UNIVERSITY OF TARTU
Faculty of Science and Technology
Institute of Computer Science
Computer Science Curriculum

Mathias Are

Monitoring of the microservice architecture: Ridango case study

Bachelor's Thesis (9 ECTS)

Supervisor: Chinmaya Kumar Dehury, Ph.D.

Tartu 2022

Monitoring of the microservice architecture: Ridango case study

Abstract:
Monitoring is an essential part of the software production lifecycle, as it provides feedback about the state and well-being of the observed system, allowing to detect issues and to make informed decisions based on the gathered data. Whether a system is sufficiently monitored depends mostly on its purpose, but often also on its general architecture and structure. In recent years, microservice architecture (MA) has been a common choice for IT systems of all sizes, favored for the various benefits it provides that streamline the development and deployment of modern applications. The MA, however, is significantly more difficult to monitor than most of its predecessors due to the inherent complexity of its distributed and dynamically changing structure, requiring the user to navigate through the various levels of virtualization and correlate the gathered metrics between a service and the rest of the system's components. Observability tools which have also gained popularity among the software engineering community, aim to solve this problem by combining the monitoring data of metrics, logs, and traces into a single platform while providing real-time analysis of the incoming data with the help of AI models. In this study, we compare three observability tools: New Relic, IBM Instana, and Datadog in their ability to monitor microservices in a self-hosted Kubernetes cluster of a mid-sized IT firm *Ridango* based on the gathered requirements from a conducted user requirements analysis. Finally, we describe a proposed monitoring solution for the company that is adjusted to its business requirements and the particularities of monitoring the MA.

Keywords:
Software engineering, monitoring, software architecture, microservice architecture

CERCS: P170 Computer science, numerical analysis, systems, control

Mikronteenuste arhitektuuri monitoorimine Ridango näitel

Lühikokkuvõte:

Monitoorimine on oluline osa tarkvaraarenduse protsessist - see võimaldab saada vahetut tagasisidet jälgitava süsteemi seisundi ja heaolu kohta. Samuti võimaldab monitoorimine tuvastada tarkvarateenustes leiduvaid vigu ning teha teadlike otsuseid süsteemi parendamiseks. Küllaldase monitoorimise lahenduse loomiseks on tarvilik arvestada jälgitava süsteemi eesmärkidega, kuid tihti ka selle arhitektuurist tulenevate erisustega. Mikroteenuste arhitektuur (MA) on saanud viimaste aastate vältel järjest populaarsemaks valikuks infotehnoloogiliste süsteemide kujundamisel oma mitmete kasulike omaduste tõttu, mis võimaldavad kiirendada uute tarkvaraliste funktsionaalsuste arendamist ja väljastamist. Seevastu on mikroteenuste arhitektuuri oluliselt keerulisem monitoorida kui mitmeid selle alternatiive. Põhjuseks on MA hajus ja dünaamiliselt muutuv struktuur, milles on süsteemi jälgijal keeruline orienteeruda ning ühelt komponendilt kogutud andmeid kogu ülejäänud süsteemiga vastavusse viia. Tarkvaraarenduse kogukonna seas populaarsust kogunud jälgitavuse tööriistad (ingl k *observability tools*) üritavad seda probleemi lahendada tehnoloogiate abil, mis koguvad eri liiki monitoorimise andmed ning analüüsivad neid reaalajas masinõppe mudelite või tehisnärvivõrkude abil. Antud bakalaureusetöö raamest katsetatakse kolme jälgitavuse platvormi: New Relic, IBM Instana ja Datadog, ning võrreldakse nende suutlikust monitoorida tarkvaraettevõtte Ridango lokaalselt majutatud Kubernetese klastrit ning sellel töötavaid mikroteenuseid. Koostatud võrdluse aluseks on eelnevalt läbi viidud ettevõtte sisene kasutajanõuete analüüs (ingl k *user requirements analysis*). Lisaks on bakalaureusetöös esitletud Ridango ärivajadusi ning MA erisusi arvestavat monitoorimislahendust, mis on loodud *Datadog*'i platvormi põhjal.

Võtmesõnad:

Tarkvaratehnika, tarkvaraarendus, monitoorimine, tarkvara arhitektuur, mikroteenused, mikroteenuste arhitektuur

CERCS: P170 Arvutiteadus, arvanalüüs, süsteemid, kontroll

Contents

1 Introduction

The practices of developing and releasing software have changed significantly over the last few decades. The dominating mindset of a maintenance-only approach in software development with infrequent release cycles and rarely changing architectural design has been gradually replaced by paradigms like agile development, DevOps, and microservice architecture (MA) [17]. These new methodologies allow faster implementation of new features as they reduce the risk of failure while also allowing to simplify and automate most of the deployment and version integration processes. The MA, in particular, has had a considerable impact on this progress since it has allowed the development teams to decouple large applications into smaller, more maintainable chunks of code where they can introduce changes or adopt new technologies without concerning themselves with dependencies from other parts of the product [15]. Along with the virtualization tools like Docker containers and Kubernetes, the MA has become a popular choice among cloud and locally hosted systems. However, the level of complexity that the MA provides through the numerous instances of microservices all coexisting and communicating in the production environment poses a new kind of problem in the context of monitoring [15]. The total number of microservices and layers of virtualization makes it increasingly difficult to detect and mitigate errors compared to the traditional monolithic architecture of software applications. Accordingly, conventional monitoring tools designed to observe monoliths are often insufficient for monitoring MA because they provide data only about the individual components of the system and lack vital information about the transactions between them that form a cohesive application [18].

1.1 Motivation

Internal observations in the company of Ridango[1] indicate that a similar situation is taking place with their server-side microservice environments. Development teams take full advantage of the MA by decoupling services, frequently deploying new versions of software, and choosing their preferred technologies. Still, the toolset of monitoring has remained to serve the needs of traditional monolithic applications. Our primary motivation for this study is to find the tools and practices that consider the particularities of the MA and improve the monitoring setup in the firm. Observability tools have become a recent trend among the IT firms and enterprises as they offer a convenient SaaS platform to bring all the monitoring data into a single tool while providing a wide range of features and AI capabilities to actively detect issues and faults in the system. However, due to their relatively high pricing, it is hard to justify adopting them without relying on some kind of proof of concept or conducted research. Several studies have been conducted with a focus on the separate features of these tools, but there is a limited

[1]https://ridango.com/

number of published works that assess and compare these tools in a complete scope. The second motivation of this study is to test these tools on an actual technology stack and compare them with the existing monitoring setup in Ridango to better understand their value and usage scenarios.

1.2 Goal

In order to detect critical errors in their systems, increase the reliability and speed of deploying new versions of software and guarantee better visibility into the state and health of its products, many of the modern IT firms that decide to adopt the MA face the necessity to improve their monitoring solution to better address the complexity of a distributed system architecture. The goal of this thesis is to describe this kind of process as a case study within an Estonian IT company Ridango to provide an example and research material for monitoring solution redesign.

1.3 Contributions

The contributions of this thesis can be summarized as follows:

1. Conduct a stakeholder identification and user requirements analysis to map the requirements along with their priorities for all the involved parties in order to understand the internal monitoring needs within the company.

2. Test and compare 3 more widely used observability tools along the currently used monitoring tools in Ridango.

3. Construct a proposed monitoring setup for Ridango using the selected set of tools from part 2.

4. Provide an example case study of a monitoring solution redesign.

1.4 Thesis outline

The contents of this thesis are structured as follows. In this section we gave a brief overview for this thesis and introduction to the general subjects discussed in the study. Section 2 presents the background on related tools and concepts. Section 3 provides the results of the user requirements analysis. Section 4 describes the methodology and conclusions of the monitoring tools comparison. Section 5 describes the proposed monitoring setup for Ridango and proposes new workflows and conventions to maximize the value of the monitoring setup. Finally, section 6 provides the conclusion and discusses the possibilities for future work of this thesis.

2 Background

This section introduces the general concepts of monitoring, observability, observability tools, and the MA. Finally, the section gives a short overview of the IT firm Ridango and its server-side environment of microservices in Kubernetes, which is the primary focus of this case study.

2.1 The Microservice Architecture

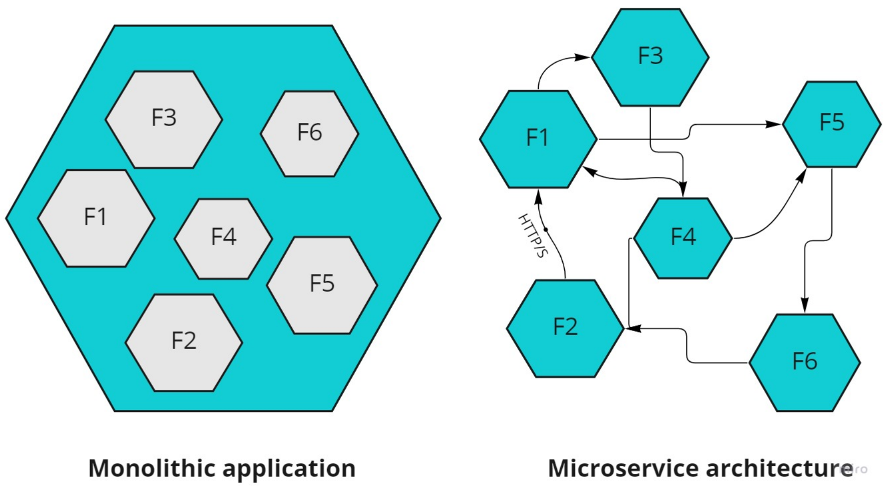

Figure 1. Monolithic application and the Microservice architecture

Microservice architecture is a software architecture paradigm that uses virtualization concepts like containerization and communication protocols like HTTP to divide an application into smaller components that together provide all the functionalities that the equivalent application is supposed to have [15]. Compared to the traditional monolithic architecture (Figure 1), which usually has its functionalities separated into functions and/or classes, the MA has its business logic in separately runnable microservices. These microservices contain only a minimal set of classes and functions designed to complete a single step in a larger chain of events that is needed to complete a task [15].

The benefits of MA are usually associated with the implementation and deployment stages of software development as the aspect of separation makes it easier to upgrade or replace a microservice than to refactor a monolithic application. The authors [15] also mention the possibility of using different languages and technologies for each microservice, allowing development teams to use their own preferences for each functionality they create. However, there is an added complexity to MA on the account of the necessary communications that often need to handle a large volume of traffic and numerous simultaneous processes happening in the background.

2.2 Monitoring

Monitoring is a practice in software maintenance and operations that provides immediate feedback about the state and quality of the monitored component. In "The Art of Monitoring" [20] James Turnbull states that the primary goal of monitoring is to translate metrics about the monitored application into useful information that contains valuable knowledge to the business that allows being reacted upon - for example, to further invest into the product in order to improve its usability or fix detected bugs and issues.

2.3 Observability

Observability is a unit of measure that describes how well a state of the system can be determined by its external outputs [19]. Observability is often associated with the qualitative features that a modern monitoring solution should possess. Complete observability is achieved by combining the three forms of data that are collected for monitoring applications: metrics, logs and traces [19]. All of these have a purpose that enable the observer to understand the state of the system better. Metrics, as the name suggests, allow to measure the quality of the system by defined numerical data points. It is up to the observer to define the meaning and boundaries behind each metric to make sense of them, but in most cases the monitoring tools available help to track them constantly when the necessary setup is configured. Logs on the other hand provide insight to the code level logic in a level of detail that metrics cannot give. A log is a text-based record of a past event that is completed with a timestamp that indicates when it occurred and a payload that offers context about what exactly happened [7]. The readability and usefulness of logs depend heavily on how logging is implemented in the code of the software. A trace is a relatively new type of monitoring data introduced mainly for monitoring microservices and other kinds of distributed systems. Traces are representations about a complete chain of transactions from a single point of beginning. These traces can encapsulate actions in multiple microservices, databases and other software components and hold useful temporal metadata like timestamps and duration that allow them to be associated with metrics and logs that were collected at the same time as the trace itself [7]

Observability tools in this thesis are defined as monitoring tools that combine these forms of monitoring data into a single platform and correlate it for the user to provide a better observability of the monitored system.

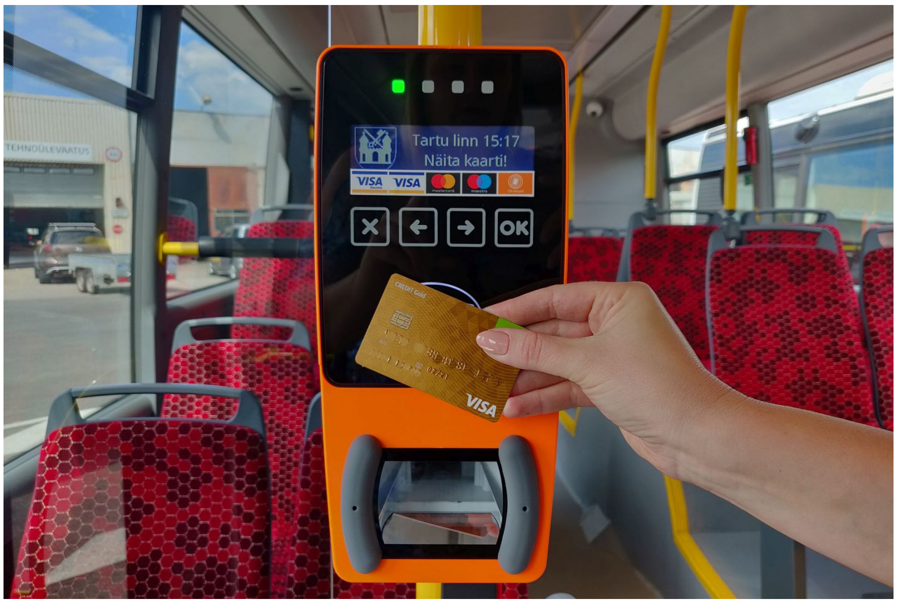

Figure 2. Ridango AFC

2.4 About Ridango

Ridango[2] is an Estonian company founded in 2009. Its primary focus is to provide public transportation management solutions such as automated fare collection and realtime passenger information systems [1]. The company's primary clients are municipalities and cities, but also private transportation providers around Europe. Ridango's biggest clients are currently the cities of Kiev and Tallinn which have a combined total of 3.2 million habitants that can all access the public transportation that Ridango's services are catering to.

