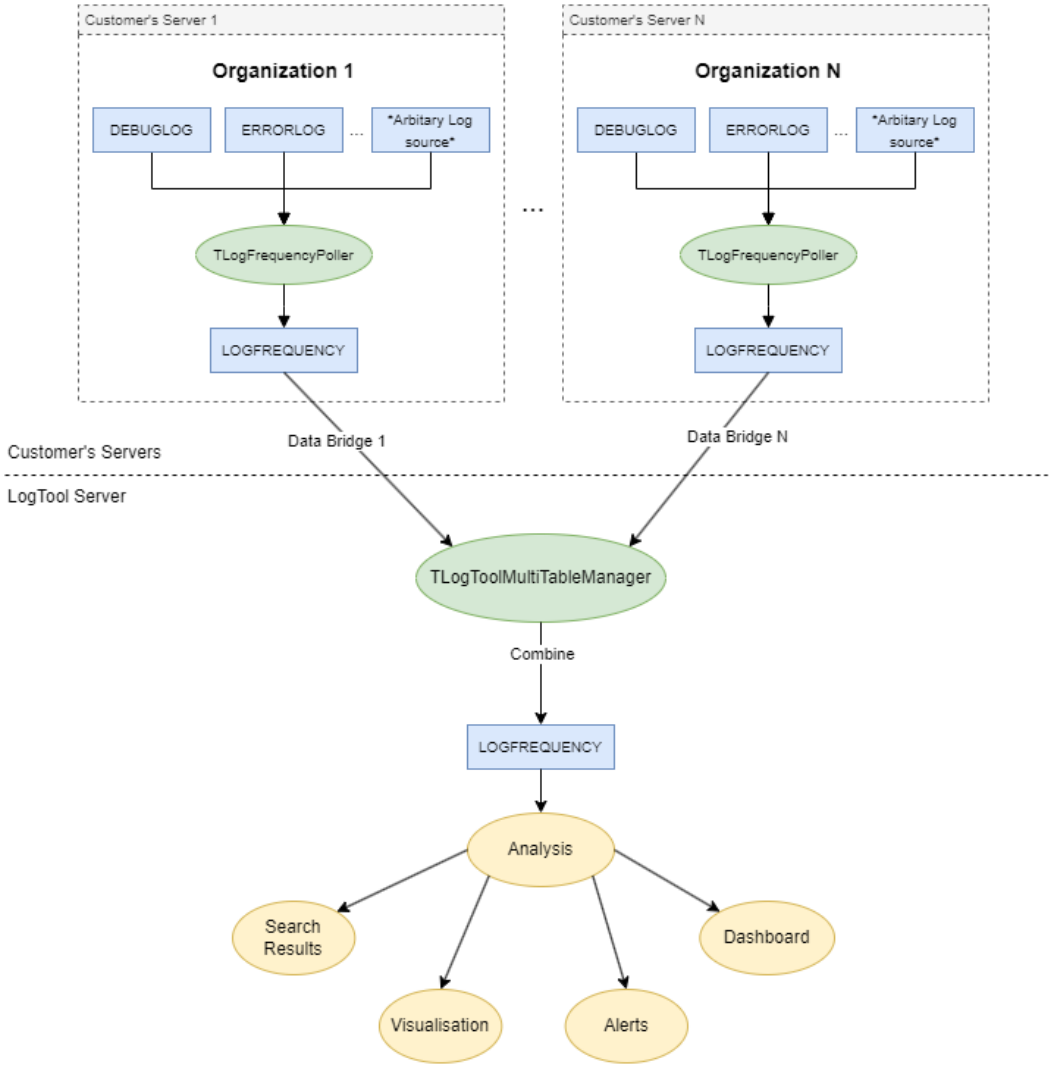

Figure 7: Flowchart illustrating the log data processing and transfer from Effector to LogTool.

Data processing starts at the customer's Effector installations, on the customer's servers. Each client's environment is represented in the diagram by a dotted box. In reality, there are 22 environments, but in the diagram, this is represented by N. In the first data processing phase, the multiple log sources (DebugLog, ErrorLog, etc.) are merged into the 'LogFrequency' SQL table, which contains the daily amount of each log event occurrence from the source tables. This is the first level of centralization of data. The 'LogFrequency' table contains all columns described in Table 1.

TLogFrequencyPoller

Pollers in Effector are recurring background tasks, each with its configurable execution interval. The 'LogFrequency' table is populated through the 'TLogFrequencyPoller' class, which executes daily. The underlying concept involves the aggregation of log data from multiple sources. Through this process, the poller groups the raw log data into daily occurrences per log event, packaging the data into a more compressed format. Since the poller is run once a day, only new log entries after the last run are included in the grouping. This makes the poller more efficient and faster, giving more resources to other pollers in Effector.

Even if the poller has to process a large number of log entries, SQL's group-by-statement is an efficient way to group large amounts of data. In the test environment, this SQL query processed about 1 million log entries in about 5 seconds, which is fast by Effector pollers' standards.

Listing 2: Example of SQL query which group the DebugLog table by using group by statement.

```
select DebugLogKey, DebugLogLevel, DebugLogSender,
       convert(date, DebugLogDate) LogDate, count(*) Amount
from DebugLog
group by DebugLogKey, DebugLogLevel, DebugLogSender,
       convert(date, DebugLogDate);
```

In other words, the 'TLogFrequencyPoller' processes each specified log source table, aggregating the data into the more user-friendly 'LogFrequency' table for utilization in LogTool. This operation takes place on individual servers for each organization, effectively distributing the workload of log data processing.

TLogToolMultiTableManager

Following the aggregation of log data within each organization, the next step involves merging the resulting 'LogFrequency' tables from various servers into a unified table for further log data usage. It must be taken into account that the combined log data must not be stored in the external database, rather, the implementation operates exclusively in memory during run time.

For this purpose, we developed a 'TLogToolMultiTableManager' class that allows normal SQL database queries to be executed on multiple servers simultaneously. In Effector, the connection to the database is made through a data bridge, an abstraction that defines the link between Effector and the SQL server. The most common type of data bridge in Effector, which is used to connect to the client's server, is the ADO Bridge. ADO connection, as described in the study by Wen. et al. [35], is an application programming interface (API) developed by Microsoft, designed to provide access to both relational and non-relational databases. 'TLogToolMultiTableManager' utilizes ADO data bridges to establish a connection to the client's servers, where each connection is formed on a separate thread. Multi-threading enables connection to different organizations and retrieving data from them in a parallel manner, which speeds up the process significantly. After fetching data, the data is merged into one data table, which is located and accessed in memory. In the merging process, information about the source organization is also added to the final log data.

After the second level merging of the data, log data can be used in LogTool in different functionalities such as in the dashboard, search, and visualization. This two-level data merging enables efficient and functional data flow and data processing from Effector to LogTool so that the data remains on the customer's server. On top of that, the log source does not necessarily need to be in Effector but any SQL Server will suffice.

3.2.2 Structure of the program

The main units of the program and their relations are illustrated in Figure 8. Program units are represented by blue squares, while green ovals signify code classes, and yellow ovals signify child classes. The program is implemented in a tree-like manner as can be seen from the diagram. The program starts with a login window where the user has to enter the username and password for the centralized password management software. Password management software has all the credentials stored to connect to the customer database and the API for that software allows LogTool to access these credentials. In the login window, the user can also choose which of the organization's servers to connect to and which database to use (production, test, or training).

The functionality of the program is built around the 'LogToolMain' unit, which acts as the main handler for the program. This unit handles all initialization and deinitialization calls and the creation of instances of the required classes. This unit also contains the main form on which the UI is built. The 'LogToolMain' unit serves as the origin of all sub-units responsible for containing the primary functions of the program. These sub-units include 'LogToolHome', 'LogToolAlert', and 'LogToolSearch'.

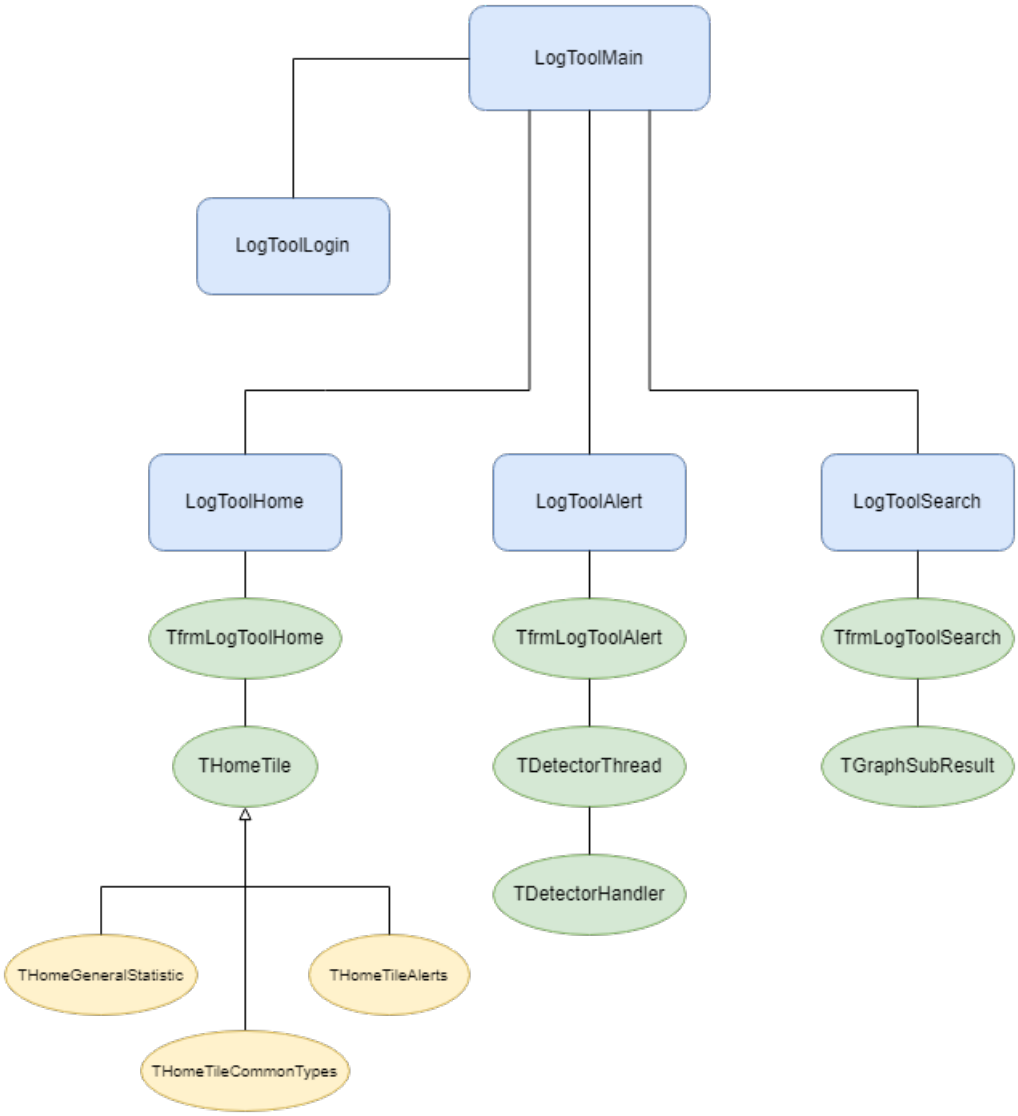

Figure 8: Architecture of the LogTool represented by the main units and classes.

