Institute of Parallel and Distributed Systems

University of Stuttgart
Universitätsstraße 38
D–70569 Stuttgart

Masterarbeit

Generic Templates for Monitoring Agents

Marc Weise

Course of Study:	Softwaretechnik
Examiner:	Prof. Dr.-Ing. habil. Bernhard Mitschang
Supervisor:	Dipl.-Inf. Mathias Mormul
Commenced:	May 14, 2018
Completed:	November 14, 2018

Abstract

This thesis presents an agent-centric approach for monitoring IT resources, which enables the execution of preprocessing and aggregation steps directly on the target systems in order to limit data transfers to a central server and allow a local event detection and treatment. To keep the agent behavior definition as simple as possible, an extendable template model is introduced which can be used to define Monitoring Pipelines by chaining individual processing steps. Furthermore this work demonstrates how a graphical editor can be implemented which also allows non-experts in the field of monitoring to create and modify Monitoring Templates.

Kurzfassung

Diese Abschlussarbeit stellt einen agenten-zentrierten Ansatz für die Überwachung von IT-Ressourcen vor, der es ermöglicht Aggregations- und Vorverarbeitungsschritte direkt auf den Zielsystemen durchzuführen um damit Datenübertragungen an einen zentralen Server zu beschränken und Ereignisse lokal zu erkennen und zu behandeln. Um die Definition des Agentenverhaltens für den Nutzer so einfach wie möglich zu gestalten wird ein erweiterbares Template-Modell eingeführt, welches es erlaubt Monitoring Pipelines durch die Verkettung einzelner Verarbeitungsschritte zu definieren. Die Arbeit zeigt außerdem, wie ein grafischer Editor für diese Monitoring Templates umgesetzt werden kann, um auch Nutzern ohne Fachwissen im Bereich Monitoring das Erstellen und Bearbeiten von Monitoring Templates zu erlauben.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

AI Artificial Intelligence. 13

CEP Complex Event Processing. 13

CPU Central processing unit. 7

CSP Cloud Service Provider. 13

CSV Comma-separated values. 52

DAG Directed acyclic graph. 33

DB Database. 33

FR Functional Requirement. 29

GB Gigabyte. 51

GUI Graphical User Interface. 46

HTTP Hypertext Transfer Protocol. 44

IaaS Infrastructure as a Service. 16

IoT Internet of Things. 13

IT Information technology. 13

JMX Java Management Extensions. 25

JSON JavaScript Object Notation. 33

JVM Java Virtual Machine. 25

MVM Monitoring VM. 20

NIST National Institute of Standards and Technology. 15

NR Non-functional Requirement. 29

OS Operating System. 13

PaaS Platform as a Service. 16

QoS Quality of Service. 13

RAM Random-Access Memory. 51

REST Representational state transfer. 7

RMCM Runtime Model for Cloud Monitoring. 5, 25

RMI Remote Method Invocation. 26

SaaS Software as a Service. 16

SLA Service-level agreement. 13

SPA Single-page application. 46

SSH Secure Shell. 26

SVG Scalable Vector Graphics. 46

TOML Tom's Obvious, Minimal Language. 46

UC Use case. 40

UDP User Datagram Protocol. 45

UI User Interface. 7

UML Unified Modeling Language. 7

VM Virtual Machine. 14

VPN Virtual private network. 51

XML Extensible Markup Language. 33

1 Introduction

With a steadily growing adoption of cloud computing across many fields, an effective and efficient monitoring of Information technology (IT) resources becomes critical for many businesses. Although monitoring was studied for many years, today it is needed more than ever before: In the era if Big Data and Cloud Computing, huge amounts of resources have to be monitored and controlled in order to ensure Quality of Service (QoS) for the customers and detect Service-level agreement (SLA) violations of the Cloud Service Provider (CSP). Monitoring is needed for many activities like capacity and resource planning, management of data centers, billing, troubleshooting, security and performance management for both cloud providers and customers [ABDP12]. Movements like Internet of Things (IoT), Industry 4.0 and Edge Computing lead to the integration of more and more systems with different system architectures/Operating Systems (OSs) and amounts of computing power. Splitting services into smaller parts (Microservices) makes aggregation of monitoring data more important for troubleshooting, because more services mean more moving parts inside a system and errors rather occur during the interaction of multiple services than within a single one.

Today a huge variety of monitoring tools is available, both commercial and open source. But many of the established solutions were rather developed for Grid or Cluster Computing then for Cloud Computing [ABDP13]. Therefore many of the existing solutions face challenges related to the huge amount of data they have to handle, the dynamic nature they face in a cloud computing scenario and the heterogeneity of the systems to be monitored. Many of the existing monitoring approaches rely on sending raw data to a central server without prior filtering or aggregation, which has many drawbacks: It introduces a single point of failure and a bottleneck in terms of computing power, memory consumption and network bandwidth and thereby limits the scalability of the affected systems. The resulting data transfers may hurt the performance of the applications running on the observed systems in the first place or produce unnecessary costs if the pay-per-use model also applies to the network traffic produced by these systems. Despite this the integration of new tooling like Artificial Intelligence (AI) and Complex Event Processing (CEP) with existing solutions is often rather cumbersome because the lack of appropriate interfaces or insufficient extendability of the tooling in general. The multitude and variety of monitoring solutions available makes it hard to choose the right tool for a specific monitoring task and after choosing one often leads to vendor lock-ins because it is hard to migrate existing configurations from one tool to another. Furthermore configuring monitoring tools is a tedious and error-prone task: It often requires editing huge configuration files which are scattered all over a remote system and follow an obscure and non-standard syntax. Therefore it requires a lot of experience in operating such systems. In contrast to that, movements like DevOps encourage developers with less experience in this field to also accept responsibility for running their applications.

The goal of this thesis is to overcome the problems mentioned above by applying an agent-centric monitoring approach: Filtering, preprocessing and aggregation steps should rather be executed directly on the Monitoring Agent than on a central server. Furthermore technology agnostic templates for Monitoring Agents are introduced to simplify the management of the involved agents

and its configurations. These templates describe the agent behavior by modeling its internal data flow: From gathering metrics on the target system over executing preprocessing and aggregation steps where data is ingested up to triggering actions locally and sparingly sending aggregated data over the network. The proposed agent templates can be put under version control to make monitoring configuration more maintainable and reusable. They are designed to be extendable and provide interfaces for an easy Integration with tools for AI and CEP. The creation of these templates can be done using a simple web-based UI and allows the automatic generation of technology specific configuration files for the agents afterwards. This makes setting up and configuring Monitoring Agents much easier, especially for people with less experience in running these kinds of tools.

Running Example

For illustrating the monitoring approach presented in this thesis, a running example is introduced: Let's assume a web application like an online shop which consists of a distributed deployment of multiple Microservices running on Virtual Machines (VMs) hosted in public clouds of different CSPs. Basic metrics like CPU, memory and disc usage should be monitored on each VM in order to detect issues like memory leaks and to be able to scale in or out depending on the current workload and ensure a high availability of each service involved. When a monitored VM gets overloaded or runs out of memory these events should be detected as soon as possible. Depending on the kind of issue detected, either automatic actions should be triggered (e.g. scaling horizontally by starting new instances of an overloaded service) or an alert should be send out e.g. via email if manual actions are needed.

Document Outline

The remainder of this thesis is structured as follows: Chapter 2 defines some basic terms and taxonomies in the context of monitoring to build up a common ground for the following chapters. Chapter 3 presents some recent implementations of Monitoring Agents and discusses its advantages and disadvantages. Chapter 4 provides an overview of related scientific work in the field of modeling Monitoring Agents and differentiates them from the approach presented in this thesis. Chapter 5 defines desirable properties for Monitoring Agents and concrete use cases for the monitoring system. Chapter 6 shows the structure of the Monitoring Templates and explains its different parts. Chapter 7 describes structure and individual parts of the monitoring system together with its behavior and implementation. Chapter 8 evaluates the implementation and discusses the results of the evaluation. Finally Chapter 9 concludes this work and gives a brief outlook into future directions.

2 Background and Definitions

This chapter includes some background information about related topics and defines terms and taxonomies which are either used in the following chapters or whose knowledge is crucial for understanding them.

2.1 Cloud Computing

It is assumed that the applications and systems to be monitored are running in a cloud computing environment. To build a foundation for the following chapters it is important to first define this context: One of the most common definitions of Cloud Computing is from the National Institute of Standards and Technology (NIST): According to the latter it is a "model for enabling ubiquitous, convenient, on-demand network access to a shared pool of configurable computing resources [...] that can be rapidly provisioned and released with minimal management effort or service provider interaction" [MG+11]. This definition is further refined by descriptions of essential characteristics (Section 2.1.1) as well as service (Section 2.1.3) and deployment models (Section 2.1.4).

With the increasing popularity of this computing paradigm, Cloud Computing is one of the most important areas of application for monitoring.

2.1.1 Characteristics

According to the NIST definition [MG+11] there are five essential characteristics of cloud computing: First of all there is an *on-demand self-service* where the Cloud Consumer can provision resources by himself without requiring human interaction. The offered capabilities of a cloud service have a *broad network access* which means that they can be accessed via the network by using standard mechanisms. Provided resources are taken from *resource pools* and can be used by multiple different tenants. *Rapid elasticity* regarding provisioning and releasing of resources enables scalability. Scaling seems to be unlimited and can even be automated. Finally cloud offerings are *measured services*: Resource usage is transparently measured and these measurements are used for optimization and billing.

2.1.2 Layers

According to the Cloud Security Alliance [Spr11a; Spr11b] there are seven layers in cloud computing where each layer is controlled either by the CSP or the cloud customer. These layers are described as follows:

Facility refers to the facilities in a data center.

Network refers to the connection between CSPs facilities and the internet

Hardware in the data center, e.g. processors and disks

OS of the VM

Middleware refers to functionality between OS and application such as runtime environments

Application running on top of the middleware which is used by the end user

User refers to the end user of a cloud service

Depending on the layer selected, different parameters can be measured [ABDP12; ABDP13].

2.1.3 Service Models

According to the NIST [MG+11] there are three different service models in cloud computing. Beside defining what the Cloud Consumer has to pay for, the service model also influences how many of the layers mentioned above are controlled by CSP.

Infrastructure as a Service (IaaS) is about the provisioning of fundamental IT resources such as processing, storage and networks. The Cloud Consumer can deploy arbitrary applications on these resources and control OS and storage but does not control the underlying infrastructure.

Platform as a Service (PaaS) allows the Cloud Consumer to deploy applications build with programming languages services and tools supported by the CSP. The consumer does not control server, OS, storage or network but does control the deployed application and may be able to configure the hosting environment.

Software as a Service (SaaS) allows the Cloud Consumer to use applications deployed on the infrastructure of the CSP. Cloud applications can be accessed via client programs such as browsers. The Cloud Consumer does not control servers, OS, storage, network and parts of the application but may control some predefined application settings.

Depending on the selected service model monitoring capabilities are restricted for the Cloud Consumer because e.g. infrastructure can not be accessed in PaaS or SaaS. In the running example of this thesis mainly the IaaS service model is applied. Therefore custom monitoring agents can be installed by the cloud customer and agent-based monitoring can be used to gather metrics about the involved VMs.

2.1.4 Deployment Models

The NIST definition of Cloud Computing describes four different deployment models. Depending on the deployment model the group of people which share resources is more or less restricted.

Private Cloud is used by multiple consumers within a single organization. It can be operated by this organization or a third party and exist on or off premise.

Community Cloud is used by a community with shared concerns. It is operated by an organization within this community or a third party and may exist on or off premise.

Public Cloud can be used by the general public and exist on premise of the CSP.

Hybrid Cloud refers to a combination of two or more of the deployment models above.

The selection of the deployment model has implications on privacy because the cloud consumer has more or less control over the networks involved. Regarding deployment models the running example can be classified as a Public or Hybrid Cloud.

2.1.5 Roles

There are different stakeholders in Cloud Computing and each stakeholder can be assigned to one or more of the following roles [NSHS14; SWWM10]:

The *Cloud Operator* owns the IT resources and is concerned about the running status of the hardware, its utilization and usage anomalies for planning resource provisioning. The *Service Developer* is concerned about the running status of his services e.g. he wants to be informed if one of his services crashes and in case of such an incident he is interested in log files and stack traces for troubleshooting and debugging the issue. The *End User* is concerned about the availability of the services he is paying for, e.g. he wants to see whether there are SLA violations.

2.2 Edge Computing

Beside Cloud Computing which was described above, recent efforts to connect more and different kinds of devices with each other (known as IoT) foster the need to do more processing at the edge of the emerging networks. This computing paradigm is called “Edge Computing” and “edge” refers to IT resources which are located between data sources and the data centers of cloud computing [SCZ+16].