The technology stack of Ridango has changed over time, but the general architecture has mostly remained the same. The organization uses Android devices on the public transport vehicles connected to back-end services over a standard API, which provides the necessary business functionalities and persisting the data in PostgreSQL databases. Ridango also hosts its own websites for back-office management and customer portals for online ticket sales. Historically these customer portals have been implemented as a monolithic PHP Laravel application, but in recent years the web team has refactored them into more lightweight and separate Angular websites. Additionally, in order to scale for bigger clients, technologies like Kubernetes, Redis, and numerous functionalities from the Amazon Web Services (AWS) platform have been included to Ridango's technological toolkit.

[2]https://ridango.com/

2.5 Microservices and monitoring setup in Ridango

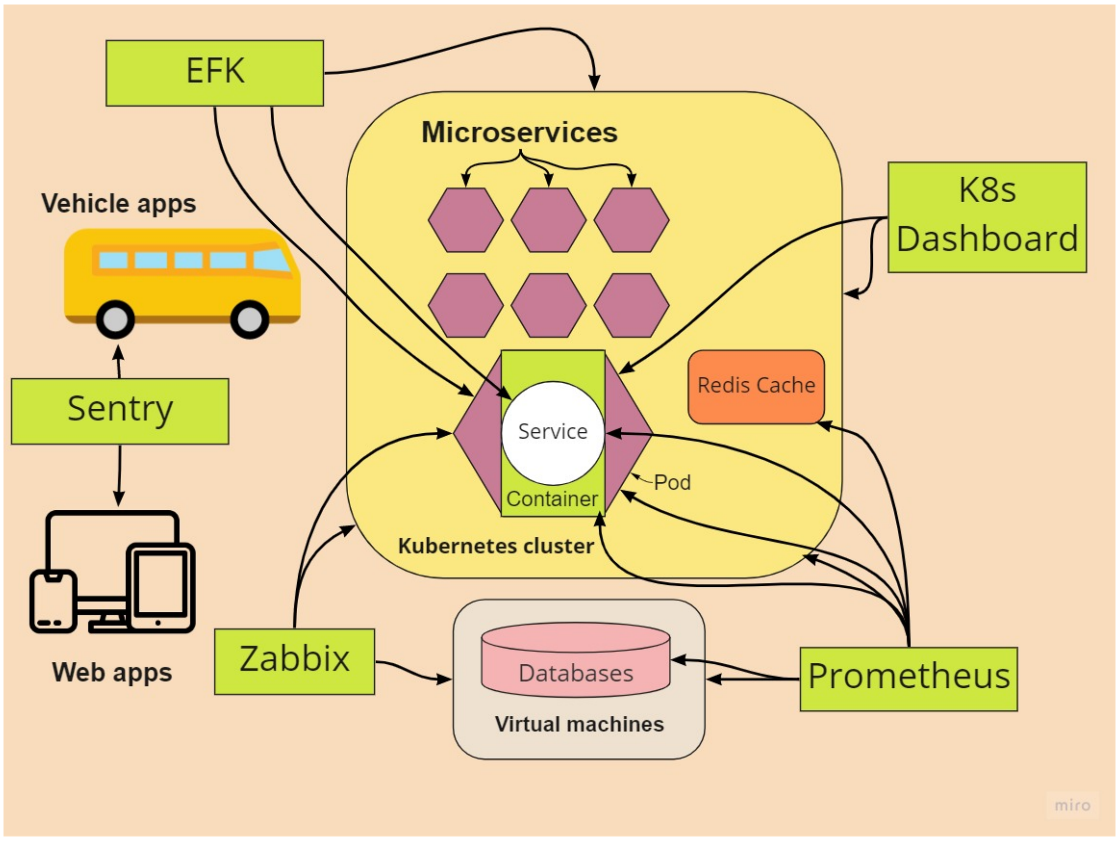

Figure 3. Monitoring solution of Ridango

Since 2019 with the adoption of self-hosted Kubernetes, Ridango has shifted considerably towards utilizing the microservice architecture. This has helped the company to reduce the time and complexity of releasing new versions of software, but more importantly, to make its services available without any downtimes and operational delays. This is a crucial benefit for the domain of public transport since most of these organizations have their vehicles operating for most of the 24 hours. However, there are still a few central applications in Ridango that can be considered monolithic in nature. The most significant for Ridango are a few Java Spring applications that essentially act as internal APIs between primary databases and all the other components that need access to them. These projects have over 400 000 lines of code and are responsible for the most common services a modern ticketing solution provides: ticket validation, sales, providing and storing data about user activities and transactions etc. The mentioned applications are progressively decoupled as the development continues, but the time to reach a complete microservice setup for Ridango can still take some time.

In terms of monitoring, Ridango has never had any strict policies associated with it. Each development team is responsible for a set of services or projects, including their sufficient monitoring. This means that a developement team decides what it deems to be valuable enough to monitor and what should be done to achieve a sufficient monitoring setup for the team. The only aspect of monitoring that is decided at the management level is the set of tools that are available for use: the reason is that usually, these tools have some kind of cost of maintenance and implementation which needs to be approved.

Currently, the development teams have the following tools available:

2.5.1 Zabbix (Zab)

Zabbix is an open-source monitoring tool for collecting and visualizing metrics from servers, networks, virtual machines, and cloud services [8]. In Ridango, it is used to observe the state of on-premise servers, virtual machines, and Kubernetes nodes.

2.5.2 Prometheus and Grafana (PG)

Prometheus is a widely used metrics collector software known for its multi-dimensional data model and pulls model of the agents [6]. It has very flexible and powerful built-in API-s that allow it to be integrated with many other technologies. Ridango has coupled it with a popular visualization and dashboarding tool Grafana. This setup is hosted in the firm's AWS cloud and monitors our entire server stack - including infrastructure, virtual machines, Kubernetes, and all the microservices hosted on them. Development teams have implemented their own automatic alerting system and custom metrics that are specific to Ridango's services.

2.5.3 Elasticsearch, FluentD and Kibana (EFK)

Elasticsearch is essentially a search engine and analytics software that allows the allocation of the necessary information with a minimal delay for text-based search [3]. FluentD is an log collection agent software that is used in Ridango to parse logs and forward them to the Elasticsearch cluster. Kibana is a data visualization UI tool that is mostly used to create an environment where Elasticsearch can be accessed by users [4]. The EFK is a combination of the three tools that together provide a complete and open-source version for a modern search engine tool. As is the case in Ridango, it is often used as logs' storage and querying solution. At the time of writing this thesis, all of the logs of the server stack(defined under Prometheus and Grafana) can be queried through this interface.

2.5.4 Sentry

Sentry is a SaaS (Software as a Service) solution for automated runtime crash and fault detection for various types of software applications [14]. At the time of writing this thesis, Sentry is used in Ridango by the web and device development teams for monitoring erroneous behavior in websites and android applications. We decided not to include Sentry in the scope of the user requirements analysis and the monitoring tools comparison of this study as it is not directly related to monitoring the server-side microservices of the firm.

2.5.5 Kubernetes and Kubernetes Dashboard

Kubernetes is an open-source container orchestration platform. Its primary purpose is to manage the deployment of multiple containers that are needed to provide a service without downtimes and provide a dynamically adapting environment that can handle hundreds of containers simultaneously [12]. Kubernetes dashboard (K8sD) is a web UI for managing, monitoring and troubleshooting the components in the Kubernetes cluster. It enables the user to view simple metrics, execute commands, and view configuration files and logs of the applications running in the Kubernetes environment [11].

2.6 Other related technologies

Docker is an open-source containerization technology that enables to isolate applications from other processes on the same host with a lightweight virtualization mechanism that uses the kernel of the host's OS [19]. These containers are used to automate and simplify the process of software deployment.

Ansible is an automation tool used to create automatic pipelines for application deployment, manage configurations, and orchestration between services. It uses a simple format in YAML files called Playbooks to define a step-by-step task execution process [5].

3 User requirements analysis

The decision to conduct a user requirements analysis (URA) in Ridango as a part of the research was driven by the lack of awareness about the needs of all the internal stakeholders and the state of the current monitoring solution. Additionally, there was an apparent necessity to identify the stakeholders and roles regarding monitoring to gather their feedback as input in the monitoring tool comparison stage of this thesis (Section 4). The contents of this section describe the methodology and the results of the conducted URA.

3.1 Methodology

The user requirements analysis was conducted following the methodology described by Martin Maguire and Nigel Bevan in "User Requirements Analysis: A Review of Supporting Methods" [16]. The mentioned approach consists of 4 phases:

1. Information gathering
2. User needs identification
3. Envisioning and evaluation
4. Requirements specification

Phases 3 and 4 are combined in this study as phase 3 was more of an aid (that was undocumented) in analyzing the gathered data to list the final requirements in the final stage of the study.

3.1.1 Information gathering

Stakeholder identification
Using the described methodology, the preliminary step before carrying out any observation or inquiry is to identify the "users" addressed in the URA, in this case, the stakeholders of the internal software monitoring solution. There were multiple possible options and categorization patterns for defining the stakeholders. The two approaches that we considered the most were: team-based division and role-based division. The idea behind the team-based approach was that stakeholders would be derived from each team in Ridango, for example, "Member of Realtime team". This solution would address the situation that each team uses monitoring tools differently and has its own configuration for them. This logic would also include positions like product owners and project managers since they belong to their respective teams as well. The negative aspect of this approach would be that in the development teams, there are many roles such as developer,

QA engineer, engineering manager, etc. that can have completely different requirements for the monitoring tool that they use. These requirements might get mixed or ignored in the data analysis of the study. To address this issue the second categorization option was chosen: the role-based approach. This method defines the stakeholders by their role or position within Ridango. An example would be “DevOps Engineer” or “Software Developer”. As in many cases, this approach for some internal roles would be too abstract (e.g. Developer) and it would ignore the team-based differences that were priorly discussed, we decided to split the more general roles by the example of the team-based approach. The initial “Developer” role, for instance, was divided to “Backend developer”, “Web developer” and “Device developer” to understand the differences between each of them in the later steps of this research. Additionally, there was the question about the number of stakeholders that would be reasonable to define, as there were many minor stakeholders who would be affected only in the slightest margins. Following an internal debate, the decision was taken to focus more on technical roles for whom the monitoring tools are integral in their regular workflows.

Task Analysis
Task analysis is also an important part of this research as it enables one to understand the system’s existing state and the issues that are met with each task, which often reveal opportunities to address the user’s needs[1]. The task analysis was performed with the information gathered about monitoring use cases during the initial stakeholder interviews.

3.1.2 User needs identification

The user needs identification stage was performed in the format of interviews with the users/stakeholders defined in the previous phase.The stakeholder interviews were all performed in video calls in which all members of a stakeholder group who volunteered to participate in the survey were present. There were at least 5 participants present for each stakeholder group, which should be a sufficient sample size considering there are about 10-15 employees in each engineering team. The participants were all asked to give an answer to each question provided below. After the initial interviews, participants were also asked to watch introductory videos for Datadog, New Relic, and Instana and answer three additional questions about the observability tools in written form.

The questions provided in the interviews:

1. Name and describe the primary situations that require monitoring in your work.
2. What tools are you mostly using at the moment for monitoring? What functionalities do they serve?
3. What are the negatives with the monitoring tools you are currently using?

4. What are the positive aspects and useful features of the monitoring tools you are currently using?

5. What are the requirements that a complete monitoring solution should have to support your use cases of monitoring?

   After introducing and familiarizing stakeholders with Observability tools:

6. What is the most important/useful feature for you in the presented tool?

7. In your estimation, how much of your time would this tool save you per-incident or bug-fix task?

8. By the initial impression, which tool seems to be the most potent to you?

3.1.3 Evaluation and requirements specification

The requirements were formulated based on the information gathered from the stakeholder interviews and further consulting with the technical management board of the company (to specify the organizational requirements as well). The requirements were categorized as follows:

- User requirements - functional requirements that allow users to perform necessary tasks
- Usability requirements - non-functional requirements that contribute towards the effectiveness of using the software and the satisfaction of the users who are operating with it.
- Organizational requirements - general requirements that support the policies and structure of the organization.

3.2 Summary of the findings

This subsection contains the summary and conclusions of the internal stakeholder analysis of monitoring in Ridango. In the original data analysis document, the results were grouped by each stakeholder but then converted to a more concise format for the thesis.

3.2.1 Defined stakeholders

The involved stakeholders were defined as follows:

- DevOps Engineer/System administrator
- Backend Developer

- Web Developer
- Device Developer
- Support Engineer.

3.2.2 Task analysis

The following workflows and usage scenarios of the defined users were gathered as result of the task analysis phase:

Error detection and investigation

Related stakeholders: Support engineer, Sysadmin/DevOps engineer

This scenario describes the situation when an error has been reported automatically by a configured alert of a monitoring tool or by any member of the technical staff.

The usual process:

1. An issue is spotted with an automatic alert or manual observation by support staff or other employees.
2. Affected clients are notified that there is currently an issue (if needed).
3. Top-down mitigation process is conducted: investigator tries to understand which functionalities and clients are affected and narrow down the issue to a more specific cause.
4. A task is created with as much useful information about the problem as possible and assigned to a developer of the responsible team.

Monitoring is needed to detect the error or fault with the automatic alerts or manual observation. Also, the top-down mitigation process requires more detailed monitoring data, usually the investigation of recorded logs.

Report from a client

Related stakeholder(s): Support engineer

Currently, most of the incidents and errors from the live environment are reported by the clients of Ridango. The clients are encouraged to create an issue ticket in the firm's support website or contact them directly via email or phone, communicating with the support engineers. The support engineers generally try to investigate the issue as thoroughly as possible to reduce the time it takes for the developer to apply a fix to the given issue.

The usual process:

1. An issue is reported by a client with some details of how the error happened and what is the nature of it. Usually, the level of detail is not very high.
2. Similar top-down mitigation process as with "Error detection".
3. Other affected clients are notified.
4. A task is created with as much useful information about the problem as possible and assigned to a developer of the responsible team.

In this scenario, most of the details about the error are provided by the client, but for a more comprehensive and technical overview monitoring tools are still needed as an aid and also for the error mitigation process.

