Monitoring of a Cloud-Based IT Infrastructure

Master of Science in Technology Thesis
University of Turku
Department of Computing
Software Engineering
2021
Janne Sirviö

UNIVERSITY OF TURKU
Department of Computing

Janne Sirviö: Monitoring of a Cloud-Based IT Infrastructure

Master of Science in Technology Thesis, 75 p., 7 app. p.
Software Engineering
May 2021

The amount of software to maintain increases continuously. New systems are being built and old systems are not phased out at the same pace. The arrival of microservices, SaaS and platform businesses have increased the complexity of software and the requirements for availability of service. Software maintenance has become troublesome. In this thesis, monitoring is used to mitigate the problem.
Objective of this thesis is to build a monitoring system for a SaaS company that is suffering from the increased complexity. The research questions of this thesis ask: 1) How to build an effective monitoring system with OSS tools? 2) What are the important parts of the company's IT infrastructure that should be monitored? 3) How to detect anomalies from monitoring data to set threshold values for alerting in a scalable way? 4) Is it possible to predict the moment of time in the future, when the system is next going to face an anomaly, from the monitoring data gathered in the past?
Research consists of a literature review on the preferred monitoring methods, an interview on the employees of the company to figure out the parts of company's IT infrastructure that monitoring will benefit the most and implementation of a monitoring system for the company with OSS tools (Prometheus and Grafana). In addition, automated anomaly detection methods are introduced and predictive monitoring is discussed.
As a result of this thesis, three key areas were found in the company's IT infrastructure where the built monitoring system focused on. First area was sign-ins which use now centralized logs for easier problem tracking. Second area was sluggishness of certain services that was solved with server, application and database metrics. Third area was MyHealth questionnaires that were supposed to use Prometheus MongoDB query exporter to expose the changes in the database, but current version of the database didn't allow the plan to succeed. For automated anomaly detection several solutions were provided in the thesis and the most suitable for the company was OSS tool called Prometheus anomaly detector. Theory for predictive monitoring was found on AI playing chess.

Keywords: software monitoring, software maintenance, automated anomaly detection, predictive monitoring, monitoring process, Prometheus, Grafana

TURUN YLIOPISTO
Tietotekniikan laitos

JANNE SIRVIÖ: Monitoring of a Cloud-Based IT Infrastructure

Diplomityö, 75 s., 7 liites.
Ohjelmistotekniikka
Toukokuu 2021

Ylläpidettävien ohjelmistojen määrä lisääntyy jatkuvasti. Uusia ohjelmistoja julkaistaan eikä vanhoja poisteta käytöstä samaan tahtiin. Mikropalveluarkkitehtuurin, SaaSin ja alustaliiketoiminnan yleistymisen myötä ohjelmistojen monimutkaisuus ja saatavuuden vaatimukset ovat kasvaneet. Ohjelmistojen ylläpidosta on tullut hankalaa. Tässä lopputyössä monitorointia hyödynnetään ongelman lievittämiseksi. Tämän lopputyön tavoitteena on rakentaa monitorointijärjestelmä SaaS-yritykselle, joka kärsii monimutkaisuuden lisääntymisestä. Tämän työn tutkimuskysymykset kysyvät: 1) Miten rakentaa monitorointijärjestelmä avoimen lähdekoodin työkaluilla? 2) Mitkä ovat yrityksen ohjelmistojärjestelmän kohdat, joihin monitorointi kannattaa keskittää? 3) Miten havaita monitorointidatasta poikkeavuuksia, joita käyttää hälytysten raja-arvojen asettamiseen skaalautuvasti? 4) Onko kerätystä monitorointidatasta mahdollista ennustaa ajanhetki, jolloin järjestelmä tulee seuraavaksi kohtaamaan virheen?
Tutkimus sisältää kirjallisuuskatsauksen monitorointitekniikoista. Haastatteluosuuden, jossa yrityksen työntekijöitä haastateltiin monitoroitavien kohteiden selvittämiseksi. Toteutusosuuden, jossa yritykselle toteutettiin monitorointijärjestelmä avoimen lähdekoodin työkaluilla (Prometheus ja Grafana). Lisäksi työssä esitellään automaattisen virheiden havainnoinnin tekniikoita ja pohditaan keinoja toteuttaa ennakoivaa monitorointia.
Työn tuloksena määritettiin kolme avainaluetta, johon rakennettu monitorointijärjestelmä keskittyy. Ensimmäinen alue on kirjautumiset, joissa hyödynnettiin keskitettyä lokitusta helpottamaan virheiden selvitystä. Toinen alue on tiettyjen palvelujen hitaus, joiden selvittämiseen käytettiin palvelimen, applikaation ja tietokantojen metriikkaa. Kolmas alue oli Omavointi-kyselyt, joihin oli tarkoitus käyttää MongoDB query exporter nimistä ohjelmistoa, mutta tietokannan versio vesitti tämän suunnitelman. Automaattiseen virheiden havainnointiin työssä esiteltiin useita vaihtoehtoja, joista yrityksen näkökulmasta sopivin on avoimen lähdekoodin työkalu nimeltä Prometheus anomaly detector. Teoria ennakoivalle monitoroinnille löytyi tekoälyllä toimivasta shakki-robotista.

Asiasanat: ohjelmistojen monitorointi, ohjelmistojen ylläpito, automaattinen virheiden havainnointi, ennakoiva monitorointi, monitorointiprosessi, Prometheus, Grafana

Contents

List of acronyms

AI	Artificial Intelligence
API	Application Programming Interface
CPU	Central Processing Unit
DSR	Disease-Specific Register
HTTP	HyperText Transfer Protocol
ICA	Independent Component Analysis
IPla	Integration Platform
ITIL	Information Technology Infrastructure Library
JSON	JavaScript Object Notation
JVM	Java Virtual Machine
K8s	Kubernetes
ML	Machine Learning
OOMKiller	Out Of Memory Killer
OOPS-errors	Software errors that show the "Oops something went wrong" page for the user

OSS	Open Source Software
PCA	Principal Component Analysis
QA	Quality Assurance
QPS	Queries Per Second
RAM	Random Access Memory
RFC	Request For Comments
SaaS	Software as a Service
SLA	Service Level Agreement
SMS	Short Message Service
SNMP	Simple Network Management Protocol
SQL	Structured Query Language
TSDB	Time Series DataBase
UDP	User Datagram Protocol
XOR	Exclusive Or

1 Introduction

Since the beginning of the software industry, in 1960s, the amount of software to maintain has increased, when new systems are being built and old systems can't be shutdown at the same pace. In top of that, both the complexity of software systems, and the requirements for availability of service have raised, when microservices, SaaS (Software as a Service) and platform businesses have gained popularity in the past decade.

Monitoring is without a doubt one of the most effective tools for software maintenance. It is helping the maintenance of a large number of services with automation in problem detection. It is giving the solution for increased complexity by adding transparency. It is increasing the availability of service by forecasting the future behaviour of a system.

1.1 Monitoring in General

Monitoring means different things to different people. Some people think that monitoring is following server CPU status, others say it is reading application logs. Neither of these assumptions are wrong. Actually, both of the before mentioned things are part of the monitoring. The term monitoring contains a wide spectrum of things. Monitoring is profiling, finding all the tiniest details of software run-time information and analysing that information for the sake of finding a solution for a problem at hand. Monitoring is collecting metrics data and storing the metrics data into a database to learn new things about software behaviour. Monitoring is collecting logging data for the assistance in problem detection and problem solving. Monitoring is distributed tracing where problems in distributed systems can be solved with centralized logs.

Like it was mentioned before, monitoring is a broad concept and it contains many sub-fields. Monitoring process is no exception, it also contains several phases, which are:

- Data collection,
- data storage,
- visualization,
- analysis,
- alerting and recovering. [1]

These phases can be carried out by single service (monolithic monitoring system) or by multiple services (distributed monitoring system) [1]. Current trend is going towards distributed systems, but there is still many monolithic monitoring systems in production.

1.2 Why to Monitor?

Software will always work against it was supposed to and software will always have bugs. With monitoring, those bugs can be discovered earlier and solved in less time.

Monitoring also helps to solve complex problems that are usual in complex systems. For example, the performance issues are almost impossible to solve without monitoring. Monitoring can also be used in automated capacity planning, where resources will automatically be added to some part of the system if it is under heavy load.

Monitoring and well tuned alerting releases operation engineers time from checking and fixing problems that could be automatically fixed. Operation engineers can focus on doing something productive when they can rely on the monitoring system that will alert if there is a problem that requires human interaction.

At the end of the day, monitoring makes the software more transparent and predictable which shows in reduced problem solution times and it also reflects on the customer satisfaction, when customers use software that is working correctly.

1.3 Research Problem

Maturity of monitoring in a company can be divided into three phases: SaaS monitoring, OSS (Open Source Software) monitoring and Custom built monitoring [1]. In the SaaS monitoring phase, the company is using ready made SaaS tools for monitoring. Those tools are easy to operate and integrate as a part of existing system. In the OSS monitoring phase, the company is taking advantage of OSS tools to implement monitoring for their software environment. OSS tools provide more room for customisation than SaaS tools, but also require more configuration and time to get the monitoring working. In the last phase, in a custom built monitoring phase, the company builds monitoring service for their special purposes. Companies that are on the last phase of monitoring and use a custom-built system, are usually huge worldwide technology companies for which the available monitoring solutions are not fitting because of scalability or customization issues. It is common that the monitoring solutions that these huge companies build for their own use, are later open sourced and published for wider use. Reasons for open sourcing can be cost reduction and greater innovation capability [2]. One example of the monitoring tool that was initially built for the purpose of a single company and was later gained popularity as OSS tool, is Prometheus. Prometheus was originally built in SoundCloud to serve their expanded needs in monitoring. Prometheus is introduced in more detail in Section 2.4.

This thesis is made for the company named BCB Medical. BCB Medical is moving their monitoring solutions from SaaS tools to OSS tools. The main goal for this thesis, is to build the first version of a working monitoring system with OSS tools for BCB Medical. The first version is delimited to solve certain set of problems that are defined as a part of this thesis. The problem area is going to be from the perspective of the customer support team and focus on solving the problems that employ the customer support team the most. With monitoring solutions the customer support team of BCB Medical can success better in their job of maintaining the high customer satisfaction with quick problem solution times. To reach the goal, the following research questions have been defined to be solved in this thesis:

- **RQ1:** How to build an effective monitoring system with OSS tools?
- **RQ2:** What are the important parts of BCB Medical's IT infrastructure that should be monitored?
- **RQ3:** How to detect anomalies from monitoring data to set threshold values for alerting in a scalable way?
- **RQ4:** Is it possible to predict the moment of time in the future, when the system is next going to face an anomaly, from the monitoring data gathered in the past (a.k.a. predictive monitoring)?

1.4 Research Methods

For the first research question (RQ1) there is going to be used a literature review, where different kind of monitoring methods and monitoring data analyzing methods are introduced. Monitoring is really practical topic and there are not many scientific publications available in the field, so the literature review chapters will mainly be based on the monitoring books. However, for some smaller subjects in monitoring, like anomaly detection, there are scientific articles available and in those parts the articles will be used.

For the second research question (RQ2) there is going to be used a qualitative research method. In the research, all five members of the customer support team will be interviewed. The goal of the interviews is to find out the parts of BCB Medical's IT infrastructure that employ the customer support team or are problematic in other way. These parts are the definition of problem areas that monitoring will benefit the most.

In order to solve the third research question (RQ3) the information gathered in the literature review (RQ1) needs to be combined with the defined problem areas from the RQ2. This information will then be used to plan and build a monitoring system for BCB Medical that solves the defined problem areas. The data gathered by the monitoring system will be analysed and it will be proved that the anomalies on the monitoring data could be detected with machine learning, which can also be used for a scalable way to set threshold values for alerts.

The final research question, number four (RQ4), will be discovered with literature review and the theories will be discussed in conclusion. RQ4 also gives a good basis for further research on predictive monitoring.

1.5 Structure of This Document

The structure of this Thesis is divided into three phases. First phase is based on a literature and it contains Chapters 2, 3 and 4. The goal of the first phase is to gain a theoretical background to the research work and answer to the first research question (RQ1). The first phase follows the structure of monitoring process, that was mentioned earlier in Section 1.1. Second phase is about the research work, where the interviews and their results are documented (RQ2). The theories gained in the first phase are put into a real world test, when planning and implementing the monitoring system in BCB Medical's IT infrastructure. RQ3 will also be answered by telling how to set the threshold values for alerts in a scalable way, in the context of BCB Medical's environment. The second phase contains Chapters 5 and 6. Third phase is for the conclusion. Summary of the solutions for the research questions are presented and a few interesting topics related to the monitoring are discussed, including predictive monitoring (RQ4). Third phase contains Chapter 7.

In Chapter 2 monitoring data collection and storage methods are introduced. The Chapter gives brief introduction of different parts of IT infrastructure and of the items to monitor in those parts. In addition, in the final Section of the Chapter, significant monitoring tools are introduced. Significant in this context means that the monitoring tool is either going to be used in the business case of this Thesis or it was mentioned multiple times in the literature as a good or bad example of monitoring.

Chapter 3 is about visualization, anomaly perception and anomaly anticipation. In the beginning of the chapter data visualisation methods are introduced, what is the definition of a good dashboard. Next, the anomaly perception methods are compared to build a theoretical base for anomaly detection that is needed in scalable threshold setting. Last, proofs of anomaly anticipation, or predictive monitoring, are researched.

Chapter 4 describes alerting and recovery practices. What type of anomalies exist and which one of those should be alerted. In the end of the Chapter, ITIL's (Information Technology Infrastructure Library) process for incident management is went through to describe a good example of recovery from anomaly.

In Chapter 5 the business case is explained and the plan for the research work is documented. Research work will contain interview part and implementation part.

Chapter 6 contains the documentation of the interviews, the analysis of the interview results, plan for the implementation, the implementation and the scalable way to set threshold values for alerts. In addition, at the end of the Chapter further development of monitoring will be discussed in the form of cyber security and scalability issues in Prometheus. Also, Kubernetes-based monitoring is compared to server-based monitoring.

In Chapter 7 summary of the answers for research questions will be presented and predictive monitoring will be discussed by providing one theoretical approach to build a predictive monitoring system.

2 Data Collection and Storage

This Chapter covers the first two phases of the monitoring process that was mentioned in Chapter 1, data collection and data storage. Firstly, the different parts of IT infrastructure that monitoring data can be collected from are introduced in Section 2.1. Secondly, different monitoring methods and techniques are being compared in Section 2.2. Thirdly, storing of monitoring data is handled in Section 2.3 including introduction and comparison of different database types. In the last Section of this Chapter, Section 2.4, three monitoring tools that use techniques mentioned in earlier Sections, are introduced.

2.1 Monitoring Subjects

This Section introduces the common parts of IT infrastructure and the metrics that can be monitored in those parts. The IT infrastructure is divided by its concrete building blocks: server, application, network and database. Monitoring subjects introduced in following subsections can be found in compact form in Table 2.1.

2.1.1 Server

To get a better visibility on a server important metrics to monitor are introduced in this Subsection. The example destinations for introduced metrics are from devices that have Linux based operating system. Same metrics are available on devices that are using other operating systems as well, but the destination of a metric may

Part	**Subject**
Server	CPU
	Memory
	Network
	Storage
	Load
Application	Application instrumentation
	Framework support for monitoring
Database	Connections
	Queries per second (qps)
	Slow queries
	Server metrics
Network	Incoming and outgoing messages
	Agent-based/agentless monitoring

Table 2.1: Parts of IT infrastructure and monitored subjects in those parts.

vary. Most of the introduced metrics in this Subsection are fetched by default in monitoring tools. For example, Prometheus uses Node exporter [3] to reveal all needed server metrics. However, when getting to know in monitoring it is important to understand what happens under the hood and why certain metrics are monitored.

CPU (Central Processing Unit) status is usually described with CPU utilization metric. CPU utilization is a common metric to see at monitoring dashboards and usually one of the first metrics that comes to mind when considering server monitoring. It is also the most misinterpreted metric. Against common misconception, CPU utilization metric doesn't really tell the used capacity of CPU. Instead, it tells also the time that CPU waits for I/O devices, like memory. [4] For example, if server memory is exhausted, the CPU utilization is constantly at 100%, while really only

10% of the CPU capacity is used. For that reason, monitoring of CPU utilization metric alone doesn't really give any insight on the CPU status. Therefore, CPU status needs to be monitored together with I/O metrics or through other metrics. One option to monitor on is instructions per cycle (IPC) metric. It tells how many instructions CPU has completed in a clock cycle. From that metric it can be clearly seen, on what level the CPU is performing, when maximum IPC value is known.

Memory is monitored through memory metrics that describe total, used and free memory. On Linux based devices those values can be found in /proc/meminfo file. When memory runs out, the server will automatically release memory by killing resource exhaustive processes, which can lead to unexpected results. OOMKiller (Out Of Memory Killer) chooses the process to be killed by maximizing the released memory and minimizing the amount of processes to be killed and the importance of a process [5]. However, sometimes the process that is unimportant for a server might be important for a user, so memory issues should be handled before the OOMKiller steps into the picture. The easiest way to notice that server has ran out of memory is to search for the existence of OOMKiller in server logs [6].

Server network is monitored through sent and received packages. On Linux based devices the information is available in /proc/net/dev file. If the count of sent and received packages remains unchanged for a long time, it can be deduced that there is some problem in the network connection. [1]

Storage space is monitored through read and write activity on a disk. On Linux based devices the information is available in /proc/diskstats file. Sudden rise or drop of disk activity is usually a sign of an anomaly in a system.

Server load is a measurement that tells how many processes are waiting to be served by the CPU. The load is presented with three values: 1 minute average, 5 minutes average and 15 minutes average. It is advised to not make any interpretations on single load value [1], because load is effected by not only the CPU delay but also the delay in I/O devices [7]. Instead, the load values should be interpreted together. If all three load values are zero, the system is on idle. If 1 minute load value is higher than 5 and 15 minute values, the load is increasing. If 1 minute load value is lower than 5 and 15 minute values, the load is decreasing. [8] The load information is available at /proc/loadavg file on Linux based devices.

2.1.2 Application

Application monitoring is a field with a lot of freedom in implementation of monitoring, yet it is the part that is usually put aside or not implemented at all [1]. One reason for this might be that application monitoring is not a component that you can glue on the side of the system. It needs to be implemented in the source code of a system. However, when the gained benefits and the made sacrifice are being compared it is clear that application monitoring is a highly productive thing to do and a thing that is gaining popularity in the modern software [1].

One method of application monitoring, is to add to the source code of a software parts that save information about success rate and running time of methods. For example, a simple method that handles sign ins, could increment a counter value always when someone tries to sign in and information about the sign in being successful or not. This data can then be exported in a structured log file or in a time-series database from where it can be forwarded to the monitoring system. [1]

Application monitoring can be implemented from zero by just simply building everything from ground up or by using a ready-made framework. One good open source option for the framework to use is StatsD. It is based on a client-server architecture and implemented with the Node.js programming framework. StatsD can be used with multiple programming languages. Basically, StatsD client plugin needs to be imported to the source code of the software and then metrics can be gathered with plugin methods. The client will send the data with UDP (User Datagram

Protocol) to the back-end server as constructed log input. The back-end will then calculate different values from data and send them forward to the visualization tool like Graphite. [9] [10]

It is said in the book Practical monitoring [1] that application monitoring should be delivered with the application, so it would be in the same server or part of the application. This is because the application monitoring system should be kept up to date with the application to make sure that it is functioning properly.