Shi et al. define Edge Computing as “enabling technologies allowing computation to be performed at the edge of the network, on downstream data on behalf of cloud services and upstream data on behalf of IoT services” [SCZ+16]. This new computing paradigm favors data processing to take place on devices at the proximity of data sources. According to Shi, recent movements like the IoT will lead to huge amounts of data being produced and consumed at the edge of the network and Cloud Computing will not be efficient enough to handle this amount of data due to lack of bandwidth.

Shi [SCZ+16] identifies several benefits of Edge computing in comparison to Cloud Computing: Response time, bandwidth costs and energy consumption can be reduced. But doing more data processing at the edge of the network may also lead to new challenges like naming and addressing huge numbers of mobile devices forming dynamic topologies. Also developing software may be more difficult because environments are much more heterogeneous than in the cloud.

2.3 Monitoring

Built on the definitions of the environment above, the following sections focus on the monitoring activity to be executed in this context including definitions, taxonomies and important properties.

Monitoring is the activity of periodically checking a set of predefined resources for abnormal behavior, trying to automatically fix issues if they occur (auto heal) and notifying support teams via different means if issues can not be fixed in an automated manner [ARM+15].

There are many different taxonomies about monitoring, the following sections summarize some of them which are relevant for monitoring using agents.

2.3.1 Centralized vs. Decentralized Monitoring

The structure of distributed monitoring tools can either be centralized or decentralized [ARM+15]. In *centralized monitoring* on the one hand all status updates from monitored resources are sent to a central server. On the other hand *decentralized monitoring* implies that no component is more important than another one: The different nodes are organized in a structured, unstructured or hybrid peer-to-peer network.

Centralized monitoring systems are typically easier to set up and manage but however the central server represents a single point of failure and network connections, storage and compute power of this central server can easily become a bottleneck for the overall monitoring system. In contrast to that, decentralized monitoring systems avoid the single point of failure but are much more complicated to set up and to operate.

2.3.2 Interaction Model

Povedano-Molina et al. [PLL+13] identifies three models for the interaction between different entities in a distributed monitoring system: When following the *Push Model* the monitored entities asynchronously send their measurements to the monitoring entity. In contrast to that in the *Pull Model* the monitoring entity synchronously requests the measurements from monitored ones. There are also *Mixed Models* which combine both models above.

2.3.3 Properties of Monitoring Systems

Aceto et al. [ABDP12] define important properties of monitoring systems which can be used to compare different performance aspects of monitoring systems. Therefore following chapters refer to these properties as they are defined in the sections below:

Scalability describes the ability of a monitoring system to deal with increasing amount of monitoring data and probes. A monitoring system does scale if the amount of data to be processed can be increased by adding more resources like compute power or memory to the system. Ideally, the amount of resources added should be proportional to the processing power gained. In this case the monitoring system does scale well. A good scalability is especially important in cloud scenarios where a big number of resources must be monitored.

Elasticity is concerned with the behavior of a monitoring system when the entities to be monitored change: Adding and removing entities should be handled in a correct and efficient way. Being a key feature of cloud computing, a good elasticity is also very important when monitoring resources in the cloud.

Adaptivity describes the ability of a monitoring system to cope with different amounts of computational or network load without becoming invasive (e.g. by hurting the overall performance of the involved systems).

Timeliness of a monitoring system refers to whether incidents are detected in time. Which delay is acceptable depends on the intended use but in general the amount of time between an incident and its detection should be as low as possible.

Autonomicity describes whether a monitoring system is able to manage its resources on its own. It should be able to react on changes and adjust its resource usage accordingly. Furthermore the involved complexity should be hidden from the environment.

Comprehensiveness refers to whether a monitoring system supports many different resources, covers different types of monitoring data and can be used by multiple tenants.

Extensibility describes whether it is easy to add support for new resources or types of monitoring data to the monitoring system.

Intrusiveness of a monitoring system refers to the amount of modifications needed in order to run in a specific environment.

Resilience describes whether a monitoring system keeps working in case of changes and component failures.

Reliability refers to whether a monitoring system continues to provide its functionality under adverse conditions.

Availability refers to whether the provided functionality of the monitoring system can be used whenever requested.

Accuracy describes whether the values measured by the monitoring system are close to the real values.

2.3.4 Monitoring Architectures

Calero [CG15] describes five different architectures which can be used for monitoring in the cloud.

Traditional Internal Monitoring Architecture monitors only physical machines from inside the internal network of the cloud service provider. This architecture can only be used by the CSP.

Extended Internal Monitoring Architecture extends the previous architecture and makes use of a hypervisor plugin in order to collect a small set of basics metrics about the virtual machines running on a physical machine. This monitoring can only be used by the CSP and is hidden from the cloud consumer.

Extended and Adaptive Internal Monitoring Architecture adds a module which monitors topological changes in the virtual environment.

Sparse External Monitoring Architecture is an external monitoring architecture which means that the monitoring is done by a separate Monitoring VM (MVM). It can be used and controlled by the cloud service consumer.

Concentrated External Monitoring Architecture is similar to the Sparse External Monitoring Architecture, but there is a special consumer called MonPaaS which controls all MVMs.

2.3.5 Aggregation

Since monitoring may produce huge amounts of data, raw data should be minimized before being persisted or sent via network. There are two types of aggregations [CLC14; FRWZ07] which can be used to do this: On the one hand lossless aggregations allow the complete reconstruction of the original data because raw data is only concatenated into frames and redundant information is discarded. On the other hand lossy aggregations can be used. In this case a complete reconstruction is not possible, because arithmetic operations like sum or average are applied and therefore information is lost. In contrast to lossless aggregations, lossy aggregations often allow a better compression of the raw data. Furthermore lossy aggregations can also be used if raw data should not be shared for privacy reasons.

2.4 Software agents

In general software agents are software components which autonomously execute tasks within a target system [NSHS14]. They might be able to move their execution from one system to another, adapt their behavior according to their environment or communicate with remote systems which could be other software agents as part of a distributed system. Software agents can be used to cross system boundaries and access data which is only accessible within a system or perform actions which can only be executed within a system.

The task to be executed by the agent could be anything and therefore there are many different types of software agents. This thesis is focused on Monitoring Agents which collect data on the system where they are executed. Some state of the art implementations of Monitoring Agents will be presented in the following chapter.

3 Monitoring Agent Implementations

There are already many monitoring solutions available which could be used to monitor a distributed system like the one in the running example. In such a multi- or hybrid cloud scenario, monitoring can not rely on a solution provided by a single CSP. Therefore this chapter presents some recent implementations of agent-based monitoring tools together with their specific advantages and disadvantages. It also discusses whether these agent implementations are suited for using them in an agent-centric approach were processing, aggregation, alerting is done directly on the agent.

3.1 Universal and Heavy Forwarders of Splunk Enterprise

Universal and Heavy Forwarder are two different variants of the agent component used in Splunk Enterprise. Splunk Enterprise [SPLE] is a proprietary software which can ingest and index monitoring data, visualize it in dashboards, enables searches and trigger alerts. Processing of monitoring data is done by different components depicted in Figure 3.1 which can be deployed on different machines in order to enable horizontal scaling: A set of Forwarders shown on the bottom ingest monitoring data and send it to one or more Indexers in the middle of the figure. Indexers and Search Heads enable searching the monitoring data.

All types of forwarders have some basic capabilities like adding metadata, buffering, compressing and encrypting the data they forward [SPLFR]. Additionally the Heavy Forwarder [SPLFT] is a full instance of Splunk Enterprise and therefore can parse incoming data and route it depending on its content. Forwarded data can also be locally indexed and searched by the heavy forwarder and alerts can be triggered. The universal forwarder [SPLUF] has a much smaller resource footprint than the heavy forwarder and supports more platforms. It only forwards data and does not parse it. Therefore, the universal forwarder can neither index nor search the data and is not able to send alerts.

One advantage of Splunk is its support for distributed deployments which in theory makes it very scalable. But there are also some disadvantages: If monitoring data should be preprocessed, heavy forwarders have to be used which need a lot of resources because they are basically standalone instances of Splunk. Furthermore being proprietary software there is a high risk for vendor lock-in when Splunk is used.

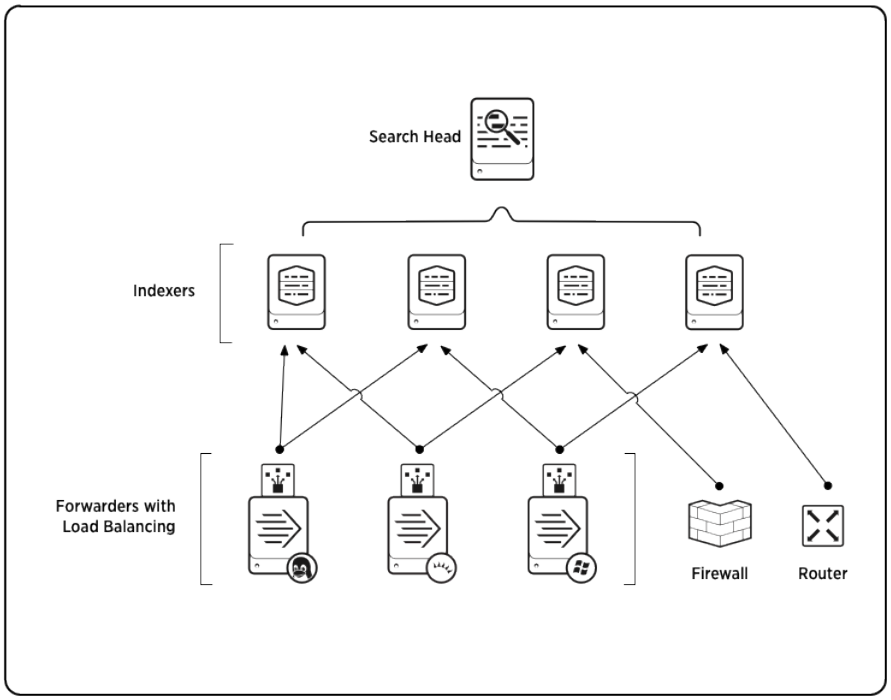


Figure 3.1: Overview of Splunk Enterprise components taken from [SPLCD]

3.2 Telegraf (TICK-Stack)

Telegraf is an agent for collecting metrics and events and part of the TICK stack [TICK] which is an open source platform for time series data mainly written in Go. Figure 3.2 shows an overview of the platform: Beside Telegraf for collecting on the left the TICK stack also includes InfluxDB for storing time series data in the middle, Chronograf for visualizing the collected data in dashboards on the top and Kapacitor for real-time processing and alerting on the bottom.

Telegraf makes extensive use of plugins and uses them in different parts of the agent: There are plugins for connecting to different data sources from reading low level metrics like CPU, memory or disk usage of a system to application statistics of databases, web servers, message queues and more. Beside that there are also plugins for processing and aggregating data and finally for sending the treated data to different destinations like log files, network endpoints, message queues or databases following the push model.

At the time of writing there are already over one hundred plugins available [TG] and new plugins can easily be created by implementing simple interfaces [TGD]. Telegraf's architecture follows the well known pipes-and-filter pattern to create a pipeline by connecting multiple of these input, processing, aggregation and output plugins mentioned before. The desired data flow is described using filters on the participating plugins. These filters and plugin configurations are defined in a configuration file.