Bug or Incident Management task

Related stakeholders: Backend developer, Web developer, Device developer

According to the interviews and the time usage statistics of Ridango's technical staff, this use case is by far the most commonly occurring and time-consuming process that depends on monitoring tools.

The usual process:

1. The developer tries to understand the problem described in the received task description, asks for further details from the support staff or product owner if needed
2. The developer tries to reproduce the issue (if possible).
3. The developer tries to narrow down the process to a root cause.
4. The developer tries to find and apply a fix to the root cause.

For this scenario, monitoring is required for step 3 in locating the root cause. This requires usually either the ability to replicate the problem or some detailed monitoring data such as error stack traces and other retained logs.

In the case of web developers and device developers, these tasks are often only related to the client-side of Ridango's services, but since they depend on many server-side microservices in fetching and storing data, for example, clear visibility into the backend services is also required.

Software deployment

Related stakeholders: Backend developer, Web developer, Device developer

Deploying new versions of software is the responsibility of developers in Ridango. Especially considering microservices, this process has become much faster and easier thanks to the heavy emphasis that has been put into developing automated tests and pipelines. Additionally, the rolling update of Kubernetes removes the pod running the previous version of the service only when the new service has passed all the health checks and does not crash on start.

The usual process:

1. After testing the new feature and integration tests a new release task is created.
2. The developer who is assigned with releasing the feature initiates the continuous deployment pipeline for the live environment.
3. The developer observes and ensures the completion of the pipeline.
4. Developer monitors the if the new service is operational and assures that there are no immediate problems with it.

   Monitoring is needed for steps 3 and 4 in order to observe the completion of the pipeline and that the newly deployed service is operational.

Infrastructure changes and deployments

Related stakeholders: DevOps Engineer/System administrator

Infrastructure management and updates are the responsibility of system administrators and DevOps Engineers in the company. The changes and deployments on this level are usually very delicate and have to be conducted during the least active times for the clients. Since there is the possibility to test most of the changes in test environments, the risk is minimized, but caution and close observation are required. Monitoring tools help to observe the state of infrastructure, virtualization environments, and processes to ensure that there are no issues or side effects with the deployment.

3.2.3 Usage of the existing tools

The data gathered from the stakeholder interviews suggested that since there are many different scenarios that require monitoring (also listed in the previous paragraph), the monitoring tools that are used often fulfill different monitoring needs and in many situations are not used cohesively at all.

The ELK setup (Elasticsearch and Kibana) is the most used monitoring solution across all the stakeholders and is mostly used in error mitigation scenarios in incident and bug tasks (see "Bug or Incident Management task" page X) where logs provide the most detailed information about the occurred incident. However, it is also worth noting that most of the stakeholders also agree that ELK has a relatively high learning curve and requires time to get used to it. Members of the web team have chosen to use command-line tools such as grep instead for this reason.

Prometheus and Grafana is the second most frequently used monitoring tool. It is often used to find the exact time of a disruption or a crash regarding a microservice. Some of the interviewees mentioned checking the metrics about the underlying virtualization components and hardware if there is the suspicion that an infrastructure-related issue causes the error: CPU and memory overloads. Its primary contribution to the monitoring ecosystem in Ridango seems to be more related to its automatic alerts feature that is used to direct notifications to the support and engineering team's communication channels to allow them to react to any new incident related to the monitored services and environment.

According to the URA, **Zabbix** is only used regularly by the Sysadmin/DevOps and Support engineer stakeholders. Zabbix is used to collect and visualize performance metrics related to server hardware, databases, virtual machines, and Kubernetes nodes. Almost all of the functionality that Zabbix provides is covered by the Prometheus and Grafana setup, but Zabbix is still used as a reliable alternative.

The Kubernetes dashboard is considered a centric tool for any stakeholder who is regularly involved in deploying microservices in the Ridango's self-hosted Kubernetes clusters in test and development environments. It is primarily used for monitoring and troubleshooting deployments.

3.2.4 Defined requirements

The requirements that were derived from the stakeholder interviews are listed in Table 1.

The defined requirements were later correlated with the features of the monitoring tools that are assessed in the feature comparison in Section 3. This allowed us to calculate a quality rating and coverage percentage of these requirements for each compared monitoring tool. The full references to the feature identifiers in the "Correlated features" column are listed in Table 2 and Table 3.

3.2.5 Potential for saving developer time

After the initial stakeholder interviews, participants were asked to watch introductory videos about some of the observability tools discussed in this thesis and provide their approximation on the amount of time these new tools could potentially save in their daily work. The results of this inquiry were promising. For developers, even the most pessimistic answers remained over 10% and some of the participants predicted that these tools could save even over 35%. System administrators predicted their time saving on infra issues would be around 8%. An interesting discovery in the study was that support engineers predicted that their time consumption per incident would actually most likely increase, as they would be able to analyze each error in a more detailed manner, often finding the root cause by themselves thanks to the capabilities of an observability tool.

3.2.6 Missing practices of monitoring

In the course of the interviews, it became clear that the stakeholders from the development teams did not know what to expect and value in terms of the features that are not represented in the currently adopted monitoring tools in the company. Thus, in the following testing phase of the observability tools, more emphasis was also put on investigating their viability in the context of Ridango's products and current practices of monitoring.

These are the missing practices of monitoring that were identified in the course of the study (provided with a brief explanation):

Version comparison and analytics
A monitoring tool feature that allows analyzing different versions of the same software on multiple levels, such as general performance metrics, types of errors detected, response times, thread count, etc.

Performance analysis of database queries
A feature that records and provides statistics about the specific database queries of a microservice. This is used to optimize the speeds of slower queries to improve the usability of the products.

Table 1. Monitoring requirements of Ridango

ID	**Requirement**	**Correlated features**
	User requirements	
R1	Supports the process of Error mitigation - Time to find the cause of an error is minimal	G6,G8,G9
R2	Supports the process of Performance issue mitigation - time to find the cause of performance issue is minimal	G6,G9
R3	Supports the process of Error detection (automatic and manual) - time to detect error is minimal	G7, F34, F64
R4	Supports the process of Performance issue detection - time to detect performance issue is minimal	G7,G9
R5	Supports the process of software deployment	F3,F6,F1, G7, G8
R6	Supports the process of infrastructure deployment	F61, F64 ,G7, G8
R7	Has tools to perform Version comparison	F36
R8	Enables the investigation of transactions between services	F37
R9	Provides necessary metrics for each service and infrastructure that are available in the current monitoring stack	F38, F59
R10	Has mapping tools to visualize the whole system and state of each component	F40,F60
R11	Allows to define automatic alerts that notify the users	F43
R12	Allows to configure the alerts based on logs and metrics	F43,F49,F54
R13	Allows to calibrate the alerts to the correct treshold in order to guarantee their usability	F44,F46
R14	Has he ability to view the state of the infrastructure at the time of any metric, trace or log was gathered	F67
R15	Alerts can be sent to the support channels in Microsoft Teams or Slack	F48
R16	Logs can be viewed and queried using the tool	F53
R17	Logs of different formats are parsed correctly in the tool	F58
R18	Logs can be viewed togheter with relevant traces and metrics	F51
R19	Unuseful logs can be filtered out of the system	F52
R20	The tool supports monitoring of Kubernetes and its underlying processes, pods, nodes etc.	F59, F63, F65, F66
R21	The tool support monitoring of Kafka and Zookeeper nodes	F23
R22	The tool provides specific metrics for the language/technology used in the firm's services and databases	F17,F18,F20, G4
R23	It is possible to create graphs and dashboards with the present metrics in the tool	F42
R24	The tool supports monitoring of CI/CD tools existing in the company	F24,F25.F26
R25	The tool provides automatic error detection and problem highlights	F34,F35, F45, F61, F64
	Usability requirements	
R26	The UI is intuitive and easy to use	F1
R27	The learning curve of the tool is minimal	F2
R28	The tool is fast for the user, load times for the 95th percentile should be below 1.5 seconds.	F3
R29	It is possible to share the pages and graphs of the tool with other users.	F4
R30	There are extensive documentation and guides available about the tool	F5
R31	There is a tech support service available.	F6
R32	The general depth of customization is high	F7
R33	Time to add new monitored component is minimal	F10, F11
R34	Time to configure a feature in the monitoring tool on the monitored components is minimal	F12
R35	Amount of code needed to be added in each project is minimal	F13
R36	The resource usage of the agent is minimal	G3
R37	The monitoring tool is easily scalable	G3
R38	The time to update and stop monitoring tool agents is minimal	G13
R39	Time to configure alerts is minimal	F44
R40	Alerts time to reach the support team is minimal	F48
R41	The amount of false positives of the alerts is minimal	F43,F45,F15,F8
R42	The logs are available in the system for at least 14 days	F50
R43	The data is provided in real time about the monitored components	F8
	Organisational requirements	
R44	An on-premises option excists for the tool	F33
R45	User rights management feature excists for the tool	F27,F28
R46	User's actions can be logged	F30
R47	Accounts can be made automatically with the Azure SSO	M31
R48	Sensitive information for the company can be filtered out before displaying in the monitoring tool	M32
R49	The ROI of the tool is greater for the company than its total maintenance cost.	Decided in ROI calculation

Anomaly detection with AI models

This feature is present in relatively few monitoring tools that are available today. It requires a capable AI model with high accuracy to be reliable enough for detecting anomalies, which are essentially mismatches in the general pattern of incoming metrics in a certain period. These mismatches state that the service performs differently from its usual routine, which suggests that there might be a faulty behavior occurring with the given application.

4 Comparison of monitoring tools

One of the primary contributions of this thesis and the research for Ridango was to find a way to improve the firm's monitoring setup to address the particularities of the MA. There was a clear expectation in the technical teams that an APM or Observability tool would be adopted. Primarily, three observability tools were tested and compared in this comparison: Datadog, New Relic, and IBM Instana. These tools were chosen to be used in this comparison based on preliminary research of provided features, pricing models, supported technologies and ratings form already published reviews [9] [10]. The initial plan was to also test Splunk Observability and Dynatrace observability tools in this comparison, but we had to exclude them due to the time constraints of this research. Since the primary objective was to conduct a comprehensive research that would allow the board of Ridango to make an informed decision, the existing tools were also represented in the comparison to identify if the newer observability tools were indeed more capable alternatives. It is important to note that the existing monitoring tools were represented in the comparison as they were configured already by Ridango. Some of these existing tools have features that are marked as non-existing in the comparison, while they might still exist as an additional feature for these technologies. This is because these features were not implemented in the existing setup of the particular monitoring tool. Consequently, this study does not qualify as a full comparison of these 7 technologies but rather as a comparison of the 3 mentioned observability tools in the context of the existing monitoring setup in Ridango.

4.1 Methodology

The general methodology that was used in comparing the monitoring tools was a generic software evaluation method which compares the features of the software against the requirements that were posed to it, in this case, the requirements defined in Table 1 in Section 3. To simplify the process of grading the defined requirements for each tool, we decided to list and compare the features of the monitoring tools and later map them to the requirements that they are fulfilling. It is also important to note that most of these requirements defined were not easily testable with a definitive result. Thus, many objectively unquantifiable results are based on our opinionated justifications.

The process of the comparison can be described as follows:

1. All the features and subfeatures of the tools were graded with a quality rating of 0-5 which were then justified with a few sentences.

2. Each grade was then multiplied by the total coefficient of the importance rating from the stakeholders in the conducted URA to form a final grade for each requirement per each tool.

3. For each solution, the final grades were then added together to calculate a total rating of each tool.
4. Finally, the features were correlated with the requirements defined in section 3, to measure the coverage of the requirements for each tool. The final ratings for the fulfillment of each requirement per tool were derived by calculating the sum of scores (calculated in step 2) of the correlated features for each requirement.

The architectures of these solutions were also compared as a preliminary step of this research to better understand the limitations, risks, and technical requirements that these solutions bring to Ridango's server environments if we test or implement them.

4.2 Architecture

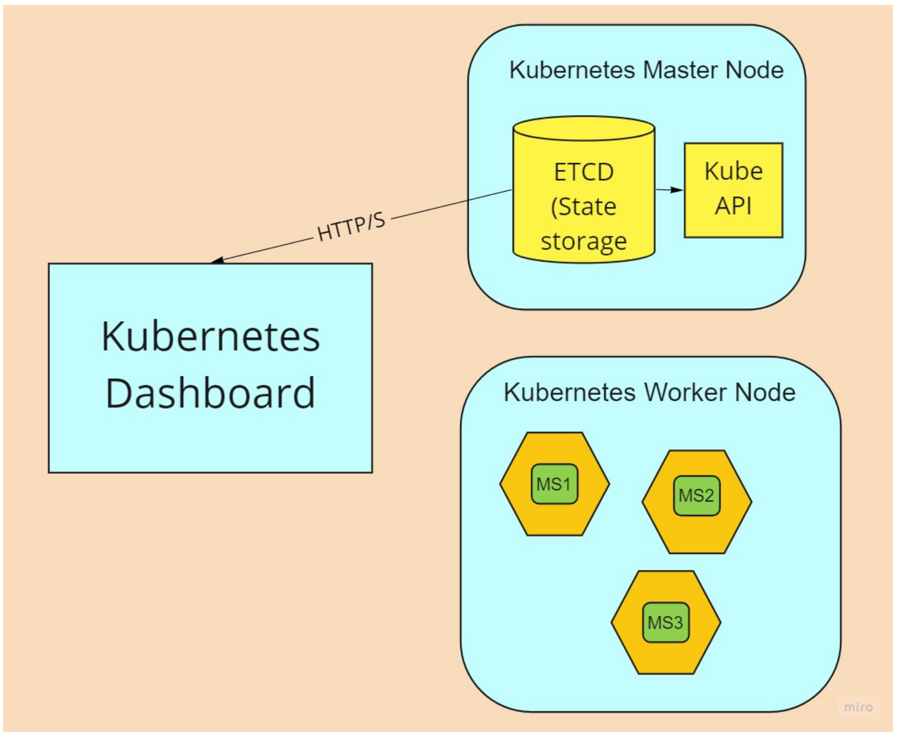

Figure 4. Kubernetes Dashboard

Kubernetes Dashboard
Kubernetes dashboard (Figure 4) is a web-based UI that is possible to setup with the Kubernetes engine itself [11]. It can be deployed with the kubectl command-line tool on the same host as kubernetes. After the installation and setting up a proxy for the tool, its

back-end starts to request and visualize data from the Kubernetes API server that access the ETCD state storage databases in the master nodes. The ETCD itself is updated within the Kubernetes environment with the all the metrics and logs that are sent by the nodes, processes and pods in the Kubernetes cluster. Since the ETCD holds only the state of the system and is a very lightweight storage, historical data is not persisted for a longer than a few hours.