'LogToolHome' is the first unit the user sees after logging in. This unit contains the 'TfrmLogToolHome' class, which contains the form on which the UI of the dashboard is built. The form has a grid-like structure similar to a dashboard, with cells containing instances of the 'THomeTile' abstract class. The 'THomeTile' class is inherited by three different child classes that form the tiles of the dashboard. The child class structure of the dashboard makes it easy to implement new tiles as needed.

The 'THomeGeneralStatistic' class displays information about the information generated from the log data, such as how many log entries there are in the data and the time interval over which the data was collected. In this tile it is also possible to group log data by log attributes, which can be used, for example, to see how many log entries each organization contains. The 'THomeTileCommonTypes' class displays the most common log attributes in a pie graph. The user can change which attribute is visualized from which time interval. With this tile, the user can quickly

visually see the structure of the log data. The last tile implemented in the dashboard is 'THomeTileAlerts', which displays a line graph of the number of alerts in the last 14 days.

The 'LogToolAlert' sub-unit contains all functionality related to anomaly detection and alerts. This includes the functionality to search and list the occurred alarms as well as run the anomaly detectors and save occurred alerts to the database. 'TDetectorThread' is a helper class for the 'TDetectorHandler', which runs the detector handler in its own thread so that the LogTool can continue to run normally even if the detector handler is running. Implementation of the detector handler and anomaly detection is explained in more detail in Section 3.2.3.

The final sub-unit is the 'LogToolSearch', which contains all implementation and visualization related to the search functionality of the log data. The search window is implemented in the 'TfrmLogToolSearch' class which includes all the filters for the search and handles rendering of the result table. This form also includes the 'TGraphSubResult' class, which handles the rendering of the histogram from the retrieved log event. The screenshots of the windows are represented and discussed more preciously in Section 4.2.2.

In conclusion, the LogTool is implemented by using the Delphi framework, which provides a comprehensive UI editor and supports object-oriented programming (OOP). OOP allows a tree-like structure, which has been heavily used in the implementation of LogTool. The tree-like structure gives flexibility to the program, enabling the integration of new functionalities without necessitating a complete rewrite of the existing code. This is an important feature due to the changing nature of log data.

3.2.3 Anomaly detection

The UML diagram of the anomaly detection in LogTool is presented in Figure 9. Anomaly detection relies on the 'TDetectorHandler' class, which serves as a practical handler for various anomaly detection algorithms. The detector handler includes the source storage of the log data which is iterated so that a time series is generated for each key group. These resulting time series are then subject to classification through the 'ClassifyTimeSeries' function, which determines the appropriate anomaly detector algorithm to apply to each specific time series.

Each anomaly detection algorithm derives from the 'TAbstractAnomalyDetector' class, which establishes a shared interface across all detection algorithms. The 'TAbstractAnomalyDetector', along with its subclasses, features a 'CheckAnomalies' function that outputs detected anomalies within the time series. If the zero date ratio of the time series is greater or equal to 0.95, the 'TRareAnomalyDetector' class is used, otherwise, the 'TSmoothedZAnomalyDetector' class is used. This architecture allows other algorithms to be added to the implementation, but currently only these two are implemented. Algorithms of the 'TRareAnomalyDetector' and 'TSmoothedZAnomalyDetector' are respectively presented in Listing 3 and 4, found in the appendix.

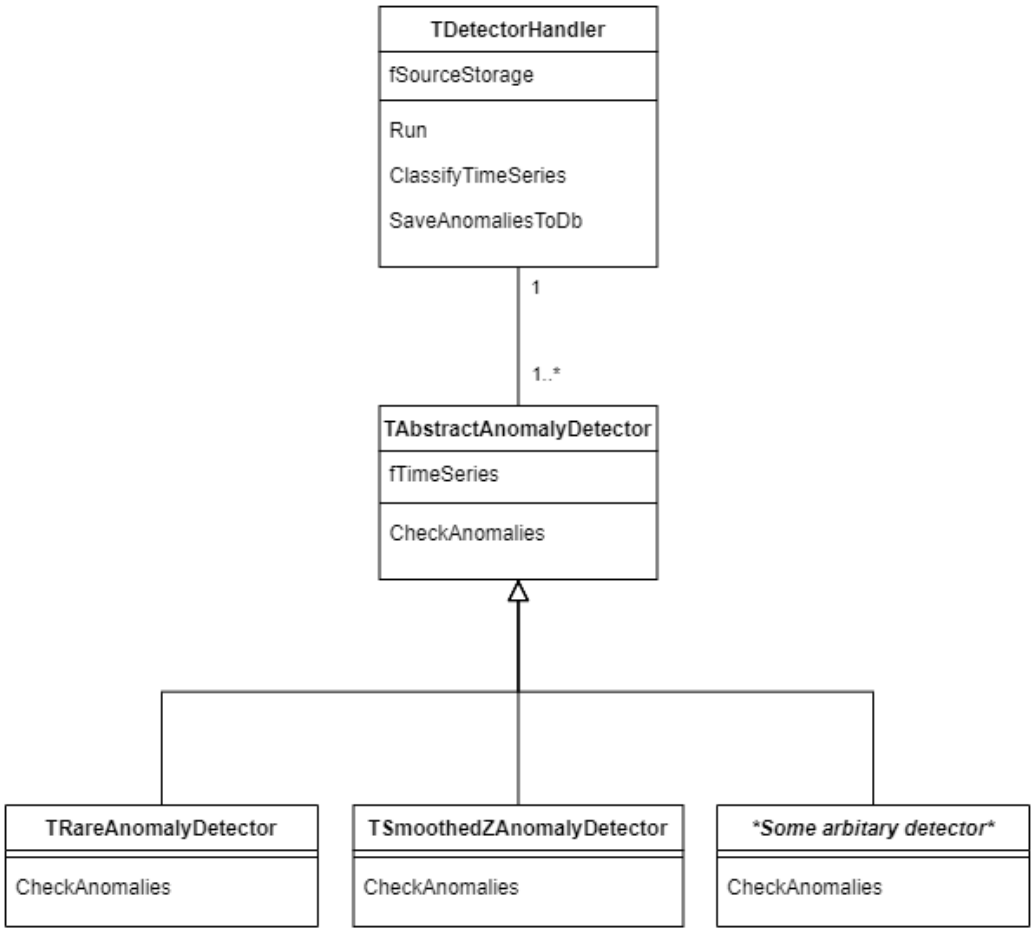

Figure 9: UML-diagram of the anomaly detector.

According to Listing 4, the smoothed Z-score algorithm begins by initializing the mean and standard deviation windows through the calculation of the mean and standard deviation from the initial window of the input data. The initial window is cropped from the first element to the Nth element, as defined by the lag parameter. The input data is iterated from the lag to the end of the data, so the data points before the lag value will not be analyzed for anomalies.

The Z-value for the iterated data point is calculated by using the mean and standard deviation windows represented in Equation 9. If the Z-value exceeds the chosen threshold value, the data point is classified as an anomaly by updating the result array with -1 or 1 depending on the previous data point from the mean window. If the iterated data point is greater than the previous mean, the resulting anomaly is marked as 1 and vice versa. If the anomaly is detected, the so-called adjusted signal is updated by using the iterated data point and previous data point from the adjusted signal, both influenced by the influence parameter as presented in Equation 6. The higher the influence parameter is, the more the input signal affects the adjusted signal. If the anomaly is not detected for the current data point, the result signal is

updated with 0 and the adjusted signal is updated with the input signal without the influence.

At the end of the iteration, the new values for the mean and standard deviation windows are calculated by using the current sliding window. The new mean and standard deviation values are used to determine the Z-value for the next data point in the next iteration. The iteration continues until the last data point is iterated.

3.2.4 Tuning of the smoothed Z-score algorithm

The smoothed Z-score algorithm can be fine-tuned by using three parameters: Threshold, Lag, and Influence. To facilitate the tuning process, we developed a simple program that enables easy adjustment of these parameters and visual assessment. First, the correct threshold was determined. Figures 10 and 11 both show two histograms. The top histogram displays the daily occurrence of the specific log entries, and the bottom histogram shows the anomalies observed in the time series. Both figures represent the same time series but the configuration differs. In Figure 10, the algorithm was set to Threshold = 4.5, Lag = 30, and Influence = 0.5, and in Figure 11 the algorithm was set to Threshold = 2.0, Lag = 30, and Influence = 0.5, so the Threshold is about twice as large in Figure 10.

Figure 11 shows that the algorithm is too sensitive, as anomalies are detected frequently even though the number of log entries does not vary much. On the other hand, figure 10 shows a single anomaly that occurs at a clearly detectable peak. This suggests that the threshold value of 2 is too low and that a better value for the threshold is close to 4.5. This same method was applied to other time series and it was found that a slightly smaller threshold of 4.0 gave consistently good results.

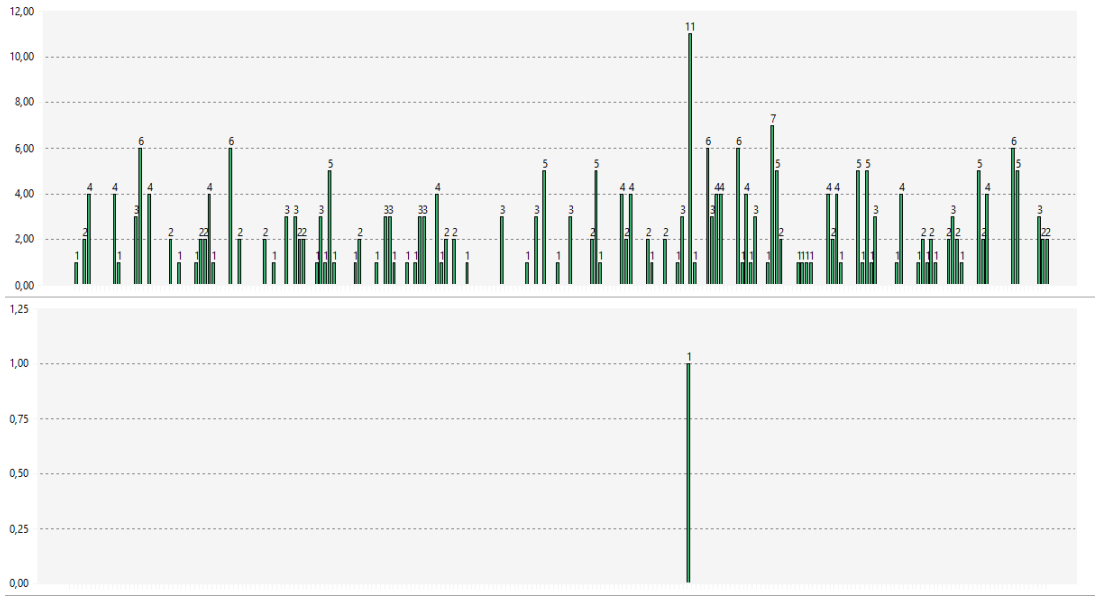

Figure 10: Time series (Top) and detected anomalies (Bottom). Threshold = 4.5, Lag = 30, and Influence = 0.5.