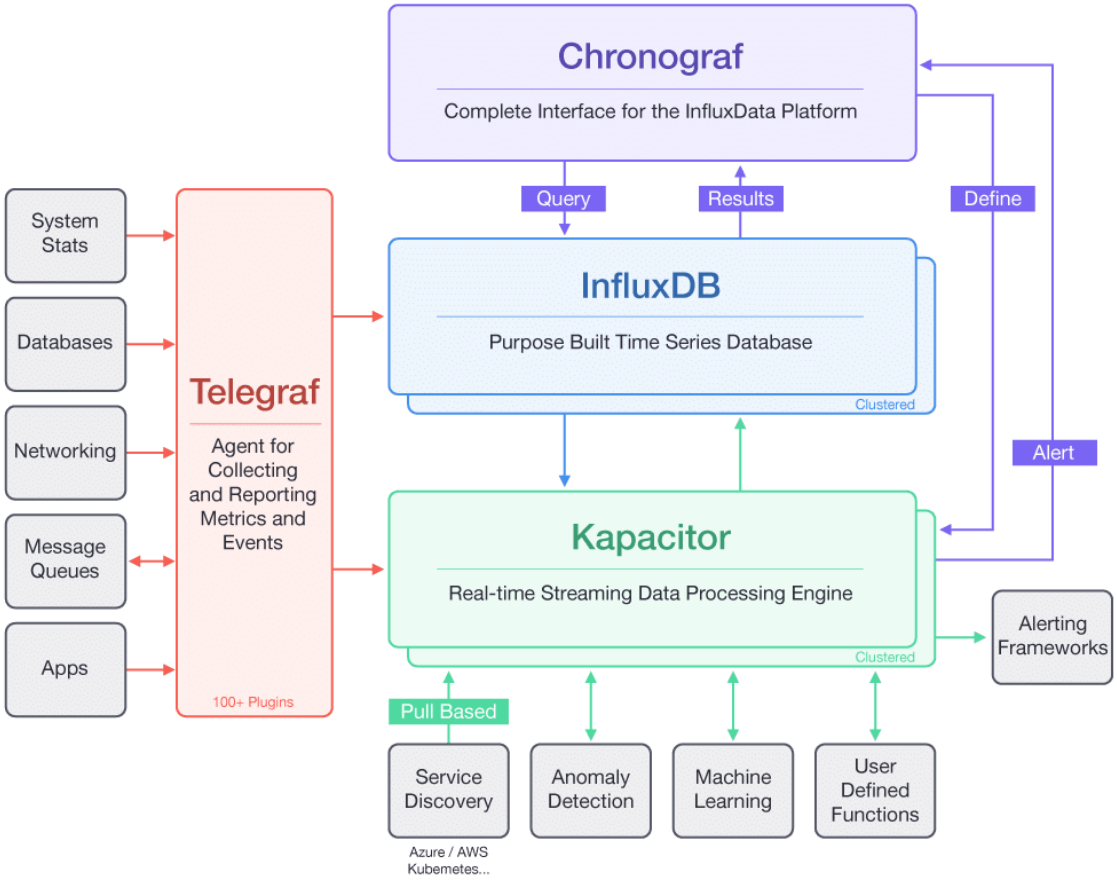


Figure 3.2: Overview of the TICK stack taken from [TICK]

On the one hand this simple but powerful plugin mechanism makes Telegraf very extendable, on the other hand configuring Telegraf can be cumbersome due to the overwhelming amount of possibilities how plugins can be connected and configured, resulting in huge configuration files: At the time of writing the commented example configuration file[1] for all included plugins was over four thousand lines long.

Telegraf's plugin mechanism makes it easy to perform processing and aggregation directly on the agent. Alerting and integration of artificial intelligence can also be performed on the agent by implementing special input and output plugins. Therefore Telegraf agents are used as a basis for the approach presented in this thesis.

[1] https://github.com/influxdata/telegraf/blob/master/etc/telegraf.conf

3.3 The Need for Modeling

As shown in the previous sections, existing monitoring implementations suffer from a complex configuration and the danger of introducing vendor lock-ins. By modeling the agent behavior, both of these issues can be tackled at the same time: On the one hand the configuration complexity of a monitoring solution can be reduced by introducing a suitable model which uses an intuitive metaphor to hide technical aspects and provide a good overview. On the other hand such a model can also mitigate vendor lock-ins: If multiple monitoring solutions provide similar functionalities, a single model can be mapped to multiple vendor-specific configurations and thereby allow the migration from one monitoring solution to another.

4 Related Work

Based on the shortcomings of existing agent-based monitoring solutions, a need for modeling was identified in previous chapter. Therefore the following sections summarizes related work which also applies modeling in the field of agent-based monitoring. It also discusses similarities and differences compared to the approach presented in the following chapters of this thesis.

4.1 Runtime Model for Cloud Monitoring (RMCM)

In [SWWM10] Shao et al. identifies and addresses three challenges in cloud monitoring: First of all many services produce a huge amount of runtime information. Second, heterogeneous components used in the cloud require different ways to monitor this information. Finally, there are many different concerns for monitoring, therefore the monitoring overhead should be kept at a minimum.

A model for cloud monitoring called Runtime Model for Cloud Monitoring (RMCM) is introduced: Different roles have different views on RMCM which models entities in an extendable hierarchy including infrastructure components, platforms, applications and end users.

Shao et al. also present a monitoring framework which implements the RMCM for the usage with services written in Java. In this framework, there are agents using different monitoring mechanisms to ingest data. Depending on the entity to be monitored, native resource monitoring libraries, a Java Virtual Machine (JVM) agent, request interceptors, Java Management Extensions (JMX) interfaces or service probes using code instrumentation can be used to gather data. Monitoring Agents check if alerts have to be triggered and store the gathered information in a central database. The overhead produced by monitoring can be dynamically adjusted at runtime by using instrumentation features of the Java programming language which allows hot-swapping of class information during runtime.

Shao's and the approach presented in this thesis both include a model for monitoring in the cloud. They both use agents to retrieve the data and allow configuration adjustments at runtime addressing the tradeoff between monitoring capability and overhead. In contrast to this work, Shao's model is rather abstract: It allows to define entities to be monitored, but it does not allow to model the exact data flow including preprocessing steps like aggregations. RMCM also has a special focus on Java programming language and platform whereas this approach is language agnostic.

4.2 GMonE

Montes et al. [MSM+13] describe a taxonomy for cloud monitoring, which consists of a monitoring level and a monitoring vision. The former describes what is monitored (hardware, infrastructure, platform or applications) and the latter describes what this information is intended for following

either the client side or the CSP side vision. According to this taxonomy, Montes identifies three types of monitoring: Client-oriented monitoring on the client side and virtual and physical system monitoring on the CSP side.

Based on this taxonomy, Montes presents an architecture and implementation (GMonE) for a general-purpose monitoring tool, which covers all the aspects mentioned above. It is based on the Publish-Subscribe paradigm and consists of four parts:

1. There is the element to be monitored (GMonEMon), which could be a physical or a virtual resource. GMonEMon ingests monitoring data using plugins, applies an aggregation function and publishes the results to a monitoring manager on a regular basis.
2. The plugins used by GMonEMon to ingest data implement a simple interface and therefore make this approach very flexible and extendable.
3. The monitoring manager (GMonEDB) subscribes to monitoring data and stores it into a database. It also manages the relevant data and provides access to the stored data depending on the users needs.
4. Access to GMonE is provided via a data provider (GMonEAccess).

GMonE and the approach presented in this work both use a plugin mechanism for data ingestion, processing and output. In contrast to this approach, GMonE describes the whole monitoring tooling and does not focus on the Monitoring Agent. Montes focuses on covering the three types of monitoring identified and uses Java and Remote Method Invocation (RMI) for the implementation.

4.3 Role-based Templates for Cloud Monitoring

Nguyen et al. [NSHS14] describe a model based approach for monitoring: Monitoring Templates are used to model the requirements for monitoring depending on roles. These templates are stored in a central database to enable reuse. Each template consists of three components: First of all there is the *cloud endpoint* which describes where monitoring should be executed and how to connect to this endpoint (e.g. by providing Secure Shell (SSH) credentials). Secondly *monitoring variables* describe which metrics should be measured at this endpoint. And finally *monitoring policies* define how this monitoring should be done.

These Monitoring Templates are used to configure three types of agents: In the beginning, an *Activation Agent* parses the cloud endpoint and moves itself to this destination. If all requirements for monitoring are met there, it installs and starts the *Sensor Agent* on the endpoint. The Sensor Agent executes the monitoring by following the monitoring policies defined in the template and sends its results to a *Collector Agent*. The Collector Agent receives the measurements from the Sensor Agent and in return sends commands back depending on the incoming data.

Nguyen's monitoring approach uses CEP engines for both Sensor and Collector agents. On the sensor level these CEP engines are used to filter monitoring data and reduce the amount of data sent to the collector. At the collector level CEP is used to correlate data from different sensors.

Nguyen's and this approach both use agents to retrieve monitoring data on the target machine. Both approaches use modeling and define the behavior of agents in a template which is stored in a central database to enable reuse. They both try to reduce the amount of data sent over the network by providing ways to aggregate data and do alerting directly on the agent. Whereas the agents in Nguyen's approach can move itself to another cloud endpoint, the agents in this approach are assumed to be already installed on the target system e.g. by using a deployment system. Furthermore Nguyen's approach focuses on the different roles in cloud monitoring and includes high-level templates instead of an explicit and precise model to define the internal data flow of the Monitoring Agent including multiple filter, preprocessing and aggregation steps.

In summary the existing models for agent-based monitoring presented above are either very abstract and focus on being comprehensive according to specific taxonomies or they use language-specific features to monitor applications written for a specific platform. None of them provides means to model the internal data flow of Monitoring Agents and thereby enables a precise definition of preprocessing and aggregation steps to be executed on the agent side.

5 Monitoring Agent Requirements

In order to address the issues identified in existing monitoring solutions, this chapter defines the most important requirements which agents in an agent-centric monitoring system should fulfill. Based on the pros and cons of data locality for processing monitoring data discussed in Section 5.1 this chapter describes concrete requirements, divided into Functional Requirements (FRs) in Section 5.2 and Non-functional Requirements (NRs) in Section 5.3.

5.1 Data Locality

Many benefits and drawbacks of Edge Computing mentioned in Section 2.2 also apply to processing of monitoring data: Huge amounts of data are produced at the edge of the network and have to be processed.

There are many advantages of executing filtering, preprocessing, aggregation and alert checks directly on the Monitoring Agent: First of all it is more efficient: The amount of data to be sent via network is reduced by filtering and aggregation. Secondly sending less data over the wire also saves bandwidth which therefore improves the scalability. Thirdly the response time for alerts is much lower if they can be detected locally because there is no delay due to data transport and processing of another server. This enables a fast reactions to data changes and reduces the time to insight of the monitoring system. Finally processing data locally rather than sending raw data via untrusted networks can also improve data privacy [SCZ+16] and reduce the attack surface where confidential information can be leaked or stolen.

Despite these advantages performing more of the activities above directly on the agent also has drawbacks: First it is more complicated to update filtering, preprocessing or aggregation behavior since there is no central entity where the changes can be applied. Second, accessing raw data for an in-depth analysis becomes more complicated if only aggregated data is sent to a central database. This might especially be a problem if lossy aggregations are used and the raw data is not stored on the agent side. If raw data is stored on the agent side, problems concerning state management and synchronization will arise because raw data is scattered across different machines in a distributed system. This might lead to inconsistencies or data being temporary or permanently unavailable due to connection problems or data loss. Finally preprocessing on the agent also increases the monitoring overhead there and has to be taken into account when deciding whether monitoring overhead is acceptable or not.

In conclusion choosing the right place to process the monitoring data considerably depends on the context. In general a monitoring solution should support doing more processing on the target system because of the advantages mentioned above.

5.2 Functional Requirements (FRs)

This section enumerates concrete requirements inferred from the shortcomings of both existing implementations and models identified in the previous chapters. These requirements describe necessary functionalities a Monitoring Agent should implement to follow an agent-centric approach and illustrate them with some examples.

FR 1 Support different simple and composite metrics

As already mentioned in Section 2.1.5, each stakeholder has its own view on a specific system and wants different aspects of a system to be monitored. Therefore Monitoring Agents should be able to ingest data from various sources. Both *simple metrics* like CPU utilization and *composite metrics* computed from multiple simple ones should be supported. Furthermore Monitoring Agents should be able to parse and process textual data in order to be comprehensive: In our running example there are many Microservices which write into log files. For web services these logs might include HTTP status codes which should be parsed and counted over time.

FR 2 Configurable time intervals between measurements

When doing monitoring, there is always a trade-off between the accuracy of measurements and the overhead produced by executing them on the target systems and networks. Because the granularity needed may vary depending on what is monitored, time intervals between measurements should be configurable at a per-metric basis. In our running example the interval between collecting memory usage might be bigger then the one between measuring CPU utilization since memory usage is not expected to change as fast as the utilization of the CPU does. Furthermore the required accuracy of measurements may also change over time. Therefore there should also be a way to adjust time intervals at runtime to support adaptivity of the monitoring system. For example immediately after a new version of a service is deployed, the service developer might be interested of its impact concerning resource consumption and therefore needs short intervals for memory measurements.

FR 3 Locally detect user-specified situations

There should be a simple way for users to describe situations in which they want to be notified or which require automatic actions to be executed. These situations should be described using a syntax which is both easy to understand and expressive. For example it should provide a way to define both static and relative thresholds, ranges for specific metrics as well as defining number of incidents in a time frame. Monitoring Agents should frequently check the collected data in order to detect the incidence of user-specified situations locally. This local detection avoids network traffic and supports timeliness of the monitoring system. If automatic actions are executed after the detection this also supports the autonomicity of the Monitoring Agent.

FR 4 Trigger alerts

When Monitoring Agents detect situations where there are major deviations between the measured values and the ones expected they should provide means to trigger alerts. This could be sending out messages via email or push message or triggering automatic actions like executing a shell script. When an event is detected the alert should be send as soon as possible to support timeliness of the Monitoring Agent.