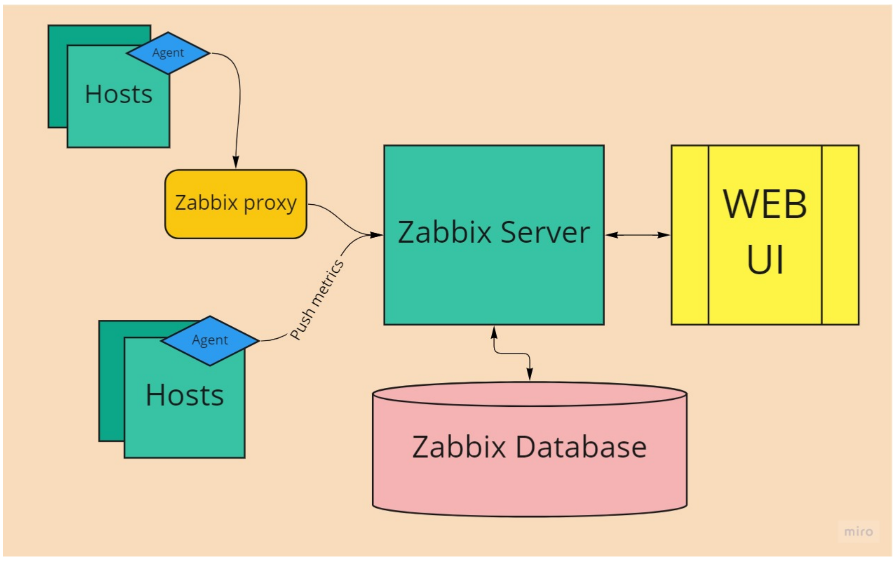

Figure 5. Zabbix

Zabbix

Zabbix uses a classical agent-based solution that supports both the pulling and pushing mechanism for transmitting data to the Zabbix server (Figure 5). The agent uses the pulling mechanism in a passive mode, which sends the data only when the server requests it. The active mode, which takes advantage of the pushing mechanism, sends data constantly within a specific interval. Additionally, data can be sent with the help of a proxy, which should improve the stability and performance of the monitoring solution. The lightweight Zabbix agent, written in the C language, can be deployed to various host operating systems and even application environments to collect metrics. The Zabbix server then parses the metrics for the Web UI, persists the data in a database, and allows users to access and visualize the data with the help of the Web UI.

Prometheus and Grafana

Prometheus uses a simple pull-based architecture to collect metrics from all kinds of hosts, services, and short-lived processes (Figure 6). Prometheus works especially well

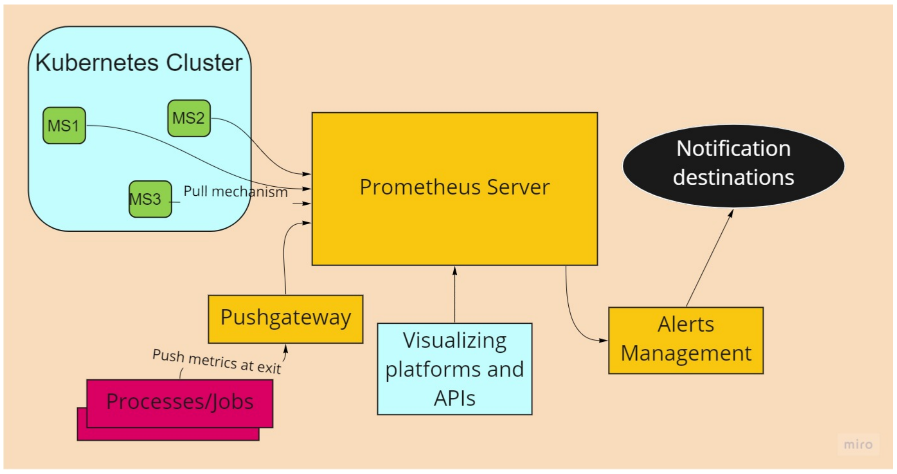

Figure 6. Prometheus & Grafana

with Kubernetes since Kubernetes API and Kube-stat-metrics (KSM) service require minimal configuration to implement it. The pull-based architecture of Prometheus allows autodiscovery of all the microservices that are inside Kubernetes pods without requiring any additional software agents. Prometheus server scrapes metrics from the pods that have an endpoint and port available for requesting the data, which are both specified in the Prometheus configuration file. The metrics from the services and Kubernetes metrics from KSM and Kubernetes API are all pulled to the hosted Prometheus backend server which processes the data and sends it to visualization platforms like Grafana. The "Alerting manager" inside the Prometheus allows sending notifications about the alerts that are defined in the Prometheus server. These alerts can be configured using Grafana and other Web UIs.

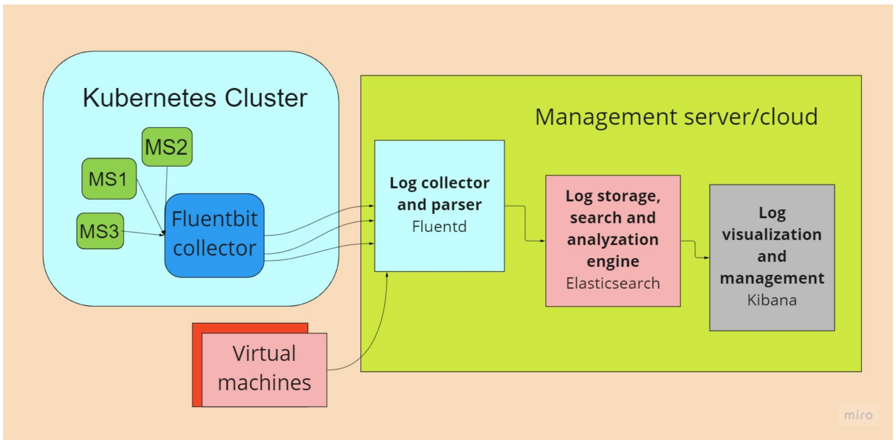

Figure 7. EFK

EFK

The Elasticsearch & Kibana solution for logging allows different technologies to be used for log collection. The two most frequently used technologies for doing this in Kubernetes are Logstash and the Fluentd. On Figure 7, an architecture diagram with the Fluentd technology is shown, as it is the logs collector that is currently implemented in Ridango's Kubernetes clusters. In this scenario, logs are collected by a locally deployed Fluentbit service which is a lighter version of the Fluentd collector. The Fluentbit collector sends the collected logs to the Fluentd service in the management server, further aggregating the logs and adding tags to enable better quering when using Elasticsearch. After the data is in a readable and searchable format, the logs are then stored in the Elasticsearch engine in a manner that they can be fetched and queried within a minimal time. The stored logs can be now accessed by using the Kibana UI, that allows to query and define metrics or alerts based on the logs.

New Relic (NR)

The New Relic data collection architecture (Figure 8) is based on 2 types of components. First is the host agent that collects metrics, logs, and traces from the infrastructure hosts. The second component type is the APM agent that is installed on microservices or the containers which are running them. These APM agents send the data directly to the New Relic cloud without communicating with the Infrastructure agent on the same host. Both types of agents use push-based logic. Notably, the log collection from the microservices is done separately by the infrastructure agent which collects the logs from container standard outputs or log files. There are many versions of infrastructure agents and APM agents that support different technologies and language environments. The New Relic platform can also collect data from other metric, log and trace collecting technologies which can change the logic of how the data is retrieved. For Kubernetes, New

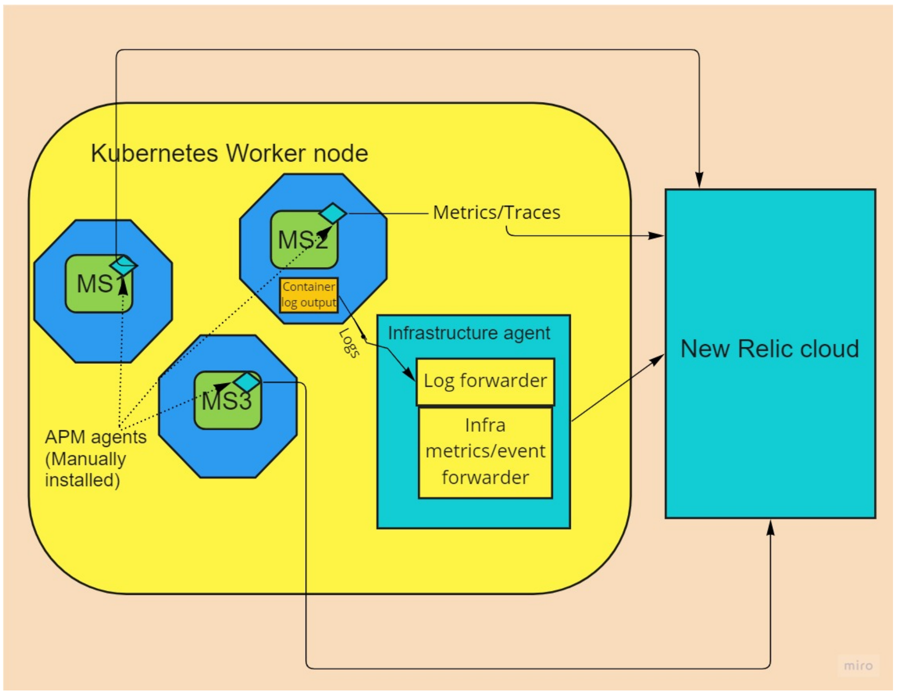

Figure 8. New Relic - Architecture

Relic also recommends using a technology called Pixie that can instrument a Kubernetes cluster with all its hosted microservices via a similar autodiscovery pattern described for IBM Instana. Unfortunately this solution exceeded the resource usage limitations of firm's locally deployed Kuberenetes clusters and we were unable to use it while testing the tool itself.

Instana (Ins)

Instana's monitoring data collection architecture (Figure 9) is one of the biggest strengths of this SaaS monitoring solution, because it allows an extremely fast deployment without a significant overhead in terms of resource requirements. Instana uses a single agent per host, which deploys micro-agents called sensors that locate and collect data from all the services, applications, databases, and processes running on the host. The discovery of these components is done continuously in the host agent. This works well with the MA and Kubernetes environments where instances of the microservices are constantly deployed and shut down. Unlike the New Relic APM agents, Instana's sensors send their data to the host agent, performing compression and relaying to the Instana cloud backend.

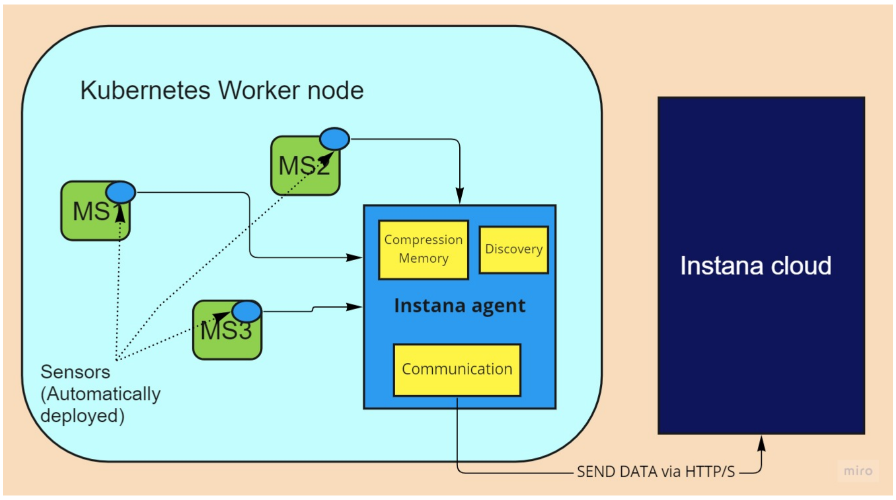

Figure 9. IBM Instana - Architecture

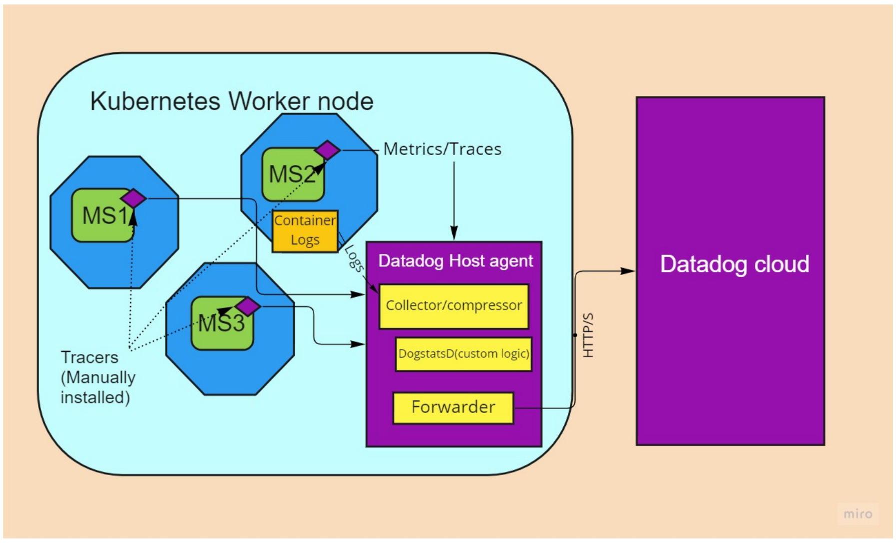

Figure 10. Datadog - Architecture

Datadog (DD)

The telemetry data collection architecture of the Datadog platform (Figure 10) is almost identical to the New Relic's solution. The only notable difference is that Datadog APM agents (that are called tracers) send their data to the host agent which then compresses it to a smaller size and sends it over to the Datadog cloud. The APM agents and the host agents need to be manually installed. The host agent uses a YAML file for configuration, where it is possible to enable or disable features of the monitoring solution for the specific host. Datadog has also many supported integrations with technologies that collect monitoring data. However, none of them currently offer automatic discovery and implementation of microservices.