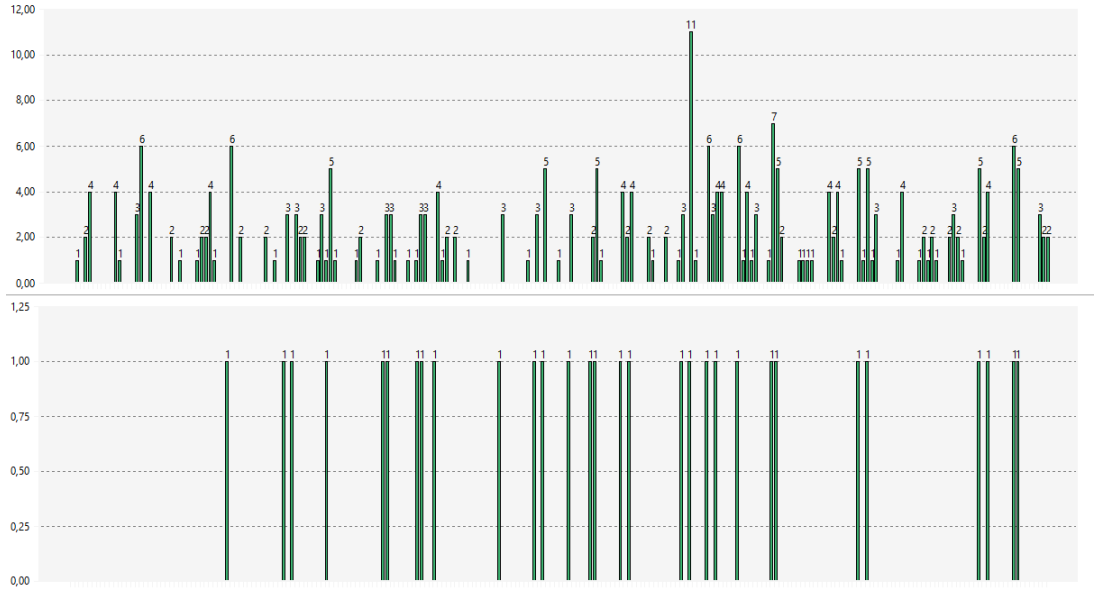

Figure 11: Time series (Top) and detected anomalies (Bottom). Threshold = 2.0, Lag = 30, and Influence = 0.5.

Figures 12 and 13 illustrate a different kind of time series and anomaly response. A consistent daily error rate emerges within a defined time frame in this scenario. The onset of errors is identified as a positive anomaly, while the cessation of errors is identified as a negative anomaly. In Figure 12, the algorithm was set to Threshold = 4.0, Lag = 30, and Influence = 0.5, and in Figure 13 the algorithm was set to Threshold = 4.0, Lag = 30, and Influence = 0.1.

This situation gives a good example of how the influence parameter works. When the influence parameter is high, the faster the algorithm adapts to the new normal, and new values of the same magnitude are no longer interpreted as anomalies. When the influence is lower, the adaptation to the new normal takes longer. If influence were 0, every data point in the figure would be interpreted as an anomaly, as the algorithm would not adapt to the new normal at all.

In LogTool, some adaptation is desired because the system may drift into a new normal state. In these scenarios, a controlled number of anomalies is preferred, avoiding excessive repetition of the same alerts. This is why an influence value of 0.5 is preferable over 0.1 in this situation. The value of 0.5 for influence was also tested with other time series and showed good results in those cases as well.

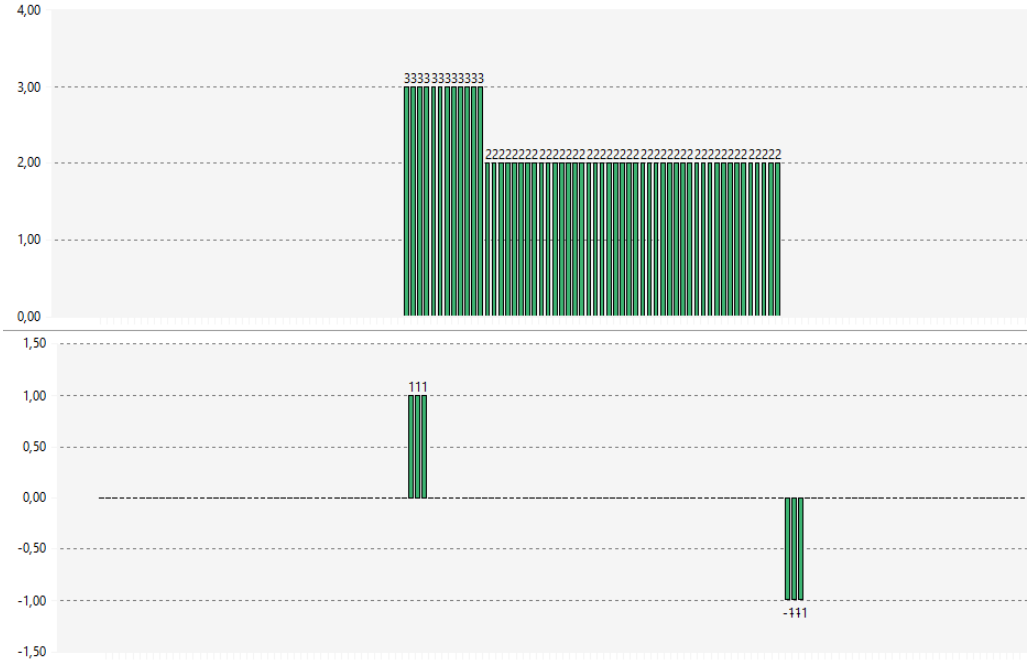

Figure 12: Time series (Top) and detected anomalies (Bottom). Threshold = 4.0, Lag = 30, and Influence = 0.5.

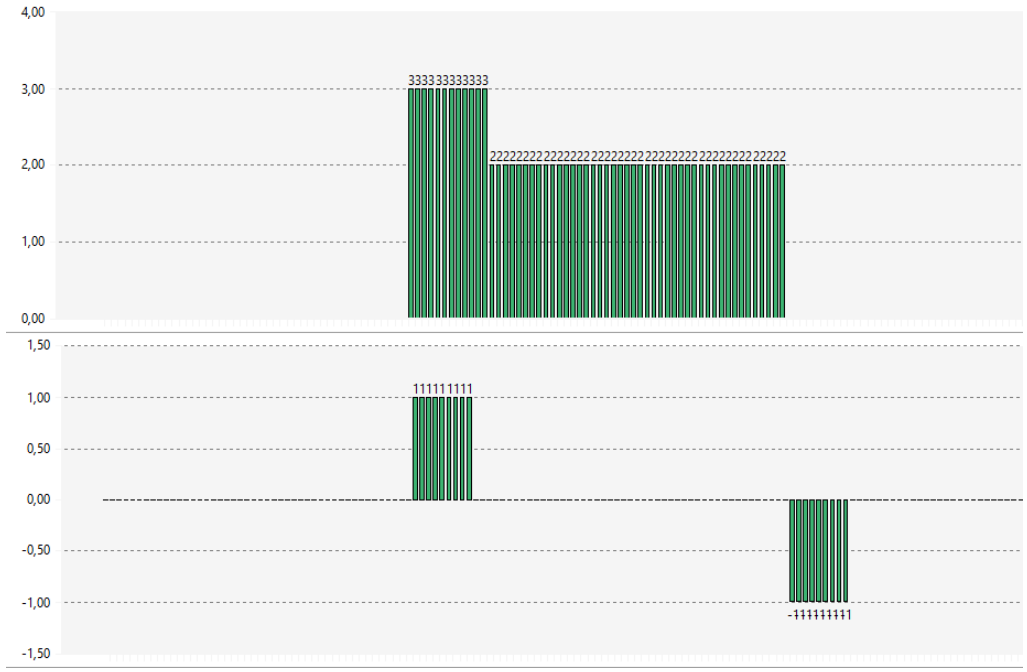

Figure 13: Time series (Top) and detected anomalies (Bottom). Threshold = 4.0, Lag = 30, and Influence = 0.1.

The lag parameter corresponds to the size of the sliding window and in LogTool the unit of window size is one day. For example, a lag value of 30 means a window size of 30 days. If the window size is too small, the algorithm examines only a small portion at a time, providing an incomplete view of the data. On the other hand, if the window is too large, significant deviations can distort the result too much. A lag value of 30 turned out to be a good value for this log data, as one month is sufficient to provide information on the log data without being too prone to large deviations.

The final configuration ended up being a Threshold = 4.0, Lag = 30, and Influence = 0.5. During the configuration process, it was necessary to consider that modifying one parameter typically required an adjustment to another parameter. This correlation arises due to the shared impact of these parameters on the same functionality. Consequently, the absence of an explicit method for determining an optimal configuration underscores the requirement for estimation and informed assessment.

4 Evaluation

In this chapter, we evaluate the results of the LogTool. Real log data from Effector is employed to present both general and anomaly statistics, accurately depicting log data in practical scenarios. We evaluate the performance of LogTool using real data, thoroughly assessing its strengths and limitations. Furthermore, we demonstrate the use of LogTool through case studies.

4.1 Analysis of the log data

The data analysis employs real-world log data obtained from three distinct Effector organizations. We selected three organizations of varying sizes as the source entities to ensure a comprehensive evaluation of LogTool. These include a small-scale organization (referred to as Organization A), a medium-sized organization (referred to as Organization B), and a large organization (referred to as Organization C). For security considerations, the actual names of these organizations have been concealed. Data has been gathered starting from the beginning of the year (January 1, 2023) up to (August 21, 2023), aggregated from two separate Effector log tables; DebugLog and ErrorLog.

As shown in Table 2, organization C holds close to half a million log entries from the source log storage. This quantity is around twice as large as organization B and a significant forty times larger than organization A. The number of logs seems to be strongly influenced by how much Effector is used and how many different functionalities Effector has enabled per installation. Effector usage and the organization scales are estimated in Table 2 by the "Logins" column, which shows how many logins have been made in a given period. The scale of the functionality of Effector can be estimated by the number of modules that are enabled per organization. This is presented in Table 2 by the "Modules" column.

