Institute of Software Technology
Reliable Software Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Bachelorarbeit

Application Performance Monitoring in Microservice-Based Systems

Valentin Seifermann

Course of Study:	Softwaretechnik
Examiner:	Dr.-Ing. André van Hoorn
Supervisor:	Georg Högl, Industry Partner, Dr.-Ing. André van Hoorn
Commenced:	May 30, 2017
Completed:	November 30, 2017
CR-Classification:	C.4, H.3.4

Abstract

Nowadays, cloud computing including its functionality and service delivery model, turned into a common paradigm to provide on-demand IT resources and scalability. This also leads to a change of architecture from monolithic systems to a microservice-based architecture with high scalability and elasticity as well as independent developments and deployments. Due to this change, IT environments are getting more complex and highly distributed. Thus, the requirements of Application Performance Management (APM) for these types of systems are also changing.

While monitoring monolithic applications and infrastructures focuses on a few single components with their callstacks and the surveillance of the general state of health, the monitoring of microservice-based systems requires other approaches.

This work will elaborate the actual state-of-the-art of APM in microservice-based systems on the basis of an industrial case study. Furthermore, the challenges of monitoring microservice-based systems will be elaborated. As part of the industrial case study, an existing microservice monitoring tool will be evaluated on different environments and integrated into the state-of-the-art business-oriented monitoring strategy "System Management: Inform-Locate-Escalate" (SMILE), which consists of various monitoring and service management tools. The evaluation will be done by conducting use cases, defined in the context of the thesis to meet the challenges.

In addition, the work proposes an experimental concept for APM in microservice-based systems. This concept consists of a selected monitoring stack with different open-source tools and an existing microservice monitoring solution. Furthermore, it contains different dashboards with decisive metrics and other monitoring-specific data.

Kurzfassung

Heutzutage ist Cloud Computing mit seiner Funktionsweise und den verschiedenen Diensten zu einem beliebten Paradigma geworden, um dynamisch IT-Ressourcen bereit zu stellen und horizontale Skalierbarkeit zu gewährleisten. Der damit verbundene Wandel von monolithischen Systemen hin zu Microservice-basierten Systemen bringt eine hohe Skalierbarkeit und Elastizität sowie unabhängige Entwicklung und Deployments mit sich. Infolgedessen werden IT-Umgebungen hierdurch komplexer und verteilter. Dies führt auch zu einer Veränderung der Anforderungen im Bereich des Application Performance Management (APM).

Während das Monitoring von monolithischen Applikationen und Infrastrukturen den Fokus auf einzelne wenige Softwarekomponenten mit ihren Callstacks und die Überwachung des betrieblichen Allgemeinzustandes setzte, erfordert das Monitoring von Microservices und daraus bestehende Umgebungen eine andere Herangehensweise.

Ziel dieser Arbeit ist es, anhand einer industriellen Fallstudie den aktuellen Stand des Monitorings von Microservice-basierten Systemen zu ermitteln. Hierzu werden die Herausforderungen für das Monitoring dieser Umgebungen erörtert. Als Teil der Fallstudie wird hierfür ein bereits existierendes Monitoring Tool mit seinen Ansätzen anhand verschiedener Umgebungen evaluiert und in die Business-orientierte Monitoring-Strategie SMILE integriert. Die Evaluierung erfolgt durch Use-Cases, welche anhand der Anforderungen im Rahmen der Arbeit definiert werden.

Zusätzlich schlägt diese Arbeit ein experimentelles Konzept für das Monitoring von Microservice-basierten Systemen vor. Dieses Konzept besteht aus einem ausgewählten Monitoring Stack mit verschiedenen open-source Tools sowie einer bestehenden Monitoring-Lösung. Außerdem beinhaltet es verschiedene Dashboards und andere Monitoring-spezifische Daten.

Contents

List of Figures

List of Tables

List of Acronyms

APM Application Performance Management

AWS Amazon Web Services

Amazon EC2 Amazon Elastic Compute Cloud

HTTP Hypertext Transfer Protocol

IaaS Infrastructure as a Service

JSON JavaScript Object Notation

JVM Java Virtual Machine

KPI Key Performance Indicator

PaaS Platform as a Service

REST Representational State Transfer

SaaS Software as a Service

SMILE "System Management: Inform-Locate-Escalate"

SOI CA Service Operations Insight

List of Listings

“The pace of scale and dynamism of the infrastructure and application environment are accelerating, with the advent of containers, microservices, autoscaling and software-defined ‘everything’ stressing the capabilities of existing Application Performance Management (APM) tools”

Cameron Haight, VP Gartner’s IT Systems, Security and Risk research group

From “APM Needs to Prepare for the Future”, June 2016

Chapter 1

Introduction

1.1. Motivation

Due to the concept of agile development and for the purpose of continuous delivery, developers are tending to develop applications into independent and autonomous microservices to ensure single scalability and independent deployments. This microservice adaption is driven by the separation of applications on granular levels which leads to autarchic components [MS12]. Within these environments, microservices are small functional units providing a service with a loose connection to other services. Therefore, microservices are distributed into containers as several deployment units with the help of container management systems such as Kubernetes.

Companies like Netflix, Amazon and Google accomplished this transition [ROAM17] and also automotive companies are tending to develop the backends of car applications as microservice-based environments on cloud infrastructures. This also leads to an adaption of the existing monitoring tools and strategies. As a result, it is necessary to inspect how microservice monitoring can be achieved and integrated into these industrial environments since Haigt [Hai16] also mentions, that previous APM solutions are reaching their limits due to the new technology stacks and paradigms in regard to the monitoring and development of microservice-based systems. Also Heinrich et al. [HHK+17] are pointing to upcoming challenges in regard to microservice monitoring. As shown in Figure 1.1, the scalability and the different technologies used in microservice-based environments are leading to challenges in regard to metrics, profiling, analytics, anomaly detection as well as data collection.

In regard to these challenges it is necessary to inspect how already existing monitoring solutions are overcoming the challenges. Since the previous case study [BSS10] was focusing on conventional monitoring solutions and monolithic environments, a new case study for microservice monitoring has to be designed to see if existing solutions are overcoming the new challenges. In addition, the new microservice monitoring solutions also have to be integrated into existing IT monitoring landscapes and strategies to trigger automatic alert processing and business oriented monitoring. This includes the connection to service management tools such as CA Service Operations Insight (SOI) which are part of the SMILE strategy. Moreover, a new monitoring concept with microservice-specific dashboards and metrics has to be elaborated.

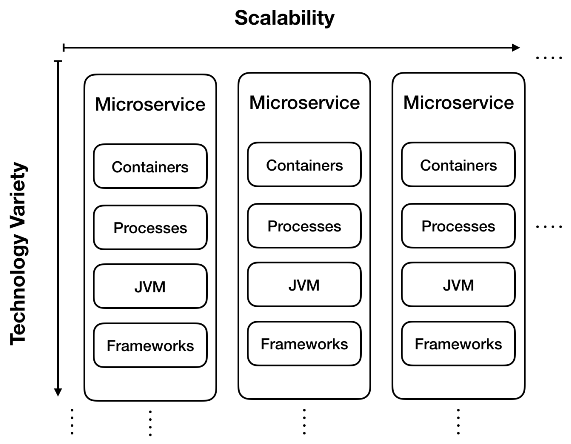


Figure 1.1.: Scalability and technology variety

1.2. Industry Partner & Microservices in industrial systems

The thesis will be conducted in cooperation with an industrial partner which is an automotive company. The industry partner also uses cloud infrastructures and agile development strategies for upcoming car application services. Moreover, he is also tending to take benefits of cloud delivery models and backend services build with the microservice paradigm. Due to the use of these microservice-based architectures, the industry partner aims for extracting advantages from scalability and DevOps processes. Further, the roll out of big releases as monolithic components will be avoided. This offers the possibility to estimate the current status in comparison to the defined requirements and to react dynamically to possible delays of the development. In contrast, challenges are rising in terms of monitoring, infrastructure and IT security requirements.

1.3. Goals

The goal of this work is to elaborate and assess the state-of-the-art of microservice monitoring based on a real-world industrial setting. Therefore, an industrial case study will be designed by using defined use cases as well as a commercial microservice monitoring solution. During the case study, the focus will also be set on the integration of the monitoring solution into the existing monitoring strategy SMILE by using connectors to connect to SOI. Moreover, the collected information from the case study will be used to develop an experimental microservice monitoring concept with specific dashboards by using the Instana Represental State Transfer (REST) API, Grafana and InfluxDB.

1.4. Thesis Structure

The structure of the thesis will be presented in the following:

Chapter 2 (Foundations and State of the Art) provides important information about technologies, paradigms and challenges as a foundation for understanding the thesis. Furthermore, the work which is related to previous case studies and the elaborated concept, will be discussed.

Chapter 3 (Case Study) provides the case study including the case study systems, the defined use cases and the evaluation.

Chapter 4 (The Monitoring Concept) presents the elaborated concept

Chapter 5 (Conclusion) summarizes the results of the thesis and provides directions for future work.

Chapter 2

Foundations and State of the Art

This chapter provides fundamentals and related work for the industrial case study and the elaborated monitoring concept. Section 2.1 provides fundamentals about technologies and paradigms in regard to APM and microservice-based systems. Section 2.2 presents the challenges of monitoring microservice-based systems, while Section 2.3 discusses the work related to the thesis topic in regard to the industrial case study and the elaborated concept. At least, Section 2.4 will give an overview of the tools used for the industrial case study and the elaborated concept.

2.1. Fundamentals

2.1.1. Cloud Computing

Cloud computing is a major technology that offers scalable and flexible IT infrastructure and resources as different services reachable through the internet. According to the NIST definition of cloud computing, Mell et al. [MG11] are defining this model, as a "*model for enabling ubiquitous, convenient, on- demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, and services) that can be rapidly provisioned and released with minimal management effort or service provider interaction*". Moreover, the underlying compute resources of cloud-based services and software are entirely abstracted from the consumer of the services. As a result, the vendor of cloud-based resources is taking responsibility for the performance, reliability and scalability of the computing environment and the delivered IT services. The provided services can be categorized in different services as shown in Figure 2.1, namely Software as a Service (SaaS), Platform as a Service (PaaS) and Infrastructure as a Service (IaaS). SaaS provides the use of on-demand applications over the internet. In contrast PaaS provides an development environment including software development frameworks and platform layer resources on the existing cloud infrastructure. The fundamental of the cloud delivery model is IaaS. This service includes the provision of bare resources like computational power and memory [ZCB10]. Furthermore, it is possible to instantiate virtual machines with operating systems as fundamentals for microservice-based environments.

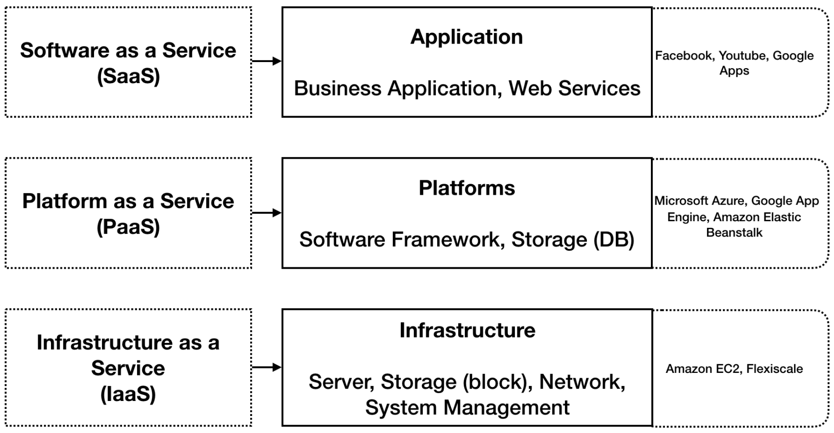

Software as a Service (SaaS)
Application
Business Application, Web Services
Facebook, Youtube, Google Apps
Platform as a Service (PaaS)
Platforms
Software Framework, Storage (DB)
Microsoft Azure, Google App Engine, Amazon Elastic Beanstalk
Infrastructure as a Service (IaaS)
Infrastructure
Server, Storage (block), Network, System Management
Amazon EC2, Flexiscale

Figure 2.1.: Cloud computing services

2.1.2. Microservice Architectures & Technologies

Microservices

The main approach of microservices in context of service oriented architecture is the break down of monolithic systems into small services communicating with each other. Figure 2.2 displays the microservice paradigm in comparison with monolithic systems. It shows that microservices are small services with own data storage. In contrast the monolithic figure simplifies that all functional elements including the data storage are integrated in a single component. As a result, the microservice paradigm leads to single autonomous components, which are loosely coupled. Today´s microservices are using REST as a common way to communicate with other microservices. The underlying data transfer will be done with the Hypertext Transfer Protocol (HTTP). Due to the independence between microservice they are mostly autonomously developed and easily scalable. In context of continuous deployment, microservice can be deployed independently. In summary, Newman [New15] defines microservice with the following characteristics:

- Fine-grained
- Bounded by contexts

- Autonomously developed
- Independently deployable
- Decentralized

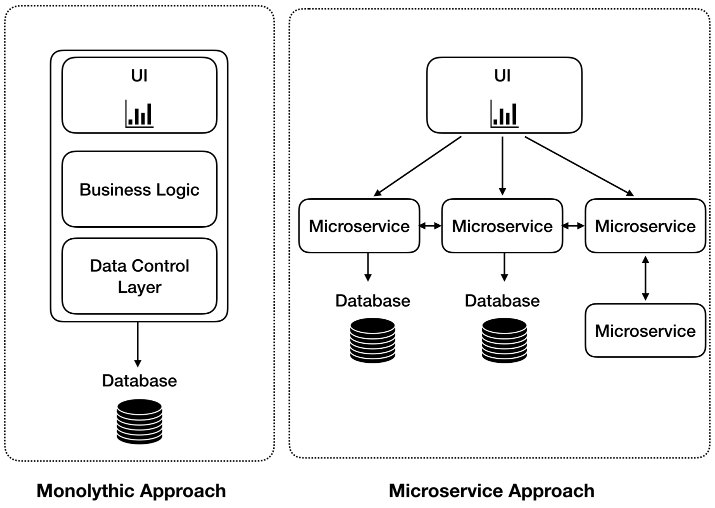


Figure 2.2.: Monolith and microservice architecture

Container-based Virtualization

The container-based virtualization is a lightweight virtualization in contrast to hypervisor-based virtualization and a possibility to realize microservice environments. There is no typical overhead because it is not necessary to install an operating system. Containers are encapsulating an application and its dependencies by using the same operating system and kernel of the host system. The isolation and limitation of resources will be achieved by using Linux kernel features like kernel namespaces and control groups. An example container solution are Linux Containers (LXC) [Mou16]. Docker [Docker] extends this concept with interfaces and the possibility to create portable images for instantiating containers. Images can be created by defining a Dockerfile. This file acts as a blue print for images containing the application and its dependencies. 2.1 illustrates an example of a Dockerfile for building an image with the service of the monitoring concept presented in Chapter 4. To write a Dockerfile, various commands will be used. In regard to the example Dockerfile, the **FROM** command will be used to use *openJDK alpine* as the base image. The **COPY** command will copy the jar file from the source directory to the target file system directory of the container. In contrast **CMD** will be executed during the start of the container to run the jar file. At least **EXPOSE** is necessary to open specific ports to allow communication from the container to the outside.

Listing 2.1 Example Dockerfile

```
FROM openjdk:8-jre-alpine
COPY service.jar /home/service.jar
CMD ["java","-jar","service.jar"]
EXPOSE 3000 8086
```

Besides the Dockerfiles, Docker also provides a platform, which consists of the Docker Engine and the Docker Hub. The Docker Engine is responsible for the creation and execution of containers. The docker-hub is a cloud services with the possibility to share docker images. Figure 2.3 shows the architecture of the container environment. Due to the container architecture, containers can be deployed as independent units. Thus, a microservice-based system can be realized by developing different microservices as well as using container technologies.

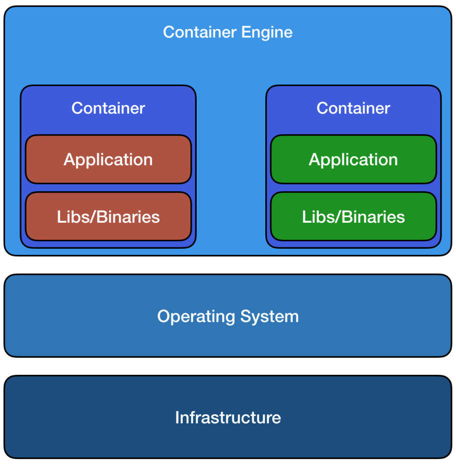


Figure 2.3.: Docker architecture [Mou16]

Container Management System

Container management systems are designed to manage containers at scale and to support the deployment and management of containers and their resource usage in complex distributed systems with multiple physical hosts, such as clouds. Kubernetes [Kubern] is a Container Management System that supports the formation of container clusters by decoupling the containers of an application from the host systems they are running on. The core approach is a shared persistent store, with components watching for changes to relevant objects [BGO+16]. Figure A.1 shows the basic concept of Kubernetes. As seen in the figure, a Kubernetes cluster consist of a master and different nodes, which are different interconnected hosts. Each node consists of different services necessary to run pods while the master consists of components to control the nodes and to schedule pods. Pods are the fundamental unit of a cluster, which are scheduled to the nodes. They are a grouping of one or more containers bounded by a shared context [Ber14]. More precisely, the containers are using a shared storage, network and IP address. To deploy any applications with Kubernetes, a specific YAML file will be used as displayed in 2.2. This file contains all the important information about the replicated pods, the underlying containers and images as well as the open ports.