FR 5 Aggregate monitoring data

Monitoring Agents should be able to aggregate data to save bandwidth. A way to apply either lossy or lossless aggregations on the monitoring data should be provided in order to improve adaptivity and scalability of the Monitoring Agent.

5.3 Non-functional Requirements (NRs)

Aside from the functional requirements mentioned above there are also some non-functional requirements for Monitoring Agents. In contrast to the functional ones the non-functional requirements often can not be pinned to a single agent function and rather affect the behavior of the whole agent.

NR 1 Low Monitoring Overhead

When using agents for monitoring, one or more additional processes are executed on the target system which need memory, processing power and network bandwidth. Therefore there is always a tradeoff between monitoring capabilities, accuracy and timeliness on the one hand and overhead of measuring on the other hand. How much overhead is acceptable highly depends on the context (e.g. amount of hardware resources available), but in general it should be kept low to ensure scalability of the monitoring system.

NR 2 Interoperability

The Monitoring Agent should be able to run in different environments including different architectures and OSs (e.g. Linux and Windows). This also includes to not require extensive modifications on the target system for the Monitoring Agent to run, hence being non-intrusive.

NR 3 Robustness

Monitoring Agent should be able to deal with slow network connections and package loss and should be able to reconnect to the database in case of a connection loss. If the Monitoring Agent crashes it should be restarted automatically. This supports resilience, reliability and availability of the Monitoring Agent.

NR 4 Timeliness

In order to keep the time to insight low, latency for ingesting and processing measurements should be kept short. Alerts should be triggered in time even if monitoring data is aggregated over a long period and therefore the intervals between reports to the database are considerably longer.

NR 5 Extension Mechanism

Agents should be extendable in order to ingest data from new sources and enable the integration of new tools e.g. for AI or CEP. This also supports the comprehensiveness of the Monitoring Agent.

6 Monitoring Templates

In spite of the various capabilities a Monitoring Agent should have according to Chapter 5, configuring these agents should be kept as simple as possible. The configuration should not require any expert knowledge about monitoring solutions nor involve installing special tooling or editing huge configuration files. In order to archive this, a template model with tool support is introduced in this chapter, which allows the configuration of Monitoring Agents in a graphical and intuitive way and therefore ensure usability of the overall monitoring system. It is built on top of the plugin mechanism of Telegraf presented in Section 3.2: Each node represents the instantiation of a Telegraf plugin.

6.1 Overview

A Monitoring Template describes the behavior of a Monitoring Agent including data ingestion, processing, aggregation and output. This template is modeled as a Directed acyclic graph (DAG) with four different kinds of nodes, one for each of these activities (Input, Processor, Aggregator and Output Nodes). The edges between the nodes represent the data flow and describe Monitoring Pipelines.

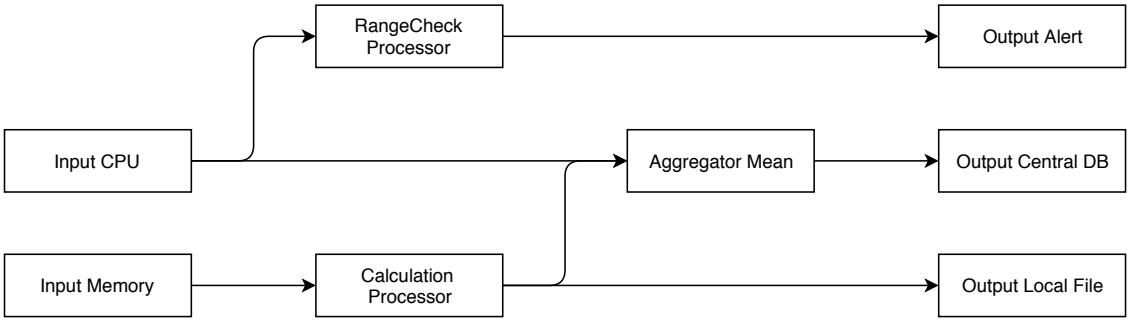


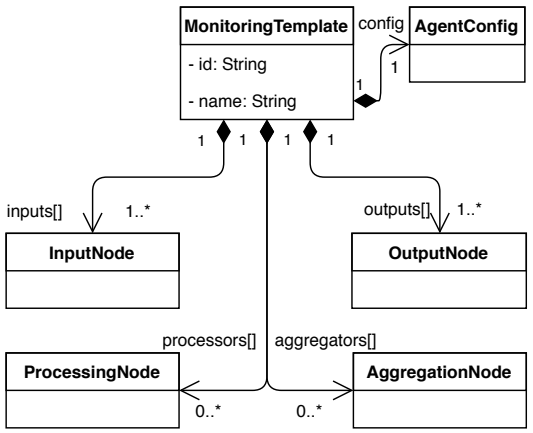
Figure 6.1: DAG showing Monitoring Pipelines for CPU and memory measurements

Figure 6.1 shows a high-level overview with multiple Monitoring Pipelines for CPU and memory measurements: There can be multiple Input Nodes which ingest data from various sources, processing steps including Processor and Aggregator Nodes and finally Output Nodes for storing measurements and sending alerts. Measurements can be piped towards multiple successor nodes e.g. CPU utilization is piped to a node which checks whether it is in a specific range and in parallel measurements are aggregated before being persisted in a Database (DB).

The Monitoring Templates are defined in JavaScript Object Notation (JSON) which can be read by humans, is easily extendable and can be processed in different programming languages and tools. JSON was chosen over Extensible Markup Language (XML) because it is less verbose than XML, less strict and easier to extend and there is native support for JSON when implementing web applications in JavaScript. The schema for these template is defined in JSON-Schema which facilitates validation of templates and enables tool support like auto completion in text editors or automatic generation of forms.

6.2 Template Definition

Figure 6.2a shows a class diagram of a Monitoring Template definition: Every template contains a unique identifier (`id`), a display `name` and lists of node definitions for each kind of node used in the template (`inputs`, `processors`, `aggregators` and `outputs`). It may also include a map with global configurations which should be passed to the agent (`config`). These configuration could be used to implement a lossless aggregation (FR 5) by increasing the `flush_interval` of the Monitoring Agent. Every template has to include at least one Input and Output Node. Additionally it may contain any number of Processor and Aggregator Nodes. Figure 6.2b shows an abbreviated example of a template definition.




```
1  {
2    "id": "template1",
3    "name": "Monitoring Template",
4    "inputs": [ /* ... */],
5    "processors": [ /* ... */],
6    "aggregators": [ /* ... */],
7    "outputs": [ /* ... */],
8    "config": {}
9  }
```

(a) UML class diagram showing Monitoring Template **(b)** Abbreviated example for template definition

Figure 6.2: Monitoring Template

The monitoring data flow is described by the sequence of edges between the nodes. These edges are defined implicitly by sets of next references inside each node: Since the monitoring data only flows one way, a singly linked list is sufficient as the nodes do not need to know of its predecessors.

6.3 Node Definitions

For each kind of node there are several subtypes, e.g. there are different types of Input Nodes, one for each kind of input source. Figure 6.3 shows a brief overview of the hierarchy of template nodes without the intend of being exhaustive.

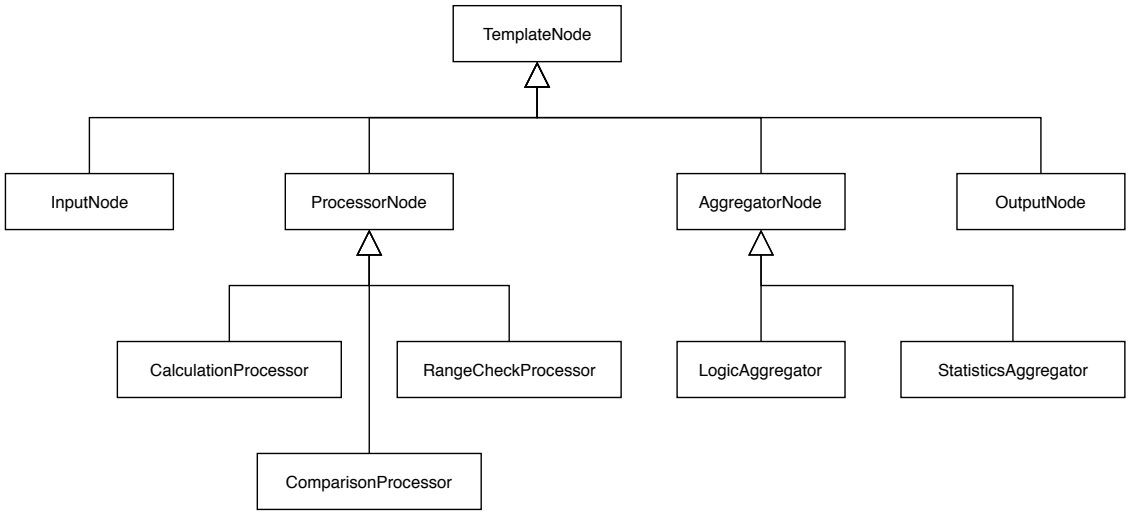


Figure 6.3: UML class diagram showing hierarchy of template nodes

Each node definition contains a unique identifier (`id`), the `type` of node, its configuration as a map (`config`) and a list of its successor nodes (`next`). JSON does not support the concept of pointers, therefore node identifiers are used to reference nodes when linking them. This enables reuse of nodes and sub-graphs within a template. Depending on the kind of node, successor links are restricted, e.g. after a Processor Node there could be another Processor Node, an Aggregator Node or an Output Node, but no Input Node. Output Nodes do not include any references to successor nodes.

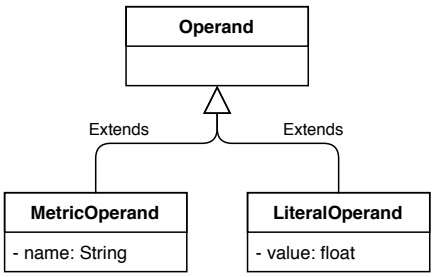


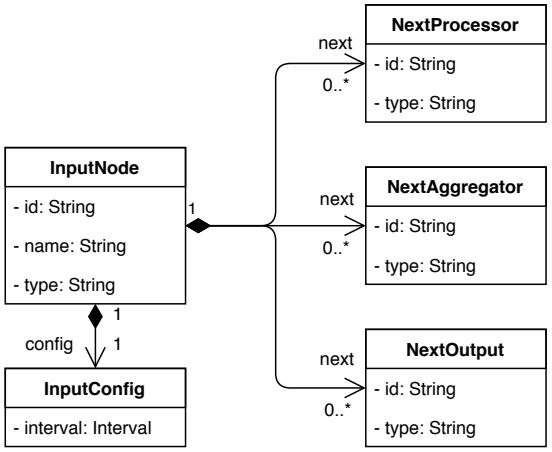
Figure 6.4: UML class diagram showing different operands

Some of these template nodes include parameters in their configurations which can either refer to a fixed number or the current value of a metric (Metric Operand) depending on the users needs. In order to enable this, the concept of an operand shown in Figure 6.4 is introduced: Where ever an operand is needed, the user may pass in a Literal Operand with a fixed value or a Metric Operand with the name of a metric which value is evaluated at runtime.

The following sections include more detailed descriptions of the different types of template nodes and their individual configurations in the order they are typically processed in a Monitoring Pipeline. They also reference the functional and non-functional requirements implemented by these nodes.

6.4 Input Nodes

At the beginning of every Monitoring Pipeline there must be an Input Node which collects data to be processed in subsequent nodes. Figure 6.5a shows the general structure of an Input Node: Each Input Node has a unique `id`, a display `name` and a `type`. The `type` of an Input Node refers to an input plugin which is used to ingest data from various sources e.g. utilization of hardware, statistics from databases and message queues etc. This implements FR 1. By using a string property to reference the plugin, new input plugins can be used within a Monitoring Template without changing the schema. This makes it very extendable (NR 5) and could be used e.g. to ingest events detected by AI or a CEP engine in the future.