Organization	LogFrequency Count	Source Log Count	Logins	Modules
A	1 677	10 948	20 247	96
B	2 789	267 441	21 200	113
C	6 964	481 632	130 748	138

Table 2: Log data entry volumes across organizations of various sizes. The column labeled "LogFrequency Count" represents the aggregated count of log entries derived from the source log data.

The size of the processed 'LogFrequency' data is influenced by the frequency of identical log entries occurring within a single day. When numerous instances of the same log entries are present throughout the day, the 'LogFrequency' table will contain fewer entries. This reduction occurs because identical log entries from the same day are merged into a single entry.

4.1.1 General statistics

The examination of the log data involves extracting various statistical characteristics from the 'LogFrequency' data. The subsequent analysis focuses solely on log entries that have been saved to the database. Therefore, the possibility of zero events is eliminated, as log types occurring zero times within a day will not be stored in the database. Zero days are only taken into account in the time series. Table 3 represents the mean, standard deviation, median, minimum, and maximum of the log frequency data from three organizations.

Org.	Log Count	Mean	SD	Median	Min	Max
A	1 682	6	11,62	2	1	162
B	2 805	95	1 121,6	3	1	26 340
C	6 986	69	268,3	5	1	12 621

Table 3: Statistical characteristics of log data using all log levels.

As we can see from Table 3, the standard deviations and maximum values are significantly high for B and C organizations. This clearly raises the average in these cases. However, the small median reveals that most log events are in the order of three log entries per day. Therefore, the results clearly show a significant increase in daily log events for organizations B and C. Further examination of the log data reveals that this increase is due to a log event categorized as "Debug" level indicating that it is debug log data and not an error log.

Statistical characteristics are taken only from log entries with an "Error" level to further analyze the error logs. Statistical characteristics of "Error" logs are represented in Table 4.

Org.	Log Count	Mean	SD	Median	Min	Max
A	876	8	13,16	3	1	162
B	1 288	6	16,45	2	1	248
C	4 003	18	29,49	3	1	332

Table 4: Statistical characteristics of log data using error log level.

As we can see in Table 4, the daily frequency of error logs maintains a notably steadier trend compared to the overall data set. This could be due to the tendency for operations to halt when an error arises, whereas instances of debug logs allow for ongoing operations and the possibility of generating new debug logs. The medians and means now exhibit similar magnitudes both within themselves and across different organizations. While a correlation between the maximum value's magnitude and the log count is observable, the sample size is not sufficient to validate this observation. Comparing the log counts of the two tables, it is noticeable that the error data constitutes approximately 60 % of the total data set.

4.1.2 Anomaly statistics

Table 5 presents the findings from the implemented anomaly detector, displaying the number of detected anomalies for each organization and their respective counts relative to the log data volume. The table also provides statistics on the algorithms responsible for detecting the anomalies.

Org.	Anomalies count	Rare count	Smoothed Z count	Ratio to logs
A	283	150	133	0,32
B	401	201	200	0,31
C	1 214	604	610	0,30

Table 5: Anomalies count.

As depicted in Table 5, the correlation between the number of anomalies and the quantity of log data is evident. This logical relationship is rooted in the understanding that a larger amount of logged data corresponds to a higher likelihood of anomalies occurring. Both rare- and smoothed Z -algorithms detected about the same number of anomalies in each organization. The ratio of anomalies to the volume of log data is approximately one-third for each organization. The consistent nature of this ratio signifies that the algorithm operates uniformly across all organizations.

The progression of anomalies over time was assessed by computing the weekly anomaly rate for each organization, as demonstrated in Figure 14.

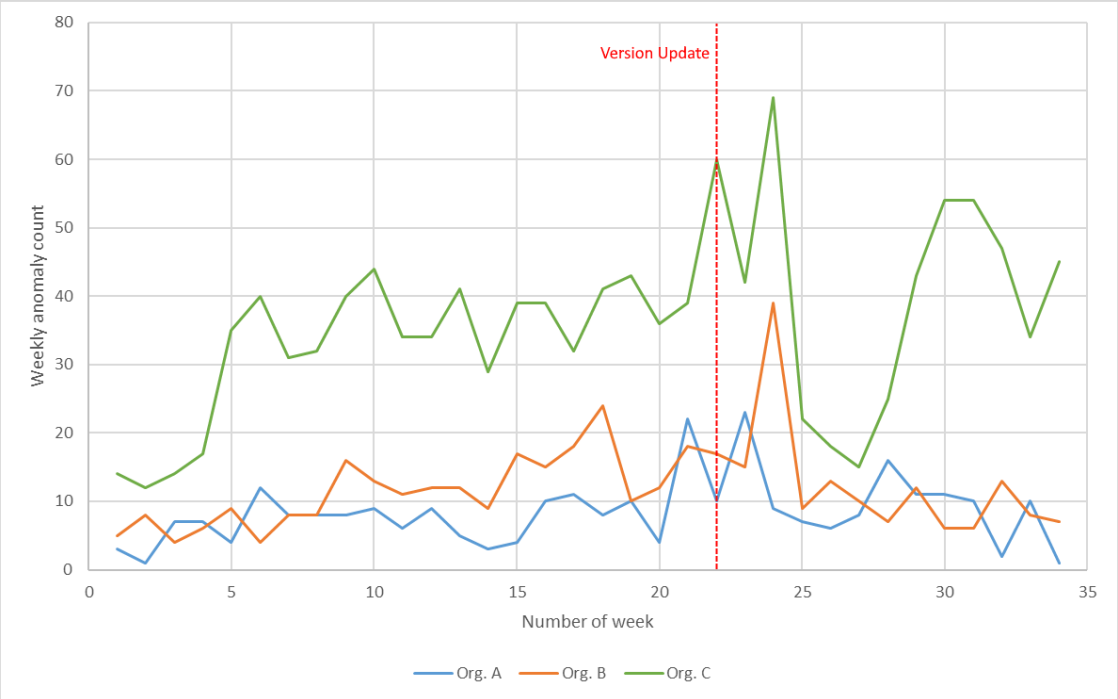

Figure 14: Weekly anomaly count of three organizations.

Between weeks 1 and 20 of the observation period, the number of anomalies remained relatively stable, discounting minor fluctuations. Organizations A and B both exhibit a weekly anomaly count of approximately 10, while for organization C, the weekly anomaly count is roughly 35. Notably, a sharp increase in weekly anomalies occurred around week 24 for all organizations, suggesting that some major change to the code was made in production around this time. The version update of Effector was released in week 22, with customers installing the new version in their environment after that. This week corresponds with the peak on the graph, indicating the algorithm's capability to detect version updates as well.

4.2 Assessment of implementation success against requirements

In the introduction, we established four requirements for the program. In this section, we evaluate the program's performance against the requirements. The evaluation assesses the extent to which the program has successfully fulfilled the predefined requirements, providing insights into its overall effectiveness and suitability for real-life use. The assessment is presented in the following sections for each requirement.

4.2.1 Retrieving data from multiple environments

The first requirement was that the software must be able to retrieve log data from multiple log tables and multiple organizations simultaneously. LogTool currently allows data aggregation from multiple log tables, which was implemented with a poller that goes through the internal log tables of Effector and aggregates them. The poller is currently run once a day, so LogTool does not collect data in real-time, which is a minor setback. However, the poller can be adjusted to a higher frequency in the future if it is found that LogTool requires it.

The poller has currently been enabled to eight organizations in production for about two months and we have not observed any problems with the operation of the poller and the running of other pollers. This has been verified by the Effector event log, which records all poller runs, their success, and duration. For example, for a large organization C, the initial run of the poller lasted about 63 seconds, after which the daily runs lasted only about 3 seconds. Thus, we can conclude that LogTool meets the requirement of retrieving the data from multiple data tables.

LogTool also combines log data from different servers simultaneously. For this purpose, we implemented functionality in LogTool that allows multiple ADO connections to any number of different databases. This allows log data to be examined simultaneously from multiple organizations, which was not possible before. However, with the current implementation, as the number of connected organizations increases, the startup and initialization time of LogTool experiences a noticeable slowdown. For instance, when launching LogTool with only organization A, the startup time is 5 seconds. However, if LogTool is opened with organizations A, B, and C, the startup time increases to 11 seconds. Nevertheless, this does not prevent the use of the software, and the real-time performance of the implementation is not required. Thus,

we can conclude that LogTool achieves the requirement of being able to connect to multiple organizations simultaneously.

4.2.2 Visualization of the log data

The second requirement was that the software must be able to visually represent log data in a manner that enables the tracking of log data evolution over time. The software should support searching and filtering of log data based on log attributes and presenting basic data statistics to the user.

In LogTool, data visualization was implemented on two pages. The first is a home window that includes a dashboard. The second is a search window comprising a search feature, along with time series visualization and data statistics. The home window is presented in Figure 15, and the search window in Figure 16.

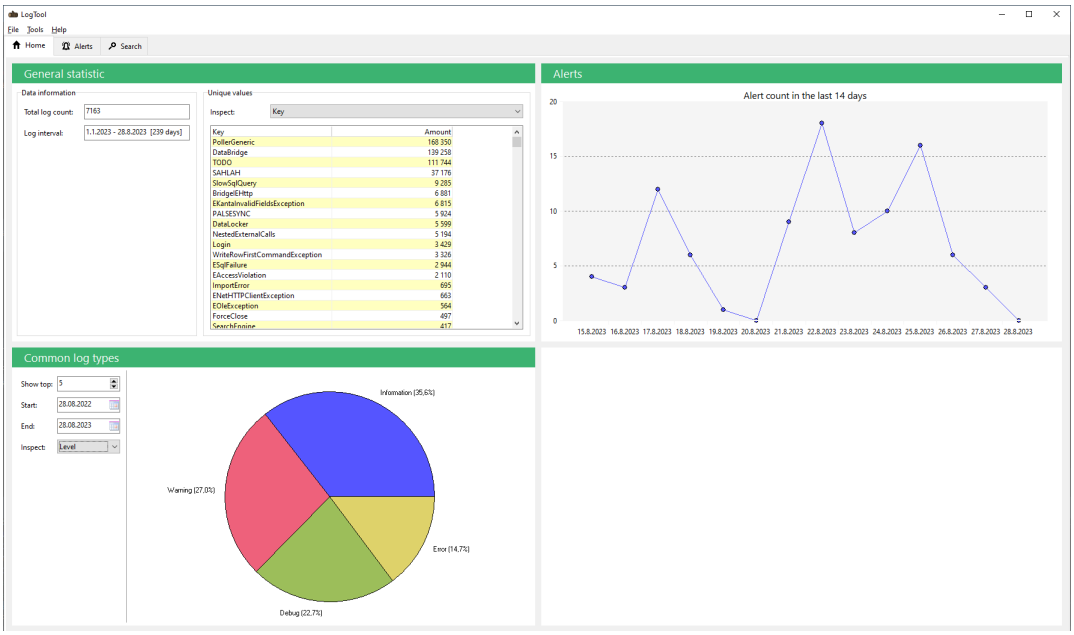

Figure 15: Home window and the dashboard of LogTool.