Table 2. Feature comparison scores F1-F33

ID	Feature name/Category	Multip.	NR	Ins	DD	PG	EFK	K8sD	Zab
G1	General features and attributes	2.1	72.5	62.7	77.7	70.2	64.5	59.6	54.5
F1	Quality of the UI	2.7	10.8	10.8	13.5	10.8	13.5	10.8	8.1
F2	Learning curve	2.2	8.8	11	8.8	6.6	6.6	11	8.8
F3	Speed of the website	2.5	7.5	7.5	10	12.5	7.5	10	10
F4	Data sharing	1.7	8.5	6.8	6.8	8.5	5.1	3.4	5.1
F5	Quality of Documentation	2.3	11.5	4.6	11.5	11.5	11.5	9.2	6.9
F6	Quality of Tech support	1.7	5.1	5.1	5.1	0	0	0	5.1
F7	General depth of customization options	1.7	6.8	3.4	8.5	6.8	6.8	1.7	5.1
F8	Real time data flow	2.7	13.5	13.5	13.5	13.5	13.5	13.5	5.4
F9	Ability to create custom dashboards	2.3	11.5	9.2	11.5	9.2	9.2	0	6.9
G2	Configuration	1	14	19	15	18	18	20	14
F10	Time to add new tech	1	2	5	3	4	3	5	3
F11	Time to add already conf.ed tech	1	4	5	4	4	5	5	4
F12	Code needed to be added within the component repo	1	3	4	3	5	5	5	3
F13	Scalability of the configuration solution	1	5	5	5	5	5	5	4
G3	Performance and resource usage	2.6	21.2	31.4	23.7	34.5	31.6	39.5	39.5
F14	Agent CPU usage	2.7	5.4	10.8	8.1	13.5	10.8	13.5	13.5
F15	Memory usage	2.7	10.8	8.1	8.1	13.5	10.8	13.5	13.5
F16	Amount of pods needed	2.5	5	12.5	7.5	7.5	10	12.5	12.5
G4	Integrations	2.56	100.5	106.5	115	106.5	90.5	0	106.5
F17	Integrates with Java Spring boot	2.8	14	14	14	14	14	0	14
F18	Integrates with PHP - fpm, phalcon, laravel	2.8	14	14	14	14	14	0	14
F19	Nginx	2.2	11	11	11	11	11	0	11
F20	NodeJS	2.2	11	11	11	11	11	0	11
F21	PostgreSQL	2.8	14	14	14	14	14	0	14
F22	Redis	2.8	14	14	14	14	14	0	14
F23	Kafka & Zookeeper	2.5	12.5	12.5	12.5	12.5	12.5	0	12.5
F24	Bitbucket	1.7	0	0	8.5	0	0	0	0
F25	Ansible	2	10	10	10	10	0	0	10
F26	Jenkins	1.2	0	6	6	6	0	0	6
G5	Management	1	28	28	27	24	24	24	24
F27	Role based user rights management	1	5	5	4	3	3	3	3
F28	User management	1	5	5	5	3	3	3	3
F29	Billing controls	1	5	5	5	5	5	5	5
F30	Audit Log	1	5	5	5	5	5	5	5
F31	Single-Sign On	1	5	5	5	5	5	5	5
F32	Privacy configurations	1	3	3	3	3	3	3	3
F33	On prem available	1	0	5	0	5	5	5	5

4.3 Feature comparison

This subsection contains the results of the conducted feature comparison. The results are marked in Table 2, Table 3, 4 and visualized on Figure 11. The results are then summarized and explained by each defined feature category in the following paragraphs.

Table 3. Feature comparison scores F34-F67

ID	Feature name/Category	Multip.	NR	Ins	DD	PG	EFK	K8sD	Zab
G6	APM	2.4	84	61.8	90	19.2	0	18.9	0
F34	Automatic error detection	2.7	10.8	8.1	13.5	0	0	10.8	0
F35	Performance issue detection	2.7	10.8	8.1	13.5	0	0	0	0
F36	Version comparison	2.2	8.8	4.4	11	0	0	0	0
F37	Tracing	2.5	7.5	10	12.5	0	0	0	0
F38	Software metrics	2.3	11.5	4.6	9.2	9.2	0	0	0
F39	Release monitoring	2.7	10.8	8.1	13.5	0	0	8.1	0
F40	MA dependecy maps	1.7	6.8	8.5	6.8	0	0	0	0
F41	Retention time for metrics and traces	2	6	10	10	10	0	0	0
F42	Code analysis		0	0	0	0	0	0	0
G7	Alerting	2.3	73.6	66.6	78.1	25.2	26	0	35.1
F43	Ability to define alerts	3	12	12	15	12	9	0	12
F44	Synthetic monitoring	2	8	6	10	6	8	0	8
F45	Anomaly detection	2.2	8.8	8.8	11	0	0	0	0
F46	Calibrations of alerts	2.7	13.5	8.1	10.8	0	0	0	0
F47	Preconfigured alerts	2.2	8.8	11	8.8	0	0	0	0
F48	Channels supported (Slack, Teams, Mail)	1.8	9	7.2	9	7.2	9	0	9
F49	Alerts for endpoints	2.7	13.5	13.5	13.5	0	0	0	0
G8	Logs	2.06	72.5	53.5	83.3	0	65	2.3	0
F50	History/retention of logs	1.7	8.5	5.1	6.8	0	8.5	0	0
F51	In context with traces and metrics	2.3	11.5	11.5	11.5	0	0	0	0
F52	Log filtering based on rules	2.2	11	6.6	11	0	11	0	0
F53	Log display and quering	2.3	11.5	6.9	11.5	0	11.5	2.3	0
F54	Alerts and anomaly detection based on logs	1.8	0	0	9	0	9	0	0
F55	Log pattern recognition	1.2	6	2.4	6	0	0	0	0
F56	Metrics generation based on logs	1.5	6	0	7.5	0	0	0	0
F57	Log archiving and retention from archive	2	6	6	8	0	10	0	0
F58	Ability to parse logs in different formats used in Ridango	3	12	15	12	0	15	0	0
G9	Infrastructure	2.42	91	95	96.1	44.8	0	46.2	42.3
F59	Infrastructure and Kubernetes metrics	2.7	13.5	8.1	13.5	13.5	0	13.5	10.8
F60	Host maps	1.8	7.2	9	5.4	0	0	0	0
F61	Problem highlights/warnings	3	12	15	12	0	0	6	9
F62	Infrastructure problems correlation with software	2.3	6.9	11.5	9.2	0	0	0	0
F63	Kubernetes maps and dashboards	2.3	9.2	9.2	11.5	6.9	0	6.9	6.9
F64	Automatic health checks	2.8	14	14	14	11.2	0	8.4	8.4
F65	Container monitoring	2	10	10	10	6	0	6	0
F66	Process monitoring	1.8	9	9	9	7.2	0	5.4	7.2
F67	Ability to see infrastructure state in context of APM	2.3	9.2	9.2	11.5	0	0	0	0

Table 4. Total feature category scores

ID	**Feature name/Category**	**NR**	**Ins**	**DD**	**PG**	**EFK**	**K8sD**	**Zab**
G1	General features and attributes	72.5	62.7	77.7	70.2	64.5	59.6	54.5
G2	Configuration	14	19	15	18	18	20	14
G3	Performance and resource usage	21.2	31.4	23.7	34.5	31.6	39.5	39.5
G4	Integrations	100.5	106.5	115	106.5	90.5	0	106.5
G5	Management	28	28	27	24	24	24	24
G6	APM	84	61.8	90	19.2	0	18.9	0
G7	Alerting	73.6	66.6	78.1	25.2	26	0	35.1
G8	Logs	72.5	53.5	83.3	0	65	2.3	0
G9	Infrastructure	91	95	96.1	44.8	0	46.2	42.3
T	Total scores	557.3	524.5	605.9	342.4	319.6	210.5	315.9

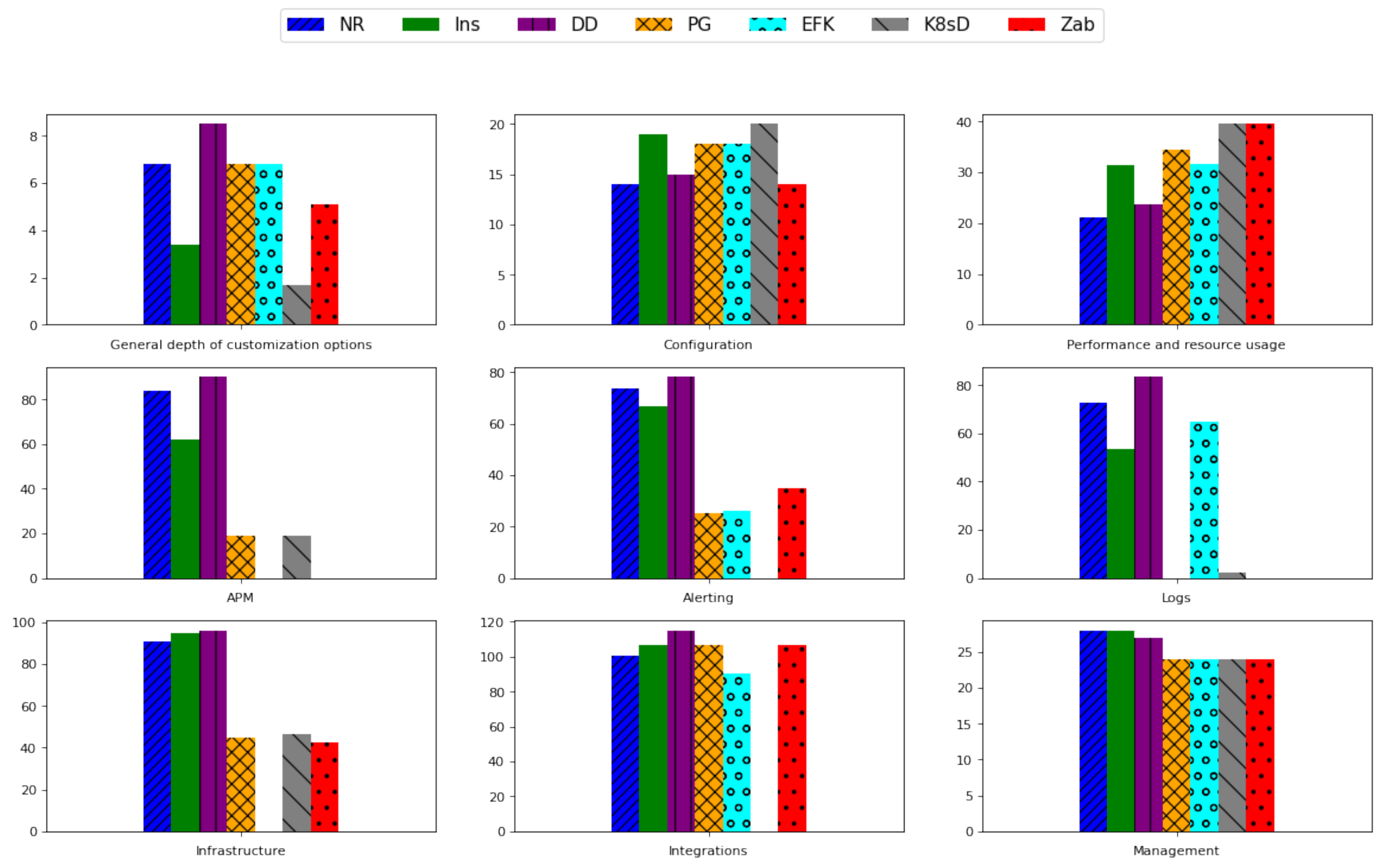

Figure 11. Comparison of feature categories

4.3.1 General features and attributes

This feature category is focused on the fundamental qualities and attributes that are represented in the monitoring tool. These are not associated with exact features and mostly cover some of the usability requirements.

New Relic and Datadog are the highest-rated tools in this category, justified by the quality of their UIs, good documentation, and customizability of every feature and page in their platforms. Datadog is only slightly higher rated because of the placement and layout of the pages which are more intuitive. The tools from the existing monitoring setup and IBM Instana are not too far behind in the score rating, as all of them have modernized UIs that allow a smooth user experience in performing their supported functionalities. In terms of the onboarding experience and the learning process, most of these tools are difficult to use at first: Elasticsearch and Prometheus, which depend on effective usage of their own defined query languages to find the necessary information. IBM Instana seems to have the most shallow learning curve, as it follows a consistent design pattern, offers explanations, and has an interactive tutorial. However, due to the questionable quality of the documentation and its smaller variety of configuration options in its features, Instana has a lower score in this category than Datadog and New Relic.

4.3.2 Application performance monitoring (APM)

This category tests the abilities of the monitoring tools to analyze the performance of the software in the context of time, version deployments, and other changes in the environment. In order to have a more accurate comparison for this category, we used 3 different mock scenarios to assess the effectiveness of the monitoring tools. Each of the scenarios contained a different performance issue or error that simultaneously affected multiple microservices.

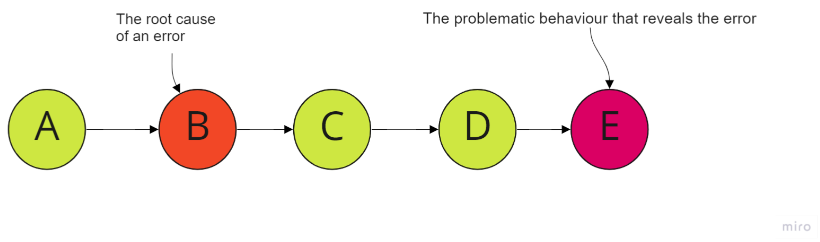
The root cause of an error
The problematic behaviour that reveals the error
A
B
C
D
E
miro

Figure 12. Erroneous behavior and root cause in microservice architecture

As depicted in figure 12, the idea was to create incidents where a fault in a backend microservice causes erroneous behavior in the frontend applications. These experiments allowed us to compare tracing, automatic error/performance issue detection, version comparison, and all the other sub-features of APM tools to better understand what benefits can these new features bring in terms of monitoring the microservice architecture as almost none of them were represented in the existing monitoring tools in Ridango.