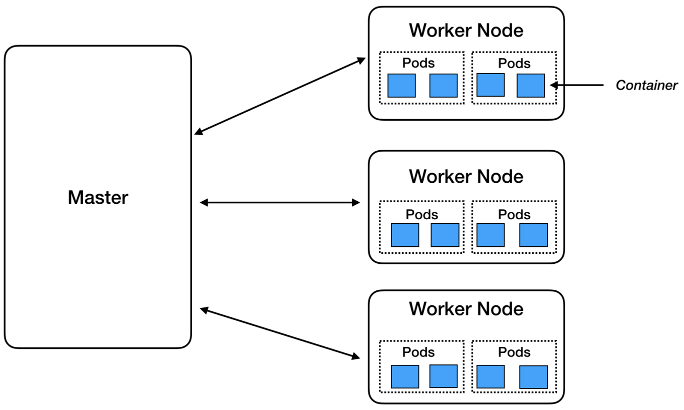


Figure 2.4.: Basic Kubernetes concept

Listing 2.2 Example of a Kubernetes YAML file [KubeDoc]

```
 apiVersion: apps/v1beta2
kind: Deployment
metadata:
  name: nginx-deployment
  labels:
    app: nginx
spec:
  replicas: 3
  selector:
    matchLabels:
      app: nginx
  template:
    metadata:
      labels:
        app: nginx
    spec:
      containers:
      - name: nginx
        image: nginx:1.7.9
        ports:
        - containerPort: 80
```

Application Performance Management

APM is essential for ensuring performance stability and availability of an application. According to Heger et al. [HHMO17], APM is defined as a "*core IT operations discipline, which aims to achieve an adequate level of performance during operation, by using different techniques for continuously monitoring as well as detecting and diagnosing performance-related problems*". To achieve this, commercial APM tools, such as CA Introscope, AppDynamics and Instana are using agents deployed within the application environments to instrument components and to collect metrics and other data. Furthermore, the collected data will be used to analyze the environment behavior and calculate thresholds and baselines. Figure 2.6 shows the basic components of a commercial APM tool. This also includes the important competencies of APM tools, especially the data collection, data storage, data processing and the visualization of metrics [HHMO17]. In regard to the monitoring it is necessary to collect metrics from different components. Table 2.1 display different components and the corresponding metrics.

Component	**Metrics**
Application	error rate, response time, latency,
Middleware	garbage collection, errors, pooling
Hardware	I/O activity, CPU load & usage, Memory usage

Table 2.1.: Example APM metrics

Distributed Tracing & Profiling

Distributed tracing is an approach to monitor the information flow in distributed systems. Due to this approach, different transactions of an application passing through different system components and service can be associated. This will be achieved by propagating metadata through the services when a transaction enters or leaves each service. The propagation takes place by crossing the context of processes and threads. Sigelman et al. [SBB+10] applied distributed tracing by developing the tracing system Dapper, which uses specific identifiers for messages and time stamps for received and sent messages. Therefore, a trace consist of different spans, which are single calls. A unique ID will be set to the initial request and will be propagated to all the corresponding spans. Newman [New15] defines this trace ID as an important identifier to correlate and reconstruct the sub calls and to follow the asynchronous control paths. Furthermore, each span has its own span ID and the parent ID of the parent span. Figure 3.9 shows a trace model with the dependencies including the parent and span IDs. In regard to the approach of Sigelman et al. [SBB+10], each span contains important metadata, especially the start and end time. The system can also handle the control paths within threads by writing the trace metadata into the thread-local storage. Since the tracing of each request would produce too much traffic and a large application overhead, the system only gathers a few traces by using an adaptive sampling rate for high and low traffic services. Also Tracing systems and monitoring tools like Instana [Instana], Zipkin [Zipkin], Jaeger [Jaeger] and Kieker [HWH12] contain this technique.

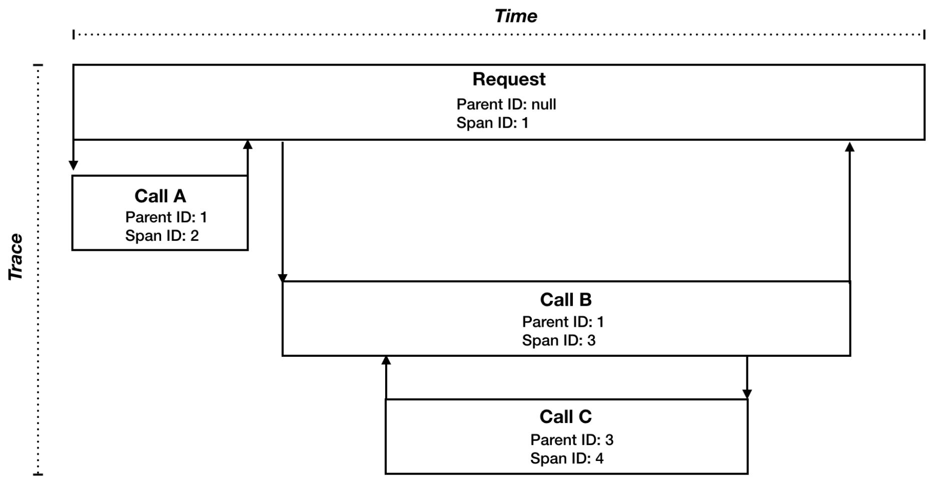


Figure 2.5.: Trace Model

2.2. Challenges

Due to the rise of microservices within IT environments, challenges in regard to APM occur. These challenges will be presented in the following.

2.2.1. Transaction Profiling

Microservice architectures are often highly distributed and complex, which makes the profiling of transactions within these type of environments a difficult task. Traces are often distributed over various services, which leads to a high amount of different callstacks. This leads to new tracing approaches besides the conventional callstack sampling, which is not able to correlate these information to a complete trace. As a result, its not possible to decompose each call and investigate the time spent within each service and across the borders of microservices.

2.2.2. Thresholds & Anomaly Detection

The scalability and the continuous changes of microservice environments can affect the response time of services, which also leads to false anomaly detections. Started and stopped services as well as continuous deployment of updated services are also affecting the metrics. Furthermore, it is difficult to create thresholds, when the load and the response time varies. Figure 1.1 simplifies the challenges in context of scalability and technology variety. An other fact is the dynamic environment and runtime context. Due to container management systems and the paradigm of Continuous Delivery, components are changing their status or will be moved. In some cases conventional APM solutions are detecting moved components as new ones.

2.2.3. Data Collection & Instrumentation

Due to the high amount of technologies in microservice environments, the data collection and gathering is also challenging. Commercial APM tools are often using agents to collect data and to fulfill callstack sampling or Bytecode instrumentation on services. Since each microservice consists of various technologies and components as shown in Table 2.2, it is challenging to instrument the technologies to collect important metrics and information.

Physical Components	Logical & Application Components	Communication
Physical Machines Docker Containers Runtime Processes Clusters	Logical Services Microservices and Endpoints Kubernetes Clusters Middleware Code deployments Configuration changes	Ingoing requests Outgoing requests Service & API endpoints

Table 2.2.: Infrastructure Components

2.2.4. Analytics

According to Heinrich et al.[HHK+17], the behavior of microservice architectures also differs from normal behavior. As a result, it is challenging to create any baselines or knowledge approaches about the behavior of microservice environments.

2.2.5. Metrics

The use of microservice-based environments also leads to changes in regard to APM metrics. Due to the use of new technologies, new metrics exist in regard to containers and container management systems. Furthermore, it is a challenge to define decisive metrics for the determination of the service health. Table 2.3 shows possible metrics for monitoring microservices.

Host	Container	Cluster	Application
CPU usage Memory usage Network connection	Container CPU usage Memory usage I/O activity	Service CPU usage Replicas running Pods running|stop.|termin.	response times trace data & timings error rate latency

Table 2.3.: Infrastructure Components

2.3. Related Work

This section contains related work in regard to the thesis topic. Subsection 2.3.1 presents related work for the design of the industrial case study, while Subsection 2.3.2 and Subsection 2.3.3 present approaches and concepts for the design of the monitoring concept.

2.3.1. APM Case Studies

In regard to the previous work [BSS10], an industrial case study has been conducted to evaluate two different APM tools. To achieve this, different industrial IT environments have been used. The case study was designed to inspect how the monitoring tools are managing the different monitoring activities and disciplines including hardware monitoring, analytics and proactive monitoring. The different environments were based on monolithic approaches without the use of microservices and the underlying technologies, such as Docker and Kubernetes. As a result, it is necessary to design a case study with use cases related to dynamic microservice environments including the new technologies. Some of the use cases can be used in microservice environments, such as conventional hardware monitoring. In contrast, new use cases have to be designed in regard to container monitoring, application discovery and pattern recognition, since the behavior of the new industrial IT environment differs from the behavior of the previous ones. Especially the use cases with regard to baselines and analytics have to be redesigned to see how the new microservice paradigm is affecting the creation of baselines and other analytic approaches. Further, the previous work did not focus on the integration of APM tools into service management softwares and strategies, which is also an important aspect for companies with complex monitoring landscapes.

2.3.2. Microservice Monitoring Concept & Dashboards

Meyer et al. [MW17] are proposing a monitoring concept as well as an experimental dashboard for microservice monitoring and management based on information collected from a conducted survey. The survey consists of different questions about metrics and information related to application and infrastructure component. As a result of the survey, following metrics and information were considered to be important:

- Service API
- Service version
- Number of service instances

- Service metrics (CPU usage, response time, memory usage)
- Service interaction

These metrics will also be used for the monitoring concept presented in Chapter 4. While Meyer et al. are using the REST API of Dynatrace[Dynatrace] for providing service and infrastructure information, the elaborated concept will use Instana including their REST API. Furthermore, they are using different views perspectives for stakeholders, including developers and operators. The elaborated concept in this work will only focus on developer views by combining information and metrics related to the infrastructure and the application. Moreover, Meyer et al. are also using different dashboards to provide runtime information for each service (throughput,response time) as well as information about the service interaction.

2.3.3. Tools for Monitoring Distributed Systems

To monitor application environments and distributed systems, various monitoring stacks are available. The underlying concept of monitoring stacks is the use of different tools to fulfill the following APM activities:

- Collecting metrics
- Storing metrics
- Visualizing metrics

Also the elaborated concept has to fulfill these activities by using a specific monitoring stack with different tools and an implemented service. For this purpose, Kufel [Kuf16] proposes different monitoring tools for monitoring distributed systems. Besides the commercial monitoring solutions, Kufel also presents open-source platforms, such as TIK. TIK is a platform consisting of Telegraf, InfluxDB, Chronograf and Kapacitor [TICK] for collecting, storing and visualizing data as well as alerting on events. In regard to the monitoring concept, Telegraf will be exchanged with Instana and an implemented service, while Grafana will be used instead of Chronograf and Kapacitor.

2.4. Tooling

Besides general approaches and related work, the following tools will be used for the case study and the elaborated concept.

2.4.1. Instana

Instana [Instana] is an APM tool for monitoring microservice-based systems developed by Instana Inc.. It was mainly developed for the purpose of managing dynamic applications in terms of agile development and continuous delivery due to the rise of container orchestration, which leads to constant changes. Therefore, it consists of approaches from Lange et al. [LBNA16] and Baron et al. [BLNA16].

Dynamic Discovery

Lange et al. [LBNA16] are providing an application performance management system with an automatic discovery approach by using deployed agents with sensors on the host system to discover system environments in context of application performance problem detection. The management system architecture including the agents is displayed in Figure 2.6. The agents are executing different discover commands and after receiving the information from the queries they will request sensors from the backend.The agent sensors are able to pull metrics from discovered components or to hook itself into the components and send information back to the agent. With this function it is possible to load a native Java agent into the Java Virtual Machine (JVM) and to fulfill bytecode instrumentation on detected Java applications. The data transfer from agents to the backend will be done trough an communication interface. The discovery process consists of discovering processes, software and hardware of the host machine. After detecting the software components on the target machine the sensors will be deployed to interfaces with the relying components like Java applications, webservers or databases. To identify components on a host machine Lange et al. [LBNA16] are using fingerprints or signatures. The unique fingerprints will be created because of the recognition of executed processes, applications, containers and network location bounded to a component. A host system can be uniquely identified by using the UNIX name, results of sysctl commands, listening network ports or processes running on the machine. Furthermore, the system can also create a grouping ID for clusters of machines. Moreover, changes in the infrastructure will also be reported from the sensors. In regard to the changes, already identified components that have changed their status will be recognized again by using a threshold of similarity and the defined parameters for the fingerprint. With this threshold it is also possible to avoid a detection of re-instantiated or moved components as a new component.

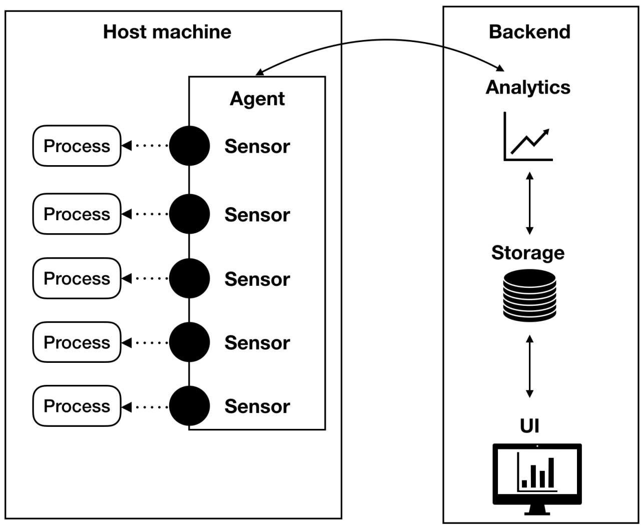


Figure 2.6.: Application performance management system [LBNA16]

Collective Learning & Knowledge Engine

After discovering the environment with the system, Baron et al. [BLNA16] are using a collective learning approach to identify the behavior of the target environment. To achieve this, the system receives first and second metrics of different components from different agents deployed on different target systems. Both components are part of the same component class. Due to these metrics, the system is able to calculate a performance pattern for this class of components. In combination with an algorithm the calculated pattern will be used to detect performance problems for the specific component class. As part of this approach the system receives again metrics from the first component and applies the algorithm modified with the calculated pattern to detect performance problems of the first component. Figure 2.7 shows a part the collective learning process. With this approach Baron at al. are able to identify the status of a component and making predictions about incoming issues. By combining the dynamic discovery approach with their developed knowledge engine including a dynamic graph function, they are able to describe dependencies between the components. Due to this combination it is possible to see how issues are affecting other components. Moreover, the approach also includes fix suggestion for each occurring issue.

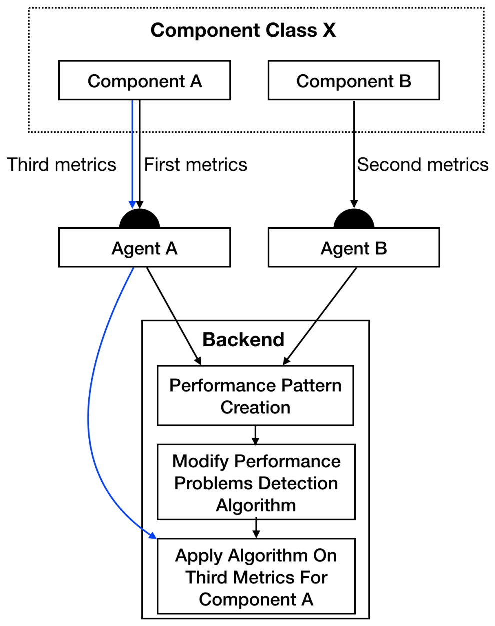
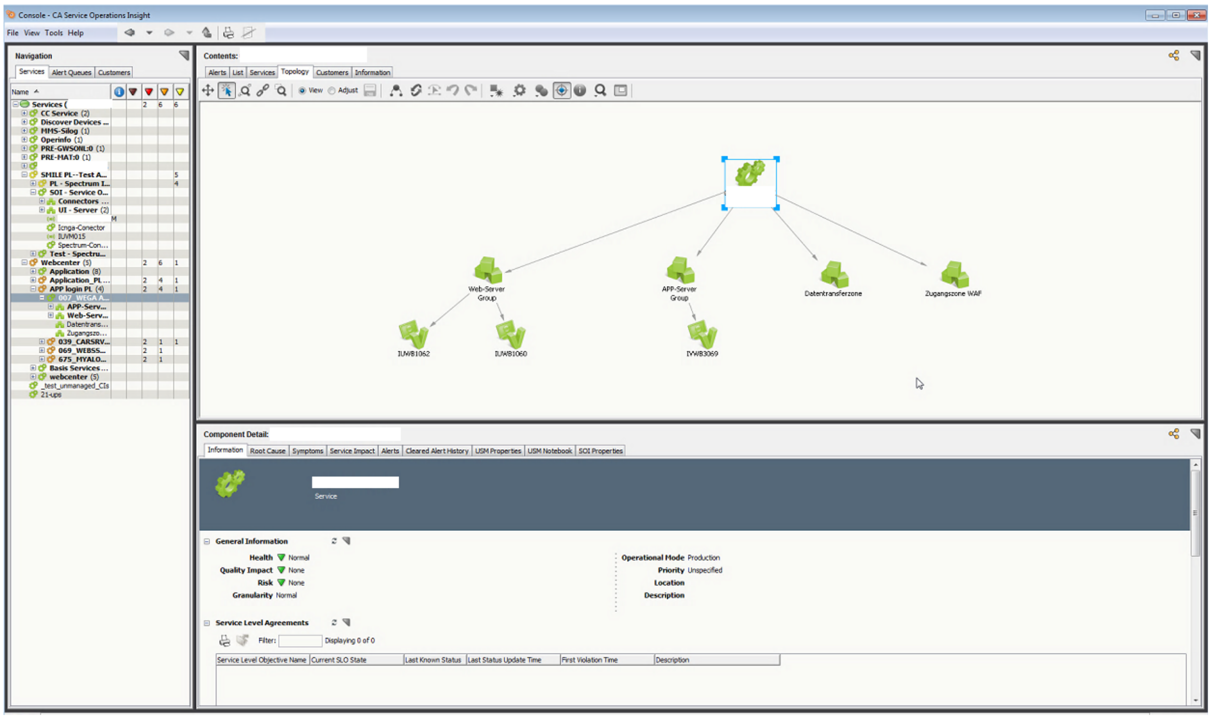

Component Class X
Component A
Component B
Third metrics
First metrics
Second metrics
Agent A
Agent B
Backend
Performance Pattern Creation
Modify Performance Problems Detection Algorithm
Apply Algorithm On Third Metrics For Component A

Figure 2.7.: Collective learning process [BLNA16]

2.4.2. CA Service Operation Insight

SOI is a service management software for managing business services of IT environments. It collects the data about application performance, infrastructure and other IT information from monitoring tools for analyzing and visualizing the business services. With the use of this information, a dynamic model will be created including application performance, alerts, availability and quality of the services. SOI also provides a connector interface for integrating third party monitoring solutions, such as AppDynamics [AppDyn]. The connector accelerator architecture supports the sending of alerts from various alarm sources to SOI via REST calls. As a result, each third party monitoring solution has its own connector implementation. Figure 2.8 and Figure 2.9 are displaying the alert and service dashboards.