In the home window, log data is visualized and presented with a table and a pie chart. The table displays the precise count of inspected log data attributes, while the pie chart conveniently illustrates the proportional sizes of the attribute values. In Figure 15, the table presents the unique key attributes along with their respective counts from the log data. Meanwhile, the pie chart visually conveys the relative sizes of the level attribute values. Inspected attributes can be changed from the drop-down menu. In addition, the home window also visualizes the number of anomaly alarms in a line graph, which immediately shows how many alarms have occurred each day over 14 days.

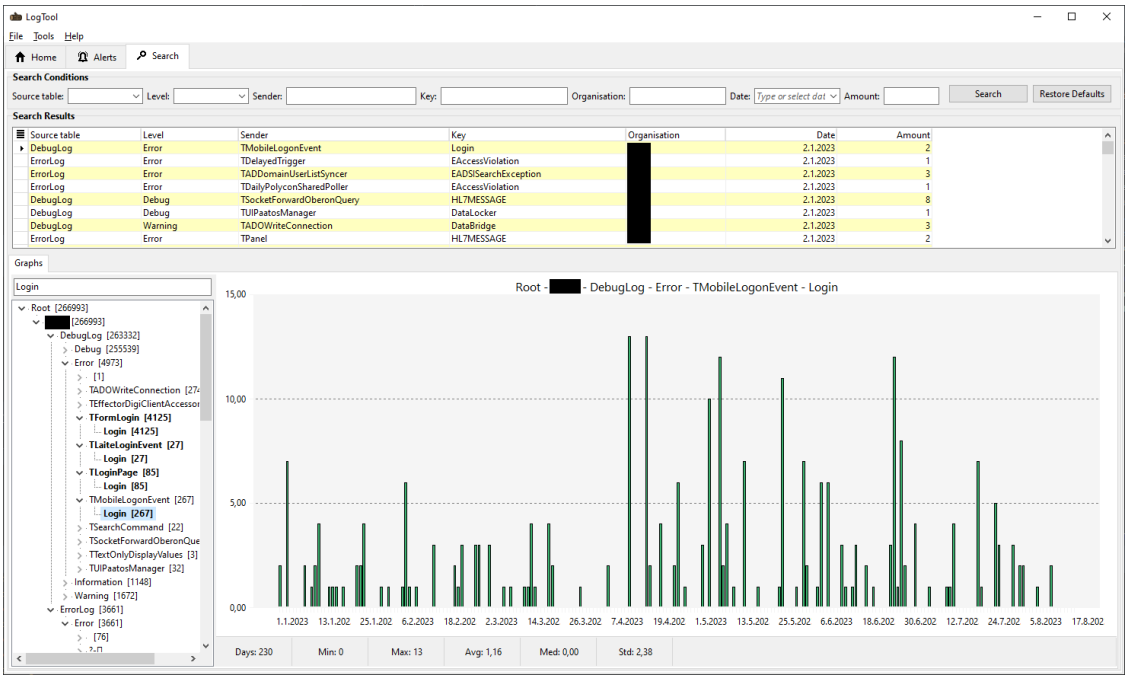

Figure 16: Screenshot from the LogTool with search as the active window.

The histogram is displayed at the bottom of the window and the results of the search, as well as the filter, are located at the top of the window. The histogram provides a convenient visual representation of the daily log count for the chosen log event. There is also a tree view explorer next to the histogram, which provides a convenient way to change the log entry to be rendered. Moreover, utilizing the tree view on the left allows for the selection of higher-level attributes. For example, the tree view enables the visualization of log events aggregated solely from the ErrorLog table. Below the histogram locates a status bar that shows general statistics from the selected time series such as the number of data points, minimum and maximum value, mean, median, and standard deviation.

As presented above, LogTool includes a lot of data visualization and statistics. The graphs are clear, dynamic, and easy to use. However, the old Polycon internal graph library is utilized to render the graphs, resulting in a simple and somewhat grainy visual appearance. This is due to the lack of anti-aliasing in the graph library, which would otherwise smooth the edges of the graphs. Furthermore, when multiple organizations are simultaneously viewed, the graph update time increases, thereby decreasing user-friendliness. However, these are minor problems in this scope, so we can conclude that LogTool achieves the requirement of data visualization.

4.2.3 Anomaly detection and alerts

The third requirement was that the software must be able to automatically detect anomalistic, unusual occurrences of log data and alert them to the user. LogTool includes an anomaly detector that can be manually triggered from the alert window. Moreover, the detector is activated automatically upon starting the software. Two

algorithms were used to detect anomalies, the rare and smoothed Z-score algorithms. The results showed that both algorithms detected approximately the same number of anomalies, so splitting the detector into several smaller algorithms was a justified and successful decision. Detected anomalies are saved as alerts to the database and they can be searched and viewed on the alerts window, illustrated in Figure 17.

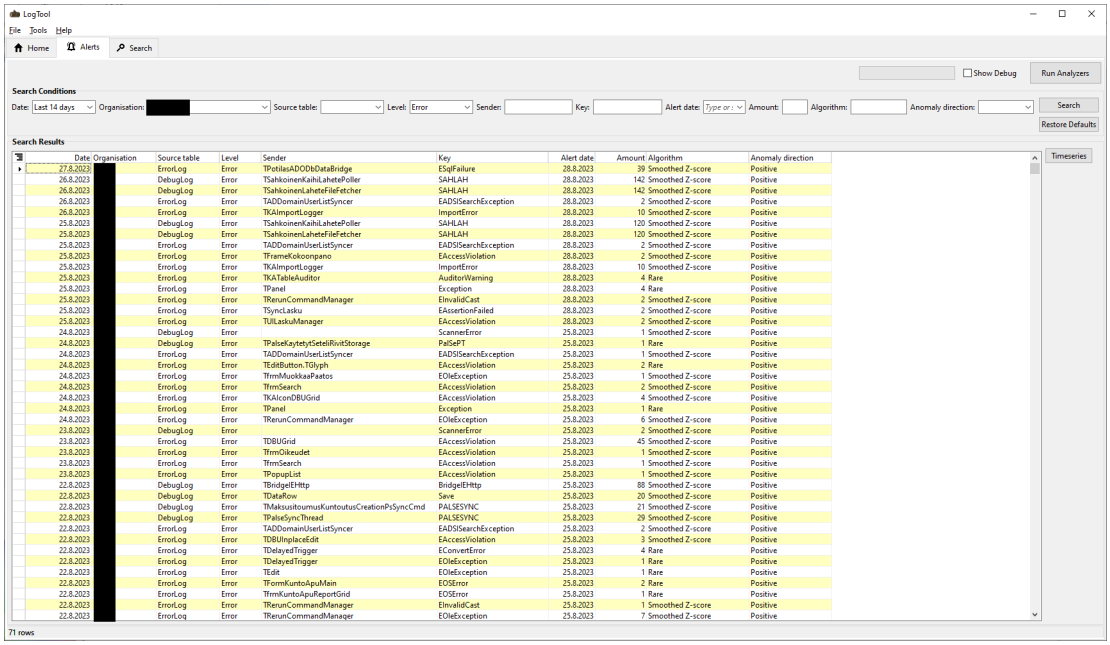

Figure 17: Alerts window of the LogTool.

The anomaly detector has been in test use for about 2 months, and its results seem very promising. When testing the functionality, the algorithm detects point and collective anomalies with good accuracy and their reporting in the alert window is clear. However, the algorithm does not take into account the contextual type of anomalies, but the modular implementation of the detector allows for the easy addition of its implementation to the detector in the future if the need arises. Thus, we conclude that LogTool achieves the requirement of automatic anomaly detection.

4.2.4 Data security

The fourth and final requirement for the project was that the log data could not be saved outside of the customer's database. Data security must also be considered in general, as the tool handles real-life log data created by many customers. LogTool takes this into account by directly fetching data from the customer's database and storing log data temporarily in memory. The login process of LogTool is facilitated through the API of centralized password management software, eliminating the need to develop a new authentication method, which further enhances security. The implementation has worked perfectly during the testing period, confirming the requirement has been accomplished.

4.3 Demo cases

In this section, we assess LogTool through real-life demonstration cases that showcase its practical application and performance. These demonstrations provide valuable insights into the functionality of LogTool and its ability to address real-world logging and analysis needs.

4.3.1 Case: Verifying the functionality of a new feature

In this case, a new functionality was applied to Effector and we needed to verify the correct functionality of the changes using LogTool. This was a big change to a class called 'TDefaultTarraManager', which is responsible for printing labels, among other things. The first thing we need to do is login to LogTool and select the organizations where the new functionality was to be deployed. If, for some reason, the organizations are not known, we can select all organizations.

After login, there are two ways to proceed in LogTool to determine the functionality of the new feature. In the first approach, the 'TDefaultTarraManager' class name can be searched in the alerts search window to inspect whether the anomaly detector has detected the corresponding anomaly. In the second approach, the class name can be directly searched in the search window, which in turn displays the histograms depicting the evolution of the respective log event.

As we can see in Figure 15, when searching for 'TDefaultTarraManager' using the sender filter in the alerts search window, it is evident that three anomalies have occurred: two with the key attribute 'EPrinter' and one with 'EAccessViolation'.

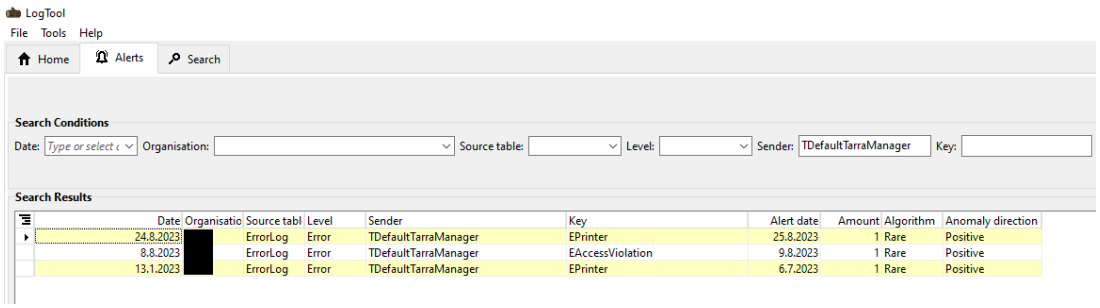

Figure 18: Searching specific log entries from the alerts.

Each of the anomalies has been detected by the rare anomaly detector, suggesting that there have not been additional log entries related to 'TDefaultTarraManager'. This can be further confirmed by searching for the specific log entry in the search window and analyzing the time series through a histogram. On the alerts page, a button labeled 'Time series' enables you to open the chosen alert in the search window and review the time series.