The results from these tests indicated that the existing monitoring tools were not designed to perform an in-depth investigation to assess problems with software performance by themselves. These tools rather provide visibility into these applications, allowing users to manually search for them and investigate them by figuratively "asking the right questions". Observability tools take this process one step further and already perform the analysis for the user, knowing what are the data points that the user might be interested in and how to visualize them. This is crucial for monitoring the microservice architecture, as its complex structure is hard to traverse and investigate with just the manual investigation of metrics and logs.

The APM feature of the three observability tools proved to be similar in the sense of the functionalities they provided, but the experience in using them was quite different. Datadog seems to have the most intuitive workflow when used to find root causes for the mock errors we created. Every step of the process was just a single click away in most cases, displaying detailed data about the affected services, related transactions, logs, and infrastructure associated with the incident. The other major strength of Datadog are its version comparison tools that automatically notify if new errors are detected or if the latency (or some other metric) has become worse after the latest release. New Relic's most powerful feature was its variety of performance metrics and intuitive querying language that allowed to find most of the necessary traces and logs in a relatively short time. Instana's most useful feature seems to be its well-designed 3D infrastructure map that also contains all the underlying services highlighted if an issue is detected. This feature allows a smooth top-down mitigation process that can easily pinpoint which services are affected by an issue. However, as we tried to find the root causes for the mock scenarios with all of the tools in this comparison, Datadog proved to enable the shortest time to detect (TTD) and time to mitigate (TTM) for the user in all 3 scenarios, thus being the highest rated tool in this category.

4.3.3 Alerting

According to the gathered requirements from the stakeholder interviews, alerting was one of the most requested features that the firm's monitoring tool should have. The ability to configure automatically triggered alerts is a feature that enables support engineers, system administrators, and developers to detect erroneous behavior in a much shorter response time. This is crucial to ensure that the clients of Ridango are able to perform their mission-critical tasks without any major disturbances. In this category, we compared the monitoring platforms' functionalities of creating, modifying, and calibrating alerts and the options these tools offer for setting thresholds and triggers for the them. For example, if the tool enables to define alerts based on certain logs or traces as well.

The existing tools besides the Kubernetes dashboard, which does not have an alerting feature, performed reasonably well in this category in terms of the mere ability to set automatic alerts sent to the notification channels in Microsoft Teams. They were all relatively equal in terms of options they offered. Zabbix had a slightly more intuitive UI and configuration processs for alerts than Prometheus & Grafana and EFK setups.However, the problem with the existing alerts setup is that it is distributed between 3 different tools requiring the alerting feature to be maintained in all of them. Additionally, the currently used tools do not support anomaly detection with AI models, which proved to be a useful feature in the observability tools.

The observability tools ranked higher in this category mainly because of the more extensive set of options to define alerts. These features encouraged the user to think more about the mission-critical processes that exist for the company, providing ways to monitor them in an end-to-end fashion. Datadog had the widest variety of parameters and options for defining alerts. For example, its unique feature was to define forecast monitoring alerts, which uses the abilities of its machine learning models to predict if a threshold is being surpassed in the future according to the recent trends. The tool also enabled to define synthetic tests for browsers by just recording the browser journey of the user who is configuring the alert. Instana and New Relic both supported synthetic monitoring, but they required the programmatic approach where the configurer must define each step of the test manually. IBM Instana's advantage was its large library of predefined alerts for different technologies it detects in the system. These alerts proved to be valuable, as some of them were able to find problems that the developers would have hardly noticed. New Relic had the best set of features to calibrate the alerts based on how often these alerts have been triggered in the past. This helps to avoid the situation where one of the alerts starts triggering too often and loses its credibility. Also, New Relic had the option to define groups of alerts that correspond to each team of users that is defined in the platform. This enables to easily delegate which team should react to an alert that has been set off. Ultimately, the deciding factor that made Datadog the highest-rated tool for this category was the ease of configuration and wide variety in the alerting feature. However, Instana and New Relic were still almost equal to Datadog in terms of the quality they provided.

4.3.4 Logs

This category compares the functionalities to collect logs from all of the microservices and components inside the Kubernetes cluster, display them in a user-friendly UI and filter them according to rules specified by the user. The logs feature was included in only five of the seven tools compared in this study. Kubernetes dashboard allows to monitor logs that the services send to the standard output or log files, but it does not have any features to query or filter them. Instana allows to collect and display only error and warning level logs, thus ranking lower in this study than Datadog, New Relic, and EFK. The EFK setup of Ridango is a powerful tool to find logs with querying, but it does not have the means to detect log patterns and anomalies with AI or to show logs in the context of metrics and traces because these data types are not collected by it. Datadog is the highest rated tool in this category, as it has all of the mentioned features, and compared to New Relic, it allows users to define a more sophisticated pipeline in the service website that can be used to configure ingestion, indexing, retention and archiving the logs to the cloud. This level of configuration is not necessary for every solution, but this allows to filter the unnecessary logs from being sent to the system and add additional metadata to further improve the logging setup's value. During the assignment of priority ratings, we found it hard to predict the importance of logs compared to traces, which often contain even more useful information that allows the user to understand the cause of a problem or performance issue. The developers ranked the logs feature relatively high, but it would be interesting to see if this remains true when they have already gotten used to the tracing feature.

4.3.5 Infrastructure

This category was used to primarily measure the monitoring tools' abilities to provide infrastructure data in the context of monitoring microservices as this would allow the software developers to also take advantage of the feature, using the infrastructure monitoring data to mitigate problems with the software running on it. We also considered the visualization tools and maps of these solutions to be an important aspect of this category since the complexity of infrastructure in the microservice architecture is usually much more layered and complex than the monolithic approach. Thus, it seemed almost mandatory that the monitoring tool should be able to observe and visualize the state of physical hardware, virtual machines, Kubernetes clusters, nodes, pods, and the containers running inside them. As the results indicated, the existing monitoring tools were mostly equal in terms of providing relevant metrics for the infrastructure but did not offer the necessary visual maps and correlation with their hosted software components. The observability tools provided interactive maps and simple navigation between different types of related infrastructure and microservices. This allowed to clearly see if the issue with the software is related to infrastructure or, in some cases, if the software itself is affecting the state of an infrastructure component. A similar result in some situations would be achievable with Prometheus, as it collects metrics from both software and infrastructure components. Nevertheless, this would mean a significant amount of manual configuration and creating graphs for each combination of microservice and its host, which would be predictably unmaintainable in a longer period.

The observability tools in this category were almost equal in terms of the quality of their features for infrastructure monitoring. Instana had the most interactive visual map that enables the user to observe virtual machines and Kubernetes nodes in tower-like structures which contain all of the detected services running on them. This allowed understanding the impact area of a problem fast without too much manual work by the users themselves. Datadog had the feature to combine infrastructure monitoring data on graphs on the page for observing individual traces. This was again extremely useful, as the coupled data can give a lot of insight into the whole system when some kind of behavior is being investigated. For New Relic, there were similar features present, but they were usually available in separate tabs and therefore required slightly more effort from the user.

4.3.6 Integrations

The "Integrations" category was graded in a more objective manner, approaching this subject with a binary question: "Does the integration for this technology exist for the tool?" (grading 5 for positive and 0 in the case of a negative answer). There is an argument that certain integrations provide a better user experience, but during the study, it became clear that the other criteria for comparison already reflect that difference. Importantly, this comparison does not measure the complete variety of integrations that these tools offer, but it rather focuses on the technologies that are currently used in Ridango.With this in mind, certain integrations were clearly more significant to the stakeholder groups, incentivizing us to continue using the priority multipliers in this category.

Datadog had the highest overall coverage of 100% for the required integrations that are needed to monitor the entire self-hosted Kubernetes environment of Ridango. New Relic, Instana, Prometheus, and Zabbix missed the integrations for some of the CI/CD and management tools (Bitbucket and Jenkins), which had the lowest priority of all the integrations that were listed in the comparison. This means that the final score was only slightly better for Datadog and this category did not have a considerable impact on the differences in total quality scores.

4.3.7 Configuration

The "Configuration" category compares the process of integrating the monitoring tools on Ridango's server stack. Since the process of configuration cannot be considered exactly a feature of the monitoring tool, we decided to add the lowest possible priority value of 1 to each attribute in this category. In terms of results, the existing monitoring tools were ranked higher, as they offered an easier process of configuration for each added software component and technology. The problem with Datadog and New Relics configuration is that they both require the installation of host agents and APM agents that have versions for monitoring different technologies that constantly get updates and patches applied to them. Fortunately, we were able to perform most of the configuration steps in Ansible playbooks which significantly simplified these processes. Nevertheless, Datadog and New Relic required at least 4 to 5 full workdays to configure properly.

In contrast, Instana's automatic instrumentation required running a single command to set up 90% of the platform's functionalities on all components and microservices (only needing a few additional environment variables and permission allowances to be set for some technologies). In conclusion, Instana's configuration score was the highest among the observability tools. The only tool that was easier to configure than Instana in this comparison is the Kubernetes Dashboard which is a built-in feature for the Kubernetes platform and can be configured with a few simple commands.

4.3.8 Performance and resource usage

This category compares the monitoring tools in terms of the performance and resource usage metrics gathered from the hosting infrastructure components. The aspects that were compared were CPU usage (number of CPU cores used), memory utilization, and the number of pods needed from the Kubernetes cluster [3]. Generally, the existing tools used considerably fewer resources and were ranked accordingly higher in this category than the observability tools. However, Instana was surprisingly conservative in its resource usage - seemingly due to the lightweight "sensors" and the fact that it collects fewer logs. Datadog and New Relic both had a high overhead, depending heavily on the number of features that were enabled for the platforms. The exact results of the measurements in this category are given in Table 5. The results are averaged over a period of 1 week. The measurements were made by using the Kubernetes Dashboard and Prometheus monitoring tools and calculating the average between their displayed metrics.

Table 5. Resource usage comparison

Metric	**NR**	**Ins**	**DD**	**PG**	**EFK**	**K8sD**	**Zab**
Average nr. of CPU cores used	1.7	1	1.2	0.017	1	0.06	0.2
Average memory usage (GiB)	1.5	2.1	2.5	0.12	1.5	0.12	0.4
Average nr. of pods needed	12	3	9	9	6	3	2

4.3.9 Management features

The "Management features" category covers all of the functionalities in the monitoring tool that enable to manage the solution and the users who can access it. This is necessary for the company to guarantee that the monitoring data is protected from outside attacks and intrusions of privacy, manage the maintenance costs for the monitoring tool and ensure that only the qualified users can modify the existing configuration. We decided to

[3]Importantly, Zabbix did not use any pods in the Kubernetes cluster, but since it used 2 processes on the host VM we considered them equivalent.

use a priority rating of “1” for each of the features in this category because these features are not necessary to the defined stakeholders themselves.

As it is common with many enterprise-level SaaS applications, the observability tools have built-in tools for the user, organization, and billing management. For all of the existing tools, there are third-party integrations available to perform these actions, which gave them a slightly lower score, because of the required effort that is needed to implement these integrations. Since some of the existing tools are also open-source software and hosted in the company’s cloud server, the billing management comparison would not directly apply to them. Therefore, a maximal grade was given to the open-source tools for the "Billing controls" subfeature, as their maintenance cost can be easily derived from the monthly cost of the hosting cloud servers. To summarize the results, there were no significant differences in this category, as the biggest score difference was only 6 points. However, Instana had an advantage over the rest of the observability tools, because of its option to use it as an on-premises monitoring tool and the overall quality of its built-in management features.

4.3.10 Coverage of the requirements

The coverage and the total score for each requirement are represented in Table 6. We concluded from the final results that we should consider adopting one of the observability tools as they have at least two times higher requirement coverage scores and at least 16.4% higher requirement coverage percentage. IBM Instana was the only tool with 100% requirement coverage because of the “on-premises option requirement” (R44) that was missing for New Relic and Datadog. After consulting with the technical management board in Ridango, we decided that this requirement is not mandatory, and it does not eliminate the other observability tools for being adopted. However, the quality scores indicated that Datadog is the highest rated tool in the requirements comparison, leading in both the user requirements and usability requirements categories.

4.4 Cost of maintenance

Figure 13 compares the predicted monthly expenses of the observability tools and the cost of maintenance for the current setup. Plot #1 presents the extent of the total monthly cost of each monitoring tool compared to the current monitoring setup. The set of features of each tool that is included in the price is selected on the basis of the stakeholder requirements, covering the maximal amount of them while staying in the undisclosed budget that was presented by the board of the company. Since Instana’s platform does not enable to retain other than error and warning level logs, it means that we cannot substitute EFK with the solution, thus it is better to compare Instana combined with EFK with the other 2 observability tools (shown in plot #2) which have the logging feature included in their estimated price. The results indicated that Instana’s significantly lower