(a) UML class diagram showing an Input Node

```
{
  "_type": "ICPUInput",
  "ICPUInput": {
    "id": "in1",
    "name": "cpu",
    "type": "cpu",
    "config": {
      "interval": {
        "value": 10,
        "timeUnit": "s"
      },
      "percpu": true,
      "totalcpu": true,
      "collect_cpu_time": false,
      "report_active": false
    },
    "next": [
      {
        "_type": "INextAggregator",
        "INextAggregator": {
          "$type": "ref",
          "value": {
            "type": "aggregator",
            "id": "agg1"
          }
        }
      }
    ]
  }
}
```

(b) Example for Input Node definition

Figure 6.5: Input Node of Monitoring Template

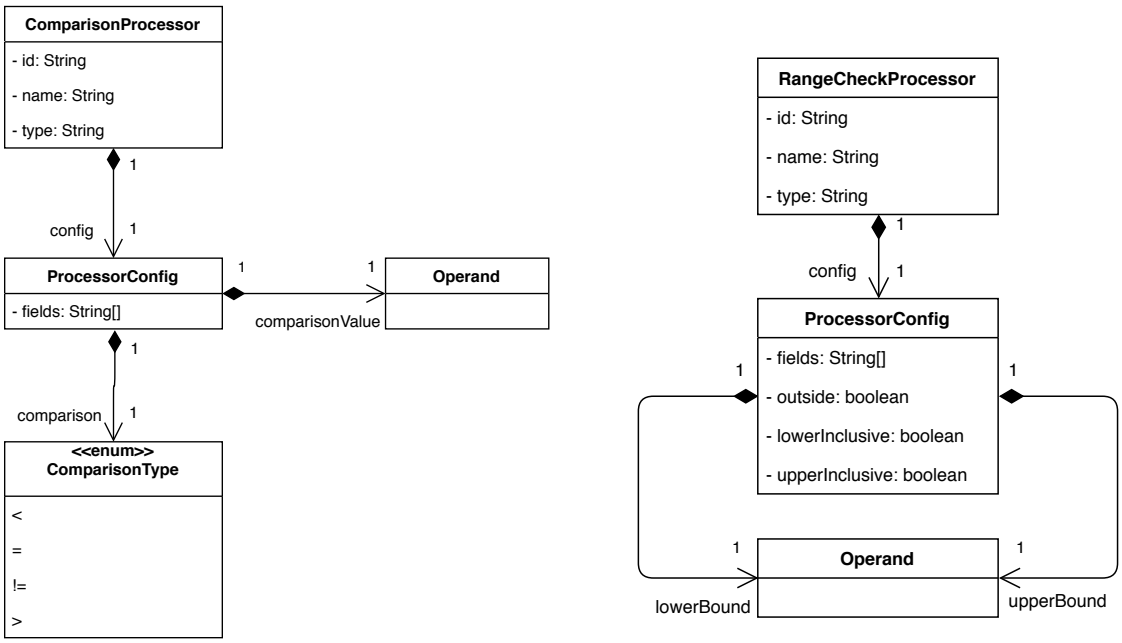
The configuration (`config`) of each Input Node has at least to include an `interval` property which defines the amount of time between each measurement. This provides a way of configuring the granularity on a per-metric basis and therefore enables the implementation of FR 2. Input Node configurations may include further properties which are passed to the referring plugin in order to configure it: For example there might be some flags which describe how CPU utilization should be collected (`percpu`, `totalcpu` etc.) like it is shown in Figure 6.5b.

An Input Node can forward data to multiple successor nodes `next` which can be Processor, Aggregator and Output Nodes. Figure 6.5b shows an example for the definition of an Input Node: This node checks CPU usage every 10 seconds and sends these measurements to the Aggregator Node with the `id` "agg1".

6.5 Processor Nodes

After monitoring data was gathered by Input Nodes, Processor Nodes can be used to either perform checks or calculations which are applied to every single measurement value which passes this node traversing its Monitoring Pipeline.

Figure 6.3 shows the different kinds of Processor Nodes: A Processor Node can be used as a filter by tagging every measurement which fulfills a predefined criteria. Thereby monitoring data which is obviously not relevant can be excluded and the amount of data to be send over the wire can be reduced. This improves the scalability of the monitoring system (NR 1). Filtering of measurements can e.g. be done by comparing metrics against one threshold (Comparison Processor Node) or by checking whether a measurement is within or outside of a specific range (Range Check Processor Node). In combination with special Output Nodes, user-specified situations can both be detected and handled appropriately and therefore implement FR 3.



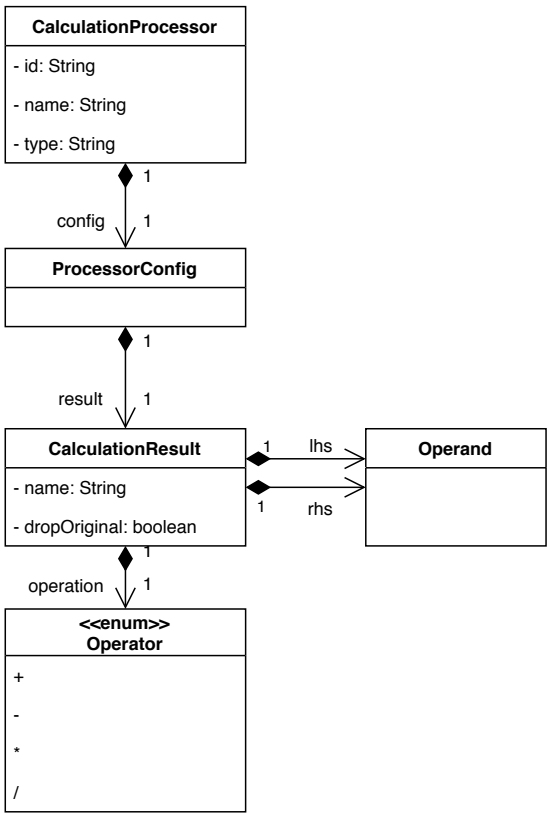
(a) Comparison Processor Node **(b)** Range Check Processor Node

Figure 6.6: UML class diagrams showing internal structure of Processor Nodes which act as filters in the Monitoring Pipeline

The Comparison Processor Node shown in Figure 6.6a performs a check on every single incoming measurement and adds a tag if this check is passed: Its processor configuration includes the comparison type and a threshold value operand which can be static (Literal Operand) or dynamic (Metric Operand) by referring to another metric.

A Range Check Processor Node works similar to the Comparison Processor Node but checks whether a set of `fields` are within or outside a specific range (depending on the `outside` flag of the processor configuration). The range is defined through two operands called `upperBound` and `lowerBound` and whether these bounds are considered as part of the range can be defined using the flags `lowerInclusive` and `upperInclusive`.

Aside from filtering, Processor Nodes can also modify incoming values and calculate new values from them by using a Calculation Processor Node. In this case the Processor Node will act similar to the well known `map()` method which applies an operation on every single value in a collection. The Calculation Processor Node can either be used to calculate composite metrics from simple ones or to convert a simple metric to another scale e.g. temperatures from Celsius to Fahrenheit. This supports the implementation of FR 1, makes the Monitoring Agent more powerful and enables reuse of parts of the Monitoring Template. Figure 6.7a shows the structure of a Calculation Processor Node and its configuration.




```
{
  "_type": "ICalculationProcessor",
  "ICalculationProcessor": {
    "id": "pro2",
    "name": "calculation",
    "type": "calculation",
    "config": {
      "result": {
        "name": "used_ratio",
        "dropOriginal": false,
        "lhs": {
          "_type": "IMetricOperand",
          "IMetricOperand": {
            "name": "used_mean"
          }
        },
        "operation": "/",
        "rhs": {
          "_type": "IMetricOperand",
          "IMetricOperand": {
            "name": "total_mean"
          }
        }
      }
    },
    "next": [
      /* ... */
    ]
  }
}
```

(a) UML class diagram showing internal structure of a Calculation Processor Node

(b) Example for the definition of a Calculation Processor Node inside of a Monitoring Template

Figure 6.7: Calculation Processor Node of a Monitoring Template

The calculation is defined by its left/right-hand side operand (`lhs`/`rhs`) and the mathematical `operation` to be applied. More complex computations can be expressed by nesting multiple of these Calculation Processor Nodes and referring to their particular results. The computed value is named according to the `name` provided and passed on along the Monitoring Pipeline. Original values might be dropped if requested through the referring flag `dropOriginal`.

Figure 6.7b shows an example for the definition of a Processor Node which calculates the percentage of memory used: The amount of memory used (`used_mean`) on the left hand side is divided by the total amount of memory (`total_mean`) on the right hand side. The result of this calculation is then added to the measurement as a field with the name `used_ratio` and passed to the next node.

After a measurement is processed, the Processor Node may pass its results to its successors which could be another Processor Node, an Aggregator Node or an Output Node. By using the Processor Nodes described above, the user has many instruments on hand to filter and process monitoring data directly on the agent.

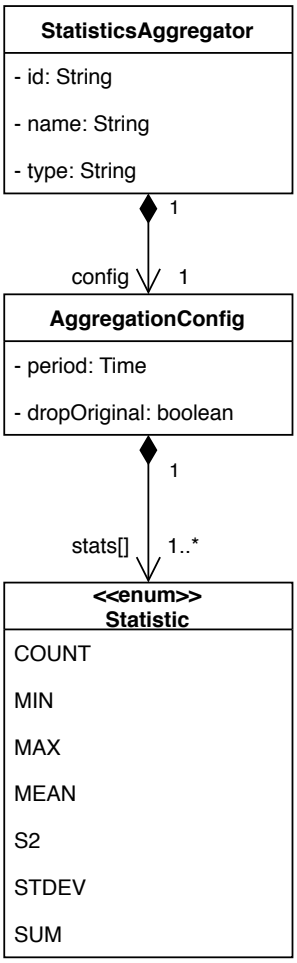
6.6 Aggregator Nodes

Beside the Processor Nodes presented in the previous section, there are also Aggregator Nodes which can be used to calculate statistics (Statistics Aggregator Node) or evaluate logic operations (Logic Aggregator Node) over a specific amount of time. Processor Nodes and Aggregator Nodes are both intermediate nodes located between Input and Output Nodes of a Monitoring Pipeline. But in contrast to Processor Nodes, Aggregator Nodes can not be considered stateless because they cache measurements for a specified `period`.

Additionally the aggregation configuration of a Statistics Aggregator Node includes properties defining a set of statistics (`stats`) which should be computed on every field of a measurement. There is also a flag indicating whether the original values should be removed (`dropOriginal`). In combination these configurations can be used to implement a lossy aggregation (FR 5). Figure 6.8b shows the definition of an Aggregator Node calculating the mean value of input values over a period of 1 minute. This node is implemented by the basic statistics aggregation plugin provided by Telegraf.

The Logic Aggregator Node evaluates logical expressions over a given period of time: It checks whether a set of two or more events occur in the same time frame, where an event is simply a measurement including a metric with the event name.

After measurements are aggregated, they can be send to another Aggregator, a Processor or an Output Node.




```
1  {
2    "_type": "IStatsAggregator",
3    "IStatsAggregator": {
4      "id": "agg1",
5      "name": "stats",
6      "type": "basicstats",
7      "config": {
8        "period": {
9          "value": 1,
10         "timeUnit": "m"
11       },
12       "dropOriginal": true,
13       "stats": [
14         "mean"
15       ]
16     },
17     "next": [
18       {
19         "_type": "INextOutput",
20         "INextOutput": {
21           "$type": "ref",
22           "value": {
23             "type": "output",
24             "id": "out1"
25           }
26         }
27       }
28     ]
29   }
30 }
```

(a) UML class diagram of Statistics Aggregator Node **(b)** Example for Statistics Aggregator Node definition

Figure 6.8: Statistics Aggregator Node of a Monitoring Template

6.7 Output Nodes

Output Nodes refer to Telegraf output plugins and can be used to write metrics to specific sinks. These sinks could be a log file or a database where monitoring data is persisted, but also an alerting framework to send out notifications e.g. via email implementing FR 4. Output Nodes could also be used to integrate other tooling e.g. for AI or CEP. An Output Node marks the end of a Monitoring Pipeline, therefore it has no links to successor nodes. Figure 6.9a shows an example for the definition of an Output Node which sends incoming messages to a local instance of InfluxDB, including the name of the `database`, a set of `urls` and `username`/`password` needed for authentication which are passed in using correspondent environment variables. Figure 6.9b shows an example of a File Output Node which writes incoming measurements into the standard output of the agent using the `influx` format.

```
{
  "_type": "IDatabaseOutput",
  "IDatabaseOutput": {
    "id": "out1",
    "name": "db",
    "type": "influxdb",
    "config": {
      "database": "telegraf",
      "urls": [
        "http://localhost:8086"
      ],
      "username": "$USER",
      "password": "$PW"
    }
  }
}
```

(a) Example for Database Output Node definition

```
{
  "_type": "IFileOutput",
  "IFileOutput": {
    "id": "out2",
    "name": "file",
    "type": "file",
    "config": {
      "file": [
        "stdout"
      ],
      "data_format": "influx"
    }
  }
}
```

(b) Example for File Output Node definition

Figure 6.9: Examples for different Output Node definitions of a Monitoring Template

6.8 Tool Support

At this point a model is defined which describes the behavior of a Monitoring Agent by defining its internal data flow. But there should also be a way to work with these Monitoring Templates without editing verbose text files. Therefore tool support for visualizing and modifying these templates in a graphical way has to be provided. The following sections enumerate and describe concrete Use cases (UCs) which a graphical editor for these monitoring templates should support.