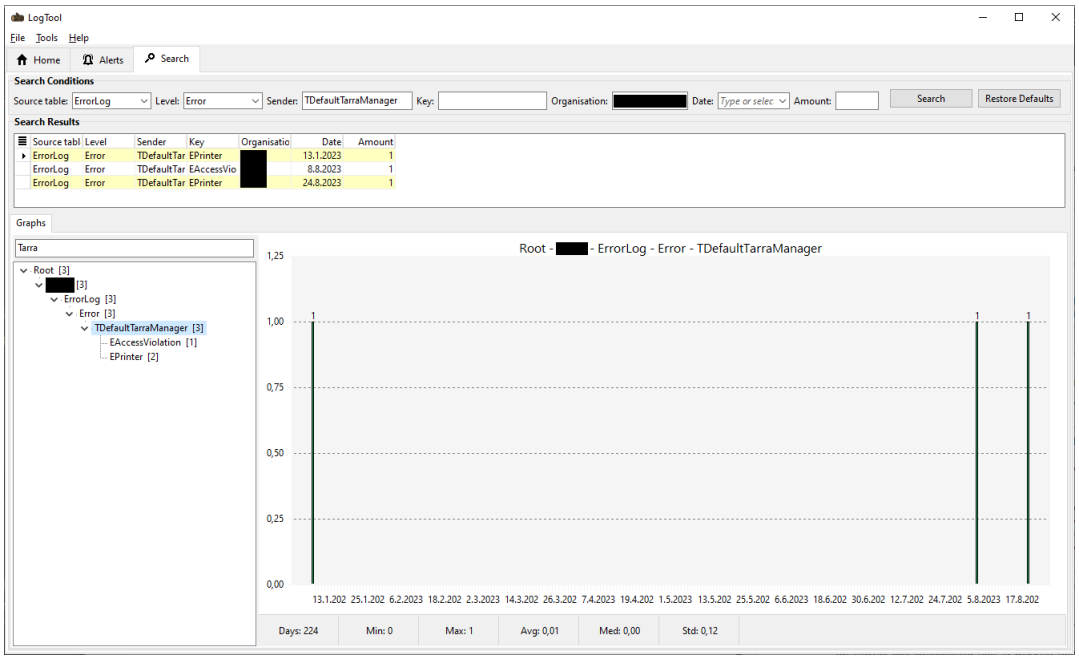

Figure 19: Searching specific log entries on the search window.

As we can see in Figure 19, it can be verified that there have only been three data entries related to 'TDefaultTarraManager', each of which has been classified as an anomaly. According to the key attribute of the log entries, two of them are printer errors, suggesting that there has been a single error in the customer's hardware but it has been resolved. This may have been due to a configuration problem with a printer or missing drivers. The access violation error is more serious and usually means that the code tries to refer to a freed instance. However, this has occurred only once, indicating that it is not a recurring problem that requires immediate attention. In such instances, the situation can be monitored in the future using LogTool to ensure that the error does not reoccur.

4.3.2 Case: Weird spikes on the log data

In addition to finding anomalies by anomaly detector, LogTool provides the capability to examine the log data at a general level. In this case, we examined the entire data set for one organization, and we investigated any anomalies found in more detail. A general log data study provides a holistic picture of the nature of the log data, which helps to detect anomalies and gives an overview of the performance of the Effector.

As we can see in Figure 20, when inspecting the top 5 most common log keys on the pie graph on the dashboard, it becomes apparent that $95,8\,\%$ of the log keys are 'HL7Service.' The second most common log key is 'Login' with $1,8\,\%$, indicating that there are significantly more log events of the 'HL7Service' key than all other keys combined.

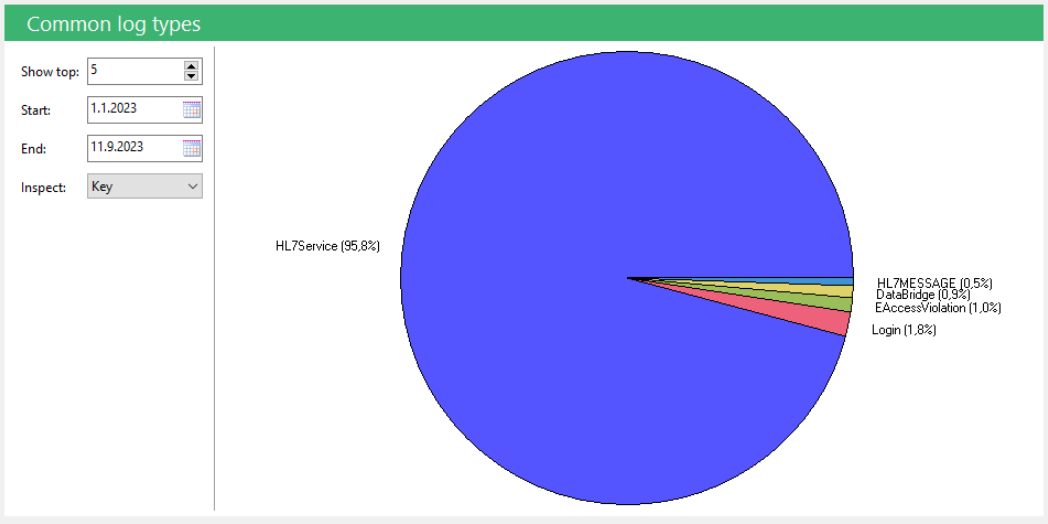

Figure 20: Pie graph of the top 5 most common log keys.

From the pie graph, we can see directly that in general there is wide variation in log data due to possible large outliers. This could be verified by examining the log data with a histogram to obtain time-dependent information. By searching with the key filter with the value 'HL7Service' on the search window presented by Figure 21, we can see from the rendered histogram that there are large spikes between February 20, 2023, and March 20, 2023. From the bottom status bar, we can see that the maximum spike has been 26 349 'HL7Service' log entries per day, which is about 130 times greater than the medium of the time series which is 209.5. The results indicate an unusual event during that period in relation to the normal operation of Effector.

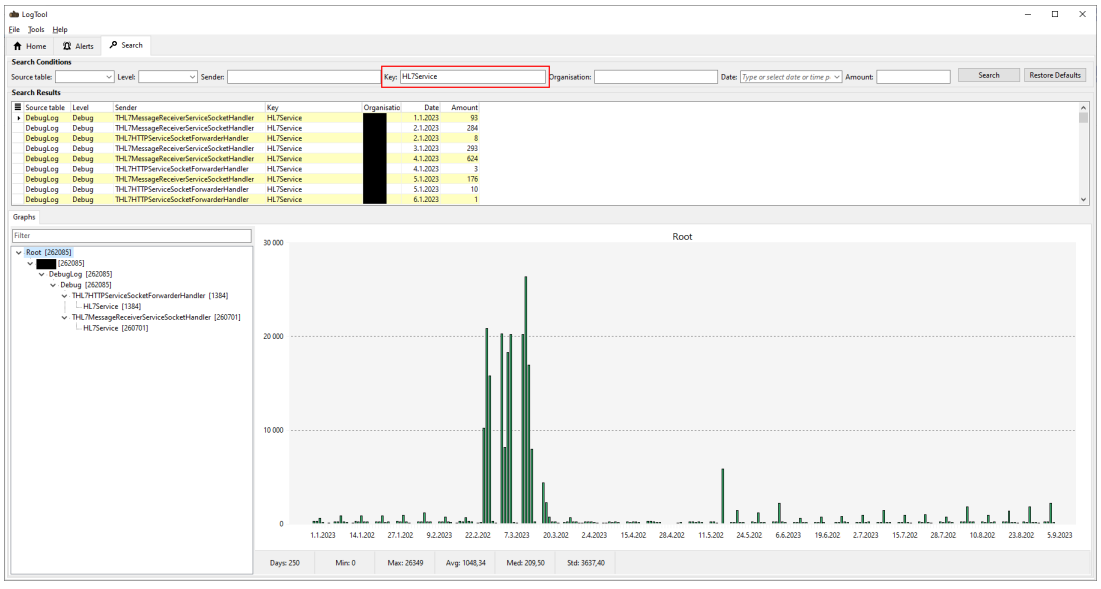

Figure 21: Searched key value 'HL7Service' displayed as a histogram on the search window of LogTool.

From the left tree view, we can see that all the 'HL7Service' log entries originate from the 'DebugLog' log table and are labeled as 'Debug' level. This indicates that it is not error log data but rather log data added for debugging purposes. Therefore, the presence of the log entry for the 'HL7Service' key does not necessarily signify an error. From the tree view, we can also see that all log entries originate from the 'THL7MessageReceiverServiceSocketHandler' and 'THL7HTTPServiceSocketForwarderHandler' methods.

Health Level Seven (HL7) is an organization responsible for establishing standards in the healthcare industry, and according to HL7 International [36], HL7 is supported by more than 1,600 participants from over 50 countries. The HL7 standard is used in Effector to transfer patient data such as address, municipality, and date of death from the patient information system to Effector. The previously mentioned HL7 methods are responsible for receiving and sending patient record requests.

Upon further examination of the log messages from the log entries during the spikes, we noticed that there were a large number of patient-generated commands that were typically logged as debug entries. This suggests that the client has imported a lot of data in bulk over a short period, either due to a migration of the application environment, a new version deployment, or other similar purposeful generation of patient-creation messages for the entire patient population of that organization. When all the large spikes are summed together, the result is approximately equal to the number of patients in that organization, suggesting that these spikes are part of the normal operation of Effector.

Here we can see how the decision of the programmer has a big impact on the nature of the log data. The programmer had decided to log debug data from patient creation (with a good reason) and this showed up as a major anomaly when the customer decided to do a mass data transfer. Although it was not a bug in the program, it was an anomalous event that is important to detect to know that Effector is working correctly under heavy load.

It is also worth noting that the log message was very useful in solving the problem. However, LogTool removes the log messages from the log entry, as parsing them was found to be an error-prone process, so the log messages had to be looked up directly in the database. This shortcoming needs to be taken into account in the further development of LogTool.

4.3.3 Case: Detected anomaly after version release

The release of a new version of the software is an important time in software development, as this is when new functionality is made available to the customer. The release of new functionality also means a higher probability of anomalies, as shown in previous Figure 13. The sooner bugs can be fixed after the version release, the better for the customer. In this demo case, LogTool was used after one week of the major version release to see if anomalies were found in the logs related to the release of the new version of Effector.

The first thing we can do is check the dashboard on the LogTool home page and look at the graph of recent alerts in the last 14 days, as shown in Figure 22. The version release was done on September 10, 2023.