Table 6. Total scores and coverage of the requirements

Req. ID	**NR**	**Ins**	**DD**	**PG**	**EFK**	**K8sD**	**Zab**
C1	1383.7	1226.8	1515.8	477.4	441.2	308.6	461.6
R1	247.5	210.3	269.4	64	65	67.4	42.3
R2	175	156.8	186.1	64	0	65.1	42.3
R3	98.4	88.7	105.6	36.4	26	19.2	43.5
R4	164.6	161.6	174.2	70	26	46.2	77.4
R5	176.5	140.7	199.4	25.2	91	21.2	35.1
R6	172.1	149.1	187.4	36.4	91	16.7	52.5
R7	8.8	4.4	11	0	0	0	0
R8	7.5	10	12.5	0	0	0	0
R9	25	12.7	22.7	22.7	0	13.5	10.8
R10	14	17.5	12.2	0	0	0	0
R11	12	12	15	12	9	0	12
R12	25.5	25.5	37.5	12	18	0	12
R13	21.5	14.1	20.8	6	8	0	8
R14	9.2	9.2	11.5	0	0	0	0
R15	9	7.2	9	7.2	9	0	9
R16	11.5	6.9	11.5	0	11.5	2.3	0
R17	12	15	12	0	15	0	0
R18	11.5	11.5	11.5	0	0	0	0
R19	11	6.6	11	0	11	0	0
R20	41.7	36.3	44	33.6	0	31.8	24.9
R21	12.5	12.5	12.5	12.5	12.5	0	12.5
R22	39	39	39	39	39	0	39
R23	11.5	9.2	11.5	9.2	9.2	0	6.9
R24	10	16	24.5	16	0	0	16
R25	56.4	54	64	11.2	0	25.2	17.4
C2	202.7	209.3	219.4	208.1	207.7	177.1	194.9
R26	10.8	10.8	13.5	10.8	13.5	10.8	8.1
R27	8.8	11	8.8	6.6	6.6	11	8.8
R28	7.5	7.5	10	12.5	7.5	10	10
R29	8.5	6.8	6.8	8.5	5.1	3.4	5.1
R30	11.5	4.6	11.5	11.5	11.5	9.2	6.9
R31	5.1	5.1	5.1	0	0	0	5.1
R32	6.8	3.4	8.5	6.8	6.8	1.7	5.1
R33	6	10	7	8	8	10	7
R34	3	4	3	5	5	5	3
R35	5	5	5	5	5	5	4
R36	21.2	31.4	23.7	34.5	31.6	39.5	39.5
R37	21.2	31.4	23.7	34.5	31.6	39.5	39.5
R38	5	5	5	5	5	5	4
R39	8	6	10	6	8	0	8
R40	9	7.2	9	7.2	9	0	9
R41	43.3	41.5	48.5	32.7	31.5	13.5	26.4
R42	8.5	5.1	6.8	0	8.5	0	0
R43	13.5	13.5	13.5	13.5	13.5	13.5	5.4
C3	23	28	22	24	24	24	24
R44	0	5	0	5	5	5	5
R45	10	10	9	6	6	6	6
R46	5	5	5	5	5	5	5
R47	5	5	5	5	5	5	5
R48	3	3	3	3	3	3	3
Total	1609.4	1464.1	1757.2	709.5	672.9	509.7	680.5
Coverage %	98%	100%	98%	81.60%	79.60%	65.30%	75.50%

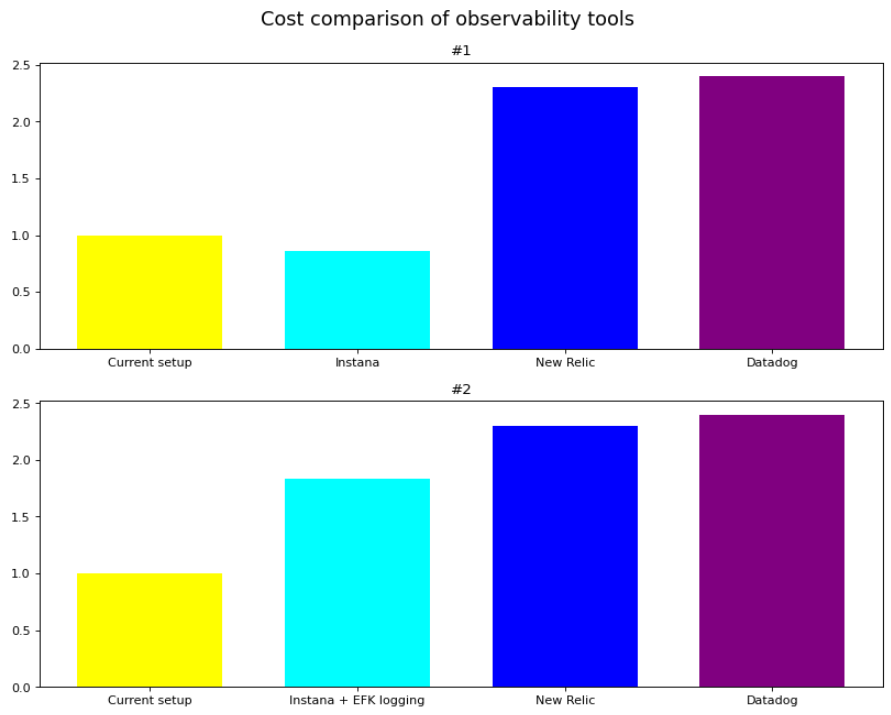

Figure 13. Comparison - Cost of maintenance

pricing might be a very good argument to adopt it in Ridango's monitoring stack. Also, we came to the conclusion that the decision of preference between Datadog and New Relic is not affected by their cost of maintenance, as their overall price would be very similar when adopted for Ridango's stack.

4.5 Return of investment

The ROIs for the tested observability tools were found in the following categories:

- Saving employees' time in resolving issues
- Savings on replacing current monitoring tools
- Reduce infra requirements through optimization
- Increase in customer satisfaction
- Sustainability and better performance of Ridango's software and services

The calculations of the total ROIs are based on the formula [13]:

$$ROI = \ Net\ Return\ on\ Investment\ /\ Cost\ of\ Investment\ \cdot 100\%$$

These results have to be taken with caution, as many of the savings and returns are based on subjective estimations because of the absence of a definitive method to predict these figures. For future work, it is necessary to confirm these predictions to better understand the business value that these tools offer.

The final predictions for the ROIs [4] were:

- New Relic: 93.33%
- IBM Instana: 130.46%
- Datadog: 107.43%

The exact calculations for these figures are not included in this study, as it is considered sensitive information for Ridango.

4.6 Summary of the findings

According to the results that were presented in the feature comparison and ROI calculations, the observability tools would be a clear improvement to the tools in the existing monitoring setup. They offer a wide range of functionalities to better observe microservices and the infrastructure beneath them, understand the causes of erroneous behaviors, and react to them sooner. The general quality of New Relic, Instana, and Datadog platforms was high and in many situations almost indistinguishable. Datadog proved to be the highest rated tool in the feature and requirements comparison while also having the highest cost of maintenance. Instana was the observability platform with the lowest score in feature comparison, but as the ROI calculations suggested, its competitive cost makes it a still a very viable choice since the tool still enables most of the functionalities that are present in the other two platforms.

5 The adopted monitoring solution

Based on the results from the feature comparison, requirements comparison, and ROI calculations, the board of Ridango decided to adopt Datadog for a trial period of 6 months. Within this period, we are able to measure if the ROI predictions and cost

[4]Calculations are based on predicted monthly net return and cost of investment.

calculations were accurate. Regardless of the remarkable abilities that Datadog presented in the research and comparison phases of this study, we are certain that the mentioned ROIs cannot be achieved by merely installing the tool and starting to use it. Instead, it is required to purposefully choose the features that are needed by the users and understand how the new tool integrates with the existing ecosystem of technologies, tools, and business logic of the company. Additionally, we concluded from the user requirements analysis in Section 3 that a more active approach is required from the technical teams to keep the monitoring solution updated and prepared for the next incident that might occur. This section contains the proposed solution to Ridango's management board about which tools and features to use and how to integrate the observability tool into the development and operations workflows more efficiently to maximize the ROI of the new setup.

5.1 Description of the setup

5.1.1 Set of tools

The adoption of Datadog incentivizes Ridango to replace many of the functionalities of the existing monitoring tools with their equivalent features in Datadog to create a more unified monitoring solution across all teams. The suggested setup takes this into account and proposes the usage of the following setup for monitoring:

Datadog - is the primary monitoring tool for the production environment to cover most use cases for the monitoring tools. The features of Datadog [2] that are proposed to be used for Ridango's setup are listed in Table 7 along with their primary targets.

Feature	**Targets**
Infrastructure monitoring	Kubernetes Live cluster, Database hosts, Kafka and Zookeeper nodes
APM & Continuous Profiler	All microservices in Kubernetes Live cluster
Log Management	Infrastructure and APM targets' logs
Database Monitoring	PostgreSQL and Redis databases
Synthetic monitoring	Most critical backend endpoints, customer service portal websites
Real User Monitoring	All web applications in the live environment

Table 7. Datadog features used for the suggested setup

Prometheus - will be still used as a metric provider for all of the development and testing environments. we also integrate Prometheus with Datadog to send the already defined custom metrics to the Datadog UI.

Kubernetes Dashboard - will be kept available (for all environments) for being able to easily observe the Kubernetes clusters and execute commands.

The features of querying logs and monitoring infrastructure from EFK and Zabbix will be completely replaced by Datadog.The new monitoring solution for Ridango is depicted on Figure 14.

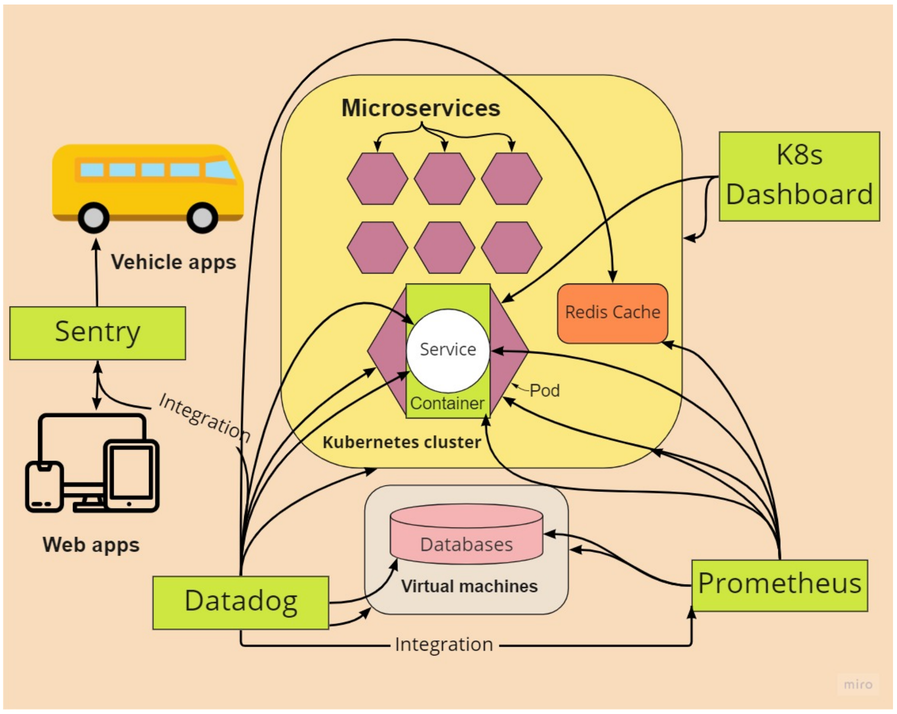

Figure 14. The proposed monitoring setup for Ridango

5.1.2 Integrating technologies and other tools

Datadog supports various technologies and developer tools to be integrated with the monitoring platform to provide a more coherent behavior with the stack and support the existing workflows within the company. However, many of these configurations can add additional complexity to the tool, making the platform harder to learn and use efficiently. Thus, it is important to consider each integration carefully, understand their purpose, and test if they indeed improve the monitoring setup in reality. Table 8 lists the technology and developer tool integrations to be considered and their purpose for this setup.

Table 8. Datadog integrations used for the suggested setup

Name of the integration	**Purpose**
Java, PHP, Node.js	Use more customized metrics, alerts and dashboards to gain better observability.
Database integrations: PostgreSQL, Redis	Monitor databases with more metrics and visibility into the used queries.
Kafka, Zookeeper	Monitor Kafka and Zookeeper nodes, use more customized metrics and alerts for these technologies.
Ansible	Track the completion and failures of ansible pipelines.
Microsoft Teams	Primary notification channel for the alerts.
Jira	Incident management, creating tickets from alerts.
Bitbucket	Add code change markers to dashboards, track deployment events
Prometheus	Ingest already defined custom metrics

5.1.3 Customization and business-oriented monitoring

In addition to implementing the mentioned features and integrations, the new setup should be also configured to observe the system from the view of mission-critical functionalities that Ridango's services provide. This would allow the users of the monitoring solution to understand during incidents what is the exact area of impact and how it affects the firm's clients. Also, this type of data can be effectively used as a feedback statistic about the products' quality of service, allowing the management board to see which parts of the system might need improvement and the additional investment of developer time. With Datadog, it is possible to define custom metrics and alerts based on traces to achieve this kind of configuration.

For Ridango the following functionalities are considered to be primary targets for business-oriented monitoring:

- Ticket sales from vehicles
- Ticket sales in online customer portals
- Ticket validations
- Ticket inspection
- Real-time arrival predictions and location providing

5.2 Integration

The process of integrating Datadog and transforming the existing monitoring setup into the proposed solution is a relatively simple task, but it contains many steps that need to be planned and gradually carried out to achieve the expected end-goal. This subsection contains the proposed plan to integrate Datadog with the stack of Ridango which is presented in the following six phases:

Phase I - Preparations

Before the installation of agents and configuration of the Datadog platform starts, a few preliminary tasks should be completed. In this phase, it is required to set up Jira and Microsoft Teams to support the alerting workflow described in Subsection 5.3. Also, it would be necessary to configure the Single-Sign-On feature for distributing Datadog access to the employees of the firm and configure the creation of audit logs to guarantee better internal security for the tool.

Phase II - Monitoring Kubernetes and microservices

In this phase, development teams should configure the monitoring setup for the Kubernetes production cluster infrastructure and the microservices on it. These configurations can be made by referencing the Kubernetes deployment files and Ansible playbooks that were created during this research. After the completion of this phase, all of the metrics, logs, and traces of every infrastructure component (virtual machine, Kubernetes node, pod, container), process, and microservice should be available in the Datadog UI.

Phase III - Monitoring Databases and Kafka

This phase consists of setting up monitoring for PostgreSQL and Redis databases and the virtual machines that host them. Additionally, this should be done for Kafka and Zookeeper nodes. The development teams should also look into the custom dashboards provided for these technologies by the Datadog platform.

Phase IV - Integrating Prometheus, Ansible, and Bitbucket

In this phase Prometheus custom metrics are migrated to Datadog, and new dashboards are created for them. Ansible and Bitbucket integrations are further tested and then integrated into the setup if they are deemed to be valuable to the monitoring setup.

Phase V - Team-based configurations

For this phase, the internal teams of the firm should configure additional dashboards and team-based views for monitoring the software components that are in their area of responsibility.

Phase VI - Shutting down replaced tools

In this phase the replaced tools of the current setup are uninstalled for the system to save on resource consumption and the cost of maintaining these tools. Currently, the replaced tools are considered to be Zabbix and EFK. This phase should be completed after the evaluation period of Datadog, in the case it was successful.