6.8.1 Use Cases (UCs)

The Monitoring Template Editor should provide basic functionality for creating, displaying and updating Monitoring Templates. Figure 6.10 shows an overview of the concrete UCs which are described in more detail in the following sections.

UC 1 Create Monitoring Template

The Monitoring Template Editor should provide a way to create a new Monitoring Template. In this case the application should start with a blank graph without any nodes or edges.

UC 2 Load and display existing Monitoring Template

The Monitoring Template Editor should provide a way to load an existing Monitoring Template from a template file and display its graphical representation.

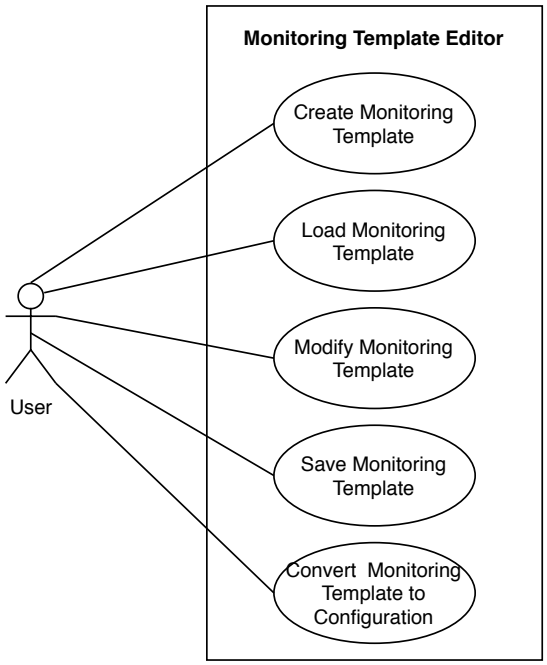


Figure 6.10: UML use case diagram for the Monitoring Template Editor

UC 3 Modify existing Monitoring Template

There should be a way to modify existing Monitoring Templates within the Monitoring Template Editor. This includes adding and removing nodes from the graph as well as connecting nodes with new edges or removing existing ones from the graph. The application should also provide a way to reconfigure individual nodes in the graph and add metadata to the template. The Monitoring Template Editor supports the user by validating its input, offering auto-completion and inserting default values which are reasonable.

UC 4 Save Monitoring Template

There should be a way to save the current state of a Monitoring Template opened in the Monitoring Template Editor as well as upload it to a central server in order to persist it and make it available for other users.

UC 5 Convert Monitoring Template into configuration for Monitoring Agent

The Monitoring Template Editor should be able to convert an existing Monitoring Template into a valid configuration file for the Monitoring Agent.

7 System Description

This chapter describes the proposed monitoring system and how it is implemented and deployed. It starts with an architectural overview of the system in Figure 7.1 and then subsequently describes each of its parts.

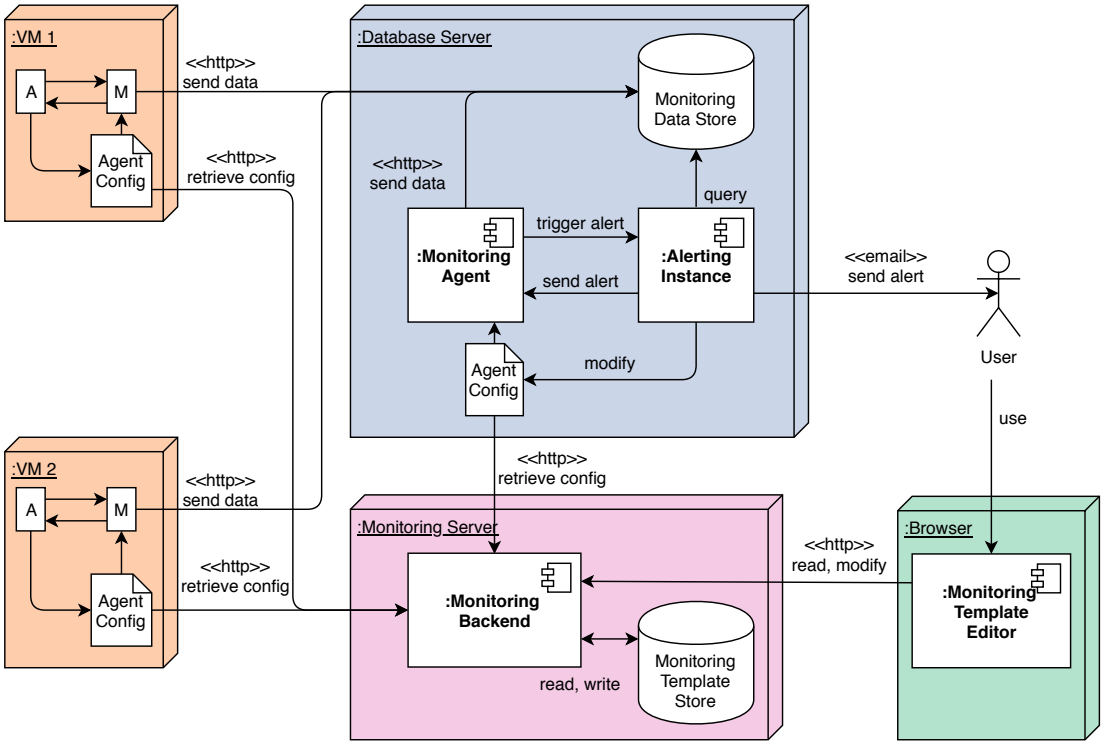


Figure 7.1: Monitoring system architecture overview

7.1 Architecture Overview

Figure 7.1 shows an overview of the architecture of the monitoring system: There are multiple Monitoring Agents (M) which are deployed on different physical or virtual machines (VM1 and VM2) shown on the left of the figure as well as on the central Database Server. These Monitoring Agents fetch their configuration files from the Monitoring Server shown on the bottom of the figure. Together with the referring Monitoring Templates these configuration files are stored in a central Monitoring Template Store and can be accessed through the Monitoring Backend. After successful startup each Monitoring Agent collects data according to its configuration and sends the it to the Database Server on the upper right where it is persisted in a central database (Monitoring Data

Store). Each Monitoring Agent can trigger alerts locally by sending data to the local alerting system instance A. Alerting system instances are deployed beside the Monitoring Agents and might react on specific incoming events: They may send out an alert to a user or the Monitoring Agent or they can adjust the local configuration file of the Monitoring Agent e.g. by increasing or reducing the intervals between specific measurements. The Monitoring Templates can be read and modified by using the Monitoring Template Editor shown on the lower right. The following sections describe each component in more detail.

7.2 Monitoring Agent

Before the Monitoring Agent is started, its configuration file is retrieved from the Monitoring Server. According to this agent configuration the Monitoring Agent then starts to ingests data from different sources, executes filtering and processing steps, aggregates data and counts events over time. Finally it sends out the preprocessed and aggregated data out to a central database or the local alerting system instance.

7.2.1 Implementation

A Telegraf agent is used for handling monitoring data on the target system because of its plugin-driven architecture mentioned in Section 3.2. Telegraf already includes plugins for various data sources and sinks as well as plugins for applying lossy aggregations by computing statistics. Some additional processing and aggregation plugins are implemented to cover all possible template nodes described in Chapter 6. This was done by implementing the interfaces for processor[1] and aggregation[2] plugins.

7.2.2 Deployment

It is assumed that a Monitoring Agent is already deployed on every system which should be monitored. The agent itself does not move itself to the target system, this deployment can be done using any system or tool, in this case Monitoring Agents and their configurations are deployed using an open source deployment tool called CloudConductor[3].

This tool utilizes a deployment agent to ensure that all services and configuration files conform to a provided deployment template. In order to archive this, the deployment agent periodically requests a deployment template from a central deployment server via Hypertext Transfer Protocol (HTTP) and then automatically resolves differences between the retrieved template and its actual state. This is done by using the package manager provided by the OS and by overwriting local configuration files with those retrieved from the deployment server. The deployment agent also ensures that all services mentioned in the deployment template are running on the target system and

[1] https://godoc.org/github.com/influxdata/telegraf#Processor
[2] https://godoc.org/github.com/influxdata/telegraf#Aggregator
[3] http://www.cloudconductor.net

automatically restarts them in case of a failure or if one of its configuration files has been updated on the deployment server. This update and restart functionality is used to support robustness of the monitoring system (NR 3) and to keep the agent configurations up-to-date on the target systems.

7.3 Alerting System

Beside the monitoring agents described in the previous section, Figure 7.1 also shows multiple instances of an alerting system which are co-located with the Monitoring Agents. These instances can be used to trigger alerts either locally or globally:

Local alerting can either be implemented internally or externally: Comparison or Range Check Processor Nodes can be used to filter specific measurements and send out alerts via special Output Nodes. Otherwise, measurements can be sent to another application running on the local system e.g. via message queues. When a local instance of the alerting system receives an event, it is able to perform local actions like executing a shell script with minimal latency: The event information does not have to be transferred over the network and there is no delay due to a remote server which decides whether the current event has to be processed or not. In our running example a local action could be changing the log level of an affected service to simplify debugging or adjusting the monitoring configuration.

Global alerting can be executed by another instance of the alerting system which may be deployed on the database server where it has access to the Monitoring Data Store. By querying this central database the instance of the alerting system is able to correlate data coming from different systems and therefore detect more complex events than a local instance is capable of. But this comes at the cost of transferring the event information to the database server and producing additional load there.

In order to decide whether local or global alerting should be used for a specific event both detection and reaction capabilities of the target systems have to be taken into account. The approach presented in this thesis supports both kinds of alerting by using special input and output plugins of the Telegraf agent.

7.4 Monitoring Data Store

The Monitoring Data Store receives the aggregated monitoring data sent by the Monitoring Agents via HTTP or User Datagram Protocol (UDP) and persist it. This database can be queried by other applications like an alerting system in order to correlate events from different systems or visualize metrics over time in dashboards. InfluxDB is used for the Monitoring Data Store because it is optimized for storing time-series data and like Telegraf part of the open source TICK stack.

7.5 Monitoring Backend

The Monitoring Backend provides access to both Monitoring Templates and configuration files by offering a simple REST interface: Text files can be created, read, updated and deleted by sending HTTP requests. Table 7.1 gives an overview of the REST endpoints provided. The interface is implemented by the Deployment tool mentioned above.

HTTP Verb	Path	Description
GET	/file	Retrieve list of existing text files in JSON
PUT	/file	Create new or update existing file
GET	/file/{name}	Retrieve metadata of file with given name in JSON
DELETE	/file/{name}	Delete text file with given name
GET	/file/{name}/data	Get text content of file with given name
PUT	/file/{name}/data	Update text content of file with given name

Table 7.1: Overview REST interface provided by the Monitoring Backend

7.5.1 Monitoring Template Store

The Monitoring Template Store stores templates for different Monitoring Agents and provides access to the generated configuration files. It can either be deployed on the same machine running the Monitoring Data Store, or on another one which is also accessible for every Monitoring Agent and the Monitoring Template Editor. It is implemented by a relational database.

7.6 Monitoring Template Editor

The Monitoring Template Editor is a web application which can be used to create, modify and manage Monitoring Templates stored in the Monitoring Template Store in a graphical and intuitive way. By providing a Graphical User Interface (GUI) it makes working with agent configurations much easier and therefore supports the usability of the overall Monitoring System.

7.6.1 Implementation

The Monitoring Template Editor is a Single-page application (SPA) implemented in TypeScript using the Angular[4] framework. The forms for configuring individual nodes (UC 3) are generated from the JSON-Schema describing the Monitoring Template using the library ngx-schema-form[5].

[4] https://angular.io
[5] https://github.com/guillotinaweb/ngx-schema-form

Furthermore the editor relies on ngx-graph[6] for rendering the graph with Scalable Vector Graphics (SVG), iarna-toml[7] for converting JavaScript objects to Tom's Obvious, Minimal Language (TOML) and Bootstrap[8] for form components and the responsive layout.

The application completely runs in the browser, template files are retrieved and saved and configuration files are uploaded by using the REST interface of the Monitoring Backend shown in Table 7.1.