2.1.3 Database

First thing to monitor in a database is usually connections [1]. Connections describe the amount of clients connected to the database. From the connections number the rough estimate of usage level of a database can be seen.

However, every client can put different load on the database. The more exact load value of the database is presented with QPS (Queries Per Second) value. QPS shows how many queries were run in one second.

Sometimes the poor performance of a database is not about the huge amount of queries. Queries may also be slow. First step, when enhancing the database containing slow queries, is finding them. Slow queries can be found in the database log file where each query is logged with execution time. [1]

One thing worth to remember when doing database monitoring is that databases also run on servers, so it is good to monitor the metrics of a database server to catch malfunctioning disks for example. [1]

2.1.4 Network

In network monitoring there are basically two approaches that one can take, agent-based or agentless monitoring. The agent-based approach means that on every device of the network there needs to be installed a piece of software that monitors incoming and outgoing packets, the downside of the solution is that it is not a scalable one. Upside of the agent-based approach is that there are no limits in the scope of what can be monitored. On the other hand, in the agentless approach the APIs (Application Programming Interface) of a device are used for the network monitoring. This approach limits the possibilities of monitoring to the options that device manufacturer has provided. However, the agentless approach is much more scalable solution. [11]

The agentless network monitoring is based on an SNMP (Simple Network Management Protocol). The SNMP is an old protocol and it was widely introduced for the first time in RFC 1067 [12] (Request For Comments) in August 1988. The SNMP is based on the UDP protocol and it uses two ports (161 and 162) in data transfer. One of the ports is used for outgoing data and the other is used for incoming data. All servers and computers have support for the SNMP, which means they can work as an agent or a manager. The agent is a device where network monitoring data can be pulled from, and the manager is a device that is fetching the data. [1]

2.2 Monitoring Methods

In this section, a different kind of monitoring techniques are introduced. The differences and similarities of those techniques are also being compared.

2.2.1 Monolith or Distributed Monitoring System

Monitoring has five phases, data collection, data storage, visualization, analytics and alerting, like was mentioned in Chapter 1. In a monolith monitoring system, all these phases are handled by one application or a service. In the past, most monitoring systems were following monolith architecture. Recently, the trend has been going towards distributed monitoring systems, where each or some of the phases of monitoring are done in different services. [1]

In general, a distributed monitoring system has the same advantages as the microservices. Smaller units are easier to scale, easier to build and maintain. Different technologies can be used in different units. The system as a whole is more resilient to errors, if some part of the system fails the other parts can still function. It is also easier to change parts of the system when they follow distributed architecture. [13]

2.2.2 Pull or Push Based Monitoring

In monitoring there are two ways to get the information from a client system. Either the client system can send the monitoring information to a monitoring system or the monitoring system can ask this information from the client system. To put it in a nutshell it is about which one makes the initiative. If the client system makes the initiative, it is called push based monitoring and if the monitoring system is the one who makes the first move, it is called pull based monitoring.

Both of these monitoring methods have their own advantages and disadvantages. In the book Art of monitoring from James Turnbull, a push based monitoring system is built from a scratch with a monitoring tool named Riemann. In the book, many advantages of the push based monitoring system are listed. It is said that the push based monitoring system will scale better than pull based because there is no need to configure each of the new client systems to the monitoring system. Instead, the location of the monitoring system needs to be configured to the client system and the monitoring information will start to flow from the client to the monitor. [14]

Another point of view to the matter is provided by a widely known open source monitoring tool, named Prometheus. Prometheus is counting on the pull based method in monitoring. One of the founders of Prometheus, Julius Volz, wrote an article to the Prometheus website where he denies all the scaling issues of the pull based monitoring system. In the article, it is said that the part where the monitoring system is asking information from a client system, won't waste more resources compared to push based monitoring system. It is also said that time is not saved in the configuration phase because if the monitoring system is built to be reliable, it is required to configure the clients to the monitoring system as well. [15]

In both of the texts, writers mention pull based monitoring system named Nagios and agree on the fact that Nagios doesn't scale well. It may be that in the Art of monitoring book the scalability problems of Nagios are generalized to the problems of all pull based monitoring systems. However, in the article on the Prometheus website, it is said that Prometheus is not working like Nagios, and they have even defined the problems that hinder the scalability of Nagios and these problems doesn't exist in Prometheus.

2.2.3 Black Box or White Box Monitoring

A black box monitoring refers to the monitoring technique where the system is observed only from outside, without knowing how the monitored system really works. With black box monitoring the symptoms of an anomaly are easy to notice in some cases, but it won't provide any help on finding the reasons for the problem. [16] Usually with black box monitoring it is reasonable to monitor if the service is available and how quickly it is responding. These type of monitoring solutions are available as a service. The upside of those services is the ease of use and security. Since, it requires only the configuration of a system that needs to be monitored, and the monitoring system itself doesn't need any access to the internal structure of a system under monitoring, it is easy to mobilize and there is no need to worry about security issues.

Downside of the black box monitoring is the lack of useful information on how to solve the problem at hand. The lack of insight on the system prevents alerts from being as good as they could be. Alerts in general and what kind of information they should contain, are discussed in more detail in the chapter 4.

White box monitoring is the preferred way of monitoring nowadays [16]. Mostly because it enables better transparency of a monitored system. White box monitoring requires more work to mobilize but it will pay back as shorter problem solution times and faster or even predictive anomaly detection.

2.2.4 Metrics or Event Logs

Metric is a quantitative data type that is used to measure the performance of an object. Metrics can be a counter type or a gauge type. The counter type data means a value that continuously increases. For example, the value that tells the amount of visitors in a website is a counter type value. The problem with the counter type data is the maximum value. At some point, every counter will reach its maximum value and need to be reset. The gauge type data is a data that describes point in time value. For example, a processor utilization rate is a gauge type value which tells how much load a processor is facing at the given time. The problem with a gauge type data is that, it doesn't tell anything about a previous state of an object that the data was fetched from. Most of the data from monitoring is usually gauge type [1].

Log data is specified information about system run-time variable values and errors in text format. Log data usually contains a timestamp that tells when the logged incident has happened. It is also reasonable to include in the logged text the location where the logging was made. Log data can be structured or unstructured. Structured in this context means that the log data is easy to read by machine. However, unstructured log data can later be transformed as a structured one. Usually structured log data is in JSON (JavaScript Object Notation) format, which is easy to parse by machine. [1]

If we compare metrics with log data, the later mentioned is more specific and it gives more precise information about system behaviour [17]. On the other hand, it is also more resource exhausting. Writing the log files demands CPU power and then keeping old log data demands data storage space [16]. It is true that older log files can be compressed, but the logs are still spending a lot more space than metrics. Especially, when the older metrics can be resampled. The resampling of metrics means that if certain gauge type metric was originally recorded with 1 minute interval, the average of five 1 minute metric values can be calculated and the metric data will compress into the space that is 1/5 of the original space [1]. The smallest changes in data will be lost, but the trend is still there and it is all that is needed from older monitoring data. Storage and compression methods of monitoring data will be introduced in the Section 2.3.

2.2.5 Distributed Tracing

Distributed tracing is a way to track requests throughout request life span. It is especially useful when figuring out problems in distributed systems, like systems built with microservice architecture. From the trace, operation engineer can see in a single screen what services the request has reached and how long it has took for each service to handle the request. Distributed tracing provides the missing information that centralized logging systems and metrics fail to provide. [18]

Like it was mentioned earlier, distributed tracing is useful when debugging systems with distributed architecture. However, that is not the whole truth, distributed tracing is handy also in systems that use monolithic architecture. In monolith systems the calls inside the system can be traced in a same way than in distributed systems. Tracing system will present all the calls and call returning times in user interface. [19]

2.3 Storing of Monitoring Data

In this section, the efficient storing methods for monitoring data are introduced. Why and how the storing should be done and what can be done with the old monitoring data to squeeze it to take less space.

In the first Subsection 2.3.1, charasteristics of TSDB (Time Series DataBase) will be introduced and different type of TSDBs will be compared. Second Subsection 2.3.2, describes the search engines, what are they for and how they are used. The last Subsection 2.3.3, introduces the compression methods for different kind of monitoring data and how the compression will change the data.

2.3.1 Time Series Database

Most of the monitoring data is time series data which means it has two factors, the moment of time when the data was gathered and the information itself, that can be in a text format (log data) or in a number format (metrics) [20].

To take full advantage of the monitoring data it needs to be stored in a database that can handle time series data efficiently. The database model made solely for this purpose is called a time series database (TSDB). TSDBs overall have gathered a lot of attention in the area of databases recently and the TSDB has been the fastest growing database model in popularity for the last two years [21]. Reasons for the popularity are monitoring of large scale systems that produce a lot of time series data and IOT devices that also create time series data [22].

The significant difference between TSDB and normal SQL (Structured Query Language) or no-SQL database is that TSDB will save the new information always as INSERTs, not UPDATEs. This fundamental difference allows TSDB to track the history of saved item [23].

In the field of TSDBs there are a lot of different options to choose for. One thing in common between them is that they are efficient in storing and restoring time series data. Time series data can be stored in a traditional database also, it may even work efficiently in a small scale if the indexing is correctly implemented in the database. However, when the amount of services and the complexity of database queries increases, like in monitoring, traditional databases will face performance issues like mentioned in blog posts [24], [25] and [26].

Most of the TSDBs store the data to the hard drive. However, in the article "Gorilla: a fast, scalable, in-memory time series database" there is described a TSDB, named Gorilla, that uses memory to temporarily store the time series data, which allows 73x faster query times. Temporary memory storage for time series data was made possible with XOR (Exclusive Or) compression algorithm, that allowed to reduce the size of a data point from 16 bytes to 1.37 bytes. At the beginning, when they launched Gorilla, they were able to fit last 26 hours of time series data into 1.3TB of RAM (Random Access Memory) space. [27]

2.3.2 Search Engines

Search engine in general is a system that retrieves information from a larger set of information. The most popular search engines are web search engines, like Google search. In this subsection, however, the interest is focused on the log search engines. One well known search engine and a solution for log management is the ELK stack. It is mentioned in both of the monitoring books that I read [1] and [14]. The ELK stack contains the Elasticsearch search engine, the Logstash log aggregation engine and Kibana visualization engine [28].

The reason why the ELK stack is introduced in this Subsection is its unique way to store and retrieve the log data. To simplify the functionality of the ELK stack, it makes an index of incoming text, for example log event, and when that log event is later searched, the ELK will run the search on indexes rather than the data, which makes the retrieving of log data much more efficient [14]. The data itself is stored as an JSON object in to the storage space. In the ELK stack one index can contain one or multiple JSON documents. [29] The indexes are depending on the use of the stored data, and queries that are frequently ran should be indexed to make the work more efficient.

Another log management tool is Grafana Loki. The solution of ELK stack compared to the solution that Grafana Loki provides for log management, in Loki Logs are stored in plaintext form instead JSON objects, set of label names and values are assigned to the text and the label pairs are indexed. The tradeoff makes operation of Loki more efficient and allows more aggressive logging from applications. The way how Loki presents logs with a set of label pairs is similar to Prometheus's way to present metrics, which makes it easy to context switch between logs and metrics when Prometheus is installed alongside Loki. [30]

2.3.3 Compression Methods of Monitoring Data

> "Data is compressed by reducing its redundancy, but this also makes the data less reliable, more prone to errors... Data compression and data reliability are therefore opposites..." [31]

Considering the monitoring data, there is a lot of redundancy both in metric and log data. Monitoring data has a lot higher value on recent data points than on older ones [27]. Recent data points are used for solving the problem at hand and therefore they need to be reliable. In older data points, however, the redundancy can be decreased because they don't have the same use case than the more recent ones. For that reason, as mentioned in Subsection 2.2.4, it is common that in time series databases the older data points are combined.

The merge of data points is made: for gauge type metric data by calculating the average of two or more data points and merging them as one, for counter type metric data by just simply removing, for example every other data point. There is also other type of monitoring data compression methods for time series data, as was described earlier in the Subsection 2.3.1, where TSDB named Gorilla used compression algorithm that reduced the size of a datapoint, instead the amount of datapoints [27]. The approach took by Gorilla has the advantage of maintaining the resolution of monitoring data.

How does the compression algorithm of Gorilla work? Each data point is a pair of 64 bit values representing the timestamp and value at given time. Both of these values will be compressed with different algorithm. For timestamps they use delta of deltas compression and for floating point values they use XOR compression.

It was noticed that timestamps in monitoring have usually constant, or almost constant, interval. It is a consequence of the constant scrape interval. If the monitoring data is scraped from a service, for example, with 60 second interval, usually the change (or delta in Figure 2.1) of a saved timestamp is something between [59,61] seconds. This knowledge was then used to save timestamps in less space with combining them to the previous timestamps and saving only the change of change (or delta of deltas in Figure 2.1). The required space for saving the timestamp depends on the value of delta of deltas. If the change between data points is constant, the saving requires only one bit (in Figure 2.1 compressed value "'0'") instead the original 64 bits. If the change is between [-63, 64] seconds, the saving requires 9 bits (in Figure 2.1 compressed value "'10':-2"). In to the tests made on Gorilla, it was discovered that 96% of timestamps can be compressed to one bit. [27]

With floating point values XOR compression algorithm was used to reduce the size of saved data point. The key finding in creating the XOR compression algorithm was that most timeseries data points don't change significantly compared to their neighbouring data points. If two values are close together, their sign, exponent and first bits of mantissa are identical. In the algorithm, two data points are XOR'd and encoded with following scheme:

1. If XOR with the previous is zero, store bit ’0’.
2. If XOR is non-zero, calculate leading and trailing zeros, store bit ’1’
   (a) If the block of meaningful bits falls within the block of previous meaningful bits, use that information for the block position and store the meaningful XOR’d value and control bit ’0’.
   (b) Else store the length of the number of leading zeros in the next 5 bits, then store the length of meaningful XOR’d value in the next 6 bits. Finally store the meaningful bits of the XORed value. [27]

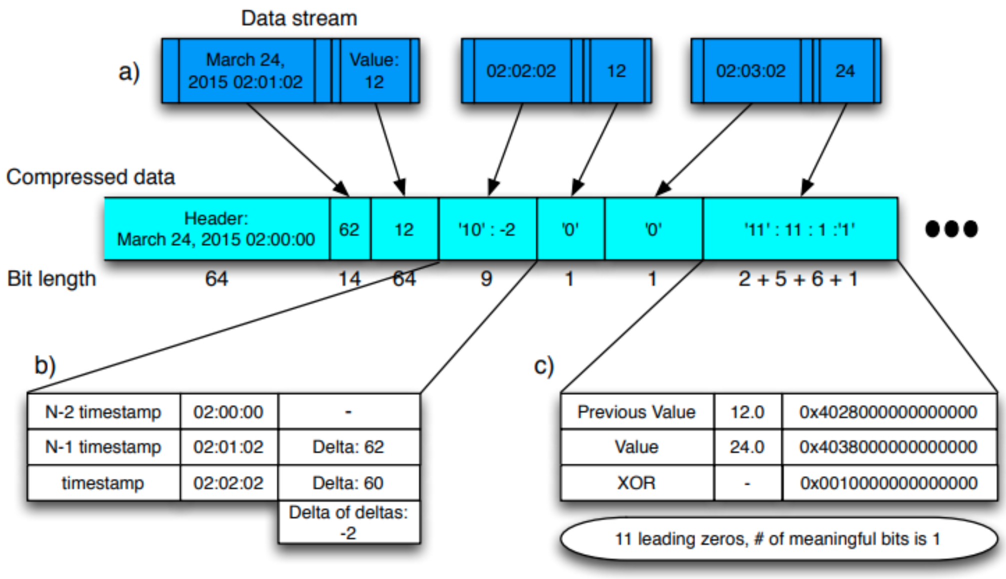


Figure 2.1: Visualisation of compression algorithm of Gorilla that uses delta of deltas compression and XOR compression (from article "Gorilla: A Fast, Scalable, in-Memory Time Series Database" [27]).

Log data can also be compressed. Like it was mentioned earlier, the recent data has greater value than the old data, so for logs it can be for example decided that log data older than a week will be compressed. As a compression method simple solution, such as bzip2 and gzip are common. In the article "Improving Compression of Massive Log Data" by Robert Christensen [32], a novel method for log compression is introduced. The method divides log data into homogeneous buckets which makes the traditional compression methods more effective.

2.4 Monitoring Tools

In this chapter, remarkable monitoring tools (for this thesis) are introduced. The focus of the introduced tools are on the data collection and storage, and the tools are Prometheus, Riemann and Nagios. These tools were chosen to be introduced in this chapter because: Prometheus is going to be used to take care of monitoring in the company that this thesis is made for, Riemann is alternative solution for Prometheus using push based monitoring and Nagios was mentioned by multiple monitoring books as a bad example of pull based monitoring.

2.4.1 Prometheus

Prometheus was built at SoundCloud in 2012 to replace their existing monitoring tools, Graphite and StatsD, which didn't serve their needs any more. At the beginning, there were four requirements that they wanted Prometheus to fulfil: A multi-dimensional data model, operational simplicity, scalable data collection and a powerful query language. [33]

Prometheus takes care of data collection, (temporary) data storage and alerting phases in BCB Medical. As it was told in the Subsection 2.2.2, Prometheus is based on the pull based method of collecting monitoring data. In order to pull metrics from a service, the Prometheus exporter (down left on Figure 2.2) needs to be installed on a server. Prometheus exporter is basically a service that provides interface for Prometheus server to fetch all available metrics of a service from a single destination. There are many ready-made exporters available, for example, for Linux server, MySQL database, Mongo database and JVM (Java Virtual Machine). Exporters can also be made for custom need if there is no official or a community made exporter available. The Prometheus website provides instructions and guidelines for exporter creation [34].