Component Detail: [redacted]

Information | Root Cause | Symptoms | Service Impact | Alerts | Cleared Alert History | USM Properties | USM Notebook | SOI Properties

[redacted]
Service

General Information

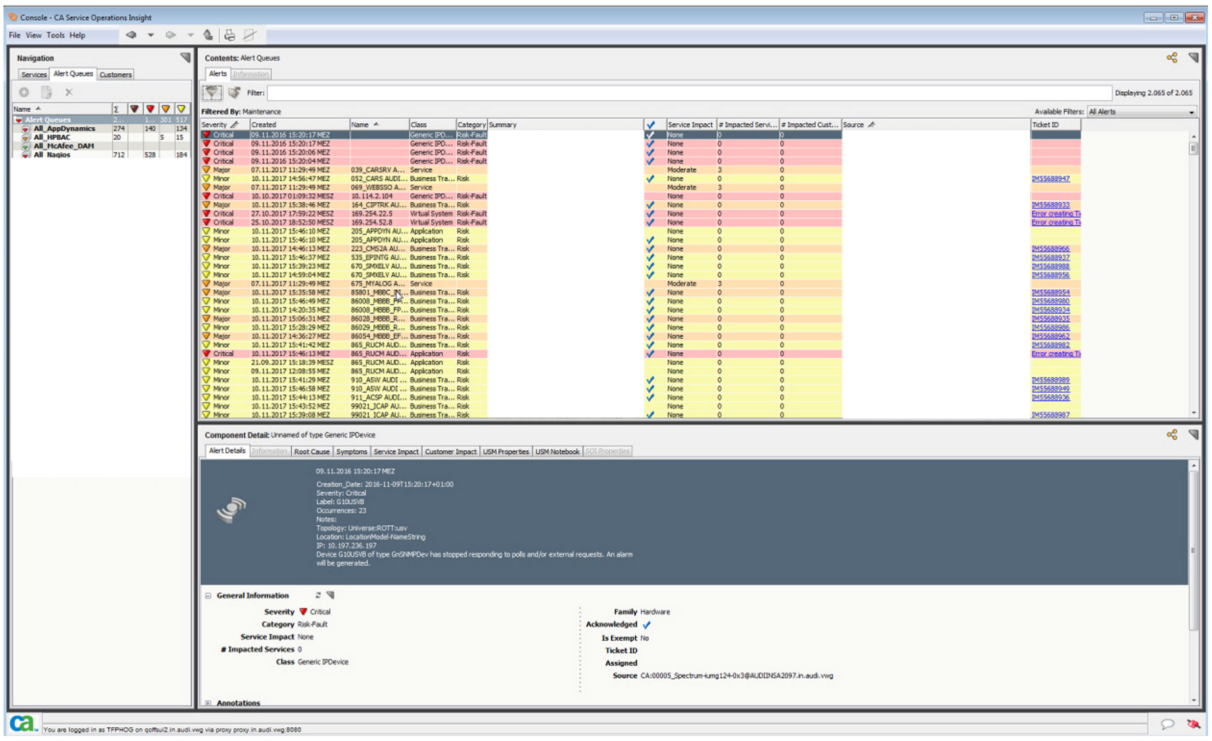
Health	Normal	Operational Mode	Production
Quality Impact	None	Priority	Unspecified
Risk	None	Location	
Granularity	Normal	Description	

Service Level Agreements

Filter: Displaying 0 of 0

Service Level Objective Name	Current SLO State	Last Known Status	Last Status Update Time	First Violation Time	Description

Figure 2.8.: Service map



Contents: Alert Queues

Alerts | Information

Filter: Displaying 2.065 of 2.065

Filtered By: Maintenance

Available Filters: All Alerts

Severity	Created	Name	Class	Category	Summary	✓	Service Impact	# Impacted Servi...	# Impacted Cust...	Source	Ticket ID
Critical	09.11.2016 15:20:17 MEZ		Generic IPD...	Risk-Fault		✓	None	0	0		
Critical	09.11.2016 15:20:17 MEZ		Generic IPD...	Risk-Fault		✓	None	0	0		
Critical	09.11.2016 15:20:06 MEZ		Generic IPD...	Risk-Fault		✓	None	0	0		
Critical	09.11.2016 15:20:04 MEZ		Generic IPD...	Risk-Fault		✓	None	0	0		
Major	07.11.2017 11:29:49 MEZ	039_CARSRV A...	Service				Moderate	3	0		
Minor	10.11.2017 14:56:47 MEZ	052_CARS AUDI...	Business Tra...	Risk		✓	None	0	0		IMS5688947
Major	07.11.2017 11:29:49 MEZ	069_WEBSSO A...	Service				Moderate	3	0		
Critical	10.10.2017 01:09:32 MESZ	10.114.2.104	Generic IPD...	Risk-Fault			None	0	0		
Major	10.11.2017 15:38:46 MEZ	164_CIPTRK AU...	Business Tra...	Risk		✓	None	0	0		IMS5688933
Critical	27.10.2017 17:59:22 MESZ	169.254.22.5	Virtual System	Risk-Fault		✓	None	0	0		Error creating Ti
Critical	25.10.2017 18:52:50 MESZ	169.254.52.8	Virtual System	Risk-Fault		✓	None	0	0		Error creating Ti
Minor	10.11.2017 15:46:10 MEZ	205_APPDYN AU...	Application	Risk			None	0	0		
Minor	10.11.2017 15:46:10 MEZ	205_APPDYN AU...	Application	Risk		✓	None	0	0		
Major	10.11.2017 14:46:13 MEZ	223_CMS2A AU...	Business Tra...	Risk		✓	None	0	0		IMS5688966
Minor	10.11.2017 15:46:37 MEZ	535_EPINTG AU...	Business Tra...	Risk		✓	None	0	0		IMS5688937
Minor	10.11.2017 15:39:23 MEZ	670_SMXELV AU...	Business Tra...	Risk		✓	None	0	0		IMS5688988
Minor	10.11.2017 14:59:04 MEZ	670_SMXELV AU...	Business Tra...	Risk		✓	None	0	0		IMS5688956
Major	07.11.2017 11:29:49 MEZ	675_MYALOG A...	Service				Moderate	3	0		
Major	10.11.2017 15:35:58 MEZ	85801_MBBC_IN...	Business Tra...	Risk		✓	None	0	0		IMS5688954
Minor	10.11.2017 15:46:49 MEZ	86008_MBBB_FP...	Business Tra...	Risk		✓	None	0	0		IMS5688980
Minor	10.11.2017 14:20:35 MEZ	86008_MBBB_FP...	Business Tra...	Risk		✓	None	0	0		IMS5688934
Major	10.11.2017 15:06:31 MEZ	86028_MBBB_R...	Business Tra...	Risk		✓	None	0	0		IMS5688935
Minor	10.11.2017 15:28:29 MEZ	86029_MBBB_R...	Business Tra...	Risk		✓	None	0	0		IMS5688986
Major	10.11.2017 14:36:27 MEZ	86054_MBBB_EF...	Business Tra...	Risk		✓	None	0	0		IMS5688962
Minor	10.11.2017 15:41:42 MEZ	865_RUCM AUD...	Business Tra...	Risk		✓	None	0	0		IMS5688982
Critical	10.11.2017 15:46:13 MEZ	865_RUCM AUD...	Application	Risk		✓	None	0	0		Error creating Ti
Minor	21.09.2017 15:18:39 MESZ	865_RUCM AUD...	Application	Risk			None	0	0		
Minor	09.11.2017 12:08:55 MEZ	865_RUCM AUD...	Application	Risk			None	0	0		
Minor	10.11.2017 15:41:29 MEZ	910_ASW AUDI ...	Business Tra...	Risk		✓	None	0	0		IMS5688989
Minor	10.11.2017 15:46:58 MEZ	910_ASW AUDI ...	Business Tra...	Risk		✓	None	0	0		IMS5688949
Minor	10.11.2017 15:44:13 MEZ	911_ACSP AUDI...	Business Tra...	Risk		✓	None	0	0		IMS5688936
Minor	10.11.2017 15:43:52 MEZ	99021_ICAP AU...	Business Tra...	Risk			None	0	0		
Minor	10.11.2017 15:39:08 MEZ	99021_ICAP AU...	Business Tra...	Risk		✓	None	0	0		IMS5688987

Component Detail: Unnamed of type Generic IPDevice

Alert Details | Information | Root Cause | Symptoms | Service Impact | Customer Impact | USM Properties | USM Notebook | SOI Properties

09.11.2016 15:20:17 MEZ

Creation_Date: 2016-11-09T15:20:17+01:00
Severity: Critical
Label: G10USVB
Occurrences: 23
Notes:
Topology: Universe:ROTT:usv
Location: LocationModel-NameString
IP: 10.197.236.197
Device G10USVB of type GnSNMPDev has stopped responding to polls and/or external requests. An alarm will be generated.

General Information

Severity	Critical	Family	Hardware
Category	Risk-Fault	Acknowledged	✓
Service Impact	None	Is Exempt	No
# Impacted Services	0	Ticket ID	
Class	Generic IPDevice	Assigned	
		Source	CA:00005_Spectrum-iumg124-0x3@AUDIINSA2097.in.audi.vwg

Annotations

You are logged in as TFPHOG on qoftsui2.in.audi.vwg via proxy proxy.in.audi.vwg:8080

Figure 2.9.: Alert panel

2.4.3. InfluxDB

InfluxDB [InfluxDB] is an open-source time series database for storing time series data. The database is commonly used within monitoring stacks by storing performance metrics. Therefore, InfluxDB provides an HTTP API for writing and reading data.

2.4.4. Grafana

Grafana [Grafana] is an open-source tool to visualize metrics and monitoring data. It supports the creation of specific Dashboards. In addition, it also supports the integration of different databases, such as InfluxDB.

2.4.5. K6

K6 [K6] is an open-source load testing tool written in go language. The main concept of the tool are virtual users (VUs), which are running scripts. The Scripts will be written in JavaScript. Furthermore, they are containing HTTP requests used for testing the system. K6 also provides a specific InfluxDB plugin to write metric data to an InfluxDB database. In regard to the metrics, the tool provides custom metrics and built-in metrics, such as HTTP-specific metrics(TTP request send, HTTP request received). In addition, checks can be fulfilled in regard to the HTTP status, transaction time or response.

Chapter 3

Case Study

3.1. Goals

The goal of the case study is to work out how Instana is overcoming the different use cases in regard to microservice monitoring. Therefore it is necessary to document the underlying approaches, techniques and metrics to fulfill APM of microservice-based systems.

3.2. Methodology

The use cases will be designed by considering the previous industrial case study mentioned in subsection 2.3.1. Further, the use cases will be modified by considering the elaborated challenges of microservice monitoring and the technologies used in microservice-based systems mentioned in Section 2.2 and Section 2.1. The case study will be executed on the industrial IT environment provided by the industrial partner. Therefore the agents of the commercial monitoring tool will be deployed with a specific YAML file. Moreover, the backend of the monitoring tool will be installed as an onPremise installation on a separated Amazon Elastic Compute Cloud (Amazon EC2) environment by using an AWS account of the industrial partner. During the conduct of the case study, the K6 [K6] open source load testing tool will be used to create any load on the environment. Therefore, a specific JavaScript file will be written to execute the K6 tests. An excerpt of the script file with the definition of an HTTP request and HTTP check can be seen in Listing 3.1. The result metrics of each load test will be stored within the time series database InfluxDB and visualized by the metrics visualization tool Grafana [Grafana]. Therefore, an individual dashboard will be designed to visualize the K6 build-in metrics and checks as shown in Figure 3.1. The dashboard and the metrics will be used to inspect if a load test will trigger any errors. As a result, Instana will be used to inspect the triggered requests within the application and to find the root cause of the issue. Furthermore, the monitoring tool will be used to investigate how it fulfills the overall monitoring of the application performance affected by the load tests. In addition, the monitoring tool will be connected to SOI to integrate it into the SMILE program. Therefore a specific generic connector will be implemented as part of a specific use case. Besides the practical proving, the case study will also be conducted with the support of documentations as well as interviews with the solutions engineer of Instana Inc. For this purpose Table 3.1 displays the used approaches and the related use cases.

Listing 3.1 Excerpt of the K6 test script

```
group("Pair User", function() {
     let result;
     result = http.put("https://"+host_url+"/ilfcpe/cpe/iccpd/"+serialNumber+"/user",
           "{    \"ssoid\":\"W759890571\", \"origin\":\"myCar\"}",
           { headers: { "Authorization" : "Bearer "+access_token+"", "Content-Type" :
                "application/json" } });

     check(res, {
           "response code was 200": (result) => result.status == 200,
           "transaction time OK": (result) => result.timings.duration < TRANSACTION_TIME,
           "response is not empty": (result) => result.body.length >= 0,
     });
  });
```

	Interview	**Practical Proving**	**Documentations**
APM-UC-01	✓		✓
APM-UC-02		✓	
APM-UC-03		✓	
APM-UC-04		✓	
APM-UC-05	✓	✓	✓
APM-UC-06		✓	✓
APM-UC-07		✓	✓
APM-UC-08	✓		
APM-UC-09		✓	✓
APM-UC-10		✓	✓

Table 3.1.: Approaches of executing the use cases

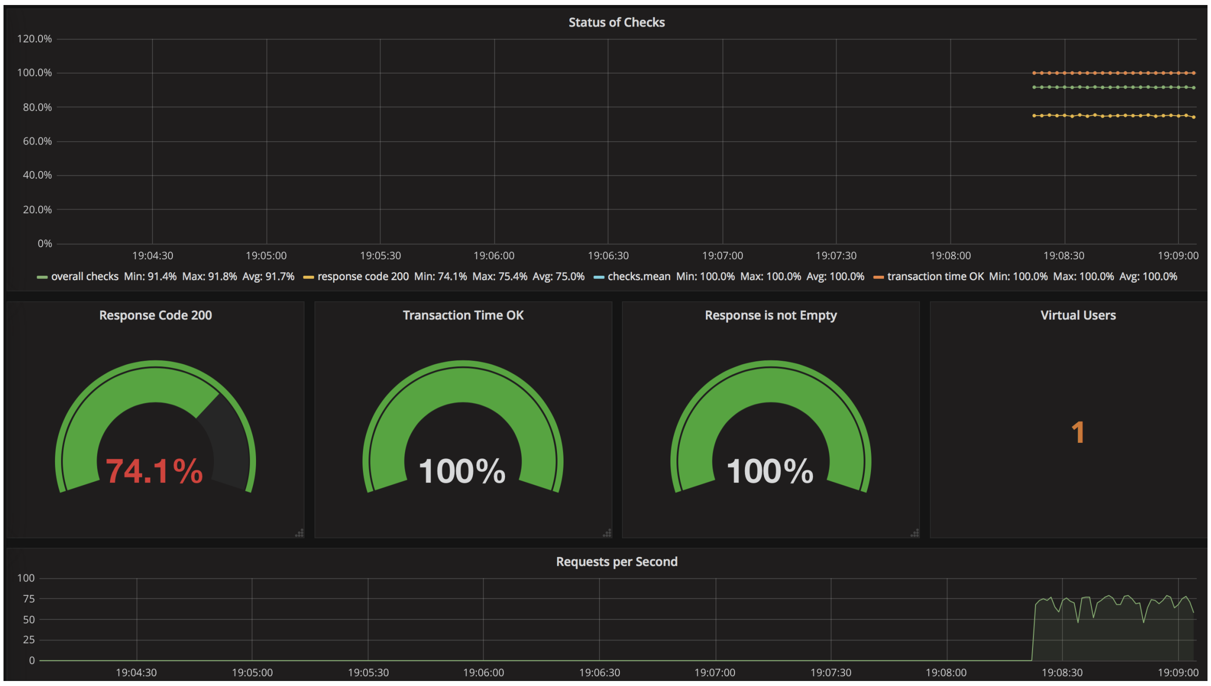


Figure 3.1.: Grafana dashboard for the k6 test results

3.3. Case Study Systems

3.3.1. Industrial IT Environment

The industrial IT environment is a project of application services for loading processes of electric cars by using a provided loading cable. This includes the authentication as well as the calculation of the power consumption and the loading state. The services are developed with SpringBoot [SpringBoot] and PostgreSQL databases [PostgreSQL] by using a microservice-based approach. The infrastructure consists of a Kubernetes platform and Docker containers. The Kubernetes platform is running on infrastructure resources provided by the cloud-hosting-service Amazon EC2 of Amazon Web Services (AWS) Moreover, ElasticSearch, fluentd and Kibana are used for the log management. In addition, the current monitoring will be fulfilled with the use of Grafana, Heapster und InfluxDB. The provisioning of the infrastructure will be achieved by using Terraform [Terra], which provides an infrastructure-as-code approach.