7.6.2 User Interface (UI)

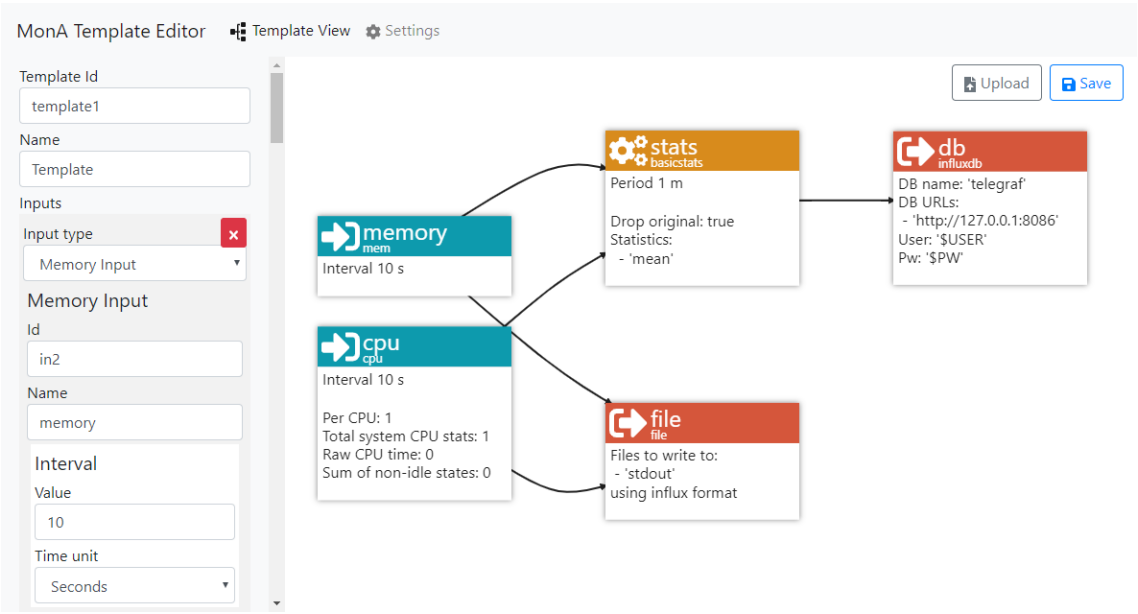


Figure 7.2: Screenshot of the UI provided by the Monitoring Template Editor

Figure 7.2 shows a screenshot of the template view in the Monitoring Template Editor: Below the navigation bar it mainly consists of a sidebar on the left and a graph view on the right. The sidebar includes a sequence of web forms to configure the individual properties of the current template and the contained nodes. The graph view shows an overview of the internal data flow of a Monitoring Agent visualized as a DAG: Each node represents either a data collection (blue), processing (green), aggregation (yellow) or output step (red) and the edges between the nodes indicate where the results of a specific step should be send to.

The Monitoring Template Editor automatically loads the last edited Monitoring Template (UC 2) or creates a new one (UC 1) if there is none available. Nodes are represented as SVG rectangles with title and a detail section. The title section includes the name of a node and its type, indicated by different background colors and a small icon on the left. The detail section below shows a brief summary of all its properties similar to the property section of an UML object or class diagram.

[6] https://github.com/swimlane/ngx-graph
[7] https://github.com/iarna/iarna-toml
[8] https://getbootstrap.com

In order to modify an existing template (UC 3), new nodes can be created using the form. Existing nodes and edges in the graph can be selected by clicking on them and new edges can be created by selecting two nodes in sequence. Selected nodes and edges can be also be removed by pressing the delete key. The graph can be zoomed and panned, but the layout of its nodes is generated and can not be manipulated by the user. In the upper right of the graph view there are buttons to save the current state of the template locally (UC 4) and to upload Monitoring Template and the generated configuration to the Monitoring Backend (UC 5). Connections to the Monitoring Backend can be configured in the settings view of the editor, e.g. access tokens can be added for authentication.

7.6.3 Configuration Mapper

Beside providing the UI presented in the previous section, the Monitoring Template Editor also includes the Configuration Mapper which generates configuration files for the Monitoring Agent and thereby implements UC 5. It traverses each monitoring pipeline in the Monitoring Template and generates a configuration table for each node in the template. In order to ensure that the monitoring data flow matches the one described in the template, each plugin appends a tag with its referring node id to each metric passing through. Additionally, filters are applied before each plugin to make them only accept metrics tagged by predecessor plugins according to the monitoring pipeline. Listing 7.1 shows the resulting sequence of configuration tables in the TOML format[9] used for the Telegraf configuration file. After this configuration file was successfully generated, it is automatically uploaded to the Monitoring Backend.

[9] https://github.com/toml-lang/toml

```
[agent]
interval = "10s"
round_interval = true

[[inputs.cpu]]
interval = "10s"
percpu = true
totalcpu = true
collect_cpu_time = false
report_active = false

  [inputs.cpu.tags]
  in1 = "1"

[[aggregators.basicstats]]
period = "1m"
drop_original = false
stats = [ "mean" ]

  [aggregators.basicstats.tagpass]
  in1 = [ "*" ]

  [aggregators.basicstats.tags]
  agg1 = "1"

[[outputs.influxdb]]
database = "telegraf"
urls = [ "http://127.0.0.1:8086" ]
username = "$User"
password = "$PW"

  [outputs.influxdb.tagpass]
  agg1 = [ "*" ]
```

Listing 7.1: Example for a Telegraf configuration file generated by the Configuration Mapper

8 Evaluation

By implementing all template nodes presented in Chapter 6 the proposed agent-centric monitoring system meets all requirements listed in Chapter 5. In addition to that, this chapter evaluates the proposed system in terms of resource consumption. This is done by comparing CPU utilization, memory usage and network traffic on an observed system while aggregating monitoring data over different periods of time or no aggregation at all.

8.1 Test-bed Description

For the testbed a deployment similar to the one shown in Figure 7.1 is used. It contains two VMs, one running the Monitoring and DB Server aside with a local Monitoring Agent and another VM which only runs a Monitoring Agent. The results presented below are collected from the latter, which will be referred to as "Agent VM" while the other VM will be called "Server VM".

With respect to the technical details, the Agent VM on the one hand runs in an OpenStack environment with 4 Gigabyte (GB) of Random-Access Memory (RAM) and Ubuntu 16.04 as OS. The Server VM on the other hand resides in a Virtuozzo environment with the same amount of RAM and running CentOs 7. Both VMs are connected via a Virtual private network (VPN) using OpenVPN.

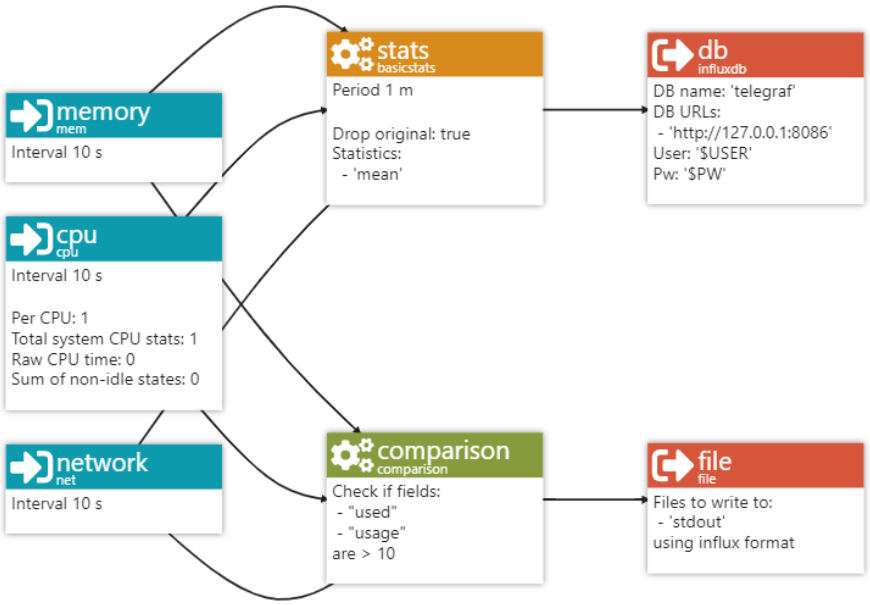


Figure 8.1: Visualization of the Monitoring Template used for the evaluation

8.1.1 Monitoring Template

Figure 8.1 shows the visualization of Monitoring Template which was used for the evaluation, taken from the Monitoring Template Editor: The template includes three Input Nodes which collect CPU utilization, memory usage and network traffic, producing measurements every 10 seconds. The collected data is checked against a fixed threshold using a Comparison Processor Node and aggregated by computing the mean. In the end the aggregated monitoring data is sent to the Server VM and the outliers detected by the Comparison Processor Node are written to the standard output.

8.1.2 Procedure

For evaluating the resource consumption of the Monitoring Agent, first a configuration file representing the Monitoring Template above was deployed on the Agent VM and the Telegraf agent was started. After the collection of ten measurements, CPU utilization, memory usage and the amount of uploaded data was extracted from the InfluxDB on the Server VM. This was done using the export of Comma-separated values (CSV) files provided by the Chronograf UI. After that the aggregation period was increased and the agent was restarted.

8.2 Expected Results

By increasing the aggregation period, the network traffic is expected to decrease because less data has to be send to the database over time. Memory usage is expected to slightly increase because there is more monitoring data which has to be kept in memory before it is send out to the database. CPU utilization is expected to approximately stay the same because the amount of checks on the collected data is not affected by changing the aggregation period.

8.3 Evaluation Results

The results of the measurements are presented in the sections below, grouped by the kind of resource which was measured.

8.3.1 CPU Utilization

Figure 8.2 shows the CPU utilization for both lossless and lossy aggregations and different aggregation periods in percent. In general the CPU utilization measured is below 0.1%. Without aggregation it is almost zero and with aggregation enabled it only slightly increases. As expected, increasing the aggregation period barely affects CPU utilization. Only the variance decreases because the impact of outliers is diminished when aggregating over an increasing period of time.

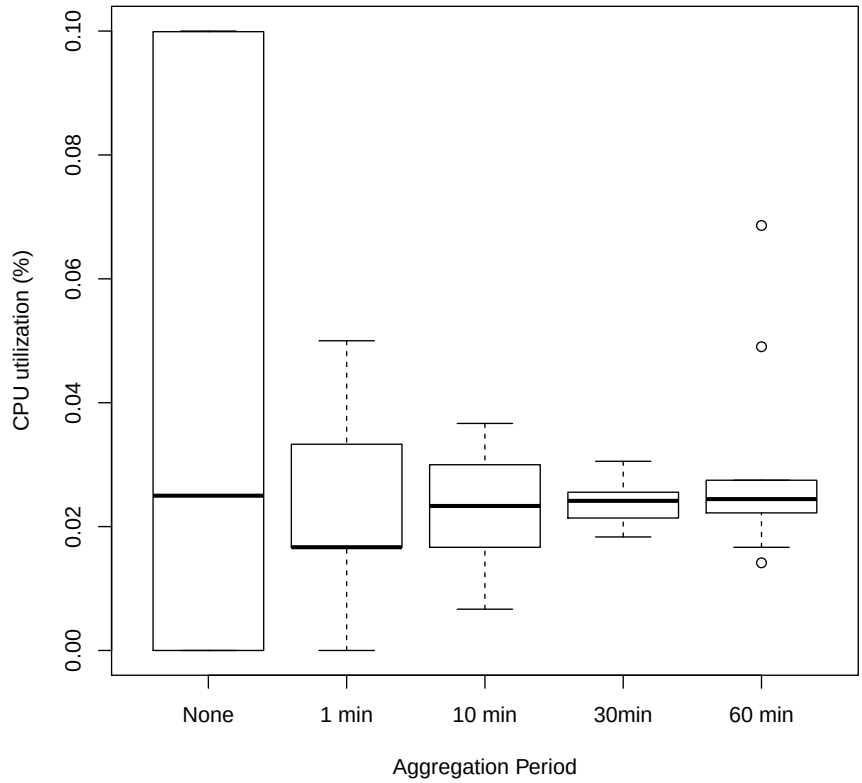


Figure 8.2: CPU utilization on Agent VM for different aggregation periods

8.3.2 Memory Usage

The amount of memory used by the Monitoring Agent stays stable even if the aggregation period is increased beyond 1 hour. For the Agent VM the total amount of used memory was always around 420 MB and approximately 50 MB for the Telegraf agent. System induced fluctuations in memory usage seem to be dominant compared to the impact of the monitoring agent.

8.3.3 Network Traffic

Figure 8.3 shows the amount of data sent from Agent VM to Server VM per minute with aggregation enabled and disabled. When using an input interval of 10 seconds and no aggregation, 35 kB of monitoring data is send over the wire every minute. With lossy aggregation enabled over a period of 1 minute, each report of aggregated data contains only 7 kB which means a reduction of 80%.

Increasing the aggregation period increases the amount of time between the reports sent to the Server VM. Figure 8.4 shows the amount of data uploaded to the Server VM when using lossless (dashed blue) or lossy (solid red) aggregation and aggregation periods of different size. With lossless aggregation, the amount of data sent over time stays stable because reports contain more data. With lossy aggregation, the amount of data sent over time decreases because the size of the report is fixed.