Despite the fact that Prometheus is a pull based monitoring system, the metrics can also be pushed to the Prometheus server, but it needs to be done through special part called push gateway (middle left in Figure 2.2). Push based method in Prometheus can be used in jobs that live only shortly and configuring them to Prometheus server would cause too much trouble. [35]

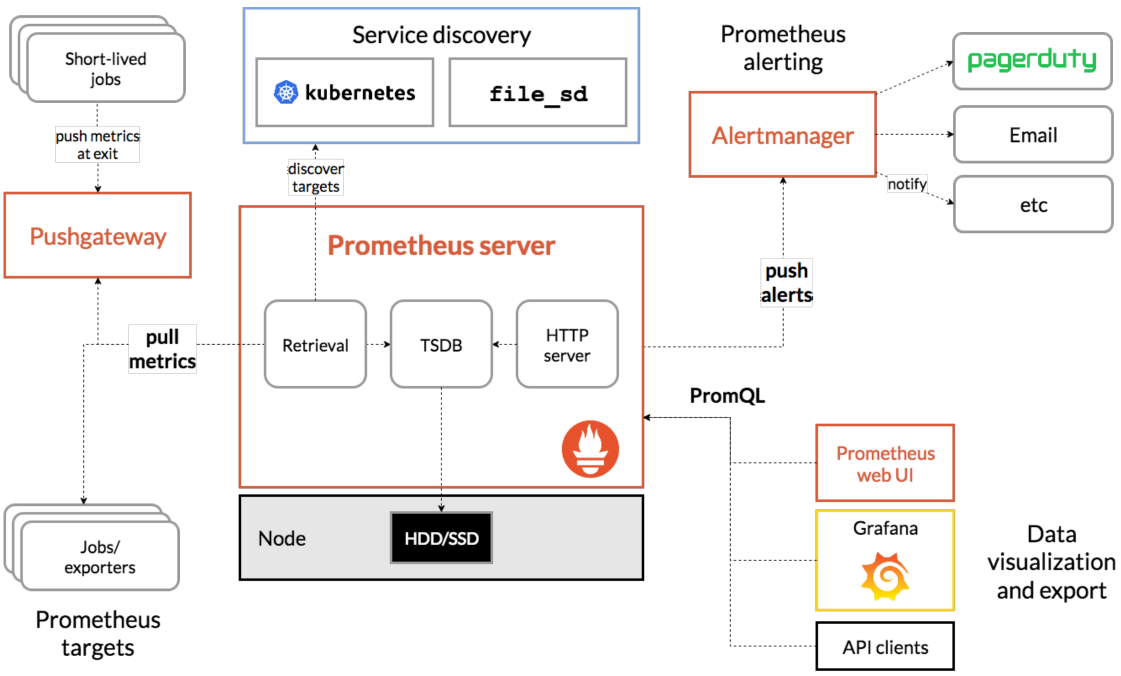


Figure 2.2: Architecture diagram of Prometheus (from Prometheus documentation [35]).

2.4.2 Riemann

Riemann is an open source monitoring system authored by Kyle Kingsbury. Riemann is written in Clojure, which is dialect of Lisp, and it runs on top of JVM. The idea of Riemann is to make monitoring easy by default. Riemann is called as an event processing engine because it aggregates events from applications and feeds them into a stream processing language. There are three main concepts in Riemann: events, streams and index. Events contain several fields of information like: host, service, state, time, description, tags, metric and ttl (time in seconds how long the event is valid). Each event is added to one or more streams. Streams are transporting events from one point to another. The index contains all information about all of the services that Riemann is tracking. [14]

The Figure 2.3 describes one possible monitoring stack that uses Riemann as monitoring system, InfluxDB as data storage and Grafana as visualisation tool.

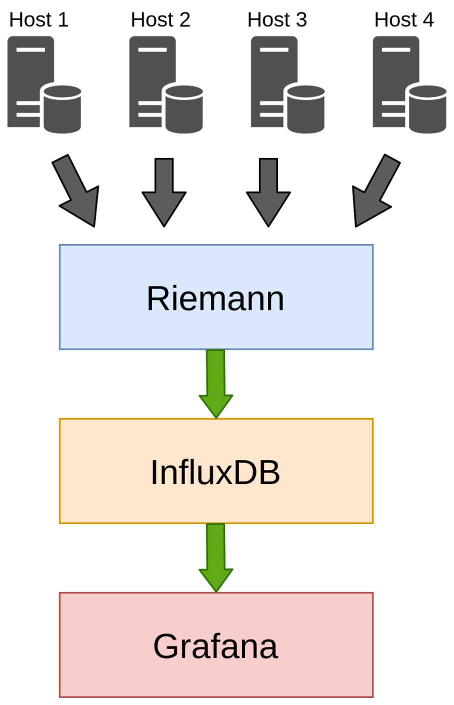


Figure 2.3: Monitoring stack with Riemann [36].

2.4.3 Nagios

Nagios is open source computer software that monitors systems, networks and infrastructure. However, the open source community of Nagios is not as active as in Prometheus for example, and it is mainly maintained by Nagios team. Nagios was originally designed by Ethan Galdstad in 1996. [37]

Nagios offers different products for different purposes. Nagios core is the monitoring backend. Nagios XI is offering enterprise version of Nagios with GUI, that uses Nagios core as backend. Nagios Log Server for logs. Nagios Network Analyzer for networks. [38] From technical point of view, Nagios core is written in C language and Nagios XI is written mostly with PHP. [37]

Nagios uses pull based method in its monitoring architecture as described in Figure 2.4. Pulling of the monitoring data is handled by the Scheduler component in Nagios server. Scheduler sends message to plugins, that are installed in the monitored hosts, and plugins return the monitoring data from host to the monitoring server. Nagios was accused on the monitoring books, that it is not a scalable pull based solution. One of the reasons why Nagios is introduced in this monitoring tools section was that it would be investigated, if Nagios still suffers from scalability issues or are they put in the past. [39]

When searched on the web, there is no recent articles or posts issuing that Nagios would still have scalability issues. On the other hand, the community of Nagios is not as large as for example, in Prometheus, so it can be that these issues haven't been investigated lately.

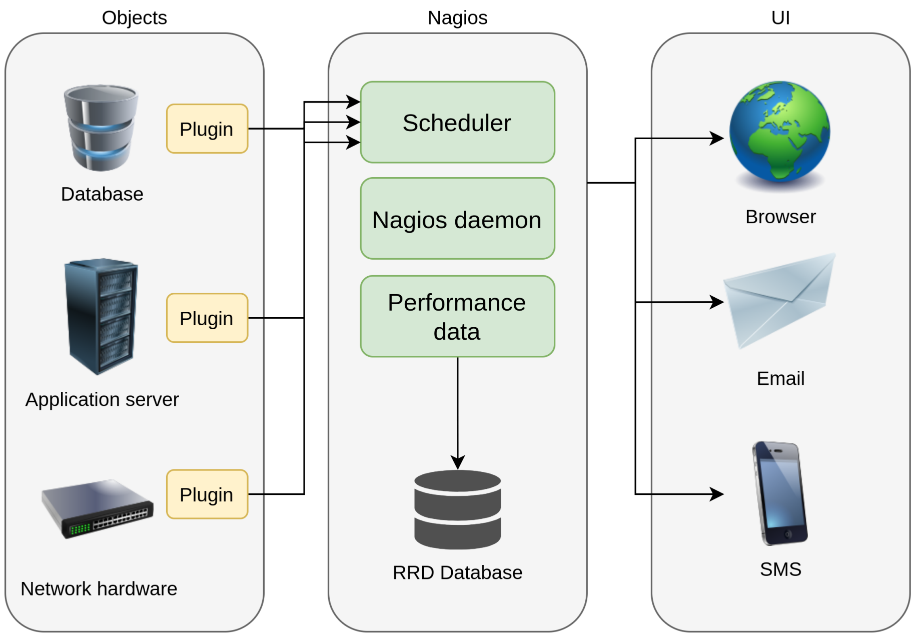


Figure 2.4: Architecture diagram of Nagios [40].

3 Visualization and Analysis

This chapter covers the visualization and analysis phases from the monitoring process mentioned in Chapter 1. In Section 3.1, the basics of visualization are introduced in order to build better dashboards for monitoring. In Section 3.2, different software anomaly detection methods are compared. In the last Section 3.3, literature proof for predictive monitoring is searched for.

3.1 Visualization

This section will clarify the problem of data visualization. Why data should be visualized; what elements good dashboard design contains; and what tools there are available on monitoring data visualization. The book "Information Dashboard Design: The Effective Visual Communication of Data" [41] is used as a main source of information in this section. The book is rather old, published in 2006, so the discoveries from the book will be complemented with more recent articles on the topics.

3.1.1 History of Data Visualization and Dashboards

It is commonly thought that graphics and data visualization are rather new invention. However, that is not the case, the earliest proofs of data visualization come from 200 B.C., when ancient Egyptian surveyors made laying-outs of the positions of towns and used an idea similar to the modern coordinate system. [42]

The beginning of modern graphics was in the first half of the 19th century. In that time, all the modern forms of data display were invented: bar charts, pie charts, histograms, time-series plots, line graphs, scatter plots and so on. [42] Nowadays, many of those data display techniques are used in modern dashboards.

EIS (Executive Information System) is a computer system introduced in the 1980s that was made to provide information from the company to the senior managers in an easily understandable format [43]. EISs took advantage of the data display techniques like line graphs and pie charts. It is said that EISs were the early stage of dashboards. However, at the time, before data collection and data analytics were popular, the quality of the data was so lousy that there were really no use for the dashboards. They were ahead of their time. [41]

3.1.2 Why to Visualize?

In the study made in 2003 in the University of Berkeley the researchers estimated that the amount of information in the world increases two exa bytes every year [44]. The excessive amount of information thrives to find out efficient ways to present the information. Visualization is needed to deliver messages quicker and in a format that is easier to understand.

3.1.3 Dashboard Design Principles

In monitoring main visualization media is a dashboard. The goal of the dashboard is to show for a certain user group the needed information about the system with one glance. In the book Information Dashboard Design [41], common properties for good dashboards are introduced. Those properties are gathered as dashboard design principles to this Subsection and introduced below.

A dashboard should have clear objective. Meaning that the design process of a dashboard should start from a problem that can be solved with a dashboard. Many times dashboards are built from the wrong end to present all the available data with fancy graphics and after that it is thought where the data could be used.

A dashboard should fit on a single computer screen. The efficiency of dashboards is based on the idea that a single glance at the dashboard would give the information that the user is looking for. If using the dashboard requires scrolling or clicking, the dashboard is already working inefficiently and part of the information is always hidden on the dashboard.

A dashboard doesn't need to look good. Many dashboards designs use effects that are there just for the sake of appearance. For example, some data can be presented in less suitable format, e.g. in a pie chart, because there were already three line graphs. Graphical representation format should always be chosen based on the objective, not on the appearance.

3.1.4 Visualisation Tools

In this Subsection a few popular OSS tools for monitoring data visualization are proposed. The basic functionalities and operating methods of the tools are introduced and some comparison between tools will be conducted.

Graphite

Graphite is introduced in their website to do two things: store numeric time series data and render graphs of the data. Graphite was designed and written by Chris Davis at the company named Orbitz in 2006. In 2008 Graphite was released under the open source Apache 2.0 license. [45]

Architecture of Graphite consists three components: Carbon, which is in charge of listening the stream of incoming time series data, Whisper that will store the time series data and Graphite-Web that handles user interface and API for data visualization as described in Figure 3.1. Graphite is mainly developed with JavaScript and Python languages. [45]

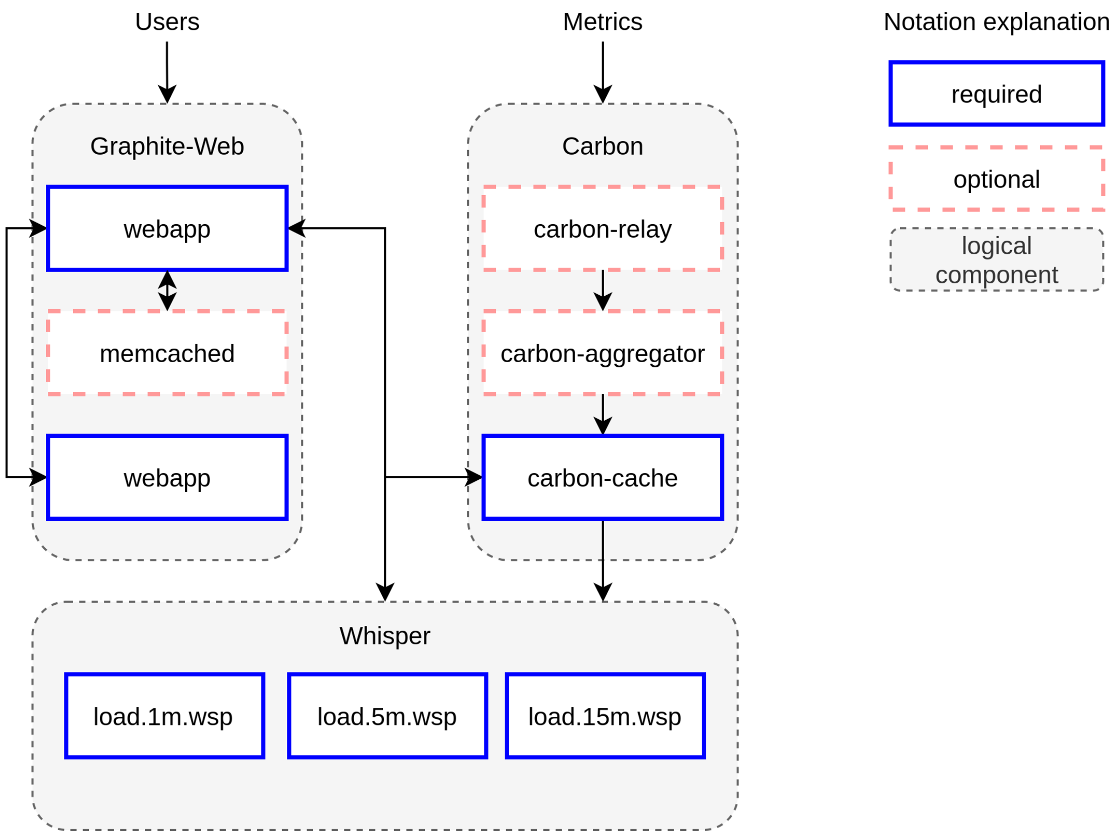


Figure 3.1: Architecture diagram of Graphite [45].

Grafana

The Grafana project was started in 2014. The initiator was Torkel Ödegaard, who developed on his spare time a custom dashboard and frontend for Graphite monitoring tool and named it as Grafana. Ödegaard was working for Ebay Sweden as a developer at the time [46].

On Grafana website it is told that nowadays Grafana ables to query, visualize and alert on metrics and logs. Grafana is usually used as a visualization tool for Prometheus, because Prometheus visualization methods are too primitive for today's demands. [47]

Grafana has an active community that developes grafana and its components. The dashboard creation in Grafana is made really simple. The user can download ready-made dashboards for Prometheus exporters from Grafana site. For example, for the metrics that Node exporter provides the dashboard presented in Figure 3.2 can be downloaded. These ready-made dashboards can then be customized to serve user's needs.

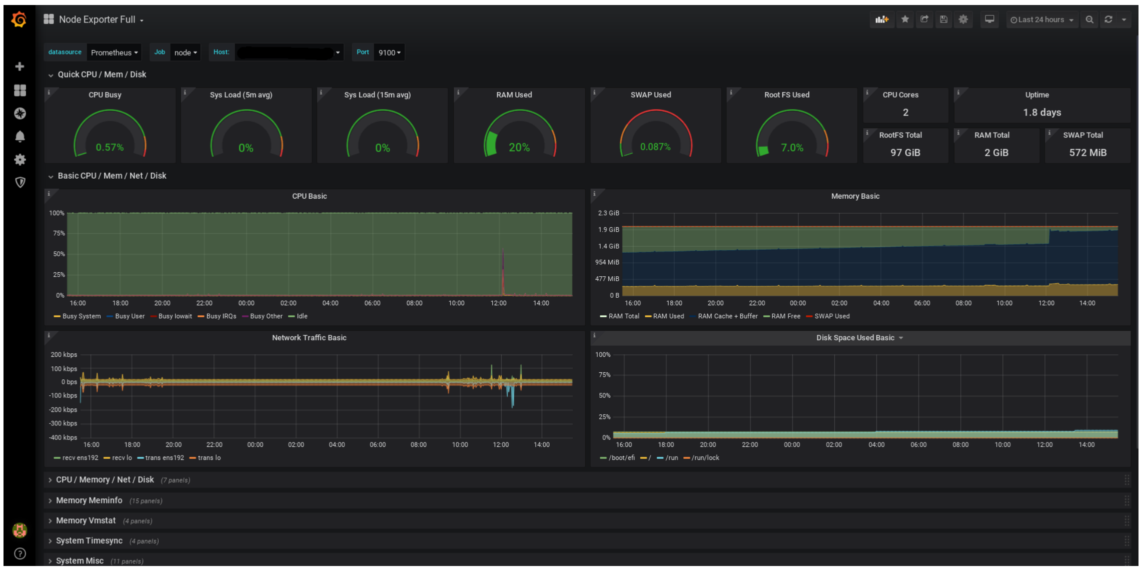


Figure 3.2: Grafana dashboard for Prometheus Node exporter [48].

3.2 Detection of Software Anomalies

Detecting a software anomaly from a visualized data described in the previous section may be effortless for a human that has prior knowledge about the behaviour of a monitored system. However, when the system is scaled up and it contains hundreds of subsystems, no human can monitor them, another way to discover anomalies needs to be invented. In this section, the focus is on the automated detection of software anomalies. Methods to detect anomalies without prior knowledge about the system or the type of anomaly, are introduced in this Section. In addition, the detection system works automatically without human interaction to make it easier to scale.

One approach to the problem at hand has been given in the article "Black Box Anomaly Detection: Is it Utopian?" published in 2006 by S. Venkatamaran, J. Caballero, D. Song, A. Blum, and J. Yates [49]. In the article, the anomaly detection tool uses certain threshold values to filter monitoring data into two groups, one that doesn't have anomalies and one that might have anomalies. Then the machine learning model is trained and tested with the data that doesn't have anomalies. At the end the created data profiles are used in the anomaly detector to detect anomalies from the real monitoring data. Whole process is described in Figure 3.3.

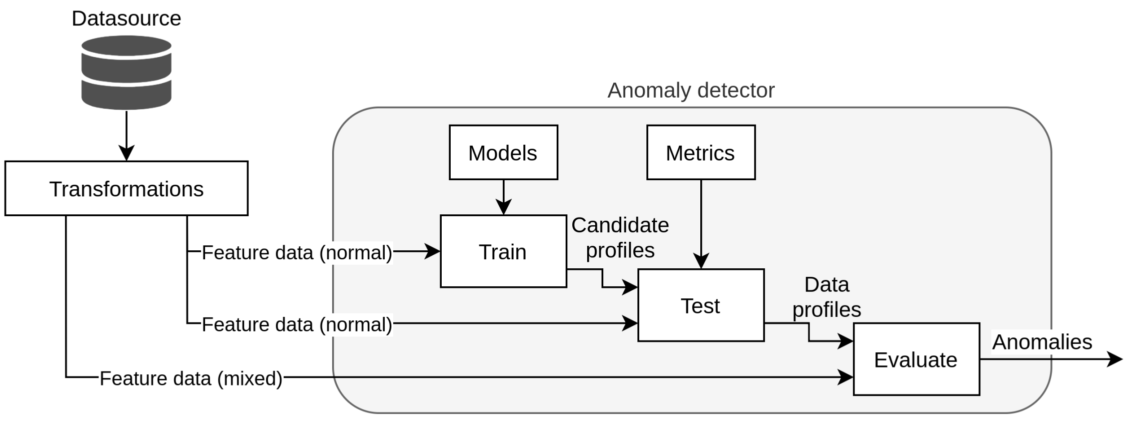


Figure 3.3: Operation principle of anomaly detection tool from article "Black Box Anomaly Detection: Is it Utopian?" [49].

The anomaly detection framework described in the article "Black Box Anomaly Detection: Is it Utopian?" fulfils the goal of a monitoring system not needing to know about the type of anomaly beforehand and it is also generic [50]. However, some level of domain knowledge of the monitored system is needed when setting threshold values at the beginning for the algorithm that divides the data into a group that doesn't contain anomalies and a group that might contain anomalies.