3.3.2. SMILE Strategy

The SMILE strategy of the industrial partner is a monitoring and alert management strategy consisting of different monitoring tools, SOI and the Service Center. As shown in Figure 3.2, every monitoring tool is connected to SOI to send events and informations while SOI fulfills the alert assessment and management to send tickets to the Service Center. The monitoring tools will be connected to SOI with an implementation of a specific connectors.

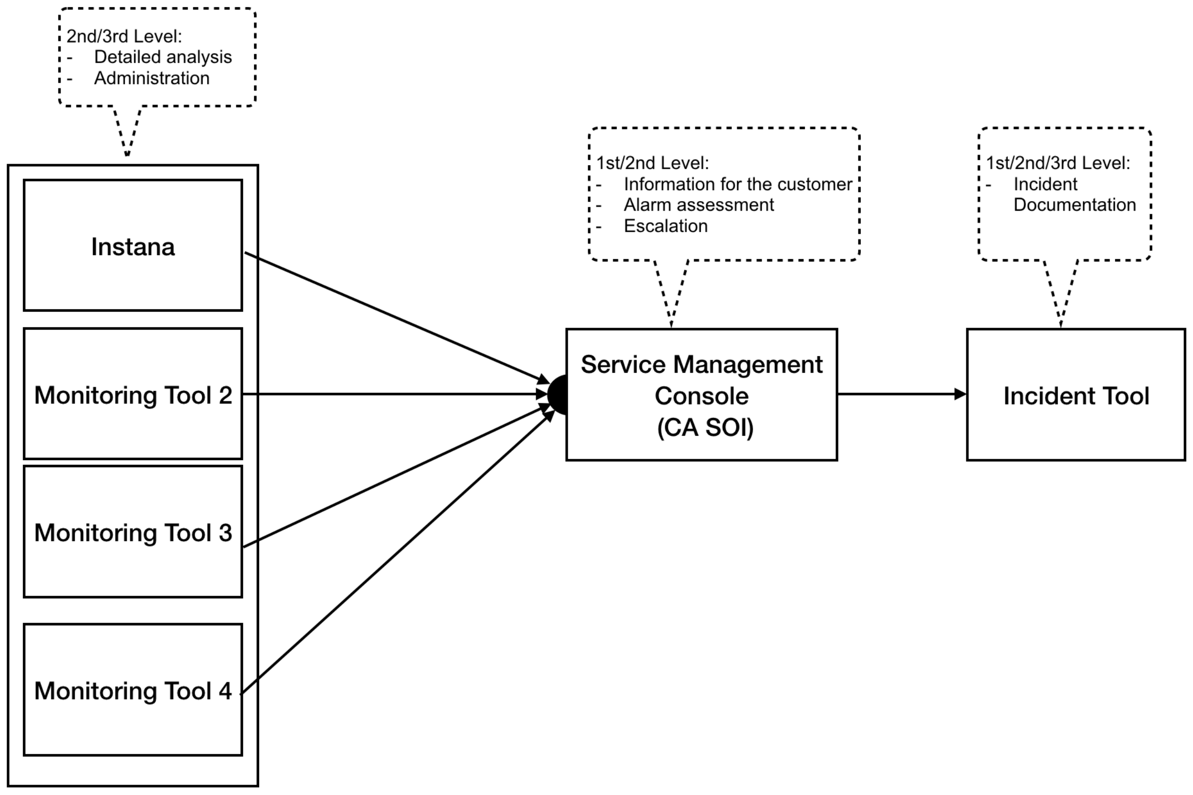


Figure 3.2.: SMILE architecture: from events to incidents

3.4. Use-Cases

This section proposes the use cases for the case study. Each use case has been designed to evaluate the commercial monitoring tool in respect to the thesis topic.

3.4.1. APM-UC-01 Agent Setup & Data Collection approaches

With this use case, the effort and configuration of the agents as well as the whole installation process will be evaluated:

- Methods of installation (container-based, host-based etc.)
- Requirements
- Approaches to collect metrics (Bytecode Instrumentation, Callstack Sampling)

3.4.2. APM-UC-02 Infrastructure Discovery

Cloud-environments consist of different layers and distributed components. In regard to this architectural style the main evaluation criteria of this use case will be:

- Detection & view of clusters
- Detection & view of infrastructure and layers
- Detection & view of applications

3.4.3. APM-UC-03 Container & Hardware Monitoring

Since microservice-based systems consist of different isolated and independent containers, it is necessary to collect informations about container metrics and statistics. Therefore, the following criteria needs to be considered:

- Usage
- Workload
- Availability

In regard to the health status and other available resources like CPU throttling, Memory, I/O etc. following criteria will be used:

- Available container metrics

3.4.4. APM-UC-04 Visualization

The visualization is one important feature of monitoring tools.The microservice and cloud monitoring differs from monitoring of monolithic systems also in case of visualization. Therefore, it is important to investigate how the collected data will be displayed. This use case will take a closer look on the configuration and creation of dashboards and views. This is exemplified by the following evaluation criteria:

- Design of the user interface
- Availability and configuration of dashboards, views and graphs
- Visualization of different drill downs
- Clarity
- Flexibility
- Intuitiveness

3.4.5. APM-UC-05 Alerting

In this use case the possibility of "pro-active" monitoring will be considered. Automated reactions of well-defined performance problems as well as creating a prediction of potential problem situations are central. Therefore, the following evaluation criteria need to be considered:

- Use of thresholds and baselines
- Alert types

3.4.6. APM-UC-06 Problem Detection & Root Cause Analysis

The priority in this use case will be the error detection, diagnosis and analytic mechanisms. The possibility to identify the root cause of already recognized performance problems and tracing will be considered. This leads to the following evaluation criteria:

- Identification of
    - Slow services
    - Slow methods

  - Critical Database accesses
- Available metrics
- Filter possibilities
- Tracing
- Microservice Monitoring
- Error information
- Analytic techniques and approaches
  - Trends (e.g daily, after release / deployment etc.)
  - Log Analytics
  - Pattern recognition
  - Other techniques and approaches

3.4.7. APM-UC-07 Integration

The integration in components and systems of existing IT services and monitoring landscapes will be considered in this use case. For this purpose the following evaluation criteria needs to be considered:

- Access to the collected data (e.g. queries, REST API)
- Integration possibilities

3.4.8. APM-UC-08 Scalability

In this use case the scalability of the monitoring concept will be vetted. Therefore, the following evaluation criteria need to be considered:

- Number of Agents
- Used Techniques like controllers, collecting units or hubs
- SaaS / OnPremise

3.4.9. APM-UC-09 Multi-Access Capability

Since large enterprises consist of different compartments and jurisdiction it is necessary to provide different views relating to the usage of controllers. In regard to this aspect the evaluation criteria consists of:

- Role systems (different developer and monitoring staff etc.)
- Compartments permissions and restrictions

3.4.10. APM-UC-10 Integration at the industry partner

This use case presents one of the main aspects for this work. Since the previous use case was focusing on integration possibilities, the following use case will set the main focus on how the monitoring solution can be practically integrated into the monitoring landscape of the industry partner including the SMILE program.

- Integration into IT landscape of industry partner
- Integration into CA SOI
- Observance of IT security requirements
  - Protection of log messages
  - Data security

3.5. Evaluation & Results

This section contains the results of the use cases.

3.5.1. APM-UC-01 Agent Setup & Data Collection Approaches

Depending on the operating systems, the Instana agent can be installed by using the following possibilities:

- Deployment to a Kubernetes cluster
- Run as a privileged Docker container
- Install as a Linux or Microsoft Windows service

The deployment to a Kubernetes cluster will be done by using a provided YAML file. With this file all necessary Kubernetes objects will be deployed into the cluster and the Instana-agent namespace. This also includes a daemonset which automatically sets up agents on every newly created node. This ensures that the agent deployment has to be done only once. Due to the support of technologies like Docker and Kubernetes, Instana agents do not need to be deployed as conventional agents running as a service or a single program on the operating systems. In regard to the agent architecture, Instana uses a single lightweight agent approach. This leads to the deployment of only one agent on every host machine supporting various languages and technologies. Every agent is able to detect the technologies running inside the environment. Moreover the agent is connected to an agent repository to fulfill any deployments of sensors after detecting the technologies of the environment or to update the sensors after new releases. As a result, updates due to new releases can be done without restarting the JVM or redeploying the agents. By using these sensors, Instana can fulfill any Bytecode Instrumentation or infrastructure state tracking to report on incidents or changes. Figure 3.3 shows the whole agent setup including the repository.

For the observation and data collection which is done by the agents, Instana mainly uses bytecode instrumentation [InstanaInst]. Callstack Sampling will only be done once to get an inisght of all the invoked methods. To fulfill the bytecode instrumentation the Java Instrumentation API will be used. With this API Instana is able to hook into the class loading process and manipulate bytecode before it will be executed. To read and write bytecode, Instana uses the Byte Buddy bytecode generation library [ByteB]. With this library Java classes can be written or modified during the runtime of a Java application. The instrumentation will only be done at the Java class library and known container as well as framework levels. 3.2 shows an code example of an Instana Agent which intercepts http requests of HttpServlets. With the *@Advice.OnMethodEnter* and *@Advice.OnMethodExit* annotations the methods enter and exit will be called before and after executing the original method. Due to this technique Instana can read and write informations like span IDs or trace IDs from or into HTTP headers to fulfill distributed tracing as mentioned in Section 2.1.2.

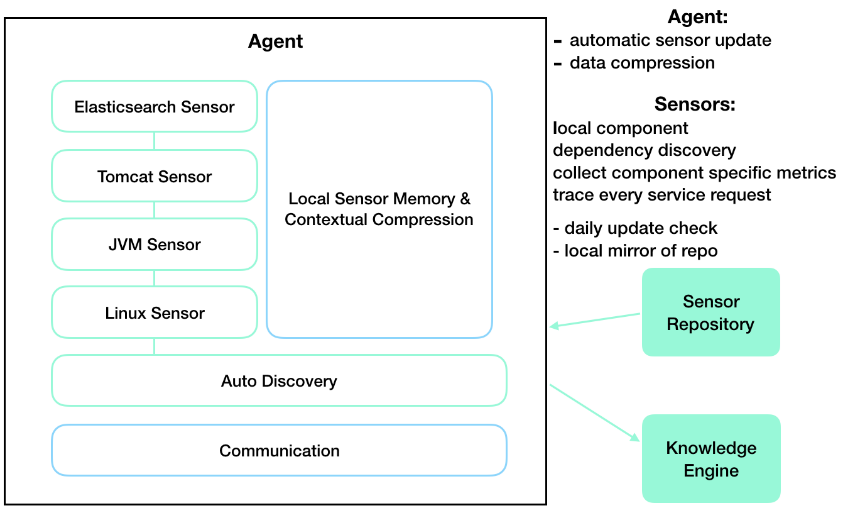


Figure 3.3.: Agent overview [InstanaDoc]

Listing 3.2 Instana bytecode instrumentation

```
public class ServerInstrumentation{
      public AgentBuilder instrument(AgentBuilder agentBuilder) {
      return AdviceRegistry.subTypesOf(HttpServlet.class).advice((method) -> {
            ParameterList<InDefinedShape> parameters = method.getParameters();
         return method.getInternalName().equals("service)
            && parameters.size() == 2
            && parameters.get(0).getType().isAssignableTo(HttpServletRequest.class)
            && parameters.get(1).getType().isAssignableTo(HttpServletResponse.class);
           }
      }, ServletAdvice.class).register(agentBuilder);
   }

   private static class ServletAdvice {
     @Advice.OnMethodEnter
     private static void enter(@Advice.Argument(0) HttpServletRequest request, @Tag int tag) {
       String method = request.getMethod();
       String requestUri = request.getRequestURI();
       String traceId = request.getHeader(TracingHeaders.TRACE_ID);
       Callbacks.find(ServletInstrumentation.class).call(1, method, requestUri, traceId);
   }
     @Advice.OnMethodExit
     private static void exit(@Advice.Argument(1) HttpServletResponse response) {
       Callbacks.find(ServletInstrumentation.class).call(2, response);
   }
  }
}
```

3.5.2. APM-UC-02 Infrastructure Discovery

Since infrastructure discovery is necessary to get an overview of all the components of the IT environment, Instana auto detects the architecture of complex applications and the infrastructure of the environment. Also changes in the environment will be detected by using a developed real-time knowledge engine and dynamic discovery approach that automatically discovers application topology and the interdependencies of components as mentioned in Section 2.4.1. Figure 3.4 displays an excerpt of the infrastructure view with the discovered components. With the information collected from sensors and traces, Instana also automatically discovers the implemented services and endpoints. Furthermore the connections between the services and the single requests flowing through the application will be detected. In general Instana distinguishes discovered components between:

- Physical components (e.g. containers, hosts)
- Logical components (e.g services, applications)
- Business components (e.g business services, business process)

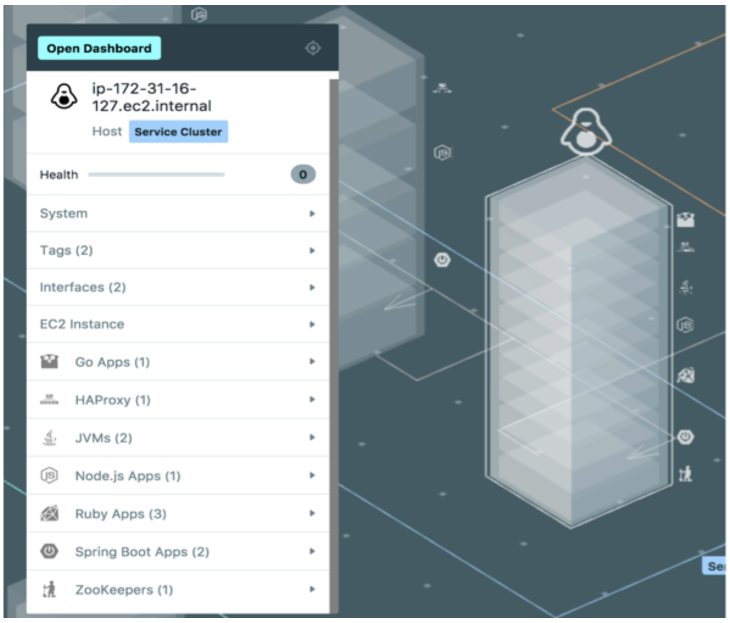


Figure 3.4.: Instana infrastructure discovery

3.5.3. APM-UC-03 Container & Hardware Monitoring

After detecting hardware and containers as part of the infrastructure, Instana collects different metrics about these components. In regard to Docker Containers, the following metrics will be collected:

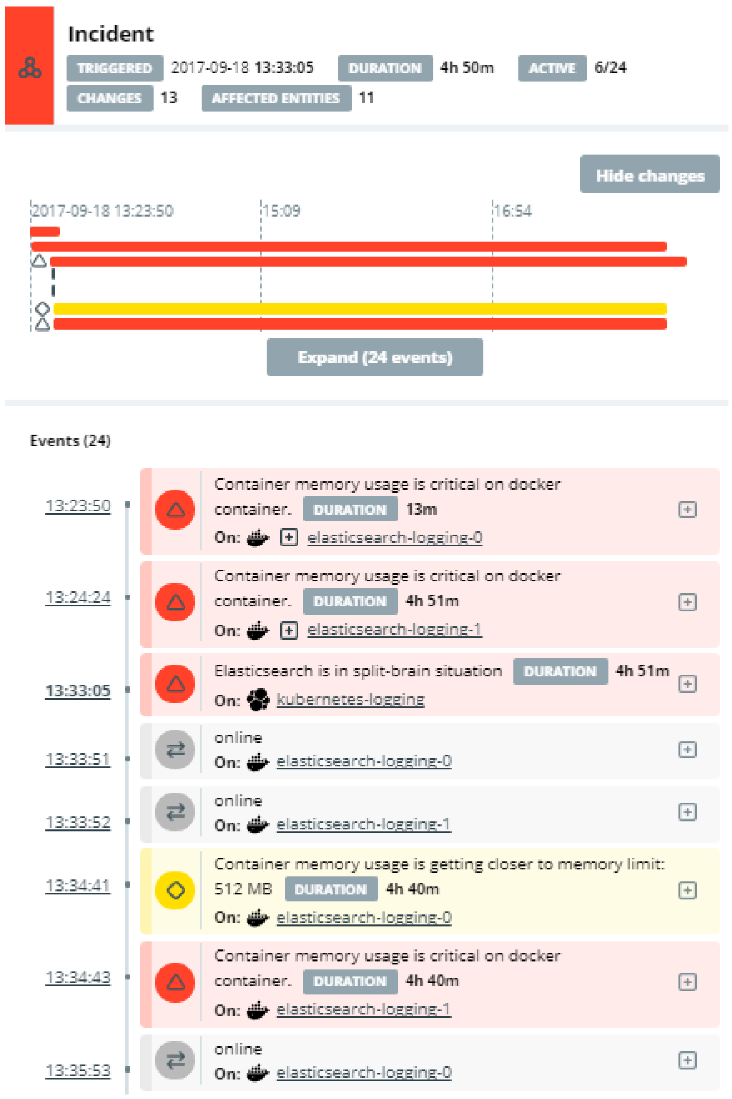
- CPU usage (total, kernel, user, throttling)
- Memory (usage, rss, cache)
- Block IO (read, write)

The metrics for host monitoring are:

- CPU (load, user, system, wait, nice, steal)
- Memory (used, max usage, RSS, cache)
- Network - TCP Activity (fails, errors, established, retransmission)

Regarding Docker containers, the monitoring solutions informs when critical changes of the metrics occur as shown in Figure 3.5. Moreover the monitoring solution gives also information about the content of the Docker containers (image) and the date when the container was created and started. Commonly, host and container monitoring are part of the Instana infrastructure monitoring. In regard to this infrastructure, monitoring metrics will also be collected about:

- Processes
- JVMs
- Application services
- Agents



Incident

TRIGGERED 2017-09-18 13:33:05 DURATION 4h 50m ACTIVE 6/24

CHANGES 13 AFFECTED ENTITIES 11

Hide changes

2017-09-18 13:23:50 15:09 16:54

Expand (24 events)

Events (24)

13:23:50 Container memory usage is critical on docker container. DURATION 13m
On: elasticsearch-logging-0

13:24:24 Container memory usage is critical on docker container. DURATION 4h 51m
On: elasticsearch-logging-1

13:33:05 Elasticsearch is in split-brain situation DURATION 4h 51m
On: kubernetes-logging

13:33:51 online
On: elasticsearch-logging-0

13:33:52 online
On: elasticsearch-logging-1

13:34:41 Container memory usage is getting closer to memory limit: 512 MB DURATION 4h 40m
On: elasticsearch-logging-0

13:34:43 Container memory usage is critical on docker container. DURATION 4h 40m
On: elasticsearch-logging-1

13:35:53 online
On: elasticsearch-logging-0

Figure 3.5.: Container issues & incidents

3.5.4. APM-UC-04 Visualization

The visualization of the microservice-based system will be achieved with the use of two different view perspectives and categories. Therefore Instana offers an infrastructure and application view. Due to a timeline which contains the marks for changes, issues and incidents, it is possible to switch between dates. The application view contains a map, a trace table with a graph of each trace and a comparison table to display each provided service and the corresponding metrics like error rate latency and calls. This view allows a deep view and a drill down into the application context and offers high granularity of each service. In contrast the infrastructure view contains a map and a comparison table to illustrate the infrastructure components like container, hosts or JVMs. Due to the splitting of views, Instana enables a simple and comprehensible way of displaying the whole environment including the possibility to enter into the application context. Figure 3.6 and Figure 3.7 are displaying the different maps of the environment. The infrastructure map, which is also the starting point for every deep-dive and root-cause analysis, displays a map as a coherent overview of the IT environment and the components. The map can be adjusted by grouping the nodes and components with attributes like availability zone, host or Docker images. All components of a node are visible as a stacked tower where each stage is a single component. Also the perspective can be changed from a host specific view to a container specific view. If an issue occurred, it will be recognizably displayed as shown in Figure 3.8. By contrast, the application map illustrates the different services and endpoints. It also clarifies their connection and communication.

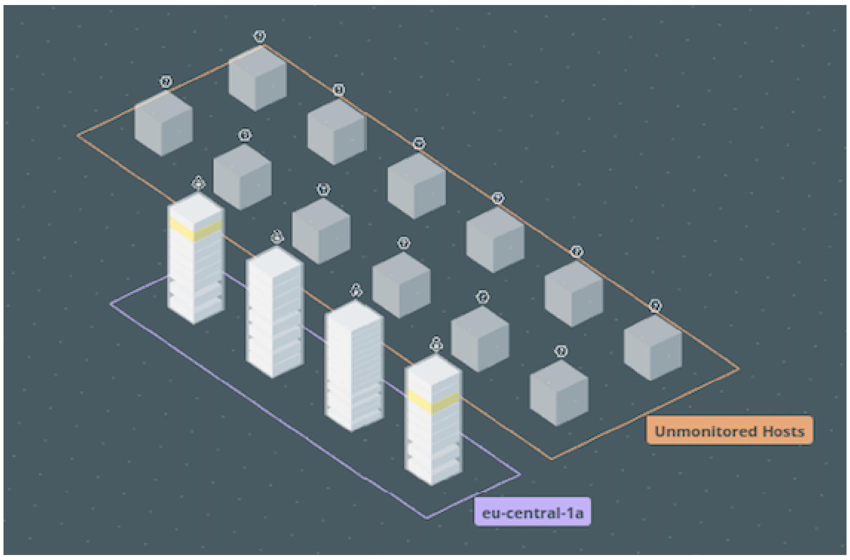


Figure 3.6.: Infrastructure view

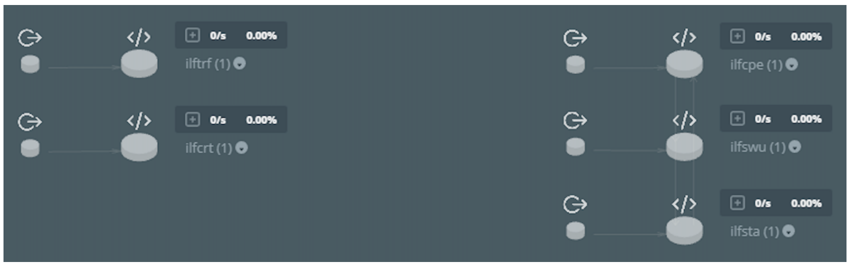


Figure 3.7.: Application view

3.5.5. APM-UC-05 Alerting

To alert on specific events Instana differentiates between the following three different event types:

- Incidents
- Issues
- Changes

By differentiating between these types, Instana can separate noises from important issues. In general, incidents are critical events directly impacting the services of the environment. To alert on this events, the dynamic graph and different Key Performance Indicator (KPI) will be used. KPIs are decisive metrics to inspect the health of a application service. In contrast, an issue is an event that changes the health of the components without directly impacting the services. This event can be used for incident prediction. Issues can be detected by using machine learning algorithms and health signatures. Figure 3.8 shows a detected issue. However changes are normal events and part of the microservice environment behavior. These changes can be events of components like containers which are going offline or online.

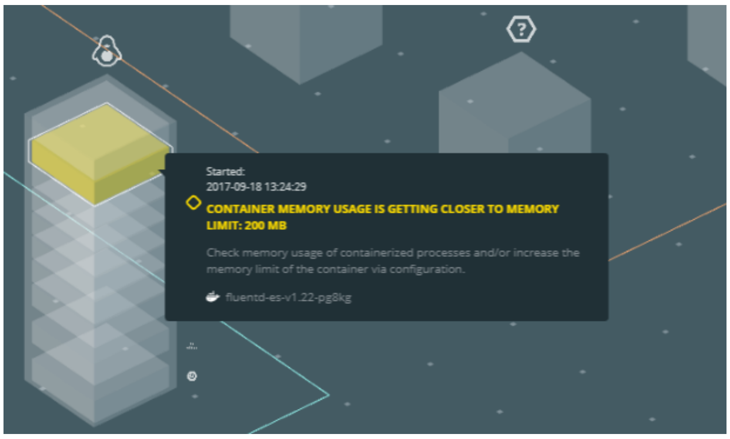

Started:
2017-09-18 13:24:29
CONTAINER MEMORY USAGE IS GETTING CLOSER TO MEMORY LIMIT: 200 MB
Check memory usage of containerized processes and/or increase the memory limit of the container via configuration.
fluentd-es-v1.22-pg8kg

Figure 3.8.: Detected issue

Besides the predefined health rules it is also possible to create custom rules with conditions. The categories for the custom rules are:

- entity
- metric
- time window

- aggregation
- operator
- value

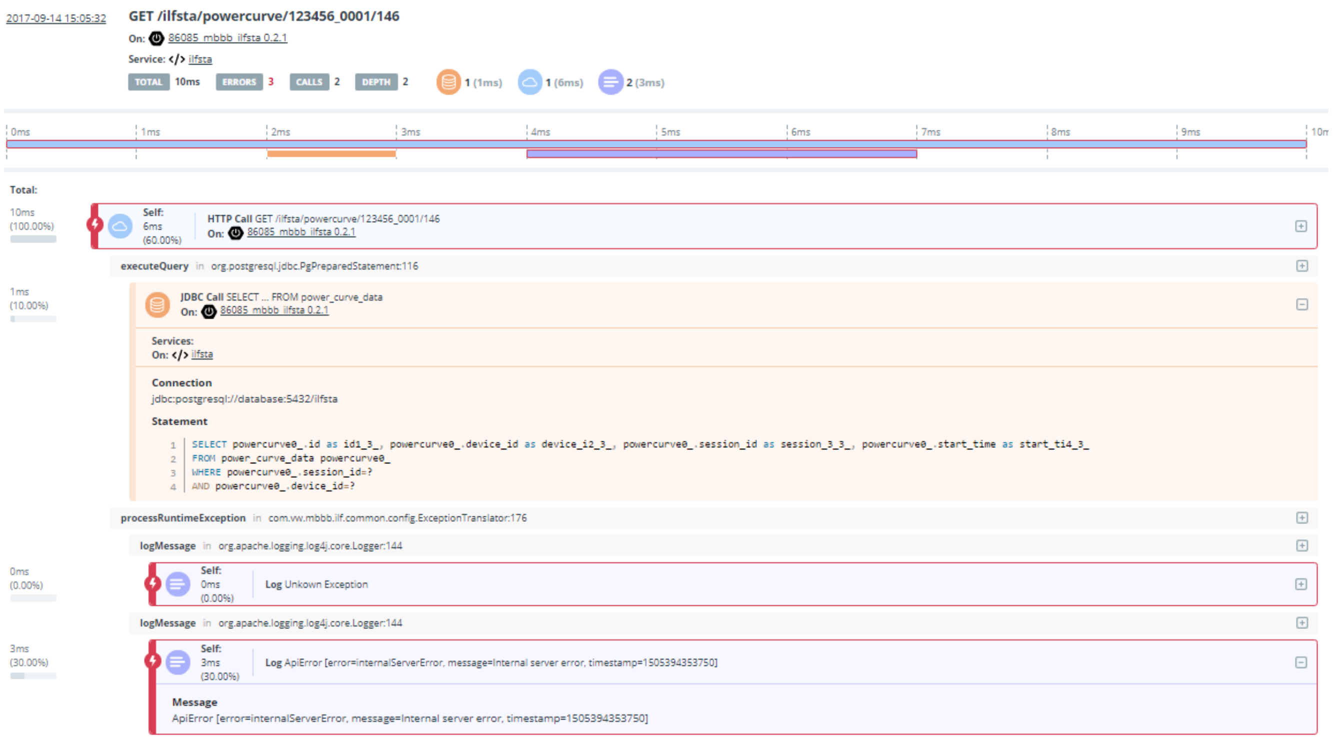
In this case entity refers to the select component where the rules should be applied. After creating the rules, it possible to enable or disable them.

3.5.6. APM-UC-06 Problem Detection & Root Cause Analysis

For any root cause analysis in complex microservice environments, Instana focuses on the components that are affected by an issue as well as correlating crucial metrics and data to detect the actual root cause behind an incident. Therefore, Instana uses a technology called Dynamic Graph [DynamicGraph]. This technology combines physical and logical dependencies of components as model of the whole application. This also includes hosts and interconnected services of the application. Each node is a part of the application with important application and infrastructure metrics as well es informations stored. Due to different algorithms and by walking trough the Dynamic Graph model, Instana is able to calculate the specific impact of changes and issues on the application or service. After analyzing the changes and issues, Instana correlates them to an incident if the service performance is affected. The service performance itself will be measured with the following important KPI[InstanaDoc]:

- Load
- Latency
- Errors
- Saturation
- Instances

The information to create and update this model will be collected by the agents and sensors. The creation of the dynamic graph and the detection of root causes will be supported by using distributed tracing. Here a Dapper-inspired approach will be used as mentioned in Section 2.1.2. By using this concept, Instana is able to resolve the interaction between the services and components. It also supports a deep view into the trace hierarchy by using a trace view to display the whole trace as a graph. Figure 3.9 shows an example of a trace of a request triggered by a load test shown in Figure 3.1. It also shows detected code sections like SQL statements, thrown exceptions or messages. Furthermore, it can be recognized which part of the request ends with an error. With the help of this approach, Instana is able to calculate the impact of changes and issues on the application or service. It is not possible to export any information of the dynamic graph.


2017-09-14 15:05:32
GET /ilfsta/powercurve/123456_0001/146
On: 86085_mbbb_ilfsta 0.2.1
Service: </> ilfsta
TOTAL 10ms ERRORS 3 CALLS 2 DEPTH 2 1 (1ms) 1 (6ms) 2 (3ms)

Total:

10ms (100.00%) — Self: 6ms (60.00%) — HTTP Call GET /ilfsta/powercurve/123456_0001/146
On: 86085_mbbb_ilfsta 0.2.1

executeQuery in org.postgresql.jdbc.PgPreparedStatement:116

1ms (10.00%) — JDBC Call SELECT ... FROM power_curve_data
On: 86085_mbbb_ilfsta 0.2.1

Services:
On: </> ilfsta

Connection
jdbc:postgresql://database:5432/ilfsta

Statement

```
SELECT powercurve0_.id as id1_3_, powercurve0_.device_id as device_i2_3_, powercurve0_.session_id as session_3_3_, powercurve0_.start_time as start_ti4_3_
FROM power_curve_data powercurve0_
WHERE powercurve0_.session_id=?
AND powercurve0_.device_id=?
```

processRuntimeException in com.vw.mbbb.ilf.common.config.ExceptionTranslator:176

logMessage in org.apache.logging.log4j.core.Logger:144

0ms (0.00%) — Self: 0ms (0.00%) — Log Unkown Exception

logMessage in org.apache.logging.log4j.core.Logger:144

3ms (30.00%) — Self: 3ms (30.00%) — Log ApiError [error=internalServerError, message=Internal server error, timestamp=1505394353750]

Message
ApiError [error=internalServerError, message=Internal server error, timestamp=1505394353750]

Figure 3.9.: Trace view

3.5.7. APM-UC-07 Integration

To automatically integrate Instana into other management processes, the following possibilities are offered:

- Email
- Webhook
- OpsGenie
- Pagerduty
- Slack
- Office 365

In regard to webhook integration, alert information will be sent in JavaScript Object Notation (JSON) format with the following metrics:

- Event type
- Event status
- ID
- Start time
- Severity
- Event text
- Host FDQN
- Host tags

3.5.8. Instana REST API

The Instana REST API can be used to extract important information from the backend by using various HTTP GET requests. The data will be sent by using HTTP POST requests. To use the API, it is necessary to create an API token and to assign access permissions. With the REST API it is possible to collect following data:

- Events
- Infrastructure and application view
- Traces
- Metrics about different components and services
- Rules

Besides this data, it is also possible to register, invite or remove users by using the REST API. Listing 3.4 displays an excerpt of an HTTP response with trace information. The listing shows, that the HTTP response contains various information about the trace. This also includes childSpans and the stacktraces. The REST call for the HTTP response and a call for receiving CPU metrics can be seen in Listing 3.3. The API token has to be added to the HTTP *Authorization* request header for each call.