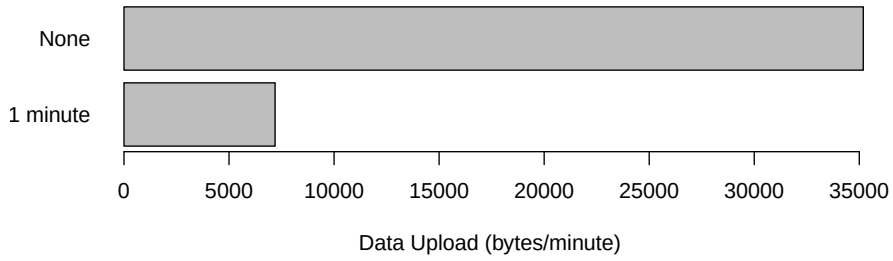


Figure 8.3: Network upload per minute with aggregation enabled and disabled

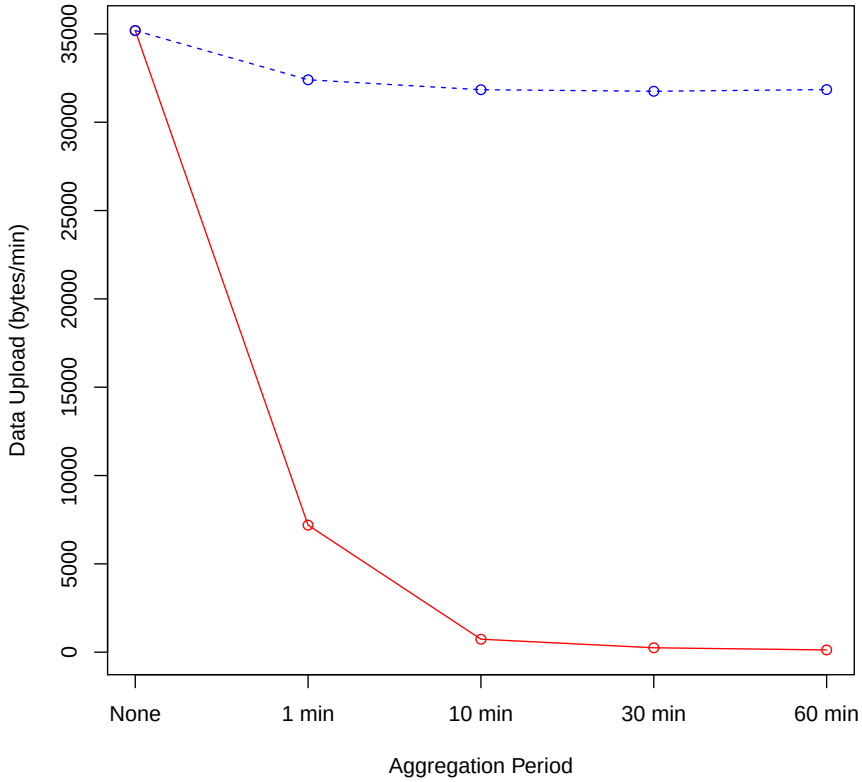


Figure 8.4: Network upload over time for different aggregation periods

8.4 Discussion

The results presented above show that by aggregating monitoring data directly where it is collected, the amount of data sent over the network can be effectively reduced without considerably increasing the computational load or the amount of memory needed on the target systems. In combination with local alerting, monitoring data can be aggregated over a long period of time without loosing the ability to react on changes quickly.

9 Conclusion and Outlook

This thesis presents an agent-centric approach for monitoring which allows the execution of preprocessing and aggregation steps directly on the target systems for cloud or edge-cloud computing scenarios. Thereby the proposed approach addresses various shortcomings identified in both existing monitoring tools and related scientific work concerning scalability, extendability and usability.

Build on top of the plugin mechanism of an existing Monitoring Agent implementation, an extendable template model is introduced: Monitoring Templates provide expressive means to define filter, preprocessing and aggregation steps, detecting user-defined situations and locally handle events or sending alerts. Beside that, a prototypical implementation of a graphical editor is presented which simplifies the definition of the agent behavior by connecting processing steps in a graphical way. This improves the usability of the monitoring solution and provides a way for non-experts to configure Monitoring Agents without editing huge configuration files.

Preliminary results show that by following the proposed monitoring approach the amount of monitoring data which has to be send over the network can be effectively reduced without having a major impact on the load of the monitored systems or loosing accuracy when detecting events.

Outlook

In future works the proposed monitoring system may be extended by integrating other systems in order to apply AI or CEP by using special Input and Output Nodes which send or receive measurements e.g. via messaging queues. Another future topic could be a smart combination of local and global alerting e.g. by automatically inferring whether specific events can be detected and handled locally.

Beside that the approach should be further evaluated in a more complex distributed deployments to analyze its scalability. For these kinds of scenarios, the Monitoring Template Editor could also be enhanced by adding a dashboard view which summarizes the state of all monitored systems including the template they follow and the last monitoring data which was reported by them. This could facilitate management when using many different Monitoring Templates.

Bibliography

[ABDP12] G. Aceto, A. Botta, W. de Donato, A. Pescapè. "Cloud monitoring: Definitions, issues and future directions". In: *1st IEEE International Conference on Cloud Networking, CLOUDNET 2012, Paris, France, November 28-30, 2012*. Ed. by G. Pujolle, R. Boutaba, M. Brunner, S. Secci. IEEE, 2012, pp. 63–67. DOI: 10.1109/CloudNet.2012.6483656. URL: https://doi.org/10.1109/CloudNet.2012.6483656 (cit. on pp. 13, 16, 18).

[ABDP13] G. Aceto, A. Botta, W. de Donato, A. Pescapè. "Cloud monitoring: A survey". In: *Computer Networks* 57.9 (2013), pp. 2093–2115. DOI: 10.1016/j.comnet.2013.04.001. URL: https://doi.org/10.1016/j.comnet.2013.04.001 (cit. on pp. 13, 16).

[ARM+15] K. Alhamazani, R. Ranjan, K. Mitra, F. A. Rabhi, P. P. Jayaraman, S. U. Khan, A. Guabtni, V. Bhatnagar. "An overview of the commercial cloud monitoring tools: research dimensions, design issues, and state-of-the-art". In: *Computing* 97.4 (2015), pp. 357–377. DOI: 10.1007/s00607-014-0398-5. URL: https://doi.org/10.1007/s00607-014-0398-5 (cit. on p. 18).

[CG15] J. M. A. Calero, J. Gutierrez-Aguado. "Comparative analysis of architectures for monitoring cloud computing infrastructures". In: *Future Generation Comp. Syst.* 47 (2015), pp. 16–30. DOI: 10.1016/j.future.2014.12.008. URL: https://doi.org/10.1016/j.future.2014.12.008 (cit. on p. 19).

[CLC14] S. Cho, H. Li, B. J. Choi. "PALDA: Efficient privacy-preserving authentication for lossless data aggregation in Smart Grids". In: *2014 IEEE International Conference on Smart Grid Communications, SmartGridComm 2014, Venice, Italy, November 3-6, 2014*. IEEE, 2014, pp. 914–919. DOI: 10.1109/SmartGridComm.2014.7007765. URL: https://doi.org/10.1109/SmartGridComm.2014.7007765 (cit. on p. 20).

[FRWZ07] E. Fasolo, M. Rossi, J. Widmer, M. Zorzi. "In-network aggregation techniques for wireless sensor networks: a survey". In: *IEEE Wireless Commun.* 14.2 (2007), pp. 70–87. DOI: 10.1109/MWC.2007.358967. URL: https://doi.org/10.1109/MWC.2007.358967 (cit. on p. 20).

[MG+11] P. Mell, T. Grance, et al. "The NIST definition of cloud computing". In: (2011). URL: https://csrc.nist.gov/publications/detail/sp/800-145/final (cit. on pp. 15, 16).

[MSM+13] J. Montes, A. Sánchez, B. Memishi, M. S. Pérez, G. Antoniu. "GMonE: A complete approach to cloud monitoring". In: *Future Generation Comp. Syst.* 29.8 (2013), pp. 2026–2040. DOI: 10.1016/j.future.2013.02.011. URL: https://doi.org/10.1016/j.future.2013.02.011 (cit. on p. 25).

[NSHS14] T. A. B. Nguyen, M. Siebenhaar, R. Hans, R. Steinmetz. “Role-Based Templates for Cloud Monitoring”. In: *Proceedings of the 7th IEEE/ACM International Conference on Utility and Cloud Computing, UCC 2014, London, United Kingdom, December 8-11, 2014*. IEEE Computer Society, 2014, pp. 242–250. DOI: 10.1109/UCC.2014.33. URL: https://doi.org/10.1109/UCC.2014.33 (cit. on pp. 17, 20, 26).

[PLL+13] J. Povedano-Molina, J. M. Lopez-Vega, J. M. López-Soler, A. Corradi, L. Foschini. “DARGOS: A highly adaptable and scalable monitoring architecture for multi-tenant Clouds”. In: *Future Generation Comp. Syst.* 29.8 (2013), pp. 2041–2056. DOI: 10.1016/j.future.2013.04.022. URL: https://doi.org/10.1016/j.future.2013.04.022 (cit. on p. 18).

[SCZ+16] W. Shi, J. Cao, Q. Zhang, Y. Li, L. Xu. “Edge Computing: Vision and Challenges”. In: *IEEE Internet of Things Journal* 3.5 (2016), pp. 637–646. DOI: 10.1109/JIOT.2016.2579198. URL: https://doi.org/10.1109/JIOT.2016.2579198 (cit. on pp. 17, 29).

[SPLCD] *Components of a Splunk Enterprise deployment - Splunk Documentation*. URL: http://docs.splunk.com/Documentation/Splunk/7.1.3/Capacity/ComponentsofaSplunkEnterprisedeployment (cit. on p. 22).

[SPLE] *About Splunk Enterprise - Splunk Documentation*. URL: http://docs.splunk.com/Documentation/Splunk/7.1.2/Overview/AboutSplunkEnterprise (cit. on p. 21).

[SPLFR] *About forwarding and receiving - Splunk Documentation*. URL: http://docs.splunk.com/Documentation/Splunk/7.1.2/Forwarding/Aboutforwardingandreceivingdata (cit. on p. 21).

[SPLFT] *Types of forwarders - Splunk Documentation*. URL: http://docs.splunk.com/Documentation/Splunk/7.1.2/Forwarding/Typesofforwarders (cit. on p. 21).

[SPLUF] *The universal forwarder - Splunk Documentation*. URL: http://docs.splunk.com/Documentation/Forwarder/7.1.2/Forwarder/Abouttheuniversalforwarder (cit. on p. 21).

[Spr11a] J. Spring. “Monitoring Cloud Computing by Layer, Part 1”. In: *IEEE Security & Privacy* 9.2 (2011), pp. 66–68. DOI: 10.1109/MSP.2011.33. URL: https://doi.org/10.1109/MSP.2011.33 (cit. on p. 15).

[Spr11b] J. Spring. “Monitoring Cloud Computing by Layer, Part 2”. In: *IEEE Security & Privacy* 9.3 (2011), pp. 52–55. DOI: 10.1109/MSP.2011.57. URL: https://doi.org/10.1109/MSP.2011.57 (cit. on p. 15).

[SWWM10] J. Shao, H. Wei, Q. Wang, H. Mei. “A Runtime Model Based Monitoring Approach for Cloud”. In: *IEEE International Conference on Cloud Computing, CLOUD 2010, Miami, FL, USA, 5-10 July, 2010*. IEEE Computer Society, 2010, pp. 313–320. DOI: 10.1109/CLOUD.2010.31. URL: https://doi.org/10.1109/CLOUD.2010.31 (cit. on pp. 17, 25).

[TG] *GitHub - influxdata/telegraf: The plugin-driven server agent for collecting and reporting metrics.* URL: https://github.com/influxdata/telegraf (cit. on p. 22).

[TGD] *telegraf - GoDoc*. URL: https://godoc.org/github.com/influxdata/telegraf (cit. on p. 22).

[TICK] *TICK Stack Built Within Open Source Platform | InfluxData*. URL: https://www.influxdata.com/time-series-platform/ (cit. on pp. 22, 23).

All links were last followed on November 7, 2018.

Declaration

I hereby declare that the work presented in this thesis is entirely my own and that I did not use any other sources and references than the listed ones. I have marked all direct or indirect statements from other sources contained therein as quotations. Neither this work nor significant parts of it were part of another examination procedure. I have not published this work in whole or in part before. The electronic copy is consistent with all submitted copies.

---

place, date, signature