There is also human interaction needed in setting threshold values and evaluating machine learning model results, which limits the scalability of the presented solution.

Another approach to the matter is described in the article "Toward Automated Anomaly Identification in Large-Scale Systems" by Z. Lan, Z. Zheng, and Y. Li [51] published in 2010. The article is written for a large scale computer system but the used anomaly detection methods can be generalized to the anomaly detection of any system. In the article, automated anomaly identification system is described to use PCA (Principal Component Analysis) and ICA (Independent Component Analysis) as feature selection methods in order to select what monitoring data to use, as described in Figure 3.4. These two methods also work as a dimensionality reduction tool when multi-dimensional monitoring data is turned into three-dimensional data. After dimensionality reduction, the outliers of the data are interpreted as anomalies. The data that they are using in their research is production monitoring data where they have injected five different type of anomalies. It is proved in the article that ICA outperforms PCA as a feature selection method when multiple anomalies exist in the system. However, the result of PCA was better than in a control group where no dimensionality reduction was done. The reason for ICA performing better than PCA, is that ICA is naturally better at finding independent groups of data, which is useful in anomaly detection.

The article "Toward Automated Anomaly Identification in Large-Scale Systems" fulfils the goals of anomaly detection without prior knowledge of the monitored system and the anomalies. Especially good in the approach is that it can discover totally new kind of anomalies, because there is no ready trained machine learning model that is trying to find certain patterns in the data (supervised learning), instead it uses unsupervised learning and tries to find similarities and outliers from data. However, the approach is not perfect fit because anomaly identification system still needs manual validation of those outliers so it doesn't work without human

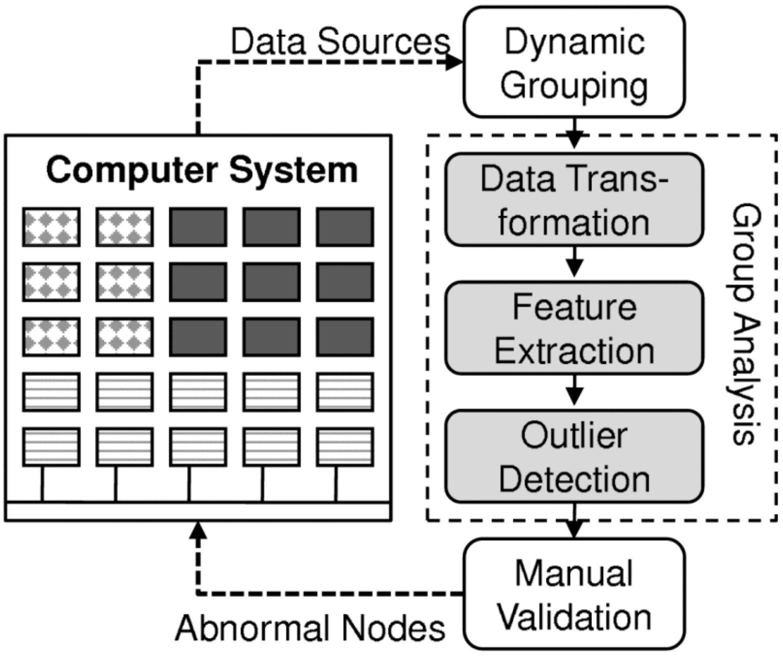


Figure 3.4: Operation principle of automated anomaly identification tool (from article "Toward Automated Anomaly Identification in Large-Scale Systems" [51]).

interaction.

Laptev, Amizadeh and Flint suggest in their article "Generic and scalable framework for automated time-series anomaly detection" [50] that machine learning can be used to solve the problem of setting threshold value. This research is the most recent one of the three researches introduced in this Section and it is published in 2015. Laptev, Amizadeh and Flint are building the automated anomaly detection system for Yahoo that is described in Figure 3.5. Yahoo has working monitoring in their systems and they get constantly time-series data from different parts of the systems. Setting reliable threshold value for each of these systems by hand would be impossible, and that's the reason why they are using machine learning for the task. The essential element of automated anomaly detection system is that the system is trying to predict the future value of a given monitoring data point (in prediction there can be used different algorithms, depending on the object of monitoring). If the predicted value of monitoring data is far enough from the real value of monitoring data, it is construed as an anomaly in a system.

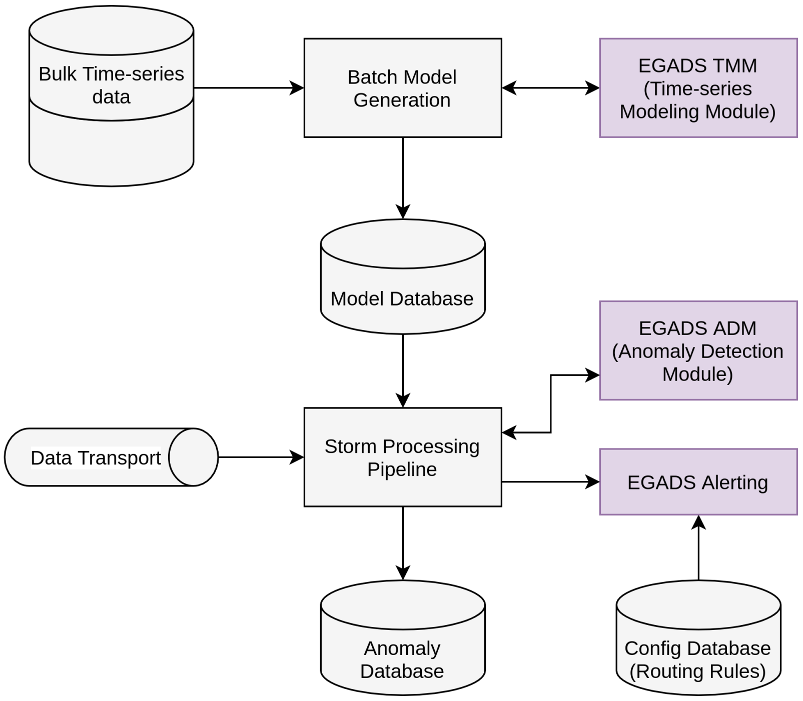

Bulk Time-series data
Batch Model Generation
EGADS TMM (Time-series Modeling Module)
Model Database
EGADS ADM (Anomaly Detection Module)
Data Transport
Storm Processing Pipeline
EGADS Alerting
Anomaly Database
Config Database (Routing Rules)

Figure 3.5: Anomaly detection tool that Yahoo monitoring services use, described in article "Generic and scalable framework for automated time-series anomaly detection" [50].

Yahoo monitoring service described in the article "Generic and scalable framework for automated time-series anomaly detection" fulfils the goals of anomaly detection. It works automatically so no human interaction is needed, it can handle any type of anomaly and it doesn't need domain knowledge about the monitored system. In addition, the solution is scalable and it even provides functionality to point out the interesting anomalies from the set of all anomalies, so the user gets notified only from the anomalies that he is interested in.

3.3 Anticipation of Software Anomalies

Anticipation of software anomalies rests on the idea that it is possible to foretell from the trend of past monitoring data when a given system is facing an error. There is no publications on the predictive monitoring solely, but in the articles that have studied automated anomaly detection [51], [50] it is said en passant that these methods used in anomaly detection could also be used in anomaly prediction. However, in the Conclusion of this document (Section 7), the anomaly detection is being taken one step further and a theory for implementation of predictive monitoring will be introduced.

4 Alerts and Recovery

This chapter is handling the last two parts of monitoring process: alerting and recovering. In this chapter the main source of information are the books Effective Monitoring and Alerting [52], and Prometheus Up and Running [53]. The reason for the use of book sources is that there is not many, if any, published articles on alerting.

4.1 Alert on Anomalies

Alerts exist because in some cases the system may face a problem that it is not able to recover from without human interaction. In these cases the monitoring system sends an alert to the person who is being on-call. To be on-call means that you are responsible to keep the software running in case of a failure. [1] On the other hand, alerting releases operation engineers from staring at the metrics and graphs that monitoring software produces. Especially, when the amount of monitored services increases the problem detection from monitoring data by hand becomes impossible. [52]

Good rule of thumb in alerting is to alert on symptoms rather than metrics. Symptoms are something that user of the monitored application can see or feel. For example, slowness or sluggishness is a symptom that user can feel when using the application. With this rule, the false alarms and unnecessary monitoring is avoided. When the symptom is discovered, the root cause of a problem can be searched for with other metrics that are fetched from the application. If connection between a metric and the symptom is found, the symptoms could be prevented in the future with correct threshold setting on the metric. [53]

Automate recovery of anomalies that don't require human intelligence. If anomalies or symptoms don't really require human investigation on the problem they should be recovered automatically. This way the amount of alerts can be minimized. The Prometheus Up and Running book suggests that one or two alerts should fire at maximum in a day [53]. When the amount of alerts increases from that, they just get ignored.

Playbooks (or runbooks) should be used in alerting. The use of playbooks is suggested in several monitoring books [1], [52] and [53]. Playbook is a guide that is delivered with the alert, and it tells general information about the system that the alert was triggered on, why the alert was triggered and how to start recovering from the problematic state that the alert is alerting. An example playbook is described in Appendix B.

Sometimes alerting system first alerts from a problem and after certain period of time the alert magically disappears. These kind of issues are usually the cause of too sensitive alerting system. The issue is easily fixable by tuning the alerting system and the threshold values. [53]

4.2 Alert Types

In the book "Effective Monitoring and Alerting" [52] anomalies are classified by the severity of two features: recoverability and impact. Recoverability is a measure that describes the likelihood of a system to recover from anomaly without operation engineer intervention. Impact is a measure that tells how severe are the effects of an anomaly to the system. Both of the features are divided into three levels of severity which outputs a total of 9 anomaly types as described in Figure 4.1.

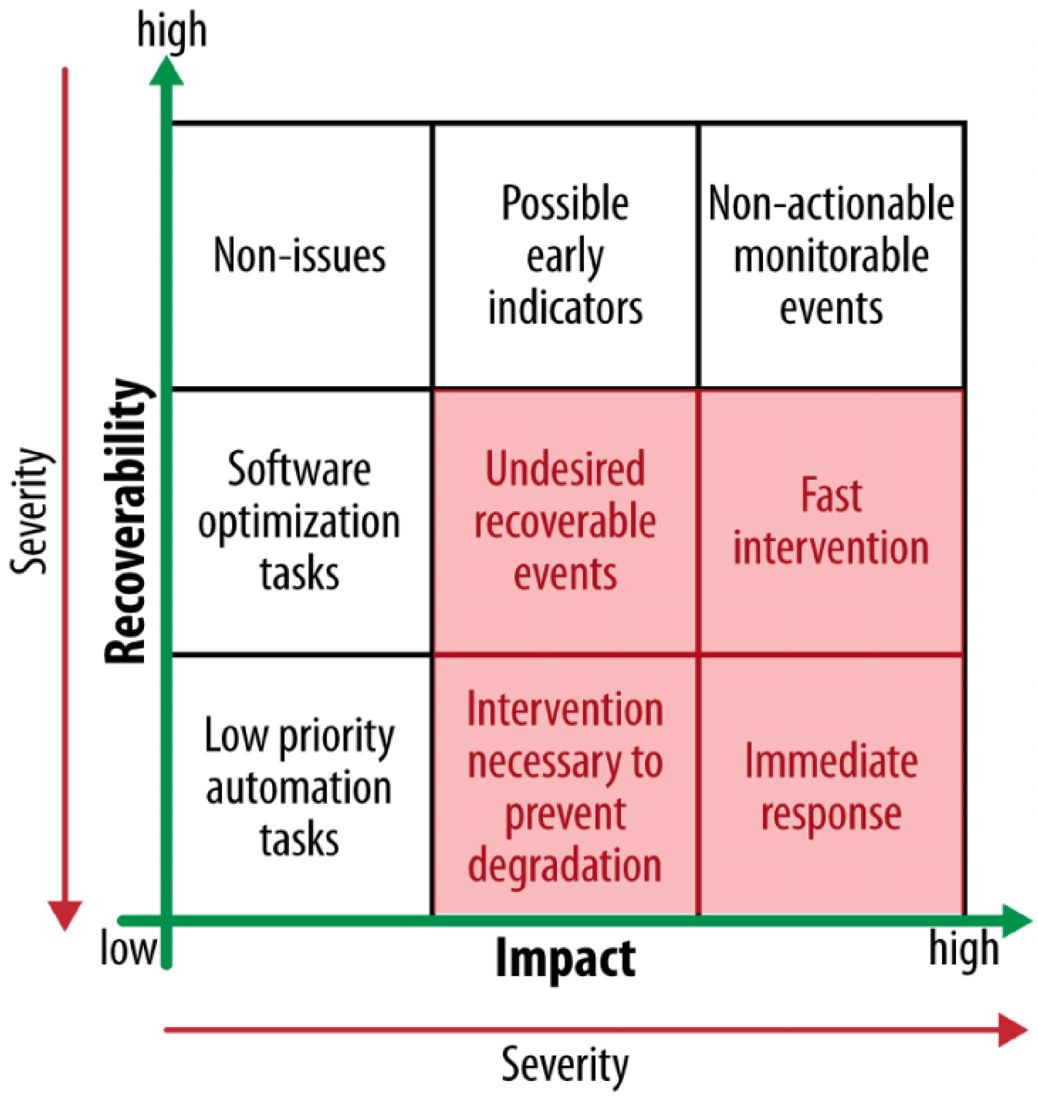

high
Non-issues
Possible early indicators
Non-actionable monitorable events
Severity
Recoverability
Software optimization tasks
Undesired recoverable events
Fast intervention
Low priority automation tasks
Intervention necessary to prevent degradation
Immediate response
low
Impact
high
Severity

Figure 4.1: Anomalies classified by the severity of recoverability and impact. Anomalies on the red background demand immediate action. [52]

On the bottom right corner in the Figure 4.1 there is the most severe anomaly type, both in impact and in recoverability. These kind of anomalies are classified as critical and they require immediate action. On the opposite corner, in top left, there are anomalies that doesn't have impact on the system and are quickly recovered. Thorough explanation of the different anomaly types in the Figure can be found on the Table 4.1.

Anomaly type	**Explanation**
Non-issues	Anomalies with no impact on the system.
Software optimization tasks	Events that cause inefficient resource utilization, could be eliminated with architectural restructuring.
Possible early indicators	Small bugs that affect a tiny group of users. Can be result of temporary resource saturation and indicates the inability of a system to handle stress.
Low priority automation tasks	Faults that doesn't have immediate impact on the system, but can cause other failures in future.
Non-actionable monitorable events	Events that cause a small fraction of users to face serious failures, which immediately disappear by themselves. Quick disappearance of the failures, makes them pointless to alert on, but the frequency of these failures should be regularly verified to ensure that they remain isolated.
Undesired recoverable events	Events that have noncritical effect on the system in present time, but can potentially develop into bigger issues in the future, if not recovered. No alarm is necessary if subsequent jobs succeed.
Intervention necessary to prevent degradation	Events that are not immediately catastrophic, but the system is at a high risk collapsing without human intervention. For example, the system may build a backlog that increases the system delay.

continues in the next page

Table 4.1 – continues from the previous page

Anomaly type	**Explanation**
Fast intervention	Events that cause loss of availability for a fraction of users or some portion of the system becomes unresponsive. These events require fast intervention to minimize impact and prevent escalation.
Immediate response	Events that block user access or impair the system operation. Critical events increase system downtime and cause losses in productivity and revenue.

Table 4.1: Explanations for different anomaly types that were introduced in Figure 4.1.

According to the Effective monitoring and alerting book [52] the last four anomaly types are the ones that should be alerted (marked with red background in Figure 4.1). The remaining six anomaly types have either low impact on the system or quick automatic recovery, so it is reasonable to not alert on those cases.

For the alerted anomaly types the trade off is made between speed and accuracy of detection. The anomalies that have high impact on the system and low recoverability value, need to be alerted as soon as possible, when the first signs of the anomaly appears. The before made decision will decrease the anomaly detection accuracy, which will possibly lead to some false alarms, but it is the price that has to be paid in order to avoid or minimize the high impact on the system that the anomalies in this group have.

On the other hand, when the impact of the anomaly type decreases, early detection doesn't play as huge role and the trade off between speed and accuracy can be moved towards the accuracy, to avoid jamming up the operation engineers that resolve the alerts. [52]

In this Section, any predefined alert groups was not established, instead all of the alerts have same priority and they should be reacted as soon as possible when an alert is notified. The severity of the anomaly is defined in to the monitoring system in a continuous way with the decision, whether to emphasize on speed or accuracy in detection.

4.3 Default Alerts

Effective monitoring book [52] suggests some default alerts that can be made on any system. Those alerts will be introduced and discussed in this section. However, it is also mentioned in the book that the domain of a system that is being monitored is always an important aspect to consider, since different software systems demand different features from monitoring, so the default alerts introduced below are not set in stone.

From resource level it is said that common metrics that are alerted are network latency, packet loss, CPU utilization, available disk space and available memory. Network latency and packet loss can easily be measured by pinging an internal and external location and saving the response time and counting the times when packet loss occurred. [52] It is also reasonable to alert on network problems, since they usually appear for the user as a sluggishness of the system or a system that doesn't deliver its purpose. CPU utilization is controversial thing to alert, as was described in the Section 2, high CPU utilization rate doesn't always show any signs for the end user and often it is not the root cause for problems. Available disk space and memory are reasonable things to alert, since when they reach certain limit it is certain that it will affect on the system performance in various ways.

From platform level the book says that alerts can be fired based on the turnaround times and HTTP (HyperText Transfer Protocol) response codes [52]. Turnaround time describes the time between submission of the process and completion of the process [54]. Turnaround time is one of the factors that has a great effect on customer satisfaction. If service is working slowly, customers won't waste their time on using it and they will look for alternative options that work better. So turnaround time is a good item to add on the list of alerted items. HTTP response codes tell if the request made by user has returned an error or the wanted result. The rise of HTTP response codes tells immediately if some part of the site is unreachable, or if some link on the site is routing to an item that doesn't exist. Again, this kind of alert is good because it tells about the symptoms that user has faced.

In application level the alerts are suggested to be made on availability, error rate and content freshness. The availability should be checked from multiple external location to get reliable results. The threshold values should correspond to the SLAs. Error rate is monitored through the parameters that constitute the user errors, for example the word error could be searched from the logs and made as a metric. If application contains evolving content, the freshness of that content can be evaluated with age metric. Age metric tells the difference in seconds between current time and the time of last content update. [52] Availability alert is one of the first alerts that is usually implemented when monitoring is being built. Availability is a good metric to alert and monitor because it helps first to minimize the downtime of an application when problems are noticed quicker and secondly it also gives concrete numbers about the downtime, which can then be compared to SLAs. When alerting on error rate, it would be good to line out the minor errors that don't cause any problem for the end user to avoid jamming up the alerting system and operation engineers behind the alert system that take care of those problems. Content freshness is an alert which depends a lot on the domain that the application is on. If the freshness of a content is important for the application it should be alerted but there are also many cases where the content freshness doesn't have an effect on the user experience.