Listing 3.3 Example REST calls

```
https://industrial-partner.instana.io/api/traces/-1827782056356300223

https://industrial-partner.io/api/metric?metric=cpu.used&time=1511851225863
&aggregation=mean&snapshotId=R6x_nuhhLZq1wrr9X7aVH1N0nCs&rollup=1000
```

Listing 3.4 HTTP response with trace information

```
{
  "traceId":"-1827782056356300223",
  "spanId":"-1827782056356300223",
  "name":"spring-web",
  "start":1511561782924,
  "duration":2,
  "batchSize":0,
  "data":{
    "http":{
      "rawUrl":"/health",
      "path":"/health",
      "method":"GET",
      "host":"172.16.78.12:80",
      "status":"200"
    }
  },
  "errorCount":0,
  "totalErrorCount":0,
  "childSpans":[
    {
      "spanId":"-3992774284853197077",
      "parentId":"-1827782056356300223",
      "name":"mongo",
      "start":1511561782925,
      "duration":0,
      "batchSize":1,
      "data":{
        "mongo":{
          "protocol":"com.mongodb.connection.CommandProtocol",
          "service":"orders-db:27017",
          "namespace":"data.$cmd",
          "json":"{ \"buildInfo\" : 1 }",
          "command":"command"
        }
      },
      "errorCount":0,
      "childSpans":[
      ],
      "stackTrace":[
        {
          "method":"execute",
          "class":"com.mongodb.connection.CommandProtocol"
        }
      ]
    }
  ],
  "stackTrace":[
    {
      "method":"doService",
      "class":"org.springframework.web.servlet.DispatcherServlet"
    },
  ]
}
```

3.5.9. APM-UC-08 Scalability

In regard to the scalability there are no limitations for the amount of agents. Statistics about the use of Instana in massive production systems have clarified that Instana can still handle metrics and instant analytics from about 7.000 containers and about 15.000 entities with their specific instant analytics approach. Moreover each agents is directly connected to the backend without using any controller units as a subcomponent between agents and backend. The SaaS environment can handle any scalable systems due to the direct adjustment of resources for the backend during operation. In contrast the OnPremise solution has to be reinitialized due the required hardware resources when systems are scaling massive.

3.5.10. APM-UC-09 Multi-Access Capability

To allow different access to the backend, Instana uses an access control system with different roles. For this purpose the users are:

- Owner
- Service Mapper

To grant users access to the backend, an email invitation will be sent. It is not possible to directly create a user with a username and password. Moreover, an email server has to be integrated to send the invitation emails. Only the owner has access to all configurations while the service mapper can obtain different permissions. Instana does not offer the classification of user groups but each user can be grouped with the different permissions.

3.5.11. APM-UC-10 Integration at the industry partner

Integration into the IT landscape of the industry partner & IT security

The integration of Instana into the existing IT landscape of the industry partner can be achieved by installing the backend as an On Premise installation either on an instance in the AWS cloud or a physical machine running inside the B2X environment. Due to the security of the collected data it is recommended to install the backend inside the B2X environment at the industry partner. The Instana backend server is the central component and is responsible for processing and storing of the collected informations. Therefore, the components of the backend platform are using remote software repositories to install and update packages. The requirements including the communication of the components can be seen in Figure 3.10. For the monitoring of microservice environments it is necessary to provide a connection from the host machines through port 1444 to the backend. This connection will be used to send data from the agent to the backend. Furthermore, the Instana User Interface can be reached through port 443. By installing the agents on microservice-based systems which are hosted at the industry partner, it is recommendable to mirror the agent repository locally. This leads to an Agent Lifecycle Management without establishing a connection to the outside of the industry partner zones. If the agents are deployed at a provided cloud platform, a direct connection to the agent repository can be established. Due to the connection between agents running on an AWS EC2 cloud platform and the backend running inside the Application Monitoring Zone at the B2X environment, it is necessary to provide a direct and secure communication to the outside of the zones of the industry partner. To solve this challenge, the AWS Direct Connection can be used. In general the AWS Direct Connection supports a dedicated network connection from local datacenters or server environments to public AWS EC2 cloud or an Amazon Virtual Private Cloud (VPC). As a result this connection can be used to send data from the agents to the backend inside the Application Monitoring Zone. By establishing such a connection it is important to allow fast data transfer since Instana Agents are using a streaming approach to send informations every second to the backend. In this case data buffering is challenging. In regard to the IT security of the industry partner, it is recommended to install the Instana backend in the Monitoring & Management are as displayed in Figure 3.12.

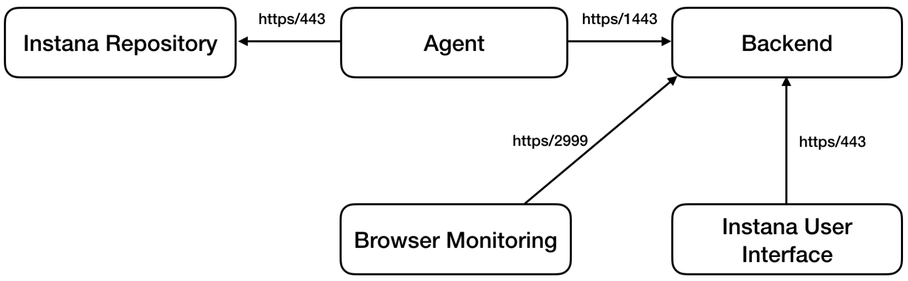


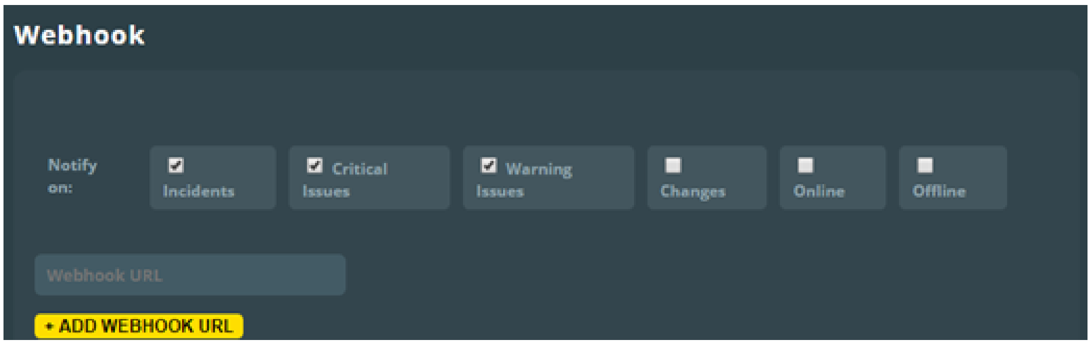
Figure 3.10.: Instana data model and network requirements

Integration into SMILE

To integrate Instana into the SMILE strategy, the alerts have to be integrated into the service management tool CA SOI. For this purpose, the use of the Instana Webhook is recommended. Furthermore, a generic controller has to be developed as an interface between Instana and SOI. Every alert will be pushed to the generic controller via an

HTTP POST request. The controller will convert the information from JSON to SOI valid attributes and values and will send it to the SOI as a HTTP GET request. The information categories of an alert sent via the Instana Webhook are mentioned in subsection 3.5.5. In contrast SOI offers the following attributes as an alert integration:

- alarmsource
- hostname
- severity
- alertkey
- message
- summary


Webhook

Notify on: [✓ Incidents] [✓ Critical Issues] [✓ Warning Issues] Changes Online Offline

Webhook URL

Figure 3.11.: Webhook settings

In regard to the alert levels, Instana uses numbers while SOI uses words as severities. Table 3.2 shows the severities of Instana and the corresponding SOI severities. It is recommended to only alert on issues and incidents since changes do not affect the health of a system. Furthermore a marker should be set on this type of events in the webhook settings as shown in Figure 3.11.

Instana	**SOI**
-1 (changes)	Minor
5 (issues)	Major
10 (incidents)	Critical

Table 3.2.: Instana & SOI severities

The generic controller can be a program running on a physical machine which can be reached with an URL from the Instana backend. Figures 3.5 and 3.6 are illustrating two different implementations of a generic controller. The first one is a generic controller implemented with a socket which listens on a specific port while the second one is a Java Servlet which can be run on a Tomcat server. Both are transforming the incoming JSON name-value pairs in to a similar HTTP query with name-value pairs. To do this both are using Googles GSON library. Furthermore, the Instana severities have to be mapped to similar SOI severities.

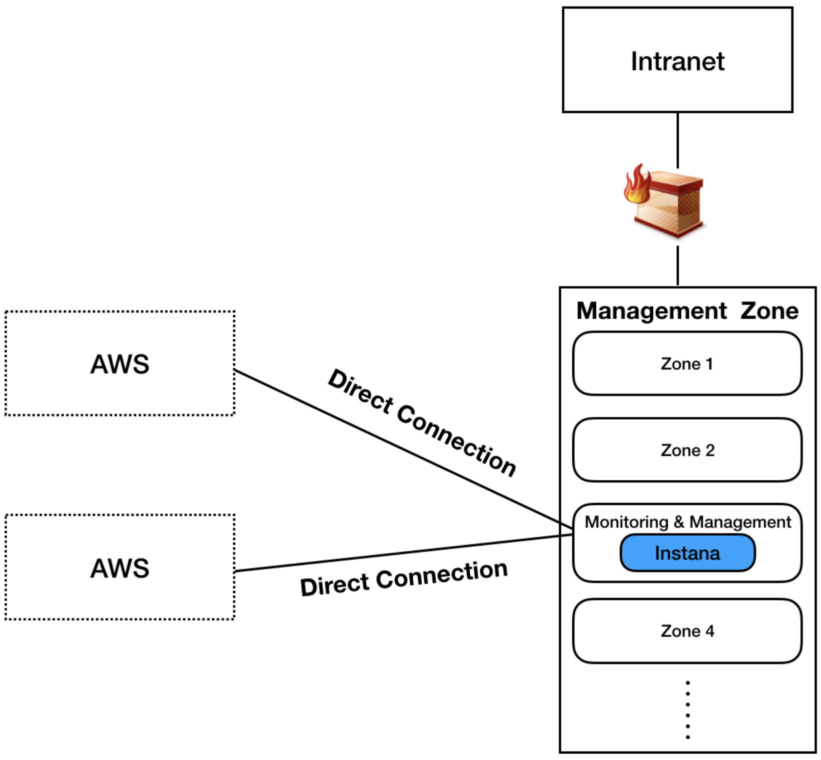


Figure 3.12.: Instana at the industrial partner

Listing 3.5 Generic connector as a socket

```
while (true) {
 Socket clientSocket = serverSocket.accept();
 BufferedReader br = new BufferedReader(new InputStreamReader(clientSocket.getInputStream()));
 boolean headersFinished = false;
 int contentLength = -1;

 while (!headersFinished) {
      String line = br.readLine();
      headersFinished = line.isEmpty();
        if (line.startsWith("Content-Length:")) {
                String cl = line.substring("Content-Length:".length()).trim();
                contentLength = Integer.parseInt(cl);
              }
        }
 //read instana webhook content
 char[] buf = new char[contentLength];
 br.read(buf);
 Gson gson = new Gson();
 InstanaIssue instanaIssue = gson.fromJson(String.valueOf(buf),InstanaIssue.class);

 if(instanaIssue.getIssue().getSeverity().equals("-1")){
             instanaIssue.getIssue().setSeverity("Minor");
 }else if (instanaIssue.getIssue().getSeverity().equals("5")) {
             instanaIssue.getIssue().setSeverity("Major");
 }else{
             instanaIssue.getIssue().setSeverity("Critical");
 }
 instanaIssue.getIssue().setText(instanaIssue.getIssue().getText().replaceAll(" ", "_"));
 instanaIssue.getIssue().setSuggestion(instanaIssue.getIssue().getSuggestion().replaceAll(" ",
      "_"));
 URIBuilder builder = new URIBuilder();
  if (instanaIssue.getIssue().getState().equals("OPEN")) {
      builder.setScheme("http").setHost("industrypartner.example.ipadress:7771")
   .setPath("/postAlert")
      .setParameter("alarmsource", "ILF").setParameter("summary", instanaIssue.getIssue().getText())
      .setParameter("hostname", "AWS_Cloud").setParameter("serverity",
          instanaIssue.getIssue().getSeverity())
      .setParameter("alertkey", instanaIssue.getIssue().getId())
      .setParameter("message", instanaIssue.getIssue().getSuggestion());
  } else {
      builder.setScheme("http").setHost("industrypartner.example.ipadress:7771")
   .setPath("/postAlert")
      .setParameter("alarmsource", "ILF").setParameter("summary", instanaIssue.getIssue().getText())
      .setParameter("hostname", "AWS_Cloud").setParameter("serverity", "Normal")
      .setParameter("alertkey", instanaIssue.getIssue().getId())
      .setParameter("message", instanaIssue.getIssue().getSuggestion());
  }
 //send request to SOI
 HttpClient httpClient = HttpClientBuilder.create().build();
 HttpGet soiRequest = new HttpGet(builder.toString());
 HttpResponse soiResponse = httpClient.execute(soiRequest);
 br.close();
 clientSocket.close();
}
```

Listing 3.6 Generic connector as a servlet

```java
@WebServlet("/InstanaSOIConnector")
public class InstanaSOIConnector extends HttpServlet {
 private static final long serialVersionUID = 1L;

 protected void doPost(HttpServletRequest request, HttpServletResponse response)
 throws ServletException, IOException {

 // Read json from webhook request
 String webhookContent = request.getReader().lines().collect(Collectors.joining());

 // convert json(webhook content) to java object
 Gson jsonObject = new Gson();
 InstanaIssue instanaIssue = jsonObject.fromJson(webhookContent, InstanaIssue.class);

 // map instana severity(-1,5,10) to SOI severity(Minor, Major, Critical)
 if (instanaIssue.getIssue().getSeverity().equals("-1")) {
      instanaIssue.getIssue().setSeverity("Minor");
 } else if (instanaIssue.getIssue().getSeverity().equals("5")) {
      instanaIssue.getIssue().setSeverity("Major");
 } else {
      instanaIssue.getIssue().setSeverity("Critical");
 }

 instanaIssue.getIssue().setText(instanaIssue.getIssue().getText().replaceAll(" ", "_"));
 instanaIssue.getIssue().setSuggestion(instanaIssue.getIssue().getSuggestion().replaceAll(" ",
      "_"));
 URIBuilder builder = new URIBuilder();

 // create or clear alert (=>Normal)
 if (instanaIssue.getIssue().getState().equals("OPEN")) {
      builder.setScheme("http").setHost("industrypartner.example.ipadress:7771")
   .setPath("/postAlert")
      .setParameter("alarmsource", "ILF").setParameter("summary", instanaIssue.getIssue().getText())
      .setParameter("hostname", "AWS_Cloud")
      .setParameter("serverity", instanaIssue.getIssue().getSeverity())
      .setParameter("alertkey", instanaIssue.getIssue().getId())
      .setParameter("message", instanaIssue.getIssue().getSuggestion());
 } else {
      builder.setScheme("http").setHost("industrypartner.example.ipadress:7771")
   .setPath("/postAlert")
      .setParameter("alarmsource", "ILF").setParameter("summary", instanaIssue.getIssue().getText())
      .setParameter("hostname", "AWS_Cloud").setParameter("serverity", "Normal")
      .setParameter("alertkey", instanaIssue.getIssue().getId())
      .setParameter("message", instanaIssue.getIssue().getSuggestion());
 }

 // send request to SOI
 HttpClient httpClient = HttpClientBuilder.create().build();
 HttpGet soiRequest = new HttpGet(builder.toString());
 HttpResponse soiResponse = httpClient.execute(soiRequest);
}
```

Chapter 4

The Monitoring Concept

This chapter presents an experimental concept for monitoring microservice-based systems by using different tools, metrics and dashboards as well as the Instana REST API mentioned in Section 3.5.8. The concept will be elaborated due to various information collected from the industrial case study and related work.

4.1. Goals

The main goal is the design of the concept by selecting different tools in combination with the monitoring tool Instana. Furthermore, it can be inspected how different metrics and information can be pulled from the Instana backend trough the REST API in combination. In addition, the visualization of the metrics and information with Grafana, as well as the design of automation processes to update the dashboards will be additional goals.

4.2. Concept Setup

Since the concept consists of various tools to achieve the basic APM activities, the whole setup will be presented. This also includes the test environment, where the concept will be applied on.

4.2.1. Concept Architecture

The architecture of the experimental monitoring concept is inspired by different monitoring stacks and concepts to fulfill the basic APM activities of collecting, storing and visualizing metrics. For this purpose, the concept consists of the Instana SaaS environment, including the REST API as well as InfluxDB, Grafana and an implemented service which creates an XML file to store component data. As shown in Figure 4.1, the implemented service requests the current component data trough the Instana REST API. To achieve this, an API token with specific permissions has been generated in the user settings of the SaaS environment. The collected data will be used for different views. Moreover, the service also gets the previous component data from a created XML file. The snapshotId of both component data will be compared to identify changes within the environment. If any changes occured, the Grafana dashboards will be updated by sending a modified HTTP POST request with JSON data to Grafana after requesting the current dashboards. After that, the XML file will be updated with the current component data. Next, the service requests metrics from the Instana backend through the REST API. After receiving the metrics from the Instana backend, the service processes the metrics and sends it to InfluxDB through an HTTP API by using a provided Java library [InfluxDB-java]. Furthermore, Grafana is configured to access the metrics stored within InfluxDB for each dashboard. To access data through the Instana REST API, a token will be generated in the Instana settings. This token will be sent with each REST call for authentication purpose. Another token will be generated in the Grafana settings to access the dashboards. This token will be sent with request too. The requests to get the Grafana dashboards as well as the Instana metrics are HTTP GET requests. In regard to Instana, the architecture and the components have been presented in Subsection 3.5.11 and basic information about the REST API has been presented Subsection 3.5.8.