Future plans

After the completion of the described integration plan, there are still some possibilities that can be researched and explored in the future. The primary aspect that was mostly uncovered during this study was the Real User Monitoring (RUM) feature for the observability tools as this is not associated directly with monitoring microservices. For future works, it would be valuable to test this feature further for web and device applications and compare its capabilities to the current error detection software Sentry. Additionally, as some of Ridango's services depend on the AWS cloud platform, it would be an opportune target for being added to the already existing Datadog monitoring solution.

5.3 Proposed monitoring conventions and workflows

5.3.1 Unified naming

The practice of unified naming conventions is used throughout the field of software engineering to improve the readability of code and to avoid misinterpretations. It would be important to use this concept in naming custom metrics and alerts in order to make them understandable for everyone, without the need of looking into the actual configuration of the alert itself. This would be especially necessary for the support staff, who need to react to the alerts or displayed graphs but do not configure them by themselves. Datadog's alert configuration allows defining special tags that can be later used to filter the alerts or send alerts to correct notification channels. This would be useful for the developing teams of Ridango, as they can create views that display the alerts based on the category of a component or the team name.

5.3.2 Efficient alerts and notifications

Besides configuring and naming the alerts, there is always the question of who should receive the alerts and how should the recipient act upon them. The reality with the existing setup in Ridango has shown, that without a clearly defined and unified approach, the monitoring tools get flooded with notifications of outdated alerts that have lost their credibility. These notifications get ignored and eventually muted, which means the coverage of these automatic alerts is low and insufficient to actually rely on them.

To counter this problem, there are a few universal rules that are proposed:

1. Every alert notification should be reacted to.
2. After every major change with a component, all of its alerts should be updated (if required).
3. After every new component is added to the monitoring setup, at least the minimal set of “Default alerts”[5] needs to be added to the component.

Rule 1 means that even if the sent notification was a false positive, the recipient should react to the alert by re-calibrating its thresholds.

To ensure that there is always a person to react to these notifications and to avoid duplicate reactions the suggestion would be to use a strict protocol based on the severity of these alerts. Datadog enables to define of alerts with five levels of severity: critical, high, medium, low, and info. The alerting workflow depicted in Figure 15 using only the critical, medium, and low levels. The idea behind the workflow is to send the alert notifications to separate alert channels where the recipients are responsible to react to them: support engineers react to critical alerts and Release Managers (RM) of development teams react to medium and low level alerts [6].In the case of a false positive, the investigator re-configures the alert with adjusted thresholds. Otherwise, a Jira ticket is created for critical and medium level alerts. Low level alerts are already resolved after the initial inquiry as these are mostly used as additional information during error mitigation. The expected reaction time for the alerts is up to 2 hours for critical alerts, 8 hours for medium, and 3 workdays for low level alerts.

5.3.3 Introducing optimization

The final proposal for Ridango would be to introduce optimization patches to the workflow of deploying major and minor releases to their services. The development and support teams of the company are used to reacting to drastic and noticeable changes in the performance of the services in the live environment while most of the optimization is performed in the testing phase of software production. With the added support of version comparison and release monitoring features in Datadog, it would be a considerable improvement to the performance of these services to actively introduce optimization patches after each major or minor release. This would also motivate developers to think more about the quality and performance of their code from the start to avoid the additional optimization tasks. The primary targets for this kind of patches would be endpoints and database queries. Collective experience in the company has shown that these two

[5]defined in Ridango’s internal documentation

[6]Release Manager is a rotational on-call role in development teams in Ridango that is responsible for solving ongoing incidents and deploying software releases.

mission-critical components of any back-end service are sometimes very difficult to performance test because there is a significant difference in the amount of traffic and data flow in the test- and production environments. Even with a large set of mock data, sometimes these slight performance issues can be unnoticed and released to production where they impact the user experience of the deployed services. The database monitoring feature and synthetic endpoint checks of Datadog should help the developing teams to catch these problems faster, reducing the negative impact on the customers' satisfaction.

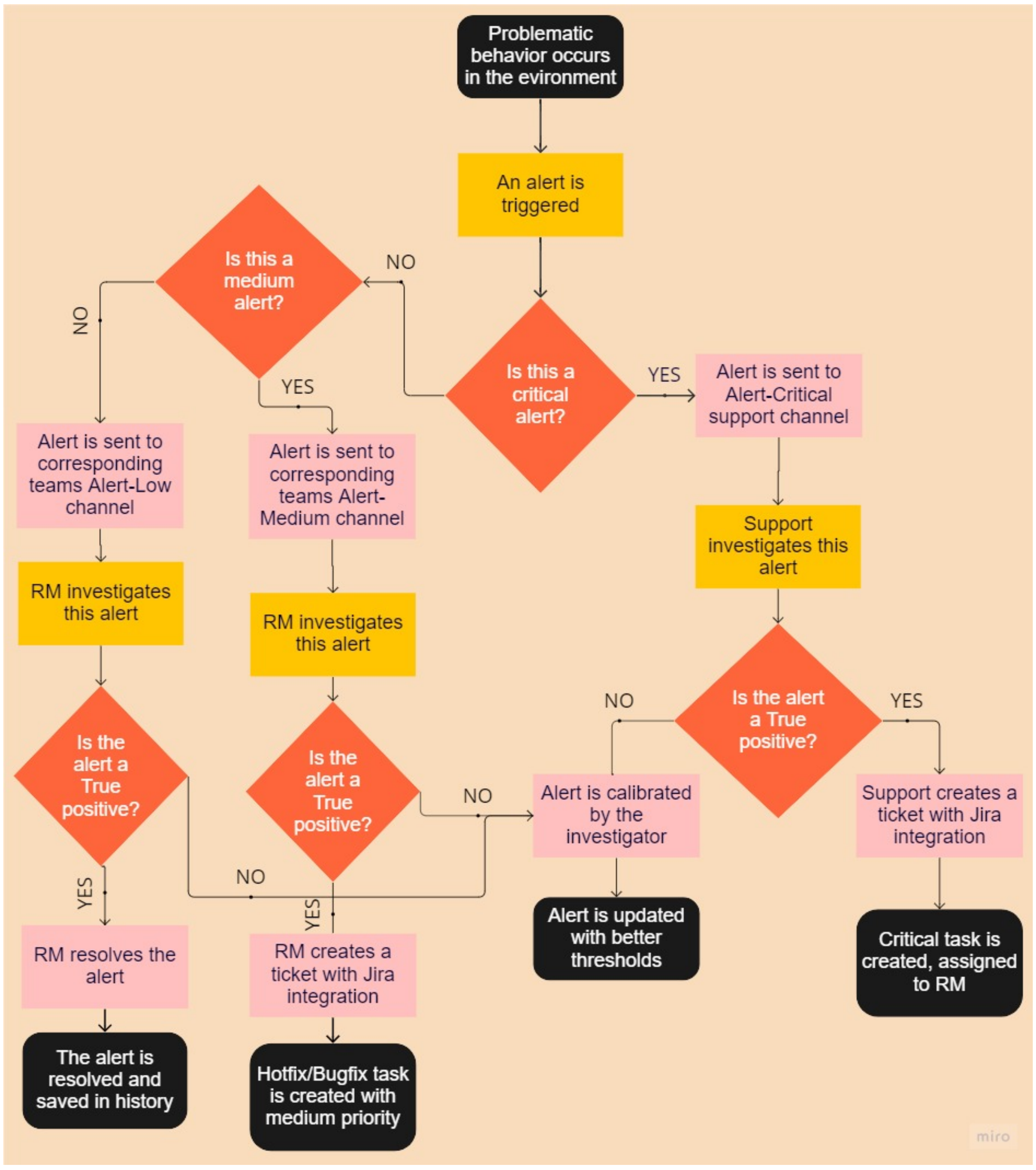
Problematic behavior occurs in the evironment
An alert is triggered
Is this a critical alert?
YES
Alert is sent to Alert-Critical support channel
Support investigates this alert
Is the alert a True positive?
YES
Support creates a ticket with Jira integration
Critical task is created, assigned to RM
NO
Alert is calibrated by the investigator
Alert is updated with better thresholds
NO
Is this a medium alert?
YES
Alert is sent to corresponding teams Alert-Medium channel
RM investigates this alert
Is the alert a True positive?
YES
RM creates a ticket with Jira integration
Hotfix/Bugfix task is created with medium priority
NO
NO
Alert is sent to corresponding teams Alert-Low channel
RM investigates this alert
Is the alert a True positive?
YES
RM resolves the alert
The alert is resolved and saved in history
NO
miro

Figure 15. Proposed workflow for responding to triggered alerts

6 Conclusion and future work

The purpose of this section is to summarize the thesis and to discuss the possibilities for future work. This thesis describes the process of redesigning a traditional monitoring setup to better support the maintenance of a distributed microservice architecture system in the example of Ridango's self-hosted Kubernetes environment. In this study, we conducted three following procedures to find a better monitoring solution for the company of Ridango:

1. Internal stakeholder identification and user requirements analysis for monitoring.
2. A comprehensive comparison between three observability tools: New Relic, Instana, and Datadog; and four monitoring tools used in the existing monitoring setup in the firm: Zabbix, Prometheus & Grafana, Kubernetes Dashboard, and EFK (Elasticsearch, Fluentd and Kibana).
3. Description of the proposed monitoring setup to Ridango based on the Datadog observability tool.

The results from the comparison of Section 4 imply that observability tools presented in this study are indeed more capable of supporting the gathered monitoring requirements and usage scenarios of the stakeholders within the company and are better adjusted to the decoupled microservice architecture than the existing monitoring setup. The observability platform of Datadog proved to be the most potent tool for monitoring Ridango's services by having the highest ROI score and saving the most amount of developer time in the conducted tests. Another important finding from the conducted research was that a more systematic and unified approach is needed to configure and maintain the monitoring solution to increase its overall benefits. The proposed monitoring setup in Section 5 addresses this issue by suggesting: a unified naming convention for metrics and alerts, a new workflow and set of rules to deal with triggered alerts, methods to monitor business-critical functionalities and to optimize software performance with the support of the observability tool of Datadog. In terms of future work, there are several limitations to this study that can be improved. Firstly, many aspects in the comparison of the monitoring tools were evaluated using subjectively determined grades by testing these tools in similar scenarios. This can be improved upon by gathering grades from a larger sample of developers and experts on this subject or defining more objectively measurable criteria. Secondly, it would be important to add more observability tools to this comparison, especially open-source tools to provide a better general overview of the state-of-the-art options for monitoring the microservice architecture. Finally, it would be beneficial to gather data about the new monitoring setup that was proposed in this study and measure if our predictions for the return of investment were accurate. This would allow us to further understand the efficiency of the observability tools and introduce additional changes to the monitoring solution.

References

[1] About Ridango. https://ridango.com/about-us/ (Last accessed 21.03.2022).

[2] Datadog pricing and list of features. https://www.datadoghq.com/pricing/ (Last accessed 04.05.2022).

[3] Elasticsearch: The Official Distributed Search & Analytics Engine. https://www.elastic.co/elasticsearch (Last accessed 05.05.2022).

[4] Kibana: Explore, Visualize, Discover Data | Elastic. https://www.elastic.co/kibana/ (Last accessed 18.04.2022).

[5] Overview, how Ansible works. https://www.ansible.com/overview/how-ansible-works (Last accessed 08.05.2022).

[6] Overview: What is Prometheus? https://prometheus.io/docs/introduction/overview/ (Last accessed 02.05.2022).

[7] What is Observability? A Beginner's Guide. https://www.splunk.com/en_us/data-insider/what-is-observability.html (Last accessed 02.04.2022).

[8] What is Zabbix? https://www.zabbix.com/documentation/5.4/en/manual/introduction/about (Last accessed 05.05.2022).

[9] Magic quadrant for application performance monitoring, 2020. https://www.gartner.com/doc/3983892 (Last accessed 02.03.2022).

[10] Best observability solution suites software, 2021. https://www.g2.com/categories/observability-solution-suites (Last accessed 08.05.2022).

[11] Kubernetes components, 2022. https://kubernetes.io/docs/concepts/overview/components/#web-ui-dashboard (Last accessed 09.05.2022).

[12] What is Kubernetes?, 2022. https://kubernetes.io/docs/concepts/overview/what-is-kubernetes/ (Last accessed 08.05.2022).

[13] Andrew Beattie. A Guide to Calculating Return on Investment (ROI). https://www.investopedia.com/articles/basics/10/guide-to-calculating-roi.asp (Last accessed 27.04.2022).

[14] Nico Faraguna, Aime Anne Nisay, Adhiraj Somani, Carla Faraguna, Erin Scherfner, Holden Page, Jude Gomila, Katrina-Kay Alaimo, Dawson Sewell, and Qwei Cont. Sentry (software company) - Wiki. https://golden.com/wiki/Sentry_(software_company)-EAAMVBX(Last accessed 01.05.2022).

[15] Martin Fowler and James Lewis. Microservices, March 2014. `https://martinfowler.com/articles/microservices.html` (Last accessed 05.05.2022).

[16] Martin Maguire and Nigel Bevan. User requirements analysis. In *IFIP World Computer Congress, TC 13*, pages 133–148. Springer, 2002.

[17] Theo Schlossnagle. Monitoring in a DevOps world. *Communications of the ACM*, 61(3):58–61, 2018.

[18] Yuri Shkuro. Observability challenges in microservices and cloud-native applications, 2019. `https://medium.com/@YuriShkuro/observability-challenges-in-microservices-and-cloud-native-applications-72857f9c` (Last accessed 04.05.2022).

[19] Jesper Simonsson, Long Zhang, Brice Morin, Benoit Baudry, and Martin Monperrus. Observability and chaos engineering on system calls for containerized applications in Docker. *Future Generation Computer Systems*, 122:117–129, 2021.

[20] James Turnbull. *The art of monitoring*. James Turnbull, 2014.

Appendix

I. List of Acronyms

Acronym	**Meaning**
MA	microservice architecture
VM	virtual machine
OS	operating system
AWS	Amazon Web Services
Zab	Zabbix
PG	Prometheus and Grafana
EFK	Elasticsearch, FluentD and Kibana
K8sD	Kubernetes dashboard
NR	New Relic
Ins	IBM Instana
DD	Datadog
URA	user requirements analysis
Sysadmin	System administrator
APM	application performance monitoring
TTD	time to detect
TTM	time to mitigate
UI	user interface
AI	aritifical intelligence
CI	continuous integration
CD	continuous deployment
ROI	return of investment
RM	release manager

II. Licence