4.4 Alerting Tool: Alertmanager

Alertmanager is a tool that handles the alerts sent from monitoring system. It deduplicates, groups, silences and finally routes the alerts to the correct receiver integration like e-mail [55]. Alertmanager is part of the Prometheus project and is supposed to be used with Prometheus, however it probably works with other monitoring tools also. The main components of Alertmanager are described in the architecture diagram in Figure 4.2. Alertmanager contains API, where monitoring systems, or alert generators as described in the diagram, can send the alerts. The same API is used also to send silences in the Alertmanager. You might want to silence your alerting system for example, for the maintenance or for a problem that is already noticed. [53]

Alertmanager provides also other functionalities besides notifications and silencing. Alert inhibition is a functionality that prevents notifications for minor alerts, if more severe alert in the same group is fired. For example, alert won't be triggered from a single service being down, if the whole datacenter has gone down. Routing is a functionality that allows the notifications to be sent to different places for different problems. For example, alerts from test environment can be routed to different location than the product environment alerts. Also if some team owns certain product, the alerts from that specific product can be routed directly to the team. The idea of routing is that organisation would only need one Alertmanager that routes the notifications from alerts to different locations. Grouping is a feature that allows to group alerts based on the environment that is monitored. Alerts from certain set of products can be grouped as one which decreases the amount of alerts in cases where problem in one product cascades to other products. Throttling and repetition is something that is done for the grouped alerts if another alert fires from same group.

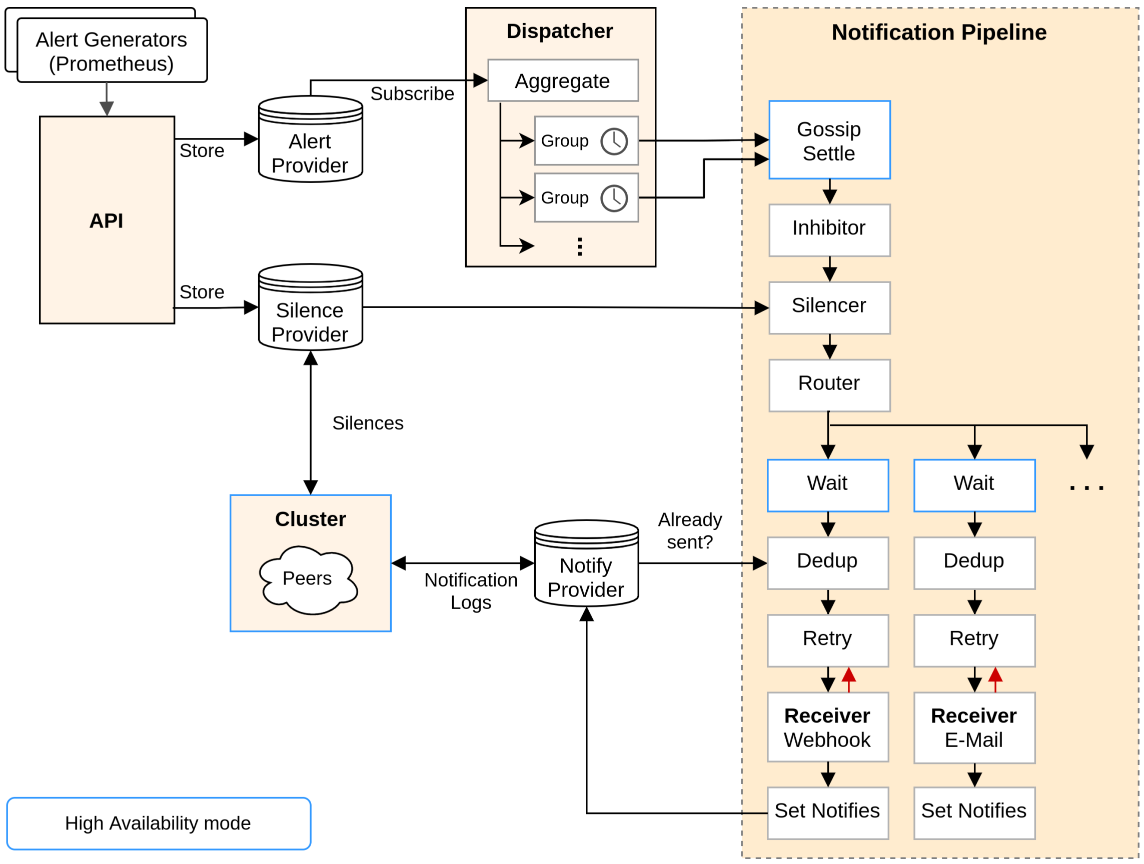


Figure 4.2: Architecture diagram of Alertmanager (from Alertmanager documentation [55]).

Throttling will be made to avoid the alerts getting lost for too long and also to avoid sending multiple alerts from the same issue. [53]

All of the before mentioned Alertmanager features are managed with configuration file in YAML-format. Configuration can be easily updated by editing the YAML-file and sending a HTTP request to reload endpoint. [53]

4.5 Recovery: ITIL's Process for Incident Management

When the alert fires, usually it is a good idea to have a process for recovering. A guideline to the recovering process of an incident is provided by the ITIL (Information technology infrastructure library). ITIL is a library of guidelines and practices about IT service management. ITIL processes are described in Figure 4.3. ITIL's process for incident management is part of the service operation process and it was published in ITIL documentation version 3. The process for incident management contains following steps:

1. Incident identification,
2. incident logging,
3. incident categorization,
4. incident prioritization,
5. initial diagnosis,
6. escalation (if needed),
7. incident resolution.

∗ Communication with the user community throughout the incident.

The first step, incident identification starts when incident is received from monitoring system or from other route, like customer announcement. In this step it is decided, if informed incident is really an incident or a request. Usually incidents coming from monitoring system are real incidents, but those coming from customers may be requests also. After identification, incident is logged in a service desk software. Monitoring system can skip the incident identification part by automatically

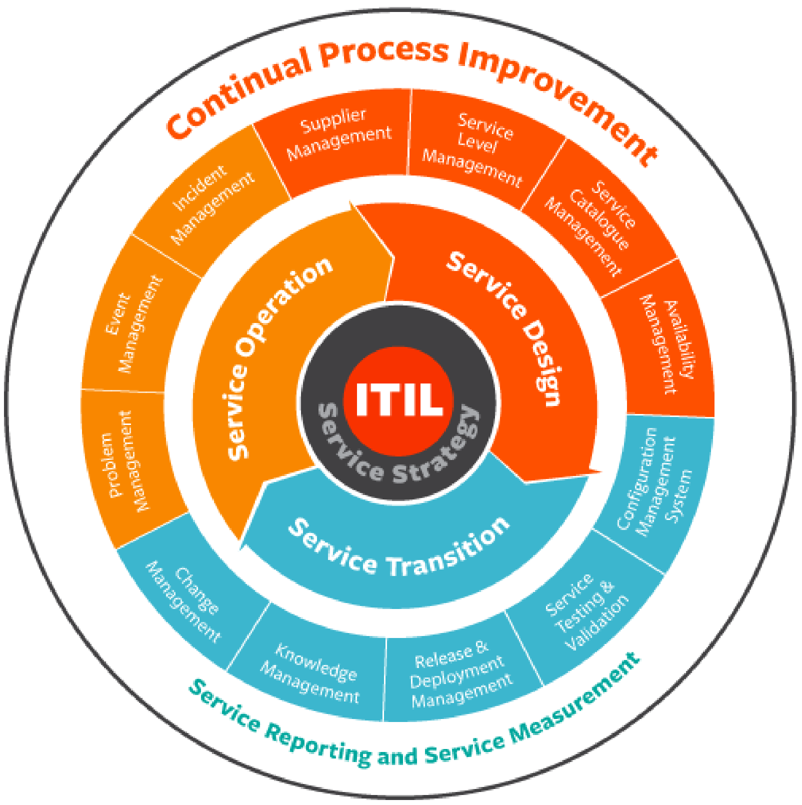


Figure 4.3: ITIL's service strategy and processes for operation, design and transition (from bmc blog [56]).

making a ticket, if all of the incidents coming from monitoring system are presumed to be real incidents. [56]

Usually, parallel with incident logging the incident categorization and incident prioritization are also being done. If announcement comes from a customer, first level support person needs to decide the category and priority of an incident, but if the incident is logged to a ticket system automatically by monitoring system, it is usual that the category and priority is also known beforehand based on the location where alert was triggered. ITIL provides three priority levels for incidents. Level 1 incident is low priority incident that doesn't interrupt the use of a service and incident can be worked around. Level 2 incident is a medium priority incident that has some affect on the use of a service, but the service is still usable. Level

3 incident is a high priority incident that interrupts the use of a service and needs quick reaction. [56]

When the incident has been prioritized the resolution can start by identifying the parts that are affected and the plan for recovery can be done. If the recover plan involves other parties the incident needs to be escalated to those parties in this step. Finally, when the incident is fixed the resolution can be logged to the system. As the last part there is also mentioned communication with the users throughout the incident. This is especially important if the resolution time is longer than expected. If the incident is resolved immediately, it is enough to inform the customer that the incident is now resolved. [56]

5 Research Work

In this chapter, the business motivation of this thesis is going to be introduced in Section 5.1, the plan for interviews and implementation work is described in Section 5.2. The practical work itself will be documented in Chapter 6.

5.1 Definition of the Business Case

BCB Medical is a SaaS company that provides software solutions for hospitals to help healthcare professionals to make better treatment decisions (more thorrow instruction about the software solutions that BCB Medical provides is in the Subsection 6.1.2). Because the software is provided as a service, the company has made SLAs (Service Level Agreement) with customers that require a certain level of availability. With monitoring the availability of service and customer satisfaction can be ensured.

Up to the present, BCB Medical has handled their monitoring with SaaS monitoring tools. Now they are moving their monitoring towards the next level of monitoring, which is taking advantage of OSS monitoring tools. OSS monitoring tools offer more freedom and prospect of modification in what comes to the monitoring. At the heart of the monitoring infrastructure there is going to be Prometheus server which gathers the metrics from different parts of the system and saves the data in the time-series database as described in Subsection 2.4.1.

My position at the firm was in the second level customer service and support, at the time when this thesis was started. My teammates and I were responsible for solving software bugs and deficiencies needing more technical knowledge. This position gave me a great general view of all the products that BCB Medical offers and the bugs that the products may face.

The main task for me to complete with this thesis was to enhance customer satisfaction with the tools of monitoring. Customer satisfaction is enhanced by doing the work of the customer service team more efficient with first defining the problem areas that employ the customer service team the most and then planning and implementing monitoring solutions that help to solve those problems in less time. The goal is that anomalies are going to be found more easily and recovering from the erroneous state is faster.

5.2 Definition of the Research Tasks

In this section, the plan for the practical work of this thesis is introduced. The practical work is divided into two phases: interview and implementation.

5.2.1 Plan for the Interviews

For the research method, a qualitative research method was chosen instead of a quantitative one because the subject group (customer support team) is small so the qualitative method will give more reliable results.

Interviews are conducted separately (each team member will be interviewed one by one) and they follow a semi structured guideline. The semi structured guideline in interview means that the questions are more open, and defining questions can be asked when more knowledge is gained from the subject. After interviews, a wrap up and discussion session will be held about the interview results with customer support team, where team members can complement their answers.

In interview, there are seven interview questions, which can be found in the

Appendix A. Interviews are conducted remotely because of the COVID-19 situation, but it shouldn't have an effect on the end result. Each interview will last 30 to 60 minutes. Interviews are recorded and the notes will be made afterwards, since making notes during an interview interferes with concentration and some points may even be forgotten to note down.

The goal of the interviews is to answer on the second research question (RQ2): "What are the important parts of the IT infrastructure that should be monitored?" In addition, some preferred features of the monitoring system will be researched. For example, the stuff that is wanted on the dashboard and the preferred alerting methods.

5.2.2 Plan for the Implementation Work

BCB Medical's IT infrastructure is harnessed with monitoring by using the methods described in Chapter 2. Monitoring is planned and implemented based on the results of the interviews. After implementation, the gathered monitoring data will be analysed with a proper anomaly detection method and it will be used to set threshold values for alerts.

In addition, predictive monitoring will be studied. There is no strong literature background on the predictive monitoring, but in this thesis it will be studied, if it is possible to predict future anomalies from monitoring data. Speculation about predictive monitoring is in Chapter 7.

6 Research Results

In this Chapter, the research results and the realized workflow of implementing the monitoring system, are presented. The Section 6.1 contains research results and the Section 6.2 contains the implementation work that was done to build the monitoring system for BCB Medical. In addition, in Section 6.3 relevant topics on the future of monitoring in BCB Medical are discussed.

6.1 Interview Results

In this Section, the interview results are presented. At the beginning, in the first Subsection, the practicalities of interviews will be introduced. Next, in the second Subsection, BCB Medical's IT infrastructure is declared to make it easier for a reader to understand the interview results. In the last part, in Subsection 6.1.3, the important parts of BCB Medical's IT infrastructure are defined, based on the interview results.

6.1.1 Practicalities

Interviews were held as was planned in the Section 5.2.1. Interviews lasted approximately 40 minutes on average, like was planned. Notes on the interviews were made afterwards from the recording, which turned out to be a successful way to work. Remote meetings didn't cause any problems, instead they were easier to organize because the interviewee and the interviewer didn't need to be at the office at the same time.

Interviews were completed in two weeks and all the customer support team members (4 interviewees) participated in the interview. All interviewees were roughly in step with their answers. For some questions there were a few different points of view but none of the answers denied other answer, so there was no strong contrast in that way.

6.1.2 Introduction to BCB Medical's IT Infrastructure

In this Subsection, BCB Medical's IT infrastructure is introduced on a high level to make it easier for a reader of this thesis to understand the interview results that are introduced in the following Subsection. In the next Subsections of this Subsection three main components of BCB Medicals infrastructure are introduced. The components are also presented in the architecture diagram in Figure 6.1.

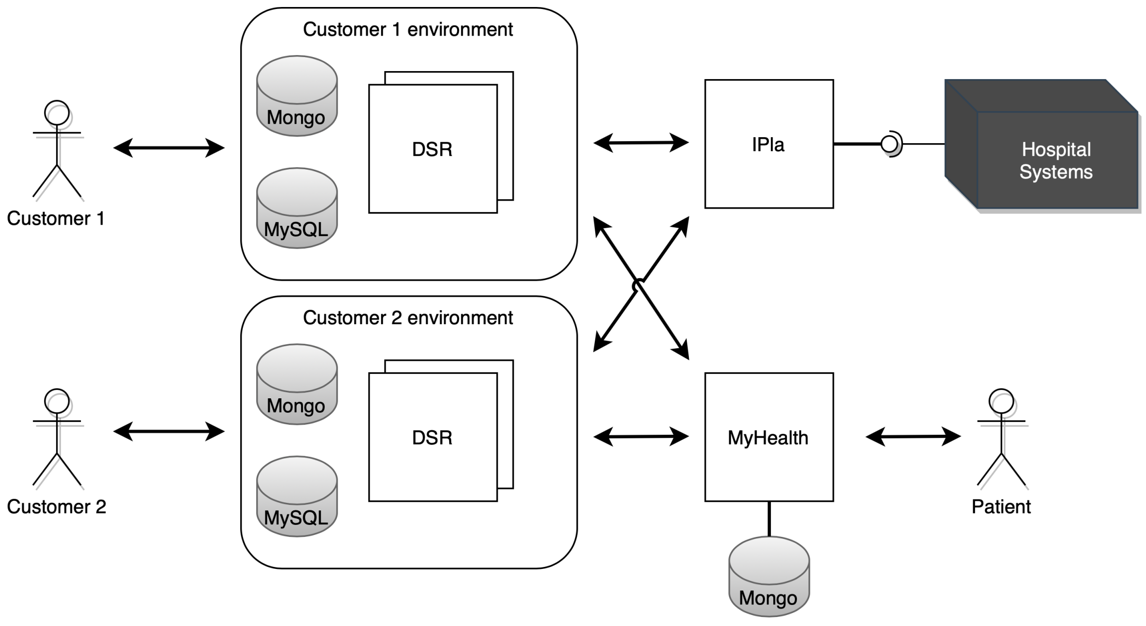


Figure 6.1: High level architecture diagram of BCB Medical's IT infrastructure.

Disease-specific registries

Disease-specific registries (DSR) cover 108 different disease groups, which include orthopaedics, oncology, neurology and cardiology, to name a few. The goal of the DSR is to monitor the effectiveness and quality of treatment, analyse clinical data and improve treatment chains. DSRs also help to reinforce the patient care by automating and harmonizing routine activities of medical treatment staff.

Integration Platform

The integration platform (IPla) takes care of integrating DSRs with hospitals' patient data systems. The IPla also transfers the treatment, surgery and examination reports generated in DSR to the medical record systems. Other things that the IPla is transferring are: Context management and user data, patient details, appointment information, laboratory information, operating room information, intensive care information, medication information and much more. IPla has transferred over 300 million integration messages this far.

MyHealth

The MyHealth service provides a forum for patients, where they can tell about their state of health by answering online questionnaires. Patients can also receive self-care instructions from MyHealth service. The MyHealth service also serves the needs of medical care professionals in gaining the general picture of the patient's health state prior to the arrival for treatment and from post-treatment recovery.

6.1.3 Important Parts of BCB Medical's IT Infrastructure

In this Subsection the important parts of BCB Medical's IT infrastructure are defined based on the interview results. Each subsection of this Subsection is an interview question, where the responses for a question are summarized into a unified answer or answers to the problem that the question states.

Problems that Are Difficult to Solve

Three of the four interviewees were on the opinion that the sign-in problems are difficult to solve. In interviews, a wide scale of sign-in related problems raised. The sign-in with a password, the single sign-on, the amount of signed in users in a system and the first sign-in of a day, were problems that were mentioned in the interviews.

The reasons, why the sign-in problems are difficult to solve, were also discussed in the interviews. One feasible explanation was that the distributed structure of the service, that handles sign-ins, is causing the problem. Especially, when there is no centralized logs available. Instead, logs are separated in different servers, which makes following a single sign-in transaction difficult. In some cases, the situation is even worse if the correct logger is not turned on, there is no logging data available from the sign-in transaction, which makes it impossible to solve problems related to it.

Three of the four interviewees mentioned also that MyHealth related problems are difficult to settle. MyHealth is sending notifications via text message and e-mail and the questionnaires can be answered on the web. The problems that were mentioned in the interviews were: questionnaire activation problems, a patient can't answer the questionnaire or the questionnaire answer was not received to the system, problems with using MyHealth through the hospital's own platform, problems with MyHealth notifications and configuration problems in a system that uses MyHealth, which prevents MyHealth to work as it should.

Possible explanations for the difficulty of solving MyHealth related issues are, again distributed structure and the fact that BCB Medical's customer support team can't be in contact directly with the patients, which makes the formation of problem description difficult.

Frequently Occurring Problems

Half of the interviewees said that the sign-in problems belong to the frequently occurring problems. Other frequently occurring problems that were mentioned, but not by multiple interviewees, were: missing components(for example, certain knee prosthesis) or equipment (used in surgeries), MyHealth problems, Integration problems and problems where user has entered faulty information to the system and is not able to remove it.

Problems that Prevent Customer from Using Product

All the interviewees thought that sign-in problems belong to the group of problems that prevent a customer from using the product. All the interviewees also said that DSR service being down prevents a customer from using the product. Three of the four interviewees mentioned that sluggishness of the DSR will also prevent the customer from using the product. Half of the interviewees said that DSR's erroneous state (OOPS-error) is an error that prevents the use of the service.