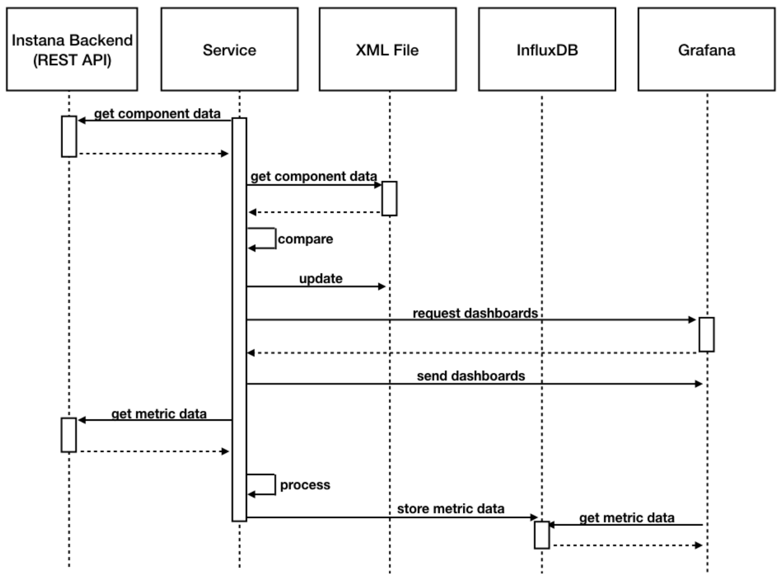


Figure 4.1.: Sequence diagram of the monitoring concept

4.2.2. Test environment

The concept will be applied on the microservice demo application Sock-Shop [Sock-Shop] deployed on the CASPA platform. The CASPA platform is a ready-to-use and extensible evaluation platform which consists of example applications and software performance analytics components [DHHP17]. Furthermore the components can be exchanged due to specific interfaces. Figure 4.2 shows the architecture of the CASPA platform including the single components. Hardware resources will be provided by an underlying OpenStack environment. The infrastructure layer is based on a Kubernetes cluster and Docker containers. The necessary Instana agents will also be deployed with the use of a YAML file.

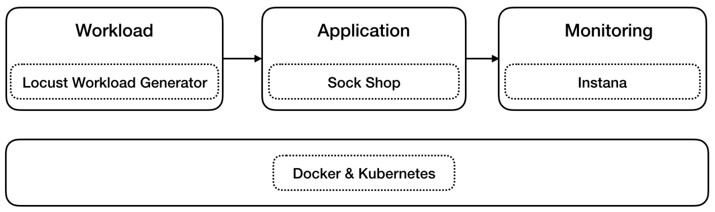


Figure 4.2.: CASPA architectural layer with components [DHHP17]

4.3. Infrastructure Dashboard

The infrastructure dashboards will be used for infrastructure monitoring. This includes the detection of hardware and infrastructure issues. Therefore, the infrastructure dashboard provides metrics and views about the whole infrastructure.

4.3.1. Infrastructure Views

To get an overview of the infrastructure, the dashboard consists of different view panels for displaying hosts and processes. Figure 4.3 displays the host view of the infrastructure. It can be seen that the infrastructure consists of the two main hosts *k8s-vs-3.novalocal* and *k8s-vs-2.novalocal* as well as outgoing connections to other unmonitored host. To create this view, a Grafana diagram plugin [Grafplugin] will be used. The HTTP response gives information about the processes running on the host as well as ingoing and outgoing connections to other hosts. In addition, the response gives also information about host clusters running in specific zones, which are similar to locations. It can also be seen, that Instana uses snapshotIDs for each component. Instana also provides a specific request including the snapshotID to get information about the component, such as label, file system or IP address. Listing 4.1 shows an excerpt of a HTTP response of a request. With the *plugin* and *label* attributes of the response, a process view is provided to display the processes on each host, such as applications and containers. Figure 4.4 displays the process view. With these views it is possible to get basic information about the infrastructure of the microservice-based system.

Listing 4.1 HTTP response with snapshotID information

```
{"id":"fnAYV",
"plugin":"docker",
"from":1510778737000,
"label":"fluentd (kube-system/fluentd-1pj7z",
"entityId":{"host":"fa:16:3e:ff:fe:52:fc:1b",,
"steadyId":"cdb9ccfe22159908754694024eb2a1196ecf527f912c254f2a52ca4188ce6a10"},
"data":{
      "Started":1510261251677,
      "memory.limit":8371642368,
      "Labels":{
            "io.kubernetes.pod.name":"fluentd-1pj7z",
            "Description":"Fluentd docker image",
                "io.kubernetes.container.terminationMessagePath":"/dev/termination log",
```

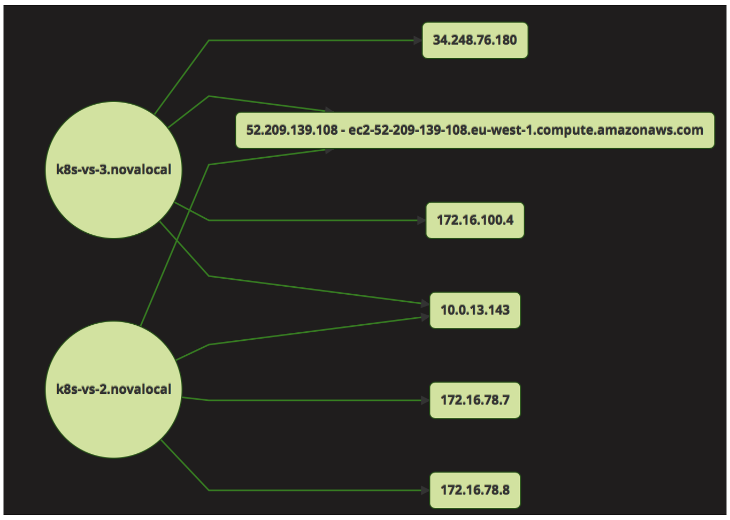


Figure 4.3.: Host view

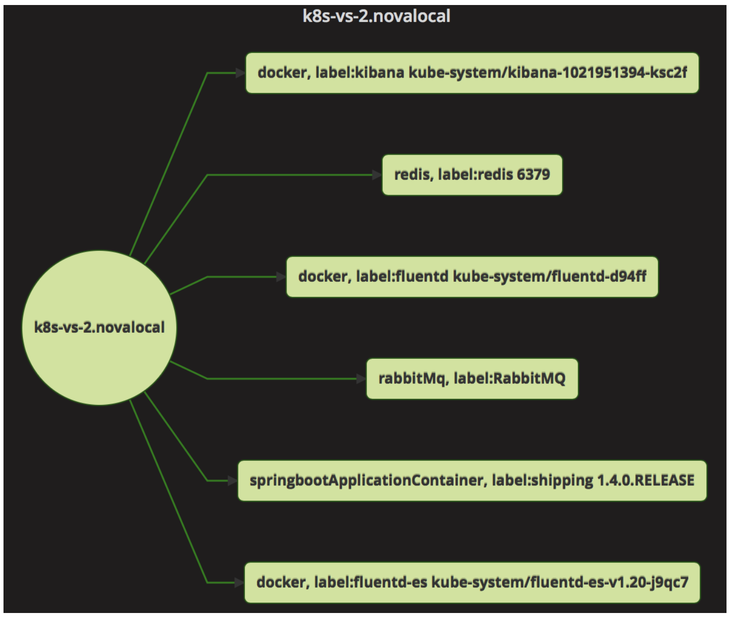

k8s-vs-2.novalocal
docker, label:kibana kube-system/kibana-1021951394-ksc2f
redis, label:redis 6379
docker, label:fluentd kube-system/fluentd-d94ff
k8s-vs-2.novalocal
rabbitMq, label:RabbitMQ
springbootApplicationContainer, label:shipping 1.4.0.RELEASE
docker, label:fluentd-es kube-system/fluentd-es-v1.20-j9qc7

Figure 4.4.: Process view

4.3.2. Infrastructure Metrics

Besides the infrastructure views, the infrastructure dashboard also displays important metrics of the hardware components. Each host has a panel for host specific metrics. Therefore, the host specific metrics are:

- CPU usage
- Memory usage
- CPU load

Due to the containerization in microservice-based systems, it is also necessary to include container monitoring to infrastructure monitoring. This leads to following container metrics:

- CPU usage
- Memory usage
- IO activity (read,write)

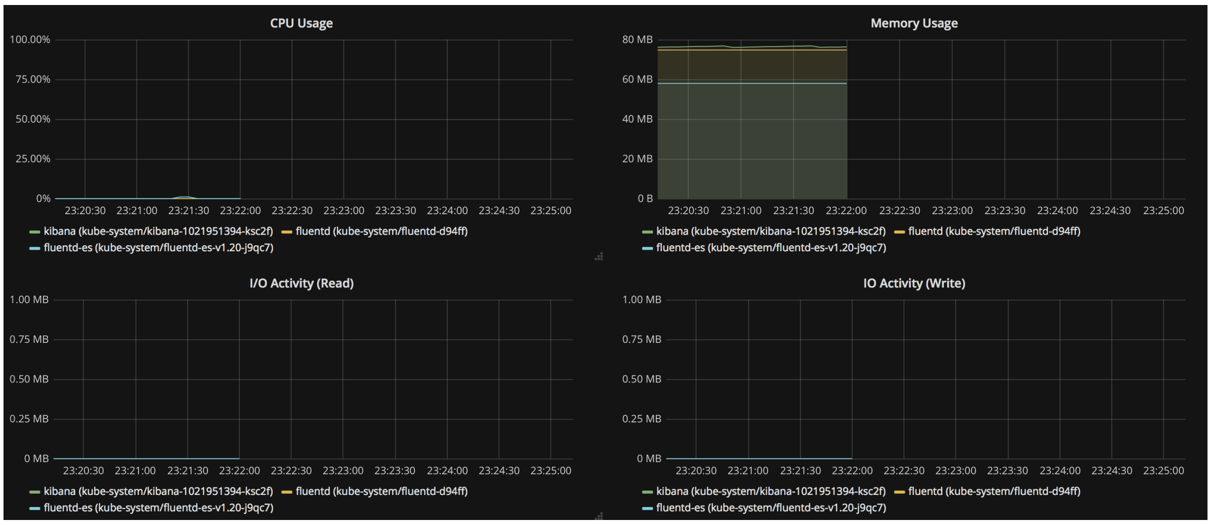

CPU Usage
Memory Usage
I/O Activity (Read)
IO Activity (Write)
kibana (kube-system/kibana-1021951394-ksc2f)
fluentd (kube-system/fluentd-d94ff)
fluentd-es (kube-system/fluentd-es-v1.20-j9qc7)

Figure 4.5.: Container metric panel

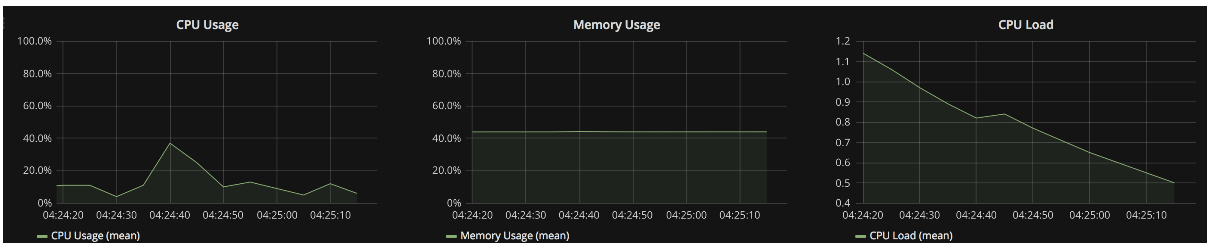


Figure 4.6.: Host metric panel

4.4. Application Dashboard

Since the infrastructure dashboard is related to the infrastructure monitoring, the application dashboard provides various metrics and information in regard to application monitoring including the different services. To achieve this, different forms of visualization will be used.

4.4.1. Application View

The application dashboard contains a service view panel for displaying the detected services and the dependencies between them. With the help of this view, the application will be broken down into the underlying services. This is necessary for root cause analysis since performance issues of one microservice can affect the performance of another microservice because of the interaction between them. To create this view, different REST requests will be used to get information about incoming and outgoing connections. Figure 4.7 shows the service view including the dependencies. In this case, requests form an outstanding service to a *Web App* have been detected. This *Web App* also interacts with a *MongoDB database*

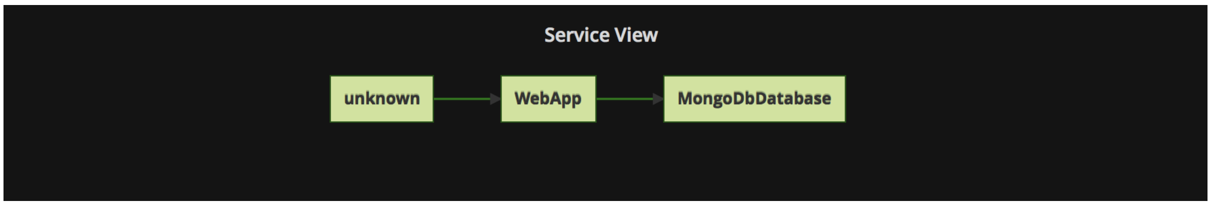


Figure 4.7.: Service view

4.4.2. Application Metrics

In regard to the application-related metrics, the application dashboard contains different metric panels for each detected service. To monitor the service performance, some of the metrics proposed by Google´s Site Reliable Engineering [BJPM16], will be combined with the Instana KPIs. This leads to following service-related metrics provided by the REST API:

- Avg. latency
- Calls per second
- Error rate

- Instances

As an example, Figure 4.8 displays the service metrics of the *Web App*.

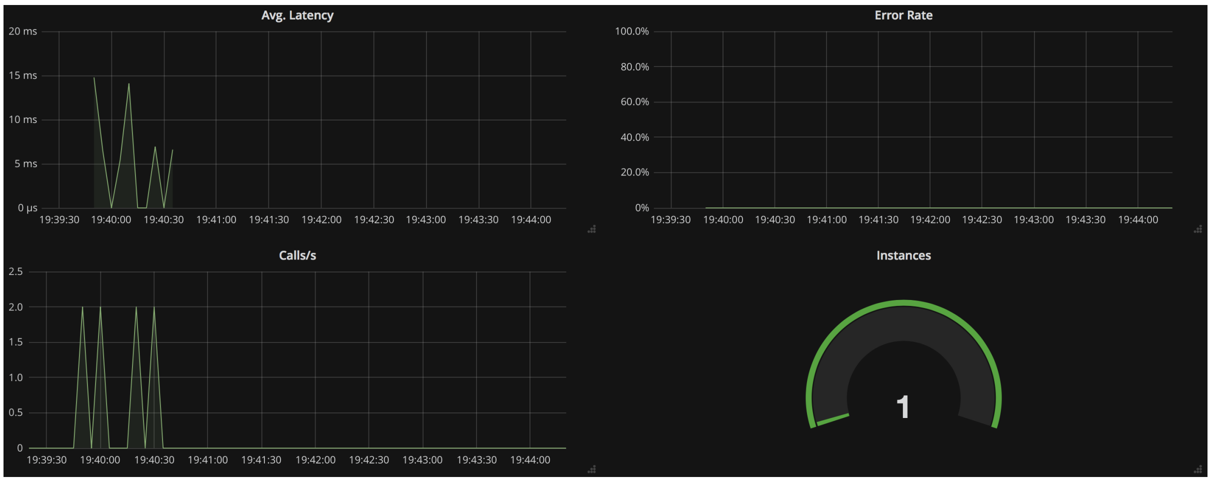
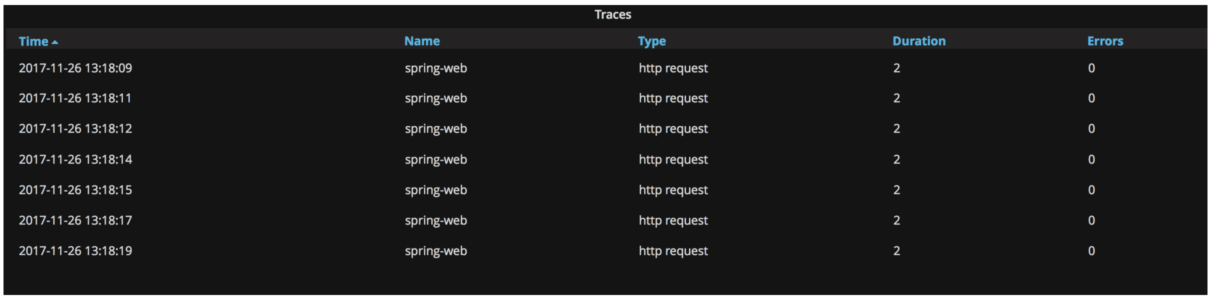


Figure 4.8.: Service metrics

4.4.3. Traces

Distributed Tracing is an underlying approach to inspect the communication and the response time of microservices within the IT environments. For this purpose, the application dashboard also contains a trace table to display important data about each trace. The information will be collected by using an HTTP request and the response with trace data, mentioned in Subsection 3.5.8. Information about the name, duration, errors and the trace type will be parsed from the HTTP response. The trace data also contains the stacktrace and the underlying spans of a trace. Figure 4.9 displays the trace table of the application dashboard.