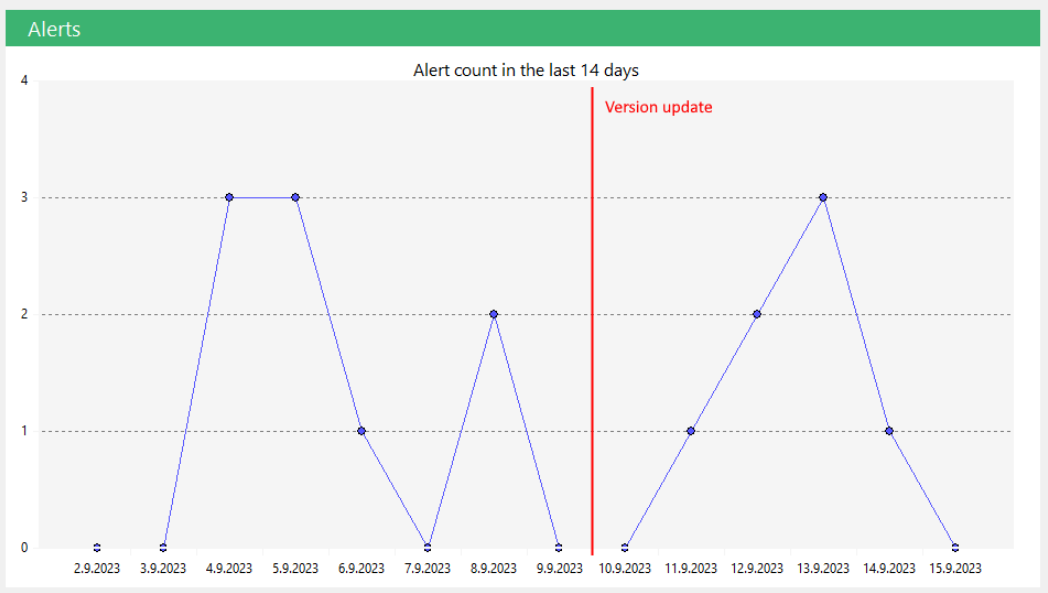

Figure 22: Screenshot from the dashboard on the LogTool. The graph shows the evolution of alert counts over the last 14 days. A red line has been added to the graph to indicate the date of the version update.

As we can see in Figure 22, the number of alerts has been steadily increasing since the version update, after which the number of alerts has decreased. However, after the release, the magnitude of the alerts does not differ from the previous week, so the graph suggests that nothing unexpected has happened. This can be further verified by looking at the alerts in more detail in the alerts window.

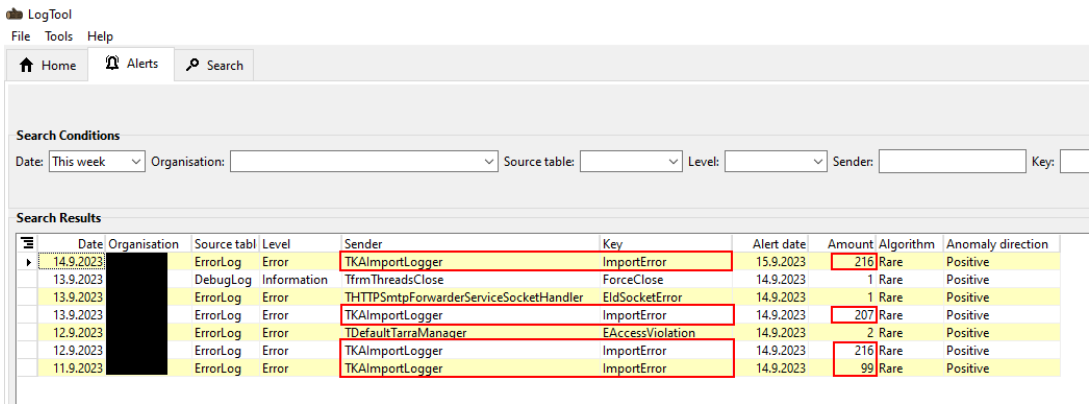

Figure 23: Search results of the alerts window. The alerts are searched using the filter 'Date' with the value 'This week'.

In the alerts window, we searched for all alerts from the past week and the search returned the alerts, which are shown in Figure 23. We can see from the results that the same import error has occurred on four consecutive days with high 'Amount' values (marked as red in Figure 23). The source log table of the log entry in question is 'ErrorLog' and its log level is 'Error', which indicates that errors have occurred during the normal execution of the program. The rare algorithm detected the errors, suggesting that this specific error normally occurs infrequently. We can verify this by examining the time series of the log entry in question using a histogram in the search window presented in Figure 24.

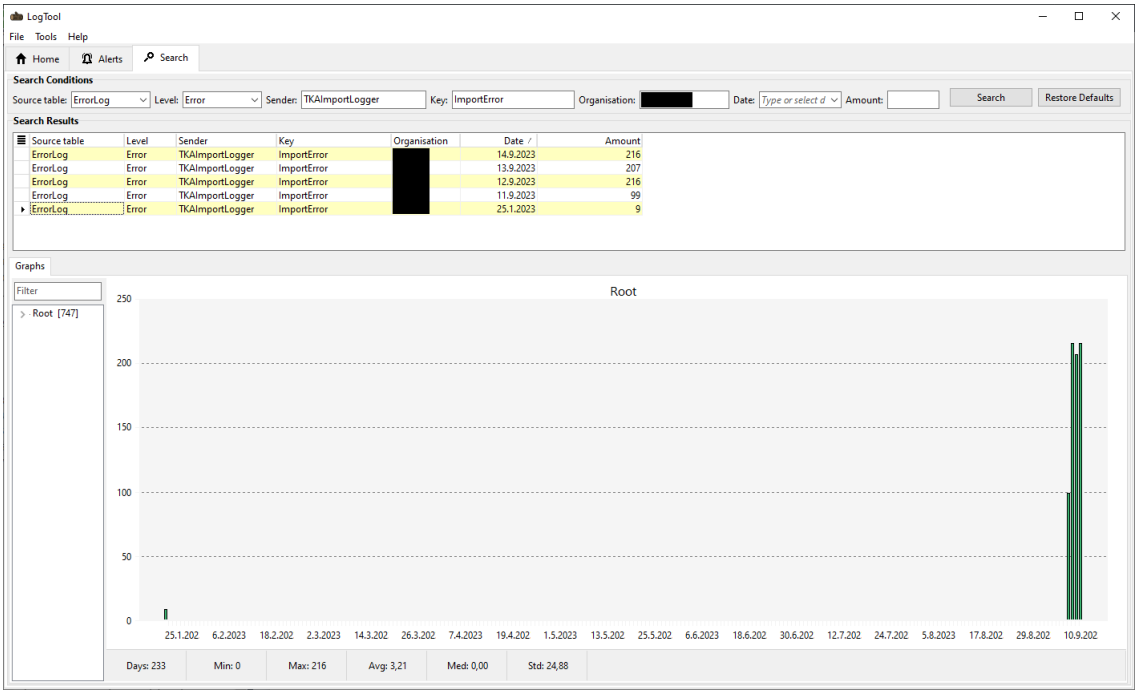

Figure 24: Investigating the time series of the import error log, which has been detected after version release.

As evident from the histogram in Figure 24, significant errors began to emerge after the release of the new version on September 10, 2023. The first occurrence of the error dates back to January 25, 2023, but since then, it has consistently manifested only after the introduction of the new version. The findings clearly suggest that functionality has been added or modified in the new version release that has caused this anomaly. The case requires further investigation and possible fixes.

While we investigated the log messages more thoroughly, we discovered that an automatic data import from a patient data system had been failing since September 11, 2023. The importer is responsible for importing diagnoses, municipalities, languages, and other data into Effector. Upon locating the error, it became evident that the issue originated from the customer's end. Consequently, the customer was informed of the situation to facilitate the necessary correction on their part.

Without the automatic anomaly detection provided by LogTool, this error would not have been discovered as quickly as it was in this case. The problem would only have been discovered when the error had manifested itself in a large enough way to be noticed by the developer or customer. In such a scenario, correcting the error could have been more challenging, given its prolonged impact. From this, we can conclude that LogTool has already proven its tangible value during the testing phase, demonstrating that automated log analysis enhances the log analysis process in Effector.

5 Conclusion

In this section, we present the conclusions drawn from the results obtained through the implementation of LogTool and their relevance to the original research question. We also reflect on the outcomes of LogTool and discuss potential future improvements. The research question of the thesis was:

RQ: How to improve log data analysis in Effector?

5.1 Answer to the research question

In this thesis, we found that we can present log data in various formats. However, logs often share common features, such as the header part which contains all the metadata of the log entry, and the log message part which contains more detailed information about the log event in question. Both the literature and the results obtained from LogTool reveal that the nature of log data is significantly shaped by the decisions of programmers. It is the responsibility of the programmer to decide where in the code to put the logging statement and what the log message should contain. Ultimately, the log format used is also the responsibility of the programmer. Thus, the first way to improve log analysis in Effector is to clarify and standardize log data collection and format to make log data as homogeneous as possible. This will further assist the analysis of log data both manually and automatically.

Another way in which log data analysis can be made more efficient is by reducing the need for manual analysis. This can be done by an automatic log analysis software. The thesis presented methods and commercial tools to perform automatic log data analysis, but due to customer constraints, we implemented the log analysis tool, LogTool, from scratch. The implementation process was presented and design decisions were justified by the literature and requirements.

We implemented an anomaly detector in LogTool to successfully detect anomalies. Manual log data analysis was facilitated by log data visualization, as well as a search window, which functionality was demonstrated through demo cases. The results showed that the program could be used with real-life data, which could be retrieved from multiple organizations and log data sources simultaneously.

5.2 Future improvements

LogTool is being extensively rolled out within Polycon, and as a result, it requires some fine-tuning to ensure maximum usability. One of the most noticeable areas for improvement in LogTool is the time delay when starting the software and updating UI components when multiple organizations are inspected simultaneously. This suggests that LogTool's multi-table manager needs optimization, which could involve implementing data caching, for example. The UI of the rendered graphs is also out of date, so these could be improved with a newer graph library or by adding anti-aliasing to the graph rendering.

In the data processing phase of LogTool, we made a deliberate choice to exclude the log message part of log entries. We found that parsing log messages was an error-prone and computationally demanding process. However, the log message contains valuable information about the operation of the software that could be valuable in determining what caused the anomaly. For this purpose, we could implement a log message parser in LogTool, or alternatively, we could add a new window where raw log data could be searched using the multi-table manager. This would allow a more accurate locating process of the error in the code, which would further improve log analysis in Effector.

As mentioned in the evaluation part of the thesis, the anomaly detector is only capable of detecting point and collective anomalies. However, the modular structure of the detector allows new algorithms to be added to the implementation, so future research will focus on the detection of contextual anomalies. The detected anomalies could also be sent as a report message, for example to the email of the person in charge of LogTool. In this case, LogTool could be used only when there is a real need for it. For this purpose, the functionality should be added to LogTool to run the software in the background, so that the anomaly detector is run at regular intervals and the report is sent according to the detected anomalies. To address identified needs, we implemented LogTool in a modular way, making it straightforward to incorporate additional functionality.

5.3 Closure

LogTool will be used alongside Effector in the future, and the true value of the program can only be determined after extensive use. However, LogTool showed real benefits during the development phase, as it was able to detect anomalies that would not have been detected otherwise. LogTool also automated log analysis in Effector, which improved the debugging process. In conclusion, the LogTool achieved all the requirements assigned in the introduction and it received good feedback from colleagues who will be using LogTool in the future. In the end, we can therefore conclude that LogTool improved the log analysis of Effector.