Parts of the System that Need More Transparency

Answers that came up for the question, about the parts that need more transparency, were not as unified as the answers for the previous questions, but the following things were mentioned. DSRs that are working slow or going down regularly would need more transparency. MyHealth service overall needs more transparency, including MyHealth questionnaire activations, sent SMS (Short Message Service) messages, sent emails and received questionnaire answers from patients. Status of the virtualization platform that DSRs run on. OOPS-errors (Errors that show the "Oops something went wrong" page for the user), including faster notification and easier root cause exploration for erroneous states that DSRs face.

Worst Case Scenario

All the interviewees stated that data corruption would be one of the worst cases that could happen. In addition, there were other answers that came up for this question as well. One of the topics discussed was data security, which has gained a lot of popularity recently, when 32 000 patient records were stolen in Vastaamo security breach [57]. Data loss, because of a server or service malfunction. Wide scale simultaneous sign-in problems, for example if a single sign-on platform gets into a faulty state. Problems in a virtualization platform that is virtualizing the environment for DSRs, if the virtualization platform fails, all DSRs would go offline.

Conclusion

To conclude this subsection the second research question (RQ2) will be answered. The prioritized list of the parts that the monitoring will focus on is:

1. Single sign-on problems,
2. sluggishness of DSRs,
3. MyHealth questionnaires.

6.1.4 Preferred Monitoring Features

In this Subsection the preferred monitoring features of customer support team members, according to the interviews, are introduced. The interview contained two questions on this topic and those questions are the topics of the subsections of this Subsection.

All the interviewees were unanimous on the fact that the monitoring shouldn't require staring or checking some dashboard on a regular basis. Instead, a working alert system should be in place and only when the alert is triggered support engineer can go to check the dashboard to figure out what is wrong and how to correct the faulty state of a service.

What Actions Needs to Be Alerted on?

Things that the members of customer support team would like to have alarms are introduced next. First of all, the critical things that need immediate human action should be alerted. Those include, service being down, service being slow, OOPS-error and sign-in problems. Basically, all the problems that prevent customer from using the product or significantly lower customer satisfaction should be alerted. On server level memory and CPU exhaustion are usually signs of service working slowly or about to go down, so those metrics could be used in alerting.

How the Alarm Is Made?

Common preference was that personal e-mail for the person on call would be better than e-mail to the whole support team. Usually, e-mails sent to the team can easily remain unnoticed by anyone, when everyone things that someone else will take care of the problem. One option that came up from interviews was a text message to the support phone, because someone is always responsible for the support phone. Another option was a Slack-message on a channel that has notifications switched on.

All in all, the conclusion was that the alert notification channel is not the greatest problem in the beginning, instead, setting the correct alert thresholds values in a way that the alerts are not spammed all the time, but still catch the important problems, is the way to build a successful alerting system. The alert notification channel can even be changed later on if team notices that some other way to make alerts would be more suitable for them.

6.2 Implementation Work

In this Section, the implementation work for this thesis is described. In the Subsection 6.2.1, the plan for a monitoring system is revealed. On a high level, the monitoring system is going to use Prometheus as a data collection and (temporary) data storage tool. Alertmanager as an alerting tool. Grafana as a visualization tool. Loki for a log management and discovery. In the Subsection 6.2.2, the building of a monitoring system is documented. In the Subsection 6.2.3, the alert threshold values will be set based on the monitoring data gathered from a system. Subsection 6.3.1, introduces and compares automated methods of setting threshold values for alerts on a conceptual level.

6.2.1 Planning the Implementation

The monitoring system is implemented based on the interview results that were introduced in the Section 6.1.3 and in the Section 6.1.4. The monitoring system is implemented on the dedicated monitoring server, first in the QA (Quality Assurance) environment, from there the working concept is going to be copied to the production when it is tested and proved to be secure.

All the installations that are going to be made more often than twice, are made with Ansible. Ansible is a tool that can be used for IT automation. The idea is that the installation scripts are made once, they are documented on the version control system and then anyone with access to the Ansible project can make installations with the tool. However, the real benefit comes from the automated installation for multiple hosts. For example, on all the DSR servers, the node-exporter, or any other exporter can be installed with single command.

A big part of the implementation is documentation of what is done. The documentation is made with Asciidoc and Antora. Antora is a site generator and Asciidoc is the markup language that Antora uses. Antora is made for technical documentation and the benefits that their site lists are: "contents are stored in a version control system; content, configuration and presentation are separated; automation is used in compilation, validation and in publishing; reuse of shared materials" [58].

The problems to be solved with monitoring, in prioritized order, were:

1. Single sign-on problems,
2. sluggishness of DSRs,
3. MyHealth questionnaires.

Architecture diagram of the monitoring system to be implemented that solves the defined problems is described in the Figure 6.2.

For single sign-on problems, the plan is to use centralized logs with Grafana Loki and Promtail. Promtail is an agent that forwards the system log files and catalina logs from the monitored server to the Grafana Loki as described in Figure 6.3. With Grafana Loki logs can be filtered, which aids to find the important information from logs that are fetched from several services. The difficult part in solving single sign-on problems is that it is time-consuming to recognize at which point of the sign-in process the failure happens. With Promtail and Grafana Loki, log files can be easily studied and the metrics of successful and failed sign-ins can even be exported as metrics based on the logged events.

For sluggishness and crashing of DSRs, the plan is to use node-exporter to expose the server metrics, blackbox-exporter to poll the user interface response times, Mysql-exporter and Mongo-exporter to monitor the databases and response times of the database queries. In addition, the application can be monitored with JMX-exporter, which allows to expose the metrics of the Java virtual machine.

MyHealth related problems are tackled with blackbox-exporter, which can be used to check that all pages are available and responding quickly. Activations, messages and answers can be monitored with custom-made Mongo-exporter that allows to show the metrics based on the database queries. In addition, the frontend application can be monitored with an exporter that reveals Grails metrics since it is implemented with the Grails 3 framework. The frontend monitoring provides needed help on solving the problems that patients are facing when answering the MyHealth questionnaires.

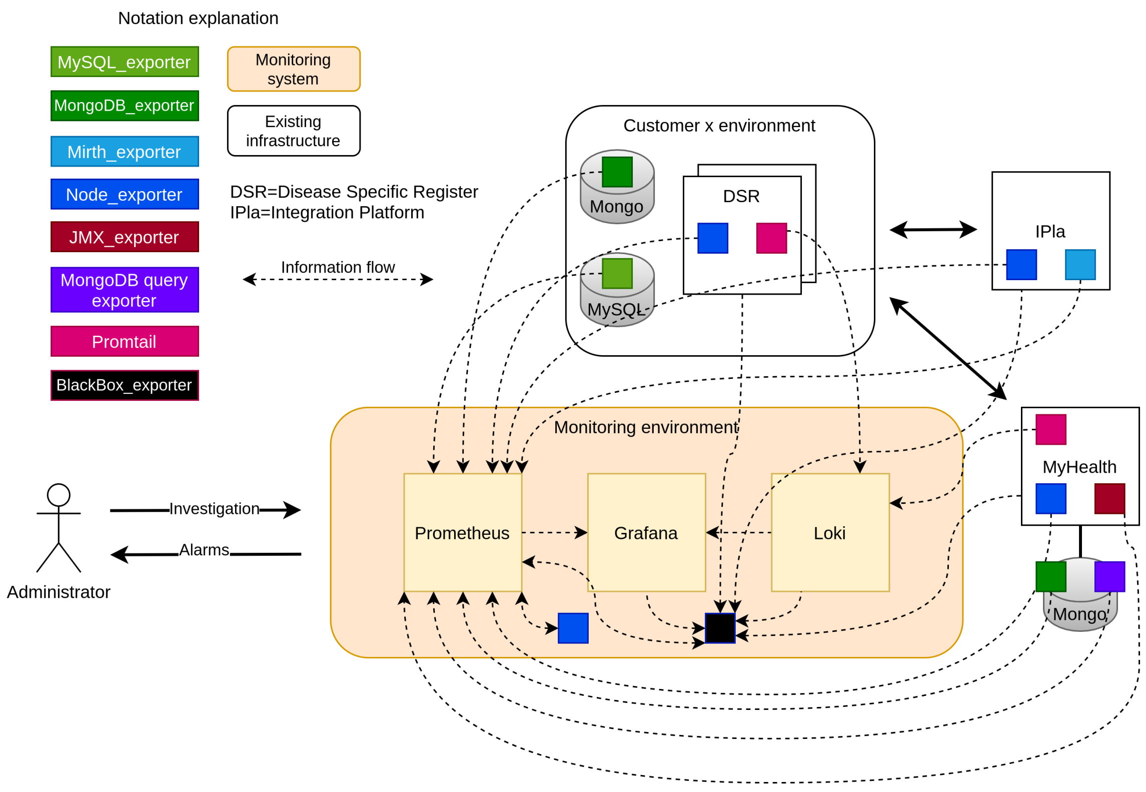


Figure 6.2: Architecture diagram of the monitoring environment to be implemented for BCB Medical.

6.2.2 Implementing the Monitoring System

The implementation work started with installing the required software in the Monitoring server ("Monitoring environment" in Figure 6.2) and documentating the installations. Default configurations and methods to update the configurations to Monitoring server were also made. After the Monitoring environment was up, I

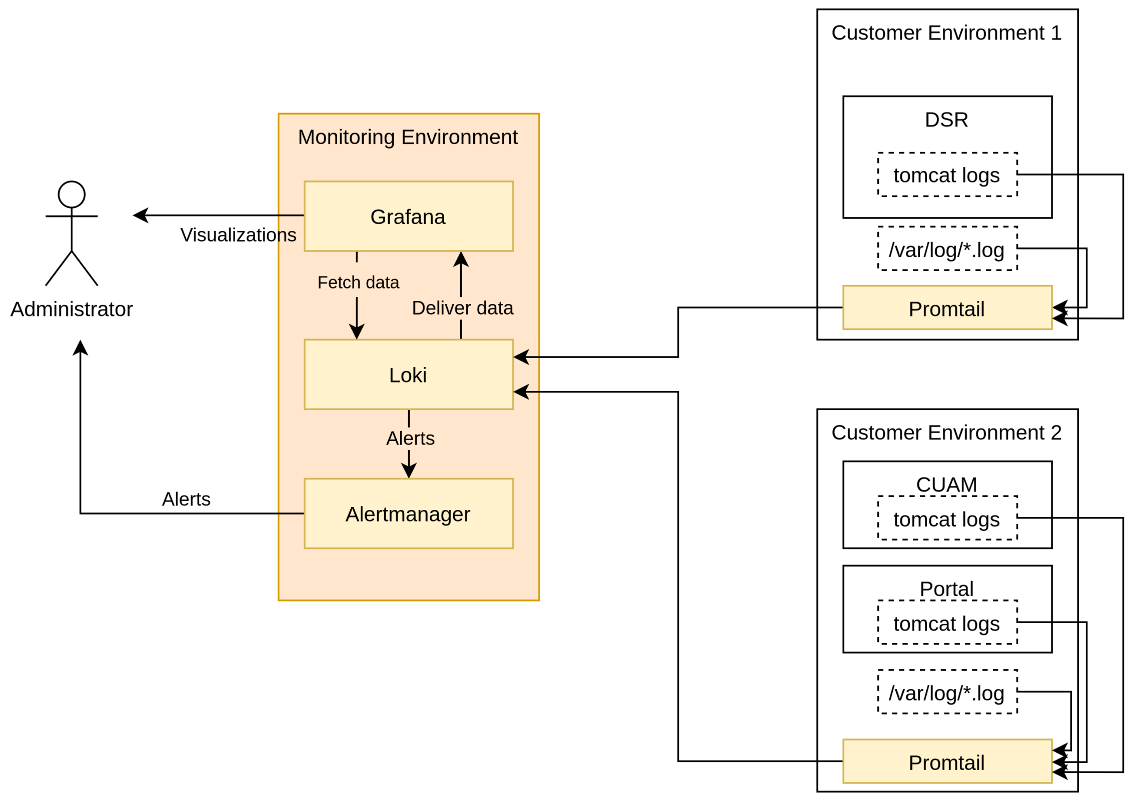


Figure 6.3: Log monitoring with Promtail and Loki in BCB Medical.

started to work on the Ansible scripts to install exporters on the monitored environments. Great help in building of Ansible scripts was offered by the Ansible Galaxy website [59], which provides open source Ansible installation Roles for different software, including Prometheus exporters. Of course, those Ansible Roles didn't fit out of the box to BCB Medical's infrastructure, but they could be used as a template when building the Roles.

The created roles were first tested in one general test environment and then used to install the exporters in the actual test environments of the environments that were defined by the interviews to be the environments that need to be monitored. The exporters were configured to Prometheus and Prometheus was configured as a data source to the Grafana. In addition, Loki was also configured to Grafana to import logs from monitored environment.

In the last step the Grafana dashboard for customer support team use was created and it can be seen in Appendix C. Firstly, the dashboard contains centralized logs to solve the single sign-on problems. The logs include a filter functionality with keyword and time range. Secondly, the dashboard contains server, JVM and database metrics to debug the reasons for the sluggishness of DSR. Thirdly, the dashboard will contain in the future metrics of activated, deactivated, sent and answered MyHealth questionnaires in real time. At this moment, it was not possible to implement MyHealth monitoring, because exporter's Mongo-plugin didn't have support for the Mongo version that BCB Medical is using in MyHealt Mongo database. However, this problem will be resolved in the future when MyHealth's Mongo database is updated.

In addition to the problems solved in the scope of this thesis, the implemented monitoring system allows solving a wide scope of problems that are not yet defined. Grafana dashboard can be easily modified to serve new needs. New metrics can be fetched from new instances easily with the installation scripts made in Ansible. Correct configurations can be set from the documentation made with Antora.

Something that needs to be further developed is the service discovery mechanisms. When the amount of monitored services increase, there needs to be an automatic way to generate a Prometheus configuration file that defines the new monitored services. Prometheus provides many methods for service discovery and one simple solution is to use Ansible to fetch all the services that have exporters installed and form a configuration file of the services. The Ansible data fetch can be done regularly or with every new exporter installation, to make sure that everything is monitored.

6.2.3 Setting Threshold Values for Alerts

Threshold values were set based on the monitoring data that was gathered. For single sign-on problems, since the problem can only be discovered from the log files, Loki's functionality was used to create metrics from log files. The functionality works in a way that certain string can be searched from logs and the occurrences of that string will be sent to Prometheus. That way the failed and succeeded single sign-ons could be tracked and the threshold value for alerts were set that if over 5% of the single sign-ons in an hour are failed, the alert will trigger.

Sluggishness of DSR's was analysed and the threshold was set on JVM heap memory. It was discovered that when the heap memory exceeds 90% the DSR couldn't recover from that so the alarm was set to trigger on the 90% use of heap memory.

As stated earlier, MyHealth questionnaire activations, deactivations, sent and answered questionnaires were not monitored at this time, but will be monitored in the future, when the database version gets updated. The plan to set threshold values for MyHealth is first to monitor the activity and then set a correct buffer for alerts to avoid false alarms. Overall, MyHealth questionnaire activations depend on the DSR that is monitored, so the generic threshold value setting is difficult. To this kind of threshold values automated anomaly detection could be used, which is introduced in the Subsection 6.3.1.

6.3 Further development

This section discusses on topics that are relevant to the future of the company. The future will hold growth in the amount of monitored services and increased involvement of Kubernetes as the environment to run different services. To the increased amount of services automated anomaly detection is utilized and the scalability of

Prometheus will be inspected. How Kubernetes changes the monitoring and how monitoring can aid in more efficient resource utilization in Kubernetes. In addition, the information security challenges in Prometheus will be inspected, which is necessary in order to fulfil the near future goals of production ready monitoring.

6.3.1 Automated Anomaly Detection

Automated anomaly detection can be used to set alert threshold value for metrics that the threshold value is complicated to set on. This kind of metrics are changing over time or have strong seasonality. For example, MyHealth questionnaire activations and deactivations mentioned in previous Subsection, have strong seasonality depending on the time of the day and on the day of the week. In addition, MyHealth activations differ based on the customer hospital that is using the product.

If the automated anomaly detection is wanted to take in use for BCB Medical, the easiest solution would be to use a ready-made open source tool for the purpose. The Yahoo EGADS system is available on Github [60]. It can be used to analyse time series data and spot anomalies of any kind. Yahoo EGADS system works out of the box only for an exported data batch, in order to integrate it as part of the Prometheus monitoring system, would require some additional work. Another approach to the anomaly detection is provided by Prometheus Anomaly Detection project in Github [61]. It provides simpler solution for Prometheus monitoring systems to use anomaly detection in threshold setting. Prometheus Anomaly Detection is based on the same principle of predicting the value of a given metric and then comparing it with the real value of the metric and if the difference is big enough, the alert will be triggered.

Difference between these two anomaly detection frameworks is that the Prometheus Anomaly Detection uses simpler machine learning algorithms (Prophet and Fourier) to train the models, which may lead inferior results when compared in wide variety of metrics. However, Prometheus Anomaly Detection is a lot easier to use, it requires just metrics as input and it provides predictions as output, which can then be fetched by Prometheus and compared in Prometheus or in Grafana.

6.3.2 Cyber Security Challenges in Prometheus

This Section focuses on the security of monitoring data transfers and endpoints. Therefore, two important subjects discussed in this Section are TLS and authentication. TLS is a cryptographic protocol that provides communication security for a computer network in the transport layer [62]. Authentication is a method of checking that the user is who he is claiming to be. One method of authentication is the basic authentication, which means that on the header of a web request a username and password is sent to the server, to authenticate the sender.

Without TLS, all monitoring data is sent in plain text from exporter to a monitoring system and anyone who has access to the network can monitor the data sent. Without authentication, anyone in the network can fetch metrics from the endpoint. The lack of TLS opens up also many vulnerabilities to cyber attackers to take advantage on. For example, Man in the middle (MITM) attack is possible where the state of monitored service can be manipulated to look different from the real state.

In order to make Prometheus production ready, before mentioned cyber security challenges need to be resolved. As it is stated by Brian Brazil, one of the Prometheus developers, in 2016 "Prometheus is a monitoring system, not a security framework. Accordingly given the massive work that'd be involved in creating and maintaining a security framework, we've decided to instead leave the task up to 3rd party systems such as Nginx and Apache" [63].

Time has past since the post of Brian Brazil and it is not completely true any more. Prometheus is starting to provide TLS encryption and basic authentication for its official exporters and tools, like Node exporter and Promtail. However, the

Prometheus API and UI won't provide TLS or authentication, so a proxy server, like Nginx or Stunnel, needs to be used in that part. In addition, there are a large group of exporters that don't provide TLS or basic authentication yet, one of those is the JMX exporter. The reason for JMX exporter not to provide TLS encryption is based on the fact that it needs to support different Java versions, since it is working as a Java agent, and different Java versions support different TLS versions, which makes it complicated to make a general solution. [64] Options for this problem is to build a custom support for TLS and basic authentication for the exporter or use a proxy server. If considered maintainability wise, the more durable solution would be to use a proxy server, to avoid checking the compatibility of self-build TLS encryption and authentication with every version update.