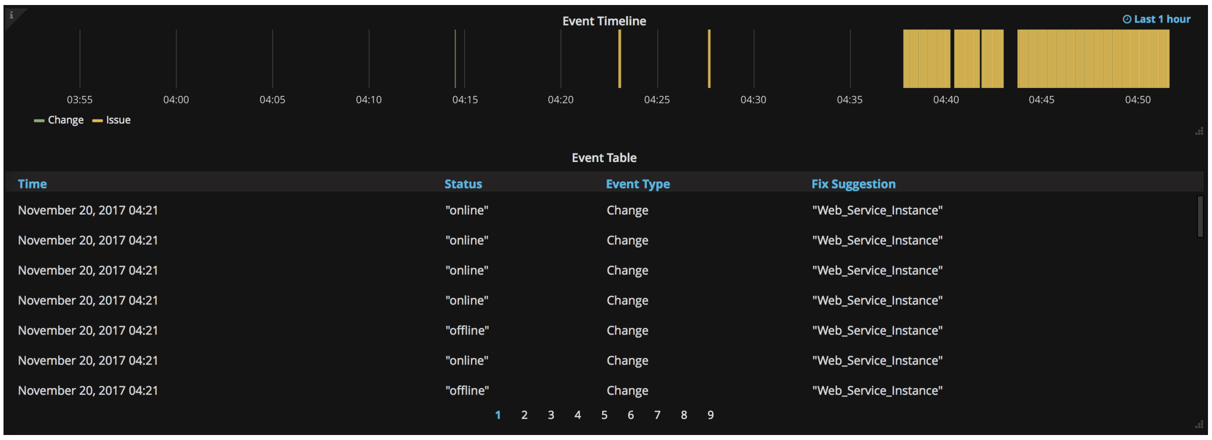
Traces

Time ▴	Name	Type	Duration	Errors
2017-11-26 13:18:09	spring-web	http request	2	0
2017-11-26 13:18:11	spring-web	http request	2	0
2017-11-26 13:18:12	spring-web	http request	2	0
2017-11-26 13:18:14	spring-web	http request	2	0
2017-11-26 13:18:15	spring-web	http request	2	0
2017-11-26 13:18:17	spring-web	http request	2	0
2017-11-26 13:18:19	spring-web	http request	2	0

Figure 4.9.: Trace data table

4.5. Event Dashboard

Besides the dashboards consisting of metrics and information about the microservice environment, it is also important to focus on occurred events. For this purpose, the event dashboard contains data about each event within the microservice-based system. A timeline will be used to support the identification of events, while a table will be used to display the information in regard to each event.



Event Table

Time	Status	Event Type	Fix Suggestion
November 20, 2017 04:21	"online"	Change	"Web_Service_Instance"
November 20, 2017 04:21	"online"	Change	"Web_Service_Instance"
November 20, 2017 04:21	"online"	Change	"Web_Service_Instance"
November 20, 2017 04:21	"online"	Change	"Web_Service_Instance"
November 20, 2017 04:21	"offline"	Change	"Web_Service_Instance"
November 20, 2017 04:21	"online"	Change	"Web_Service_Instance"
November 20, 2017 04:21	"offline"	Change	"Web_Service_Instance"

Figure 4.10.: Event dashboard

4.6. Automation & Updates

As the industrial case study already clarified, microservice environments including their components are tending to perform different change events. Due to these changes, it is also important to focus on automation processes for microservice monitoring concepts, which are detecting changes and finally updating the dashboards. This also includes metrics and other collected data. In regard to this monitoring concept, any updates will be recognized by storing the component information and snapshotIDs into a separate XML file. Moreover, the implemented service will compare the current snapshotIDs of the components with the previous snapshotIDs stored within the XML file within each time interval. The time interval will be set on one second. If a change occurred, the XML file will be updated immediately. In addition, the current dashboards will be pulled trough the provided Grafana HTTP API. Since the Grafana HTTP API allows to update already existing dashboards, the received JSON data from the request will be modified and sent back to Grafana. Listing 4.2 shows an example HTTP POST request to update the infrastructure dashboard. To achieve this, a token has to be generated in the Grafana settings. The token ID has to be sent with each request to ensure the authorization. In addition, the *overwrite* attribute has to be added with the value *true*. Token ID and the attribute are marked within the example.

Listing 4.2 HTTP POST request example

```
POST http://localhost:3000/api/dashboards/db HTTP/1.1
Accept: application/json
Content-Type: application/json
Authorization: Bearer
    eyJrIjoiWW8xU0dNNm5uUHQ5QVB5bDhJNmZWN3VFRm9DVzNOTHMiLCJuIjoidGhlc2lzIiwiaWQiOjF9

{
 "__inputs": [],
 "__requires": [],
  "editable": true,
  "gnetId": null,
  "graphTooltip": 0,
  "hideControls": false,
  "id": null,
  "links": [],
  "refresh": "5s",
    "rows": [
     {
     }
    ],
   "timezone": "",
   "schemaVersion": 6,
   "title": "Infrastructure Dashboard",
   "version": 30,
    "overwrite":  true
}
```

Chapter 5

Conclusion

This chapter will summarize and discus the thesis. In addition, possible entry points for future work will be proposed.

5.1. Summary

In summary the thesis clarified the actual state-of-the-art of APM in microservice-based systems including the used approaches, techniques and metrics. This also includes the rising challenges and technologies used in microservice-based systems. Furthermore, the study has shown that the basic APM activities are remaining the same while the monitoring of the infrastructure and the applications as well as the analytical approach need adaption. With the rise of container technologies, the focus should also be set on the monitoring of these types of components. Moreover, dynamic models of the environments have to be generated to understand the dependencies and interactions between the various services. This also leads to new pattern recognition and anomaly detection approaches. By designing and conducting the industrial case study, the work also clarified how already existing monitoring tools are overcoming the challenges of monitoring microservice environments.

Furthermore, the thesis proposed an elaborated experimental concept for monitoring microservice environments with the use of existing tools. This includes the selection of tools and the design of different dashboards. In addition, a service has been implemented as a main component to process the data. Due to the industrial case study, new microservice-specific metrics could be combined with the proposed metrics from the experimental dashboards of Mell et al., to create an overall monitoring concept including application and infrastructure monitoring as well as event handling. To achieve this, various JSON data has been parsed from the HTTP responses to extract the component data and metrics for the dashboards.

5.2. Discussion

Throughout the conduct of the industrial case study, several challenges and obstacles arose. Among others, it was not always possible to get detailed information about the underlying approaches and techniques of the commercial monitoring tool due to company secrets. In regard to the dynamic graph, it was not possible to inspect how the commercial monitoring tool is correlating the information of issues and metrics in detail. Furthermore, the industrial IT environment was still under development which did not enable the inspection on how Instana practical handles massive scaled environments.

In regard to the elaboration of the concept, static thresholds could not be set with Grafana. Setting static thresholds led to false anomaly detections due to the unusual behavior of microservice environments. By adding the event dashboard to the concept, it was possible to redeem this disadvantage by pulling the events that Instana already recognized based on the dynamic graph and other knowledge approaches. Furthermore, it was challenging to display the data due to the limited forms of visualization of Grafana and the available plugins.

The supplemental material with the implementations for this thesis are publicly available at GitHub [VaSeife].

5.3. Future Work

This chapter contains different aspects and entry points for future work.

5.3.1. Dynamic Models

Since the monitoring of microservice-based systems is still advancing it would be interesting to see how other APM tools are correlating incidents and creating models of the environments. This might also be an entry point for other model approaches besides a dynamic graph.

5.3.2. Pattern Recognition & Analytics

Nowadays, the analysis of the microservice behavior is still challenging. Approaches like pattern recognition or other machine learning processes are still under development. It is interesting to see how this processes will be achieved in detail.

5.3.3. Case study on complex productive systems

As the industrial IT environment was still under development, it would be interesting to inspect how the monitoring tool overcomes some of the use cases in complex and massive scaled productive environments.

Appendix A

UML Diagrams

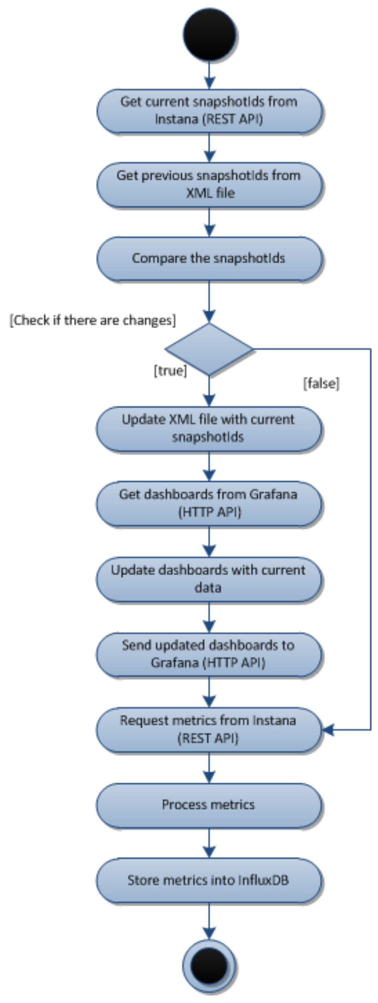

Get current snapshotIds from Instana (REST API)
Get previous snapshotIds from XML file
Compare the snapshotIds
[Check if there are changes]
[true]
[false]
Update XML file with current snapshotIds
Get dashboards from Grafana (HTTP API)
Update dashboards with current data
Send updated dashboards to Grafana (HTTP API)
Request metrics from Instana (REST API)
Process metrics
Store metrics into InfluxDB

Figure A.1.: Flow chart of the implemented service to illustrate the activity within each time interval

Appendix

Bibliography

[AppDyn] *AppDynamics - Application Performance Monitoring and Management*. URL: https://www.appdynamics.com (cit. on p. 20).

[Ber14] D. Bernstein. "Containers and Cloud: From LXC to Docker to Kubernetes." In: *IEEE Cloud Computing* (2014), pp. 81–84 (cit. on p. 11).

[BGO+16] B. Burns, B. Grant, D. Oppenheimer, E. Brewer, J. Wilkes. "Borg, Omega, and Kubernetes." In: *ACM Queue* (2016), pp. 70–93 (cit. on p. 10).

[BJPM16] B. Beyer, C. Jones, J. Petoff, N. R. Murphy. *Site Reliability Engineering*. O'Reilly Media Inc., 2016 (cit. on p. 55).

[BLNA16] P. Baron, F. Lange, M. Novakovic, P. Abrams. "Application Performance Management System With Collective Learning." Patent US 2016/0364282 A1 (US). Dec. 2016 (cit. on pp. 18–20).

[BSS10] J. Benz, M. Sassano, V. Seifermann. *Evaluation of Application Performance Management Tools in the Context of an Existing Enterprise IT Landscape*. Tech. rep. University of Stuttgart, 2010 (cit. on pp. 4, 16).

[ByteB] *Byte Buddy - code generation and manipulation library*. URL: http://bytebuddy.net/#/ (cit. on p. 31).

[DHHP17] T. Düllmann, A. van Hoorn, R. Heinrich, T. Pitakrat. "CASPA: A Platform for Comparability of Architecture-based Software Performance Engineering Approaches." In: *Proceedings of IEEE International Conference on Software Architecture Workshops*. IEEE, 2017 (cit. on p. 51).

[Docker] *Docker - Build, Ship, and Run Any App, Anywhere*. URL: https://www.docker.com/ (cit. on p. 9).

[DynamicGraph] *Monitoring Microservice Applications: Introducing the Dynamic Graph*. URL: https://www.instana.com/blog/monitoring-microservice-applications-introducing-dynamic-graph/ (cit. on p. 38).

[Dynatrace] *Dynatrace - Digital Performance Management and Application Performance Monitoring*. URL: https://www.dynatrace.com (cit. on p. 17).

[Grafana] *Grafana - The open platform for analytics and monitoring*. URL: https://grafana.com/ (cit. on pp. 22, 23).

[Grafplugin] *Diagram Plugin for Grafana*. URL: https://grafana.com/plugins/jdbranham-diagram-panel (cit. on p. 52).

[Hai16] C. Haight. “APM Needs to Prepare for the Future.” In: *Gartner* (2016). URL: https://www.gartner.com/doc/3251417/apm-needs-prepare-future (cit. on p. 3).

[HHK+17] R. Heinrich, A. van Hoorn, H. Knoche, F. Li, L. E. Lwakatare, C. Pahl, S. Schulte, J. Wettinger. “Performance Engineering for Microservices: Research Challenges and Directions.” In: *Companion Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering Companion. ICPE*. ICPE, 2017 (cit. on pp. 3, 15).

[HHMO17] C. Heger, A. van Hoorn, M. Mann, D. Okanovic. “Application Performance Management: State of the Art and Challenges for the Future.” In: *Proceedings of the 8th ACM/SPEC on International Conference on Performance Engineering*. ICPE, 2017, pp. 429–432 (cit. on p. 12).

[HWH12] A. van Hoorn, J. Waller, W. Hasselbring. “Kieker: A Framework for Application Performance Monitoring and Dynamic Software Analysis.” In: *Proceedings of the 3rd ACM/SPEC International Conference on Performance Engineering*. ICPE, 2012, pp. 247–248 (cit. on p. 13).

[InfluxDB] *InfluxData (InfluxDB) - Time Series Database Monitoring and Analytics*. URL: https://www.influxdata.com/ (cit. on p. 22).

[InfluxDB-java] *API for influx database*. URL: https://github.com/ashishdoneriya/influxdb-java (cit. on p. 50).

[Instana] *Instana*. URL: https://www.instana.com (cit. on pp. 13, 18).

[InstanaDoc] *Instana Documentation*. URL: https://docs.instana.io/ (cit. on pp. 32, 38).

[InstanaInst] *How Instana safely instruments applications for monitoring*. URL: https://www.instana.com/blog/how-instana-safely-instruments-applications-for-monitoring/ (cit. on p. 31).

[Jaeger] *Jaeger, a distributed tracing system*. URL: http://jaeger.readthedocs.io (cit. on p. 13).

[K6] *K6 - Open source load testing tool*. URL: https://k6.io/ (cit. on pp. 22, 23).

[KubeDoc] "*Kubernetes Documentation*". URL: https://kubernetes.io/docs/ (cit. on pp. xv, 11).

[Kubern] *Kubernetes - Production-Grade Container Orchestration*. URL: https://kubernetes.io/ (cit. on p. 10).

[Kuf16] Ł. Kufel. "Tools for Distributed Systems Monitoring." In: (Dec. 2016) (cit. on p. 17).

[LBNA16] F. Lange, P. Baron, M. Novakovic, P. Abrams. "Application Performance System With Dynamic Discovery And Extension." Patent US 2016/0364281 A1 (US). Dec. 2016 (cit. on pp. 18, 19).

[MG11] P. M. Mell, T. Grance. *SP 800-145. The NIST Definition of Cloud Computing*. Tech. rep. 2011 (cit. on p. 7).

[Mou16] A. Mouat. *Using Docker: Developing and Deploying Software with Containers*. O'Reilly Media Inc., 2016 (cit. on pp. 9, 10).

[MS12] A. Rajasekar, M. Wan, R. Moore, W. Schroeder. "Micro-Services: A Service-Oriented Paradigm for Scalable, Distributed Data Management." In: *Data Intensive Distributed Computing: Challenges and Solutions for Large-scale Information Management*. IGI Global, 2012, pp. 74–93 (cit. on p. 3).

[MW17] B. Mayer, R. Weinreich. "A Dashboard for Microservice Monitoring and Management." In: *International Conference on Software Architecture Workshops (ICSAW)*. IEEE, 2017, pp. 66–69 (cit. on p. 16).

[New15] S. Newman. *Building Microservices*. O'Reilly Media Inc., 2015 (cit. on pp. 8, 12).

[PostgreSQL] *PostgreSQL*. URL: https://www.postgresql.org/ (cit. on p. 25).

[ROAM17] P. D. Franceso, P. Lago, I. Malavolta. "Research on Architecting Microservices: Trends, Focus, and Potential for Industrial Adoption." In: *Proceedings of IEEE International Conference on Software Architecture Workshops*. IEEE, 2017, pp. 21–30 (cit. on p. 3).

[SBB+10] B. H. Sigelman, L. A. Barroso, M. Burrows, P. Stephenson, M. Plakal, D. Beaver, S. Jaspan, C. Shanbhag. *Dapper, a Large-Scale Distributed Systems Tracing Infrastructure*. Tech. rep. Google, Inc., 2010 (cit. on p. 12).

[Sock-Shop] *Sock Shop*. URL: https://microservices-demo.github.io/ (cit. on p. 51).

[SpringBoot] *Spring Boot*. URL: https://projects.spring.io/spring-boot/ (cit. on p. 25).

[Terra] *Terraform - Write, Plan, and Create Infrastructure as Code*. URL: https://www.terraform.io/ (cit. on p. 25).

[TICK] *TICK - Stack Based Open-Source Platform*. URL: https://www.influxdata.com/time-series-platform/ (cit. on p. 17).

[VaSeife] "*Supplemental Material for the Bachelor´Thesis "Application Performance Monitoring in Microservice-BasedSystems*". URL: https://github.com/ValeSayfa/Application-Performance-Monitoring-in-Microservice-Based-Systems (cit. on p. 60).

[ZCB10] Q. Zhang, L. Cheng, R. Boutaba. “Cloud computing: state-of-the-art and research challenges.” In: *Journal of Internet Services and Applications* (2010), pp. 7–18 (cit. on p. 8).

[Zipkin] "*OpenZipkin - A distributed tracing system*". URL: http://zipkin.io. (cit. on p. 13).

All links were last followed on November 27, 2017.

Declaration