References

[1] Zhu J, He S, Liu J, He P, Xie Q, Zheng Z, and Lyu M. Tools and Benchmarks for Automated Log Parsing: *2019 IEEE/ACM 41st International Conference on Software Engineering: Software Engineering in Practice (ICSE-SEIP)*, 2019, pp. 121-130, DOI: 10.1109/ICSE-SEIP.2019.00021.

[2] Splunk Inc. [online material], Referred to [04.08.2023], Available at: https://www.splunk.com/.

[3] Elastic Stack. [online material], Referred to [04.08.2023], Available at: https://www.elastic.co/elastic-stack.

[4] LogRhythm Inc. [online material], Referred to [04.08.2023], Available at: https://logrhythm.com/.

[5] Polycon Oy [online material], Referred to [13.07.2023], Available at: https://www.polycon.fi/.

[6] Zhang X, Xu Y, Lin Q, Qiao B, Zhang H, Dang Y, Xie C, Yang X, Cheng Q, Li Z, Chen J, He X, Yao R, Lou J, Chintalapati M, Shen F, and Zhang D. Robust Log-Based Anomaly Detection on Unstable Log Data: *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*, 2019, pp. 807–817, ISBN: 9781450355728.

[7] El-Masri D, Petrillo F, Guéhéneuc Y, Hamou-Lhadj, and Bouziane A. A Systematic Literature Review on Automated Log Abstraction Techniques: *Information and Software Technology*, 2020, vol. 122, ISSN: 0950-5849.

[8] Ryciak P, Wasielewska K, and Janicki, A. Anomaly Detection in Log Files Using Selected Natural Language Processing Methods: *Applied Sciences*, 2022, vol. 12, no. 10, ISSN: 2076-3417.

[9] Tellis V, and D'Souza D. Detecting Anomalies in Data Stream Using Efficient Techniques: A Review: 2018 International Conference on Control, Power, *Communication and Computing Technologies (ICCPCCT)*, 2018, pp. 296-298, DOI: 10.1109/ICCPCCT.2018.8574310.

[10] Gogoi P, Bhattacharyya D, Borah B, and Kalita J. A Survey of Outlier Detection Methods in Network Anomaly Identification: *The Computer Journal*, 2011, vol. 54, no. 4, pp. 570-588, DOI: 10.1093/comjnl/bxr026.

[11] Lindemann B, Maschler B, Sahlab N, and Weyrich M. A survey on anomaly detection for technical systems using LSTM networks: *Computers in Industry*, 2021, vol. 131, ISSN: 0166-3615.

[12] He P, Zhu J, He S, Li J, and Lyu M. Towards Automated Log Parsing for Large-Scale Log Data Analysis: *IEEE Transactions on Dependable and Secure Computing*, 2018, vol. 15, no. 6, pp. 931-944, DOI: 10.1109/TDSC.2017.2762673.

[13] Xu W, Huang L, Fox A, Patterson D, and Jordan M. Detecting Large-Scale System Problems by Mining Console Logs: *Proceedings of the ACM SIGOPS 22nd Symposium on Operating Systems Principles*, 2009, pp. 117-132, DOI: 10.1145/1629575.1629587.

[14] Fu Q, Lou J, Wang Y, and Li J. Execution Anomaly Detection in Distributed Systems through Unstructured Log Analysis: *2009 Ninth IEEE International Conference on Data Mining*, 2009, pp. 149-158, DOI: 10.1109/ICDM.2009.60.

[15] Tukey J. Exploratory Data Analysis (1^{th} ed.): *Pearson*, 1977, ISBN-10: 0201076160.

[16] Hodge V, and Austin J. A Survey of Outlier Detection Methodologies: *Artificial Intelligence Review 22*, 2004, pp. 85–126, DOI: https://doi.org/10.1023/B:AIRE.0000045502.10941.a9.

[17] Ramaswamy S, Rastogi R, and Shim K. Efficient Algorithms for Mining Outliers from Large Data Sets: *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, 2000, vol. 29, no. 2, pp. 427-438, DOI: 10.1145/342009.335437.

[18] Sehar S, Aslam M, and Shaukat M. A REVIEW AND EMPIRICAL COMPARISON OF UNIVARIATE OUTLIER DETECTION METHODS.: *Pakistan Journal of Statistics 37.4*, 2021, vol. 37, no. 4, pp. 447-462.

[19] Brakel J. Robust peak detection algorithm using z-scores: *Stack Overflow* [online material], Referred to [20.07.2023], Available at: https://stackoverflow.com/questions/22583391/peak-signal-detection-in-realtime-timeseries-data/22640362#22640362 (version: 2020-11-08).

[20] Perkins P, and Heber S. Identification of Ribosome Pause Sites Using a Z-Score Based Peak Detection Algorithm: *2018 IEEE 8th International Conference on Computational Advances in Bio and Medical Sciences (ICCABS)*, 2018, pp. 1-6, DOI: 10.1109/ICCABS.2018.8541902.

[21] Karanjit S, and Upadhyaya S. Outlier detection: applications and techniques: *International Journal of Computer Science Issues (IJCSI)*, 2012, vol. 9, no. 1, pp. 307-323.

[22] Zhang J. Advancements of outlier detection: A survey: *ICST Transactions on Scalable Information Systems*, 2013, vol. 13, no. 1, pp. 1-26.

[23] Lischner R. Delphi in a Nutshell (1^{th} ed.): *O'Reilly & Associates, Inc.*, 2000, ISBN: 1-56592-659-5.

[24] Embarcadero Technologies. Delphi Run-Time Errors [online material], Referred to [02.08.2023], Available at: https://docwiki.embarcadero.com/RADStudio/Alexandria/en/Delphi_Run-Time_Errors (version: 2016-1-4).

[25] Carasso D. Exploring splunk (1^{th} ed.): *CITO Research New York*, 2012, ISBN: 978-0-9825506-7-0.

[26] Chhajed S. Learning ELK stack: *Packt Publishing Ltd*, 2015, ISBN: 978-1-78588-715-4.

[27] Jayathilake D. Towards structured log analysis: *2012 Ninth International Conference on Computer Science and Software Engineering (JCSSE)*, 2012, pp. 59-264, DOI: 10.1109/JCSSE.2012.6261962.

[28] Ammar A, Mohd Z, Salama A, Aida M, and Mustafa H. A Case Study on B-Tree Database Indexing Technique: *Journal of Soft Computing and Data Mining*, 2020, vol. 1, no. 1, pp. 27–35, DOI: https://doi.org/10.30880/jscdm.2020.01.01.004.

[29] Bent S, Sleator S, and Tarjan R. Biased Search Trees: *SIAM Journal on Computing*, 1985, vol. 14, no. 3, pp. 545-568, DOI: 10.1137/0214041.

[30] Kavis M. Architecting The Cloud design decisions for cloud computing service models: *John Wiley & Sons, Ltd*, 2014, ISBN: 9781118691779.

[31] Lemmens E. Outlier detection in event logs by using statistical methods: *Master's Thesis, Eindhoven University of Technology*, 2018.

[32] Minjae K, and Hargrove L. A gait phase prediction model trained on benchmark datasets for evaluating a controller for prosthetic legs: *Frontiers in Neurorobotics*, 2023, vol 16, DOI: 10.3389/fnbot.2022.1064313.

[33] Catalbas M, and Dobrisek S. Dynamic speaker localization based on a novel lightweight R–CNN model: *Neural Computing and Applications*, 2023, vol. 35, no. 14, pp. 10589-10603, DOI: 10.1007/s00521-023-08251-3.

[34] Phiri B, Fèvre D, and Hidano A. Uptrend in global managed honey bee colonies and production based on a six-decade viewpoint, 1961–2017: *Scientific Reports*, vol. 12, no. 1, DOI: 10.1038/s41598-022-25290-3.

[35] Wen H, Dong X, Ma Y, and Nan J. The research of the databases connection methods in LabVIEW based on ADO: *2010 International Conference on Computer Application and System Modeling (ICCASM 2010)*, 2010, vol. 7, pp. 229-233, DOI: 10.1109/ICCASM.2010.5619387.

[36] Health Level Seven International. [online material], Referred to [14.9.2023], Available at: https://www.hl7.org/.

A Anomaly detection algorithms

Implementation of the rare anomaly detector and the smoothed Z-score anomaly detector. 'TTimeSeries' is defined as **array of Double** and 'TAnomalies' is defined as **array of Integer**. Listing 3 and 4 are implemented in the Object Pascal programming language.

Listing 3: Implementation of the rare anomaly detector.

```
function TRareAnomalyDetector.CheckAnomalies : TAnomalies;
var
  i : Integer;
begin
  SetLength(Result, Length(fTimeSeries));
  for i := 0 to Length(fTimeSeries) - 1 do
  begin
    if fTimeSeries[i] <> 0 then
      Result[i] := 1
    else
      Result[i] := 0;
  end;
end;
```

Listing 4: Implementation of the smoothed Z-score anomaly detector.

```
function TSmoothedZAnomalyDetector.CheckAnomalies : TAnomalies;
begin
  Result := StartAlgo(fTimeSeries, 30, 4, 0.5);
end;

function TSmoothedZAnomalyDetector.StartAlgo(
  AInput : TTimeSeries; ALag : Integer;
  AThreshold, AInfluence : Double) : TAnomalies;
var
  AFilteredY : TTimeSeries;
  AAvgFilter : TTimeSeries;
  AStdFilter : TTimeSeries;

  AInitialWindow : TTimeSeries;
  AMean, AStdDev : Double;
  ASlidingWindow : TTimeSeries;

  i : integer;
begin
  SetLength(Result, Length(AInput));
  AFilteredY := Copy(AInput);
```

```
  SetLength(AAvgFilter, Length(AInput));
  SetLength(AStdFilter, Length(AInput));

  AInitialWindow := Copy(AFilteredY, 0, ALag);
  MeanAndStdDev(AInitialWindow, AMean, AStdDev);
  AAvgFilter[ALag - 1] := AMean;
  AStdFilter[ALag - 1] := AStdDev;

  for i := ALag to Length(AInput) - 1 do
  begin
    if ( Abs(AInput[i] - AAvgFilter[i-1]) >
       (AThreshold * AStdFilter[i-1]) ) then
    begin
      if (AInput[i] > AAvgFilter[i - 1]) then
        Result[i] := 1
      else
        Result[i] := -1;
      AFilteredY[i] := AInfluence * AInput[i] +
        (1 - AInfluence) * AFilteredY[i - 1];
    end
    else
    begin
      Result[i] := 0;
      AFilteredY[i] := AInput[i];
    end;

    ASlidingWindow := Copy(AFilteredY, i-ALag, ALag + 1);
    MeanAndStdDev(ASlidingWindow, AMean, AStdDev);
    AAvgFilter[i] := AMean;
    AStdFilter[i] := AStdDev;
  end;
end;
```