One example of providing security for Prometheus with Stunnel is described in a blog post "Authentication and encryption for Prometheus and its exporters" [65]. Stunnel is a proxy server that provides authentication with private client certificate and public CA certificate. Authentication with certificate is more secure than the basic authentication that requires sending the username and password in every request.

For cyber security it is also important to make sure that the monitoring tools can be updated regularly. Many monitoring tools are distributed as binary packages. The management of installed monitoring tools is difficult when tools are installed from binary packages. For example, if a security exploit is found in certain Prometheus exporter. Then it would be important to know what versions of exporters are installed on the different instances, in order to update them. One solution for the problem is provided by the package managers of OS. For example, in Ubuntu exporters could be packed in Debian packages for easier maintenance.

6.3.3 Scalability of Prometheus

The biggest scalability issue in Prometheus is the data storage. The data storage shipped with Prometheus is established as a local database on the same machine where Prometheus instance is running. Meaning that the data store is not replicated or clustered, which makes it unreliable and not able to scale infinitely. Reliability and scalability are two features that one would hope for long term data storage to have. For those reasons it is stated, also on the Prometheus documentation, that the database that Prometheus offers is meant for temporary storage only [66].

To define the needed data storage space, Prometheus documentation provides a formula:

$$NDS = RTS * ISPS * BPS,$$

where NDS = Needed Disk Space, RTS = Retention Time Seconds,

ISPS = Ingested Samples Per Second, BPS = Bytes Per Sample

The formula can be used when defining if the local data store is enough for the infrastructure that is monitored. [66]

When it turns out that local data store is not enough for the infrastructure, Prometheus provides API for remote data storage. One of the most popular systems to provide scalability and long term data storage for Prometheus is a system called Thanos. Thanos contains external data storage that is replicated and clustered, which provides the needed scalability and reliability that Prometheus alone was missing. In addition, Thanos provides a way to manage multiple Prometheus instances, which helps to scale the metric collection as well.

Thanos can be used in two ways with Prometheus, either with a sidecar or with a receiver. In sidecar use, Thanos sidecar needs to be installed on the Prometheus server that reads the local database of Prometheus and exports the data to external storage space as described in Figure 6.4. Thanos receiver is an external component and it uses Prometheus API to fetch the monitoring data from Prometheus instance.

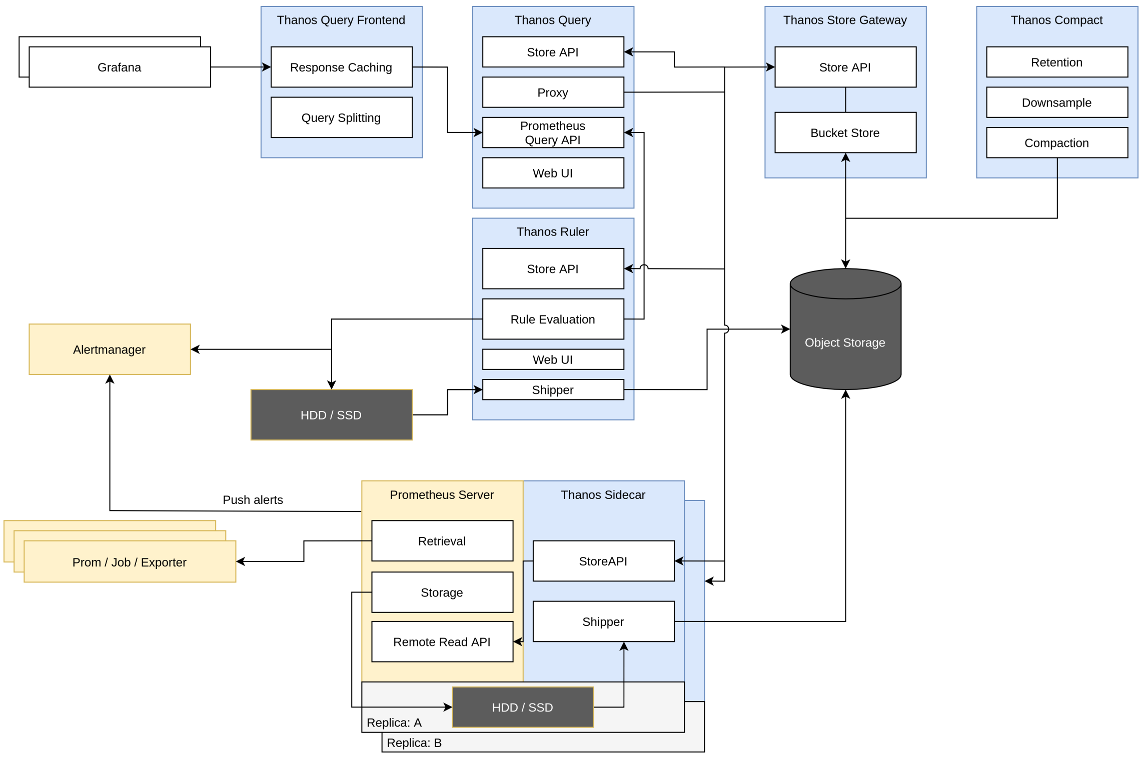


Figure 6.4: Architecture diagram of Thanos in Sidecar use [67].

6.3.4 Monitoring in Kubernetes with Prometheus

The popularity of Prometheus is partly based on the fact that it works well with Kubernetes (K8s). Kubernetes is an open-source container orchestration service developed originally by Google. Kubernetes helps deployment, management and scaling of containerized applications. The advantages of monitoring in Kubernetes environment versus server environment are in service discovery and in automated problem solving.

In server environment there is no overall system that manages everything, so usually service discovery needs to be made by hand or by a management tool, like Ansible, that fetches the information from servers and creates a Prometheus configuration file based on the information. In Kubernetes all running pods can be automatically discovered, which helps the configuration of Prometheus.

Another advantage of Prometheus in Kubernetes is possibility to auto-scale the resources based on the load. Auto-scale will make the resource utilization more efficient when resources won't be wasted in parts where they are not needed. For the deployment of auto-scaling, there are ready-made tools available, like Prometheus Adapter [68]. Prometheus Adapter pulls metrics form Prometheus and uses them to control Pod Autoscaler of Kubernetes to increase or decrease the amount of resources based on the load.

7 Conclusion

In this Chapter the conclusion to the research will be given in the form of solutions to the research questions stated in Chapter 1. This Chapter also discusses the topic that is relevant to the future research in the field of monitoring. It can be assumed that future research in monitoring will contain ML (Machine Learning), because ML and AI (Artificial Intelligence) methods are evolving and becoming more common in a wide range of applications. In monitoring ML could be used in predictive monitoring as described in Section 7.2.

7.1 Solution to Research Problem

Research question one (RQ1) stated the question "How to build an effective monitoring system with OSS tools?". The plan to answer for RQ1 was to conduct a literature review in the monitoring books and in the scientific articles published related to monitoring. In addition, lighter materials were used, like blog posts from the professionals of the field. Altogether, literature review Chapters 2, 3 and 4 provide good general background information for building an effective monitoring system. Those Chapters also provide many options to choose for, in different parts of monitoring, and present reasoning for each option.

Second research question (RQ2) asked "What are the important parts of the company's IT infrastructure that should be monitored?". The important parts of the company's IT infrastructure were defined with a qualitative research method, which in this particular case turned out to be interviews. The interview results are presented in Chapter 6 and they were used to forming a prioritized list of problems that the monitoring system being built will first to focus on. The list contained following subjects: Single sign-on problems, the sluggishness of DSRs and MyHealth questionnaire activations.

The third research question (RQ3) introduced the problem of setting threshold values in a large group of services with question "How to detect anomalies from monitoring data to set threshold values for alerting in a scalable way?". This problem was little more complex and in order to answer for the problem, firstly, the theory behind the topic needed to be looked into in the form of literature review and secondly, it needed to be proved that the theory could be used in the domain of the company. Scientific publications on the topic were found and the monitoring system was being built for the company in order to find out if the theory could be used in the company to set threshold values for alerting in a scalable way. The implementation of automated anomaly detection and then the analysis of the system turned out to be too laborious for the scope of this thesis, so the use of automated anomaly detection was left on the level of concept and the best fitting tool for the company would be the Prometheus Anomaly Detector as described in the Section 6.3.1.

The final research question (RQ4) was defined to add the research value of this thesis and it asked "Is it possible to predict the moment of time in the future, when the system is next going to face an anomaly from a monitoring data gathered in the past (a.k.a. predictive monitoring)?" This problem was only shortly discussed in the literature review part, because there were very little written on the topic in the scientific literature. However, in this Chapter, in the last Section of this Thesis, Section 7.2, the topic of predictive monitoring will be discussed and analysed.

7.2 Predictive Monitoring

Predictive monitoring is based on the idea that from the change of monitored metrics it is possible to predict a failure in software. Prediction in general is usually based on the prior knowledge of monitored subject. Therefore, the prediction can only be made on the cases that the system has faced before. Making predictions in software is easier than in a real world because software has a limited set of variables that has effect on the system and a limited set of states that the system can end up. However, the system to face all possible ways to fail naturally, would take a really long time, and whenever the system is updated, the field of ways to fail changes.

One solution to the problem can be found in the game of chess. The possible states of software are the plays that one can make from the initial setup of chess game and the failure of the software is comparable to winning the chess game. The first successful AI to play chess was IBM's chess robot named Deep Blue [69]. Deep Blue beat the reigning chess world champion Garry Kasparov in 1997. The operation principle of Deep Blue was based on the raw processing power of calculating different options for the moves and then use predefined plays that were made by chess grandmasters. In a same way, the possible states of software could be computed with raw processing power and then evaluated with predefined principles if the program state is faulty. However, this would be the waste of computing power and resources when there might be a better option available.

In 2017 a computer program called AlphaZero was developed to beat the former chess machine champion. The former chess machine champion was based on the same operation principle as Deep Blue [70], so it will be called as Deep Blue 2.0 in this text. AlphaZero had only 1/100th of the computing power that Deep Blue 2.0 had. However, AlphaZero used a different operation principle. AlphaZero used machine learning to learn the best play strategy, by playing against itself. The only predefined thing in AlphaZero was the rules of the game of chess. When AlphaZero and Deep Blue 2.0 were set head-to-head in the match of 100 chess games, AlphaZero lost none of the games. AlphaZero won 28 games and the rest of the games were drawn.

My theory is that similar method of machine learning could be used in predictive monitoring that was used in AlphaZero. The monitored software could be used by AlphaZero in a test environment and AlphaZero would find different methods of breaking the software. Then the monitoring metrics that preceded the breaking point would be learned by another machine learning model that would then spot similar signs in production environment and interpret them as the signs of software to fail in the future. The machine learning model could also predict the time of failure in future if no corrective measures is taken.

References

[1] M. Julian, *Practical Monitoring: Effective Strategies for the Real World.* 2017, p. 229.

[2] M. Andersen-Gott, G. Ghinea, and B. Bygstad, “Why do commercial companies contribute to open source software?”, *International Journal of Information Management*, vol. 32, no. 2, pp. 106–117, Apr. 2012, ISSN: 02684012. DOI: `10.1016/j.ijinfomgt.2011.10.003`.

[3] *GitHub - prometheus/node_ exporter: Exporter for machine metrics.* [Online]. Available: `https://github.com/prometheus/node%7B%5C_%7Dexporter` (visited on 10/08/2020).

[4] B. Gregg, *CPU Utilization is Wrong.* [Online]. Available: `http://www.brendangregg.com/blog/2017-05-09/cpu-utilization-is-wrong.html` (visited on 11/22/2020).

[5] *How does the OOMkiller decide which process to kill first*, 2014. [Online]. Available: `https://unix.stackexchange.com/questions/153585/how-does-the-oom-killer-decide-which-process-to-kill-first` (visited on 11/09/2020).

[6] *Out Of Memory Management*, 2004. [Online]. Available: `https://www.kernel.org/doc/gorman/html/understand/understand016.html` (visited on 11/09/2020).

[7] N. J. Gunther, “Understanding Load Averages and Stretch Factors”, Tech. Rep., 2007. [Online]. Available: `http://www.perfdynamics.com/Papers/LoadAvg2007.pdf`.

[8] B. Gregg, *Linux Load Averages: Solving the Mystery*, 2017. [Online]. Available: `http://www.brendangregg.com/blog/2017-08-08/linux-load-averages.html` (visited on 11/09/2020).

[9] I. Malpass, *Measure Anything, Measure Everything - Code as Craft*, 2011. [Online]. Available: `https://codeascraft.com/2011/02/15/measure-anything-measure-everything/` (visited on 11/09/2020).

[10] E. Baez, *StatsD Lets You Measure Anything in Your System—Here’s How*, 2020. [Online]. Available: `https://www.scalyr.com/blog/statsd-measure-anything-in-your-system/` (visited on 11/09/2020).

[11] M. Brattstrom and P. Morreale, “Scalable Agentless Cloud Network Monitoring”, in *Proceedings - 4th IEEE International Conference on Cyber Security and Cloud Computing, CSCloud 2017 and 3rd IEEE International Conference of Scalable and Smart Cloud, SSC 2017*, Institute of Electrical and Electronics Engineers Inc., Jul. 2017, pp. 171–176, ISBN: 9781509066438. DOI: `10.1109/CSCloud.2017.11`.

[12] Case, Fedor, Shoffstall, and Davin, *RFC 1067 - Simple Network Management Protocol*, Aug. 1988. [Online]. Available: `https://tools.ietf.org/html/rfc1067%7B%5C#%7Dpage-2` (visited on 11/19/2020).

[13] S. Newman, *Building microservices: designing fine-grained systems.* O’Reilly media, 2016, pp. 1–8.

[14] J. Turnbull, *The Art of Monitoring.* 2016, p. 780.

[15] J. Volz, *Pull doesn't scale - or does it? | Prometheus*, 2016. [Online]. Available: https://prometheus.io/blog/2016/07/23/pull-does-not-scale-or-does-it/ (visited on 09/17/2020).

[16] C. Sridharan, *Distributed systems observability.* O'Reilly media, 2018, p. 36. [Online]. Available: https://www.humio.com/free-ebook-distributed-systems-observability.

[17] S. Banerjee, H. Srikanth, and B. Cukic, "Log-based reliability analysis of Software as a Service (SaaS)", in *Proceedings - International Symposium on Software Reliability Engineering, ISSRE*, 2010, pp. 239–248, ISBN: 9780769542553. DOI: 10.1109/ISSRE.2010.46.

[18] *A beginner's guide to distributed tracing and how it can increase an application's performance | Grafana Labs.* [Online]. Available: https://grafana.com/blog/2021/01/25/a-beginners-guide-to-distributed-tracing-and-how-it-can-increase-an-applications-performance/?pg=docs (visited on 02/07/2021).

[19] *Getting started with tracing and Grafana Tempo | Grafana Labs.* [Online]. Available: https://grafana.com/go/webinar/getting-started-with-tracing-and-grafana-tempo/ (visited on 02/08/2021).

[20] A. Kulkarni and R. Booz, *Time-Series Database Analysis & Data Modeling Guide*, 2020. [Online]. Available: https://blog.timescale.com/blog/what-the-heck-is-time-series-data-and-why-do-i-need-a-time-series-database-dcf3b1b18563/ (visited on 12/07/2020).

[21] *DB-Engines Ranking per database model category*, 2020. [Online]. Available: https://db-engines.com/en/ranking%7B%5C_%7Dcategories (visited on 12/07/2020).

[22] T. Dunning and E. Friedman, *Time Series Databases: New Ways to Store and Access Data.* O'Reilly media, 2014, p. 72. [Online]. Available: `https://www.academia.edu/29891282/Time%7B%5C_%7DSeries%7B%5C_%7DDatabases%7B%5C_%7DNew%7B%5C_%7DWays%7B%5C_%7Dto%7B%5C_%7DStore%7B%5C_%7Dand%7B%5C_%7DAccess%7B%5C_%7DData`.

[23] *Why time series databases are exploding in popularity - TechRepublic.* [Online]. Available: `https://www.techrepublic.com/article/why-time-series-databases-are-exploding-in-popularity/` (visited on 03/07/2021).

[24] Rob Kiefer, *MongoDB Time-Series - A NoSQL vs. SQL Database Comparison*, 2018. [Online]. Available: `https://blog.timescale.com/blog/how-to-store-time-series-data-mongodb-vs-timescaledb-postgresql-a73939734016/` (visited on 12/07/2020).

[25] Lee Hampton, *Benchmarking Cassandra vs. TimescaleDB for time-series data*, 2018. [Online]. Available: `https://blog.timescale.com/blog/time-series-data-cassandra-vs-timescaledb-postgresql-7c2cc50a89ce/` (visited on 12/07/2020).

[26] Mike Freedman, *TimescaleDB vs. InfluxDB: Purpose-built for time-series data*, 2020. [Online]. Available: `https://blog.timescale.com/blog/timescaledb-vs-influxdb-for-time-series-data-timescale-influx-sql-nosql-36489299877/` (visited on 12/07/2020).

[27] T. Pelkonen, S. Franklin, J. Teller, P. Cavallaro, Q. Huang, J. Meza, and K. Veeraraghavan, "Gorilla: A Fast, Scalable, in-Memory Time Series Database", *Proc. VLDB Endow.*, vol. 8, no. 12, pp. 1816–1827, Aug. 2015, ISSN: 2150-8097. DOI: `10.14778/2824032.2824078`. [Online]. Available: `https://doi.org/10.14778/2824032.2824078`.

[28] *ELK Stack: Elasticsearch, Logstash, Kibana | Elastic*. [Online]. Available: https://www.elastic.co/what-is/elk-stack (visited on 03/08/2021).

[29] *The Complete Guide to the ELK Stack | Logz.io*. [Online]. Available: https://logz.io/learn/complete-guide-elk-stack/%7B%5C#%7Delasticsearch (visited on 03/08/2021).

[30] *Comparisons | Grafana Labs*. [Online]. Available: https://grafana.com/docs/loki/latest/overview/comparisons/ (visited on 03/08/2021).

[31] D. Salomon, *Handbook of Data Compression*, eng, G. Motta, Ed. London: Springer London, 2010, ISBN: 9781848829039. DOI: 10.1007/978-1-84882-903-9.

[32] R. Christensen, “Improving Compression of Massive Log Data”, Tech. Rep., 2013.

[33] *Prometheus: Monitoring at SoundCloud | SoundCloud Backstage Blog*. [Online]. Available: https://developers.soundcloud.com/blog/prometheus-monitoring-at-soundcloud (visited on 02/15/2021).

[34] *Writing exporters | Prometheus*. [Online]. Available: https://prometheus.io/docs/instrumenting/writing%7B%5C_%7Dexporters/ (visited on 02/15/2021).

[35] *Overview | Prometheus*. [Online]. Available: https://prometheus.io/docs/introduction/overview/ (visited on 02/15/2021).

[36] Š. Obetko, “Monitoring distributed systems with Riemann”, Tech. Rep., 2016, p. 63. [Online]. Available: https://is.muni.cz/th/fia5e/fi-pdflatex.pdf.

[37] *Nagios - Network, Server and Log Monitoring Software*. [Online]. Available: https://www.nagios.com/ (visited on 04/30/2021).

[38] *Prometheus vs Nagios | Logz.io*. [Online]. Available: https://logz.io/blog/prometheus-vs-nagios-metrics/ (visited on 03/20/2021).

[39] *Nagios Tutorial for Beginners: What is, Installation, Architecture.* [Online]. Available: https://www.guru99.com/nagios-tutorial.html (visited on 03/19/2021).

[40] *A Guide to Monitoring Servers with Nagios - Boolean World.* [Online]. Available: https://www.booleanworld.com/guide-monitoring-servers-nagios/ (visited on 03/18/2021).

[41] S. Few, *Information Dashboard Design: The Effective Visual Communication of Data.* 2006, p. 166.

[42] C.-h. Chen, W. Härdle, A. Unwin, and M. Friendly, "A Brief History of Data Visualization", in *Handbook of Data Visualization*, Springer Berlin Heidelberg, 2008, pp. 15–56. DOI: 10.1007/978-3-540-33037-0_2. [Online]. Available: https://link.springer.com/chapter/10.1007/978-3-540-33037-0%7B%5C_%7D2.

[43] D. E. Leidner and J. J. Elam, "Executive information systems: Their impact on executive decision making", *Proceedings of the Annual Hawaii International Conference on System Sciences*, vol. 3, pp. 206–215, 1993, ISSN: 15301605. DOI: 10.1109/HICSS.1993.284314.

[44] Peter Charles, Nathan Good, Laheem Lamar Jordan, Joyojeet Pal, "How Much Information?", 2003. [Online]. Available: https://groups.ischool.berkeley.edu/archive/how-much-info-2003/.

[45] *Graphite.* [Online]. Available: https://graphiteapp.org/ (visited on 02/05/2021).

[46] *Torkel Ödegaard - Co-Founder - Grafana Labs | LinkedIn.* [Online]. Available: https://se.linkedin.com/in/torkel-odegaard (visited on 02/07/2021).

[47] *Grafana | Grafana Labs.* [Online]. Available: https://grafana.com/oss/grafana/ (visited on 02/07/2021).

[48] *Node Exporter Full dashboard for Grafana | Grafana Labs*. [Online]. Available: `https://grafana.com/grafana/dashboards/1860` (visited on 03/20/2021).

[49] S. Venkataraman, J. Caballero, D. Song, A. Blum, and J. Yates, "Black Box Anomaly Detection: Is It Utopian?", in *HotNets*, 2006.

[50] N. Laptev, S. Amizadeh, and I. Flint, "Generic and Scalable Framework for Automated Time-series Anomaly Detection", in *Proceedings of the 21th ACM SIGKDD International Conference on Knowledge Discovery and Data Mining - KDD '15*, New York, New York, USA: ACM Press, 2015, ISBN: 9781450336642. [Online]. Available: `http://dx.doi.org/10.1145/2783258.2788611.`.

[51] Z. Lan, Z. Zheng, and Y. Li, "Toward Automated Anomaly Identification in Large-Scale Systems", *IEEE Transactions on Parallel and Distributed Systems*, vol. 21, no. 2, pp. 174–187, Feb. 2010, ISSN: 1558-2183. DOI: `10.1109/TPDS.2009.52`.

[52] S. Ligus, *The Effective Monitoring and Alerting.* O'Reilly media, 2012, p. 150.

[53] B. Brazil, *Prometheus: Up & Running: Infrastructure and Application Performance Monitoring.* O'Reilly media, 2018, p. 386.

[54] *What is Turnaround Time (TAT)? - Definition from Techopedia.* [Online]. Available: `https://www.techopedia.com/definition/23798/turnaround-time-tat` (visited on 03/30/2021).

[55] *GitHub - prometheus/alertmanager: Prometheus Alertmanager.* [Online]. Available: `https://github.com/prometheus/alertmanager` (visited on 03/31/2021).

[56] *ITIL Incident Management: An Introduction – BMC Software | Blogs.* [Online]. Available: `https://www.bmc.com/blogs/itil-v3-incident-management/` (visited on 03/31/2021).

[57] *Potilaiden tietoja vietiin psykoterapiakeskuksen tietomurrossa, yritys kertoo joutuneensa kiristyksen uhriksi - Kotimaa | HS.fi.* [Online]. Available: `https://www.hs.fi/kotimaa/art-2000006676407.html` (visited on 04/30/2021).

[58] *Antora Documentation :: Antora Docs.* [Online]. Available: `https://docs.antora.org/antora/2.1/` (visited on 03/07/2021).

[59] *Ansible Galaxy.* [Online]. Available: `https://galaxy.ansible.com/` (visited on 04/03/2021).

[60] *GitHub - yahoo/egads: A Java package to automatically detect anomalies in large scale time-series data.* [Online]. Available: `https://github.com/yahoo/egads` (visited on 04/07/2021).

[61] *GitHub - AICoE/prometheus-anomaly-detector: A newer more updated version of the prometheus anomaly detector (https://github.com/AICoE/prometheus-anomaly-detector-legacy).* [Online]. Available: `https://github.com/AICoE/prometheus-anomaly-detector` (visited on 04/07/2021).

[62] *SSL/TLS in Detail | Microsoft Docs.* [Online]. Available: `https://docs.microsoft.com/en-us/previous-versions/windows/it-pro/windows-server-2003/cc785811(v=ws.10)?redirectedfrom=MSDN` (visited on 04/30/2021).

[63] *Prometheus Security: Authentication, Authorization and Encryption – Robust Perception | Prometheus Monitoring Experts.* [Online]. Available: `https://www.robustperception.io/prometheus-security-authentication-authorization-and-encryption` (visited on 04/04/2021).

[64] *Metrics endpoint to support TLS · Issue #442 · prometheus/jmx_ exporter · GitHub.* [Online]. Available: `https://github.com/prometheus/jmx%7B%5C_%7Dexporter/issues/442` (visited on 04/04/2021).

[65] *Authentication and encryption for Prometheus and its exporters.* [Online]. Available: https://0x63.me/tls-between-prometheus-and-its-exporters/ (visited on 04/04/2021).

[66] *Storage | Prometheus.* [Online]. Available: https://prometheus.io/docs/prometheus/latest/storage/ (visited on 04/07/2021).

[67] *GitHub - thanos-io/thanos: Highly available Prometheus setup with long term storage capabilities. A CNCF Incubating project.* [Online]. Available: https://github.com/thanos-io/thanos (visited on 04/30/2021).

[68] *GitHub - kubernetes-sigs/prometheus-adapter: An implementation of the custom.metrics.k8s.io API using Prometheus.* [Online]. Available: https://github.com/kubernetes-sigs/prometheus-adapter (visited on 04/20/2021).

[69] F. H. Hsu, "IBM's Deep Blue chess grandmaster chips", *IEEE Micro*, vol. 19, no. 2, pp. 70–81, Mar. 1999, ISSN: 02721732. DOI: 10.1109/40.755469.

[70] D. Silver, T. Hubert, J. Schrittwieser, I. Antonoglou, M. Lai, A. Guez, M. Lanctot, L. Sifre, D. Kumaran, T. Graepel, T. Lillicrap, K. Simonyan, and D. Hassabis, "Mastering chess and shogi by self-play with a general reinforcement learning algorithm", *arXiv*, Dec. 2017, ISSN: 23318422. arXiv: 1712.01815. [Online]. Available: https://arxiv.org/abs/1712.01815v1.

Appendix A Interview Questions

Translated interview questions in English:

1. What kind of problems are difficult to solve?
2. What kind of problems customer reports in most cases?
3. What kind of problems prevent a customer from using the product, in other words, what kind of problems require immediate action?
4. What part of the system needs more transparency?
   (a) What would the customer support team's dashboard contain?
   (b) What other ways are there to present/transmit monitoring information besides the dashboard? Slack, e-mail?
5. What would be the worst case scenario?
6. On what actions do we want alert to be triggered?
7. How do we alert? Via Slack, via e-mail, with a text message?

Original interview questions in Finnish:

1. Minkä tyyppiset ongelmat ovat vaikeita selvittää?
2. Minkä tyyppisistä ongelmista asiakas useimmiten ilmoittaa?
3. Mitkä ongelmat estävät asiakasta käyttämästä tuotetta eli mitkä ongelmat vaativat välitöntä reagointia?
4. Minkä järjestelmän osan toiminnan läpinäkyvyyttä olisi syytä parantaa?
    (a) Mitä haluttaisiin nähdä supportin dashboardilla?
    (b) Mitä muita tapoja voisi olla esittää/välittää monitorointi tietoa kuin dashboard? Slack, s-posti?
5. Mikä on pahinta mitä voi tapahtua? (Worst case scenario)
6. Mistä asioista halutaan hälytys?
7. Miten hälyttäminen tehdään? Slack, s-posti, txt-viesti?

Appendix B Example Playbook: Demo Application

This is an example playbook of a Demo Application. The idea of the playbook is to describe the system that the alert was made on, in order to decrease the resolution time of a problem. The template for this playbook, or runbook as it is called in the book, is from Practical Monitoring book [1]. This playbook will contain, first, general information about the application, then some metadata, including service owner and codebase location, escalation procedure, external dependencies, internal dependencies, tech stack, available metrics and logs, and alerts.

B.1 Demo App

The Grails Demo App is an application that gathers demo persons to the system. Application contains own user management and it is also coupled with external **user management system** and **Demo App 2** which, works with the Demo App.

B.2 Metadata

The codebase of Demo App can be found in http://www.demogit.com/demoapp. The service owner is **Demo Owner**.

B.3 Escalation Procedure

In the case of assistance needed, the service owner has requested to be in contact with the team that he is in: **Demo Team 1** (link to the Slack channel).

B.4 External Dependencies

Application has two external dependencies. **Centralized user management** and **Demo Application 2**.

B.5 Internal Dependencies

MongoDB running in mongo-123.com

MySQL database running in mysql-123.com

B.6 Tech Stack

Backend is made with **Grails 3**.

Frontend is made with **React**

B.7 Metrics and Logs

The app emits the following **metrics**:

- User login (counter type),
- User logout (counter type),
- Logins successfull (counter type),
- Logins failed (counter type),

- User login time (timer type),
- User logout time (timer type),

The app emits following **logs**:

- Stacktrace.log
- Accesslog.log

B.8 Alerts

B.8.1 User Sign-in Failure Rate

Alert fires when the **failure rate is above 10%**.

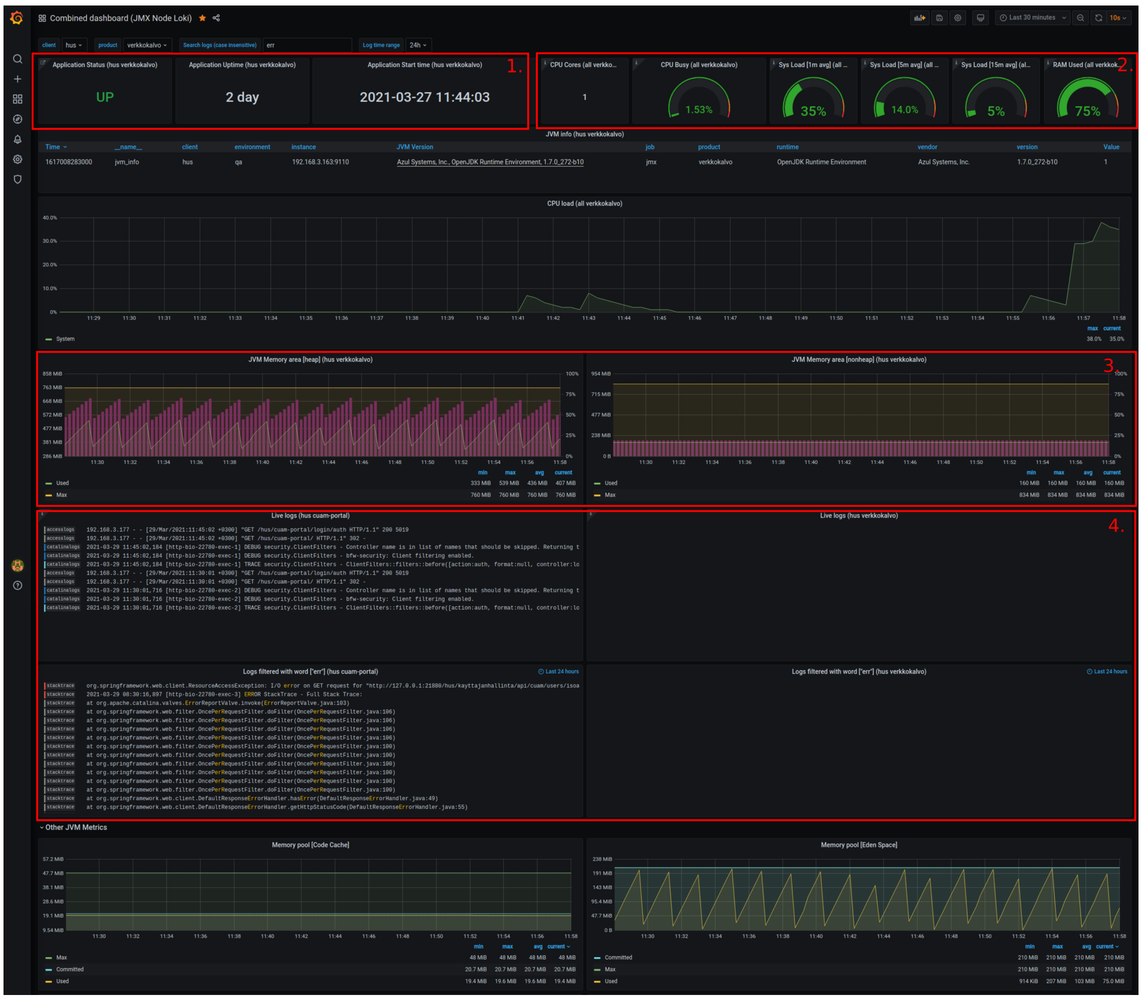
Tip: Potential cause is brute force attack.

B.8.2 User Login Time Too High

Alert fires when **login takes more than 2 seconds**.

Tip: Check database performance.

Appendix C Grafana Dashboard for Support Team


Combined dashboard (JMX Node Loki)
Last 30 minutes
10s
1.
2.
3.
4.
CPU load (all verkkokalvo)
JVM Memory area [heap] (hus verkkokalvo)
JVM Memory area [nonheap] (hus verkkokalvo)
Memory pool [Code Cache]
Memory pool [Eden Space]

client: hus · product: verkkokalvo · Search logs (case insensitive): err · Log time range: 24h

Application Status (hus verkkokalvo)	Application Uptime (hus verkkokalvo)	Application Start time (hus verkkokalvo)
UP	2 day	2021-03-27 11:44:03

CPU Cores (all verkko...	CPU Busy (all verkkokalvo)	Sys Load [1m avg] (all ...	Sys Load [5m avg] (all ...	Sys Load [15m avg] (al...	RAM Used (all verkkok...
1	1.53%	35%	14.0%	5%	75%

JVM info (hus verkkokalvo)

Time	__name__	client	environment	instance	JVM Version	job	product	runtime	vendor	version	Value
1617008283000	jvm_info	hus	qa	192.168.3.163:9110	Azul Systems, Inc., OpenJDK Runtime Environment, 1.7.0_272-b10	jmx	verkkokalvo	OpenJDK Runtime Environment	Azul Systems, Inc.	1.7.0_272-b10	1

CPU load (all verkkokalvo)

	max	current
System	38.0%	35.0%

JVM Memory area [heap] (hus verkkokalvo)

	min	max	avg	current
Used	333 MiB	539 MiB	436 MiB	407 MiB
Max	760 MiB	760 MiB	760 MiB	760 MiB

JVM Memory area [nonheap] (hus verkkokalvo)

	min	max	avg	current
Used	160 MiB	160 MiB	160 MiB	160 MiB
Max	834 MiB	834 MiB	834 MiB	834 MiB

Live logs (hus cuam-portal)

```
accesslogs    192.168.3.177 - - [29/Mar/2021:11:45:02 +0300] "GET /hus/cuam-portal/login/auth HTTP/1.1" 200 5019
accesslogs    192.168.3.177 - - [29/Mar/2021:11:45:02 +0300] "GET /hus/cuam-portal/ HTTP/1.1" 302 -
catalinalogs  2021-03-29 11:45:02,184 [http-bio-22780-exec-1] DEBUG security.ClientFilters - Controller name is in list of names that should be skipped. Returning t
catalinalogs  2021-03-29 11:45:02,184 [http-bio-22780-exec-1] DEBUG security.ClientFilters - bfw-security: Client filtering enabled.
catalinalogs  2021-03-29 11:45:02,184 [http-bio-22780-exec-1] TRACE security.ClientFilters - ClientFilters::filters::before([action:auth, format:null, controller:lo
accesslogs    192.168.3.177 - - [29/Mar/2021:11:30:01 +0300] "GET /hus/cuam-portal/login/auth HTTP/1.1" 200 5019
accesslogs    192.168.3.177 - - [29/Mar/2021:11:30:01 +0300] "GET /hus/cuam-portal/ HTTP/1.1" 302 -
catalinalogs  2021-03-29 11:30:01,716 [http-bio-22780-exec-2] DEBUG security.ClientFilters - Controller name is in list of names that should be skipped. Returning t
catalinalogs  2021-03-29 11:30:01,716 [http-bio-22780-exec-2] DEBUG security.ClientFilters - bfw-security: Client filtering enabled.
catalinalogs  2021-03-29 11:30:01,716 [http-bio-22780-exec-2] TRACE security.ClientFilters - ClientFilters::filters::before([action:auth, format:null, controller:lo
```

Live logs (hus verkkokalvo)

Logs filtered with word ["err"] (hus cuam-portal) · Last 24 hours

```
stacktrace    org.springframework.web.client.ResourceAccessException: I/O error on GET request for "http://127.0.0.1:21880/hus/kayttajanhallinta/api/cuam/users/isoa
stacktrace    2021-03-29 08:30:16,897 [http-bio-22780-exec-3] ERROR StackTrace - Full Stack Trace:
stacktrace    at org.apache.catalina.valves.ErrorReportValve.invoke(ErrorReportValve.java:103)
stacktrace    at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:106)
stacktrace    at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:106)
stacktrace    at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:106)
stacktrace    at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:106)
stacktrace    at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:100)
stacktrace    at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:100)
stacktrace    at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:100)
stacktrace    at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:100)
stacktrace    at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:100)
stacktrace    at org.springframework.web.filter.OncePerRequestFilter.doFilter(OncePerRequestFilter.java:100)
stacktrace    at org.springframework.web.client.DefaultResponseErrorHandler.hasError(DefaultResponseErrorHandler.java:49)
stacktrace    at org.springframework.web.client.DefaultResponseErrorHandler.getHttpStatusCode(DefaultResponseErrorHandler.java:55)
```

Logs filtered with word ["err"] (hus verkkokalvo) · Last 24 hours

Other JVM Metrics

Memory pool [Code Cache]

	min	max	avg	current
Max	48 MiB	48 MiB	48 MiB	48 MiB
Committed	20.7 MiB	20.7 MiB	20.7 MiB	20.7 MiB
Used	19.4 MiB	19.6 MiB	19.6 MiB	19.4 MiB

Memory pool [Eden Space]

	min	max	avg	current
Committed	210 MiB	210 MiB	210 MiB	210 MiB
Max	210 MiB	210 MiB	210 MiB	210 MiB
Used	914 KiB	207 MiB	103 MiB	75.0 MiB

1. Application status

2. Server status

3. JVM memory

4. Centralized logs